귀납적 데이터 타입

카테고리에서 자연수 정의하기

문순원

Algebraic data type

대수적 데이터 타입(ADT)

- 타입들을 조합하여 새로운 타입을 만들어냄
- 정의하고자 하는 데이터 타입을 보다 정밀하게 표현할 수 있다
- 귀납적으로 정의되는 타입도 있음

Algebraic data type

대수적 데이터 타입(ADT)

- 타입들을 조합하여 새로운 타입을 만들어냄
- 정의하고자 하는 데이터 타입을 보다 정밀하게 표현할 수 있다
- 귀납적으로 정의되는 타입도 있음

ADT의 기본 구성요소

- Product type
- Sum type
- Unit type
- ...

Basic types - Product and Sum

Product

- Cartesian product
- Tuple
- Record
- Conjunction

```haskell
data Prod a b = MkProd a b
>> MkProd 'A' 10 :: Prod Char Int
```

$\mathrm{Prod}(A, B) = A \times B$

Sum

- Disjoint union
- Coproduct
- Tagged union
- Disjunction

```haskell
data Sum a b = MkSum0 a | MkSum1 b
>> MkSum0 'A' :: Sum Char Int
>> MkSum1 10 :: Sum Char Int
```

$\mathrm{Sum}(A, B) = A \sqcup B$

Basic types - Unit and Bottom

Unit

- 1-type
- 0-tuple
- Terminal object

```
data Unit = MkUnit
>> MkUnit :: Unit
```

$\text{Unit} = \{1\}$

Bottom

- 0-type
- Empty set
- Initial object

```
data Bottom
-- Bottom 타입은 원소를 가지지 않음
```

$\text{Bottom} = \emptyset$

Example - Bool

Sum과 Unit을 사용해 Bool을 정의할 수 있다

```
type Bool = Sum Unit Unit
>> MkSum0 MkUnit :: Bool
>> MkSum1 MkUnit :: Bool
```

Example - Bool

Sum과 Unit을 사용해 Bool을 정의할 수 있다

```
type Bool = Sum Unit Unit
>> MkSum0 MkUnit :: Bool
>> MkSum1 MkUnit :: Bool
```

...하지만 실제론 이렇게 쓴다

```
data Bool = True | False
>> True :: Bool
>> False :: Bool
```

Example - Bool

Sum과 Unit을 사용해 Bool을 정의할 수 있다

```
type Bool = Sum Unit Unit
>> MkSum0 MkUnit :: Bool
>> MkSum1 MkUnit :: Bool
```

…하지만 실제론 이렇게 쓴다

```
data Bool = True | False
>> True :: Bool
>> False :: Bool
```

∴ 그냥 human-readable 하게 써도 알아서 잘 바꿔줌

Inductive data types

Product, Sum 타입만으로는 다룰 수 있는 대상이 너무 제한적임

Inductive data types

Product, Sum 타입만으로는 다룰 수 있는 대상이 너무 제한적임

재귀를 하면 임의의 크기를 가진 데이터도 표현 가능

```
data List a = Nil | Cons a (List a)
>> Nil :: List Int
>> Cons 0 Nil :: List Int
>> Cons 1 (Cons 0 Nil) :: List Int
>> Cons 2 (Cons 1 (Cons 0 Nil)) :: List Int
>> ...
```

Inductive data types...

```
data Nat = Zero | Succ Nat
>> Zero                  -- 0
>> Suc Zero              -- 1
>> Suc (Suc Zero)        -- 2
>> Suc (Suc (Suc Zero))  -- 3
>> ...

data Tree a = Nil | Branch a (Tree a) (Tree a)
>> Branch 1 (Branch 0 Nil Nil) (Branch 2 Nil Nil)
--      1
--    /   \
--   0     2
--  / \   / \
-- N   N N   N
```

Haskell’s Functor

Maybe 타입은 다음과 같이 정의된다

```haskell
-- type Maybe a = Sum Unit a
data Maybe a = Nothing | Just a
>> Nothing :: Maybe Int
>> Just 10 :: Maybe Int
```

Haskell’s Functor

Maybe 타입은 다음과 같이 정의된다

```haskell
-- type Maybe a = Sum Unit a
data Maybe a = Nothing | Just a
>> Nothing :: Maybe Int
>> Just 10 :: Maybe Int
```

Maybe처럼 타입변수를 인자로 받을 수 있는 타입이 특수한 조건을 만족하면 **Functor**라고 부른다
Product와 Sum만으로 이루어진 타입은 반드시 이 조건을 만족

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

fmap은 리스트에서 흔히 사용하는 map함수와 비슷한 기능을 함

Inductive types as a fixpoint of functor

Inductive type은 functor의 fixpoint로 다룰 수 있다

Inductive types as a fixpoint of functor

Inductive type은 functor의 fixpoint로 다룰 수 있다

$$X = \mathrm{Fix}(F) \Leftrightarrow X \approx F(X)$$

Inductive types as a fixpoint of functor

Inductive type은 functor의 fixpoint로 다룰 수 있다

$$X = \mathrm{Fix}(F) \Leftrightarrow X \approx F(X)$$

```haskell
data Fix f = InF (f (Fix f))

data ListF a b = ConsF a b | NilF
type List a = Fix (ListF a)

data Maybe a = Just a | Nothing
type Nat = Fix Maybe

data TreeF a b = BranchF a b b | NilF
type Tree a = Fix (TreeF a)
```

