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. **SW 엔지니어링의 목표**
   - SW Maintainability
   - FP vs OOP vs PP

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

Overall Structure

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

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

VS

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

[Procedural Promramming]

[Functional Programming]

Overall Structure

How?

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

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

VS

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

[Procedural Promramming]

[Functional Programming]

Overall Structure

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

```cpp
                                C++ (imperative)
1  int main() {
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 }
```

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

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

[Procedural Promramming]

[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

$$\Leftrightarrow \quad \texttt{Turing Machine}$$

Lambda Calculus

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

$$\Leftrightarrow \quad \texttt{Turing Machine}$$

$$\lambda x.fx$$

Lambda Calculus

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

$\Leftrightarrow$ Turing Machine

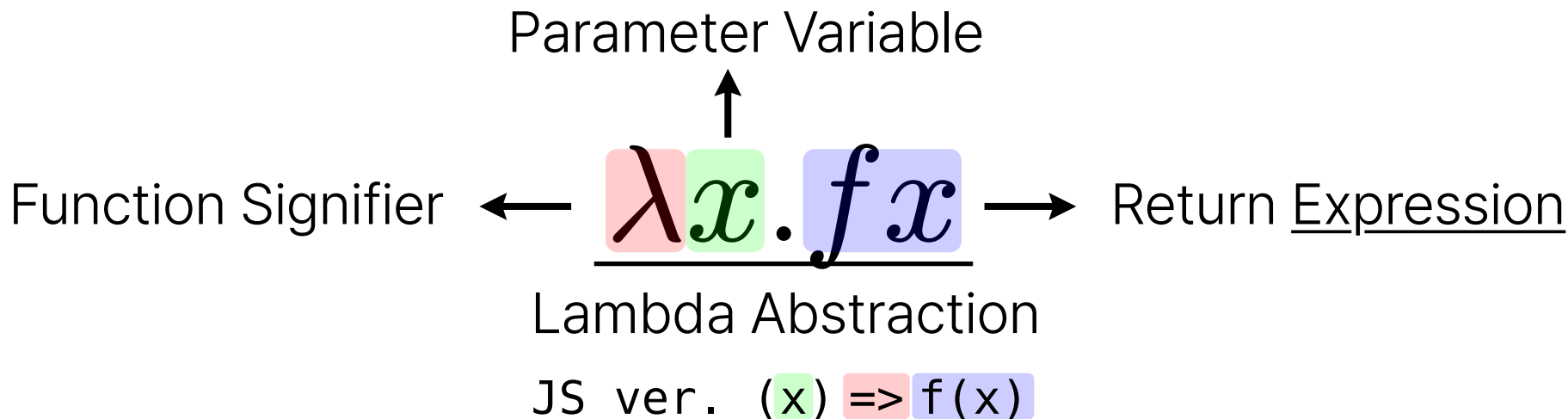$$\lambda x.fx$$

Lambda Abstraction

JS ver. (x) => f(x)

Lambda Calculus

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

$\Leftrightarrow$ Turing Machine

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

Lambda Abstraction

JS ver. (x) => f(x)

Lambda Calculus

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

$\Leftrightarrow$ Turing Machine

Parameter Variable

Function Signifier ← $\lambda x . f x$

Lambda Abstraction

JS ver. `(x) => f(x)`

Lambda Calculus

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

$\Leftrightarrow$ `Turing Machine`

Parameter Variable

Function Signifier $\longleftarrow$ $\lambda x . f x$ $\longrightarrow$ Return Expression

Lambda Abstraction

`JS ver. (x) => f(x)`

Lambda Calculus

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

$\Leftrightarrow$ `Turing Machine`

$$\text{expression} ::= \text{variable} \qquad\qquad\quad identifier$$
$$\mid \text{expression expression} \quad application$$
$$\mid \lambda \; v_1 v_2 \cdots . \; \text{expression} \quad abstraction$$
$$\mid (\; \text{expression} \;) \qquad\qquad\quad grouping$$

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 $\rightarrow$ **Category Theory**

Abstraction of numbers $\rightarrow$ Elementry Algebra

Abstraction of relationships $\rightarrow$ Graph Theory

Abstraction of vectors and their linear relationships $\rightarrow$ Linear Algebra

