

Chapter 7

재귀

두 거울 사이에 서면 자신의 모습, 자신의 모습의 반사, 그 반사의 반사 등을 보게 됩니다. 각각의 반사는 이전 반사를 통해 정의되지만, 함께 무한을 만들어냅니다.

재귀는 단일 작업을 여러 단계로 나누는 분해 패턴으로, 그 단계의 수는 잠재적으로 무한합니다.

귀납법은 불신의 일시적 중지에 기반을 둡니다. 여러분은 임의의 많은 단계가 필요할 수 있는 작업에 직면해 있습니다. 여러분은 그 작업을 해결하는 방법을 알고 있다고 가정합니다. 그 후 자신에게 질문을 던집니다: "마지막 단계를 제외한 모든 것에 대한 해답을 가지고 있다면, 마지막 단계를 어떻게 해결할 것인가?"

7.1 자연수

자연수의 대상 N 은 숫자를 포함하지 않습니다. 대상은 내부 구조를 가지고 있지 않습니다. 구조는 화살표에 의해 정의됩니다.

종단 객체로부터의 화살표를 사용하여 하나의 특별한 원소를 정의할 수 있습니다. 관례적으로, 우리는 이 화살표를 "제로"를 의미하는 Z 라고 부릅니다.

$$Z : 1 \rightarrow N$$

하지만, 모든 자연수에 대해 그 수보다 하나 큰 다른 수가 있다는 사실을 설명하기 위해 무한히 많은 화살표를 정의할 수 있어야 합니다.

이 명제를 형식화할 수 있습니다: 자연수 $n : 1 \rightarrow N$ 를 만드는 방법을 알고 있다고 가정합시다. 그 다음 단계, 즉 다음 수인 후속자를 가리키는 단계를 어떻게 만들까요?

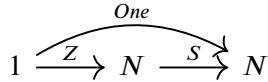
다음 단계는 n 을 N 에서 N 으로 되돌아오는 화살표와 단순히 합성하는 것보다 더 복잡할 필요는 없습니다. 이 화살표는 정체성이 되어서는 안 됩니다. 왜냐하면 숫자의 후속자가 그 숫자와 다르길 원하기 때문입니다. 그러나 단일 그런 화살표, 즉 "후속자"를 의미하는 S 라고 부르는 것으로 충분합니다.

n 의 후속자에 해당하는 요소는 다음과 같이 합성됩니다:

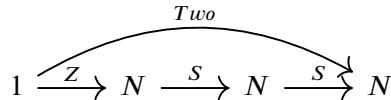
$$1 \xrightarrow{n} N \xrightarrow{S} N$$

(반복되는 화살표를 곧게 그리기 위해, 때때로 동일한 객체를 하나의 다이어그램에서 여러 번 그립니다.)

특히, *One*을 *Z*의 후속자로 정의할 수 있습니다:



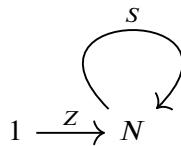
그리고 *Two*는 *Z*의 다음으로 간주됩니다.



등을 포함합니다.

소개 규칙

두 화살표 *Z*와 *S*는 자연수 객체 *N*에 대한 소개 규칙으로 사용됩니다. 한 가지 특이한 점은 이들 중 하나가 재귀적이라는 것입니다: *S*는 *N*을 소스뿐만 아니라 타겟으로도 사용합니다.



두 개의 도입 규칙은 Haskell로 직접 번역됩니다.

```

data Nat where
  Z :: Nat
  S :: Nat -> Nat
  
```

이를 사용하여 임의의 자연수를 정의할 수 있습니다; 예를 들어:

```

zero, one, two :: Nat
zero = Z
one  = S zero
two  = S one
  
```

이 자연수 타입의 정의는 실제로는 크게 유용하지 않습니다. 그러나 이는 각 숫자가 고유한 타입인 타입 수준의 자연수를 정의할 때 자주 사용됩니다.

