Curried Functions in Haskell

Principles of Programming Languages

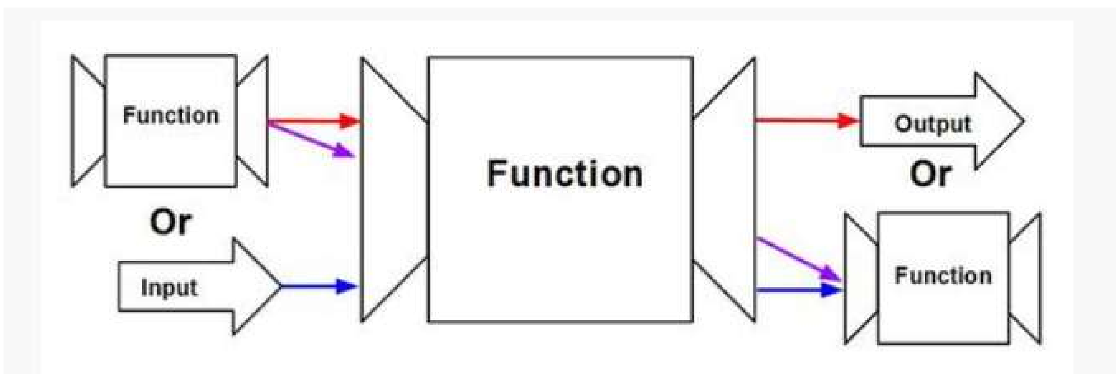
Curry

A Tasty dish?

Haskell Curry!

Higher order functions

- Haskell functions can take functions as parameters and return functions as return values.
- A function that does either of those is called a higher order function.



Curried Functions

- Currying is a functional programming technique that takes a function of N arguments and produces a related one where some of the arguments are fixed.
- All the functions that accepted several parameters so far have been curried functions.

A tasty dish?

- Currying was named after the Mathematical logician Haskell Curry (1900-1982)
- Curry worked on combinatory logic.
- A technique that eliminates the need for variables in mathematical logic.
- and hence computer programming!
  - At least in theory
- The functional programming language Haskell is also named in honor of Haskell Curry

Functions in Haskell

- *All* functions in Haskell are curried, i.e., *all Haskell functions take just single arguments.*
- This is mostly hidden in notation and is not apparent to a new Haskeller.
- Let's take the function `div :: Int -> Int -> Int` which performs integer division.
- The expression `div 11 2` evaluates to 5
- But it's a two-part process
  - div 11 is evaluated & *returns a function* of type Int -> Int
  - That function is applied to the value 2, yielding 5

Curried functions

- Functions with multiple arguments are 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.**

Note:

- 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 curried functions, celebrating the work of Haskell Curry on such functions.

Curried functions with multiple argument

- 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 and integer **z** and returns the result **x * y *z.**

Curried functions

- Let’s take an example - the max function.
- It looks like it takes two parameters and returns the one that's bigger.
- Doing max 4 5 first creates a function that takes a parameter and returns either 4 or that parameter, depending on which is bigger. Then, 5 is applied to that function and that function produces our desired result.
- The following two calls are equivalent:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Curried functions

- Let's examine the type of max : max :: (Ord a) => a -> a -> a
- This can also be written as max :: (Ord a) => a -> (a -> a)
- That could be read as: max takes an a and returns (that's the ->) a function that takes an a and returns an a.
- If we call a function with too few parameters, we get back a partially applied function, meaning a function that takes as many parameters as we left out.

Why is currying useful?

- Curried functions are more flexible than functions on tuples, because useful functions can often be made by partial applying a curried function.

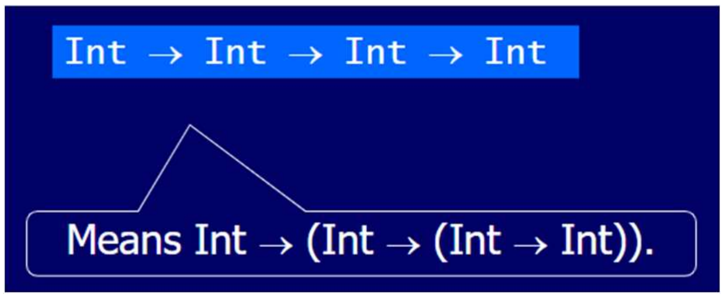
- For example

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

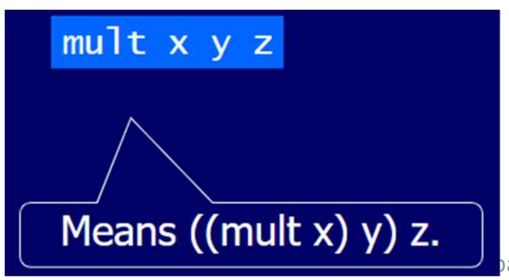
Currying Conventions

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

1. The arrow → associated to the right.



2. As a consequence, it is natural for function application to associate to the **left**.



Polymorphic functions

Polymorphic functions

- A function is called polymorphic(“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.