Category Theory

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

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

Category

$$\text{id}_{O_1} \quad O_1 \xrightarrow{m_1} O_2 \quad \text{id}_{O_2}$$

$$m_2 \circ m_1 \qquad m_2$$

$$O_3 \quad \text{id}_{O_3}$$
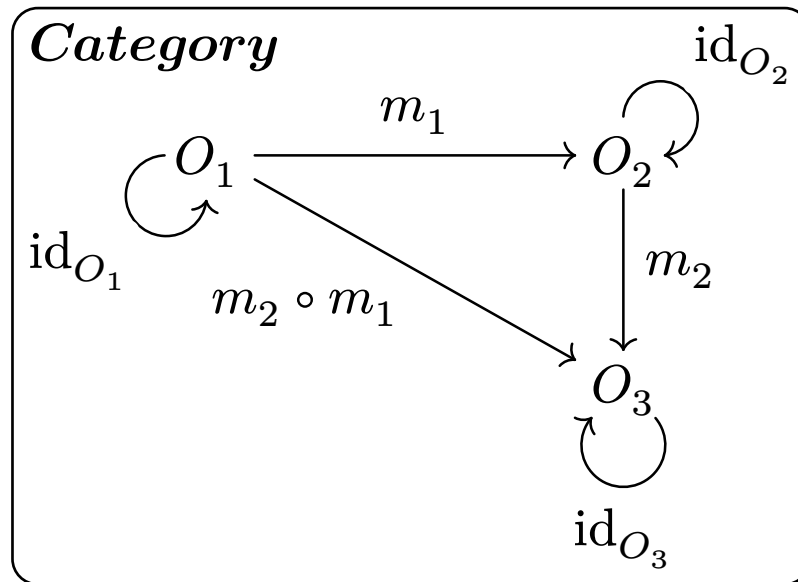
Category Theory

