10장 함수형 언어

10.6 이론적 기초

예 10.45
매핑으로서의 함수

수학적으로 함수는 단일 값 매핑이다. 즉, 함수는 한 집합(정의역)의 모든 원소를 다른 집합(치역)의 (최대) 한 원소에 연결한다. 전통적인 표기법에서는 다음과 같이 정의역과 치역을 나타낸다.

$$\texttt{sqrt} : R \rightarrow R$$

물론 둘 이상의 변수를 가지는 함수, 즉 정의역이 카테시안 곱인 함수를 정의할 수도 있다.

$$\texttt{plus} : [R \times R] \rightarrow R$$

정의역의 모든 원소에 대한 매핑을 제공하는 함수를 **전체 함수**라고 한다. 그 외의 함수는 **부분 함수**다. 위의 sqrt 함수는 부분 함수다. 이 함수는 음수에 대해서는 매핑을 제공하지 않기 때문이다. 함수의 정의역을 음수가 아닌 수로 다시 정의할 수도 있지만 이러한 변경은 가끔 불편하거나 심지어 불가능하기도 하다. 즉, 이러한 변경은 모든 수학 함수가 R에 대해 동작하는 것이 바람직하기 때문에 불편하며, 정의역의 어느 원소가 매핑을 가지고 어느 원소가 매핑을 가지지 않는지를 모를 수 있기 때문에 불가능할 수 있다. 예를 들어 모든 자연수 a를 원주율의 소수점 이하 자리에서 a가 나오는 최소 자릿수를 나타내는 자연수 b로 매핑하는 함수 f를 생각해보자. π=3.14159... 이므로 f(59) = 4라는 것은 분명하다. 하지만 f(428945028)이나 일반적으로 임의의 n에 대한 f(n)은 어떤가? 수 이론의 결과를 모르더라도 f가 정의된 값을 특징짓는 것은 전혀 분명하지 않다. 이런 경우에는 부분 함수를 피할 수 없다.

예 10.46
집합으로서의 함수

가끔 함수를 집합, 좀 더 명확히 말하면 정의역과 치역의 카테시안 곱의 부분 집합으로 특징짓는 것이 유용하다.

$sqrt \subset [R \times R]$

$plus \subset [R \times R \times R]$

전통적인 집합 표기법을 사용해서 어느 부분 집합인지 명시할 수 있다.

$sqrt \equiv \{(x, y) \in R \times R \mid y = x^2\}$

$plus \equiv \{(x, y, z) \in R \times R \times R \mid z = x + y\}$

이런 종류의 정의는 sqrt와 같은 함수의 값이 무엇인지는 말해주지만 이를 어떻게 계산하는지는 말해주지 않는다. 이 차이에 대해서는 아래에서 좀 더 자세히 다룬다.

예 10.47

멱집합 원소로서의 함수

집합 기반 기술의 장점 중 하나는 함수가 보통의 수학적 객체라는 것을 분명하게 해준다는 점이다. A에서 B로의 함수는 $A \times B$의 부분 집합이라는 것을 안다. 이는 이 함수가 $A \times B$의 멱집합의 원소 중 하나라는 것을 의미한다. 멱집합은 $A \times B$의 모든 부분 집합의 집합으로 $2^{A \times B}$로 나타낸다.

$sqrt \in 2^{R \times R}$

이와 마찬가지로

$plus \in 2^{R \times R \times R}$

여기서 표기법의 다중 정의가 쓰였음에 주목하자. 멱집합 2^A를 지수법과 혼동하면 안 된다. 다만 유한 집합 A에 대해 $n=|A|$, 즉 n이 A의 개체 수일 때 유한 집합 A의 멱집합의 원소 수는 2^n이라는 것은 사실이다.

