# 타입클래스에 대한 타념

Author: lumiknit(aasr4r4@gmail.com)

의식의 흐름대로 쓴 글이라 두서가 없습니다.

## 타입클래스와 대수구조

컴퓨터과학에서는 시작할 때부터 (있는지는 모르겠지만) 죽을 때까지 추상화를 강조한다. 예를 들자면 사람들이 줄을 서면서 먼저 온 사람부터 들어오는 상황을 큐를 이용해 나타낸다든가 말이다. 사실 현실에서 일어나는 일을 적어도 현실과 무관한 무언가를 통해 설명을 하는 것 만으로도 추상화를 했다고 하기는 충분하지만, 수학을 조금 공부하다보면 이것만으로는 묘하게 부족하다는 느낌이 든다.

컴퓨터과학자나 프로그래머도 비슷한 생각을 많이 했는지, 단순히 현실을 추상화하는 것 뿐만 아니라 일정한 동작 방식을 일반화하는 여러가지 방법을 생각해왔다. 결국에는 컴퓨터가 메모리와 CPU로 동작하기 때문에 추상화를 한다고 성능이 좋아지지야 않겠다만, 개발하는 속도라던가 유지보수에 드는 비용을 줄이고 즐겁게(?) 프로그램을 만들려면 일반화를 해서 잃을 것은 크지 않다고 생각하는 것이겠지.

Java 같은 경우에는 객체지향인 만큼 class를 이용해서 추상화를 시도하는데, 그 중 가장 추상적인 것이 `interface`일 것이다. 다른 어떠한 정보도 없고 순수하게 어떤 함수가 정의되어 있어야 하는지만 명세를 해둔다. 즉, 이 interface가 되기 위해서 필요한 조건을 특정 함수의 존재성으로 열거하게 된다. 그리고 이것들이 구현된 interface에서 당연히 성립해야 하는 메소드를 `default`로 만들 수 있다. 이런 사고방식은 정상적으로 만들어진 객체지향 언어라면 대부분 흉내를 낼 수 있다.

한편 Haskell은 조금 다르게 접근한다. Type class라는 것을 사용해서 어떤 type이 그 class이기 위한 조건을 몇가지 대상의 존재성으로 기술한다. (이 type class는 뭐라고 번역하는게 적절할지 도저히 모르겠다. Type이라는 말 자체도 '형'은 너무 짧고, '유형'은 적절하기는 한데 거부감이 들고 '자료형'은 너무 구체적인 값에 의존하는 느낌이다. 게다가 class도 '분류' 정도가 아닌가 싶은데 '유형 분류'라고 하면 type class의 본질을 못 살리는 것 아닌가.)

이렇게만 말하면 알겠지만 사실 위의 interface와 완전히 대척점에 있는 개념은 아니다. 실제로 코드로 적어도 둘 다 비슷하게 나타나는 경우가 많다.

```
interface Group<T> {
  T mult(T lhs, T rhs);
  T ident();
  T inv(T v);
}
```

```
class Group a where
  mult :: a -> a -> a
  ident :: a
  inv :: a -> a
```

다만 type class가 interface보다는 표현력이 더 좋은 편이다. 예를 들어서 위에서 Java 코드에서는 어떤 타입 `G`에 군 구조를 주고싶다면 `Group<G>`를 상속한 어떤 class를 만들고, 그 위에서 메소드를 구현해야한다. 그 class의 객체는 빈 객체겠지만 말이다. 그래서 이 경우 `G`가 `Group`으로서의 조건을 만족시킨다기보다는 `G`에 대해 `Group`의 조건들을 만족시키는 새로운 타입과 그 객체를 만들어서 사용한다에 가깝다. 말고도 Haskell에서는

```
class Monad m where
  unit :: a -> m a
  bind :: m a -> (a -> m b) -> m b
```

와 같은 코드를 작성할 수 있지만, Java의 interface에서는 이를 작성하는 것이 까다롭다. 이런 문제는 결국에는 interface가 어떤 구체적인 class의, 그 자신을 `this`로 받아오는 메소드 명세의 모음 (`T <: I` 라면 `interface I`는 `T -> ...` 꼴의 함수의 모음) 이라는 것에서 생긴다.

이 때문인지 최근에는 흔히 `trait`라고 불리는 typeclass와 interface의 중간쯤에 있는 개념을 갖다 쓰는 경우가 잦다.

재밌는 것은 어떻게 되었든 간에 셋 다 어떤 타입에 대한 조건들을 나열한 것이다보니, 그 조건을 만족할 경우 파생되는 함수를 만들 수 있다는 것이다. interface는 static method로 구현을 하면 되고, trait나 typeclass에서는 'trait나 typeclass를 만족시키는 타입에 대해'를 기술할 수 있도록 해준다.

한편 위는 컴퓨터 관점이고, 수학에서는 묘하게 다르다. 위처럼 어떤 대상들의 모임인 타입, 그 타입에 관하여 만족해야하는 함수들을 하나의 구조로 다루는 것은 대수학에서 자주 하는 일이다. 정확하게는 타입 대신에 집합을 사용하고 타입뿐만 아니라 함수들까지 묶어서 특정 구조를 만족시킨다고 하지만 말이다.

