Currying, Function Application and Partial Application

Doh Young Mo
Seoul National University

July 2, 2014

What is Functional Programming?

- Function as 1st class value
  - Function as
    - argument
    - return
    - data type
- Function can be placed in anywhere, e.g., the position of the variable.

Higher order function

a function of a function of a ... = Higher order function

This concept is used in the currying.

Let the $f(x, y) = x + y$: multivariable function.

Evaluation procedure is

$f(x, y) = x + y$

$g(y) \equiv f(x_0, y)$

$f(x_0, y_0) = g(y_0) = x_0 + y_0$

$x \mapsto (y \mapsto (x + y))$.

Higher order function

a function of a function of a ... = Higher order function

This concept is used in the currying.

Let the $f(x,y) = x + y$: multivariable function.

This evaluation procedure can be generalized into

```
f :: a -> (b -> c).
```

This technique is called currying.

Because `->` is right associative, we can drop the parenthesis.

```
f :: a -> b -> c
```

Currying and Uncurrying

$f(x, y) = x + y$

Currying:

```
f :: (Num a)=>a->a->a
f x y = x + y
```

Uncurrying:

```
f :: (Num a)=>(a,a)->a
f (x,y) = x + y
```

Uncurrying is a dual transformation of currying:

$f\,x\,y \sim f(x, y)$

Mathematical View

Tuple is the cartesian product :

$X \times Y = \{(x, y)|x \in X, y \in Y\}.$

We define the following set

$B^A \equiv \{f|f :: A \to B\}.$

This notation is good because it satisfies the properties of the exponential function. It can be proved formally, but I want to give some heuristic justification.

Mathematical View

Let $A = \{a_1, a_2, ..., a_n\}$ and $B = \{0, 1\}$.

$\{f | f :: A \to B\} = B^A = \{(x_1, x_2, ..., x_n) | x_i = 0 \text{ or } 1, i = [1..n]\}$

In B^A, $|B|$ represents possible values of each element of the tuple, $|A|$ means the length of the tuple.

$\{f | f :: B \to A\} = A^B \sim A^2 = A \times A = \{(a_1, a_2) | a_i \in A\}$

It can be easily generalized to $B = \{b_1, b_2, ..., b_m\}$.

Mathematical View

(uncurrying f :: (A, B) -> C) $= C^{(A\times B)} = (C^B)^A$

$= A$ -> $C^B = A$ -> B -> C

$f\,x\,y \sim f(x, y)$

More stuff...

$A \times B = \{(a, b)\}$ is like a and b. Then what is $A + B$? The answer is a or b. Then by the previous notation,

$A + B \to C = C^{(A+B)} = C^A \times C^B = (A \to C, B \to C)$ This is implemented in Haskell as Either.

```
either :: (a->c) -> (b->c) -> Either a b -> c
```

What is the virtue of the currying?

Function application in haskell

```
f x y = ((f x) y)
```

This means function application is done by infix operator ␣, and ␣ is left associative.

- currying is automatically done.
- defining a new function naturally.
- From now we think the function add.

```
add :: (Num a)=>a->a->a
add x y = x + y
```

Currying the add function

We can make a function with fewer arguments by just giving some arguments.

```
(add 3) :: (Num a)=>a->a
(add 3) x = 3 + x

map :: (a -> b) -> [a] -> [b]

map (add 3) [1,2,3,4]
= [4,5,6,7]
```

How the currying can be useful?

The key is the type system.

- Type system provides two advantages:
  - Safety
  - Convenience

How the currying can be useful?

The key is the type system.

Safety:

```
print :: Show a=>a->IO()

print add 3 4       (X)
print (add 1)       (X)
print (add 1 2 3)   (X)
```

Convenience:

```
add :: a -> a -> a
(add 3) :: a -> a
-- You have to put
-- this in your code
```

Currying is good, but is it good enough?

Let's think about following example.

```
add3 = add 3
add3 (add3 (add3 (add3 3)))
= 15
```

What I did is just $3 + 3 + 3 + 3 + 3$.

Many parantheses, so we adopt function application operator ($).

It makes the function application right associtative.

Function Application Operator

```
add3 add3 add3 add3 3 = (((add3 add3) add3) add3) 3 (X)
add3 $ add3 $ add3 $ add3 $ 3
= add3 (add3 (add3 (add3 3)))
```

Second expression is much more readable because there's no parantheses and it's similar to our notation for math!

Function composition operator (.)

Let's think about $f(g(h(x)))$.

$f(g(h(x))) = (f \circ g \circ h)(x)$

x can be drop out because we know types of functions f,g,h.

For this purpose, we adopt (.).

```
(.) :: (b->c) -> (a->b) -> (a->c)
```

```
add3 (add3 (add3 (add3 3)))
= add3 $ add3 $ add3 $ add3 $ 3
= add3 . add3 . add3 . add3 $ 3
= (add3 . add3 . add3 . add3) 3
```

Example

In haskell, we use $++$ operator to combine lists.

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys

[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]
```

Example

In haskell, we use $++$ operator to combine lists.

```
(++) :: [a] -> [a] -> [a]
(++) []     ys = ys
(++) (x:xs) ys = x : xs ++ ys
```

```
[1,2,3] = 1:2:3:[]
```

```
[1,2,3] ++ [4] = 1:[2,3]++[4] = 1:2:[3]++[4]
= 1:2:3:[]++[4]=1:2:3:[4]
```

It's $\mathcal{O}(n)$ operaton, which is expensive.

Example

In haskell, we use $++$ operator to combine lists.

```
DList :: [a] -> [a]
(cons a) = (cons a) .
(snoc a lst) = lst . (cons x)
```

Function composition is stored as a thunk due to lazy evaluation. It's $\mathcal{O}(1)$ operation.

Example

In haskell, we use $++$ operator to combine lists.

```
DList :: [a] -> [a]
cons a lst = a . lst
snoc lst b = lst . b
(DList [a]) [] = [a]
```

Function composition is stored as a thunk due to lazy evaluation. It's $\mathcal{O}(1)$ operation.

If we want to know the value of the list, we should apply this function to the empty list.

Conclusion

- We have learned (->), ($) and (.).
- (->) is right associative. It naturally gives currying.
- ($) makes the function application right associative.
- (.) makes the point-free expression possible.

End

Thank you for your attention!