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?
   - Optimizing with Purity
   - Effect Handling Basics

`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

Overall Structure

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

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

[Procedural Promramming]

VS

```
f(g(h(..)))
```

[Functional Programming]

Overall Structure

How?

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

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

[Procedural Promramming]

VS

```
f(g(h(..)))
```

[Functional Programming]

Overall Structure

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

C++ (imperative)

```cpp
int main() {
  int n, result;
  std::cin >> n;
  for (size_t i = 0; i < n; ++i) {
    int a;
    std::cin >> a;
    result += a;
  }
  std::cout << result << '\n';
  return 0;
}
```

[Procedural Promramming]

Haskell (declarative)

```haskell
main =
  interact
    (show . sum .
      map read . drop 1 . words)
```

Python3 (declarative)

```python

from sys import stdin

print(sum(map(
  int, stdin.read().split()[1:]
)))
```

[Functional Programming]

Overall Structure

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

Overall Structure

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)

Lambda Calculus

1. Very Basics
2. Boolean in Action

Category Theory

1. Very Basics
2. Functor in Action

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

$$\lambda x.fx$$

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

$$\underline{\lambda x.fx}$$

Lambda Abstraction

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

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

Function Signifier $\longleftarrow$ $\underbrace{\boxed{\lambda} x.fx}_{\text{Lambda Abstraction}}$

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

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

Parameter Variable

Function Signifier $\longleftarrow$ $\underline{\lambda x . f x}$

Lambda Abstraction

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

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

Parameter Variable

Function Signifier $\leftarrow$ $\underbrace{\lambda x . f x}_{\text{Lambda Abstraction}}$ $\rightarrow$ Return Expression

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

Lambda Calculus

Function Encoding

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

$\Leftrightarrow$ `Turing Machine`

$$
\begin{aligned}
\text{expression} ::= {} & \text{variable} && \textit{identifier} \\
\mid {} & \text{expression expression} && \textit{application} \\
\mid {} & \lambda\ v_1 v_2 \cdots\ .\ \text{expression} && \textit{abstraction} \\
\mid {} & (\ \text{expression}\ ) && \textit{grouping}
\end{aligned}
$$

Lambda Calculus

β-reduction

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

Lambda Calculus

Function Encoding

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

⇔ `Turing Machine`

`ex) Church Encoding: Boolean` JS

Category Theory

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

Henri Poincaré

Category Theory

Abstraction!

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

Henri Poincaré

Category Theory

Abstraction of numbers	→	Elementry Algebra
Abstraction of relationships	→	Graph Theory
Abstraction of vectors and their linear relationships	→	Linear Algebra

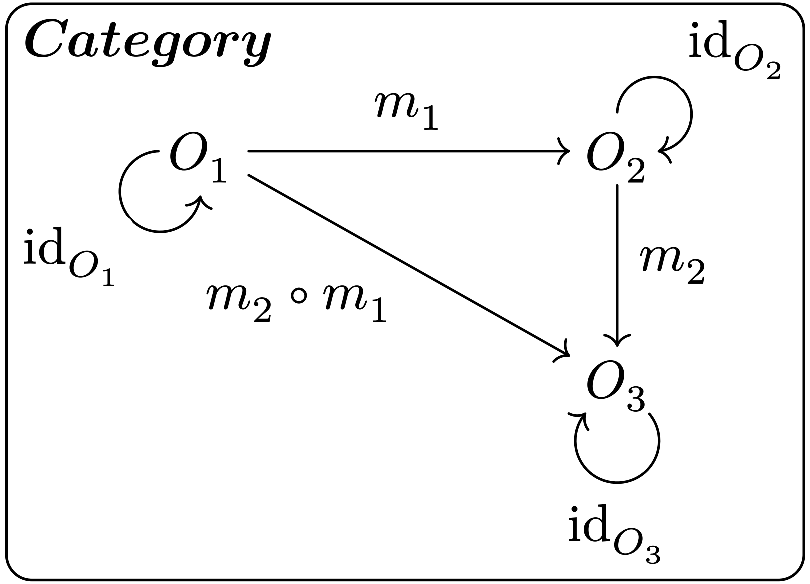
Category Theory

Abstraction of composition	→	**Category Theory**
Abstraction of numbers	→	Elementry 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