이 구조는 페아노 산술이라는 이름으로 접할 수 있습니다.

소멸 규칙

도입 규칙들이 재귀적인 사실은 소멸 규칙 정의를 약간 복잡하게 만듭니다. 우리는 먼저 *N*의 사상을 가정하는 방식에서 시작하는 이전 장의 패턴을 따르겠습니다:

$$h: N \rightarrow a$$

그리고 거기서 무엇을 추론할 수 있는지 봅시다.

이전에는 그러한 *h*를 더 간단한 사상들로 분해할 수 있었습니다 (합과 곱에 대한 두 쌍의 사상; 지수함수에 대한 곱의 사상).

*N*에 대한 도입 규칙은 합에 대한 규칙과 유사합니다 (*Z*이거나 후속자이기 때문에), 그래서 *h*가 두 개의 화살로 분리될 수 있다고 예상할 수 있습니다. 실제로 *h* \circ *Z*를 조합하여 우리는

첫 번째 화살을 쉽게 얻을 수 있습니다. 이것은 a 의 요소를 선택하는 화살입니다. 이를 *init*이라고 부릅니다:

$$init : 1 \rightarrow a$$

하지만 두 번째 화살을 찾는 명확한 방법은 없습니다.

그것을 보기 위해, N 의 정의를 확장해봅시다:

$$1 \xrightarrow{Z} N \xrightarrow{S} N \xrightarrow{S} N \dots$$

h 와 *init*을 여기에 대입합니다:

$$1 \xrightarrow{Z} N \xrightarrow{S} N \xrightarrow{S} N \dots$$

$init \searrow \downarrow h \quad \downarrow h \quad \downarrow h$

$a \quad a \quad a$

화살표 N 에서 a 로 가는 직관은 a 의 요소들로 구성된 순서 a_n 을 나타냅니다. 제로 번째 요소는

$$a_0 = init$$

다음 요소는

$$a_1 = h \circ S \circ Z$$

그 다음은

$$a_2 = h \circ S \circ S \circ Z$$

그리고 계속 이어집니다.

따라서 하나의 화살표 h 를 무한히 많은 화살표 a_n 으로 대체했습니다. 물론, 새 화살표들은 a 의 요소들을 나타내기 때문에 더 단순하지만, 그 수는 무한합니다.

문제는, 어떤 식으로 보든지 N 으로부터의 임의의 매핑은 무한한 양의 정보를 포함하고 있다는 것입니다.

이 문제를 급격하게 단순화해야 합니다. 단일 화살표 S 를 사용하여 모든 자연수를 생성했던 것처럼, 모든 a_n 요소들을 생성하기 위해 단일 화살표 $a \rightarrow a$ 를 사용해 볼 수 있습니다. 이 화살표를 *step*이라고 부르겠습니다:

$$1 \xrightarrow{Z} N \xrightarrow{S} N$$

$init \searrow \downarrow h \quad \downarrow h$

$a \xrightarrow{step} a$

N 에서 생성된 이러한 쌍, *init* 및 *step*에 의한 사상들을 재귀적이라고 합니다. 모든 N 의 사상이 재귀적인 것은 아닙니다. 사실, 아주 적은 수의 사상만이 재귀적입니다. 하지만 재귀적 사상들만으로 자연수 객체를 정의하기에 충분합니다.

우리는 위의 다이어그램을 제거 규칙으로 사용합니다. 우리는 N 에서 나오는 모든 재귀적 사상 h 가 쌍 *init* 및 *step*와 일대일 대응 관계에 있음을 선언합니다.

이는 임의의 화살표 $h : N \rightarrow a$ 에 대해 평가 규칙(주어진 h 에 대해 $(init, step)$ 를 추출하는 규칙)을 공식화할 수 없음을 의미합니다. 오직 이전에 *init* 및 *step* 쌍을 사용하여 재귀적으로 정의된 화살표에 대해서만 가능합니다.