- 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 variable must begin with lower case letter, usually named as a,b,c etc

Polymorphic functions

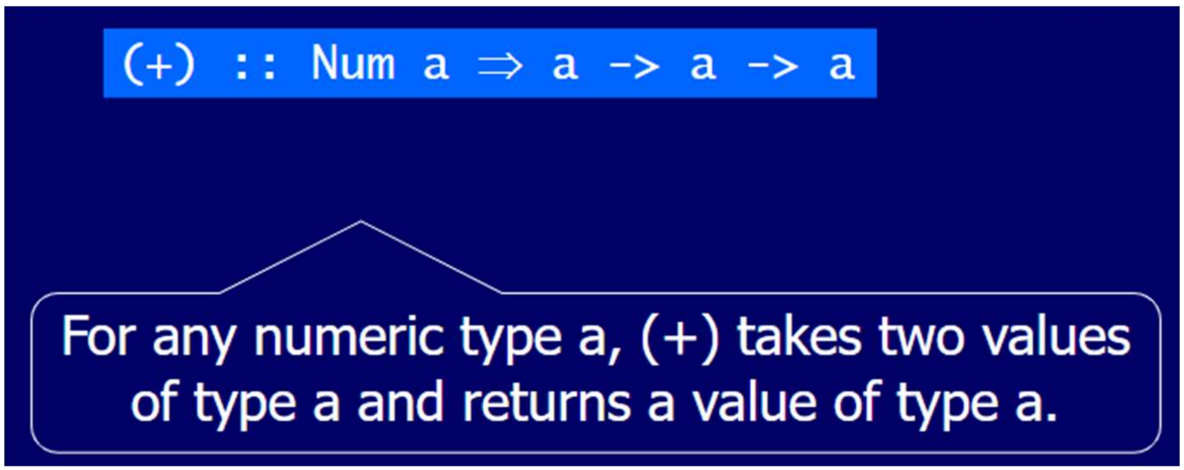
- Many functions defined in the standard prelude are polymorphic.
- For example

```
fst  :: (a,b) → a

head :: [a] → a

take :: Int → [a] → [a]

zip  :: [a] → [b] → [(a,b)]

id   :: a → a
```

Overloaded functions

Overloaded functions

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



Overloaded functions

- Haskell has number of type classed, 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
```

- Constraints type variable 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

Recursion

Something is recursive if it's defined in terms of itself

Rules for recursion

- Recursive functions play a central role in Haskell and are used throughout computer science and mathematics generally.
- Recursion is basically a form of repetition, and we can understand it by making distinct what it means for a function to be recursive, as compared to how it behaves.

Recursion

- In Haskell, many problems are solved using recursion.
- The main idea: divide the problem into smaller subproblems and try to solve these subproblems as simplest cases first.
- The simplest case is called the base case.

Recursion – Sum of n natural numbers

- Let's assume natSum is a function to compute sum of n natural numbers. Let’s observe the working with an example

```
natSum 5  ⇒ 5 + natSum (5 - 1)
          ⇒ 5 + natSum 4
          ⇒ 5 + (4 + natSum (4 - 1))
          ⇒ 5 + (4 + natSum 3)
          ⇒ 5 + (4 + (3 + natSum (3 - 1)))
          ⇒ 5 + (4 + (3 + natSum 2))
          ⇒ 5 + (4 + (3 + (2 + natSum (2 - 1))))
          ⇒ 5 + (4 + (3 + (2 + natSum 1)))
          ⇒ 5 + (4 + (3 + (2 + (1 + natSum (1 - 1)))))
          ⇒ 5 + (4 + (3 + (2 + (1 + natSum 0))))
          ⇒ 5 + (4 + (3 + (2 + (1 + 0))))
          ⇒ 15
```

The definition of natSum is called recursive, because natSum itself is used in the definition of natSum — i.e., a recursive function is a function that makes use of itself in its definition.

Recursion – Sum of n natural numbers

- Recursive function definitions have at least two cases:
  - The base case specifies what to do in the simplest form of input (where the function stops calling itself).
  - The stepping case includes the recursive use of the function by itself.