예를 들어서 어떤 집합 `G`와 `G` 위에서의 이항연산인 `*: G × G → G` 가 존재하면 `(G, *)`를 magma라고 부른다. 만약 이 `*`가 결합법칙을 만족시키면 `(G, *)`가 semigroup이 된다. 만약 항등원 `e: G` (또는 initial object를 사용해서 `e: 1 → G` 정도로 적기도 하다만) 이 존재한다면 `(G, *, e)`는 monoid가 된다. 그리고 모든 객체에 대해 역원이 존재하는 것은 역원을 만드는 함수 `i: G → G`가 구성될 수 있음으로 보일 수 있고, 이 경우 `(G, *, e, i)`가 group이라고 하게 된다.

이걸 좀 더 코드처럼 기술한다면

```
(G, *, e, i) is an algebraic structure 'Group' where
  G: set
  *: G * G -> G                -- Binary operator
  e: 1 -> G                    -- Identity
  i: G -> G                    -- Inverse
  for all a, b, c, g: G,
    (a * b) * c = a * (b * c)  -- Associativity
    e * a = a * e = a          -- Identity Law
    a * i(a) = i(a) * a = e    -- Inverse Law
```

정도가 될 것이다. 이것은 위의 typeclass에서 적은 것보다 더 복잡한데,

- 집합 `G`만으로 공리를 만족시키는 것이 아니라 `G`와 특정 연산들이, 그 연산들로 구성된 공리를 만족시킨다. (물론 저 연산들이 매우 자연스럽거나 전형적인 경우에는 다 생략하고 `G`라고 적기도 한다.)
- 만족시켜야 하는 성질이 있다. 원래는 typeclass 등에서도 이런 무조건 성립해야하는 성질이 있기야 하지만, 이걸 자동으로 검증하는 것 자체가 매우 힘든 일이고 실제 계산에는 별 의미가 없다보니 프로그래머가 알아서 잘 증명했을거라 신뢰하는 것이다.

이 중에서 만족시켜야 하는 성질들은 작성자를 신뢰해서 다 된다고 퉁쳐버리게 되면 어떤 구조는 1개 이상의 집합과 (집합이 없으면 유의미한 연산이 나오지 않으므로) 그 집합에 대한 1개 이상의 연산이 (연산이 없으면 그냥 집합 이상의 의미를 갖지 않으므로) 모여서 특정 구조가 될 수 있을 것이다. 특히 집합이 하나만 있는 경우에 연산은 음이 아닌 임의의 정수 `n`에 대해 `n`개의 집합의 원소를 받아 그 집합의 원소를 내놓는 함수들이 전부이므로 (2개 이상의 원소를 내놓는 함수도 있겠다만, 그 함수에 projection을 합성하여 여러 함수로 분해할 수 있으므로 별 차이 없다.) 연산의 구조를 음이 아닌 정수에 대응시킬 수 있다. (`n <=> G^n -> G <=> G^G^n`) 이를 바탕으로 구조에 필요한 조건을 정수의 카르테시안 곱으로 표현할 수 있다. (위에서 계속 써먹었던 group의 경우에는 `(2, 0, 1)`처럼 나타낼 수 있다.)

이렇게 구조가 정해졌다면 위 성질들을 공리로 하여 연역적으로 성립하는 성질들을 찾아볼 수 있는데, 보통 이것이 정리라고 부르는 녀석이다. 그런 정리들이 구체적인 모형들을 바탕으로 증명이 될지, 아니면 순수하게 구조에 있는 공리들로만 증명이 될지는 별개로 두고 말이다.

결국 각각에서 구조를 정의하는 방법은 대충 아래와 같은 느낌이다.

- `class T <: interface`라고 하기 위해서는 `I`에 기술된 `T -> ...` 꼴의 함수들이 `T`에 구현해야한다. (그리고 `T`의 객체에서 이를 호출할 수 있다.)
- `type T <: trait U`라고 하기 위해서는 `U`에 기술된 `T`에 관한 `F(T)` (where `F: * -> *`) 꼴의 함수들을 구현해야한다.
- `type T <: typeclass C`라고 하기 위해서는 `C`에 기술된 `T`에 관한 함수들이을 구현해야한다.
- 어떤 집합과 함수로 구성된 튜플 `T`가 어떤 대수구조 `A`라고 하는 것은 `T`의 집합과 함수들이 `A`에서 말하는 공리를 만족시킨다는 것이다.

