UTF8gbsn

Chapter 6

Function Types

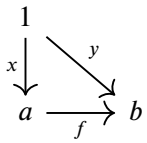
함수형 프로그래밍의 핵심에는 다른 종류의 합성이 있습니다. 함수를 다른 함수의 인자로 전달할 때 발생합니다. 외부 함수는 이 인자를 자신의 기계 장치의 교체 가능한 부분으로 사용할 수 있습니다. 예를 들어, 임의의 비교 함수를 받아들이는 일반적인 정렬 알고리즘을 구현할 수 있습니다.

함수를 객체 사이의 화살표로 모델링한다면, 함수를 인자로 가지는 것은 무슨 의미일까요?

함수를 객체화하는 방법이 필요합니다. 그래야만 "화살표의 객체"를 소스나 대상으로 하는 화살표를 정의할 수 있습니다. 함수를 인자로 받거나 함수를 반환하는 함수를 *고차* 함수라고 합니다. 고차 함수는 함수형 프로그래밍의 주축입니다.

Elimination rule

함수의 정의적 특성은 인자에 적용하여 결과를 생성할 수 있다는 것입니다. 우리는 합성의 관점에서 함수 적용을 정의하였습니다:



여기서 f는 a에서 b로의 화살표로 표현되지만, 우리는 f를 화살표의 객체의 원소로 대체할 수 있기를 원합니다. 또는 수학자들이 부르는 지수 객체 b^a로, 또는 프로그래밍에서 우리가 부르는 함수 타입 `a->b`로 대체할 수 있습니다.

b^a의 원소와 a의 원소가 주어졌을 때, 함수 적용은 b의 원소를 생성해야 합니다. 다시 말해, 원소의 쌍이 주어졌을 때:

$$f : 1 \to b^a$$
$$x : 1 \to a$$

다음과 같은 원소를 생성해야 합니다:

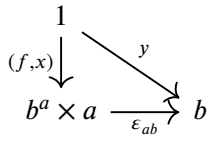
$$y : 1 \to b$$

여기서 f는 b^a의 원소를 나타냅니다. 이전에는 a에서 b로의 화살표였습니다.

우리는 원소의 쌍 (f, x)이 곱 $b^a \times a$의 원소와 동등함을 알고 있습니다. 따라서 함수 적용을 하나의 화살표로 정의할 수 있습니다:

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

이 방식으로, 적용의 결과인 y는 다음 교환 다이어그램에 의해 정의됩니다:



함수 적용은 함수 타입의 *소거 규칙*입니다.

누군가 함수 객체의 원소를 제공할 때, 할 수 있는 유일한 것은 ε을 사용하여 인자 타입의 원소에 적용하는 것입니다.

Introduction rule

함수 객체의 정의를 완성하기 위해, 도입 규칙도 필요합니다.

먼저, 어떤 다른 객체 c에서 함수 객체 b^a를 구성하는 방법이 있다고 가정합니다. 이는 화살표가 있다는 것을 의미합니다.

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

우리는 h의 결과를 ε_{ab}를 사용하여 제거할 수 있지만, 먼저 a와 곱해야 합니다. 그러니 먼저 c를 a와 곱한 후 함자성(functoriality)을 사용하여 이를 $b^a \times a$로 매핑합시다.

함자성은 두 화살표의 쌍을 곱에 적용하여 다른 곱을 얻게 해줍니다. 여기에서 화살표의 쌍은 (h, id_a)입니다(우리는 c를 b^a로 변환하고자 하지만, a를 수정하는 데에는 관심이 없습니다).

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

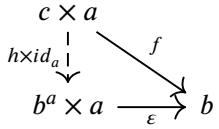
이제 함수 적용을 통해 b에 도달할 수 있습니다.

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

이 합성 화살표는 우리가 f라고 부를 매핑을 정의합니다.

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

여기 해당 다이어그램이 있습니다:



이 교환 다이어그램은 주어진 h에 대해 f를 구성할 수 있음을 알려줍니다; 하지만 우리는 역을 요구할 수도 있습니다: 모든 매핑 아웃, $f : c \times a \to b$는 고유하게 지수로의 매핑을 정의해야 합니다, $h : c \to b^a$.

이 속성, 두 세트의 화살표 사이의 일대일 대응을 사용하여 지수 객체를 정의할 수 있습니다. 이것이 함수 객체 b^a에 대한 *도입 규칙*입니다.