예 10.48

함수 공간

함수는 단일 값 매핑이므로 함수가 $2^{A \times B}$의 원소 중 일부만을 구성한다는 사실을 알 수 있다. 명확히 말해서 함수는 각 쌍의 첫 번째 구성 요소가 유일한 쌍의 모든 집합만을 구성한다. 이러한 집합의 집합을 A에서 B로의 함수 공간이라고 부르고 $A \rightarrow B$로 나타낸다. $(A \rightarrow B) \subset 2^{A \times B}$라는 것에 주목하자. 예를 들어 다음과 같다.

$sqrt \in [R \rightarrow R]$

$plus \in [(R \times R) \rightarrow R]$

예 10.49

집합으로서의 고차 함수

함수가 집합의 원소라는 것을 알면 고차 함수는 좀 더 쉽게 만들 수 있다.

$compose \equiv \{(f, g, h) \mid \forall x \in R, h(x) = f(g(x))\}$

compose의 정의역과 치역은 무엇일까? f, g, h가 $R \rightarrow R$의 원소라는 것을 알기 때문에 다음과 같이 나타낼 수 있다.

$compose \in [(R \rightarrow R) \times (R \rightarrow R)] \rightarrow (R \rightarrow R)$

ML 유형 추론 시스템(7.2.4절)의 표기법과 유사하다는 사실에 주목하자.

예 10.50

집합으로서의 커링된 함수

10.5절의 "커링(currying)" 개념을 사용하면 plus와 같은 함수에 대해서 다른 기술을 정의할 수 있다. 실수 쌍에서 실수로의 함수 대신 실수로부터 실수에서 실수로의 함수에 대한 함수로 표현할 수도 있다.

curried_plus $\in R \rightarrow (R \rightarrow R)$

커링에 대해서는 아래에서 더 자세히 다룬다.

【10.6.1】 람다 연산

책에서 설명했듯이 집합으로 함수를 나타내는 표기법의 제약 중 하나는 비구성적이라는 점이다. 즉, 이 표기법을 통해서는 주어진 점(즉, 주어진 입력)에 대한 함수 값을 어떻게 계산하는지 알 수 없다. 처치는 이러한 한계를 극복하고자 람다 연산을 설계했다. 순수한 형태의 람다 연산은 모든 것을 함수로 나타낼 수 있다. 예를 들어 자연수는 유일한 0(zero) 함수(주로 항등 함수)와 **계승 함수**로 나타낼 수 있다(일반적인 공식화 중 하나에서는 두 개의 인자를 취해 두 번째를 반환하는 select_second 함수를 사용한다. 그러면 숫자 n이 select_second를 n번 적용했을 때 항등 함수를 반환하는 함수에 의해 표현되는 방식으로 계승 함수를 정의할 수 있다[Mic89, 3.5절], [Sta95, 7.6절]. 연습문제 10.23을 보자). 산술의 이러한 공식화는 그 이론적 중요성에도 불구하고 매우 다루기 어렵다. 그러므로 이 부속절의 나머지 부분에서는 보통의 산술을 가정한다(물론 모든 실제 함수형 프로그래밍 언어가 정수와 부동점 실수 산술 모두에 대한 고유 지원을 제공한다).

람다식은 (1) 이름, (2) 글자 λ, 이름, 마침표, 람다 수식으로 구성되는 람다 추상화, (3) 두 개의 인접한 람다 수식으로 구성되는 함수 적용, (4) 괄호로 묶은 람다 수식으로서 재귀적으로 정의할 수 있다. 산술을 지원하기 위해 이 정의가 숫자 리터럴을 허용하게 확장해보자.

예 10.51

함수 적용으로서의 병기/병치

두 수식이 서로 인접한 경우 첫 번째 수식을 두 번째 수식에 적용할 함수로 해석한다.

sqrt *n*

대부분의 저자들은 적용의 우선순위가 왼쪽에서 오른쪽이라고 가정한다(그러므로 f A B는 f (A B)가 아니라 (f A) B로 해석된다). 또 적용이 추상화보다 높은 우선순위를 가진다고 가정한다(그러므로 λx.AB는 (λx.A)B가 아니라 λx.(AB)로 해석된다). ML도 이러한 규칙을 사용한다.