화살표 *init*는 항상 $h \circ Z$ 를 합성하여 회복할 수 있습니다. 화살표 *step*는 다음 방정식의 해입니다:

$$step \circ h = h \circ S$$

만약 h 가 어떤 *init* 및 *step*을 사용하여 정의되었다면, 이 방정식은 명백히 해를 가집니다.

중요한 부분은 이 해가 유일해야 한다는 점입니다.

직관적으로, 쌍 *init*과 *step*는 요소의 순서 a_0, a_1, a_2, \dots 를 생성합니다. 만약 두 화살표 *h*와 *h'*가 동일한 쌍 (*init*, *step*)에 의해 주어진다면, 이는 이들이 생성하는 순서가 동일함을 의미합니다.

따라서 만약 *h*가 어떤 방식으로든 *h'*와 다르다면, *N*은 단순히 $Z, SZ, S(SZ), \dots$ 의 요소의 순서만을 포함하는 것이 아닐 것입니다. 예를 들어, *N*에 -1 을 추가한다면 (즉, *Z*를 누군가의 후계자로 만든다면), 우리는 *h*와 *h'*가 -1 에서 다르지만 동일한 *init*와 *step*에 의해 생성될 수 있습니다. 유일성은 *Z*와 *S*에 의해 생성되는 숫자들 사이에 자연수가 없거나, 전후 관계에 있는 숫자가 존재하지 않음을 의미합니다.

여기서 논의된 제거 규칙은 원시 재귀에 해당합니다. 우리는 종속 타입의 장에서 귀납 원리에 해당하는 이 규칙의 더 발전된 버전을 살펴볼 것입니다.

프로그래밍에서

소거 규칙은 Haskell에서 재귀 함수로 구현될 수 있습니다:

```
rec :: a -> (a -> a) -> (Nat -> a)
rec init step = \n ->
  case n of
    Z      -> init
    (S m) -> step (rec init step m)
```

이 단일 함수는 *되추적기*(recursor)라고 하며, 자연수의 모든 재귀 함수를 구현하기에 충분합니다. 예를 들어, 덧셈을 이렇게 구현할 수 있습니다:

```
plus :: Nat -> Nat -> Nat
plus n = rec init step
  where
    init = n
    step = S
```

이 함수는 *n*을 인자로 받아 또 다른 숫자를 입력 받아 *n*을 더하는 함수를 생성합니다(클로저).

실제로, 프로그래머들은 재귀를 직접 구현하는 것을 선호합니다. 이는 *rec* 함수를 인라인하는 것과 동등합니다. 다음 구현은 아마도 이해하기 더 쉬울 것입니다:

```
plus n m = case m of
  Z -> n
  (S k) -> S (plus k n)
```

이를 읽을 수 있는 방법: *m*이 0이면 결과는 *n*입니다. 그렇지 않고, *m*이 어떤 *k*의 후속이라면, 결과는 *k* + *n*의 후속입니다. 이는 정확히 *init* = *n*이며 *step* = *S*라고 말하는 것과 같습니다.

명령형 언어에서는 반복이 종종 재귀를 대체합니다. 개념적으로, 반복은 순차적 분해에 해당하여 이해하기 더 쉬운 편입니다. 순차의 각 단계는 일반적으로 자연스러운 순서를 따릅니다. 이는 *n*번째 단계까지 모든 작업을 완료했다고 가정하고 그 결과를 다음 연속 단계와 결합하는 재귀적 분해와는 대조적입니다.

반면, 재귀는 리스트나 트리와 같이 재귀적으로 정의된 데이터 구조를 처리할 때 더 자연스럽습니다.

두 접근법은 동일하며, 컴파일러는 종종 재귀함수를 반복문으로 변환하는 꼬리 재귀 최적화를 수행합니다.

