CS331
Haskell Tutorial 01
Basic of Haskell

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

Outline

- Introduction: Installing, Invoking, Hello world
- Functional programming
  - Basic
- Typed
- Isomorphic
- Currying, Composition
- Lazy Evaluation

Installing Haskell: GHC and Cabal

- Ubuntu
  - sudo add-apt-repository ppa:hvr/ghc
  - sudo apt-get update
  - sudo apt-get install -y cabal-install ghc
- Windows in command mode
  - Start-Process powershell -Verb runAs
  - Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
  - Download https://get.haskellstack.org/stable/windows-x86_64-installer.exe
  - **choco install haskell-dev**
- Issue command: ghci or ghc

GHC and Cabal

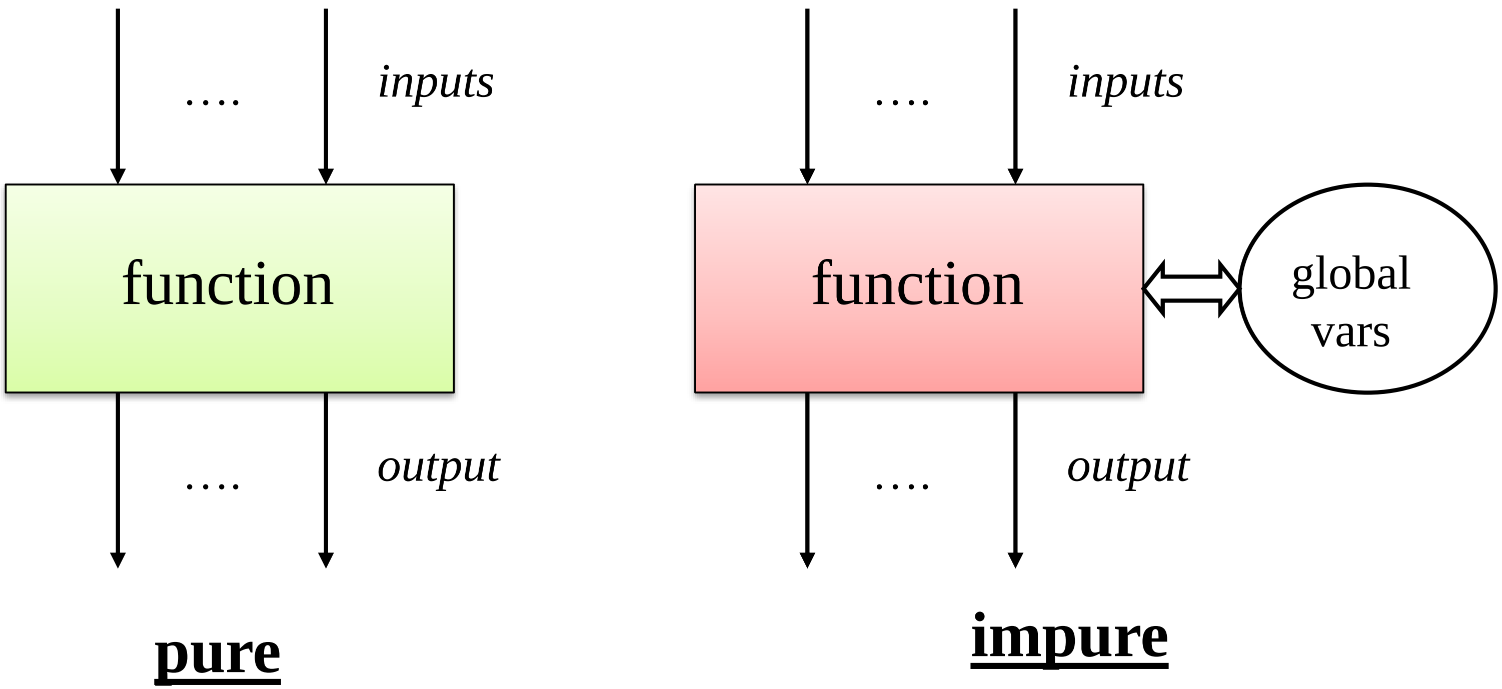
- GHC (Glasgow Haskell Compiler, version 10) is the version of Haskell I am using
  - GHCi is the REPL
  - Just enter `ghci` at the command line
- $ghci

  Prelude> "Hello, World!"