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
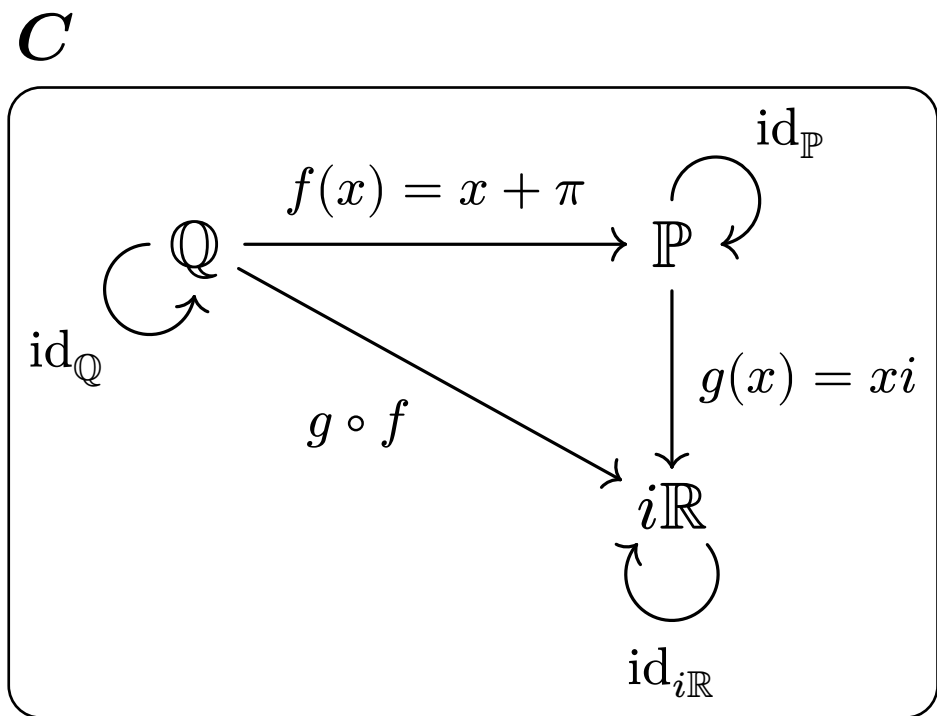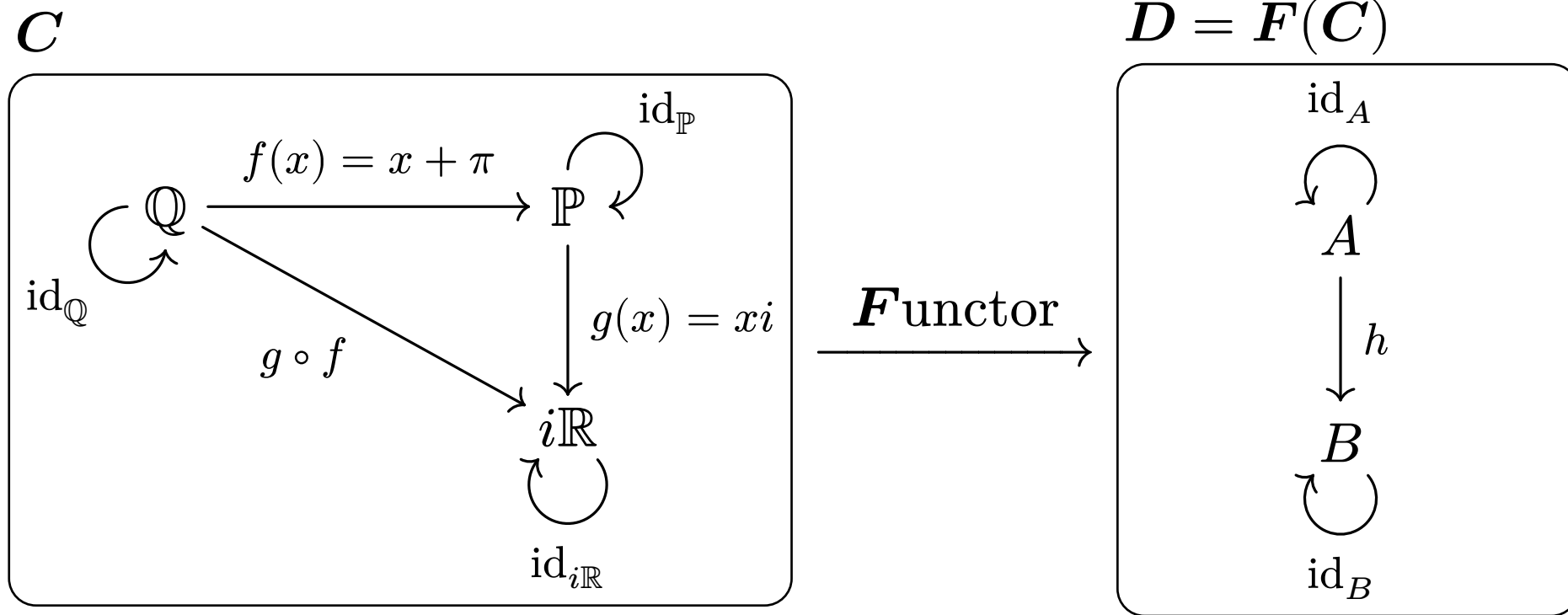
C
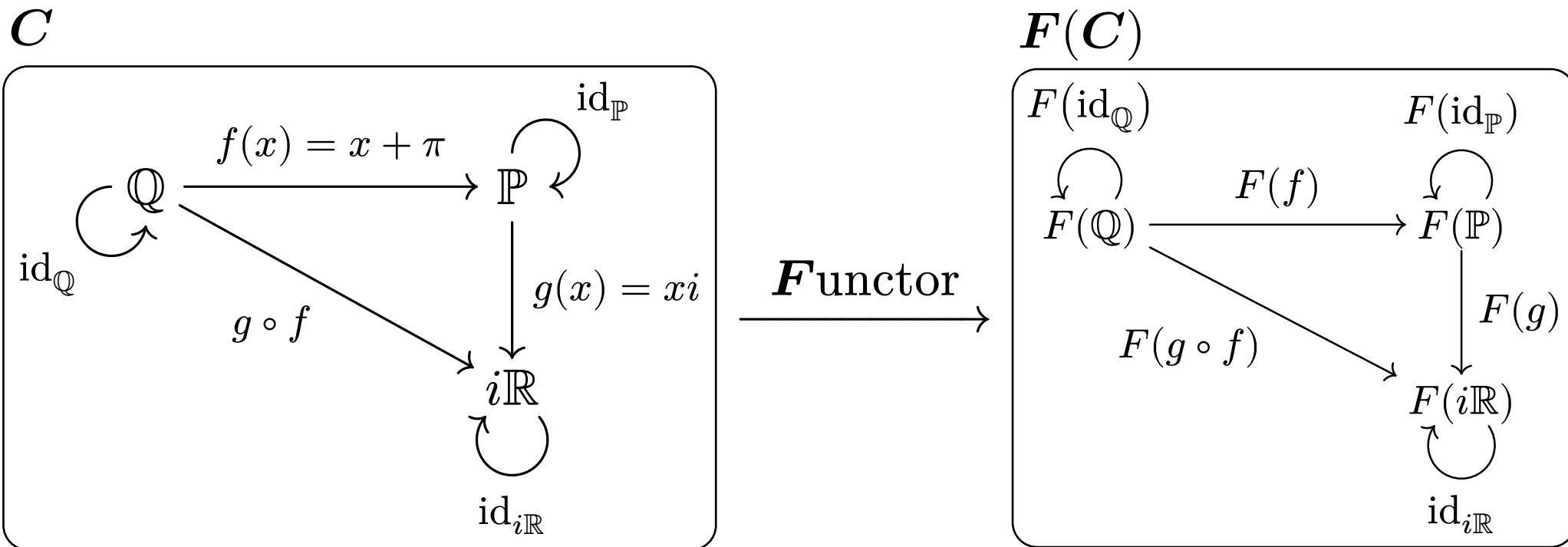
Category Theory