우리는 곱이 그 매핑-인 속성을 사용하여 정의되었음을 보았습니다. 반면에, 함수 적용은 곱의 *매핑 아웃*으로 정의됩니다.

Currying

이 정의를 바라보는 방법은 여러 가지가 있습니다. 하나는 이것을 커링의 예로 보는 것입니다.

지금까지 우리는 한 개의 인자를 가진 함수만을 고려해왔습니다. 이것은 진짜 제한이 아닙니다. 왜냐하면 우리는 항상 두 개의 인자를 가진 함수를 곱으로부터의 (단일-인자) 함수로 구현할 수 있기 때문입니다. 함수 객체의 정의에서 f는 그러한 함수입니다:

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

반면에 `h`는 함수를 반환하는 함수입니다:

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

커링은 이 두 세트의 화살표 사이의 동형사상입니다.

이 동형사상은 Haskell에서 두 개의 (고차) 함수 쌍으로 표현될 수 있습니다. Haskell에서는 커링이 모든 타입에 대해 작동하기 때문에, 이 함수들은 타입 변수를 사용하여 작성됩니다—그것들은 *다형성*입니다:

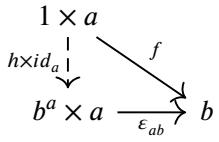
```
curry   :: ((c, a) -> b)   -> (c -> (a -> b))
```

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

다시 말해, 함수 객체의 정의에서 h는 다음과 같이 작성될 수 있습니다:

$$h = curry\ f$$

물론 이렇게 작성하면, `curry`와 `uncurry`의 타입은 화살표보다는 함수 객체에 해당합니다. 이 구분은 일반적으로 간과됩니다. 왜냐하면 지수의 *원소*와 그것들을 정의하는 *화살표* 사이에 일대일 대응이 있기 때문입니다. 우리가 임의의 객체 c를 종단 객체로 대체할 때 이것을 쉽게 볼 수 있습니다. 우리는 얻습니다:



이 경우, h는 객체 b^a의 원소이며, f는 $1 \times a$에서 b로 가는 화살표입니다. 하지만 $1 \times a$가 a와 동형이므로 실질적으로 f는 a에서 b로 가는 화살표입니다.

따라서 앞으로, 화살표 `->`를 화살표 →로 부르되 크게 구분하지 않을 것입니다. 이러한 현상에 대한 올바른 주문은 카테고리가 자기-풍부하다(self-enriched)고 말하는 것입니다.

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

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

하지만 이것은 단지 구문적인 트릭일 뿐입니다: 함수 적용은 언어 내에 내장되어 있습니다. `f x`는 `f`가 `x`에 적용된 것을 의미합니다. 다른 프로그래밍 언어에서는 함수의 인자를 괄호 안에 넣어야 하지만, Haskell에서는 그렇지 않습니다.

함수 적용을 별도의 함수로 정의하는 것이 불필요해 보일 수 있지만, Haskell 라이브러리는 그 목적을 위해 중위 연산자 `$`를 제공합니다:

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

그러나 트릭은 일반적인 함수 적용이 왼쪽으로 결합한다는 것입니다. 예를 들어, `f x y`는 `(f x) y`와 같습니다; 하지만 달러 기호는 오른쪽으로 결합합니다. 따라서

```
f  $ g x
```

는 `f (g x)`와 같습니다. 첫 번째 예에서, `f`는 (적어도) 두 개의 인자를 가진 함수여야 합니다; 두 번째 예에서는 하나의 인자를 가진 함수일 수 있습니다.

Haskell에서는 커링이 매우 흔합니다. 두 개의 인자를 가진 함수는 거의 항상 함수를 반환하는 함수로 작성됩니다. 함수 화살표 `->`가 오른쪽으로 결합하기 때문에, 이러한 유형을 괄호로 묶을 필요가 없습니다. 예를 들어, 쌍 생성자는 다음과 같은 시그니처를 가집니다:

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

이것을 두 개의 인자를 받아 쌍을 반환하는 함수로 생각할 수도 있고, 하나의 인자를 받아 하나의 인자를 가진 함수를 반환하는 함수로 생각할 수도 있습니다. `b->(a, b)`. 이 방식으로 함수를 부분 적용하는 것이 가능하며, 결과는 다른 함수가 됩니다. 예를 들어, 다음과 같이 정의할 수 있습니다:

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

Relation to lambda calculus

함수 객체의 정의를 바라보는 또 다른 방법은 c를 f가 정의된 환경의 타입으로 해석하는 것입니다. 이 경우 환경을 Γ라고 부르는 것이 관례입니다. 화살표는 Γ에서 정의된 변수들을 사용하는 표현식으로 해석됩니다.

