

How FP Deals With Effects

* PoolC 양제성

Source:  2024/01/24 (Wed)

목차

1st Session

1. 함수형 프로그래밍 Intro
 - Overall Structure
 - Historical Review (CS + Math)
2. 함수형 패러다임
 - Core of Functional Thinking
 - FP Fact-Checking
3. FP는 정말 순수한가?
 - Purity of Functions
 - File I/O Scenario

Basic Haskell Knowledge

2nd Session

1. 함수 합성을 위한 도구들
 - Partial Application
 - Kleisli Composition
2. ...중 하나인 모나드
 - Functor to Monad
 - IO Monad
3. 부수 효과의 관리
 - Action / Calculation / Data
 - Preventing Action Propagation

FP is all about **composing pure functions**.

```
int main(void) {  
    f(); g(); h(); ..  
}
```

[Procedural Programming]

VS

$f(g(h(\dots)))$

[Functional Programming]

Overall Structure

FP is all about **composing pure functions**.  How?

```
int main(void) {  
    f(); g(); h(); ..  
}
```

[Procedural Programming]

VS

$f(g(h(\dots)))$

[Functional Programming]

Overall Structure

Sum all. [stdin <- "5\n1 2 3 4 5"]

```
1 int main() { C++ (imperative)
2   int n, result;
3   std::cin >> n;
4   for (size_t i = 0; i < n; ++i) {
5     int a;
6     std::cin >> a;
7     result += a;
8   }
9   std::cout << result << '\n';
10  return 0;
11 }
```

[Procedural Programming]

```
1 main = Haskell (declarative)
2   interact
3     (show . sum .
4      map read . drop 1 . words)
```

```
1 Python3 (declarative)
2 from sys import stdin
3
4 print(sum(map(
5   int, stdin.read().split()[1:]
6 )))
```

[Functional Programming]

1. Purity

- Side Effect
- Referential Transparency
- Significance of ...

2. Immutability

- Recursion (feat. Tail Call Optimization)
- C vs Haskell in File IO

3. First Class Function

- Currying
- Linked List

1. Purity

- Side Effect
- Referential Transparency
- Significance of ...

2. Immutability

- Recursion (feat. Tail Call Optimization)
- C vs Haskell in File IO

3. First Class Function

- Currying
- Linked List

<Let 's
code!>

Historical Review (CS + Math)