Category Theory
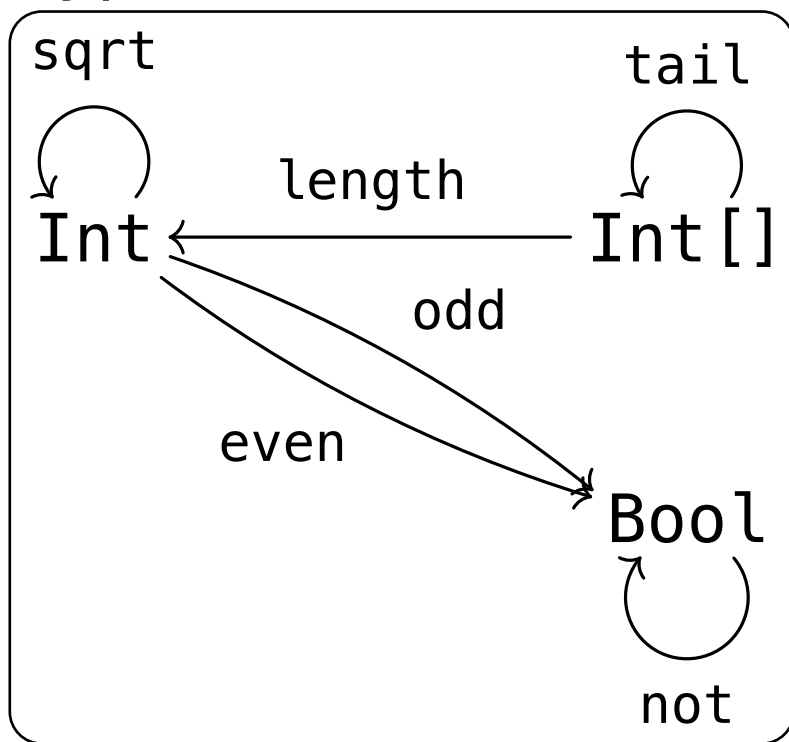
Category Theory

... let's ignore undefined situations

Category Theory

Type

sqrt
tail
length
Int
Int[]
odd
even
Bool
not
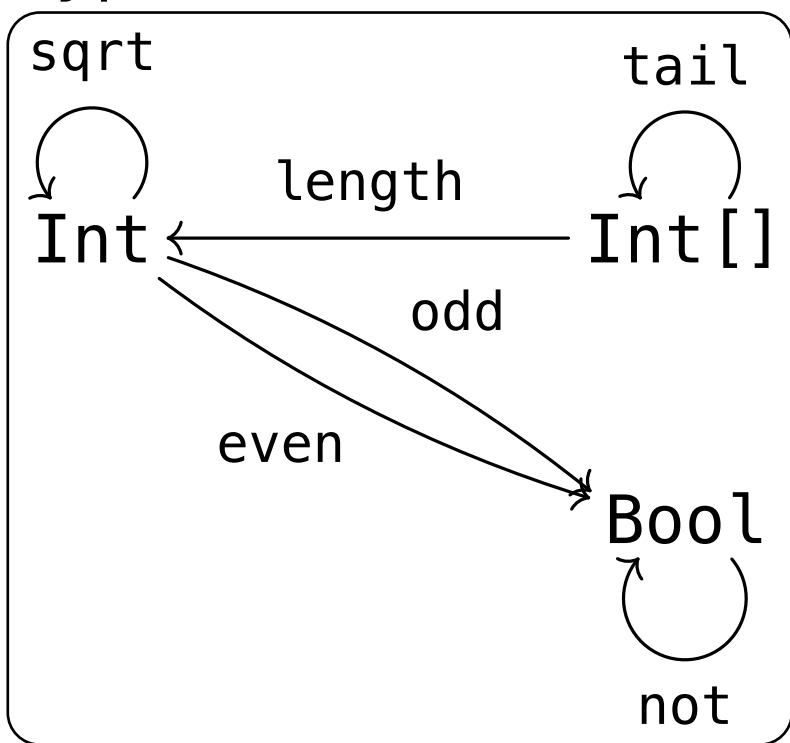
... let's ignore undefined situations

Functor

F(Type)

f(sqrt)
f(tail)
f(length)
F(Int)
F(Int[])
f(odd)
f(even)
F(Bool)
f(not)

Category Theory

... let's ignore undefined situations