

Chapter 6

함수 타입(Function Types)

다른 종류의 합성(composition)이 있는데, 이는 함수형 프로그래밍(functional programming)의 핵심입니다. 이는 하나의 함수를 다른 함수의 인자로 전달할 때 발생합니다. 외부 함수는 이 인자를 자신의 기계 장치(mechanism)의 플러그 가능한 부분으로 사용할 수 있습니다. 이를 통해 임의의 비교 함수(comparison function)를 받아들이는 일반적인 정렬 알고리즘(sorting algorithm)을 구현할 수 있습니다.

만약 우리가 함수들을 객체들 사이의 화살표(arrows)로 모델링한다면, 함수가 인수(argument)로 있다는 것은 무엇을 의미할까요?

함수들을 객체화하는 방법이 필요합니다. 왜냐하면 화살표(arrow)들의 대상으로 “화살표의 객체(object of arrows)”가 되는 것을 정의하기 위해서입니다. 함수를 인자로 받거나 함수를 반환하는 함수는 고차 함수(higher-order function)라고 부릅니다. 고차 함수들은 함수형 프로그래밍의 일꾼들입니다.

제거 규칙(Elimination rule)

함수의 정의 특징은 인수를 적용하여 결과를 생성할 수 있다는 것입니다. 우리는 구성을 통해 함수 적용을 정의했습니다:

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

여기서 f 는 a 에서 b 로 가는 화살표로 표현되었지만, 우리는 f 를 화살표 객체의 요소로 대체하거나, 수학자들이 말하는 지수 객체 b^a 로 대체하고 싶습니다; 또는 프로그래밍에서 함수형 $a \rightarrow b$ 이라고 부릅니다.

b^a 의 한 요소와 a 의 한 요소가 주어지면 함수 적용(function application)은 b 의 한 요소를 생성해야 합니다. 다시 말해, 한 쌍의 요소가 주어지면:

$$\begin{aligned} f : 1 &\rightarrow b^a \\ x : 1 &\rightarrow a \end{aligned}$$

그것은 하나의 요소(element)를 생성해야 합니다.

$$y : 1 \rightarrow b$$

여기서 f 는 b^a 의 요소를 나타냅니다. 이전에는 a 에서 b 로 가는 화살표였습니다.

우리는 (f, x) 쌍의 요소가 곱셈 $b^a \times a$ 의 요소와 동등함을 알고 있습니다. 따라서 함수 적용을 하나의 화살표로 정의할 수 있습니다:

$$\varepsilon_{ab} : b^a \times a \rightarrow b$$

이 방식으로 y , 적용의 결과가 아래의 가환 도표에 의해 정의됩니다:

$$\begin{array}{ccc} & 1 & \\ & \downarrow (f, x) & \searrow y \\ b^a \times a & \xrightarrow{\varepsilon_{ab}} & b \end{array}$$

함수 적용은 함수 타입을 위한 제거 규칙(elimination rule)입니다.

누군가가 당신에게 함수 객체의 원소를 주면, 할 수 있는 유일한 일은 그것을 ε 을 사용해 인수 타입의 원소에 적용하는 것입니다.

소개 규칙(Introduction rule)

함수 객체(function object)의 정의를 완성하기 위해, 우리는 또한 도입 규칙(introduction rule)이 필요합니다.

먼저, 어떤 다른 객체 c 로부터 함수 객체 b^a 를 구성하는 방법이 있다고 가정하겠습니다. 이는 화살표가 있다는 것을 의미합니다

$$h : c \rightarrow b^a$$

우리는 ε_{ab} 을 사용하여 h 의 결과를 제거할 수 있다는 것을 알고 있지만, 먼저 a 를 곱해야 합니다. 따라서 먼저 c 에 a 를 곱한 다음 함자성을 사용하여 이를 $b^a \times a$ 로 매핑합시다.