간단한 예를 들어 보겠습니다. 표현식:

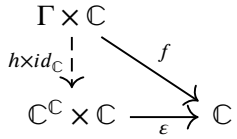
$$ax^2 + bx + c$$

이 표현식을 실수 세 개 (a, b, c)와 변수 x로 매개변수화된 것으로 생각할 수 있습니다. 여기서 x는, 예를 들어, 복소수로 간주됩니다. 이 세 개의 실수는 곱 $\mathbb{R}\times\mathbb{R}\times\mathbb{R}$의 원소입니다. 이 곱은 우리 표현식의 환경 Γ를 구성합니다.

변수 x는 $\mathbb{C}$의 원소입니다. 이 표현식은 곱 $\Gamma\times\mathbb{C}$에서 결과 타입(여기서는 $\mathbb{C}$도 마찬가지)으로 가는 화살표입니다.

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

이는 곱에서 매핑 아웃하는 것이므로, 이를 사용하여 함수 객체 $\mathbb{C}^{\mathbb{C}}$를 구성하고 매핑 $h : \Gamma \to \mathbb{C}^{\mathbb{C}}$를 정의할 수 있습니다. 이렇게 하면 Γ의 각 원소(환경)에 대해 복소수 함수를 반환하는 함수를 정의할 수 있습니다. 이 함수는 Γ에서 정의된 매개변수를 사용하여 $\mathbb{C}$에서 $\mathbb{C}$로의 함수를 만듭니다.



이 새로운 매핑 h는 함수 객체의 생성자로 볼 수 있습니다. 결과적으로 생성된 함수 객체는 환경 Γ, 즉 매개변수 (a, b, c)에 접근할 수 있는 모든 $\mathbb{C}$에서 $\mathbb{C}$로의 함수를 대표합니다.

우리의 원래 표현식 $ax^2 + bx + c$에 해당하는 특정 함수는 $\mathbb{C}^{\mathbb{C}}$에서 다음과 같이 작성할 수 있습니다:

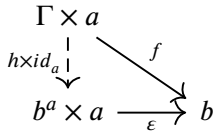
$$\lambda x.\, ax^2 + bx + c$$

또는 Haskell에서는 λ 대신 backslash를 사용하여 다음과 같이 작성할 수 있습니다:

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

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

일반적으로, 함수 객체를 정의하는 다이어그램은 다음과 같습니다:



표현식 f에 대한 자유 매개변수를 제공하는 환경 Γ는 매개변수의 유형을 대표하는 여러 객체의 곱입니다(우리 예제에서는 $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$였습니다).

빈 환경은 종단 객체 1, 곱의 단위로 표현됩니다. 그 경우, f는 단순히 $a \to b$로의 화살표이고, h는 f에 해당하는 함수 객체 b^a에서 원소를 선택합니다.

일반적으로, 함수 객체는 외부 매개변수에 의존하는 함수를 대표합니다. 이러한 함수는 *클로저*라고 불립니다. 클로저는 그들의 환경에서 값들을 캡처하는 함수들입니다.

다음은 우리 예제를 Haskell로 번역한 것입니다. f에 해당하는 표현식은 다음과 같습니다:

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

만약 우리가 haskDouble을 이용하여 *mathbbR*을 근사화한다면, 우리의 환경은 hask(Double, Double, Double)의 곱셈에서 나옵니다. 타입 haskComplex는 다른 타입에 의해 매개변수화되며, 여기서는 haskDouble을 다시 사용하였습니다.

```
type C = Complex Double
```

`Double`에서 `C`로의 변환은 허수 부분을 0으로 설정하여, `(a :+ 0)`와 같이 수행됩니다.

해당 화살표 h는 환경을 받아서 `C -> 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

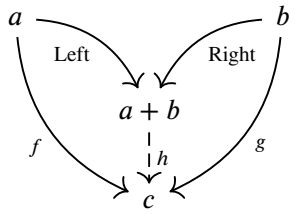
논리학에서, 함수 객체는 함축에 해당합니다. 종단 객체에서 함수 객체로의 화살표는 그 함축의 증명입니다. 함수적용 ϵ은 논리학자들이 *modus ponens*라고 부르는 것에 해당합니다: 만약 당신이 함축 $A \Rightarrow B$의 증명과 A의 증명을 가지고 있다면, 이는 B의 증명을 구성합니다.