다들 그렇게 느끼는지는 모르겠지만, 프로그래밍에서와 수학에서는 서술하는 순서가 약간 다르다고 생각한다. Trait 등과 같은 경우에는 '이러이러한 조건을 만족시키는 것을 특별한 구조로 볼거야. 그리고 나는 이 타입에 대해 이러이러한 함수를 (새로) 정의할테니, 이 타입은 이 특별한 구조라 볼 수 있어.'와 같은 순서지만, 대수구조에서는 '이 집합에 대해 이러이러한 연산을 만들어볼 수 있어. 그러고나서 보면 이 집합과 연산들은 특별한 구조의 조건을 만족시키니까 그 구조로 취급해도 돼.'에 가깝다. 프로그래밍에서는 타입에 함수들을 접합해버리지만, 수학에서는 그러지는 않는다고 해야될까.

이 때문에 그리 문제가 되지는 않지만, 아쉬운 상황이 종종 생기기도 한다. 예를 들어서, 다음과 같은 monoid를 생각해보자.

```
class Monoid m where
  mult :: m -> m -> m
  id :: m
```

그런데 우리는 정수집합에 자연스럽게 주어지는 두 연산 `+`, `*`가 있다는 것을 알고 있고, 정수집합은 각 연산과 함께 monoid를 이룬다:

```
-- Monoid (Z, +)
instance Monoid Int where
  mult a b = a + b
  id = 0
-- Monoid (Z, *)
instance Monoid Int where
  mult a b = a * b
  id = 1
```

그리고 monoid에 대해서는 monoid의 여러 점들의 합을 구하는 것을 비교적 간단하게 구현할 수 있다. (사실 monoid를 잘 써먹으려면 병렬적으로 계산하는 방식이 좋겠지만, 여기서는 단순하게 구현하자.)

```
prod :: Monoid a => [a] -> a
prod [] = id
prod (x : xs) = mult x (prod xs)
```

여기까지 마쳤다면 `prod [1, 2, ..., n]`으로 1부터 n까지의 합이나 곱을 구할 수 있다. 문제는 합을 할지 곱을 할지 고르는 것이지만 말이다. 내가 알기로는 애초에 저런 모호한 경우를 없애기 위해서 Haskell에서는 instance를 두번 정의하는 것이 안 된다고 알고 있는데 ghc extension으로 어떻게 해결이 가능한지는 모르겠다. 어쨌든 저렇게 여러가지로 정의해서 가지고 놀게 되면 어떤 구조를 사용할지 추론하지 못하고 명시를 해야되겠지만 말이다.

## 다변수 타입클래스

특이하다고 할 정도는 아니지만, Haskell의 typeclass는 일변수만 허용된다. 일단 영변수, 즉 인수가 없는 타입클래스는 의미가 없다. 그냥 전역에 함수를 정의하고 끝이기 때문이다. 만약에 증명시스템이라도 있어서 연산간에 성립해야하는 성질이 성립하는지 증명이라도 할 수 있다면 몰라도, 그것을 작성자 몫으로 넘기는 상황에서는 연산만으로는 할 수 있는 것이 없다. 그리고 결정적으로 구조 자체가 타입에 들러붙는다는 설계 상 타입만으로는 뭘 할 수가 없을 것이다.

일변수인 경우에는 위의 group과 같은 집합 하나짜리 대수구조와 비슷하지만, 연산에 들어가는 집합이 구조에 해당하는 집합이 아니어도 상관없다는 차이가 있겠다.

변수가 2개 이상인 경우는 의미가 없는 것도 아니고, 예시가 없는 것도 아니지만, 구현 및 성능에 조금 심각한 영향을 미치게 된다. 예를 들어서 다음과 같은 구조를 생각해보자.

```
(A, B; f) is 'equipotent' when
  A, B are sets
  f: A >->> B     -- bijective (>-> for monic, ->> for
epic)
```

이를 Haskell 문법과 비슷하게 나타내는 것은 간단하다.

```
class Equipotent A B where
  f :: A -> B
```

문제는 이 typeclass의 구현이다. 예를 들어서 (bijectivity를 만족시키지는 않더라도) 정수를 `A`에 두어서 구현을 하는 방법은 여럿 존재한다:

```
instance Equipotent Int Float where
  f n = fromIntegral n
instance Equipotent Int String where
  f n = show n
instance Equipotent Int Int where
  f n = n
```

여기서 만약 `f 3`와 같은 표현식은 이것만으로는 타입이 결정되지 않는다. `Int`와 `Equipotent`한 타입이 3개나 있기 때문이다. 그래서 `f 3 :: String` 처럼 `B`가 무엇인지 명시를 해야한다. 그런데 그렇다고 하면 `f` 자체는 타입이 무엇이 되야하며 어떤 함수를 가져야 할까.

일변수의 경우에는 비교적 쉽게 처리할 수 있다. 보통 OOP에서 virtual table을 유지하는 것처럼 typeclass의 인자타입에 대한 function table을 유지하면 된다. 그리고 만약 함수에서 어떤 typeclass를 요구하면 그 function table을 세트로 인자로 전달하면 된다는 것이다. 예를 들자면 `f :: Monad m => m Int -> m Int` 라는 함수가 있다면, 코드 상에서는 `a :: Maybe Int`로 그대로 호출하여 `f a`처럼 인자 하나를 넘기지만, 내부적으로는 `f :: Monad m -> m Int -> m Int`처럼 `Monad`에 관한 함수 구현을 담는 테이블을 인자로 받도록 하고 호출 시 `f (Monad_instance_of_Maybe) a`처럼 테이블을 넘겨주는 코드를 자동으로 추가해준다. 저 테이블이 하나의 타입으로 결정되므로 특정 인자를 받는 시점에서 같이 넘겨받도록 하면 문제가 없다.