함자성(Functoriality)은 우리가 곱(product)에 화살표쌍을 적용하여 또 다른 곱을 얻을 수 있게 해줍니다. 여기서 화살표쌍은 (h, id_a) 입니다 (우리는 c 를 b^a 로 변환하려고 하지만, a 를 수정하는 것에는 관심이 없습니다).

$$c \times a \xrightarrow{h \times id_a} b^a \times a$$

우리는 이제 함수 적용을 따라가 b 에 도달할 수 있습니다.

$$c \times a \xrightarrow{h \times id_a} b^a \times a \xrightarrow{\varepsilon_{ab}} b$$

이 합성 화살표(composite arrow)는 우리가 f 라고 부를 매핑(mapping)을 정의합니다:

$$f : c \times a \rightarrow b$$

다음은 대응 다이어그램입니다.

$$\begin{array}{ccc} c \times a & & \\ \downarrow h \times id_a & \searrow f & \\ b^a \times a & \xrightarrow{\varepsilon} & b \end{array}$$

이 교환 도표(commuting diagram)는, 주어진 h 가 있을 때, f 를 구성할 수 있음을 알려줍니다; 또한 우리는 역도 요구할 수 있습니다: 모든 출발하는 사상(mapping out), $f : c \times a \rightarrow b$ 는 지수함수로의 사상(mapping into the exponential), $h : c \rightarrow b^a$ 를 유일하게 정의해야 합니다.

이 속성을 사용할 수 있습니다. 화살표 두 집합 사이의 일대일 대응을 사용하여 지수 객체를 정의할 수 있습니다. 이것은 함수 객체 b^a 에 대한 도입 규칙입니다.

우리는 곱(product)이 그 사상-인(mapping-in) 속성을 사용하여 정의된 것을 보았습니다. 반면에 함수 적용(function application)은 곱의 사상-아웃(mapping out)으로 정의됩니다.

커링(Currying)

이 정의를 보는 여러 가지 방법이 있습니다. 하나는 이를 커링(currying)의 예로 보는 것입니다.

지금까지 우리는 하나의 인수를 갖는 함수만을 고려해 왔습니다. 이것은 실제 한계가 아닙니다. 왜냐하면 우리는 항상 두 개의 인수를 갖는 함수를 하나의 인수만을 갖는 함수로 구현할 수 있기 때문입니다. 함수 객체(function object)의 정의에서 f 는 그러한 함수입니다:

```
f :: (c, a) -> b
```

h 반면에 함수(function)를 반환하는 함수(function)입니다:

```
h :: c -> (a -> b)
```

Currying은 화살표 집합 사이의 동형사상(isomorphism)입니다.

이 동형사상(isomorphism)은 한 쌍의 (고차) 함수들로 Haskell에서 표현될 수 있습니다. Haskell에서는 커링(currying)이 어떤 타입에도 적용될 수 있기 때문에, 이 함수들은 타입 변수들을 사용하여 작성됩니다. 다시 말해, 이 함수들은 다형성(polymorphic)를 가집니다:

```
curry :: ((c, a) -> b) -> (c -> (a -> b))
```

```
uncurry :: (c -> (a -> b)) -> ((c, a) -> b)
```

다시 말해, 함수 객체의 정의에 있는 h 는 다음과 같이 쓸 수 있습니다

$$h = \text{curry } f$$

물론, 이렇게 작성할 경우, `curry`와 `uncurry`의 타입(types)은 화살표(arrow)보다는 함수 객체(function objects)에 해당됩니다. 이 구분은 일반적으로 신경 쓰지 않는데, 이는 지수(exponential)의 원소(elements)와 그것을 정의하는 화살표(arrow) 간의 일대일 대응이 있기 때문입니다. 이는 임의의 객체 c 를 말단 객체(terminal object)로 대체할 때 쉽게 이해할 수 있습니다. 우리는 다음과 같이 얻습니다:

$$\begin{array}{ccc} 1 \times a & & \\ \downarrow h \times id_a & \searrow f & \\ b^a \times a & \xrightarrow{\epsilon_{ab}} & b \end{array}$$

이 경우, h 는 객체(object) b^a 의 요소이고, f 는 $1 \times a$ 에서 b 로 가는 화살표(arrow)입니다. 그러나 $1 \times a$ 는 a 와 동형(isomorphic)임을 알고 있으므로, f 는 사실상 a 에서 b 로 가는 화살표입니다.