Category Theory

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

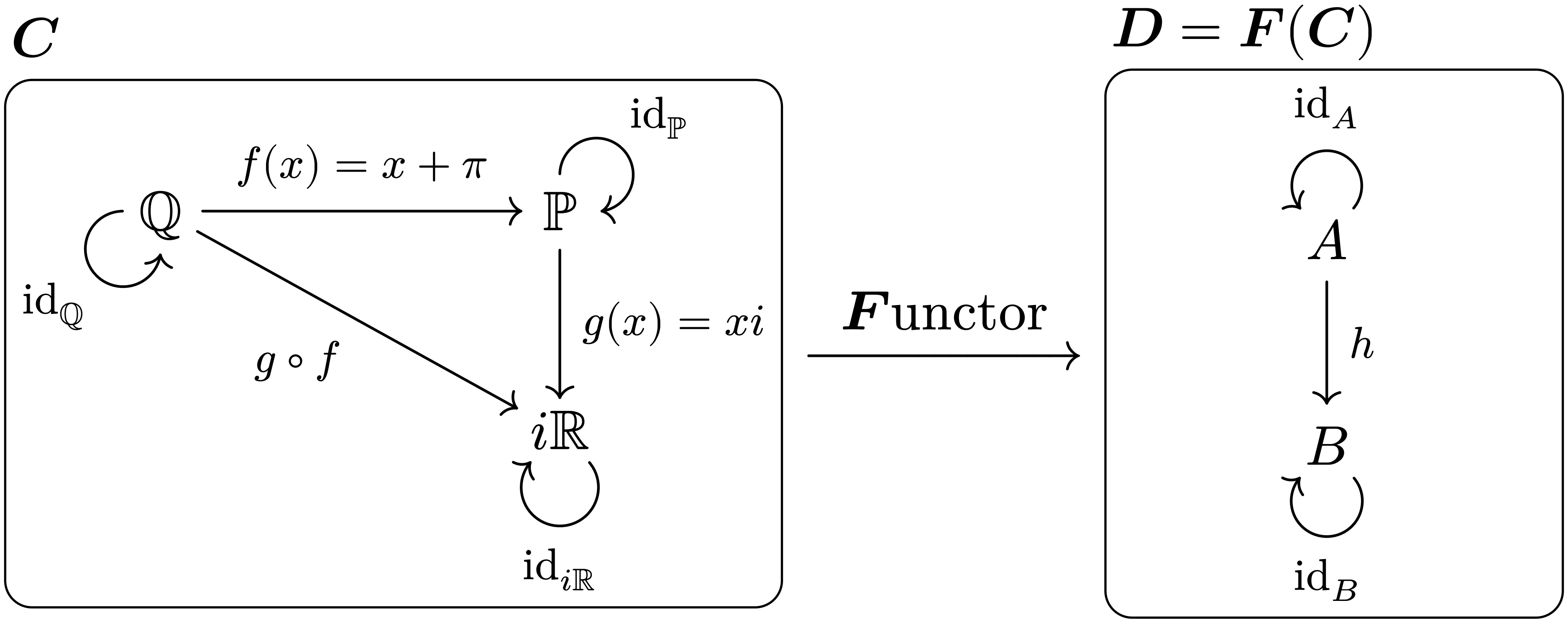
Components	For example...
Objects	$\mathbb{Q},\ \mathbb{P} = \mathbb{R} - \mathbb{Q},\ i\mathbb{R} = \mathbb{C} - \mathbb{R}$
Morphisms (a.k.a. Arrows)	$f : \mathbb{Q} \to \mathbb{P},\ g : \mathbb{P} \to i\mathbb{R}$
Composition of morphisms	$g \circ f : \mathbb{Q} \to i\mathbb{R}$

