Chapter 5

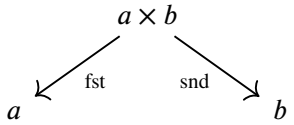
# Product Types

주어진 타입의 가능한 값들을 나열하기 위해 합 타입(sum types)을 사용할 수 있습니니다, 하지만 인코딩이 낭비될 수 있습니니다. 0에서 9까지의 숫자를 인코딩하기 위해 꼬꼬하도록 열 개의 생성자(constructors)가 필요했습니니다.

```
data Digit = Zero | One | Two | Three | ... | Nine
```

하지만 두 자릿수를 하나의 데이터 구조로 결합하면, 두 자릿수의 십진수를 통해 백 개의 숫자를 인코딩할 수 있습니니다. 혹은, 노자의 말을 빌리자면, 네 자릿수만 있으면 만 개의 숫자를 인코딩할 수 있습니다.

두 타입을 이런 방식으로 결합한 데이터 타입을 곱(product) 또는 *카테시안 곱*이라 합니다. 그 정의의 중요한 요소는 제거 규칙입니다: $a \times b$로부터 나오는 두 개의 화살표가 있습니다; 하나는 "fst"이라고 불리고 $a$로 가며, 다른 하나는 "snd"라고 불리고 $b$로 갑니다. 이들은 (카테시안) *사영*이라고 불립니다. 이를 통해 우리는 곱 $a \times b$에서 $a$와 $b$를 추출할 수 있습니다.

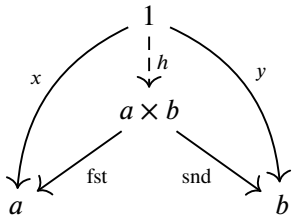



가정해 봅시다, 누군가 여러분에게 곱의 원소, 즉 종말 대상 1에서 $a \times b$로의 화살표 $h$를 주었다고 합시다. 여러분은 단순히 합성을 사용하여, $a$의 원소와 $b$의 원소 한 쌍을 쉽게 추출할 수 있습니다: $a$의 원소는

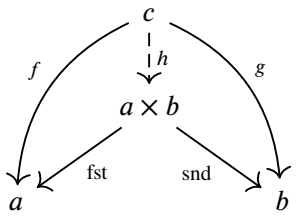
$$x = \text{fst} \circ h$$

그리고 $b$의 요소는

$$y = \text{snd} \circ h$$




사실, 임의의 객체 $c$에서 $a \times b$로 가는 화살표가 주어진다면, 합성을 통해 화살표 쌍 $f : c \to a$와 $g : c \to b$를 정의할 수 있습니니다.




이전에 합 타입과 함께 했던 것처럼, 이 아이디어를 뒤집어 사용하여 이 도표를 제품 타입을 *정의*하는 데 사용할 수 있습니다: 함수 $f$와 $g$의 쌍이 $c$에서 $a \times b$로의 *mapping in*과 일대일 대응 관계에 있음을 조건으로 합니다. 이것이 제품의 *도입* 규칙입니다.

특히, 터미널 객체(terminal object)로의 사상(mapping out)은 Haskell에서 곱 타입(product type)을 정의하는데 사용됩니다. 두 요소 `a :: A`와 `b :: B`가 주어졌을 때, 우리는 그 곱을 구성합니다.

```
(a, b) :: (A, B)
```

내장 구문은 단지 그것: 괄호 한 쌍과 중간의 쉼표입니다. 이는 두 타입의 곱 `(A, B)`를 정의하는 것과 두 요소를 받아서 짝을 짓는 데이터 생성자 `(a, b)` 모두에 대해 작동합니다.

프로그래밍의 목적을 절대 잊어서는 안 됩니다: 복잡한 문제를 일련의 더 간단한 문제로 분해하는 것입니다. 우리는 이를 곱(product)의 정의에서도 다시 볼 수 있습니다. 곱으로 *매핑(mapping)*을 구성해야 할 때마다, 이를 곱의 요소 중 하나로 매핑하는 두 개의 함수 쌍을 구성하는 작은 작업 두 개로 분해합니다. 이는 두 개의 값을 반환하는 함수를 구현하려면, 쌍(pair)의 요소 각각을 반환하는 두 개의 함수를 구현하는 것으로 충분하다는 말과 같습니다.