따라서, 이제부터 화살표 \rightarrow 를 화살표 \rightarrow 라고 부르겠습니다. 이에 대해 더 이상 신경 쓰지 않겠습니다. 이러한 현상에 대한 정확한 표현은 카테고리가 자기-강화(self-enriched)되었다고 말하는 것입니다.

ϵ_{ab} 를 Haskell 함수 `apply`로 작성할 수 있습니다:

```
apply :: (a -> b, a) -> b
apply (f, x) = f x
```

하지만 이것은 단지 구문적인 요령일 뿐입니다: 함수 적용(function application)은 언어에 내장되어 있습니다: f x 는 f 가 x 에 적용됨을 의미합니다. 다른 프로그래밍 언어는 함수의 인수(arguments)가 괄호로 둘러싸여 있어야 하지만, Haskell에서는 그렇지 않습니다.

비록 함수 적용을 별도의 함수로 정의하는 것이 중복적으로 보일 수 있지만, Haskell 라이브러리는 이를 위한 접두사 연산자 \$(달러 기호)를 제공합니다:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

그러나 요령은 정규 함수 적용은 왼쪽으로 결합되지만, 예를 들어, `f x y`는 $(f x) y$ 와 동일합니다; 그러나 달리 기호는 오른쪽으로 결합되기 때문에

```
f $ g x
```

는 `f (g x)`와 동일합니다. 첫 번째 예제에서 `f`는 (최소한) 두 개의 인수를 가지는 함수여야 합니다; 두 번째 예제에서는 하나의 인수를 가지는 함수일 수 있습니다.

Haskell에서 커링(currying)은 어디에서나 보편적입니다. 두 인자의 함수는 거의 항상 함수를 반환하는 함수로 작성됩니다. 함수 화살표 `->`가 오른쪽으로 끊이기 때문에 이러한 타입을 팔호로 끊을 필요가 없습니다. 예를 들어, 쌍 생성자는 다음과 같은 시그니처를 갖습니다:

```
pair :: a -> b -> (a, b)
```

이를 두 개의 인수를 받아 쌍을 반환하는 함수나, 하나의 인수를 받아 하나의 인수를 반환하는 함수라고 생각할 수 있습니다, `b->(a, b)`. 이렇게 하면 그러한 함수를 부분적으로 적용해도 괜찮고, 결과는 또 다른 함수가 됩니다. 예를 들어, 우리는 다음과 같이 정의할 수 있습니다:

```
pairWithTen :: a -> (Int, a)
pairWithTen = pair 10 -- partial application of pair
```

람다 계산법(lambda calculus)과의 관계

함수를 객체로 정의하는 또 다른 방법은 c 를 f 가 정의된 환경의 타입으로 해석하는 것입니다. 그런 경우 환경을 Γ 로 부르는 것이 일반적입니다. 화살표는 Γ 에 정의된 변수를 사용하는 표현으로 해석됩니다.

단순한 예제, 표현식을 고려해 봅시다:

$$ax^2 + bx + c$$

이를 실수 세 수 (a, b, c) 의 삼중항과 변수 x 로 매개변수화된 것으로 생각할 수 있습니다, 여기서 변수 x 는 복소수라고 가정합시다. 이 삼중항은 곱집합 $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ 의 한 요소입니다. 이 곱집합은 우리 표현식의 환경 Γ 입니다.

변수 x 는 \mathbb{C} 의 원소입니다. 이 표현은 제품 $\Gamma \times \mathbb{C}$ 에서 결과 타입(여기서는 또한 \mathbb{C} 입니다)으로의 화살표입니다.

$$f : \Gamma \times \mathbb{C} \rightarrow \mathbb{C}$$

이것은 곱(product)으로부터의 매핑이므로, 함수 객체(function object) $\mathbb{C}^{\mathbb{C}}$ 를 구성하기 위해 이를 사용할 수 있으며 매핑 $h : \Gamma \rightarrow \mathbb{C}^{\mathbb{C}}$ 를 정의할 수 있습니다.

$$\begin{array}{ccc} \Gamma \times \mathbb{C} & & \\ h \times id_{\mathbb{C}} \downarrow & \searrow f & \\ \mathbb{C}^{\mathbb{C}} \times \mathbb{C} & \xrightarrow{\epsilon} & \mathbb{C} \end{array}$$

이 새로운 매핑 h 는 함수 객체의 생성자로 볼 수 있습니다. 결과적인 함수 객체는 \mathbb{C} 에서 \mathbb{C} 로의 모든 함수들을 나타내며, 이는 환경 Γ 에 접근할 수 있습니다. 즉, 파라미터의 삼중 (a, b, c) 에 접근할 수 있습니다.

