

펑터(Functors)

8.1 카테고리

지금까지 우리는 하나의 범주 — 타입과 함수들 — 만을 보았습니다. 그래서 범주에 관한 필수 정보를 빠르게 모아보겠습니다.

카테고리란 객체와 그들 사이를 연결하는 화살표들로 구성된 집합입니다. 모든 짝의 결합 가능한 화살표들은 결합될 수 있습니다. 결합은 결합 법칙(associative)을 따르며, 각 객체에는 그 객체로 돌아오는 항등 화살표(identity arrow)가 있습니다.

타입과 함수가 카테고리를 형성한다는 사실은 다음과 같이 합성(composition)을 정의함으로써 Haskell에서 표현될 수 있습니다:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)  
g . f = \x -> g (f x)
```

두 함수 `f` 뒤에 `g`를 합성한 것은 먼저 `f`를 그 인수에 적용한 다음, `g`를 그 결과에 적용하는 새로운 함수입니다.

정체성(Identity)은 다형(polymorphic)적인 “아무 것도 하지 않는” 함수(function)입니다:

```
id :: a -> a  
id x = x
```

여러분은 이러한 합성이 결합법칙(associative)을 만족하며, `id`와 합성하는 것은 함수에 아무런 영향을 미치지 않는다는 것을 쉽게 확인할 수 있습니다.

카테고리(category)의 정의를 바탕으로, 우리는 온갖 종류의 이상한 카테고리를 생각해낼 수 있습니다. 예를 들어, 객체(object)와 화살표(arrow)가 없는 카테고리가 있습니다. 이는 모든 카테고리의 조건을 자명하게 만족합니다. 또 다른 예로, 단일 객체와 단일 화살표를 가진 카테고리도 있습니다(어떤 화살표일지 짐작할 수 있습니까?). 두 개의 연결되지 않은 객체를 가진 카테고리도 있으며, 두 객체가 단일 화살표로 연결된 카테고리(그리고 두 개의 항등 화살표(identity arrow)도 포함됩니다)가 있습니다. 이들은 제가 막대 그림 카테고리(stick-figure category)라 부르는 것의 예입니다—작은 수의 객체와 화살표를 가진 카테고리들입니다.

집합의 범주(Category of sets)

우리는 또한 모든 화살을 (항등 화살을 제외하고) 제거할 수 있습니다. 이러한 단순 객체 범주(bare-object category)를 이산(discrete) 범주(discrete category) 또는 집합이라고 합니다¹. 화살을 구조(structure)와 연관시키기 때문에, 집합은 구조가 없는 범주입니다.

¹“크기” 문제를 무시하고

집합들은 자신들만의 범주(영문 용어: category)를 형성하며, 이를 **Set**이라 부릅니다². 그 범주의 대상(영문 용어: objects)들은 집합들이고, 화살표(영문 용어: arrows)는 집합들 사이의 함수들입니다. 이러한 함수들은 특별한 종류의 관계로 정의되며, 그 관계들은 쌍들의 집합으로 정의됩니다.

최소 근사치로, 우리는 집합(category of sets) 범주에서 프로그래밍을 모델링할 수 있습니다. 우리는 종종 타입을 값들의 집합(set)으로 생각하고, 함수를 집합-이론적 함수(set-theoretical functions)로 생각합니다. 이는 아무런 문제가 없습니다. 사실 지금까지 묘사한 모든 범주론적 구성은 그들의 집합-이론적 뿌리를 가지고 있습니다. 범주론적 곱(product)은 집합의 데카르트 곱(cartesian product)의 일반화이며, 합(sum)은 분리합(disjoint union)이고, 그 외 등등이 있습니다.

범주 이론이 제공하는 것은 더 정확해진다는 점입니다: 반드시 필요한 구조와 불필요한 세부 사항 사이의 미세한 구분을 제공합니다.

집합이론적 함수(set-theoretical function)는 예를 들어, 우리가 프로그래머로서 작업하는 함수의 정의에 맞지 않습니다. 우리의 함수는 어떤 물리적 시스템에 의해 컴퓨팅될 수 있어야 하기 때문에 기본적인 알고리즘을 가져야 합니다. 컴퓨터든 인간의 뇌든 상관없이 말입니다.

반대 범주(Opposite categories)

프로그래밍에서는 유형(타입)과 함수의 범주에 중점을 둡니다 (category of types and functions), 그러나 이 범주를 출발점으로 사용하여 다른 범주들을 구성할 수 있습니다.

하나의 그러한 범주는 역 범주(opposite category)라고 불립니다. 이 범주에서 모든 원래 화살표(arrows)들이 뒤집혀집니다: 원래 범주에서 화살표의 출발지(source)라고 불리던 것이 이제는 그 목표지(target)로 불리며, 그 반대도 마찬가지입니다.

카테고리 C 의 반대(대칭) 카테고리(opposite category)를 C^{op} 이라 합니다. 우리는 이 카테고리에 대해 이중성(duality)을 논할 때 잠깐 다뤘습니다. C^{op} 의 객체(objects)는 C 의 객체들과 동일합니다.

C 에서 $f : a \rightarrow b$ 라는 화살표가 있을 때, C^{op} 에서는 그에 대응하는 화살표 $f^{op} : b \rightarrow a$ 가 있습니다.

두 화살표 $f^{op} : a \rightarrow b$ 와 $g^{op} : b \rightarrow c$ 의 합성 $g^{op} \circ f^{op}$ 은 화살표 $f \circ g$ 에 의해 주어집니다 (순서가 반대로 된 것을 주목하세요).

C 의 종말 대상(terminal object)은 C^{op} 의 시작 대상(initial object)이며, C 의 곱(product)은 C^{op} 의 합(sum)과 같고, 그 외에도 그렇습니다.

곱 카테고리들(Product categories)

두 범주 C 와 D 가 주어지면, 우리는 곱 범주 $C \times D$ 를 구성할 수 있습니다. 이 범주의 대상들은 대상들의 쌍 $\langle c, d \rangle$ 이며, 화살표들은 화살표들의 쌍입니다.

만약 C 에서의 사상(arrow) $f : c \rightarrow c'$ 과 D 에서의 사상 $g : d \rightarrow d'$ 가 주어지면 $C \times D$ 에서는 이들에 대응하는 사상 $\langle f, g \rangle$ 이 있습니다. 이 사상은 $C \times D$ 의 객체 $\langle c, d \rangle$ 에서 $\langle c', d' \rangle$ 로 갑니다. 이러한 두 사상은 각각 C 와 D 에서 그 성분들이 결합(composable)될 경우 합성(composed)될 수 있습니다. 항등 사상(identity arrow)은 항등 사상들의 쌍입니다.