Exercise 7.1.1. 다음의 N 으로부터 함수 객체 N^N 까지의 사상으로 덧셈의 커리화된 버전을 구현합니다. 힌트: 순환기에 이러한 타입들을 사용하십시오:

```
init :: Nat -> Nat
step :: (Nat -> Nat) -> (Nat -> Nat)
```

7.2 리스트

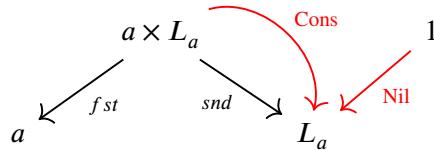
어떤 것들의 리스트는 비어 있거나, 하나의 것 뒤에 또 다른 것들의 리스트가 따라오는 형태입니다.

이 재귀적 정의는 L_a , 즉 a 의 리스트 유형에 대한 두 가지 도입 규칙으로 번역됩니다:

$$\begin{aligned} \text{Nil} &: 1 \rightarrow L_a \\ \text{Cons} &: a \times L_a \rightarrow L_a \end{aligned}$$

Nil 원소(element)는 빈 리스트를 나타내며, Cons 는 머리(head)와 꼬리(tail)로부터 리스트를 구성합니다.

다음 다이어그램은 프로젝션(projections)과 리스트 생성자(list constructors) 간의 관계를 묘사합니다. 프로젝션은 Cons 를 사용하여 구성된 리스트의 머리와 꼬리를 추출합니다.



이 설명은 즉시 Haskell로 번역될 수 있습니다:

```
data List a where
  Nil :: List a
  Cons :: (a, List a) -> List a
```

제거 규칙(Elimination Rule)

리스트 a 에서 임의의 타입 c 로의 매팅이 있다고 가정해 봅시다:

$$h : L_a \rightarrow c$$

이것이 우리가 리스트의 정의에 적용하는 방법입니다:

$$\begin{array}{ccccc} 1 & \xrightarrow{\text{Nil}} & L_a & \xleftarrow{\text{Cons}} & a \times L_a \\ & \searrow \text{init} & \downarrow h & \downarrow id_a \times h & \\ & & c & \xleftarrow{\text{step}} & a \times c \end{array}$$

우리는 곱의 함자성(functionality)을 사용하여 짝 (id_a, h) 을 곱 $a \times L_a$ 에 적용하였습니다.

자연수 객체와 유사하게, 두 개의 사상 $init = h \circ \text{Nil}$ 과 $step$ 을 정의할 수 있습니다. 사상 $step$ 은 다음 식을 만족하는 해입니다:

$$step \circ (id_a \times h) = h \circ \text{Cons}$$

다시 말해, 모든 h 가 이런 쌍의 사상으로 축소될 수 있는 것은 아닙니다.

하지만, $init$ 과 $step$ 이 주어진다면, h 를 정의할 수 있습니다. 이런 함수를 폴드(fold)나 리스트 카타모르피즘(catamorphism)이라고 합니다.

이것은 Haskell의 리스트 재귀자(recursor)입니다:

```
recList :: c -> ((a, c) -> c) -> (List a -> c)
recList init step = \as ->
  case as of
    Nil          -> init
    Cons (a, as) -> step (a, recList init step as)
```

$init$ 과 $step$ 을 주어지면, 리스트의 매핑을 생성합니다.

리스트는 매우 기본적인 데이터 타입이라 Haskell에는 이를 위한 내장 구문이 있습니다. 타입 (`List a`)은 `[a]`로 표기됩니다. `Nil` 생성자(constructor)는 빈 대괄호, `[]`이며 `Cons` 생성자는 접미사 콜론 (`:`)입니다.