## Logic

논리학에서, 곱형(product type)은 논리적 결합(logical conjunction)에 해당합니다. $A \times B$ ($A$와 $B$)를 증명하려면 *둘 다* $A$ 와 $B$의 증명을 제공해야 합니다. 이것들이 $A$와 $B$를 대상으로 하는 화살표입니다. 소거 규칙(elimination rule)은 $A \times B$의 증명을 가지고 있다면 자동으로 $A$의 증명(fst를 통해)과 $B$의 증명(snd를 통해)을 얻을 수 있다고 합니다.

## Tuples and Records

노자의 말처럼, 만 가지 물체의 곱(product)은 단지 만 가지 투영으로 이루어진 하나의 물체입니다.

Haskell에서는 튜플 표기법을 사용해 임의의 곱(product)을 형성할 수 있습니다. 예를 들어, 세 가지 타입의 곱은 `(A, B, C)`로 작성됩니다. 이 타입의 항(term)은 세 가지 요소로부터 `(a, b, c)`로 구성될 수 있습니다.

수학자들이 "표기 악용"이라고 부르는 방식으로, 0개의 타입의 곱은 공튜플(())으로 작성되며, 이는 우연히도 종말 대상(terminal object) 또는 단위 타입(unit type)과 동일합니다. 이는 곱셈이 숫자의 곱셈과 매우 유사하게 작동하기 때문인데, 이때 종말 대상이 1의 역할을 합니다.

Haskell에서 우리는 모든 튜플에 대한 별도의 사영을 정의하는 대신 패턴 매칭 문법을 사용합니다. 예를 들어, 트리플에서 세 번째 요소를 추출하기 위해 다음과 같이 작성합니다.

```
thrd :: (a, b, c) -> c
thrd (_, _, c) = c
```

구성 요소(component) 중 무시하고 싶은 부분에는 와일드카드(wildcard)를 사용합니다.

Lao Tzu가 말하길, “이름 붙이기는 모든 특정한 것들의 기원이다.”라고 했습니다. 프로그래밍에서 특정한 튜플의 구성 요소들의 의미를 파악하기는 이름을 붙이지 않으면 어렵습니다. 레코드(record) 구문을 사용하면 투영(projection)에 이름을 붙일 수 있습니다. 이것이 레코드 스타일로 작성된 곱(product)의 정의입니다:

```
data Product a b = Pair { fst :: a, snd :: b }
```

Pair는 데이터 생성자(data constructor)이고 fst와 snd는 프로젝션(projections)입니다.

이것은 특정한 쌍(pair)을 선언하고 초기화하는 방법입니다:

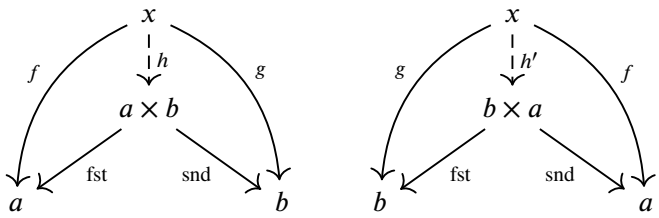
```
ic :: Product Int Char
ic = Pair 10 'A'
```

## 5.1 Cartesian Category

하스켈(Haskell)에서 우리는 임의의 두 타입의 곱을 정의할 수 있습니다. 모든 곱이 존재하고 끝 대상(terminal object)이 존재하는 범주를 *cartesian*이라고 합니다.

### Tuple Arithmetic

소스와의 대응에 의해 곱이 만족하는 아이덴티티(identity)들을 유도할 수 있습니다. 예를 들어, $a \times b \cong b \times a$를 증명하기 위해 다음 두 개의 다이어그램을 고려하십시오.




그들은 어떠한 객체 $x$에 대해 $a \times b$로의 화살표가 $b \times a$로의 화살표와 일대일 대응 관계에 있음을 보여줍니다. 이는 이러한 화살표 각각이 동일한 쌍 $f$와 $g$에 의해 결정되기 때문입니다.

자연성 조건이 충족되는지 확인할 수 있는데, 이는 $k : x' \to x$를 사용하여 관점을 이동할 때, $x$에서 시작하는 모든 화살표들이 사전 구성 $(- \circ k)$에 의해 이동되기 때문입니다.