  Prelude> putStrLn "Hello World"
- Create object file: put putStrLn "Hello World" in file hw.hs
  - $ghc –o hello hw.hs
  - $./hello

Functions

- Function is a black box that converts input to output. A basis of software component.



Using Haskell

- You can do arithmetic at the prompt:
  - `Prelude> 2 + 2`
    `4`
- You can call functions at the prompt:
  - `Prelude> sqrt 10`
    `3.16228`
- The GHCi documentation says that functions must be loaded from a file:
  - `Prelude > :l "test.hs"`
    `Reading file "test.hs":`
- But you can define them in GHCi with `let`
  - `let double x = 2 * x`

Lexical issues

- Haskell is case-sensitive
  - Variables begin with a lowercase letter
  - Type names begin with an uppercase letter
- Indentation matters (braces and semicolons can also be used, but it's not common)
- There are two types of comments:
  - `--` (two hyphens) to end of line
  - `{-` multi-line `{-` these may be nested `-}` `-}`

Semantics of Haskell

- The best way to think of a Haskell program is as a single mathematical expression
  - In Haskell you do not have a sequence of "statements", each of which makes some changes in the state of the program
  - Instead you evaluate an expression, which can call functions
- Haskell is a functional programming language

Functional Programming (FP)

- Functions are first-class objects. That is, they are **values,** just like other objects are values, and can be treated as such Functions can be
  - assigned to variables, passed as parameters to higher-order functions, returned as results of functions
  - There is some way to write function literals
- Functions should *only* transform their inputs into their outputs
  - A function should have no side effects
    - It should not do any input/output
    - It should not change any state (any external data)

Functional Programming (FP)

- Given the same inputs, a function should produce the same outputs, every time--it is deterministic
- If a function is side-effect free and deterministic, it has referential transparency—all calls to the function could be replaced in the program text by the result of the function
    - But we need random numbers, date and time, input and output, etc.

Types

- Haskell is strongly typed…
- But type declarations are seldom needed, because Haskell does type inferencing
- Primitive types: `Int`, `Float`, `Char`, `Bool`
- Lists: `[2, 3, 5, 7, 11]`
  – All list elements must be the same type
- Tuples: `(1, 5, True, 'a')`
  – Tuple elements may be different types

Bool Operators

- `Bool` values are `True` and `False`
  - Notice how these are capitalized
- "And" is infix `&&`
- "Or" is infix `||`
- "Not" is prefix `not`
- Functions have types
  - "Not" is type `Bool -> Bool`
  - "And" and "Or" are type `Bool -> Bool -> Bool`

Arithmetic on Integers

- `+ - * / ^` are infix operators
  - Add, subtract, and multiply are type
    `(Num a) => a -> a -> a`
  - Divide is type `(Fractional a) => a -> a -> a`
  - Exponentiation is type
    `(Num a, Integral b) => a -> b -> a`
- `even` and `odd` are prefix operators
  - They have type `(Integral a) => a -> Bool`
- `div, quot, gcd, lcm` are also prefix
  - They have type `(Integral a) => a -> a -> a`

Floating-Point Arithmetic

- `+ - * / ^` are infix operators, with the types specified previously
- `sin`, `cos`, `tan`, `log`, `exp`, `sqrt`, `log`, `log10`
  - These are prefix operators, with type
  - `(Floating a) => a -> a`
- `pi`
  - Type `Float`
- `truncate`
  - Type `(RealFrac a, Integral b) => a -> b`

Operations on Chars

- These operations require `import Data.Char`
- `ord` is `Char -> Int`
- `chr` is `Int -> Char`
- `isPrint`, `isSpace`, `isAscii`, `isControl`, `isUpper`, `isLower`, `isAlpha`, `isDigit`, `isAlphaNum` are all `Char-> Bool`
- A string is just a list of `Char`, that is, `[Char]`
  - `"abc" == ['a', 'b', 'c']`