예 10.52

람다 연산 구문

괄호는 기본 그룹화를 재정의하기 위해 사용한다. 명확히 말하면 함수로 사용하는 람다 수식과 인자로 사용하는 람다 수식을 구별한다면 다음과 같이 모호하지 않은 CFG를 사용해서 최소 개수의 괄호를 가지는 람다 수식을 생성할 수 있다.

expr → name | *number* | λ name . *expr* | *func arg*

func → name | (λ name . *expr*) | *func arg*
arg → name | *number* | (λ name . *expr*) | (*func arg*)

말로 풀어 쓰면 함수나 인자로 사용되는 추상화 주변과 인자로 사용되는 적용 주위를 괄호로 감쌌다.

예 10.53
λ를 이용한 매개변수 바인딩

글자 λ를 사용해서 람다 연산에 형식 매개변수와 동일한 것을 도입할 수 있다. 다음과 같은 람다 수식은 인자의 제곱을 반환하는 함수를 나타낸다.

λ*x*.times *x x*

λ에 의해 도입된 이름(변수)은 마침표 다음에 나오는 수식 내부에서 바인딩된다. 프로그래밍 언어의 용어로 말하면 이 수식이 변수의 유효 범위가 된다. 바인딩되지 않은 변수는 **자유 변수**라고 한다.

예 10.54
자유 변수

어휘적으로 유효 범위가 정해지는 프로그래밍 언어에서와 마찬가지로 자유 변수는 어떤 주변 유효 범위에서 정의될 필요가 있다. 예를 들어 수식 λx.λy.times.xy를 생각해보자. 안쪽 수식 (λy.times.xy)에서 y는 바인딩되어 있지만 x는 자유 변수다. 바인딩된 변수의 사용에는 아무런 제약이 없다. 바인딩된 변수는 함수, 인자, 혹은 둘 모두의 역할을 할 수 있다. 그러므로 매우 자연스럽게 고차 함수를 생각할 수 있다.

예 10.55
향후 참조를 위한 명명 함수

수식을 추후에 참조하고자 하는 경우에는 수식에 이름을 부여할 수 있다.

square ≡ λ*x*.times *x x*
identity ≡ λ*x*.*x*
const7 ≡ λ*x*.7
hypot ≡ λ*x*.λ*y*.sqrt (plus (square *x*)(square *y*))

여기서 ≡은 대략적으로 "...는 ...의 약어다"를 의미하는 메타기호다.

예 10.56
값 계산 규칙

람다 연산을 이용해서 계산하기 위해서는 수식을 값 계산하기 위한 규칙이 필요하다. 다음과 같은 세 가지 규칙으로 충분하다는 것이 증명된 바 있다.

- **베타 축약**: 임의의 람다 추상화 λx.E와 임의의 수식 M에 대해 다음과 같이 쓸 수 있다.

  $(\lambda x.E)M \rightarrow_{\beta} E[M\backslash x]$

  여기서 $E[M\backslash x]$는 모든 자유 변수 x를 M으로 대체한 수식 E를 나타낸다. M의 자유 변수가 $E[M\backslash x]$에서 바인딩되는 경우에는 베타 축약을 허용하지 않는다.

- **알파 변환**: 임의의 람다 추상화 $\lambda x.E$와 E에서 자유 변수로 나오지 않는 임의의 변수 y에 대해서는 다음과 같이 쓸 수 있다.