6.1 합과 곱에 대한 재방문

함수가 다른 타입의 요소와 동등한 지위를 얻을 때, 우리는 다이어그램을 코드로 직접 변환할 도구를 가지게 됩니다.

합 타입

합의 정의부터 시작해봅시다.



우리는 화살표 쌍 (f, g)이 합에서 mapping h를 고유하게 결정한다고 말했습니다. 이를 고차 함수를 사용하여 간결하게 작성할 수 있습니다.

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

where:

```
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 인자로 받아서, 한 함수를 반환합니다.

먼저, 우리는 `(f, g)`의 쌍을 패턴 매칭하여 `f`와 `g`를 추출합니다. 그리고 나서 우리는 람다를 이용하여 새로운 함수를 작성합니다. 이 람다는 `Either a b` 타입의 인자를 받아서, `aorb`라고 부르고 이를 case 분석합니다. 만약 이것이 `Left`을 이용하여 만들어졌다면, 우리는 `f`를 그의 내용에 적용하고, 그렇지 않으면 `g`를 적용합니다.

반환하는 함수가 닫힌 함수라는 점에 유의하세요. 이 닫힌 함수는 `f`와 `g`를 그의 환경에서 포착합니다.

우리가 구현한 함수는 다이어그램을 밀접하게 따르지만, 일반적인 Haskell 스타일로 작성된 것은 아닙니다. Haskell 프로그래머들은 여러 개의 인자를 가진 함수를 curry하는 것을 선호합니다. 또한 가능하면 람다를 제거하는 것을 선호합니다.

여기 Haskell 표준 라이브러리에서 가져온 동일한 함수의 버전이 있습니다. 여기서 그 함수의 이름은 `either` 입니다.

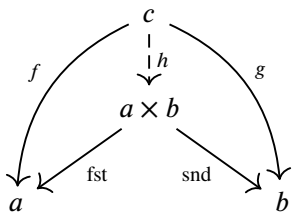
```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f _ (Left x)     =  f x
either _ g (Right y)    =  g y
```

이동 사상의 다른 방향, 즉 h에서 쌍 (f, g)로의 방향도 도표의 화살표를 따릅니다.

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

곱 타입

곱 타입은 그들의 대응-안 속성에 의해 이중으로 정의됩니다.



이 다이어그램의 바로가기 형태의 Haskell 읽기 방법이 있습니다.

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

그리고 이것은 하스켈 스타일로 작성된 중위 연산자 &&&로 표현된 스타일화된 버전입니다.

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

쌍응 (bijection)의 다른 방향은 다음과 같이 주어집니다:

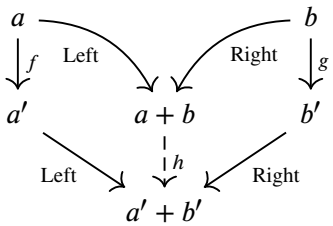
```
fork :: (c -> (a, b)) -> (c -> a, c -> b)
fork h = (fst . h, snd . h)
```

또한, 다이어그램의 해석을 정확하게 따르고 있습니다.

다시 본 함수적 성질

합과 곱 모두 함수적이어서, 그 내용에 함수를 적용할 수 있습니다. 이제 이러한 다이어그램을 코드로 변환할 준비가 되었습니다.

다음은 합 타입의 함수적 성질입니다.



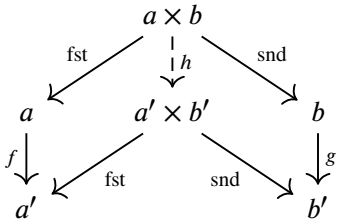
이 다이어그램을 읽어보면, 우리는 즉시 either를 사용하여 *h*를 작성할 수 있습니다.

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

또는 이를 확장하여 bimap이라고 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)
```

마찬가지로, 곱셈 타입에 대해서도:



h는 다음과 같이 표현될 수 있습니다:

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

혹은 그것은 확장될 수 있습니다.

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

두 경우 모두에서, 이 higher-order 함수를 `bimap`이라고 부릅니다. 왜냐하면 Haskell에서의 합과 곱은 더 일반적인 클래스인 `Bifunctor`의 인스턴스이기 때문입니다.