Inductive types as a fixpoint of functor

Inductive type은 functor의 fixpoint로 다룰 수 있다

$$X = \mathrm{Fix}(F) \Leftrightarrow X \approx F(X)$$

```haskell
data Fix f = InF (f (Fix f))

data ListF a b = ConsF a b | NilF
type List a = Fix (ListF a)

data Maybe a = Just a | Nothing
type Nat = Fix Maybe

data TreeF a b = BranchF a b b | NilF
type Tree a = Fix (TreeF a)
```

Functor가 주어졌을 때 fixpoint를 어떻게 찾을수 있을까?

지금부터 카테고리 이야기함 아 ㅋㅋ

Category

카테고리 C란

- collection of object : $\mathrm{ob}(C)$
- collection of morphism : $\hom_C(X, Y)$ where $X, Y \in \mathrm{ob}(C)$
- composition : $\circ : \hom_C(Y, Z) \times \hom_C(X, Y) \to \hom_C(X, Z)$

Category

카테고리 C란

- collection of object : $\mathrm{ob}(C)$
- collection of morphism : $\hom_C(X, Y)$ where $X, Y \in \mathrm{ob}(C)$
- composition : $\circ : \hom_C(Y, Z) \times \hom_C(X, Y) \to \hom_C(X, Z)$

또한 카테고리법칙을 만족해야 한다

- identity morphism의 존재 : $\mathrm{id}_X : X \to X$ where $X \in C$

$$\mathrm{id}_Y \circ f = f \circ \mathrm{id}_X = f \text{ where } f : X \to Y$$

- $\circ$의 associativity :

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

Category

카테고리 C란

- collection of object : $\mathrm{ob}(C)$
- collection of morphism : $\hom_C(X, Y)$ where $X, Y \in \mathrm{ob}(C)$
- composition : $\circ : \hom_C(Y, Z) \times \hom_C(X, Y) \to \hom_C(X, Z)$

또한 카테고리법칙을 만족해야 한다

- identity morphism의 존재 : $\mathrm{id}_X : X \to X$ where $X \in C$

$$\mathrm{id}_Y \circ f = f \circ \mathrm{id}_X = f \text{ where } f : X \to Y$$

- $\circ$의 associativity :

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

identity morphism은 항상 유일하다

$$\mathrm{id}_1 = \mathrm{id}_1 \circ \mathrm{id}_2 = \mathrm{id}_2$$

Functor

펑터 $F : C \to D$는

- object간 매핑 : $F(X) \in D \text{ where } X \in C$
- morphism간 매핑 : $F(f) : F(X) \to F(Y) \text{ where } X, Y \in C \text{ and } f : X \to Y$

Functor

펑터 $F : C \to D$는

- object간 매핑 : $F(X) \in D$ where $X \in C$
- morphism간 매핑 : $F(f) : F(X) \to F(Y)$ where $X, Y \in C$ and $f : X \to Y$

펑터가 만족해야 하는 법칙은

- identity morphism의 보존 : $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$
- morphism 합성의 보존 : $F(g \circ f) = F(g) \circ F(f)$

Endofunctor

자기 자신으로 가는 함자 $F : C \to C$를 endofunctor라고 한다

Endofunctor

자기 자신으로 가는 함자 $F : C \to C$를 endofunctor라고 한다

하스켈의 functor는 모두 endofunctor이다

```
-- Maybe : Type -> Type
data Maybe a = Just a | Nothing
```

$\mathrm{Maybe} : C \to C$

$\mathrm{Maybe}(X) = X \coprod *$

Endofunctor

자기 자신으로 가는 함자 $F : C \to C$를 endofunctor라고 한다

하스켈의 functor는 모두 endofunctor이다

```haskell
-- Maybe : Type -> Type
data Maybe a = Just a | Nothing
```

$\text{Maybe} : C \to C$

$\text{Maybe}(X) = X \coprod *$

functor의 fixpoint는 당연히 endofunctor에서만 논할 수 있다

Initial object

initial object $\perp \in C$는

- unique morphism $u : \perp \to x$ for any $x \in C$

로 정의되는데...

Initial object

initial object $\perp \in C$는

- unique morphism $u : \perp \to x$ for any $x \in C$

로 정의되는데...
initial object가 존재한다면,“unique up to unique isomorphism”이다

initial object $\perp_1, \perp_2$가 존재한다면
$u_1 : \perp_1 \to \perp_2, u_2 : \perp_2 \to \perp_1, u_3 : \perp_1 \to \perp_1$이 각각 유일하고
identity morphism의 존재성에 의해 u_3 는 identity morphism이다 그런데
$u_2 \circ u_1 : \perp_1 \to \perp_1$ 이므로 $u_2 \circ u_1 = u_3$
합성순서가 반대여도 마찬가지의 이유로 identity morphism이 됨
즉, u_1, u_2 는 유일한 isomorphism쌍이 된다.