  $\lambda x.E \rightarrow_{\alpha} \lambda y.E[y\backslash x]$

■ **에타 축약**: "여분"의 람다 추상화를 제거하기 위한 규칙이다. E가 Fx 형태며 x가 F에서 자유 변수로 나오지 않는 임의의 람다 추상화 $\lambda x.E$에 대해 다음과 같이 쓸 수 있다.

$$\lambda x.Fx \rightarrow_{\eta} F$$

예 10.57
산술을 위한 델타 축약

산술을 지원하기 위해서는 op *x y*와 같은 형태의 수식도 허용해야 한다. 여기서 x와 y는 숫자 리터럴이며 op는 소규모의 표준 함수 집합 중 하나로 이러한 형태의 수식은 계산 후 산술 값으로 대체된다. 이 대체를 **델타 축약**이라고 한다. 앞서 살펴본 예에서는 다음과 같이 plus, minus, times 함수만 있으면 된다.

```
plus 2 3 →δ5
minus 5 2 →δ3
times 2 3 →δ6
```

예 10.58
에타 축약

베타 축약은 이름에 의한 호출 매개변수의 사용(8.3.1절)과 유사하다. 그러나 알골 60과 달리 람다 연산은 인자가 자신의 참조 환경을 함께 가져올 수 있는 방법을 제공하지 않는다. 그러므로 인자는 변수명이 다른 의미를 가지는 유효 범위로 변수를 이동시키면 안 된다. 알파 변환은 베타 축약을 가능하게 하기 위해 이름을 변경하는 역할을 한다. 에타 축약은 상대적으로 덜 중요하다. 위와 같이 square를 정의한다면 에타 축약을 통해 다음과 같이 쓸 수 있다.

$$\lambda x.\texttt{square}\ x \rightarrow_{\eta} \texttt{square}$$

우리말로 하면 square는 인자를 제곱하는 함수다. 즉, $\lambda x.$square x는 x를 제곱하는 x에 대한 함수다. 후자는 square가 함수(즉, 인자를 취한다)라는 것을 명시적으로 상기시켜주지만 전자가 약간 더 명확해 보인다.

예 10.59
가장 간단한 형태로의 축약

베타 축약과 알파 변환(그리고 아마도 에타 축약)의 반복적인 적용을 통해 람다 수식을 가장 간단한 형태로 축약하고자 시도할 수 있다. 가장 간단한 형태는 더 이상 베타 축약이 불가능한 형태를 말한다. ©(심화학습에 있는)그림 10.2와 같은 예가 가능하다. 그림 10.2에 나온 유도의 행 (2)에서는 g로 치환해 넣어야 할 인자가 g의 유효 범위 안에서 바인딩되어 있는 자유 변수(h)를 포함하기 때문에 알파 변환을 적용해야 한다. 바인딩된 h를 (k로) 재명명하지 않고 행 (3)에서 치환을 수행하면 자유 변수 h도 적용되면서 수식의 의미가 잘못된 방향으로 변하게 된다.

이 유도의 행 (5)에서 어느 하위식을 축약할지 정할 수 있다. 이 시점에서 수식 전체는 인자가 함수 적용 자체인 하나의 함수 적용으로 구성된다. 여기서는 값 계산이 되지 않은 주 인자($(\lambda x.xx)\ (\lambda x.xx)$)를 주 람다 추상화의 몸체로 치환하게 선택했다. 이러한 선택을 **정규 순서 축약**이라고 하며, 이는 6.6.2절과 10.4절에서 논의한 프로그래밍 언어의 인자의 정규 순서 값 계산과 일치한다. 일반적으로 정규 순서 축약은 둘 이상의

베타 축약이 가능할 때마다 전체 수식에서 λ가 가장 왼쪽에 있는 것을 선택한다. 이러한 방법은 인자를 축약하기 전에 함수로 치환한다. 다른 방법으로는 **적용 순서 축약**이 있으며, 이는 모든 함수 적용의 인자 부분을 함수 적용의 함수 부분으로 치환하기 전에 인자 부분과 함수 부분 모두를 가장 간단한 형태로 축약한다.