우리의 원래 표현 $ax^2 + bx + c$ 에 해당하는 $\mathbb{C}^{\mathbb{C}}$ 안에는 다음과 같이 쓸 수 있는 특정 함수가 있습니다:

$$\lambda x. ax^2 + bx + c$$

또는, 하스켈(Haskell)에서는 λ 를 대체하는 백슬래시(backslash)를 사용하여,

```
\x -> a * x^2 + b * x + c
```

화살표 $h: \Gamma \rightarrow \mathbb{C}^a$ 는 화살표 f 에 의해 고유하게 결정됩니다. 이 매핑은 우리가 $\lambda x. f$ 라고 부르는 함수를 생성합니다.

일반적으로, 함수 객체(function object)에 대한 정의 다이어그램은 다음과 같습니다:

$$\begin{array}{ccc} \Gamma \times a & & \\ h \times id_a \downarrow & \searrow f & \\ b^a \times a & \xrightarrow{\epsilon} & b \end{array}$$

환경 Γ 는 표현식 f 에 대해 자유 파라미터(프리 파라미터)들을 제공하며, 이는 파라미터들의 타입을 나타내는 여러 객체들의 곱이다(우리의 예시에서는, 그것이 $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ 이었다).

빈 환경은 곱의 단위인 종단 객체 1에 의해 표현됩니다. 이 경우 f 는 단순히 화살표 $a \rightarrow b$ 이며, h 는 f 에 대응하는 함수 객체 b^a 에서 원소를 선택합니다.

일반적으로 함수 객체는 외부 매개변수에 의존하는 함수를 나타낸다는 점을 염두에 두는 것이 중요합니다. 이러한 함수는 클로저(closures)라고 불립니다. 클로저는 환경으로부터 값을 포착하는 함수입니다.

다음은 Haskell로 번역된 우리의 예시입니다. f 에 대응하는 표현식이 있습니다:

```
(a :+ 0) * x * x + (b :+ 0) * x + (c :+ 0)
```

만약 **Double**을(를) 사용하여 \mathbb{R} 을(를) 근사한다면, 우리의 환경은 (**Double**, **Double**, **Double**)의 곱(product)입니다. **Complex** 타입은 다른 타입에 의해 매개변수화됩니다(parameterized)—여기서 우리는 다시 **Double**을(를) 사용했습니다:

```
type C = Complex Double
```

Double에서 **C**로의 변환은 허수 부분을 0으로 설정하여 수행됩니다. 예를 들어 $(a :+ 0)$ 와 같이 합니다.

대응하는 화살표 h 는 환경을 받아 **C** \rightarrow **C** 타입의 클로저를 생성합니다:

```
h :: (Double, Double, Double) -> (C -> C)
h (a, b, c) = \x -> (a :+ 0) * x * x + (b :+ 0) * x + (c :+ 0)
```

Modus ponens (모두스 포넨스)

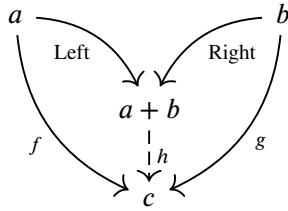
논리학에서 함수 객체(function object)는 함의(implication)에 해당합니다. 최종 객체(terminal object)에서 함수 객체로의 화살표는 그 함의의 증명입니다. 함수 적용 ϵ 은 논리학자들이 *modus ponens*라고 부르는 것에 해당합니다: 만약 $A \Rightarrow B$ 의 증명과 A 의 증명을 가지고 있다면 이는 B 의 증명을 구성합니다.