함수형 프로그래밍 Intro

Lambda Calculus

1. Very Basics
2. Boolean in Action

Category Theory

1. Very Basics
2. Functor in Action

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application



Turing Machine

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application

⇔

Turing Machine

$$\lambda x. f x$$

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application

\Leftrightarrow

Turing Machine

$\lambda x. f x$

Lambda Abstraction

Py ver. `lambda x: f(x)` | JS ver. `(x) => f(x)`

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application

⇔

Turing Machine

Function Signifier ← $\lambda x. f x$
Lambda Abstraction

Py ver. `lambda x: f(x)` | JS ver. `(x) => f(x)`

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application

\Leftrightarrow

Turing Machine

Parameter Variable

Function Signifier \leftarrow

$\lambda x. f x$

Lambda Abstraction

Py ver. `lambda x: f(x)` | JS ver. `(x) => f(x)`

Lambda Calculus

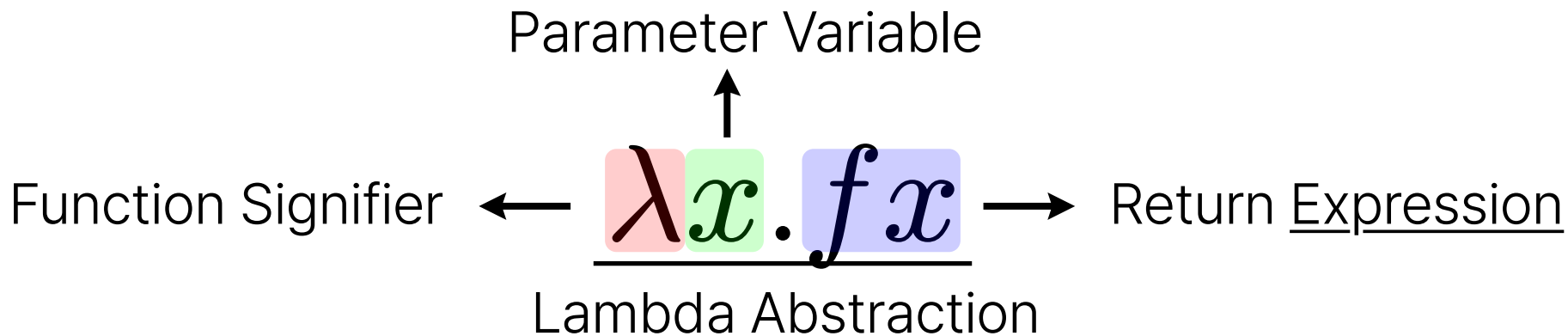
함수형 프로그래밍 Intro

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application

\Leftrightarrow

Turing Machine



Py ver. `lambda x: f(x)` | JS ver. `(x) => f(x)`

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application

\Leftrightarrow

Turing Machine

expression ::= variable

identifier

| expression expression

application

| $\lambda v_1 v_2 \cdots .$ expression

abstraction

| (expression)

grouping

β -reduction

$$\begin{aligned} & ((\lambda a.a) \lambda b.\lambda c.b)(x) \lambda e.f \\ &= (\lambda b.\lambda c.b)(x) \lambda e.f \\ &= (\lambda c.x) \lambda e.f \\ &= x \text{ } (\beta\text{-normal form}) \end{aligned}$$

Function Encoding

1. Variables (Immutable)
2. Functions (Curried)
3. Application



Turing Machine

ex) Church Encoding: Boolean 

"Mathematics is the art of giving
the **same name** to **different things**"

Henri Poincaré

Abstraction!

"Mathematics is the art of giving
the **same name** to **different things**"

Henri Poincaré

Abstraction of numbers

→

Elementary Algebra

Abstraction of relationships

→

Graph Theory

Abstraction of vectors and
their linear relationships

→

Linear Algebra

Abstraction of composition

→

Category Theory

Abstraction of numbers

→

Elementary Algebra

Abstraction of relationships

→

Graph Theory

Abstraction of vectors and
their linear relationships

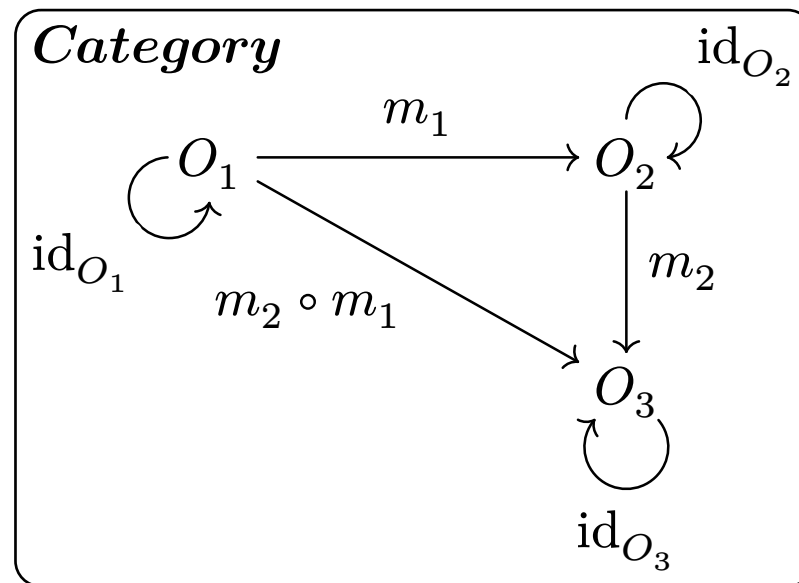
→

Linear Algebra

Category Theory

A **category** is a collection of...

