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

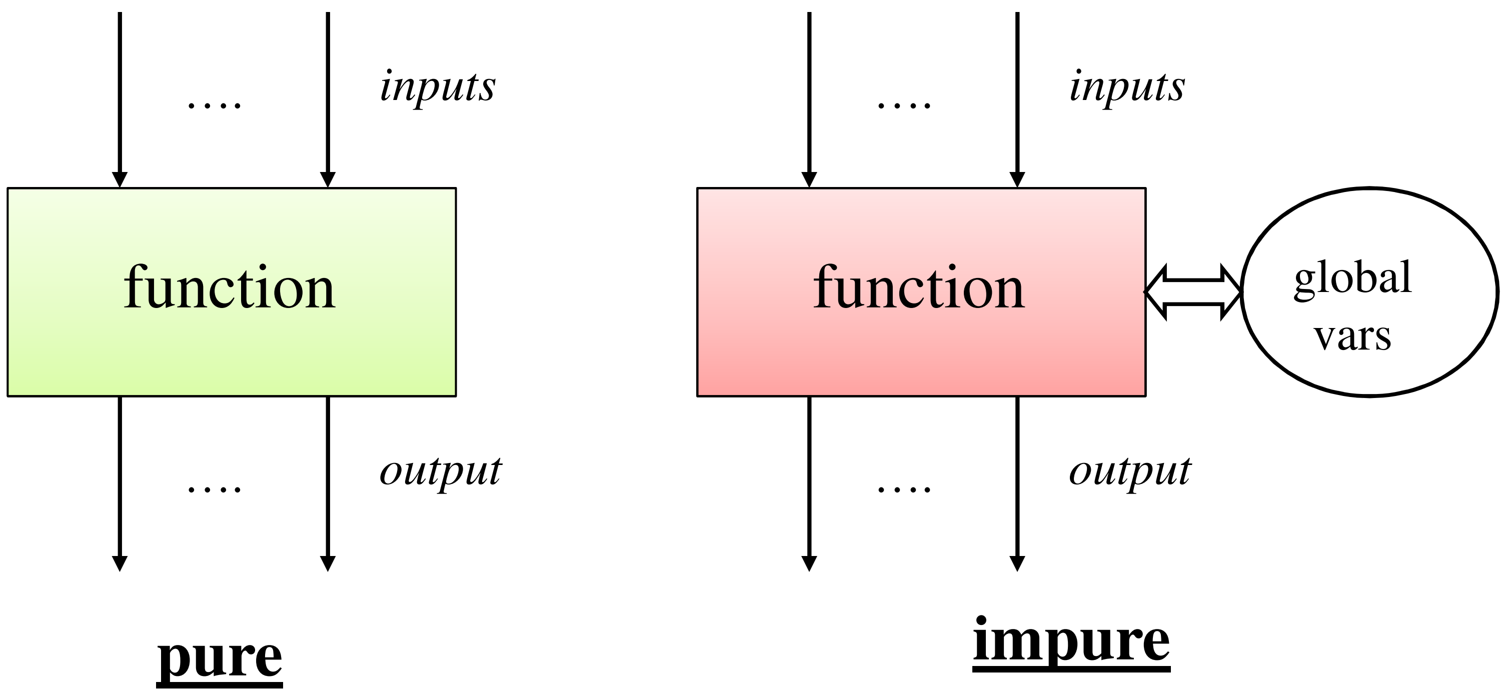
- GHC (Glasgow Haskell Compiler, version 10) is the version of Haskell I am using
  - GHCi is the REPL
  - Just enter `ghci` at the command line
- $ghci

  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)

Function Literals

- A Function literal is a function that is not declared but that is passed in as an expression. Lambdas and anonymous functions
- Can be assigned to variable or called directly
- Can appear anywhere that an expression can appear
- Higher order function
- Can reference variables defined in surrounding function and making them closure

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`

crazyyy

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:

  $e \rightarrow .. \rightarrow v$

- 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 $v :: t$

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 opetation to simple syntax