6.1 합과 곱 재검토

함수들이 다른 유형의 요소들과 동일한 지위를 가지게 되면, 다이어그램(diagram)을 코드로 직접 변환할 수 있는 도구들을 가지게 됩니다.

합 타입(Sum types)

합의 정의(Definition of the sum)부터 시작합시다.



우리는 화살표(f, g)의 쌍이 합에서의 사상 h 를 고유하게 결정한다고 말했다. 이를 고차 함수 (higher-order function)를 사용하여 간결하게 쓸 수 있습니다:

```
h = mapOut (f, g)
```

어디:

```
mapOut :: (a -> c, b -> c) -> (Either a b -> c)
mapOut (f, g) = \aorb -> case aorb of
    Left a -> f a
    Right b -> g b
```

이 함수는 두 함수의 쌍을 인수로 받아들여 하나의 함수를 반환합니다.

먼저, 우리는 쌍 (f, g)을 패턴 매칭하여 f 와 g 를 추출합니다. 그런 다음 람다를 사용하여 새로운 함수를 구성합니다. 이 람다는 `Either a b` 타입의 인수 `aorb`를 받아서 케이스 분석을 수행합니다. 만약 `Left`를 사용하여 구성되었다면, 그 내용에 f 를 적용하고, 그렇지 않으면 g 를 적용합니다.

함수가 클로저(closure)로 반환된다는 점에 유의하십시오. 이는 환경에서 f 와 g 를 캡처합니다.

우리가 구현한 함수는 다이어그램을 충실히 따르고 있으나, 일반적인 Haskell 스타일로 작성되지는 않았습니다. Haskell 프로그래머들은 여러 개의 인수를 갖는 함수들을 커리 (curry)하는 것을 선호합니다. 또한, 가능하다면 람다(lambda)도 제거하는 것을 선호합니다.

다음은 Haskell 표준 라이브러리에서 `either`(이더)라는 이름으로 제공되는 동일한 함수의 버전입니다:

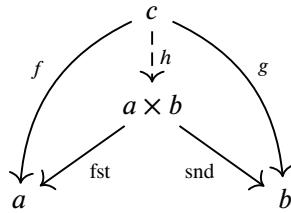
```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)      =  f x
either _ g (Right y)     =  g y
```

다른 방향의 전단사(bijection), h 에서 쌍 (f, g)로의 경우 또한 도표의 화살표를 따릅니다.

```
unEither :: (Either a b -> c) -> (a -> c, b -> c)
unEither h = (h . Left, h . Right)
```

곱 타입(Product types)

곱형(product types)은 그들의 매핑-인 속성(mapping-in property)에 의해 이중적으로 정의됩니다.



다음은 이 도표의 직접적인 Haskell 읽기입니다

```
h :: (c -> a, c -> b) -> (c -> (a, b))
h (f, g) = \c -> (f c, g c)
```

그리고 이것은 Haskell 스타일로 인피스(infix) 연산자로 작성된 스타일 버전 `&&&`입니다.

```
(&&&) :: (c -> a) -> (c -> b) -> (c -> (a, b))
(f &&& g) c = (f c, g c)
```

다른 방향의 전단사 함수(bijection)는 다음과 같이 주어집니다:

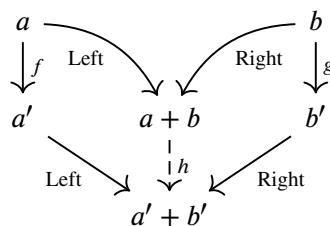
```
fork :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

이는 또한 다이어그램의 읽기와 매우 밀접하게 관련됩니다.

함자성(Functoriality) 재검토

합과 곱은 함수적(functorial)입니다. 이는 우리가 해당 내용(content)들에 함수를 적용할 수 있음을 의미합니다. 우리는 이제 이러한 다이어그램을 코드로 변환할 준비가 되었습니다.

이는 합 타입(선형형식)의 함자성(functoriality)입니다:



이 다이어그램을 읽으면 즉시 `h`를 `either`(또는) 사용하여 쓸 수 있습니다:

```
h f g = either (Left . f) (Right . g)
```

아니면 이를 확장하여 `bimap`이라고 부를 수 있습니다:

```
bimap :: (a -> a') -> (b -> b') -> Either a b -> Either a' b'
bimap f g (Left a) = Left (f a)
bimap f g (Right b) = Right (g b)
```

유사하게, 곱(product) 타입에 대해:

$$\begin{array}{ccccc}
 & & a \times b & & \\
 & \swarrow \text{fst} & \downarrow h & \searrow \text{snd} & \\
 a & & a' \times b' & & b \\
 \downarrow f & \swarrow \text{fst} & & \searrow \text{snd} & \downarrow g \\
 a' & & b' & & b'
 \end{array}$$

h 를 다음과 같이 쓸 수 있습니다:

`h f g = (f . fst) &&& (g . snd)`

혹은 그것을 확장하여

`bimap :: (a -> a') -> (b -> b') -> (a, b) -> (a', b')`
`bimap f g (a, b) = (f a, g b)`

두 경우 모두, 이 고차 함수(higer-order function)를 `bimap`이라고 부릅니다. 왜냐하면 Haskell에서 합(sum)과 곱(product) 모두 **Bifunctor**라고 불리는 더 일반적인 클래스의 인스턴스이기 때문입니다.

6.2 함수형 타입의 함자성(Functoriality of the Function Type)

함수형, 또는 지수형도 함자적(functorial)입니다, 그러나 약간의 변형이 있습니다. 우리는 b^a 에서 $b'^{a'}$ 로의 매핑(mapping)에 관심이 있으며, 여기서 1차 표시된 객체들은 비 1차 표시된 객체들과 몇 가지 화살(arrow)들에 의해 연결됩니다—이를 결정해야 합니다.

지수 함수는 그 함수의 매핑-인 속성(mapping-in property)으로 정의됩니다. 그래서 우리가 찾고 있는 것이

$$k : b^a \rightarrow b'^{a'}$$

다이어그램을 그려야 합니다. 이 다이어그램은 k 를 $b'^{a'}$ 로 매핑하는 것으로, b^a 를 c 로 대체하고 비-프라임 객체를 프라임 객체로 대체하여 원래 정의에서 얻습니다.

$$\begin{array}{ccc}
 b^a \times a' & & \\
 \downarrow k \times id_a & \searrow g & \\
 b'^{a'} \times a' & \xrightarrow{\epsilon} & b'
 \end{array}$$

질문은: 이 도형을 완성하기 위해 화살표 g 를 찾을 수 있는가?

$$g : b^a \times a' \rightarrow b'$$

만약 우리가 그러한 g 를 찾는다면, 그것은 우리의 k 를 고유하게 정의할 것입니다.

이 문제를 생각하는 방법은 g 를 어떻게 구현할 것인지 고려하는 것입니다. 이는 $b^a \times a'$ 의 곱(product)을 인수로 가집니다. 이를 쌍(pair)으로 생각해 보세요: a 에서 b 로 가는 함수 객체(function object)의 요소와 a' 의 요소입니다. 함수 객체로 할 수 있는 유일한 일은 이를 다른 것에 적용하는 것입니다. 그러나 b^a 는 타입 a 의 인수가 필요하며, 우리가 사용할 수 있는 것은 a' 뿐입니다. 누군가가 우리에게 $a' \rightarrow a$ 화살표(arrow)를 주지 않는 한 우리는 아무것도 할 수 없습니다. 이 화살표를 a' 에 적용하면 b^a 의 인수를 생성합니다. 그러나 적용 결과는 타입 b 이며, g 는 b' 를 생성해야 합니다. 다시 말해, 과제를 완료하기 위해서는 $b \rightarrow b'$ 화살표가 필요합니다.

이것은 복잡하게 들릴 수 있지만, 요점은 우리는 첨자(primed)된 객체와 비첨자(non-primed)된 객체 사이에 두 개의 화살표가 필요하다는 것입니다. 여기서의 반전은 첫 화살표가 a' 에서 a 로 가는데, 이는 보통의 함자성(functoriality) 고려와는 반대 방향으로 느껴진다는 점입니다. b^a 를 $b'^{a'}$ 로 매핑하기 위해 저희는 한 쌍의 화살표가 필요합니다:

$$\begin{aligned} f &: a' \rightarrow a \\ g &: b \rightarrow b' \end{aligned}$$

이것은 Haskell에서 설명하기가 더 쉽습니다. 우리의 목표는 $h :: a \rightarrow b$ 함수를 주어진 $a' \rightarrow b'$ 함수를 구현하는 것입니다.

이 새로운 함수는 a' 타입의 인수를 받습니다. 그러므로 h 에 전달하기 전에 a' 를 a 로 변환해야 합니다. 그래서 우리는 $f :: a' \rightarrow a$ 라는 함수가 필요합니다.

h 는 b 를 생성하고, 우리는 b' 를 반환하고자 하므로, 다른 함수 $g :: b \rightarrow b'$ 가 필요합니다. 이 모든 것은 하나의 고차 함수에 잘 맞습니다:

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

`bimap`은 **Bifunctor** 타입 클래스(typeclass)의 인터페이스인 것과 유사하게, `dimap`은 **Profunctor** 타입 클래스의 멤버입니다.

6.3 이중 데카르트 닫힌 범주(Bicartesian Closed Categories)

카테고리 안의 객체 쌍에 대해 곱(product)과 지수(exponential)가 정의되어 있고 끝 객체(terminal object)가 있는 카테고리를 데카르트 닫힌(cartesian closed) 카테고리라고 합니다. 이는 hom-집합(hom-sets)이 해당 카테고리에 속하고, hom-집합을 형성하는 연산에 대해 “닫혀 있다”는 것을 의미합니다.

만약 범주가 합(공소산, coproducts)과 초기 대상(initial object)을 갖는다면, 이를 이중 카르테지안 닫힌 범주(bicartesian closed)라고 합니다.

이것은 프로그래밍 언어를 모델링하기 위한 최소한의 구조입니다.

이러한 연산을 사용하여 구성된 데이터 타입들은 대수적 데이터 타입들(algebraic data types)이라고 합니다. 우리는 타입들의 덧셈, 곱셈, 거듭제곱(하지만 뺄셈이나 나눗셈은 아님)을 가지고 있습니다; 우리가 고등학교 대수학에서 알던 모든 친숙한 법칙들이 있습니다. 이들은 동형성(isomorphism)까지 만족합니다. 아직 논의하지 않은 또 하나의 대수적 법칙이 있습니다.

분배법칙(Distributivity)

숫자의 곱셈은 덧셈에 대해 분배된다. 우리는 이와 동일한 것이 이연계 닫힌 범주(bicartesian closed category)에서도 기대되어야 할까요?

$$b \times a + c \times a \cong (b + c) \times a$$

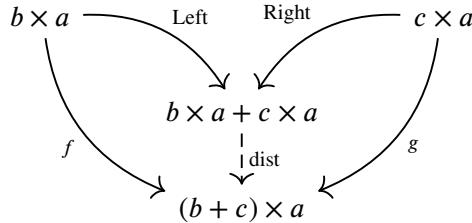
왼쪽에서 오른쪽으로의 매핑(mapping)은 구성하기 쉽습니다. 이는 합(sum)에서의 매핑이자 곱(product)으로의 매핑이 동시에 이루어지기 때문입니다. 우리는 이를 점진적으로 더 단순한 매핑들로 분해함으로써 구성할 수 있습니다. Haskell에서는 이것이 함수를 구현하는 것을 의미합니다.

