Programming Languages

Lec 2: Functional Programming in OCaml

김재호

“Functional” is not a silver-bullet, but…

- Recursion과 higher-order functions는 functional programming의 핵심!
- Recursion(재귀)
  - 모든 loop는 사실 recursive하게 표현할 수 있으며, 더 간결하게 표현되기도 함
    - e.g., 1부터 n까지 반복하며 더한 결과를 구하라 → n + (1부터 n-1까지의 합)
  - 대부분의 computational problem, structure는 재귀적으로 정의됨
    - e.g., Binary Tree -> (Root, (Left subtree), (Right subtree))
    - e.g., List -> [1; 2; 3; 4] == 1::2::3::4::[] == Cons(1, Cons(2, Cons(3, Cons(4, []))))

“Functional” is not a silver-bullet, but…

- Recursion과 higher-order functions는 functional programming의 핵심!
- Higher-order Function(고차함수)
  - 함수를 값으로 사용함으로써, 강력한 abstraction(추상화)을 구사할 수 있다
  - 함수를 하나의 구멍뚫린 문장으로 표현한다 생각해 보자 (구멍이 함수의 parameter)
    - 지금까지의 함수: 주어, 목적어만 parametrize
      - e.g., 리스트 OO의 원소들의 합을 구한다, 수 OO에 O를 더한 뒤 제곱한다
    - Higher-Order Func: 주어, 목적어에 서술어까지 parametrize
      - e.g., 리스트 OO의 원소들을 OO한다, 수 OO에 O를 OO한 뒤 OO한다

Recursion

Recursive Functions

Example: list length

- 어떤 리스트의 길이는 다음과 같이 재귀적으로 정의할 수 있다:
  - 만약 리스트가 비어 있다면, 그 길이는 0이다.
  - 그렇지 않다면, 리스트를 hd::tl로 쪼개 생각할 수 있고, tl의 길이에 1을 더하면 된다.

∴

```
let rec length l =
  match l with
  | [] → 0
  | hd::tl → 1 + length tl
```

Recursive Functions

Exercise 1: append

- 이번엔 두 리스트를 받아 이어 붙이는 함수 append를 만들어 보자
  - 첫 번째 리스트가 빈 리스트라면, append 결과는 두 번째 리스트 그대로일 것이다.
  - 그렇지 않다면, 첫 번째 리스트를 hd::tl로 쪼갠 뒤…

```
let rec append l1 l2 =
```

Recursive Functions

Exercise 1: append

- 이번엔 두 리스트를 받아 이어 붙이는 함수 append를 만들어 보자
  - 첫 번째 리스트가 빈 리스트라면, append 결과는 두 번째 리스트 그대로일 것이다.
  - 그렇지 않다면, 첫 번째 리스트를 hd::tl로 쪼갠 뒤…

```
let rec append l1 l2 =
  match l1 with
  | [] → l2
  | hd::tl → hd::(append tl l2)
```

Recursive Functions

Exercise 2: reverse

- 이번엔 리스트를 받아 그 리스트 순서를 뒤집어 주는 함수 reverse를 만들어 보자

```
let rec reverse l =
```

Recursive Functions

Exercise 2: reverse

- 이번엔 리스트를 받아 그 리스트 순서를 뒤집어 주는 함수 reverse를 만들어 보자

```
let rec reverse l =
  match l with
  | [] → []
  | hd::tl → append (reverse tl) [hd]
```

Recursive Functions

Exercise 3: insert

- 이번엔 원소 하나와 정렬된 리스트를 받아 원소가 추가되었고 여전히 정렬된 리스트를 만들어 주는 함수 insert를 만들어 보자

```
let rec insert a l =
```

Recursive Functions

Exercise 3: insert

- 이번엔 원소 하나와 정렬된 리스트를 받아 원소가 추가되었고 여전히 정렬된 리스트를 만들어 주는 함수 insert를 만들어 보자

```
let rec insert a l =
  match l with
  | [] → [a]
  | hd::tl → if a < hd then a::l else hd::(insert a tl)
```