우리가 가장 관심 있는 두 가지 곱(product) 범주(categories)는 $C \times C$ 와 $C^{op} \times C$ 입니다. 여기서 C 는 우리가 익숙한 타입과 함수의 범주(category)입니다.

이 두 범주(카테고리)의 경우, 객체는 C 에서 온 객체들의 쌍입니다. 첫 번째 범주인 $C \times C$ 에서는 $\langle a, b \rangle$ 에서 $\langle a', b' \rangle$ 로 가는 사상(모르피즘, morphism)은 $\langle f : a \rightarrow a', g : b \rightarrow b' \rangle$ 이라는 쌍입니다. 두 번째 범주인 $C^{op} \times C$ 에서는 사상(모르피즘, morphism)은 $\langle f : a' \rightarrow a, g : b \rightarrow b' \rangle$ 이라는 쌍입니다. 여기서 첫 번째 화살표(arrow)는 반대 방향으로 갑니다.

²다시 말해, "크기" 문제(특히 모든 집합들의 집합의 비존재)를 무시합니다.

슬라이스 카테고리(Slice categories)

잘 정돈된 우주에서는, 객체는 항상 객체이고 화살표는 항상 화살표입니다. 하지만 때때로 화살표의 집합을 객체로 생각할 수 있습니다. 그러나 분할 범주(slice categories)는 이 깔끔한 구분을 깨뜨립니다: 그들은 개별 화살표를 객체로 바꿉니다.

A slice category C/c 는 특정 객체 c 가 그의 범주 C 의 관점에서 어떻게 보이는지를 설명합니다. 이는 c 를 가리키는 모든 화살들의 총합입니다. 그러나 하나의 화살을 명시하기 위해 우리는 그 끝점을 모두를 명시해야 합니다. 이 끝점을 중 하나는 c 로 고정되어 있기 때문에, 우리는 다른 하나만 명시하면 됩니다.

쪼개진 범주 C/c (또는 오버 범주로도 알려져 있음) 내의 객체는 $p: e \rightarrow c$ 인 $\langle e, p \rangle$ 의 쌍입니다.

두 객체 $\langle e, p \rangle$ 및 $\langle e', p' \rangle$ 사이의 화살표는 다음 삼각형이 가환하게 만드는 C 의 화살표 $f: e \rightarrow e'$ 입니다:

$$\begin{array}{ccc} e & \xrightarrow{f} & e' \\ p \searrow & & \swarrow p' \\ & c & \end{array}$$

코슬라이스 카테고리(Coslice categories)

이중 개념의 코슬라이스 범주 c/C 가 존재합니다. 이는 언더-카테고리(under-category)라고도 알려져 있습니다. 이것은 고정된 객체 c 에서 나오는 화살표들의 범주입니다. 이 범주의 객체들은 $\langle a, i: c \rightarrow a \rangle$ 쌍들입니다. c/C 의 사상들은 관련된 삼각형을 가환시키는 화살표들입니다.

$$\begin{array}{ccc} & c & \\ i \swarrow & & \searrow j \\ a & \xrightarrow{f} & b \end{array}$$

특히, 범주(category) C 가 종말 객체(terminal object) 1을 가진다면, 코슬라이스(coslice) $1/C$ 는 C 의 모든 객체의 글로벌 원소(global elements)를 객체로 가집니다.

$1/C$ 의 사상(Morphisms)은 $f: a \rightarrow b$ 에 해당하며, a 의 전역 원소(global elements) 집합을 b 의 전역 원소 집합으로 매핑합니다.

$$\begin{array}{ccc} & 1 & \\ x \swarrow & & \searrow y \\ a & \xrightarrow{f} & b \end{array}$$

특히, 타입과 함수들의 범주(category)로부터 코슬라이스 범주(coslice category)를 구성하는 것은 타입을 값들의 집합으로 이해하는 우리의 직관을 정당화하며, 타입의 전역 원소로 값이 표현됩니다.

8.2 함자(Functors)

우리는 대수 데이터 유형을 논의할 때 함자성(functoriality) 예제를 보았습니다. 이러한 데이터 유형은 생성된 방식을 “기억”하며, 화살표를 그 “내용”에 적용하여 이 기억을 조작할 수 있습니다.

어떤 경우에는 이 직관이 매우 설득력 있습니다: 우리는 곱 유형(product type)을 그것의 구성 요소들을 “포함하는” 쌍으로 생각합니다. 결국, 투영(projection)을 사용하여 그들을 검색할 수 있습니다.

This is less obvious in the case of function objects. You can visualize a function object as secretly storing all possible results and using the function argument to index into them. A function from `Bool` is obviously equivalent to a pair of values, one for `True` and one for `False`. It's a known programming trick to implement some functions as lookup tables. It's called *memoization*(메모이제이션).

비록 자연수를 인수로 취하는 함수들을 메모이즈(memoize)하는 것이 실용적이지는 않지만; 우리는 여전히 이를(무한, 또는 심지어 셀 수 없는) 조회 테이블(lookup tables)로 개념화할 수 있습니다.

데이터 타입을 값들의 컨테이너(container of values)로 생각할 수 있다면, 이 값들을 변환하는 함수를 적용하여 변환된 컨테이너를 생성하는 것이 합리적입니다. 이것이 가능할 때, 우리는 그 데이터 타입이 함자적(functorial)이라고 말합니다.

다시 말해, 함수 타입(function types)은 더 많은 믿음의 보류를 요구합니다. 당신은 함수 객체(function object)를 어떤 타입으로 키(Key)된 조회 테이블(lookup table)로 시각화합니다. 만약 당신이 키로서 다른, 관련된 타입을 사용하고 싶다면, 새로운 키를 원래 키로 변환하는 함수를 필요로 합니다. 이것이 함수 객체의 함자성(functoriality)이 하나의 화살표를 반대로 가지는 이유입니다:

```
dimap :: (a' -> a) -> (b -> b') -> (a -> b) -> (a' -> b')
dimap f g h = g . h . f
```

content: 당신은 `a` 타입의 값을 받는 “수용체”를 가진 함수 `h :: a -> b`에 변환을 적용하려고 합니다, 그리고 `a'` 타입의 입력을 처리하는 데 사용하고자 합니다. 이는 `a'`에서 `a`로 변환하는 `f :: a' -> a` 컨버터가 있을 때만 가능합니다.

어떤 데이터 타입이 다른 타입의 값을 “포함”하는 아이디어는 하나의 데이터 타입이 다른 타입에 의해 파라미터화된다고 말할 수 있습니다. 예를 들어, 타입 `List a`는 타입 `a`에 의해 파라미터화되어 있습니다.

다시 말해, `List`는 타입 `a`를 타입 `List a`로 매핑합니다. 인자 없이 단독으로 사용하는 `List`는 타입 생성자(type constructor)라고 불립니다.

함자들(Functors) 사이의 범주들

범주 이론(category theory)에서, 타입 생성자(type constructor)는 객체들의 대응(mapping)으로 모델링 됩니다. 이는 객체들에 대한 함수입니다. 이는 범주의 구조의 일부인 객체들 간의 화살표(arrow)와 혼동해서는 안 됩니다.

사실, 카테고리 사이의 매핑을 상상하는 것이 더 쉽습니다. 출발 카테고리(source category)의 모든 객체(object)는 목표 카테고리(target category)의 객체로 매핑됩니다. 만약 a 가 C 의 객체라면, 해당하는 객체 Fa 가 D 에 있습니다.

함자적 사상(functorial mapping) 또는 함자(functor)는 객체(object)들 뿐만 아니라 그들 사이의 화살표(arrow)들도 매핑한답니다. 모든 화살표

$$f : a \rightarrow b$$

첫 번째 카테고리에 있는 화살표는 두 번째 카테고리에 대응되는 화살표를 가지고 있습니다:

$$Ff : Fa \rightarrow Fb$$

$$\begin{array}{ccc}
 a & \dashrightarrow & Fa \\
 \downarrow f & & \downarrow Ff \\
 b & \dashrightarrow & Fb
 \end{array}$$

우리는 객체의 대응(mapping)과 화살표의 대응(mapping) 모두에 대해 동일한 문자 F 를 사용합니다.

범주(카테고리)가 구조(structure)의 본질을 정제한 것이라면, 함자(функция)는 이 구조를 보존하는 사상입니다. 원천 범주(source category)에서 관련된 객체들은 목표 범주(target category)에서도 관련되어 있습니다.

범주의 구조는 화살표와 그것들의 합성(composition)에 의해 정의됩니다. 따라서 함자(functor)는 합성을 보존해야 합니다. 한 범주에서 합성된 것은:

$$h = g \circ f$$

두 번째 카테고리에서도 구성된 상태로 유지되어야 합니다.

$$Fh = F(g \circ f) = Fg \circ Ff$$

우리는 C 에서 두 화살표를 합성한 후 이를 D 로 매핑할 수 있거나, 개별 화살표들을 매핑한 후 D 에서 합성할 수 있습니다. 우리는 그 결과가 동일하다고 요구합니다.

$$\begin{array}{ccc}
 a & \dashrightarrow & Fa \\
 \downarrow f & & \downarrow Ff \\
 b & \dashrightarrow & Fb \\
 \downarrow g & & \downarrow Fg \\
 c & \dashrightarrow & Fc
 \end{array}$$

$g \circ f$ (blue) $F(g \circ f)$ (red) $Fg \circ Ff$ (black)

마지막으로, 함자(functor)는 항등 화살(identity arrows)을 보존해야 합니다:

$$F id_a = id_{Fa}$$

$$\begin{array}{ccc}
 id_a & & id_{Fa} \\
 \text{blue circle} & & \text{red circle} \\
 a & \dashrightarrow & Fa
 \end{array}$$

이 조건들을 종합하여 함자(functor)가 범주의 구조를 보존한다는 것이 무엇을 의미하는지 정의합니다.

아울러 어떤 조건들이 정의의 일부가 아님을 인식하는 것도 중요합니다. 예를 들어, 함자(functor)는 여러 객체를 동일한 객체로 매핑할 수 있습니다. 또한 끝점이 맞는 한 여러 화살표를 동일한 화살표로 매핑할 수도 있습니다.