$$
\begin{aligned}
& (\lambda f.\lambda g.\lambda h.\, f\, g\, (h\, h))\ (\lambda x.\lambda y.\, x)\ h\ (\lambda x.\, x\ x) \\
\rightarrow_{\beta}\ & (\lambda g.\lambda h.\, (\lambda x.\lambda y.\, x)\ g\ (h\, h))\ h\ (\lambda x.\, x\ x) && (1)\\
\rightarrow_{\alpha}\ & (\lambda g.\lambda k.\, (\lambda x.\lambda y.\, x)\ g\ (k\, k))\ h\ (\lambda x.\, x\ x) && (2)\\
\rightarrow_{\beta}\ & (\lambda k.\, (\lambda x.\lambda y.\, x)\ h\ (k\, k))\ (\lambda x.\, x\ x) && (3)\\
\rightarrow_{\beta}\ & (\lambda x.\lambda y.\, x)\ h\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x)) && (4)\\
\rightarrow_{\beta}\ & (\lambda y.\, h)\ ((\lambda x.\, x\ x)\ (\lambda x.\, x\ x)) && (5)\\
\rightarrow_{\beta}\ & \mathrm{h} && (6)
\end{aligned}
$$

그림 10.2 | 람다 수식의 축약. 맨 윗줄은 세 개의 인자를 적용한 하나의 함수로 구성된다. 첫 번째 인자(밑줄)는 "첫 번째 인자 선택" 함수로 두 개의 인자를 취해 첫 번째 인자를 반환한다. 두 번째 인자는 기호 h로 상수나 어떤 둘러싸는 유효 범위(여기서는 보이지 않음)에 바인딩된 변수여야 한다. 세 번째 인자는 "자기 자신으로의 적용" 함수로 하나의 인자를 취해 이를 자기 자신에게 적용한다. 이 그림에서 보인 축약들은 정규 순서다. 축약은 가장 간단한(정규) 형태인 h로 끝난다.

예 10.60

끝나지 않는 적용 순서 축약

처치와 로서는 1936년에 가장 간단한 형태가 유일하다는 것을 증명했다. 그러므로 더 이상 축약할 수 없는 형태의 수식으로 끝나는 모든 종류의 축약이 동일한 결과를 산출한다. 그러나 모든 축약이 종료되는 것은 아니다. 특히 어떤 순서의 축약으로도 끝나지 않는 수식도 있으며, 정규 순서 축약으로는 끝나지만 적용 순서 축약으로는 끝나지 않는 수식도 있다. 그림 10.2의 수식에 적용 순서 축약을 적용하면 무한 "계산"이 초래된다. 이를 확인하기 위해 행 (5)의 수식을 보자. 이 행은 인자 $(\lambda x.\, xx)\ (\lambda x.\, xx)$에 적용된 상수 함수$(\lambda y.\, h)$로 구성된다. 인자를 함수로 치환하기 전에 먼저 인자를 값 계산하려고 하면 다음과 같은 단계들을 거치게 된다.

$$
\begin{aligned}
& (\lambda x.\, x\ x)\ \ (\lambda x.\, x\ x) \\
\rightarrow_{\beta}\ & (\lambda x.\, x\ x)\ \ (\lambda x.\, x\ x) \\
\rightarrow_{\beta}\ & (\lambda x.\, x\ x)\ \ (\lambda x.\, x\ x) \\
\rightarrow_{\beta}\ & (\lambda x.\, x\ x)\ \ (\lambda x.\, x\ x) \\
& \ldots
\end{aligned}
$$

처치와 로서는 가장 간단한(정규) 형태의 유일성을 증명한 것 외에 어떤 값 계산 순서가 종료된다면 정규 순서는 반드시 끝난다는 것도 증명했다. 이러한 두 개의 증명 결과를 **처치–로써 정리**라고 한다.

【10.6.2】 제어 흐름

이전 부속절의 처음 부분에서는 람다 연산에서 유일한 0 함수(보통 항등 함수)와 계승 함수를 사용해서 산술을 모델화할 수 있다는 것을 알아보았다. 제어 흐름 구성소, 특히 선택과 재귀의 경우는 어떨까?