다만 다변수는 인자를 하나 받는다고 특정되지가 않기 때문에, typeclass의 각 변수에 해당하는 값을 받을 때 테이블 하나가 아닌 가능한 모든 테이블을 넘겨받아야한다. 그리고 추가적으로 값을 받을 때 해당 타입 외의 경우를 지워나가다가 하나만 남으면 그 테이블을 사용해야한다. 예를 들어 위의 `Equipotent`로 예를 들자면, `g :: Equipotent A B => A -> B -> Bool`가 주어졌을 때 `g 3`과 같이 호출하면 `A = Int`인 것이 확정되지만, 이를 만족하는 instance가 3개 있으므로 이 3개를 전부 인수로 넘겨야한다. 그리고 다음 인수로 어떤 타입이 들어오냐에 따라 이 instance 중 하나를 골라내야 한다.

따라서 위 instance의 모음을 적당히 뭉쳐서 저장해둘 방법과, 실행시점에 인수의 타입에 따라 그 중 하나를 집어내는 코드가 필요하게 되는데, 이것만으로도 시간/공간 소모가 커지게 된다.

그리고 이렇게 구현할 수 있는 typeclass가 상상을 초월할 정도로 커질 수도 있다. 예를 들어서 두 객체가 동일한지 비교하는 `Eq`가 있겠는데, 지금은 같은 타입인 것들만 비교하지만, 서로 다른 두 타입을 비교하도록 한다면 `n`개의 타입에 대해 `n^2`개나 되는 instance가 생길 수도 있다. 만약 프로그램 상의 타입 개수가 `n`개이며, 이를 바탕으로 `m`-변수 typeclass를 만든다면 `n^m` 크기의 테이블 목록이 나올 것을 각오해야한다.

그렇다고 타입을 명시를 하자니, ad-hoc polymorphism을 사실상 포기하는 것이니 애매하기도 하다.

물론 수학쪽에서는 함수를 넘겨주거나 하는 경우가 없으니 전혀 상관없는 얘기긴 하다.

## 동형사상

수학에서 대수적 구조야 여러 면에서 중요하지만, 이 중요성에 큰 기여를 하는 구상들 중에서 (준)동형사상 (homo/isomorphism)을 빼놓는 것은 섭섭할 것이다. 준동형사상 `f: A -> B` 자체는 `A`와 `B`가 각각, 또는 그 부분구조와 얼마나 비슷하게 동작하는지를 보여주는 함수지만, 구조들을 쪼개고 분류하거나 호몰로지를 구성해서 공간의 특성을 뽑아내는 등 매우 유용한 도구임이 틀림없다. 정의 자체가 대수적 구조, 정확히는 연산을 보존하는 함수라는 것처럼, 계산에서도 유용하게 쓰여왔다. 예를 들자면 `(R>0, *)`에서의 곱을 `log`와 `exp`로 `(R, +)` 공간으로 옮겨서 연산하는 것이 있겠다.

프로그래밍에서 예를 들자면 임의의 리스트와 리스트 연결 연산자를 모아둔 `([a], ++, [])`과 음이 아닌 정수의 집합 `U`와 덧셈 연산자를 모아둔 `(U, +, 0)`, 부호 없는 8비트 정수를 모아둔 집합 `U8`과 덧셈 연산자를 모아둔 `(U8, +, 0)`이 있겠다. 셋 다 monoid `(M, *, e)` 구조에 그 성질을 만족시킨다는 것은 자명하다. 여기서 우리는 다음과 같은 함수들을 만들어 볼 수 있다.

```
len :: [a] -> U
len [] = 0
len (x : xs) = 1 + len xs

mod8 :: U -> U8
mod8 n = n % 256
```

여기서 이 `len`과 `mod8`이 준동형사상이라는 것은 단순계산으로 보일 수 있다.

```
-- identity
len [] = 0
-- op, for n-list x and m-list y,
len (x ++ y)
  = len ((x_1 : ... : x_n : []) ++ (y_1 : ... : y_m : []))
  = len (x_1 : ... : x_n : y_1 : ... : y_m : [])
  = 1 + ... + 1 + 1 + ... + 1
  = len (x_1 : ... : x_n : []) + len(y_1 : ... : y_m : [])
  = len x + len y
-- By properties of modulus,
mod8 0 = 0
mod8 (a + b) = mod8 a + mod8 b
```

추가적으로 `len`과 `mod8` 모두 surjective이므로, `[a] > U > U8`이라는 submonoid 관계도 확인할 수 있다. 다만 실제 프로그램에서는 각 타입별 부분구조 관계가 그다지 의미가 없어보여서 문제지만 말이다.