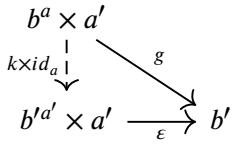
6.2 함수 타입의 Functoriality

함수 타입 또는 지수 함수는 또한 functorial입니다만, 약간의 트위스트가 있습니다. 우리는 b^a에서 $b'^{a'}$로의 매핑에 관심이 있으며, 여기에서 프라임이 붙은 객체는 일부 화살표들을 통해 프라임이 없는 객체들과 관련이 있습니다—결정해야 합니다.

지수는 그 mapping-in 속성에 의해 정의되므로, 만약 우리가 찾고 있다면

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

우리는 k를 $b'^{a'}$에 대한 매핑으로 가진 다이어그램을 그려야 합니다. 우리는 원래의 정의에서 b^a를 c 대신에 넣고, non-primed 객체들 대신 primed 객체들을 넣어 이 다이어그램을 얻었습니다.



질문은: 이 다이어그램을 완성하기 위해 화살표 g를 찾을 수 있을까요?

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

만약 우리가 그런 g를 찾는다면, 이것은 우리의 k를 고유하게 정의할 것입니다.

이 문제에 대해 생각하는 방법은 g를 어떻게 구현할 것인가를 고려하는 것입니다. 이것은 곱 $b^a \times a'$을 그것의 인자로 취합니다. 그것을 한 쌍으로 생각하십시오: a에서 b로의 함수 객체의 요소와 a'의 요소입니다. 우리가 함수 객체에 대해 할 수 있는 유일한 것은 그것을 어떤 것에 적용하는 것입니다. 그러나 b^a는 a 형태의 인자를 필요로 하며, 우리가 사용할 수 있는 것은 a'뿐입니다. 만약 누군가가 $a' \to a$ 사상을 주지 않는다면 우리는 아무것도 할 수 없습니다. 이 화살표는 a'에 적용하면 b^a의 인자를 생성합니다. 그러나, 이 적용의 결과는 b 형태이며, g는 b'를 생성해야 합니다. 이를 위해선, $b \to b'$ 화살표가 다시 필요로 합니다.

이것은 복잡하게 들릴 수 있지만, 결론적으로는 우리는 유일한 문자가 있는 객체와 그렇지 않은 객체 사이에 두 화살표를 필요로 합니다. 요체는 첫 번째 화살표가 a'에서 a로 가는 것이며, 이는 일반적으로 고려하는 함수성에 비해 반대로 느껴질 수 있습니다. b^a를 $b'^{a'}$로 매핑하기 위해선, 화살표 쌍이 필요합니다.

$$f : a' \to a$$
$$g : b \to b'$$

이것은 Haskell에서 설명하기 좀 더 쉽습니다. 우리의 목표는 함수 `h :: a -> b`가 주어진 경우, 함수 `a' -> b'`를 구현하는 것입니다.

이 새 함수는 `a'` 타입의 인자를 받기 때문에, 우리가 그것을 `h`에 전달하기 전에 `a'`를 `a`로 변환해야 합니다. 그래서 우리는 `f :: a' -> a` 함수가 필요합니다.

`h`가 `b`를 생성하고, 우리가 `b'`를 반환하길 원하기 때문에, 우리는 또 다른 함수 `g :: b -> b'`가 필요합니다. 이 모든 것이 더 높은 차수의 함수로 잘 맞아떨어집니다:

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

`bimap`이 `Bifunctor` 타입 클래스에 대한 인터페이스인 것처럼, `dimap`은 `Profunctor` 타입 클래스의 일원입니다.

6.3 양종 카르테시안 폐쇄 범주

객체쌍에 대해 곱과 지수가 모두 정의된 범주는 '카르테시안 폐쇄'라고 부릅니다. 터미널 객체가 있는 경우도 같습니다. 넘-집합들이 해당 범주에서 이질적인 것이 아니라는 것이 중요합니다: 범주는 넘-집합들를 형성하는 연산 하에 "폐쇄"되어 있습니다.

범주가 더함(코프로덕트)과 초기 객체도 가지고 있다면, 이를 '양종 카르테시안 폐쇄'라고 부릅니다.

이것이 프로그래밍 언어 모델링을 위한 최소한의 구조입니다.

