Index

Lecture 16

Evaluation order

Evaluation order

- Most languages use "call by value" for evaluation order:

- To evaluate "f (foo, bar)",
  evaluate foo and bar first (any one first depends on the language),
  then plug into f's body,
  evaluate the body.

- Example:

  If there is a function defined as f(x, y) = x:
  f (4+4, div(4, 2)) eval a parameter,
  arithmetic → f (8, div(4, 2)) eval the other parameter,
  arithmetic → f (8, 2) ready to plug in at last → 8

Evaluation order

-

A problematic parameter can cause an error/exception even if it would be unused:

f (4+4, div(1, 0)) eval a parameter,
arithmetic → f (8, div(1, 0)) eval the other parameter,
arithmetic -→div(1,0) will give error

In Functional Programming we use Lazy Evaluation

Lazy Evaluation

- Lazy evaluation is an evaluation strategy which holds the evaluation of an expression until its value is needed. It avoids repeated evaluation.
- **Haskell** is a good example of such a functional programming language whose fundamentals are based on Lazy Evaluation.

Example

- const x y = x
- Evaluation of const (4+4) (div 1 0):
- const (4+4) (div 1 0) plug in →4+4 arithmetic → 8
- (No error about dividing by zero.)

The Zip Function

A useful library function is zip, which maps two lists to a list of pairs of their corresponding elements.

```
zip :: [a] → [b] → [(a,b)]
```

For example

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

The Zip Function

- Using zip we can define a function returns the list of all <u>pairs</u> of adjacent elements from a list:

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]

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

Pattern matching

- When defining functions, you can define separate function bodies for different patterns.
- This leads to really neat code that's simple and readable.
- You can pattern match on any data type — numbers, characters, lists, tuples, etc.
- Example:

```
lucky :: (Integral a) => a -> String
lucky 9 = "LUCKY NUMBER Nine!"
lucky x = "Sorry, you're out of luck "
```

Pattern matching

- Pattern Matching for Strings

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

- Pattern Matching for Tuples

```
addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Recursive Functions

In Haskell, functions can also be defined in terms of themselves. Such functions are called <u>recursive</u>.

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

fac maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

Example

```
  fac 3
=
  3 * fac 2
=
  3 * (2 * fac 1)
=
  3 * (2 * (1 * fac 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Recursion on Lists

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product :: Num a ⇒ [a] → a
product []     = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

Example

```
product [2,3,4]
=
2 * product [3,4]
=
2 * (3 * product [4])
=
2 * (3 * (4 * product []))
=
2 * (3 * (4 * 1))
=
24
```

Recursion on Lists

Using the same pattern of recursion as in product we can define the length function on lists.

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

Example

```
length [1,2,3]
=
1 + length [2,3]
=
1 + (1 + length [3])
=
1 + (1 + (1 + length []))
=
1 + (1 + (1 + 0))
=
3
```

Recursion on Lists

Using a similar pattern of recursion we can define the reverse function on lists.

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

Example

```
   reverse [1,2,3]
=
   reverse [2,3] ++ [1]
=
   (reverse [3] ++ [2]) ++ [1]
=
   ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
   (([] ++ [3]) ++ [2]) ++ [1]
=
   [3,2,1]
```

Pattern matching: Example on list

```
noVowels :: [Char] -> [Char]
noVowels "" = ""
noVowels (x:xs) = if x `elem` "aeiouAEIOU"
                  then noVowels xs
                  else x: noVowels xs
```

Exercise:

Find the sum of numbers in a list

Lecture 17

Higher Order Functions

High Order Functions

A function is called <u>higher-order</u> if it takes a function as an argument or returns a function as a result.

```
twice :: (a → a) → a → a
twice f x = f (f x)
```

twice is higher-order because it takes a function as its first argument.

High Order Functions : Examples

```
add1 :: Int -> Int
add1 x = x+1
```

```
g :: Int -> (Int -> Int)
g x = add1
```

```
f :: (Int -> Int) -> Int
f x = 3
```

```
h :: Int -> (Int -> Int)
h x y = x + y
```

The Map Function

The map function can be defined in a particularly simple manner using a list comprehension:

```
map :: (a → b) → [a] → [b]
```

The higher-order library function called map applies a function to every element of a list.

```
map f xs = [f x | x ← xs]
```

The Map Function

The map function can also be defined using recursion:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

The Map Function

Example

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

```
>map   add1  [1,2,3,4]

[2,3,4,5]
```

The Map Function

Any Haskell Function that appears to take multiple arguments can be partially applied

```
h :: Int -> (Int -> Int)
h x y = x + y
```

```
Prelude  > h 3 4
Prelude > (h 3) 4
```

```
Prelude> (max  3)  [1,2,3,4,5]
[3,3,3,4,5]
```

Function Types

A function is a mapping from values of one type to values of another type:

```
not  :: Bool → Bool

even :: Int → Bool
```

In general:

$t1 \rightarrow t2$ is the type of functions that map values of type t1 to values to type t2.