Components
Objects
Morphisms (a.k.a. Arrows)
Composition of morphisms

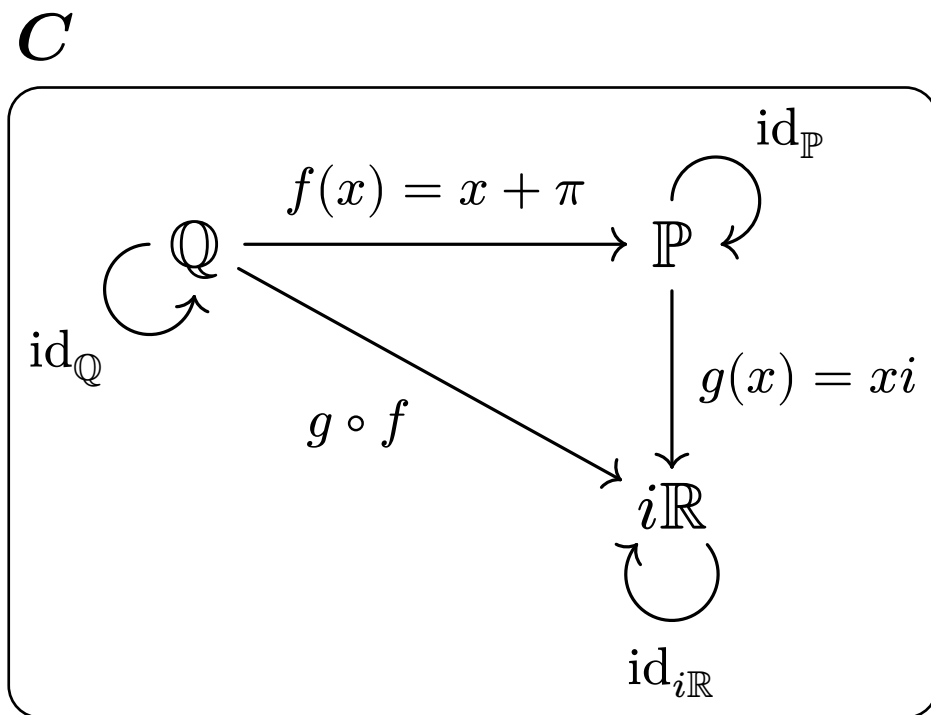


A **category** is a collection of...

Components	For example...
Objects	$\mathbb{Q}, \mathbb{P} = \mathbb{R} - \mathbb{Q}, i\mathbb{R} = \mathbb{C} - \mathbb{R}$
Morphisms (a.k.a. Arrows)	$f : \mathbb{Q} \rightarrow \mathbb{P}, g : \mathbb{P} \rightarrow i\mathbb{R}$
Composition of morphisms	$g \circ f : \mathbb{Q} \rightarrow i\mathbb{R}$