Category Theory

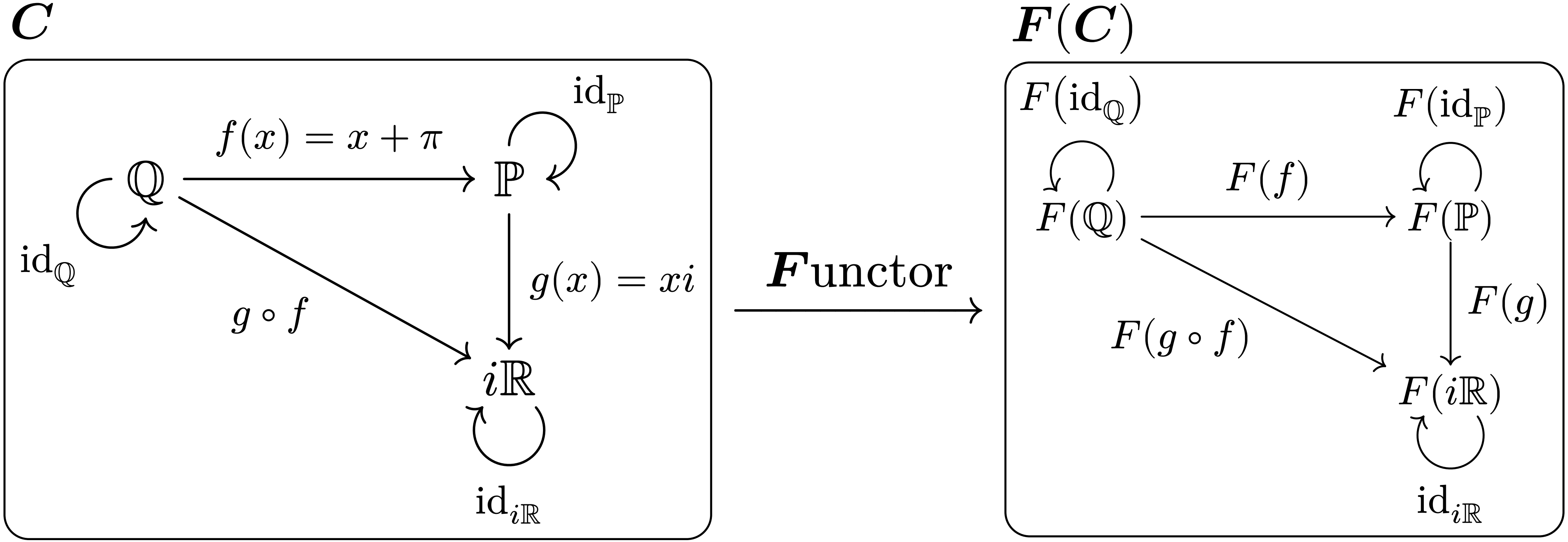
$\boldsymbol{C}$

$$
\begin{tikzcd}
\mathbb{Q} \arrow[r, "f(x) = x + \pi"] \arrow[dr, "g \circ f"'] \arrow[loop left, "\mathrm{id}_{\mathbb{Q}}"'] & \mathbb{P} \arrow[d, "g(x) = xi"] \arrow[loop above, "\mathrm{id}_{\mathbb{P}}"] \\
& i\mathbb{R} \arrow[loop below, "\mathrm{id}_{i\mathbb{R}}"]
\end{tikzcd}
$$