Function Types

The argument and result types are unrestricted.

For example, functions with multiple arguments or results are possible using lists or tuples:

```
add          :: (Int,Int) → Int
add (x,y)     = x+y

zeroto       :: Int → [Int]
zeroto n      = [0..n]
```

Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

```
add'     :: Int → (Int → Int)
add' x y = x+y
```

add' takes an integer x and returns a function add' x.
In turn, this function takes an integer y and returns the result x+y.

Curried Functions

add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time:

```
add  :: (Int,Int) → Int

add' :: Int → (Int → Int)
```

Functions that take their arguments one at a time are called <u>curried</u> functions

Curried Functions

Functions with more than two arguments can be curried by returning nested functions:

```
mult        :: Int → (Int → (Int → Int))
mult x y z = x*y*z
```

mult takes an integer x and returns a function mult x, which in turn takes an integer y and returns a function mult x y, which finally takes an integer z and returns the result x*y*z.

Curried Function

Example:

```
sum3 :: Int  ->  (Int -> (Int -> Int))
sum3  a  b  c = a + b + c
```

```
Prelude >  sum3  4  5  6
Prelude > ((sum  3)  4)  5)  6
```

Why is Currying Useful?

useful functions can often be made by <u>partially applying</u> a curried function.

For example:

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

In Haskell all functions are automatically curried

Currying Conventions

The arrow $\rightarrow$ associates to the right.

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

```
Int → Int → Int → Int
```

Means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int)).

Currying Conventions

It is natural for function association to be Left.

```
mult x y z
```

Means ((mult x) y) z.

All functions in Haskell are normally defined in curried form.

Example

Any Haskell Function that appears to take multiple arguments can be partially applied

```
h :: Int -> (Int -> Int)
h x y = x + y
```

```
Prelude  > h 3 4
Prelude > (h 3) 4
```

```
Prelude> (max  3)  [1,2,3,4,5]
[3,3,3,4,5]
```

```
> map  (/10)   [1,2,3,4]
> map  (10/)   [1,2,3,4]
```

Curried function : Example

Haskell functions are curried functions

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

```
prelude> let aaa = multThree 9
prelude> aaa 2 3
54
prelude> let bbb = aaa 2
prelude> bbb 10
180
```

The Filter Function

The higher-order library function <u>filter</u> selects every element from a list that satisfies a predicate.

```
filter :: (a → Bool) → [a] → [a]
```

For example:

```
> filter even [1..10]

[2,4,6,8,10]
```

```
>filter (>5) [1..10]

[6,7,8,9,10]
```

Filter can be defined using a list comprehension:

```
filter p xs = [x | x ← xs, p x]
```

Alternatively, it can be defined using recursion:

```
filter p [] = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise = filter p xs
```

zipWith Function

```
>zipWith  (+)   [1,2,3,4]   [5,6,7,8]


>zipWith  (*)   [1,2,3,4]   [5,6,7,8]
```

Flip function

```
prelude> :t flip
flip :: (a -> b -> c) -> b -> a -> c
```

- The first argument is a curried function of two arguments of types a and b that returns something of type c, where a, b and c are arbitrary types.

- The second argument is of type b, as was the second argument of the input function

- Third argument is of type a, as was the first argument of the input function. And the result is of type c.

.

Flip function

So, when we apply flip to a curried function of two arguments, we are left with a curried function of two arguments, with types of arguments “flipped”, i.e. their position changed.