Haskell에서, 이러한 동형사상(isomorphism)은 자기 자신의 역함수(inverse)인 함수로 구현될 수 있습니 다:

```
swap :: (a, b) -> (b, a)
swap x = (snd x, fst x)
```

Here’s the same function written using pattern matching:여기 패턴 매칭을 사용하여 작성된 같은 함수가 있습니다:

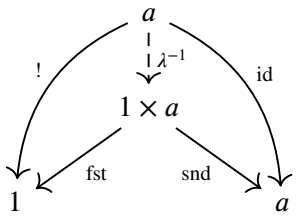
```
swap (x, y) = (y, x)
```

제품이 대칭적임을 명심하는 것이 중요합니다. 이는 “동형사상 (isomorphism)만큼만” 대칭적이라는 뜻입니다. 순서쌍의 순서를 바꾸어도 프로그램의 동작이 변하지 않는다는 의미는 아닙니다. 대칭성은 바뀐 순서쌍의 정보 내용이 동일하다는 것을 의미하지만, 접근 방식은 수정이 필요합니다.

단말 객체는 곱(Product)의 단위(unit)로, $1 \times a \cong a$를 만족합니다. $1 \times a$와 $a$ 사이의 동형 사상을 증명하는 화살표(arrow)를 *좌단위원(left unitor)*이라고 합니다.

$$\lambda : 1 \times a \to a$$

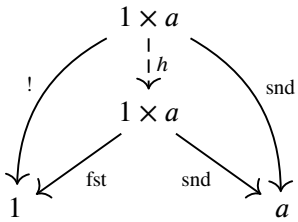
다음과 같이 구현할 수 있습니다: $\lambda$ = snd. 그것의 역함수 $\lambda^{-1}$는 다음 다이어그램에서 유일한 화살표로 정의됩니다:




$a$에서 1로 가는 화살표는 ! (발음은, *뱅*)이라고 불립니다. 이는 정말로

$$\text{snd} \circ \lambda^{-1} = \text{id}$$

우리는 $\lambda^{-1}$가 snd의 왼쪽 역원이라는 것을 증명해야 합니다. 다음의 다이어그램을 고려하세요:




당연히 $h = id$에 대해 교환합니다. $h = \lambda^{-1} \circ \text{snd}$에 대해서도 교환합니다. 왜냐하면 우리는 다음과 같습니다:

$$\text{snd} \circ \lambda^{-1} \circ \text{snd} = \text{snd}$$

$h$가 유일하다고 가정하기 때문에, 우리는 다음과 같이 결론을 내립니다:

$$\lambda^{-1} \circ \text{snd} = id$$

이러한 보편적 구성(universal constructions)으로의 추론(reasoning)은 매우 일반적입니다.

다음은 Haskell로 작성된 몇 가지 다른 동형사상입니다 (역함수를 가지는 증명 없이). 이는 결합 법칙입니다:

```
assoc :: ((a, b), c) -> (a, (b, c))
assoc ((a, b), c) = (a, (b, c))
```

그리고 이것은 오른쪽 유닛(unit)입니다.

```
runit :: (a, ()) -> a
runit (a, _) = a
```

이 두 함수는 *결합자*에 해당합니다.

$$\alpha : (a \times b) \times c \to a \times (b \times c)$$

and the *오른쪽 단위자*:

$$\rho : a \times 1 \to a$$

**Exercise 5.1.1.** *Show that the bijection in the proof of left unit is natural. Hint, change focus using an arrow* $g : a \to b$.

**Exercise 5.1.2.** *Construct an arrow*

$$h : b + a \times b \to (1 + a) \times b$$

*Is this arrow unique?*

*Hint: It's a mapping into a product, so it's given by a pair of arrow. These arrows, in turn, map out of a sum, so each is given by a pair of arrows.*

*Hint: The mapping* $b \to 1 + a$ *is given by* $(Left \circ !)$

**Exercise 5.1.3.** *Redo the previous exercise, this time treating* $h$ *as a mapping* out *of a sum.*

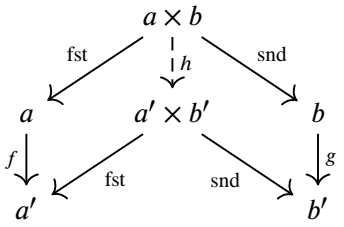
**Exercise 5.1.4.** *Implement a Haskell function* `maybeAB :: Either b (a, b) -> (Maybe a, b)`. *Is this function uniquely defined by its type signature or is there some leeway?*

### Functoriality

화살표가 $a$와 $b$를 각각 $a'$와 $b'$로 사상한다고 가정하면:

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

이 화살표들은 각각 fst와 snd 투영(projection)과 합성(composition)하여 곱셈(product) 사이의 매핑(mapping) $h$를 정의하는 데 사용할 수 있습니다:




이 다이어그램을 나타내는 축약 표기법은:

$$a \times b \xrightarrow{f \times g} a' \times b'$$

This property of the product is called *함자성(functoriality)*. You can imagine it as allowing you to transform the two objects *곱(product)* 내에서 새로운 곱을 얻기 위해 변환시키는 것처럼 상상할 수 있다. We also say that *함자성*은 우리가 곱에 대해 작용하는 화살표(pair of arrows)를 *들어올리게(lift)* 한다고 말한다.

