19CSE313-Principles of Programming Languages

Lab Exercise-2

Done By:

Adharsh S Mathew
CSE-D
AM.EN.U4CSE19302

Q1 [Record]

What is the type of the following functions? tail, sqrt, pi, exp, (^), (/=) and noOfSol? How can you query the interpreter for the type of an expression and how can you explicitly specify the types of functions in your program?

Answer

```
Prelude> :type tail
tail :: [a] -> [a]
```

Tail takes a **list of arbitrary type a as input** and returns a **list of arbitrary type a as output**.

```
Prelude> :type sqrt
sqrt :: Floating a => a -> a
```

Sqrt takes an input **of type a**, which is a **floating class** and returns the output of the same type.

```
Prelude> :type pi
pi :: Floating a => a
```

Pi does not take any input but **returns a value of type a,** which is a **floating class**.

```
Prelude> :type exp
exp :: Floating a => a -> a
```

Exp takes in a **floating class** argument and returns a **floating class**.

```
Prelude> :type (^)
(^) :: (Integral b, Num a) => a -> b -> a
```

^ takes in two values a and b. One of the **Integral type class** and the other of the **Num type class**.

```
Prelude> :type (/=)
(/=) :: Eq a => a -> a -> Bool
```

/= Takes in two values of any arbitrary type a and checks if they are not equal and returns a **Bool** value.

```
Prelude> :type noOfSol

<interactive>:1:1: error: Variable not in scope: noOfSol
```

```
Prelude> :type (**)
(**) :: Floating a => a -> a -> a
```

```
Prelude> :type (^)
(^) :: (Integral b, Num a) => a -> b -> a
```

```
Prelude> 2^3
8
Prelude> 2**3
8.0
Prelude> 2.3^3
12.166999999999996
Prelude> 2.3**3
12.166999999999998
```

```
Prelude> 2.3^3.3

<interactive>:19:1: error:
    • Could not deduce (Integral b0) arising from a use of '^'
      from the context: Fractional a
        bound by the inferred type of it :: Fractional a => a
        at <interactive>:19:1-7
      The type variable 'b0' is ambiguous
      These potential instances exist:
        instance Integral Integer -- Defined in 'GHC.Real'
        instance Integral Int -- Defined in 'GHC.Real'
        instance Integral Word -- Defined in 'GHC.Real'
        ...plus one instance involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In the expression: 2.3 ^ 3.3
      In an equation for 'it': it = 2.3 ^ 3.3

<interactive>:19:5: error:
    • Could not deduce (Fractional b0) arising from the literal '3.3'
      from the context: Fractional a
        bound by the inferred type of it :: Fractional a => a
        at <interactive>:19:1-7
      The type variable 'b0' is ambiguous
      These potential instances exist:
        instance Fractional Double -- Defined in 'GHC.Float'
        instance Fractional Float -- Defined in 'GHC.Float'
        ...plus one instance involving out-of-scope types
        (use -fprint-potential-instances to see them all)
    • In the second argument of '(^)', namely '3.3'
      In the expression: 2.3 ^ 3.3
      In an equation for 'it': it = 2.3 ^ 3.3
```

```
Prelude> 2.3**3.3
15.620749173070115
```

From the above example and definition we can derive that (^) is like (**) in operation, that is finding the power, but they take in different arguments. (a^b) can take in int/float as argument a and only int as argument b. (**) can take inputs of the floating class and return an output of the floating class.

:type/:t is used to query the interpreter to get the types of an expression.
Function :: data_type a => a->a -- this method is used to explicitly specify the type of user defined function.

Q2

Given the following definitions:

```
thrice x = [x, x, x]

sums (x : y : ys) = x : sums (x + y : ys)
sums xs           = xs
```

What does the following expression evaluate to?

```
map thrice (sums [0 .. 4])
```

Answer:

q2.hs 2

Haskell > Lab2 > q2.hs > sums

```
thrice :: a -> [a]
1  thrice x = [x,x,x]
sums :: Num a => [a] -> [a]
2  sums (x : y : ys) = x : sums (x + y : ys)
3  sums xs           = xs
```

```
Prelude> :load q2.hs
[1 of 1] Compiling Main             ( q2.hs, interpreted )
Ok, one module loaded.
*Main> map thrice (sums [0 .. 4])
[[0,0,0],[1,1,1],[3,3,3],[6,6,6],[10,10,10]]
*Main>
```

Q3 [Record]

Define a function product that produces the product fo a list of numbers, and show using your definition that product [2,3,4] = 24.

q3.hs

Haskell > Lab2 > q3.hs > products

```haskell
products :: Num p => [p] -> p
products []     = 1
products xs = foldl (*) 1 xs
```

```
(base) adharsh@adharsh-Inspiron-5570:~/Github/Haskell/Lab2$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load q3.hs
[1 of 1] Compiling Main             ( q3.hs, interpreted )
Ok, one module loaded.
*Main> products [2,3,4]
24
*Main> product [2,3,4]
24
```

Here Products is the user defined function and product is the built in function.