극단적으로, 어떠한 범주(category)도 하나의 객체와 하나의 화살표(arrow)를 가진 단일체 범주.singleton category)로 매핑될 수 있습니다.

또한, 대상 범주(target category)의 모든 객체(object) 또는 화살표(arrow)가 함자(functor)에 의해 포함될 필요는 없습니다. 극단적으로, 단일 객체 범주.singleton category)로부터 어떠한(비어 있지 않은) 범주로도 함자가 존재할 수 있습니다. 이러한 함자(functor)는 하나의 객체(object)와 그것의 항등 화살표(identity arrow)를 선택합니다.

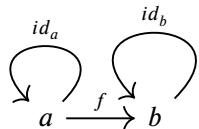
A 상수 함자(constant functor) Δ_c 는 원천 범주의 모든 객체를 대상 범주의 단일 객체 c 로 매핑하고, 원천 범주의 모든 화살표를 단일 관념 화살표(identity arrow) id_c 로 매핑하는 함자의 예입니다.

범주 이론에서 함자(functions)는 종종 하나의 범주를 다른 범주 안에 모델링하기 위해 사용됩니다. 여러 객체와 화살표를 하나로 병합할 수 있는 사실은 그것들이 원본 범주의 단순화된 뷰(view)를 생성함을 의미합니다. 그것들은 원본 범주의 일부 측면을 “추상화”합니다.

그들이 타겟 카테고리의 일부만 다룰 수 있다는 사실은 모델들이 더 큰 환경에 내재되어 있음을 의미합니다.

Functors(함자)는 어떤 최소한의 막대-그림, 범주들로부터 더 큰 범주들 속의 패턴들을 정의하는 데 사용할 수 있습니다.

Exercise 8.2.1. *Describe a functor whose source is the “walking arrow” category. It’s a stick-figure category with two objects and a single arrow between them (plus the mandatory identity arrows).*



Exercise 8.2.2. *The “walking iso” category is just like the “walking arrow” category, plus one more arrow going back from b to a . Show that a functor from this category always picks an isomorphism in the target category.*

8.3 프로그래밍에서의 함자(Functors)

Endofunctors는 프로그래밍 언어로 가장 쉽게 표현할 수 있는 함자(functions)의 종류입니다. 이러한 함수자는 카테고리(여기서는 타입과 함수들의 카테고리)를 자기 자신으로 매핑합니다.

Endofunctors(종단함자)

함수자(Endofunctor)의 첫 번째 부분은 타입을 타입으로 매핑하는 것입니다. 이는 타입 생성자(type constructor)들을 사용하여 이루어지며, 이들은 타입 레벨 함수입니다.

목록 타입 생성자인, `List`, 는 임의의 타입 `a`를 타입 `List a`로 매핑합니다.

The `Maybe` 타입 생성자(type constructor)는 `a`를 `Maybe a`로 매핑합니다.

Endofunctor의 두 번째 부분은 화살표를 매핑하는 것입니다. 주어진 함수 `a -> b`가 있을 때, 우리는 함수 `List a -> List b` 또는 `Maybe a -> Maybe b`를 정의할 수 있어야 합니다. 이것이 우리가 이전에 논의한 이 데이터 타입의 “함자성(functoriality)” 속성입니다. 함자성(functoriality)은 임의의 함수를 변환된 타입 간의 함수로 옮기는(*lift*) 것을 가능하게 합니다.

함자성을 Haskell에서 타입 클래스(typeclass)를 사용하여 표현할 수 있습니다. 이 경우, 타입 클래스는 타입 생성자 `f`에 의해 매개변수화 됩니다 (Haskell에서는 타입 생성자 변수에 소문자 이름을 사용합니다). 우리는 적절한 대응 함수의 매핑인 `fmap`이라는 함수가 있으면 `f`가 `Functor`(함자)라 말합니다:

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

컴파일러는 `f`가 타입 생성자(type constructor)라는 것을 압니다. 왜냐하면 그것이 `f a`와 `f b`에서와 같이 타입에 적용되기 때문입니다.

컴파일러에게 특정 타입 생성자가 **Functor**임을 증명하기 위해, 우리는 그에 대한 **fmap**의 구현을 제공해야 합니다. 이는 타입클래스 **Functor**의 인스턴스(instance)를 정의함으로써 가능합니다. 예를 들어:

```
instance Functor Maybe where
  fmap g Nothing = Nothing
  fmap g (Just a) = Just (g a)
```

평터는 몇 가지 법칙을 만족해야 합니다: 이를 합성(composition)과 항등성(identity)을 보존해야 합니다. 이러한 법칙은 Haskell에서 표현할 수 없지만, 프로그래머가 확인해야 합니다. 이전에 항등성 법칙을 만족하지 않은 **badMap**의 정의를 보았으나, 컴파일러에 의해 허용될 것입니다. 이는 리스트 타입 생성자 **[]**에 대한 “불법적인” **Functor** 인스턴스를 정의할 것입니다.

Exercise 8.3.1. Show that **WithInt** is a functor

```
data WithInt a = WithInt a Int
```

어떤 기본적인 함자(함자, functors)들이 있을 수 있는데, 이들은 사소해 보일지 몰라도 다른 함자들의 구성 요소로서 작용합니다.

우리는 모든 객체와 모든 사상을 자기 자신으로 매핑하는 항등 자기함자(endofunctor)를 가지고 있습니다.

```
data Id a = Id a
```

Exercise 8.3.2. Show that **Id** is a **Functor**. Hint: implement the **Functor** instance for it.

우리는 또한 모든 객체들을 단일 객체 c 로 매핑하고, 모든 화살(arrows)들을 이 객체의 항등 화살(identity arrow)로 매핑하는 상수 함자(constant functor) Δ_c 를 가지고 있습니다. Haskell에서는, 이는 대상 객체 c 에 의해 매개변수화된 함자들의 집합(family of functors)입니다:

```
data Const c a = Const c
```

이 타입 생성자는 두 번째 인수를 무시합니다.

Exercise 8.3.3. Show that $(\text{Const } c)$ is a **Functor**. Hint: The type constructor takes two arguments, but here it's partially applied to the first argument. It is functorial in the second argument.

이항 함자(Bifunctors)

데이터 생성자가 두 개의 타입을 인자로 받는 경우도 보았습니다: 곱(product)과 합(sum)입니다. 이들은 또한 함자적(functorial)입니다, 하지만 단일 함수를 올리는 대신, 함수 쌍을 올렸습니다. 범주론(category theory)에서, 이것을 $C \times C$ 범주(product category)에서 C 범주로의 함자(functors)로 정의합니다.

이러한 함자(functor)는 객체의 쌍을 객체로, 화살표(arrow)의 쌍을 화살표로 매핑합니다.

하스켈(Haskell)에서는 이러한 평터(functor)들을 **Bifunctor**라고 불리는 별도의 클래스의 구성원으로 취급합니다.

```
class Bifunctor f where
  bimap :: (a -> a') -> (b -> b') -> (f a b -> f a' b')
```

다시, 컴파일러는 **f**가 두 인수 타입 생성자(type constructor)라고 추론합니다. 왜냐하면 그것이 두 타입에 적용된 것을 보았기 때문입니다. 예를 들어 **f a b**.

컴파일러에게 특정 타입 생성자가 **Bifunctor**임을 증명하기 위해 우리는 인스턴스를 정의합니다. 예를 들어, 쌍(pair)의 bifunctionality는 다음과 같이 정의될 수 있습니다:

```
instance Bifunctor (,) where
  bimap g h (a, b) = (g a, h b)
```

Exercise 8.3.4. Show that *MoreThanA* is a bifunctor.

```
data MoreThanA a b = More a (Maybe b)
```

반변 함자(Contravariant functors)

반대 범주(opposite category) C^{op} 에서의 함수자는 반변함자(contravariant)라고 합니다. 이들은 반대 방향으로 가는 화살표를 올리는 성질을 가지고 있습니다. 일반적인 함수자는 때때로 공변함자(covariant)라고 불립니다.

In Haskell, 반변 함자(contravariant functors)들은 타입클래스 **Contravariant**를 구성합니다:

```
class Contravariant f where
  contramap :: (b -> a) -> (f a -> f b)
```

자주 functor(함자)를 생산자와 소비자의 관점에서 생각하는 것이 편리합니다. 이 그림에서, (공변) functor(함자)는 생산자입니다. 함수 $a \rightarrow b$ 를 적용하여 (**fmap**을 사용하여) a 의 생산자를 b 의 생산자로 바꿀 수 있습니다. 반대로, a 의 소비자를 b 의 소비자로 바꾸기 위해선 반대 방향으로 가는 함수, 즉 $b \rightarrow a$ 가 필요합니다.

예시: 술어(predicate)는 **True** 또는 **False** 값을 반환하는 함수입니다:

```
data Predicate a = Predicate (a -> Bool)
```

It's easy to see that it's a contravariant functor(반변 함자) :

```
instance Contravariant Predicate where
  contramap f (Predicate h) = Predicate (h . f)
```

유일하게 비자명한 예시들의 반공변 함자(contravariant functors)는 함수 객체(theme of function objects)의 변형입니다.

어떤 주어진 함수 타입이 타입 인자 중 하나에서 공변적(covariant)인지 반공변적(contravariant)인지 알아보는 한 가지 방법은 정의에 사용된 타입들에 극성을 부여하는 것입니다. 함수의 반환 타입은 양수 위치에 있으므로 공변적입니다; 그리고 인자 타입은 음수 위치에 있으므로 반공변적입니다. 그러나 전체 함수 객체를 다른 함수의 음수 위치에 놓으면, 그 극성은 뒤바집니다.

이 데이터 타입을 고려해 보세요:

```
data Tester a = Tester ((a -> Bool) -> Bool)
```

이것은 이중 부정적(negative) 위치에 a 를 갖고 있으므로 긍정적(positive) 위치에 있는 것입니다. 이것이 공변 **함자(Functor)**인 이유입니다. 이것은 a 의 생성자(producer)입니다:

```
instance Functor Tester where
  fmap f (Tester g) = Tester (g . f)
```

괄호가 여기서 중요함을 주목하세요. 비슷한 함수 $a \rightarrow \text{Bool} \rightarrow \text{Bool}$ 은 a 가 음수의 위치에 있습니다. 이것은 a 의 함수가 ($\text{Bool} \rightarrow \text{Bool}$) 함수로 반환되기 때문입니다. 동등하게, 이것을 커리-해제(uncurry) 해서 쌍을 받는 함수로 만들 수 있습니다: $(a, \text{Bool}) \rightarrow \text{Bool}$. 어떠한 경우든 a 는 음수 위치에 있습니다.

Profunctors (프로펑터)

우리는 이전에 함수형이 함수적(functorial)이라는 것을 본 적이 있습니다. 이것은 **Bifunctor** 와 마찬가지로 한 번에 두 개의 함수를 들어올리는데, 단 하나의 함수는 반대 방향으로 갑니다.

범주 이론에서 이는 두 범주의 곱의 함자(functor)에 해당합니다. 여기서 하나는 반대 범주 (opposite category)입니다: 이는 $\mathcal{C}^{op} \times \mathcal{C}$ 부터의 함자입니다. $\mathcal{C}^{op} \times \mathcal{C}$ 로부터 **Set**까지의 함자를 *profunctors*(프로펑터)라고 합니다.

Haskell에서, 프로펑터들(profunctors)은 하나의 타입클래스를 형성합니다:

```
class Profunctor f where
    dimap :: (a' -> a) -> (b -> b') -> (f a b -> f a' b')
```

프로펑터(profunctor)는 동시에 생산자이자 소비자인 유형(type)이라고 생각할 수 있습니다. 하나의 유형을 소비하고 다른 유형을 생산합니다.

함수 타입(function type)은 **Profunctor**의 인스턴스(instance)로서 (\rightarrow)로 표기될 수 있습니다.