## 5.2 Duality

아이(Child)가 화살표(arrow)를 보면, 어느 쪽 끝이 출발점(source)을 가리키고 어느 쪽이 목표점(target)을 가리키는지 알게 됩니다.

$$a \to b$$

하지만 아마도 이것은 단지 편견일 것입니다. 만약 우리가 $b$를 원천(source)이라고 부르고 $a$를 목표(target)라고 부른다면, 우주는 매우 다를까요?

우리는 여전히 이 화살표를 저 화살표와 합성할 수 있습니다.

$$b \to c$$

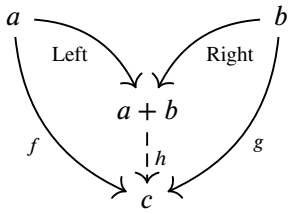
"대상(target)" $b$가 $a \to b$의 "원천(source)"과 동일한 경우, 그 결과는 여전히 하나의 화살표일 것입니다

$$a \to c$$

지금에서야 우리는 그것이 $c$에서 $a$로 간다고 말할 것입니다.

이 이중 우주에서는 우리가 "초기"라고 부르는 객체는 모든 객체로부터 오는 유일한 화살표의 "목적지"이기 때문에 "최종"이라고 불립니다. 반대로 최종 객체는 초기라고 불립니다.

이제 우리가 합 객체(sum object)를 정의하기 위해 사용한 이 도표를 고려해 보십시오:




새로운 해석에서는, 화살표 $h$가 임의의 객체 $c$ "로부터" 우리가 $a + b$라고 부르는 객체 "까지" 이어지는 것으로 봅니다. 이 화살표는 "출발점"이 $c$인 화살표 $(f, g)$ 쌍에 의해 고유하게 정의됩니다. 만약 Left를 fst로, Right를 snd로 바꾸면, 우리는 곱(product)의 정의 도표를 얻게 됩니다.

곱은 화살표가 반대로 된 합입니다.

반대로, 합(sum)은 화살표가 반대인 곱(product)입니다.

모든 구성(Construction)은 범주 이론(Category Theory)에서 그 이중(Dual)을 가집니다.

화살표 방향이 단지 해석의 문제라면, 프로그래밍에서 합 타입(영문: sum types)과 곱 타입(영문: product types)의 차이는 무엇인가요? 그 차이는 우리가 처음에 가정한 한 가지 가정으로 돌아갑니다: 초기 객체(영문: initial object)에는 (동일성 화살표 외에) 들어오는 화살표가 없다는 것입니다. 이는 종단 객체(영문: terminal object)에는 많은 나가는 화살표가 있다는 것과 대조됩니다. 우리는 이러한 나가는 화살표들을 (전역) 요소 정의에 사용했습니다. 사실, 우리는 관심 있는 모든 객체가 요소를 가지며, 가지지 않는 것들은 `Void`와 동형(영문: isomorphic)이라고 가정합니다.

함수 타입에 대해 이야기할 때 더 깊은 차이점을 보게 될 것입니다.

## 5.3 Monoidal Category

우리는 그 곱이 다음의 간단한 규칙들을 만족함을 보았습니다:

$$1 \times a \cong a$$
$$a \times b \cong b \times a$$
$$(a \times b) \times c \cong a \times (b \times c)$$

그리고 이는 함자적(functorial)입니다.

이러한 성질을 가진 연산이 정의된 범주를 *대칭 모노이드 범주*(symmetric monoidal)라고 합니다[1]. 우리는 합과 초기 객체를 다룰 때 유사한 구조를 본 적이 있습니다.