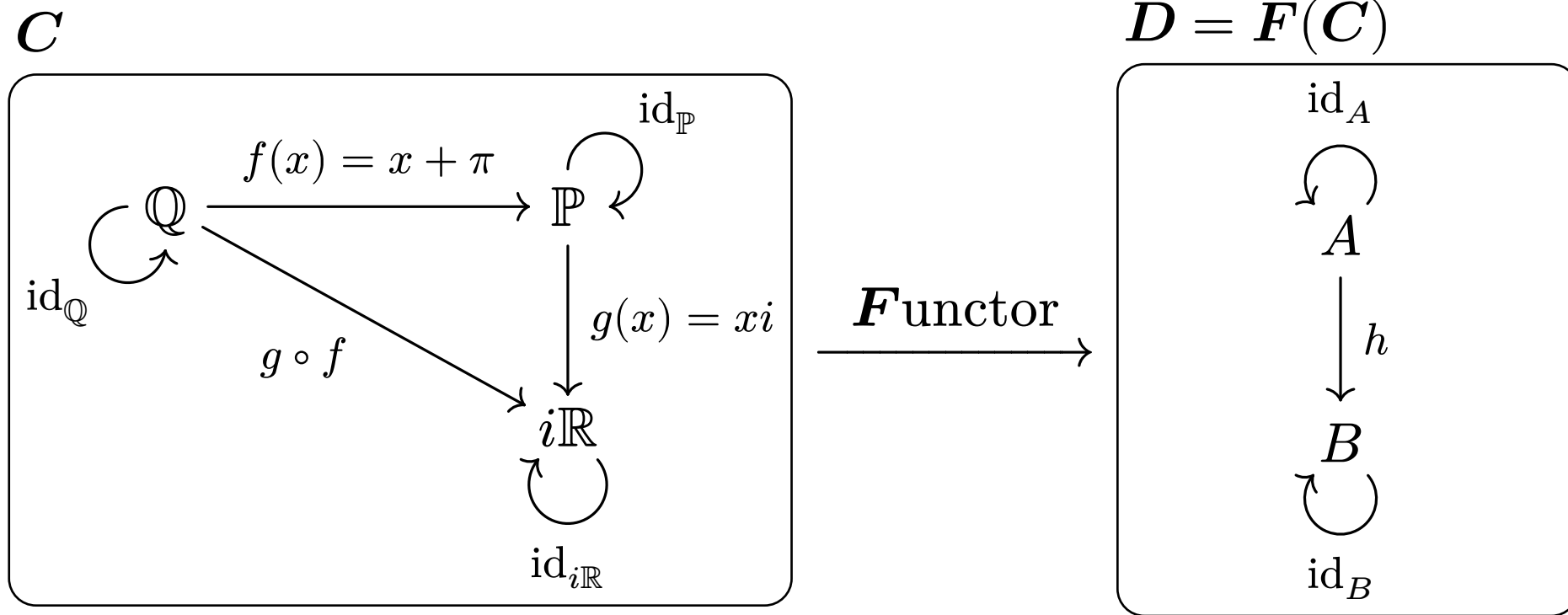
Category Theory

함수형 프로그래밍 Intro



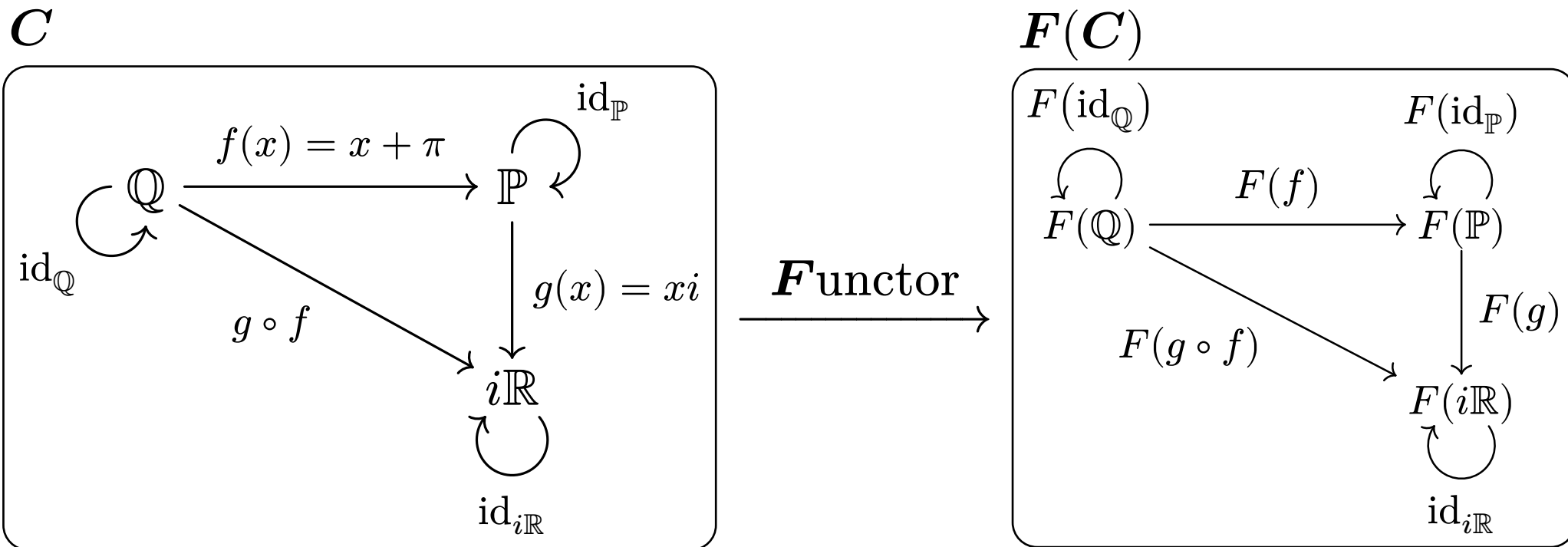
Category Theory

함수형 프로그래밍 Intro



Category Theory

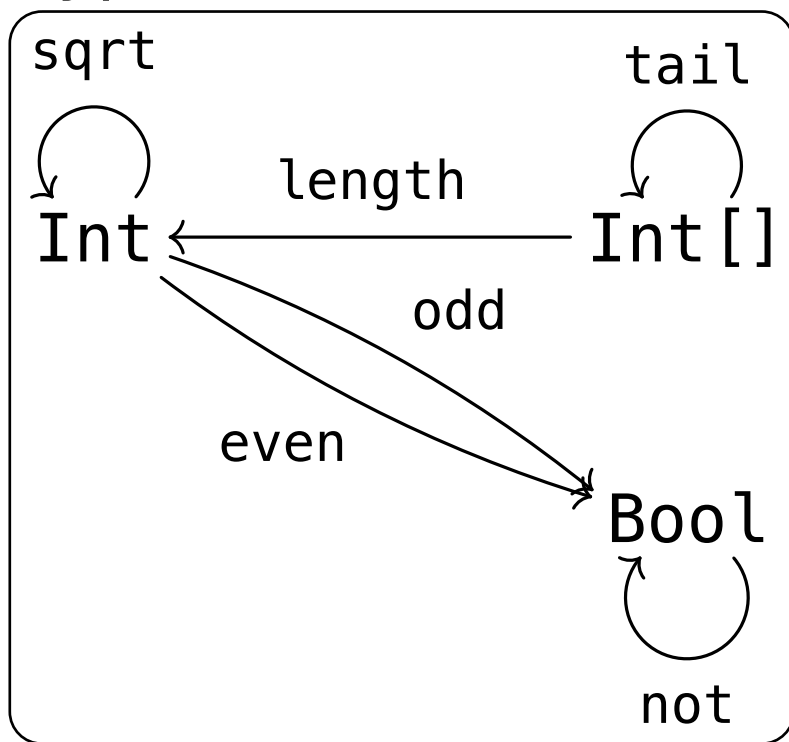
함수형 프로그래밍 Intro



Category Theory

함수형 프로그래밍 Intro

Type

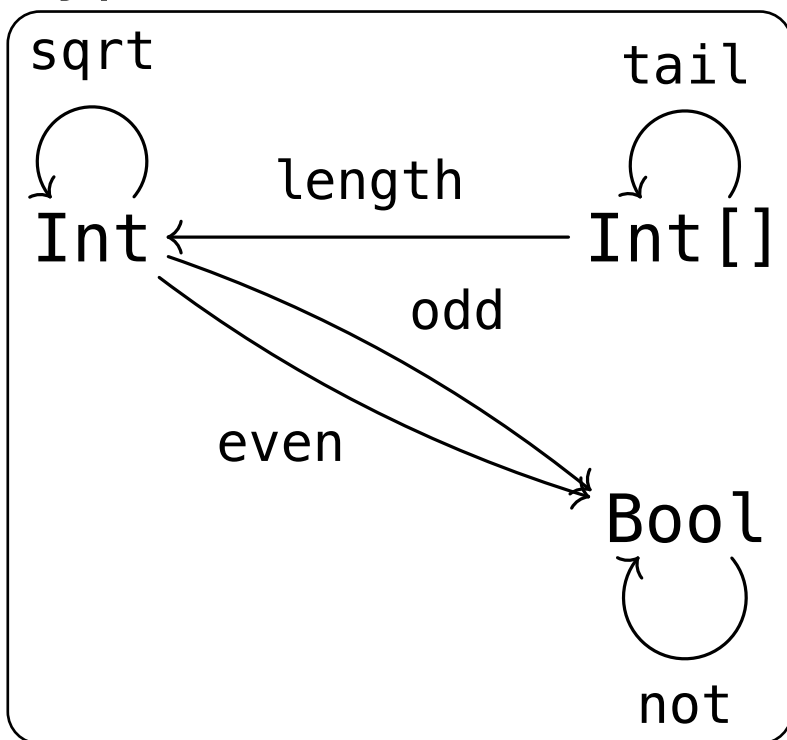


... let's ignore undefined situations

Category Theory

함수형 프로그래밍 Intro

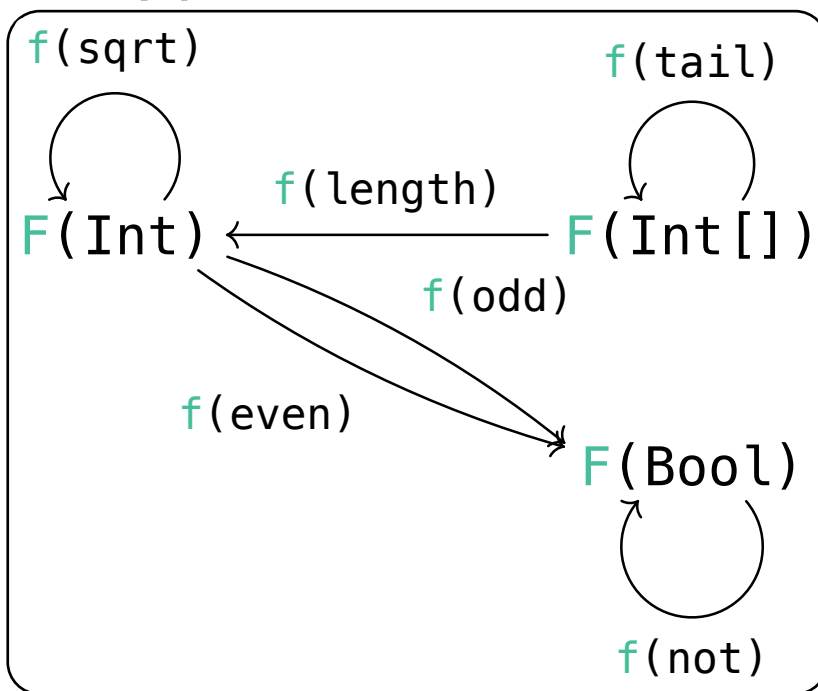
Type



... let's ignore undefined situations

Functor

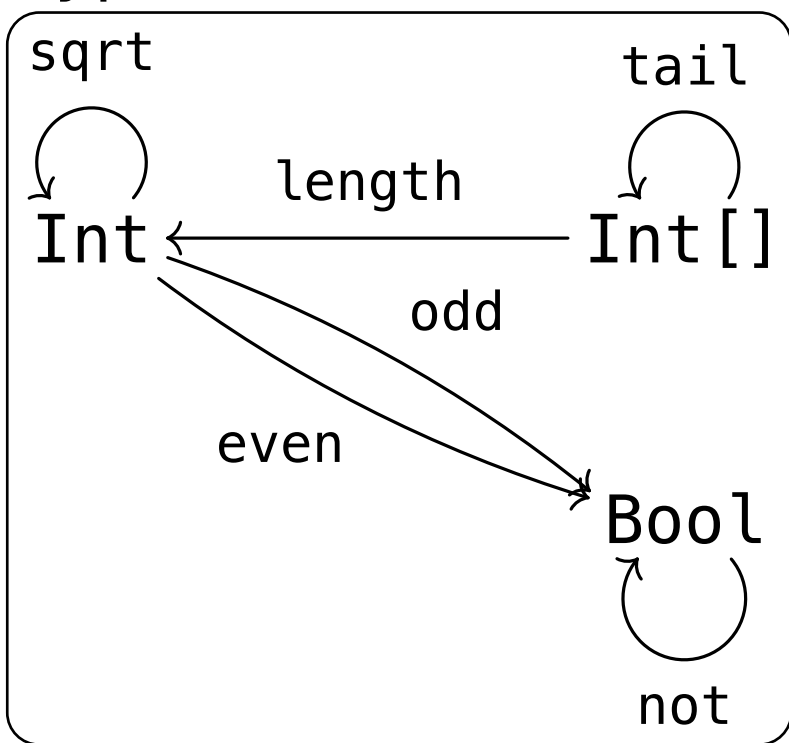
$F(\text{Type})$



Category Theory

함수형 프로그래밍 Intro

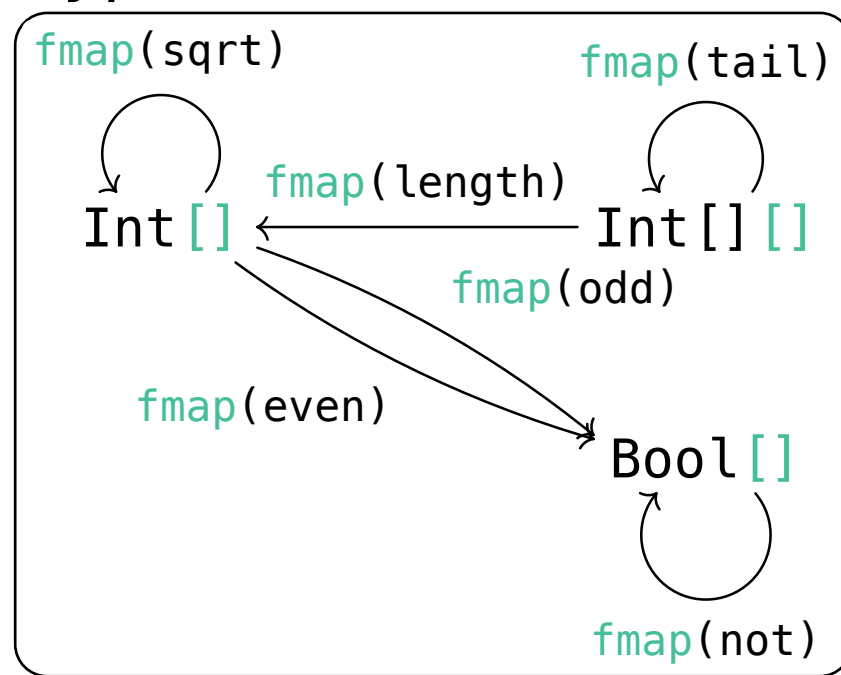