하지만 어느 길을 가느냐에 따라서 성능은 조금 달라질 수 있다. 예를 들어서 두 리스트 `x`, `y`의 길이의 합을 8bit 내에서 구하고싶다면 `len (x ++ y)`처럼 두 리스트를 연결하고 길이를 구하는 것보다는 `len x + len y`처럼 각각의 길이를 구해서 더하는 것이 시간/공간 양측 모두 더 우수할 것이다. 또한 함수호출 비용이 덧셈에서 얻는 이득보다 훨씬 크다면 `mod8 a + mod8 b` 보다는 `mod8 (a + b)`로 계산하는 것이 유리하다. 이런 경우에는 `mod8 (len x + len y)`가 최적의 성능을 내는 조합이 될 것이다.

여기서 이항연산들을 `a -> a -> a` 같은 currying된 형태 말고 `(a, a) -> a` 형태라고 해보면 `len (x ++ y)`는 `(x, y) |> (++) |> len`과 같이 `([a], [a]) -> [a] -> U` 순으로 바뀌며, `len x + len y`는 `(x, y) |> (len, len) |> (+)`와 같이 `([a], [a]) -> (U, U) -> U` 순으로 바뀌게 된다. (Diagram으로 나타내면 좋겠지만, 이 문서가 LaTeX이 아니어서 골치아프다.) 알 사람들은 알겠지만, 여기서 `len`이라는 함수를 `(len, len)`처럼 만들어 줄 때는 보통 대각사상(diagonal morphism)을 사용한다. 보통은 대문자 델타를 사용하지만, 여기서는 `diag`라고 하자. 이는 `diag :: a -> (a, a)`, `diag x = (x, x)`와 같이 쉽게 구현 가능하다.

`diag len = (len, len)`는 나름 특이한 함수다. 우리가 원하는 이 함수의 동작은 '1번째 칸의 함수는 1번째 칸의 값에, 2번째 칸의 함수는 2번째 칸의 값에 적용한 뒤 두 반환값으로 다시 튜플을 만들기' 정도가 될 것이다. 다만 `(a -> b, c -> d)`의 타입을 `(a, c) -> (b, d)` 와 동일한 것으로 취급하는 것 자체가 꺼려질 수 있고, 함수가 아닌 것을 함수로 취급하는 것도 이상할 수 있어, 이 동작은 직접 구현해야 한다. 다행히 이는 `tmap :: (a -> b, c -> d)`, `tmap (f, g) (x, y) = (f x, g y)`와 같이 쉽게 구현할 수 있다. 이를 이용해 정리하면

```
(x, y) |> (++) |> len = (x, y) |> tmap (diag len) |> (+)
len . (++) = (+) . tmap (diag len)
```

이라는 등식이 성립함을 알 수 있고, 이것이 보통 준동형사상에 대한 commutative diagram (가환도식?)에서 자주 볼 수 있는 형태다.

이런 과정을 거치는 계산은 생각보다 흔하게 볼 수 있어, 이미 Haskell 내에 이를 위한 패키지가 구현되어 있다. 대표적으로 `Control.Arrow`라는 패키지가 있어서 서로 다른 함수를 병렬적으로 합성하는 것이 가능하다. 이를 이용하면 `tmap (diag len)` 대신 `len &&& len`이라고 나타낼 수 있다. (처음에 말했던 `a -> b`와 `c -> d`를 받아서 `(a, c) -> (b, d)`를 만드는 함수가 `&&&`인 셈이다.)

이외에도 이런 과정 자체가 functor와 비슷하게 동작하기 때문에 (정확히는 functor가 homomrphism의 일종이겠지만) 2-tuple 등에 대해 적절히 구현을 해준다면 `tmap (diag len)` 대신 `fmap len` 정도로 대체할 수도 있긴 하다.

준동형사상 자체는 두 집합 사이에 존재하는 몇몇 성질을 만족하는 함수에 불과해서 그런지 따로 type class를 만들지는 않는 것 같다.

## 다형타입

프로그래밍을 하다보면 비슷한 구조를 많이 보게 되며, 이를 일반화해서 비슷한 구조를 쉽게 양산해내려는 시도를 한다. 시작이 무엇인지는 모르겠지만, 보통은 (조금 문제는 많지만) C++의 template, Java 등의 generic, 함수형 언어들의 polymorphic type이 있을 것이다. 결국 얘네들이 이를 이용하여 얻고싶은 것은 다형성(polymorphism)이다.

가장 대표적인 예시가 자료구조가 되겠다. 대표적으로 리스트는 여러 값들에 '다음 값의 위치'를 추가적으로 저장하는 자료구조인데, 정수가 들어가든 문자열이 들어가든 '값'과 '다음 위치'라는 2가지 정보가 들어가야 한다는 것은 항상 같다. 이 때문에 저 '값'의 타입을 특정하지 않고 변수처럼 받을 수 있게 만든다. 이를 Java에서는 `List<T>`처럼 쓰게 되고, Ocaml에서는 `t list`라고 하거나 Haskell에서 `List a`라고 하게 된다.