카테고리는 동시에 여러 모노이드 구조를 가질 수 있습니다. 모노이드 구조에 이름을 붙이고 싶지 않을 때는 더하기 기호(+)나 곱하기 기호(×)를 텐서 기호()로 대체하고, 중립 원소를 문자 $I$로 대체합니다. 대칭 모노이드 카테고리의 규칙들은 다음과 같이 쓸 수 있습니다:

$$I \otimes a \cong a$$
$$a \otimes b \cong b \otimes a$$
$$(a \otimes b) \otimes c \cong a \otimes (b \otimes c)$$

이 이성질체(isomorphisms)들은 종종 결합자(associators) 및 단위자(unitors)라는 가역적인 화살표의 집합으로 작성됩니다. 모노이달 카테고리가 대칭이지 않다면, 왼쪽 단위자와 오른쪽 단위자가 분리되어 있습니다.

$$\alpha : (a \otimes b) \otimes c \to a \otimes (b \otimes c)$$
$$\lambda : I \otimes a \to a$$
$$\rho : a \otimes I \to a$$

대칭은 다음과 같이 드러납니다:

$$\gamma : a \otimes b \to b \otimes a$$

함자성(Functoriality)은 우리가 쌍의 화살표들을 *끌어올릴* 수 있게 해줍니다:

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

텐서 곱에 대해 작용하기:

$$a \otimes b \xrightarrow{f \otimes g} a' \otimes b'$$

사상들을 동작으로 생각해 본다면, 이들의 텐서곱(tensor product)은 두 동작이 병렬로 수행되는 것에 해당합니다. 이는 사상들의 연속적 합성(serial composition)과 대조되며, 이는 시간적 순서를 암시합니다.

텐서곱(tensor product)을 곱(product)과 합(sum)의 최소 공약수로 생각할 수 있습니 다. 여전히 소개 규칙( introduction rule)을 가지며, 이는 두 객체(object) $a$ 와 $b$ 를 필요로 합니다; 그러나 제거 규칙(elimination rule)은 없습니다. 한 번 생성되면, 텐서곱은 어떻게 생성되었는지를 "잊어버립니다." 데카르트 곱(cartesian product)과는 달리, 투영(projections)이 없습니다.

일부 텐서곱(tensor products)에는 대칭(symmetry)이 없는 흥미로운 예들이 있습니니다.

## Monoids

Monoids는 이항 연산 및 유닛을 갖춘 매우 간단한 구조입니다. 덧셈과 0을 갖는 자연수는 *monoid*를 형성합니다. 곱셈과 1을 갖는 자연수도 마찬가지입니다.

직관적으로 설명하자면, 모노이드(monoid)는 두 개의 것을 결합하여 또 다른 것을 얻을 수 있게 합니다. 또한 하나의 특별한 것이 있어서, 그것을 다른 어떤 것과 결합해도 원래의

[1]엄밀히 말하면, 두 객체의 곱은 동형사상에 따라서 정의되지만, 모노이드 범주에서의 곱은 정확히 정의되어야 합니다. 하지만 우리는 하나의 곱을 선택함으로써 모노이드 범주를 얻을 수 있습니다

같은 것을 반환합니다. 그것이 유닛(unit)입니다. 그리고 결합 연산은 결합 법칙(associative)을 따라야 합니다.

조합이 대칭적이라는 것과 역원(역원, inverse element)이 존재한다는 것은 가정되지 않습니다.

모노이드를 정의하는 규칙들은 범주의 규칙들을 연상케 합니다. 차이점은, 모노이드에서는 어떤 두 개의 것이라도 합성 가능하지만, 범주에서는 일반적으로 그렇지 않다는 점입니다: 하나의 화살표의 목표(target)가 다른 하나의 출발지(source)일 때만 두 개의 화살표를 합성할 수 있습니다. 범주에 오직 하나의 객체(object)만 포함되어 있는 경우를 제외하면, 이 때는 모든 화살표들이 합성 가능합니다.

하나의 객체(object)만 가진 범주(category)는 모노이드(monoid)라고 합니다. 결합 연산은 화살표(arrows)의 합성(composition)이고, 단위원(unit)은 항등 화살표(identity arrow)입니다.

이는 완벽히 유효한 정의입니다. 그러나 실제로는 더 큰 카테고리에 포함된 모노이드에 관심이 있는 경우가 많습니다. 특히 프로그래밍에서는 타입과 함수의 카테고리 내에서 모노이드를 정의할 수 있기를 원합니다.