이 생성자들에 대해 패턴 매칭(pattern-matching)을 할 수 있습니다. 리스트의 일반적인 매핑은 다음과 같은 형태를 가집니다:

```
h :: [a] -> c
h []      = -- empty-list case
h (a : as) = -- case for the head and the tail of a non-empty list
```

리스트 리커서에 해당하는 `recList`과 함께, 표준 라이브러리에서 찾을 수 있는 함수 `foldr` (오른쪽 폴드)의 타입 시그니처는 다음과 같습니다:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

Here's one possible implementation:

```
foldr step init = \as ->
  case as of
    [] -> init
    a : as -> step a (foldr step init as)
```

예를 들면, 우리는 리스트의 자연수들의 합을 계산하기 위해 `foldr`를 사용할 수 있습니다:

```
sum :: [Nat] -> Nat
sum = foldr plus Z
```

Exercise 7.2.1. 리스트 정의에서 a 를 종단 객체로 대체할 때 어떤 일이 일어나는지 고려해 보세요. 힌트: 자연수의 1진법 인코딩(base-one encoding)이 무엇인가요?

Exercise 7.2.2. $h : L_a \rightarrow 1 + a$ 의 매핑이 몇 개나 있나요? 리스트 리커서(list recursor)를 사용하여 그 중 모든 매핑을 얻을 수 있나요? 다음 서명을 가진 Haskell 함수는 어떤가요:

```
h :: [a] -> Maybe a
```

Exercise 7.2.3. 리스트의 세 번째 요소를 추출하는 함수를 구현하세요, 리스트가 충분히 길다면요. 힌트: 결과 타입으로 `Maybe a`를 사용하세요.

7.3 함자성

함자성(Functoriality)은 대략적으로 데이터 구조의 “내용”을 변환할 수 있는 능력을 의미합니다. 리스트 L_a 의 내용은 탑입 a 입니다. 화살표 $f : a \rightarrow b$ 가 주어졌을 때, 리스트의 매팅 $h : L_a \rightarrow L_b$ 를 정의할 필요가 있습니다.

리스트는 매팅 아웃 특성(mapping out property)에 의해 정의되므로, 제거 규칙의 대상 c 를 L_b 로 대체해봅시다. 우리는 다음을 얻습니다:

$$\begin{array}{ccccc}
 & & L_a & \xleftarrow{\text{Cons}_a} & a \times L_a \\
 & \xrightarrow{\text{Nil}_a} & & \xleftarrow{\text{id}_a \times h} & \\
 & \searrow \text{init} & \downarrow h & & \downarrow \\
 L_b & \xleftarrow{\text{step}} & a \times L_b & &
 \end{array}$$

우리는 여기서 두 개의 다른 리스트를 다루고 있으므로, 그들의 생성자를 구분해야 합니다. 예를 들어, 우리는 다음과 같은 것이 있습니다:

$$\begin{aligned}
 \text{Nil}_a &: 1 \rightarrow L_a \\
 \text{Nil}_b &: 1 \rightarrow L_b
 \end{aligned}$$

그리고 Cons에도 마찬가지입니다.

init 의 유일한 후보는 Nil_b 이며, 이는 h 가 빈 a 리스트에 작용할 때 빈 b 리스트를 생성한다는 것을 의미합니다:

$$h \circ \text{Nil}_a = \text{Nil}_b$$

남은 것은 화살표를 정의하는 것입니다:

$$\text{step} : a \times L_b \rightarrow L_b$$

다음과 같이 추측할 수 있습니다:

$$\text{step} = \text{Cons}_b \circ (f \times \text{id}_{L_b})$$

이는 하스켈(Haskell) 함수에 해당합니다:

```

mapList :: (a -> b) -> List a -> List b
mapList f = recList init step
  where
    init = Nil
    step (a, bs) = Cons (f a, bs)
  
```

혹은, 내장된 리스트 구문을 사용하고 재귀자를 인라인하여,

```

map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (a : as) = f a : map f as
  
```

당신은 우리가 $\text{step} = \text{snd}$ 를 선택하는 것을 방해하는 것이 무엇인지 궁금할 수 있습니다. 그 결과:

```

badMap :: (a -> b) -> [a] -> [b]
badMap f [] = []
badMap f (a : as) = badMap f as
  
```

다음 장에서 왜 이것이 나쁜 선택인지 설명하겠습니다. (힌트: id 에 badMap 을 적용하면 어떻게 될까요?)