CS331
Haskell Tutorial 02

A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- Isomorphic
- Currying, Composition
- Lazy Evaluation

Example of Functions

- Double a given input.

```
square :: Int -> Int
Prelude>square x = x*x
Prelude>square 5
```

- Conversion from fahrenheit to celcius

```
fahr_to_celcius :: Float -> Float
Prelude> fahr_to_celcius temp = (temp - 32)/1.8
Prelude> :t fahr_to_celcius
```

- A function with multiple results - quotient & remainder

```
divide ::  Int -> Int -> (Int,Int)

divide x y = (div x y, mod x y)
```

Expression-Oriented

- Instead of imperative commands/statements, the focus is on expression.

- Instead of *command/statement* :

```
if e1 then stmt1 else stmt2
```

- We use conditional *expressions* :

```
if e1 then e2 else e3
```

Expression-Oriented

- An example function:

```
fact     :: Integer -> Integer
fact n   = if n=0 then 1
             else n * fact (n-1)
```

- Can use pattern-matching instead of conditional

```
fact 0            = 1
fact n            = n * fact (n-1)
```

- Alternatively:

```
fact n            = case n of
                 0 -> 1
                 a -> a * fact (a-1)
```

Conditional → Case Construct

- Conditional;

```
if e1 then e2 else e3
```

- Can be translated to

```
case e1 of
  True -> e2
  False -> e3
```

- Case also works over data structures (without any extra primitives)

```
length xs = case xs of
        [] -> 0;
          y:ys -> 1+(length ys)
```

Locally bound variables

Lexical Scoping

- Local variables can be created by let construct to give nested scope for the name space. Example:

```
let     y   = a+b
        f x = (x+y)/y
in f c + f d
```

- For scope bindings over guarded expressions, we require a where construct instead:

```
f x y       | x>z= …
    | y==z     = …
    | y<z= ….
  where z=x*x
```

Layout Rule

- Haskell uses two dimensional syntax that relies on declarations being "lined-up columnwise"

```
let  y    = a+b
     f x  = (x+y)/y
in f c + f d
```

is being parsed as:

```
let  { y   = a+b
     ; f x = (x+y)/y }
in f c + f d
```

- Rule : Next character after keywords `where`/`let`/`of`/`do` determines the starting columns for declarations. Starting *after* this point continues a declaration, while starting *before* this point terminates a declaration.

Expression Evaluation

- Expression can be computed (or evaluated) so that it is reduced to a value. This can be represented as:

  $e \rightarrow .. \rightarrow v$

- We can abbreviate above as:

  $e \rightarrow^{*} v$

- A concrete example of this is:

  inc (inc 3) → inc (4) → 5

- Type preservation theorem says that:

  if $e :: t$ Æ $e \rightarrow v$, it follows that $v :: t$

Æ : Almost everywhere

Values and Types

- As a purely functional language, all computations are done via evaluating *expressions* (**syntactic sugar**) to yield *values* (normal forms as answers).
- Each expression has a *type* which denotes the set of possible outcomes.
- v :: t can be read as value v has type t.
- Examples of *typings*, associating a value with its corresponding type are:

```
5          :: Integer
'a'        :: Char
[1,2,3]    :: [Integer]
('b', 4)      :: (Char, Integer)
“cs5”      :: String (same as [Char])
```

Syntactic sugar

- Syntactic sugar is usually a shorthand for a common operation that could also be expressed in an alternate, more verbose, form
- Example: List Comprehension in python, Operator overloading, unary operator (++, += in C++)
- Benefit
  - Conciseness: make code more concise
  - Fewer error
  - Readability: Easier to read
  - Maintainability
  - Abstraction: Complex operation to simple syntax

Built-In Types

- They are not special:

```
data Char      =   'a' | 'b' | …
data Int       =  -65532 | … | -1 | 0 | 1 | …. | 65532
data  Integer       = … | -2 | -1 | 0 | 1 | 2 | …
```

- Tuples are also built-in.

```
data (a,b)     = M2(a,b)
data (a,b,c)   = M3(a,b,c)
data (a,b.c.d)= M4(a,b,c,d)
```

- List type uses an infix operator:

```
data [a]          = []   |   a : [a]
```

`[1,2,3]` is short hand for `1:(2:(3:[]))`

User-Defined Algebraic Types

- Can describe enumerations:

```
data Bool    = False | True
data Color   = Red | Green | Blue | Violet
```