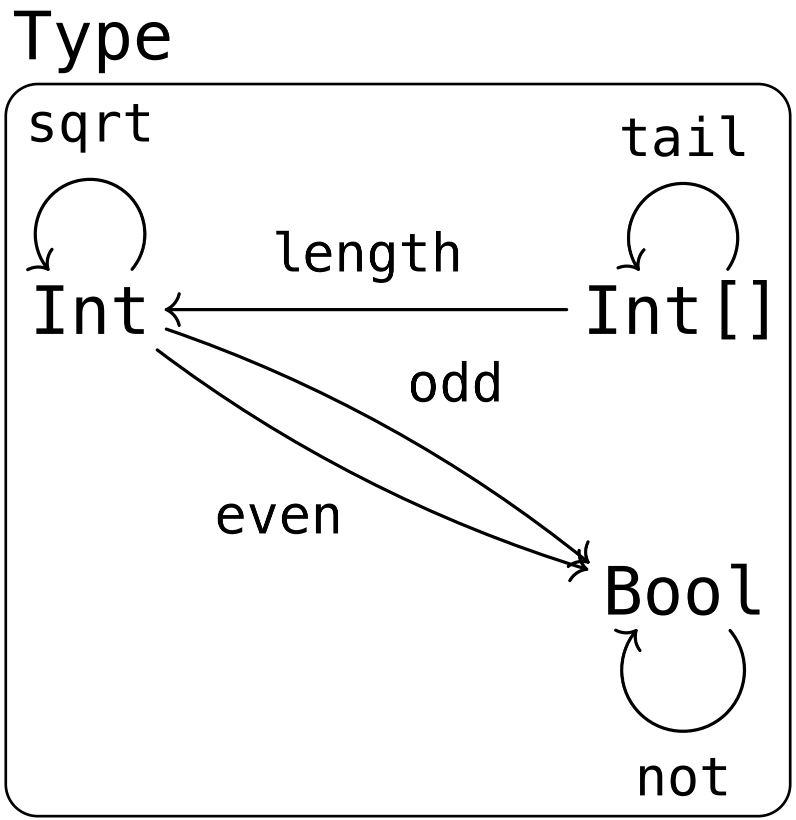
Category Theory



Category Theory

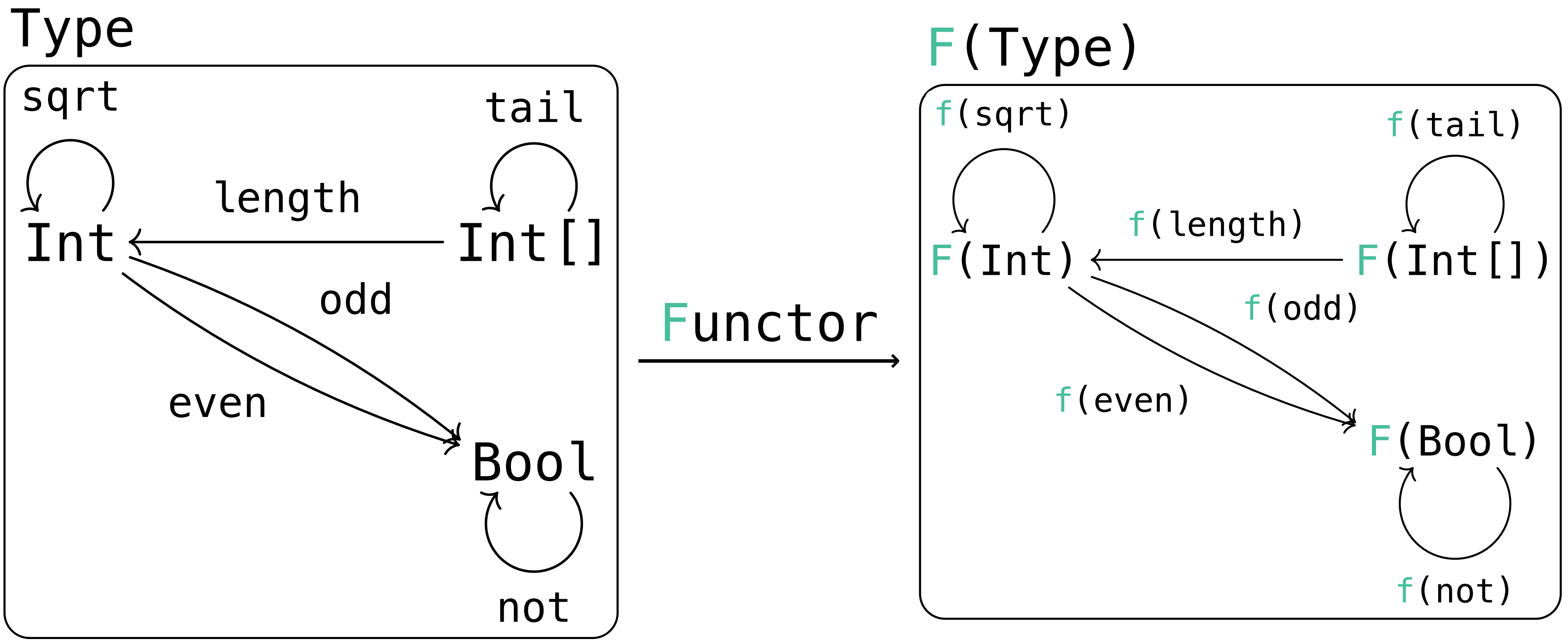


Category Theory



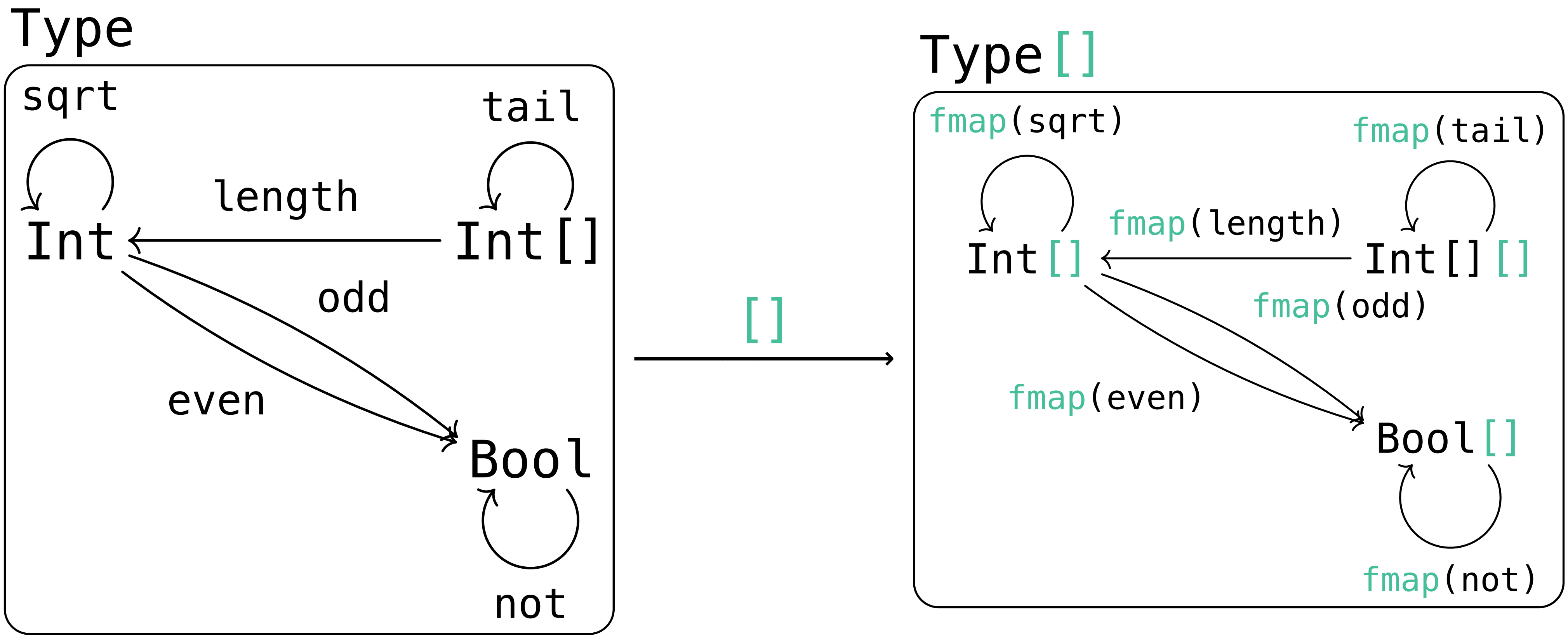
... let's ignore undefined situations

Category Theory



... let's ignore undefined situations

Category Theory



... let's ignore undefined situations

Core of Functional Thinking

Why do we make softwares?

Core of Functional Thinking

Why do we make softwares?