```
instance Profunctor ( -> ) where
    dimap f g h = g . h . f
```

이는 함수 $a \rightarrow b$ 가 a 타입의 인수를 소모하고 b 타입의 결과를 생성한다는 우리의 직관과 일치합니다.

프로그래밍에서, 모든 비자명한 프로펑터(profunctors)는 함수 타입의 변형입니다.

8.4 Hom-함수자(Hom-Functor)

두 객체 사이의 모든 화살표는 집합을 이룹니다. 이 집합은 hom-집합(hom-set)이라고 하며 보통 범주(category)의 이름 뒤에 객체의 이름을 붙여서 씁니다:

$$\mathcal{C}(a, b)$$

우리는 hom-집합 $\mathcal{C}(a, b)$ 을 a 로부터 b 를 관찰할 수 있는 모든 방법으로 해석할 수 있습니다.

Another way of looking at hom-sets는 hom-sets(사상집합)을 바라보는 또 다른 방법은 to say that they define a mapping 그것들이 매핑(mapping)을 정의한다는 것입니다 that assigns a set $\mathcal{C}(a, b)$ 집합 $\mathcal{C}(a, b)$ 을 to every pair of objects 모든 객체 쌍에 할당하는. Sets themselves 집합 자체는 are objects in the category **Set** 범주 **Set**의 객체입니다. So we have a mapping between categories 그래서 우리는 범주들 간의 매핑(mapping)을 가지고 있습니다.

이 사상(mapping)은 함자적(functorial)입니다. 이를 보기 위해, 두 객체 a 와 b 를 변환할 때 발생하는 일을 고려합시다. 우리는 집합 $\mathcal{C}(a, b)$ 를 집합 $\mathcal{C}(a', b')$ 로 사상하는 변환에 관심이 있습니다. **Set** 내의 화살표(arrows)는 일반적인 함수들이므로 집합의 개별 요소에 대한 그들의 작용을 정의하는 것으로 충분합니다.

$\mathcal{C}(a, b)$ 의 한 요소는 화살표 $h: a \rightarrow b$ 이며, $\mathcal{C}(a', b')$ 의 한 요소는 화살표 $h': a' \rightarrow b'$ 입니다. 하나를 다른 하나로 변환하는 방법은 알고 있습니다: 우리는 h 를 화살표 $g': a' \rightarrow a$ 와 함께 사전 조합(pre-compose)하고 화살표 $g: b \rightarrow b'$ 와 함께 사후 조합(post-compose)해야 합니다.

다른 말로, $\langle a, b \rangle$ 쌍을 집합 $\mathcal{C}(a, b)$ 로 대응시키는 사상(mapping)은 **프로펑터**(profunctor)입니다.

$$\mathcal{C}^{op} \times \mathcal{C} \rightarrow \mathbf{Set}$$

자주 우리는 하나의 객체만 변경하고 다른 하나는 고정하는 것에 관심이 있습니다. 소스 객체를 고정하고 타겟을 변경하면 결과는 다음과 같은 평터(functor)가 됩니다:

$$\mathcal{C}(a, -) : \mathcal{C} \rightarrow \mathbf{Set}$$

이 함자의 화살표 $g : b \rightarrow b'$ 에 대한 작용은 다음과 같이 쓴다:

$$\mathcal{C}(a, g) : \mathcal{C}(a, b) \rightarrow \mathcal{C}(a, b')$$

content: 그리고 이는 다음과 같이 주어진다: 후적합(후콤포지션, post-composition)으로:

$$\mathcal{C}(a, g) = (g \circ -)$$

b 를 변화시킨다는 것은 하나의 대상에서 다른 대상으로 초점을 전환하는 것을 의미하므로, 완비 함자 $\mathcal{C}(a, -)$ 는 a 에서 나오는 모든 화살표들을 a 의 관점에서의 범주에 대한 일관된 관점으로 결합합니다. 이는 "a에 따른 세계"입니다.

반대로, 타겟(target)을 고정하고 hom-평터(hom-functor)의 소스(source)를 변화시키면, 반변 평터(contravariant functor)를 얻습니다:

$$\mathcal{C}(-, b) : \mathcal{C}^{op} \rightarrow \mathbf{Set}$$

그 작용이 화살표 $g' : a' \rightarrow a$ 에 대해 다음과 같이 쓰여집니다:

$$\mathcal{C}(g', b) : \mathcal{C}(a, b) \rightarrow \mathcal{C}(a', b)$$

앞서 이루어진 사전 구성에 의해 제공됩니다:

$$\mathcal{C}(g', b) = (- \circ g')$$

함자 $\mathcal{C}(-, b)$ 는 모든 화살표를 b 를 가리키는 방향으로 일관성 있게 조작합니다. 이는 b 가 "세상에 의해 바라본 모습"입니다.

우리는 이제 서로 동형사상들에 관한 장에서의 결과들을 다시 정리할 수 있습니다. 만약 두 개체 a 와 b 가 동형사상(isomorphic)이라면, 그들의 hom-집합(hom-sets) 또한 동형사상입니다. 특히:

$$\mathcal{C}(a, x) \cong \mathcal{C}(b, x)$$

content: 그리고

$$\mathcal{C}(x, a) \cong \mathcal{C}(x, b)$$

다음 장에서 자연성 조건(naturality conditions)에 대해 논의하겠습니다.

Another way of looking at the hom-functor $\mathcal{C}(a, -)$ is as an oracle that provides answers to the question: "Is a connected to me?" If the set $\mathcal{C}(a, x)$ is empty, the answer is negative: " a 는 x 와 연결되지 않았습니다." Otherwise, every element of the set $\mathcal{C}(a, x)$ is a proof that such connection exists.

반대로, 반변 함자(contravariant functor) $\mathcal{C}(-, a)$ 는 "나는 a 에 연결되어 있습니까?"라는 질문에 답합니다.

함께 고려할 때, 프로펑터(profunctor) $\mathcal{C}(x, y)$ 는 객체들 사이에 증거-관련된(proof-relevant) 관계를 설정합니다. 집합 $\mathcal{C}(x, y)$ 의 모든 요소는 x 가 y 에 연결되어 있다는 증거입니다. 집합이 비어 있다면, 두 객체는 관련이 없는 것입니다.

8.5 함자 합성(Functor Composition)

마치 함수들을 합성할 수 있는 것처럼, 평터(functor)들도 합성할 수 있습니다. 두 평터가 합성 가능하려면 하나의 타겟(target) 범주와 다른 하나의 소스(source) 범주가 같아야 합니다.

대상들(objects)에 대해, 함자(functor) 합성(composition) G 를 F 뒤에 적용하는 것은 먼저 대상에 F 를 적용한 후, 그 결과에 G 를 적용합니다; 화살표들(arrows)도 마찬가지입니다.

당연히 합성 가능한 함자들만 합성할 수 있습니다. 그러나 모든 내적 함자(endofunctors)는 합성 가능하니, 그 대상 범주가 출발 범주와 동일하기 때문입니다.

Haskell에서 함자(functor)는 매개변수가 있는 데이터 타입(parameterized data type)입니다, 그래서 두 함자의 합성(composition)은 다시 매개변수가 있는 데이터 타입입니다. 대상(objects)에서는 다음과 같이 정의합니다:

```
data Compose g f a = Compose (g (f a))
```

컴파일러가 f 와 g 가 타입 생성자(type constructors)임을 알아차립니다. 이는 이들이 타입에 적용되기 때문입니다: f 는 타입 매개변수(type parameter) a 에 적용되고, g 는 그 결과 타입에 적용됩니다.

Alternatively, you can tell the compiler that the first two arguments to `Compose` are type constructors. You do this by providing a *kind signature*, which requires a language extension `KindSignatures` that you put at the top of the source file:

대신, `Compose`의 처음 두 인자가 타입 생성자(type constructors)라는 것을 컴파일러에게 알려줄 수 있습니다. 이 작업은 *kind signature*(종류 서명)를 제공함으로써 이루어지며, 이는 소스 파일의 맨 위에 `KindSignatures`라는 언어 확장을 필요로 합니다:

```
{-# language KindSignatures #-}
```

다음의 `Data.Kind` 라이브러리도 가져와야 합니다. 이 라이브러리는 `Type`을 정의합니다:

```
import Data.Kind
```

종류 서명(kind signature)은 타입 서명(type signature)과 비슷하지만, 타입에서 작동하는 함수를 설명하는 데 사용될 수 있습니다.

정규 타입은 kind `Type`을 가지고나. 타입 생성자는 `Type -> Type`을 가지는데, 이는 타입을 타입으로 매핑하기 때문입니다.

`Compose`는 두 개의 타입 생성자(type constructors)를 받아서 하나의 타입 생성자를 생성하므로, 그 종류 서명(kind signature)은:

```
(Type -> Type) -> (Type -> Type) -> (Type -> Type)
```

and the full definition is:

```
data Compose :: (Type -> Type) -> (Type -> Type) -> (Type -> Type)
  where
    Compose :: (g (f a)) -> Compose g f a
```

어떠한 두 타입 생성자(type constructors)도 이 방식으로 합성될 수 있습니다. 이 시점에서, 그것들이 평터(functors)일 필요는 없습니다.

그러나, 타입 생성자의 합성을 사용하여 함수 g 를 f 뒤에 들어올리고 싶다면, 이들은 평터(functor)이어야 합니다. 이 요구사항은 인스턴스 선언에서 제약조건으로 인코딩됩니다:

```
instance (Functor g, Functor f) => Functor (Compose g f) where
  fmap h (Compose gfa) = Compose (fmap (fmap h) gfa)
```

제약(Functor g, Functor f)는 두 타입 생성자가 Functor 클래스의 인스턴스란 조건을 표현합니다. 제약 뒤에는 더블 애로우가 옵니다.

우리가 함수자 속성을 증명하고 있는 타입 생성자는 **Compose** $f \circ g$ 입니다. 이는 두 함수자에 부분적으로 적용된 **Compose**입니다.

fmap의 구현에서, 우리는 데이터 생성자 **Compose**에 대해 패턴 매치를 합니다. 그 인자인 **gfa**는 타입 $g(f a)$ 입니다. 우리는 하나의 **fmap**를 사용하여 g 의 "하부로 들어갑니다". 그런 다음 ($fmap h$)를 사용하여 f 의 하부로 들어갑니다. 컴파일러는 타입을 분석하여 어떤 **fmap**를 사용할지 결정합니다.

한 쪽 합성 함수를 컨테이너들의 컨테이너로 시각화할 수 있습니다. 예를 들어, **[]**와 **Maybe**의 합성은 선택적(옵셔널) 값들의 리스트입니다.

Exercise 8.5.1. Define a composition of a **Functor** after **Contravariant**. Hint: You can reuse **Compose**, but you have to provide a different instance declaration.

카테고리들의 카테고리

함자(functor)들을 범주(category)들 간의 화살로 볼 수 있습니다. 방금 본 것처럼, 함자들은 합성될 수 있으며, 이 합성이 결합법칙(associativity)을 만족하는지 확인하는 것은 쉽습니다. 또한, 모든 범주마다 항등(endo-) 함자가 존재합니다. 따라서 범주들 자체가 범주를 형성하는 것처럼 보이며, 이를 **Cat** 라고 부릅니다.

그리고 여기서 수학자들은 "크기" 문제에 대해 걱정하기 시작합니다. 이는 모순이 도사리고 있다는 것을 말하는 약식 표현입니다. 그래서 올바른 표현은 **Cat**가 작은 범주(category)의 범주(category)라는 것입니다. 하지만 존재의 증명을 시도하지 않는 한, 크기 문제는 무시해도 됩니다.