이러한 연산을 사용하여 구축된 데이터 유형을 '대수 데이터 유형'이라고 부릅니다. 우리는 유형의 덧셈, 곱셈, 거듭제곱(뺄셈 혹은 나눗셈 제외)을 가지며, 이는 우리가 고등학교 대수학에서 알고 있는 모든 유형의 법칙을 가지고 있습니다. 이들은 동형사상까지 만족합니다. 아직 논의하지 않은 하나의 대수적 법칙이 더 있습니다.

분배성

숫자의 곱셈은 덧셈에 대해 분배됩니다. 우리는 같은 것을 양종 카르테시안 폐쇄 범주에서도 기대할 수 있을까요?

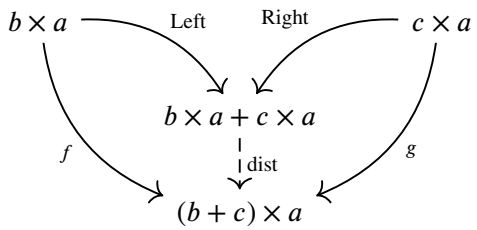
$$b \times a + c \times a \cong (b + c) \times a$$

왼쪽에서 오른쪽으로의 매핑은 쉽게 구성할 수 있습니다. 왜냐하면 그것은 동시에 합과 곱으로의 매핑이기 때문입니다. 이를 점진적으로 간단한 매핑들로 분해함으로써 구성할 수 있습니다. Haskell에서 이는 함수를 구현하는 것을 의미합니다.

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

왼쪽의 합에 대한 매핑은 화살표 쌍으로 주어집니다.

$$f : b \times a \to (b + c) \times a$$
$$g : c \times a \to (b + c) \times a$$



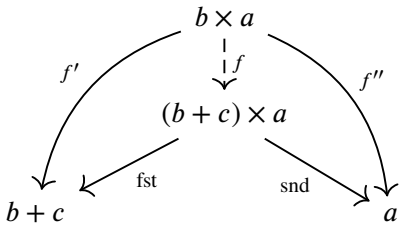
이를 Haskell로 작성할 수 있습니다.:

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

`where` 절은 하위 함수의 정의를 도입하는 데 사용됩니다.

이제 우리는 f와 g를 구현해야 합니다. 이들은 곱집합으로의 매핑이므로 각각은 화살표 쌍에 해당합니다. 예를 들어, 첫 번째는 다음 쌍에 의해 주어집니다:

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



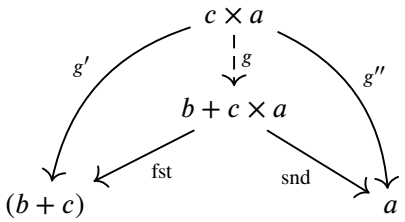
In Haskell:

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

첫 번째 화살표는 첫 번째 구성요소 b를 투영하고, 그 후 Left을 사용하여 합을 구성함으로써 구현될 수 있습니다. 두 번째는 단순히 투영 snd입니다.

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

마찬가지로, 우리는 g를 쌍 g'와 g''로 분해합니다.



이 모든 것을 결합하면, 다음을 얻습니다:

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

다음은 도우미 함수의 유형 시그니처들입니다:

```
    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
```

그들은 또한 이 간결한 형태를 생성하기 위해 인라인 될 수도 있습니다.

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

이런 프로그래밍 스타일은 인자(포인트)들이 생략되기 때문에 *point free*라고 부릅니다. 가독성을 위해, 하스켈 프로그래머들은 보다 명시적인 스타일을 선호합니다. 위의 함수는 일반적으로 다음과 같이 구현됩니다.

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

우리는 합과 곱의 정의만을 사용하였습니다. 이 이상형의 다른 방향은 지수를 사용하는데 이는 bicartesian *closed* 카테고리에서만 유효합니다. 이것은 단순한 하스켈 구현에서 바로 명확해지지 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 커링이 하스켈에서는 암시적이기 때문입니다.

다음은 이 함수의 포인트-프리 버전입니다:

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

이것은 가장 읽기 쉬운 구현체는 아닐 수 있지만, 지수를 요구하는 사실을 강조합니다: `curry`와 `uncurry`를 모두 사용하여 매핑을 구현합니다.

나중에 더 강력한 도구인 adjunctions를 갖추게 될 때, 우리는 이 정체성에 다시 돌아올 것입니다.

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.*