Q4

Record the types of the following values

['a', 'b', 'c']
('a', 'b', 'c')
[(False, '0'),(True, '1')]
([False,True],['0', '1'])
[tail, init, reverse]

```
Prelude> :type ['a', 'b', 'c']
['a', 'b', 'c'] :: [Char]
Prelude> :type ('a', 'b', 'c')
('a', 'b', 'c') :: (Char, Char, Char)
Prelude> :type [(False, '0'),(True, '1')]
[(False, '0'),(True, '1')] :: [(Bool, Char)]
Prelude> :type ([False,True],['0', '1'])
([False,True],['0', '1']) :: ([Bool], [Char])
Prelude> :type [tail, init, reverse]
[tail, init, reverse] :: [[a] -> [a]]
Prelude>
```

Q5

Record down definitions that have the following types; it does not matter what the definitions actually do as long as they are type correct.

bools :: [Bool]
nums :: Int
add :: Int -> Int -> Int -> Int
copy :: a -> (a, a)
apply :: (a -> b) -> a -> b

q5.hs

Haskell > Lab2 > q5.hs > ...

```haskell
bools :: Bool
bools = True
nums :: Int
nums = 10
adds :: Int -> Int -> Int -> Int
adds x y z = x+y+z
copy :: Int->(Int,Int)
copy a =  (a,a)
-- apply :: (Int->Int) ->Int->Int
-- apply (a,b) = a,b

```

```
(base) adharsh@adharsh-Inspiron-5570:~/Github/Haskell/Lab2$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> :load q5.hs
[1 of 1] Compiling Main             ( q5.hs, interpreted )
Ok, one module loaded.
*Main> :type bools
bools :: Bool
*Main> :type nums
nums :: Int
*Main> :type adds
adds :: Int -> Int -> Int -> Int
*Main> :type copy
copy :: Int -> (Int, Int)
*Main>
```

Q6

Record the types of the following functions

second xs = head (tail xs)
swap (x, y) = (y, x)
pair x y = (x, y)
double x = x * 2
palindrome xs = reverse xs == xs
twice f x = f (f x)

q6.hs 6

Haskell > Lab2 > q6.hs > twice

```haskell
second :: [a] -> a
second xs = head (tail xs)
swap :: (b, a) -> (a, b)
swap (x, y) = (y, x)
pair :: a -> b -> (a, b)
pair x y = (x, y)
double :: Num a => a -> a
double x = x * 2
palindrome :: Eq a => [a] -> Bool
palindrome xs = reverse xs == xs
twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

```
Prelude> :load q6.hs
[1 of 1] Compiling Main             ( q6.hs, interpreted )
Ok, one module loaded.
*Main> :type second
second :: [a] -> a
*Main> :type swap
swap :: (b, a) -> (a, b)
*Main> :type pair
pair :: a -> b -> (a, b)
*Main> :type double
double :: Num a => a -> a
*Main> :type palindrome
palindrome :: Eq a => [a] -> Bool
*Main> :type twice
twice :: (t -> t) -> t -> t
*Main>
```

Q7

Write a function named always0 ::Int → Int. The return value should always just be 0.

q7.hs 1

Haskell > Lab2 > q7.hs > always0

```haskell
always0 :: Int -> Int
always0 x = 0
```

```
Prelude> :load q7.hs
[1 of 1] Compiling Main             ( q7.hs, interpreted )
Ok, one module loaded.
*Main> :type always0
always0 :: Int -> Int
*Main> always0 10
0
*Main>
```

Q8

Write a function subtract :: Int → Int → Int that takes two numbers (that is, Ints) and subtracts them.

a. Will the above function work for Float type arguments? If not, rewrite the function.

q.hs

Haskell > Lab2 > q.hs > subtracts

```haskell
--always0 :: Int -> Int
--always0 x = 0

subtracts :: Int->Int->Int
subtracts x y = x - y
```

```
*Main> :reload
Ok, one module loaded.
*Main> subtracts 5 7
-2
*Main> subtracts 7 5
2
*Main>
```

Q9

Write a function addmult that takes three numbers. Let's call them p, q, and r. addmult should add p and q together and then multiply the result by r.

q.hs

Haskell > Lab2 > q.hs > addmult

```haskell
--always0 :: Int -> Int
--always0 x = 0

--subtracts :: Int->Int->Int
--subtracts x y = x - y

addmult :: Integral a => (a, a, a) -> a
addmult (p,q,r) = (p+q)*r
```

```
*Main> :reload
[1 of 1] Compiling Main             ( q.hs, interpreted )
Ok, one module loaded.
*Main> addmult (2,3,4)
20
*Main>
```