- Can also describe a tuple

```
data Pair    = P2 Integer Integer
data Point a       = Pt a a
```

type variable

- `Pt` is a data constructor with type `a -> a -> Point a`

```
Pt 2.0 3.1   :: Point Float
Pt 'a' 'b'   :: Point Char
Pt True False   :: Point Bool
```

Recursive Types

- Some types may be recursive:

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

- Two data constructors:

```
Leaf   :: a -> Tree a
Branch       :: Tree a -> Tree a -> Tree a
```

- An example function over recursive types:

```
fringe :: Tree a -> [a]

fringe (Leaf x)             = [x]
fringe (Branch left right) = (fringe left) ++
                                 (fringe right)
```

++ is concatenation of two lists

Polymorphic Types

- Support types that are universally quantified in some way over all types.
- **8 c. [c] denotes a family of types, for every type c, the type of lists of c.**
- Covers `[Integer], [Char], [Integer->Integer],` etc.
- Polymorphism help support *reusable* code, e.g

```
length          :: 8 a. [a] -> Integer
length []       = 0
length (x:xs)   = 1 + length xs

Prelude> :t length
```

Polymorphic Types

- This polymorphic function can be used on list of any type..

```
length [1,2,3]              )     3
length [‘a’, ’b’, ‘c’]      )     3
length [[1],[],[3]]         )     3
```

- More examples :

```
head          :: [a] -> a
head (x:xs)      = x

tail          :: [a] -> [a]
tail (x:xs)      = xs
```

- Note that head/tail are partial functions, while length is a total function?

Polymorphic Types

- Example

```
ghci>:{
ghci| length []       = 0
ghci| length (x:xs) = 1 + length xs
ghci| :}
ghci> :t length
ghci> length [1,2,3]
3
```

Polymorphic Types

- This polymorphic function can be used on list of any type..

```
length [1,2,3]              )    3
length [‘a’, ’b’, ‘c’]      )    3
length [[1],[],[3]]         )    3
```

- More examples :

```
head        :: [a] -> a
head (x:xs)    = x

tail        :: [a] -> [a]
tail (x:xs)    = xs
```

- Note that head/tail are partial functions, while length is a total function?

Principal Types

- Some types are more general than others:

  `[Char]   <:   8 a. [a]   <:  8 a. a`

- An expression's *principal type* is the *least general type* that contains all instances of the expression.

- For example, the *principal type* of `head` function is `[a]->a`, while `[b] -> a`, `b -> a`, `a` are correct but too general but `[Integer] -> Integer` is too specific.

- Principal type can help supports software reusability with accurate type information.

Functions and its Type

- Method to increment its input

```
inc x=  x+1
```

- Or through lambda expression (anonymous functions)

```
(\ x -> x+1)
```

- They can also be given suitable function typing:

```
inc          :: Num a => a -> a
(\x -> x+1)    :: Num a => a -> a
```

- Types can be *user-supplied* or *inferred*.

Anonymous Functions

- Anonymous functions are used often in Haskell, usually enclosed in parentheses
- `\x y -> (x + y) / 2`
  - the `\` is pronounced “lambda”
    - It’s just a convenient way to type λ
  - the `x` and `y` are the formal parameters
- Functions are first-class objects and can be assigned
  - `avg = \x y -> (x + y) / 2`

Functions and its Type

- Some examples

```
(\x -> x+1) 3.2  →

(\x -> x+1) 3  →

Prelude> (\x -> x+1) 3
```

- User can restrict the type, e.g.

```
inc    :: Int -> Int
```

- In that case, some examples may be wrongly typed.

```
inc 3.2  →

inc 3  →
```

Functions

- Functions can be written in two main ways:

```
add           :: Integer -> Integer -> Integer
add x y       = x+y

add2          :: (Integer,Integer) -> Integer
add2(x,y)     = x+y
```

- The first version allows a function to be returned as result after applying a single argument.

```
inc           :: Integer -> Integer
inc           =  add 1
```

- The second version needs all arguments. Same effect requires a lambda abstraction:

```
inc           = \ x -> add2(x,1)
```

Functions

- Functions can also be passed as parameters. Example:

```
map             :: (a->b) -> [a] -> [b]
map f []        = []
map f (x:xs)    = (f x) : (map f xs)
```

- Such higher-order function aids code reuse.

```
map (add 1) [1, 2, 3]    ) [2, 3, 4]
map add [1, 2, 3]        ) [add 1, add 2, add 3]
```

- Alternative ways of defining functions:

```
add              = \ x -> \ y -> x+y
add              = \ x y -> x+y
```

Expression-Oriented

- An example function:

```
fact      :: Integer -> Integer
fact n  = if n=0 then 1
             else n * fact (n-1)
```

- Can use pattern-matching instead of conditional

```
fact 0          = 1
fact n          = n * fact (n-1)
```

- Alternatively:

```
fact n          = case n of
              0 -> 1
              a -> a * fact (a-1)
```

Conditional → Case Construct

- Conditional;

```
if e1 then e2 else e3
```

- Can be translated to

```
case e1 of
  True -> e2
  False -> e3
```

- Case also works over data structures (without any extra primitives)

```
length xs = case xs of
        [] -> 0;
        y:ys -> 1+(length ys)
```

Locally bound variables

Lexical Scoping

- Local variables can be created by `let` construct to give nested scope for the name space.