Recursive Functions

Exercise 4: insertion sort

- 이제 리스트를 하나 받아 insertion sort 알고리즘을 수행해 정렬된 리스트를 만들어 주는 함수 sort를 만들어 보자

```
let rec sort l =
```

Recursive Functions

Exercise 4: insertion sort

- 이제 리스트를 하나 받아 insertion sort 알고리즘을 수행해 정렬된 리스트를 만들어 주는 함수 sort를 만들어 보자

```
let rec sort l =
  match l with
  | [] → []
  | hd::tl → insert hd (sort tl)
```

Recursive Functions

Exercise 4: insertion sort

- 만약 같은 insertion sort를 C언어로 non-recursive하게 짠다면? 으으…

```
for (int c = 1; c ⩽ n - 1; c++) {
  int d = c;
  while (d > 0 && array[d] < array[d-1]) {
    int t = array[d];
    array[d] = array[d-1];
    array[d-1] = t;
    d--;
  }
}
```

Imperative vs Declarative (Functional)

- Insertion sort의 코드를 잘 보면 알겠지만, functional programming을 잘 활용하면 훨씬 직관적이고 간결한 표현이 가능하다!
- 이는 C를 포함한 많은 프로그래밍 언어가 imperative(명령적인) programming의 형태를 가지기 때문
  - Imperative programming? 이름에서 알 수 있듯, 나열된 statement 순서대로 실행하며 변수의 값을 바꾸는 등 컴퓨터에게 “어떻게 실행할지” 명령하는 형태
  - Declarative(Functional) programming: 위와 달리, “무엇을 계산할지” 선언하는 형태, 즉 expression을 잘 서술하기만 하면, 컴퓨터는 우리가 원하는 것을 내어준다!

Recursive Functions

Is Recursion Expensive?

- C, Java 등의 imperative 언어들은 주로 메모리의 상태를 변경하도록 명령하는 statement의 나열로 이루어져 있기 때문에, 내부적으로 recursion을 구현할 때 많은 메모리 공간을 필요로 한다
- 그러나, OCaml과 같은 declarative(functional) 언어들은 그 함수가 어떻게 정의되어 있으며, 이것의 계산 결과가 어떻게 되는지만 알면 되기 때문에 내부적으로 recursion이 굉장히 효율적으로 구현되어 있다
- 조금 더 엄밀하게는, tail-recursive의 형태를 띄는 함수들은 메모리 공간을 추가적으로 사용하지 않을 수 있다!

Recursive Functions

Tail-Recursive Functions

- 함수에서 recursive call이 연산의 마지막인 경우, 컴파일러가 call stack을 쌓을 필요 없이 선형적으로 처리할 수 있음
- 이게 대체 무슨 말?

Recursive Functions

Tail-Recursive Functions

- Non-tail-recursive function:

```
let rec factorial n =
  if n = 0 then 1
  else n * factorial (n - 1)
```

- Tail-recursive function:

```
let rec fact product counter maxcounter =
  if counter > maxcounter then product
  else fact (product * counter) (counter + 1) maxcounter
```

Higher-order Functions

Higher-Order Functions

Example 1: map

- 다음 3개의 함수는 하는 일이 비슷하다:

```
let rec inc_all l =
  match l with
  | [] → []
  | hd::tl → (hd + 1)::(inc_all tl)

let rec square_all l =
  match l with
  | [] → []
  | hd:tl → (hd * hd)::(square_all tl)

let rec cube_all l =
  match l with
  | [] → []
  | hd::tl → (hd * hd * hd)::(cube_all tl)
```

Higher-Order Functions

Example 1: map

- 우리는 higher-order function을 하나 만듦으로써 세 함수가 하는 일을 요약할 수 있다:

```
let rec map f l =
  match l with
  | [] → []
  | hd::tl → (f hd)::(map f tl)

let inc_all l = map (fun x → x + 1) l
let square_all l = map (fun x → x * x) l
let cube_all l = map (fun x → x * x * x) l
```