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

Core of Functional Thinking

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

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!`

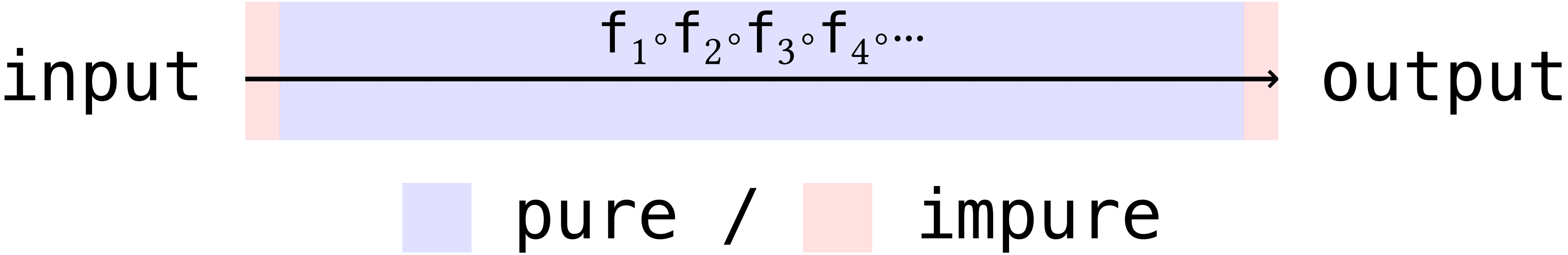
Core of Functional Thinking

Our Program

$$\text{input} \xrightarrow{f_1 \circ f_2 \circ f_3 \circ f_4 \circ \cdots} \text{output}$$

Core of Functional Thinking

Our Program



FP Fact-Checking

1. Easy Testing
2. Better Predictability
3. Fewer Bugs
4. Fearless Concurrency

Bonus. Being Declarative

FP Fact-Checking

Easy Testing

FP Fact-Checking

Easy Testing

- True for **pure functions**.
- Still need mocking stuffs to test <u>impure interactions</u>.

  `i.e. Network IO`

FP Fact-Checking

Better Predictability

FP Fact-Checking

Better Predictability

- True for **pure functions**.
- So the overall predictability **may** increase.
- Impure interactions could be <u>non-deterministic</u>.

  `i.e. Concurrent Threads`

FP Fact-Checking

Fewer Bugs

FP Fact-Checking

Fewer Bugs

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

FP Fact-Checking

Fearless Concurrency

FP Fact-Checking

Fearless Concurrency

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

FP Fact-Checking

Additionally... Being Declarative

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

Optimizing with Purity

Pure functions...

1. do not have side effects
2. exhibit referential transparency

Optimizing with Purity

Pure functions...

1. do not have side effects
2. exhibit referential transparency

Let's utilize these properties for optimization!

Optimizing with Purity

Let's assume that we are designing a purely functional language.

– How can our compiler optimize this expression?

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

```
sinh x = ((exp x) - (1 / (exp x))) / 2
```

Optimizing with Purity

$$\sinh x = \frac{e^x - e^{-x}}{2}$$

```
sinh x = ((exp x) - (1 / (exp x))) / 2
```

↓

```
sinh x = (t - (1 / t)) / 2 where t = exp x
```

Optimizing with Purity

This is the power of purely functional language!

cf. In C, this kind of optimization can't be done offensively. Why?

Effect Handling Basics

...But what about effects?

Haskell

```haskell
main = do
  firstName <- getLine
  secondName <- getLine -- called twice with same param
  putStrLn ("Hi, " ++ firstName ++ secondName)
  putStrLn "--------"
  putStrLn "--------" -- called twice with same param
  putStrLn "Today's weather: ..."
```

Effect Handling Basics

Let's assume that you're doing some simulations.

- ...in a purely functional language.
- You need to manage various <u>states</u> of objects.

`i.e. Position`

Effect Handling Basics

<Let's code!>

Let's assume that you're doing some simulations.

- ...in a purely functional language.
- You need to manage various <u>states</u> of objects.

`i.e. Position`

Effect Handling Basics

Interesting example

Haskell

```haskell
main :: IO ()
main = head [print "hi", print "hello", print "whatever"]
```

Python3

```python
def main():
    return [print("hi"), print("hello"), print("whatever")][0]
main()
```