  Example:

```
let     y  = a+b
      f x = (x+y)/y
in f c + f d
```

```
Prelude> :{
Prelude> | myf c d =
Prelude> |              let y = 7+3
Prelude> |                  f x = (x+y)/2
Prelude> |              in f c + f d
Prelude> |:}
Prelude> myf 20 30
Prelude> 35.0
```

Layout Rule

- Haskell uses two dimensional syntax that relies on declarations being "lined-up columnwise"

```
let  y     = a+b
     f x   = (x+y)/y
in f c + f d
```

is being parsed as:

```
let  { y   = a+b
     ; f x = (x+y)/y }
in f c + f d
```

- Rule : Next character after keywords `where`/`let`/`of`/`do` determines the starting columns for declarations. Starting *after* this point continues a declaration, while starting *before* this point terminates a declaration.

Lexical Scoping

- For scope bindings over guarded expressions, we require a where construct instead:

```
f x y | x>z     = …
      | y==z    = …
      | y<z     = ….
  where z=x*x
```

Writing multiline function

- Space and indentation is important in writing code
- Use space instead of Tab
- Writing multiline function; Start with :{ and end with :} , spacing and newline is must

```
Prelude> :{
Prelude| fact n = if n==0 then 1
Prelude|                else n * fact (n-1)
Prelude| :}
```

Notation

- We can abbreviate repeated left hand sides

```
absolute x | x >= 0 = x
absolute x | x < 0  = -x
```

```
absolute x | x >= 0 = x
           | x < 0  = -x
```

- Haskell also has **if then else**

absolute x = **if** x >= 0 **then** x **else** -x

```
Prelude> :{
Prelude| absolute x | x>=0 = x
Prelude|            | x<0  = -x
Prelude| :}
Prelude> absolute (-24)
24
```

Loading from HS file

- Loading Haskell script (source code) from file
- Suppose fact.hs contents this : **H**askell **S**cript

```
fact n = if n==0 then 1
         else n * fact (n-1)
```

- Any module it say as Main : from file

```
Prelude> :load fact.hs
[1 of 1] Compiling Main      ( fact.hs, interpreted )
Ok, one module loaded.
*Main> fact 4
24
*Main> :m - Main
Prelude>
```

Functions and its Type

- Method to increment its input

```
inc x=  x+1
```

- Or through lambda expression (anonymous functions)

```
(\ x -> x+1)
```

- They can also be given suitable function typing:

```
inc              :: Num a => a -> a
(\x -> x+1)      :: Num a => a -> a
```

- Types can be *user-supplied* or *inferred*.

Anonymous Functions

- Anonymous functions are used often in Haskell, usually enclosed in parentheses
- `\x y -> (x + y) / 2`
  - the `\` is pronounced "lambda"
    - It's just a convenient way to type λ
  - the `x` and `y` are the formal parameters
- Functions are first-class objects and can be assigned
  - `avg = \x y -> (x + y) / 2`

Functions and its Type

- Some examples

```
(\x -> x+1) 3.2   →

(\x -> x+1) 3   →

Prelude> (\x -> x+1) 3
```