Example:

```
from :: Int -> Int -> Int
from = flip (-)
```

```
Prelude >  5  `from`  8
```

Flip function : Example

prelude > flip (/) 1 2
Output: 2.0

Prelude > flip (>) 3 5
Output: True

Prelude > flip mod 3 6
Output: 0

Lecture 18

Higher Order Functions – Part 2

The Foldr Function

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f []     = v
f (x:xs) = x ⊕ f xs
```

f maps the empty list to some value v, and any non-empty list to some function $\oplus$ applied to its head and f of its tail.

For example:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

v = 0

$\oplus$ = +

```
product []     = 1
product (x:xs) = x * product xs
```

v = 1

$\oplus$ = *

```
and []     = True
and (x:xs) = x && and xs
```

v = True

$\oplus$ = &&

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function ⊕ and the value v as arguments.

For example:

```
sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True
```

Foldr itself can be defined using recursion:

```
foldr :: (a → b → b) → b → [a] → b
foldr f v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

For example:

```
sum [1,2,3]
```

=

```
foldr (+) 0 [1,2,3]
```

=

```
foldr (+) 0 (1:(2:(3:[])))
```

=

```
1+(2+(3+0))
```

=

```
6
```

Replace each (:)
by (+) and [] by 0.

For example:

```
  product [1,2,3]
=
  foldr (*) 1 [1,2,3]
=
  foldr (*) 1 (1:(2:(3:[])))
=
  1*(2*(3*1))
=
  6
```

Replace each (:)
by (*) and [] by 1.

Foldr Examples

foldr (+) 5 [1,2,3,4]

foldr (/) 2 [8,12,24,4]

foldr (/) 3 []

foldr (&&) True [1>2,3>2,5==5]

foldr max 18 [3,6,12,4,55,11]

foldr max 111 [3,6,12,4,55,11]

foldr (\x y -> (x+y)/2) 54 [12,4,10,6]

Foldr Example

```
firstone :: (a->Bool) -> a -> [a] ->a
firstone f = foldr (\x acc -> if f x then x else acc)
```

```
firstone (>0)  100  [-3,5,7,-2]
```

Output : 5

```
Firstone  (>10)  100  [-3,5,7,-2]
```

Output : 100

Foldl Function

foldl (/) 64 [4,2,4]

foldl (/) 3 []

foldl max 5 [1,2,3,4]

foldl max 5 [1,2,3,4,5,6,7]

foldl (\x y -> 2*x + y) 4 [1,2,3]

Other Library functions

The library function <u>all</u> decides if every element of a list satisfies a given predicate.

```
all :: (a → Bool) → [a] → Bool
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]

True
```

Dually, the library function <u>any</u> decides if at least one element of a list satisfies a predicate.

```
any :: (a → Bool) → [a] → Bool
any p xs = or [p x | x ← xs]
```

For example:

```
> any (== ’a’) "abcdef"

True
```

The library function takeWhile selects elements from a list while a predicate holds of all the elements.

```
takeWhile :: (a → Bool) → [a] → [a]
takeWhile p [] = []
takeWhile p (x:xs)
   | p x         = x : takeWhile p xs
   | otherwise = []
```

For example:

```
> takeWhile (/= ' ') "abcdef"

"abc"
```

Dually, the function <u>dropWhile</u> removes elements while a predicate holds of all the elements.

```
dropWhile :: (a → Bool) → [a] → [a]
dropWhile p [] = []
dropWhile p (x:xs)
   | p x         = dropWhile p xs
   | otherwise = x:xs
```

For example:

```
> dropWhile (== ’a’) “abc"
```

Polymorphic Functions

A function is called <u>polymorphic</u> ("of many forms") if its type contains one or more type variables.

```
length :: [a] → Int
```

For any type a, length takes a list of values of type a and returns an integer.

Polymorphic Functions

Type variables can be instantiated to different types in different circumstances:

```
> length [False,True]
2

> length [1,2,3,4]
4
```

a = Bool

a = Int

Type variables must begin with a lower-case letter, and are usually named a, b, c, etc.

Example of Polymorphic functions

```
fst :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip :: [a] → [b] → [(a,b)]

id :: a → a
```

Overloaded Functions

A polymorphic function is called <u>overloaded</u> if its type contains one or more class constraints.

```
(+) :: Num a ⇒ a -> a -> a
```

For any numeric type a, (+) takes two values of type a and returns a value of type a.

Overloaded Functions

Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> ’a’ + ’b’
ERROR
```

a = Int

a = Float

Char is not a numeric type

Haskell has a number of type classes, including:

`Num` - Numeric types

`Eq` - Equality types

`Ord` - Ordered types

For example:

```
(+)  :: Num a ⇒ a → a → a
(==) :: Eq a  ⇒ a → a → Bool
(<)  :: Ord a ⇒ a → a → Bool
```

Exercises

(1) What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail,init,reverse]
```

(2) What are 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)
```

(3) Check your answers using GHCi.