Higher-Order Functions

Example 2: filter

- 이번에는 다음과 같은 함수들을 생각해 보자:

```
let rec even l =
  match l with
  | [] → []
  | hd::tl →
    if hd mod 2 = 0 then hd::(even tl)
    else even tl

let rec greater_than_five l =
  match l with
  | [] → []
  | hd::tl →
    if hd > 5 then hd::(greater_than_five tl)
    else greater_than_five tl
```

Higher-Order Functions

Example 2: filter

- 역시나 higher-order function을 사용해 함수들을 요약할 수 있다:

```
let rec filter f l =
```

```
let even l =
let greater_than_five l =
```

Higher-Order Functions

Example 2: filter

- 역시나 higher-order function을 사용해 함수들을 요약할 수 있다:

```
let rec filter f l =
  match l with
  | [] → []
  | hd::tl →
    if f hd then hd::(filter f tl)
    else filter f tl
let even l = filter (fun x → x mod 2) l
let greater_than_five l = filter (fun x → x > 5) l
```

Higher-Order Functions

Example 2: filter

- 이런 higher-order function 역시 생각해 볼 수 있다:

```
let rec fold_right f l a =
  match l with
  | [] → a
  | hd::tl → f hd (fold_right f tl a)
```

Higher-Order Functions

Example 3: fold_right

- 이런 higher-order function 역시 생각해 볼 수 있다:

```
let rec fold_right f l a =
  match l with
  | [] → a
  | hd::tl → f hd (fold_right f tl a)
```

즉, 함수 f와 리스트 l(=[b1; b2; …; bn])과 초기값 a를 넘겨주면

a와 bn을 f한 뒤, 그 결과를 b(n-1)와 f하고, …, 그 결과를 b1과 f하는 함수

Higher-Order Functions

Example 3: fold_right

- 이런 higher-order function 역시 생각해 볼 수 있다:

```
let rec fold_right f l a =
  match l with
  | [] → a
  | hd::tl → f hd (fold_right f tl a)

let sum l = fold_right (fun elem acc → elem + acc) l 0
let prod l = fold_right (fun elem acc → elem * acc) l 1
```

Higher-Order Functions

Example 3: fold_right

- f와 a를 잘만 설계하면, 거의 모든 for문을 fold_right로 표현 가능:

```
let max_elem, sorted_lst =
  fold_right
    (fun elem (max, lst) →
     if elem > max then elem
     else (max, insert elem lst))
    [ 9; 1; 3; 7; 6; 4; 8; 2; 5 ]
    (-1, [])
```

Higher-Order Functions

fold_right vs fold_left

- fold_right는 리스트의 가장 오른쪽(끝)부터 계산해 최종적으로 가장 왼쪽(앞)을 계산
- fold_left는 리스트의 가장 왼쪽(앞)부터 계산해 최종적으로 가장 오른쪽(끝)을 계산:

```
fold_right f [x;y;z] init = f x (f y (f z init))
fold_left f init [x;y;z] = f (f (f init x) y) z
```

- fold_left가 tail-recursive하기 때문에 실제 구현에서는 fold_left 사용을 권장

Higher-Order Functions

Example 3: fold_right

- f와 a를 잘만 설계하면, 거의 모든 for문을 fold_right로 표현 가능:

```
let _, unique_lst =
  fold_right
    (fun elem (curr, lst) →
     if elem = curr then (curr, lst)
     else (elem, elem::lst))
    [ 1; 1; 2; 3; 3; 3; 4; 5; 6; 6; 6; 6; 6 ]
    (0, [])
```

Summary

- Functional Programming의 핵심과도 같은 recursion과 higher-order function을 이용해 끔찍한 imperative programming과 loop statement의 굴레에서 벗어나자!
  - Recursion을 잘 활용하면 문제를 더 간결하고 직관적으로 풀 수 있다
  - Higher-Order Function을 잘 활용하면 함수의 행동을 고차원적으로 요약할 수 있다