- User can restrict the type, e.g.

```
inc    :: Int -> Int
```

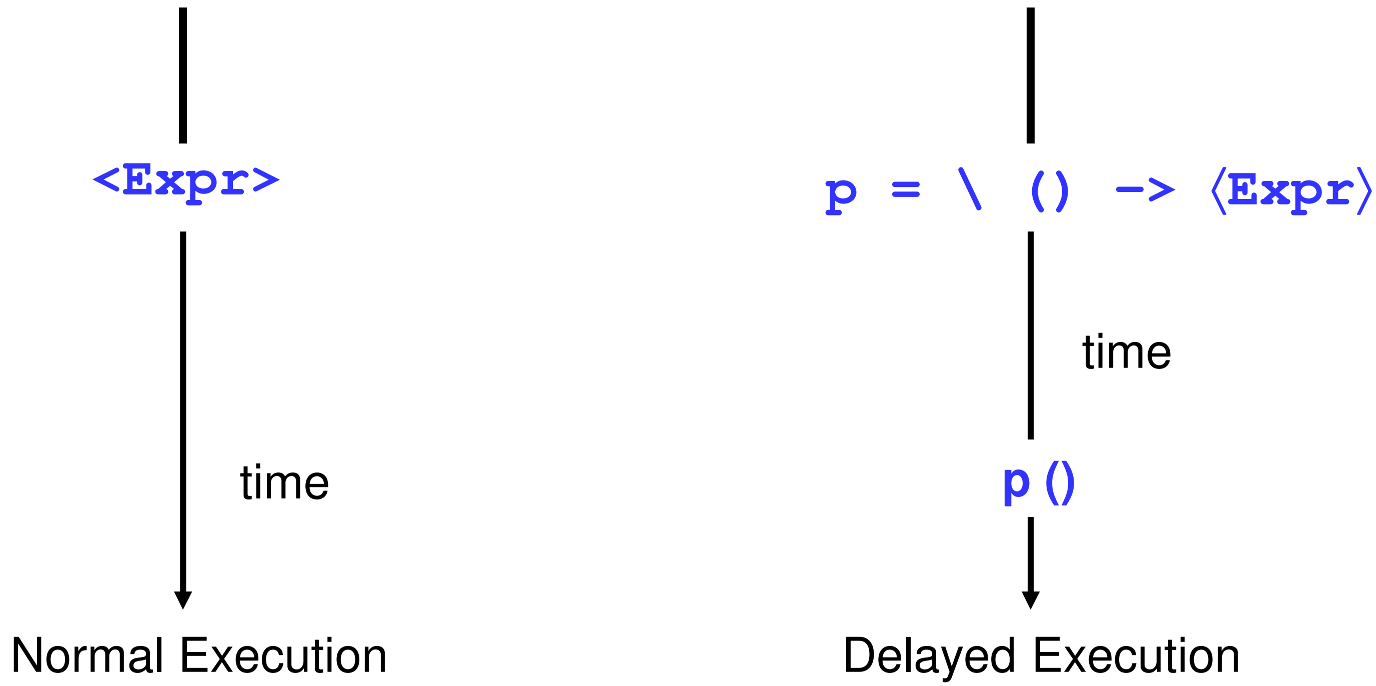
- In that case, some examples may be wrongly typed.

```
inc 3.2   →

inc 3   →
```

Function Abstraction

- Function abstraction is the ability to convert any expression into a function that is evaluated at a later time.



Higher–Order Functions

- **Higher-order programming** treats functions as first-class,
  - Allowing them to be passed as parameters, returned as results or stored into data structures.
- This concept supports generic coding,
  - and allows programming to be carried out at a more abstract level.
- Genericity can be applied to a function
  - by letting specific operation/value in the function body to become parameters.

Higher order Functions

- Functions can be written in two main ways:

```
add x y        = x+y
add2(x,y)      = x+y
```

- The first version allows a function to be returned as result after applying a single argument.

```
inc   =   add 1
```

```
Prelude> add x y = x+y
Prelude> inc = add 1
Prelude > inc 5
6
Prelude>
```

Higher order Functions

- The second version needs all arguments. Same effect requires a lambda abstraction:

```
add2(x,y)     = x+y
inc    = \x -> add2(x,1)
```

```
Prelude> add2 (x+y) = x+y
Prelude> inc =  \x -> add2(x, 1)
Prelude > inc 5
6
Prelude>
```

Functions

- Functions can also be passed as parameters. Example:

```
map            :: (a->b) -> [a] -> [b]
map f []       = []
map f (x:xs)   = (f x) : (map f xs)
```

- Such higher-order function aids code reuse.

```
map (add 1) [1, 2, 3]    ) [2, 3, 4]
map add [1, 2, 3]        ) [add 1, add 2, add 3]
```

- Alternative ways of defining functions:

```
add            = \ x -> \ y -> x+y
add            = \ x y -> x+y
```

Where example : write like math Statement

```
roots (a,b,c) = (x1, x2) where
  x1 = e + sqrt d / (2 * a)
  x2 = e - sqrt d / (2 * a)
  d = b * b - 4 * a * c
  e = - b / (2 * a)
main = do
  putStrLn "The roots of our Polynomial equation are:"
  print (roots(1,-8,6))
```

```
Prelude> :load WhereExample.hs
*Main> main
The roots of our Polynomial equation are:
(7.1622777,0.8377223)
```

Thanks