그러나, 카테고리 이론에서는 개별 요소들보다는 일괄적으로 연산을 정의하는 것을 선호합니다. 따라서 우리는 객체 $m$으로 시작합니다. 이항 연산은 두 개의 인자를 가지는 함수입니다. 곱셈의 요소는 두 요소의 쌍이기 때문에, 이항 연산은 $m \times m$에서 $m$으로의 화살(arrow)로 특징지을 수 있습니다:

$$\mu : m \times m \to m$$

단위 원소는 종점 객체 1에서 화살표로 정의될 수 있습니:

$$\eta : 1 \to m$$

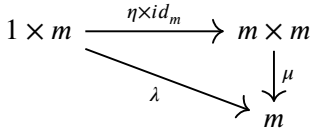
우리는 이 설명을 Haskell에 직접 변환할 수 있습니다. 두 가지 메서드를 갖춘 유형(type) 클래스를 정의하여, 전통적으로 `mappend`와 `mempty`라고 불립니다:

```
class Monoid m where
  mappend :: (m, m) -> m
  mempty  :: () -> m
```

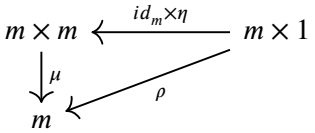
두 사상 $\mu$와 $\eta$는 모노이드 법칙을 만족해야 하지만, 다시 한 번, 우리는 요소에 의지하지 않고 일괄적으로 이를 공식화해야 합니다.

왼쪽 단위원 법칙을 제정하기 위해, 먼저 곱 $1 \times m$을 만듭니다. 그런 다음 $\eta$를 사용하여 "m에서 단위원 요소를 선택"하거나, 화살표(arrow)로 표현하면 1을 $m$으로 바꿉니다. 곱 $1 \times m$에서 작동하고 있으므로, $\langle \eta, id_m \rangle$ 쌍을 들어 올려서 $m$을 "터치하지 않도록" 해야 합니다. 마지막으로 $\mu$를 사용하여 "곱셈"을 수행합니다.

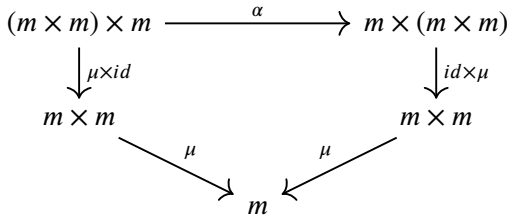
우리는 결과가 원래의 $m$ 요소와 동일하길 원하지만 요소를 언급하지 않길 원합니다. 그래서 우리는 아무 것도 건드리지 않고 $1 \times m$에서 $m$으로 가기 위해 좌단위자 $\lambda$를 사용합니다.




다음은 오른쪽 단위에 대한 유사한 법칙입니다:



결합 법칙(associativity 법칙)을 공식화하려면, 우리는 세 곱셈 항에서 시작하여 전체적으로 작용해야 합니다. 여기서, $\alpha$는 곱셈을 "섞지 않고(stirring things up)" 재배열하는 결합자(associator)입니다.




우리가 객체 $m$과 1과 함께 사용한 범주적곱(categorical product)에 대해 많은 것을 가정할 필요가 없었다는 것을 주의하세요. 특히 사영(projections)을 사용할 필요가 없었습니다. 이는 위 정의가 임의의 모노이달 범주에 있는 텐서곱(tensor product)에도 동일하게 잘 작동할 것이라는 것을 시사합니다. 대칭적일 필요도 없습니다. 우리가 가정해야 할 것은: 단위 객체가 있으며, 곱이 함자적이고(functorial), 단위와 결합 법칙이 동형사상까지 만족한다는 것입니다.

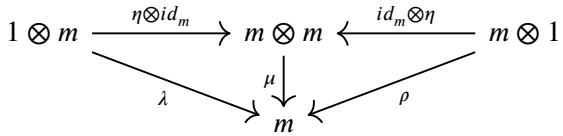
따라서 $\times$를 $\otimes$로, 1을 $I$로 대체하면 임의의 모노이드 범주에서 모노이드의 정의를 얻게 됩니다.

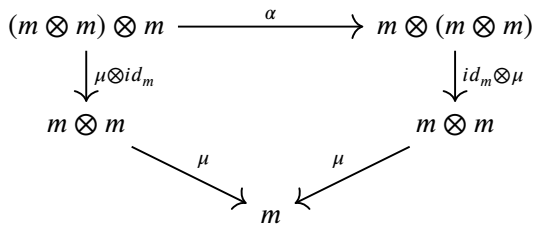
A *모노이드*는 모노이달 범주에서 두 가지 사상과 함께 주어지는 객체 $m$입니다:

$$\mu : m \otimes m \to m$$

$$\eta : I \to m$$

단위(unit)와 결합 법칙(associativity laws)을 만족시키면서:




우리는 $\otimes$의 함자성(functoriality) 덕분에 화살의 쌍을 들어 올렸습니다. 예를 들어 $\eta \otimes id_m$, $\mu \otimes id_m$ 등입니다.