Type



... let's ignore undefined situations

[]

Type **[]**



Core of Functional Thinking

함수형 패러다임

Why do we make softwares?

Core of Functional Thinking

함수형 패러다임

Why do we make softwares?

To **use** them and gain benefits from the **output**.

Core of Functional Thinking

함수형 패러다임

Why do we make softwares?

To **use** them and gain benefits from the **output**.

We DO need some **interactions** with the outside world!

Core of Functional Thinking

함수형 패러다임

Why do we make softwares?

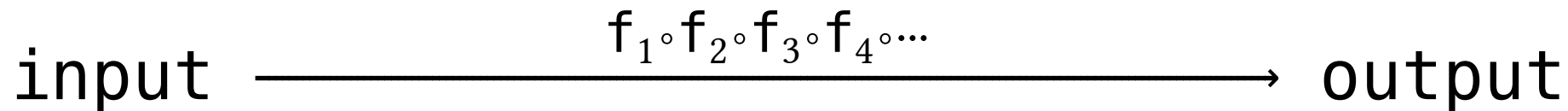
To **use** them and gain benefits from the **output**.

We DO need some **interactions** with the outside world!
== Side Effect!

Core of Functional Thinking

함수형 패러다임

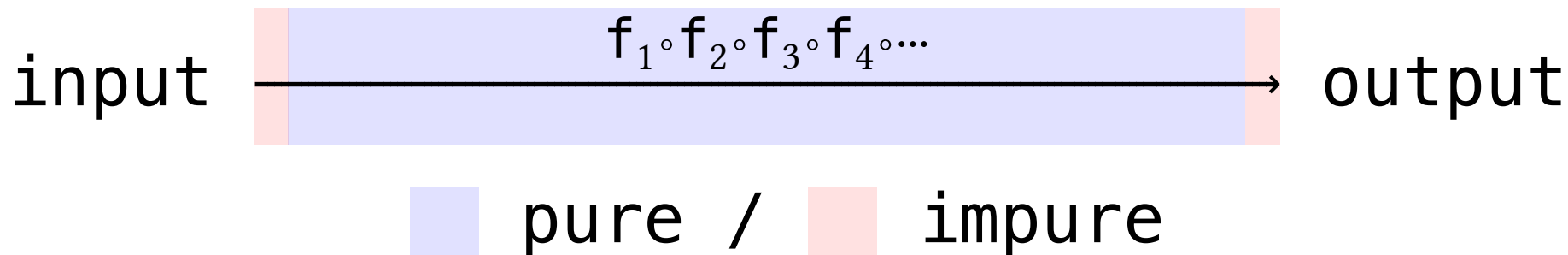
Our Program



Core of Functional Thinking

함수형 패러다임

Our Program



FP Fact-Checking

함수형 패러다임

1. Easy Testing
 2. Better Predictability
 3. Fewer Bugs
 4. Fearless Concurrency
- Bonus. Being Declarative

FP Fact-Checking

함수형 패러다임

Easy Testing

Easy Testing

- True for **pure functions**.
- Still need mocking stuffs to test impure interactions.
i.e. Network IO

FP Fact-Checking

함수형 패러다임

Better Predictability

Better Predictability

- True for **pure functions**.
- So the overall predictability **may** increase.
- Impure interactions could be non-deterministic.
i.e. Concurrent Threads

FP Fact-Checking

함수형 패러다임

Fewer Bugs

Fewer Bugs

- True for **pure functions with tests**.
- Even pure functions need testing; **trust isn't automatic**.

Fearless Concurrency

Fearless Concurrency

- True for **pure functions**.
- Concurrency control mechanisms should definitely be utilized when needed!

Additionally... Being Declarative

- True.
- However, being declarative is **not always superior**.
- Testing your declarative APIs is also essential.