Initial object

initial object $\perp \in C$는

- unique morphism $u : \perp \to x$ for any $x \in C$

로 정의되는데...
initial object가 존재한다면,“unique up to unique isomorphism”이다

initial object $\perp_1, \perp_2$ 가 존재한다면
$u_1 : \perp_1 \to \perp_2, u_2 : \perp_2 \to \perp_1, u_3 : \perp_1 \to \perp_1$ 이 각각 유일하고
identity morphism의 존재성에 의해 u_3 는 identity morphism이다 그런데
$u_2 \circ u_1 : \perp_1 \to \perp_1$ 이므로 $u_2 \circ u_1 = u_3$
합성순서가 반대여도 마찬가지의 이유로 identity morphism이 됨
즉, u_1, u_2 는 유일한 isomorphism쌍이 된다.

initial object는 타입이론의 bottom type과 대응된다

F-algebra

Endofunctor $F : C \to C$에 대한 F-algebra는 (X, α) 이다

- carrier : $X \in C$
- a morphism : $\alpha : F(X) \to X$

F-algebra

Endofunctor $F : C \to C$에 대한 F-algebra는 (X, α) 이다

- carrier : $X \in C$
- a morphism : $\alpha : F(X) \to X$

여기서 각각의 F-algebra들을 object로 가지는 카테고리를 생각할 수 있는데 F-algebra (X, α)와 (Y, β) 간의 morphism m 은 $m \circ \alpha = \beta \circ F(m)$ 를 만족하는 $m : X \to Y$로 정의된다

$$\begin{tikzcd} F(X) \arrow[r, "F(m)"] \arrow[d, "\alpha"'] & F(Y) \arrow[d, "\beta"] \\ X \arrow[r, "m"] & Y \end{tikzcd}$$

Initial algebra and Lambek’s theorem

Initial algebra란 F-algebra들의 카테고리에서의 initial object를 말한다

Initial algebra and Lambek's theorem

Initial algebra란 F-algebra들의 카테고리에서의 initial object를 말한다

그런데 Lambek's theorem에 따르면..

Theorem. Endofunctor F가 initial algebra $(X, \alpha : F(X) \to X)$ 를 가진다면 α는 X와 $F(X)$ 간의 isomorphism이다

Initial algebra and Lambek's theorem

Initial algebra란 F-algebra들의 카테고리에서의 initial object를 말한다

그런데 Lambek's theorem에 따르면..

Theorem. Endofunctor F가 initial algebra $(X, \alpha : F(X) \to X)$ 를 가진다면 α는 X와 $F(X)$ 간의 isomorphism이다

Proof. (X, α)가 F-algebra라면 $(F(X), F(\alpha))$또한 F-algebra이다. (X, α)가 initial object 이므로 F-algebra에서의 morphism $i : (X, \alpha) \to (F(X), F(\alpha))$가 존재하며 $i \circ \alpha = F(\alpha) \circ F(i)$가 성립한다. 한편 $\alpha \circ F(\alpha) = \alpha \circ F(\alpha)$는 자명하므로 α는 F-algebra 에서의 morphism이다. 즉 $\alpha : (F(X), F(\alpha)) \to (X, \alpha)$

$$\begin{tikzcd} F(X) \arrow[r, "F(i)"] \arrow[d, "\alpha"'] & F(F(X)) \arrow[d, "F(\alpha)"] \\ X \arrow[r, "i"] & F(X) \end{tikzcd} \qquad \begin{tikzcd} F(X) \arrow[d, "\alpha"'] & F(F(X)) \arrow[l, "F(\alpha)"'] \arrow[d, "F(\alpha)"] \\ X & F(X) \arrow[l, "\alpha"] \end{tikzcd}$$

따라서 F-algebra의 카테고리에서의 합성 $\alpha \circ i : (X, \alpha) \to (X, \alpha)$ 을 생각할 수 있다. 그런데 X 는 initial object이므로 $\alpha \circ i = \mathrm{id}_{(X,\alpha)}$ 이고, identity morphism은 유일하므로 $\alpha \circ i = \mathrm{id}_X$ 이다. 그러면 펑터 F에 대한 법칙에 의해$F(\alpha) \circ F(i) = \mathrm{id}_{F(X)}$ 이다. 그러면 왼쪽의 commutative diagram에 의해 $i \circ \alpha = F(\alpha) \circ F(i) = \mathrm{id}_{F(X)}$ 이므로 α 는 isomorphism이다.

Lambek's theorem의 의미

Endofunctor F의 initial algebra X가 존재한다면, X는 F의 fixed point이다.
즉 inductive data type은 initial algebra이다.

Lambek's theorem의 의미

Endofunctor F의 initial algebra X가 존재한다면, X는 F의 fixed point이다.
즉 inductive data type은 initial algebra이다.

자연수는 Maybe functor의 initial algebra이다!

생략된 이야기

- catamorphism
- function type
- Curry-Howard-Lambek correspondence
- product/coproduct in CT
- initial algebra가 아닌 recursive data type