Polymorphic Functions

- `==`   `/=`
  - Equality and inequality tests are type
    `(Eq a) => a -> a -> Bool`
- `<`   `<=`   `>=`   `>`
  - These comparisons are type
    `(Ord a) => a -> a -> Bool`
- `show` will convert almost anything to a string
- Any operator can be used as infix or prefix
  - `(+) 2 2` is the same as `2 + 2`
  - ``100 `mod` 7`` is the same as `mod 100 7`

Simple Functions

- Functions are defined using =
  - `Prelude> avg x y = (x + y) / 2`
- `:type` or `:t` tells you the type
  - `Prelude>:t avg`

    `avg: (Fractional a) => a -> a -> a`

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 → .. → 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 → v , it follows that v :: t

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])
```

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)
```

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]              )    2
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
```

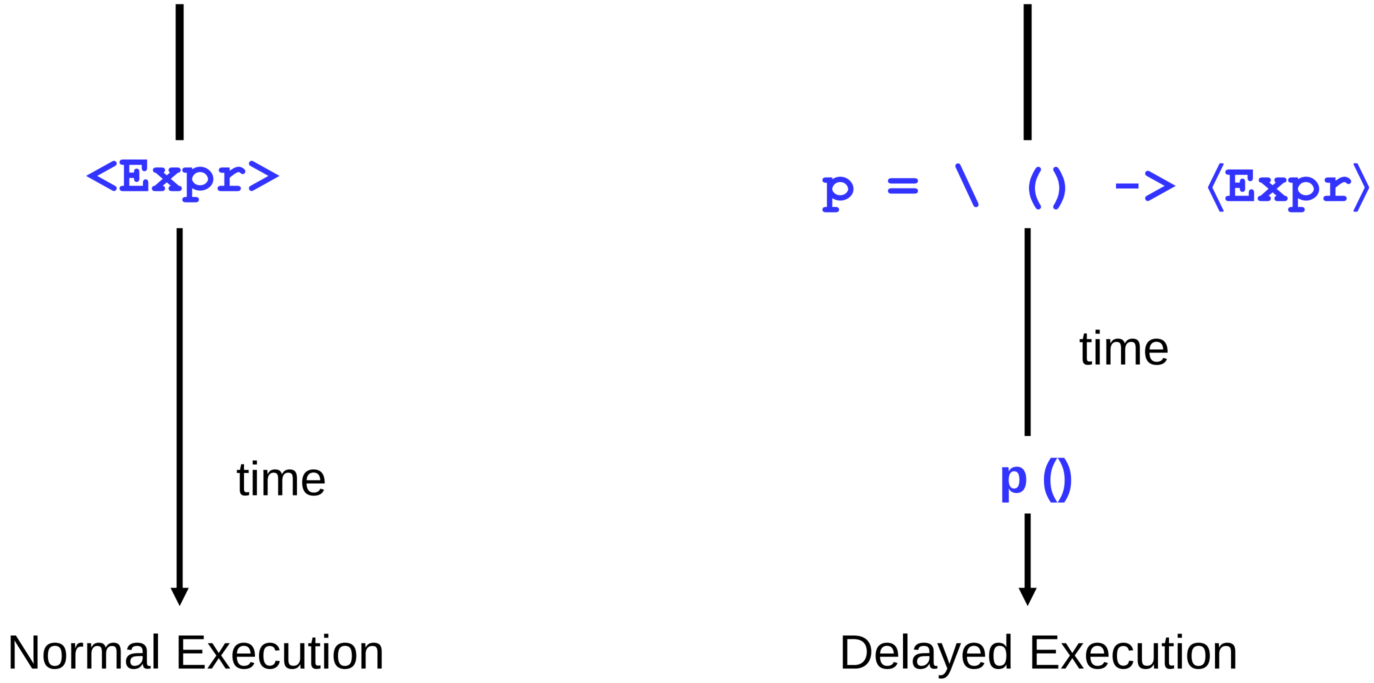
- In that case, some examples may be wrongly typed.

```
inc 3.2  →

inc 3  →
```

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.

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
```