```
natSum :: Num a => a -> a
natSum 0  = 0                      -- base case
natSum n  = n + natSum (n - 1)     -- recursive/stepping case
```

Recursion – Sum of n natural numbers

- An alternative way of writing the recursive definition of natSum would be

```
natSum :: Num a => a -> a
natSum n = if n == 0
              then 0
              else n + natSum (n - 1)
```

- It contains only one equation and makes the case distinction explicit through a conditional.

Recursion - GCD

- The greatest common divisor of two numbers is the largest number that evenly divides them both.
- For computing the greatest common divisor of two positive numbers m and n is the following

  1. if $m = 0$, then the greatest common divisor of m and n is n;
  2. if $m < n$, then swap m and n;
  3. replace m with $m - n$ and go back to step 1.

Recursion - GCD

We can visualize the algorithm at work on the numbers m = 6 and n = 15 as a sequence of states, thus

$$\begin{array}{|c|c|}\hline m & 6 \\ \hline n & 15 \\ \hline\end{array} \Longrightarrow \begin{array}{|c|c|}\hline m & 15 \\ \hline n & 6 \\ \hline\end{array} \Longrightarrow \begin{array}{|c|c|}\hline m & 9 \\ \hline n & 6 \\ \hline\end{array} \Longrightarrow \begin{array}{|c|c|}\hline m & 3 \\ \hline n & 6 \\ \hline\end{array} \Longrightarrow \begin{array}{|c|c|}\hline m & 6 \\ \hline n & 3 \\ \hline\end{array} \Longrightarrow \begin{array}{|c|c|}\hline m & 3 \\ \hline n & 3 \\ \hline\end{array} \Longrightarrow \begin{array}{|c|c|}\hline m & 0 \\ \hline n & 3 \\ \hline\end{array}$$

```
euclid :: Int → Int → Int
euclid 0 n = n
euclid m n | m < n = euclid n m
           | otherwise = euclid (m - n) n
```

Recursion Example- Factorial

- Factorial of a number n is the product of all number numbers between 1 and n. $n! = 1 \times 2 \times 3 \times \cdots \times n$
- where 0! = 1 by convention. The factorial function can be written recursively as

```
fact :: Int → Int
fact 0 = 1
fact n = n * fact (n - 1)
```

Recursion Example- Factorial

- Non-Recursive Version :

```
> fact :: (Eq a, Num a) => a -> a
> fact x = if x==0 then 1 else x * fact (x-1)
```

- Recursive version with pattern:-

```
> fact' :: (Eq a, Num a) => a -> a
> fact' 0 = 1
> fact' x = x * fact' (x-1)
```

Recursion Example – Fibonacci Series

- The sequence of Fibonacci numbers begins like this

  0 1 1 2 3 5 8 13 21 34 55 89 · · ·

- The sequence begins with 0 and 1, and every following number is the sum of the previous two.

```
> fib 0 = 0
> fib 1 = 1
> fib n = fib (n-1) + fib (n-2)
```

Recursion Example – Power

- The n-th power of an integer number a can be inductively defined as follows:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a^{n/2} \times a^{n/2} & \text{if } n > 0 \text{ and } n \text{ is even} \\ a \times a^{n/2} \times a^{n/2} & \text{otherwise} \end{cases}$$

- we can define a Haskell function for computing a^n thus

```
power :: Int → Int → Int
power _ 0 = 1
power a n | even n = a' * a'
          | otherwise = a * a' * a'
    where
      a' = power a (n `div` 2)
```

STRUCTURAL RECURSION

- Focus on recursion over a data structure.
- This is typically a LIST or a TREE, but more generally it can be any recursive structure.
- Such a recursion is called STRUCTURAL RECURSION.
- We use structural recursion to process a data structure.

Structural Recursion

- The main idea: recurse down the structure by gradually decomposing it using pattern matching and combining the results.
- Some examples can be:

```
> sum' :: Num a => [a] -> a
> sum' []     = 0
> sum' (x:xs) = x + sum' xs
```

```
> length' :: [a] -> Int
> length' []     = 0
> length' (_:xs) = 1 + length' xs
```

Next – List in Haskell