```
dist :: Either (b, a) (c, a) -> (Either b c, a)
```

왼쪽의 합에 대한 사상(mapping)은 화살표 쌍으로 제공됩니다:

$$f : b \times a \rightarrow (b + c) \times a$$

$$g : c \times a \rightarrow (b + c) \times a$$



Haskell로 다음과 같이 작성합니다:

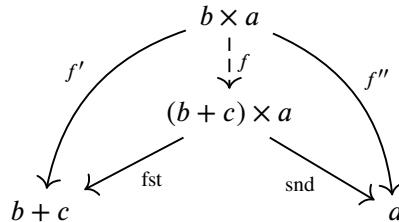
```
dist = either f g
where
    f :: (b, a) -> (Either b c, a)
    g :: (c, a) -> (Either b c, a)
```

where 절(clause)은 서브 함수(sub-functions)의 정의를 도입하는 데 사용됩니다.

이제 f 와 g 를 구현해야 합니다. 이들은 곱(product)으로 향하는 사상들이므로 각각의 사상은 화살표(arrow)의 쌍과 동일합니다. 예를 들어, 첫 번째는 다음 쌍으로 주어집니다:

$$f' : b \times a \rightarrow (b + c)$$

$$f'' : b \times a \rightarrow a$$



하스켈(Haskell)에서:

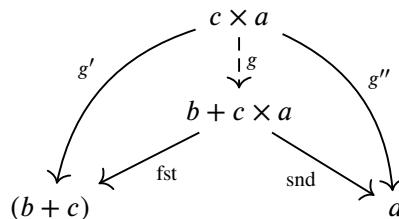
```
f = f' &&& f''
f' :: (b, a) -> Either b c
f'' :: (b, a) -> a
```

첫 번째 화살표(arrow)는 첫 번째 구성 요소(component) b 를 투사한 다음 Left 를 사용하여 합(sum)을 구성함으로써 구현할 수 있습니다. 두 번째는 그냥 투사(projection) snd 입니다:

$$f' = \text{Left} \circ \text{fst}$$

$$f'' = \text{snd}$$

유사하게, 우리는 g 를 g' 와 g'' 의 쌍으로 분해합니다:



이 모든 것을 결합하면, 우리는 다음을 얻습니다:

```
dist = either f g
  where
    f   = f' &&& f''
    f'  = Left . fst
    f'' = snd
    g   = g' &&& g''
    g'  = Right . fst
    g'' = snd
```

이들은 보조 함수들의 타입 서명(signatures)입니다:

```
f   :: (b, a) -> (Either b c, a)
g   :: (c, a) -> (Either b c, a)
f'  :: (b, a) -> Either b c
f'' :: (b, a) -> a
g'  :: (c, a) -> Either b c
g'' :: (c, a) -> a
```

이것들은 짧은 형태로 인라인(inline)될 수 있습니다:

```
dist = either ((Left . fst) &&& snd) ((Right . fst) &&& snd)
```

이 스타일의 프로그래밍은 인수(점)를 생략하기 때문에 *point free*라고 합니다. 가독성의 이유로, Haskell 프로그래머들은 더 명시적인 스타일을 선호합니다. 위의 함수는 일반적으로 다음과 같이 구현됩니다:

```
dist (Left (b, a)) = (Left b, a)
dist (Right (c, a)) = (Right c, a)
```

우리가 합(sum)과 곱(product)의 정의만을 사용했음을 주목해 주세요. 동형사상의 다른 방향은 지수함수(exponential)를 사용해야 하므로, 이는 쌍대카테시안 폐형 범주에서만 유효합니다. 이는 단순한 Haskell 구현에서는 즉시 명확하지 않습니다.

```
undist :: (Either b c, a) -> Either (b, a) (c, a)
undist (Left b, a) = Left (b, a)
undist (Right c, a) = Right (c, a)
```

하지만 그것은 커링(currying)이 Haskell에서는 암묵적이기 때문입니다.

다음은 이 함수의 포인트 프리(point-free) 버전입니다:

```
undist = uncurry (either (curry Left) (curry Right))
```

이 구현이 가장 읽기 쉬운 것은 아닐지 몰라도, 지수 함수(exponential)의 필요성을 강조합니다: 우리는 매핑을 구현하기 위해 *curry*와 *uncurry*를 모두 사용합니다.

더 강력한 도구인 수반(adjunction)들로 무장했을 때, 이 항등식으로 다시 돌아오겠습니다.

Exercise 6.3.1. *Show that:*

$$2 \times a \cong a + a$$

where 2 is the Boolean type. Do the proof diagrammatically first, and then implement two Haskell functions witnessing the isomorphism.