이 경우에는 타입 자체가 어떤 타입을 받아서 새로운 타입을 내놓는 함수처럼 동작하게 되며, 이 때문에 타입의 타입을 나타내는 kind를 써서 `List :: * -> *`처럼 표기를 하게 된다. 다른 언어에서는 `*` 만을 trait 등에 사용할 수 있도록 하지만, Haskell의 type class는 `* -> *`처럼 `*` 외의 타입(?)도 class처럼 다룰 수 있게 해준다.

근데 문제는 이것이 대수구조에서의 무엇에 대응하냐는 것이다.

예를 들어서 `Maybe`에 다음과 같이 연산을 주면 `monoid`가 성립한다.

```
instance Monoid (Maybe a) where
  mult Nothing r = r
  mult l _ = l
  id = Nothing
```

이 표현 그대로라면 `(Maybe(a), *, e) for any set a`와 같은 형태가 되는데, 여기서 `Maybe` 자체는 임의의 집합을 다른 집합으로 옮겨주는 함수여야 한다. 이렇게 생각하면 첫번째로 매우 큰 문제가 생기는데, 정의역이 모든 집합이면 그 정의역은 집합이 아니라는 것이다. 그러므로 정의역이 고유 모임(proper class)가 되고, `Maybe` 자체는 함수가 아니다. 이렇게 귀찮은 상황에서 할 수 있는 무난한 선택은 `Maybe`의 정의역 자체를 고유 모임으로 하는 범주를 만들어버리는 것이다. 즉, 모든 집합과 함수의 범주인 `Set`과 `Set`의 각 대상에 `Nothing`을 원소로 추가하고, 임의의 사상 `f: A -> B`에 대해서는 `f(Nothing) = Nothing`만 추가하여 새로운 범주 `MSet`을 만들자고 생각해볼 수 있다. (물론 이미 `Set`에는 모든 집합이 있으므로 어느 집합에도 속하지 않은 `Nothing`이라는 원소는 없다. 그래서 각 집합 `S`에 대해 `{(0, 0)} ∪ ({1}×S)`와 같이 disjoint union을 해준 뒤 `(0, 0)`을 `Nothing`으로 취급해야한다. 이 과정에서 `MSet`의 대상은 `Set`의 부분모임이 된다.) 그러면 `M^0: Set -> MSet`이라는 함자를 만들 수 있다.

그런데 실제로는 쓰지 않지만 `Maybe (Maybe Int)`처럼 `Maybe`를 중복시킬 수도 있는데, 이 경우에 위 과정을 반복해서 `MMSet`과 `M^1: MSet -> MMSet`이라고 만들 수 있다. 다만 이런 방식의 inductive data structure는 유한번에 생성이 되므로, `Set -> MSet -> MMSet -> MMMSet -> ...` 자체는 가산무한번 반복되고, 이것들 전부를 disjoint union하여 `MaybeSet`을 만들어볼 수 있다. 그러면 `Maybe`를 `MaybeSet`이라는 범주에 대한 자기함자로 생각할 수 있다.

여기까지를 정리하자면 `Maybe`라는 녀석은 모든 집합을 포괄하는 `MaybeSet`이라는 큰 범주 위의 자기함자이며, `(Maybe(a), *, e)`가 `Monoid`라는 것은 이 `MaybeSet`의 부분범주의 모든 대상 `X = Maybe(a)`에 대해 적절히 정의된 연산 `*: X -> X -> X`와 `e: X`가 존재한다는 것이다.

여기서 한가지 재밌는 말장난을 볼 수 있다.

1. 위 instance에서는 `*`와 `e`가 어떤 타입 `T`든 상관없이 정의된다.
2. 아래의 표기에서는 `*`와 `e`는 어떤 집합 `S`가 주어졌을 때 `Maybe(S)`라는 집합 위에서 정의된다.

어떤 차이냐고 할 수 있는데, 위의 instance 정의에서는 `*`와 `e`는 `Maybe`의 치역이기만 하면 어떤 집합의 원소든 상관없이 동작하도록 정의가 되어 있는데, 다르게 말해 정의역이 `Maybe(MaybeSet)`의 모든 대상의 합집합을 포함한다는 것이다. 그런데 `MaybeSet` 자체가 이미 모든 집합을 포함하기 때문에 정의역이 집합을 넘어서게 된다.

때문에 `*`가 함수일 수 없다.

반면 일단 `Maybe(S)` 자체는 집합이므로 2처럼 집합을 먼저 고정해두면 `*`는 그리 문제 없이 정의가 된다.

## 귀납적 자료구조

위에서 집합론을 벗어나 폭발해버린 근본적인 이유는 애초에 모든 집합의 모음이 너무 크기 때문이었다. 그런데 일단 Haskell에서는 모든 값이 귀납적 자료구조 (또는 대수적 자료구조)로 생성이 되는데, 이것을 모두 모으면 집합이 폭발할까?

일단 우리가 무한한 코드를 작성하여 무한히 실행하면 결과가 나오지 않으므로 '유한한 길이의 코드'를 실행하여 '유한시간 내에 종료되어' 만들어진 값들만을 우리가 실제로 써먹을 수 있는 유용한 값이라고 하고, 이 값들의 모음 `V`가 얼마나 큰지 생각해보자. (단, 여기서는 lazy evaluation와 mutation은 허용하지 않도록 하자.)

우선 유한한 코드에서 값이 생성이 되어야 하는데 오류가 없는 코드는 일단 모든 문자열의 집합의 부분집합이다. 문자열의 집합이 `{2^n | n is non-negative integer}`의 각 원소의 합집합인데, `2^n`은 유한집합이므로 유한집합의 가산무한합집합이다. 즉, 이는 가산집합이다. 따라서 오류가 없는 코드의 집합 `C` 역시 가산집합이며, `List`를 만들면 임의의 자연수를 생성할 수 있으므로 `C`는 가산무한집합이다. 각 코드가 어떤 값은 만들어낼 것이므로 `C -> V`의 함수가 존재하는데, `V` 역시 자연수를 포함하므로 `V`는 가산무한집합이다.

모든 가산무한집합의 집합은 `2^N`과 크기가 같으므로 실수의 집합과 같은 크기다. 즉, 이것만으로는 집합이 터져버리지는 않는다.

이렇게 생성가능한값과 그 값들 사이의 함수들의 범주를 `Val`이라고 하자. 그렇다면 `Maybe` 자체는 작은 범주 `Val`에서의 자기함자이자 자기함수라고 생각할 수 있으며, 크기에 관한 문제가 생기지 않는다.

이를 다르게 생각해보면 평소에 타입을 집합의 일종이나 집합의 구조를 준 것으로 생각하기도 하는데, 실제로는 생성 가능한 값들의 집합으로 제한해야 집합론을 탈출해버리는 문제가 생기지 않는다.

한편으로는 다형타입의 변수에 제한을 두지 않을 경우, 다형타입 자체는 집합론 내에서 뭐라고 기술할 방법이 없다는 것 역시 알 수 있는 듯 하다.

## `Type`

결국 컴퓨터에서 실행되기 위한 프로그램에서는 모든 유한 코드로 유한 시간 내에 생성할 수 있는 값인 `Val` 의 부분집합만 고려하면 차고 넘친다.

예를 들어서 우리가 유한한 코드로 기술할 수 있는 타입 `T`가 있으면 `T`에 해당하는 값 중 유한코드로 유한시간에 생성할 수 있는 값의 집합은 `T ∩ Val`의 원소 뿐이다. 이를 `Val(T)`라고 표기하기로 하자.

`T`와 `Val(T)`는 다를 수 있다. 예를 들자면 Haskell에서는 `a = 1 : a`와 같이 순환하는 리스트를 만들 수 있으며, 이는 엄연히 `[Int]`를 만족시킨다. 하지만 lazy evaluation이나 mutation 없이는 유한 시간에 생성하지는 못하므로 `Val([Int])`는 아니다.

여기서 `Type = {Val(T) | type T}`라고 하자. 즉, 우리가 작성할 수 있는 타입 T가 가질 수 있는 원소의 집합이 `Type`의 원소다.

귀납적 자료구조로 만들 수 있는 값들은 곱과 쌍대곱(또는 disjoint union)으로도 만들 수 있다. 그리고 유한코드/유한시간에 생성가능한 값 `a`, `b`가 있다면 곱인 `(a, b)`와 쌍대곱인 `(0, a)`, `(1, b)` 역시 유한코드/유한시간에 생성 가능하다. 즉 `Type`은 곱과 쌍대곱에 대해 닫혀있다.

단 `Type`은 함수에 대해서도 닫혀있어야 하지만, 임의의 `A, B ∈ Type`에 대해 `A -> B ⊊ B^A`이다. 유한하게 구현할 수 있는 함수는 실제로 존재할 수 있는 모든 함수보다 훨씬 더 적다.

## Parametric polymorphism

(** 아직 말이 되는지 제대로 확인 못 함 **)

구체적인 타입들을 명세로 하는 함수들은 `Type`의 원소인 것이 확실하지만, 다형성을 띄는 함수는 그렇지 않다는 것을 알 수 있다. 대표적으로 항등함수의 경우,

```
ident :: a -> a
ident x = x
```

와 같이 정의되는데, 이 ident의 타입은 `Type`의 원소라기보다는, `Type`의 어떤 원소가 들어와도 무관한 변수에서 `Type`을 내놓는 함수에 가까워보인다.

그와 별개로 `ident` 자체의 정의역과 공역은 특이한데, 다른 함수는 `Type`의 원소를 정의역과 공역으로 취하지만, `ident`는 어떤 값이 들어와도 상관 없는만큼 `Val`의 모든 원소를 받을 수 있고, `Val`의 모든 원소를 내놓는다. 즉, `ident: Val -> Val`에 가깝다. 하지만 `Type`에는 `Val`이 포함되지 않는다. (만약 `Val`이 `Type`의 원소라면 `Type`들에 대해 유한번 곱과 쌍대곱을 취해서 `Val`을 만들어낼 수 있어야하는데, 이는 불가능하다. 왜?)

따라서 `Type`에 `Val`을 일종의 top 원소로 넣어서 확장을 하고, 이를 `VType`이라고 하자.

`VType`에서는 함수가 무한히 커질 수 있냐고 물을 수 있지만, 실제로는 그렇지 않다. 예를 들어서 `Val`의 모든 원소를 받을 수 있는 `a -> *` 꼴의 함수를 생각해보면 아래와 같이 분류된다.

1. 유한개의 `Type`의 원소의 값에 대해서는 그 타입대로 핸들링을 할 수 있다. (무한개의 `Type`의 원소를 개별적으로 처리하려면 무한히 긴 코드가 필요하므로 불가능하다.)
2. 나머지 값들은 뭉탱이로 처리를 해야되기 때문에 다음과 같은 두가지 경우 밖에 없다.
    a. 만약 `a -> a` 꼴이면 나머지는 항등함수처럼 입력된 값을 그대로 반환해야한다.
    b. 만약 `a -> T` 꼴이면 `T`에 해당하는 값 중 하나를 반환해야한다.

여기서 1은 유한개의 함수를 곱한 것과 같으며, 그 외 나머지를 2에 따라 항등함수로 처리할지 상수함수로 처리할지 결정을 해야한다. 즉, `VType`가 함수의 개수때문에 집합이 터져버리는 일은 발생하지 않을 것이다.

이 경우에 `ident`를 `VType` 내의 원소 중 하나인 `Val -> Val`이라고 할 수 있다.

`List`의 길이를 계산하는 `len`은

```
len :: [a] -> Int
len [] = 0
len (x : xs) = 1 + len xs
```

와 같이 정의되는데, 이 경우 `len`의 타입을 `List(Val) -> Int`와 같이 `VType` 내에서 생각해볼 수 있다.

## 타입클래스

타입클래스를 정의할 때는 종속되는 타입의 개수와, 그 타입들에 대해 정의되어야 하는 함수들의 타입이 주어진다. 즉, `n`개의 타입과 `m`개의 함수에 대해 정의한다고 하면 `TypeClass = {C | C: VType^n -> VType^m}`가 타입클래스 명세의 집합이 된다. 각 명세는 `n`개의 타입을 받아서 `m`개의 함수의 타입을 내놓는 것이다.

어떤 타입클래스 `C`에 대한 객체의 집합은 `Instance(C) = {(T_1, ..., T_n; f_1, ..., f_m) in VType^n × Val^m | (f_1, ..., f_m) in C(T_1, ..., T_n)}`이라고 할 수 있다.

## 타입 제약

Haskell의 타입클래스에서는 타입클래스나 함수를 정의할 때 어떤 타입이 구현해야하는 타입클래스를 조건으로 걸 수 있다.

```
(+) :: (Num a) => a -> a -> a
class (Eq a) => Ord a where ...
```

보통은 어떤 타입클래스에 필요한 조건이 중복되는 경우에 이를 편하게 작성하라고 제공해주는 기능이다.

```
class AbGrp a where
  (+) :: a -> a -> a
  id  :: a
  neg :: a -> a
class AbGrp a => Ring a where
  (*) :: a -> a -> a
class Ring a => RingWithUnity a where
  unity :: a
```

다만 타입클래스에 제약을 주는 것은 각 함수들마다 제약을 주는 것과 크게 다르지 않다.

```
class Ring a where
  (*) :: AbGrp a => a -> a -> a
```

만약 `f :: Cls a => a -> T` 꼴의 함수가 있다면, 이는 `f: Instance(Cls) -> VType` 꼴의 함수로 생각할 수 있다. 문제는 `f :: (C1 a, C2 a) => a -> T`와 같이 여러 조건이 붙는 경우에는 하나의 instance에 따라 다른 instance에 제약이 들어갈 수 있다.

타입클래스 자체에 붙는 제약은 타입클래스 자체를 확장하여 `TypeClass = {C | C: VType^n -> TypeClass^l × VType^m}`와 같이 만들 수 있겠다만, 사이클이 생기면 안 되므로 적절히 계층을 나눌 필요가 있으며, 마찬가지로 중간의 타입이 동일한지 확인할 필요가 있을 것이다.

## 왜 이런 생각을 했나

- 대수구조 자체가 집합과 연산이 특정 조건을 충족시킬 때 만족하는 성질들을 싹 다 모아두었다가, 어떤 집합과 연산이 비슷한 구조일 때 그 성질들을 다 끌어다가 때려박는데, 이게 다형성이랑 거의 비슷한 느낌 아닌가 해서 고민해봄.
- 구조 자체를 날로 PL에 때려박아서 더 단순한 구조를 만들 수는 없을까 생각했음.
- 애초에 타입을 날로 범주에 때려박으면 의미가 있는지 궁금했음.