예 10.61
논리형 값과 조건 값

select_first 함수, $\lambda x.\lambda y.x$는 보통 논리형 값 참(true)을 나타내기 위해 사용한다. 그리고 select_second 함수, $\lambda x.\lambda y.y$는 보통 논리형 값 거짓(false)을 나타내기 위해 사용한다. 이 값들을 각기 T와 F로 나타내자. 이러한 정의의 장점은 이를 이용해서 매우 쉽게 if 함수를 정의할 수 있다는 데 있다.

$$\text{if} \equiv \lambda c.\lambda t.\lambda e.c\ t\ e$$

다음을 생각해보자.

$$\begin{aligned}
\text{if}\ T\ 34 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)(\lambda x.\lambda y.x)34 \\
&\rightarrow^*_\beta (\lambda x.\lambda y.x)34 \\
&\rightarrow^*_\beta 3 \\
\text{if}\ F\ 34 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)(\lambda x.\lambda y.y)34 \\
&\rightarrow^*_\beta (\lambda x.\lambda y.y)34 \\
&\rightarrow^*_\beta 4
\end{aligned}$$

equal과 greater_than과 같은 함수도 숫자 값을 인자로 취해 T나 F를 반환하게 정의할 수 있다.

예 10.62
재귀를 위한 베타 추상화

재귀는 약간 까다롭다. 다음과 같은 방정식은 gcd가 좌편과 우편 모두에 나오기 때문에 실제로 정의가 될 수 없다. 앞선 정의들(T, F, if)은 단순히 줄여 쓴 것이었다. 즉, 이러한 정의를 치환해서 순수한 람다 수식을 얻을 수 있다. 이를 gcd에 시도하면 gcd라는 이름이 새로 나오면서 "정의"가 점차 더 커지게 된다. 실제 정의를 얻기 위해서는 우선 베타 추상화(베타 축약의 반대)를 이용해서 방정식을 다시 써야 한다.

$$\begin{aligned}
\text{gcd} \equiv\ & (\lambda g.\lambda a.\lambda b.(\text{if}(\text{equal}\ a\ b)\ a \\
& \quad (\text{if}(\text{greater_than}\ a\ b)(g(\text{minus}\ a\ b)\ b)(g(\text{minus}\ b\ a)\ a)))\ \text{gcd}
\end{aligned}$$

이제 방정식은 다음과 같은 형태를 갖게 된다.

$$\text{gcd} \equiv f\ \text{gcd}$$

여기서 f는 다음과 같이 완벽하게 잘 정의된(비재귀적) 람다 수식이다.

$$\begin{aligned}
& (\lambda g.\lambda a.\lambda b.(\text{if}(\text{equal}\ a\ b)\ a \\
& \quad (\text{if}(\text{greater_than}\ a\ b)(g(\text{minus}\ a\ b)\ b)(g(\text{minus}\ b\ a)\ a)))
\end{aligned}$$

gcd는 f의 명백한 고정점이다.

예 10.63

고정점 결합자 Y

람다 수식에 의해 주어진 임의의 함수 f에 대해 f의 최소 고정점이 있는 경우 주로 Y로 나타내는 다음과 같은 고정점 결합자를 적용함으로써 이를 찾을 수 있다는 사실은 증명되어 있다.

$$\lambda h.(\lambda x.h(xx))(\lambda x.h(xx))$$

Y는 임의의 람다 수식 f에 대해 Yf의 정규 순서 값 계산이 종료되면 f(Yf)와 Yf는 동일한 최소 형태로 축약된다(연습문제 10.21을 보자)는 특징을 가진다. gcd 함수의 예에서는 다음과 같다.

$$\begin{aligned}\texttt{gcd} \equiv\ & (\lambda h.(\lambda x.h(xx))(\lambda x.h(xx)))\\ & (\lambda g.\lambda a.\lambda b.(\texttt{if}(\texttt{equal}\ ab)a\\ & \quad (\texttt{if}(\texttt{greater\_than}\ ab)(g(\texttt{minus}\ ab)b)(g(\texttt{minus}\ ba)a))))\end{aligned}$$

©(심화학습에 있는) 그림 10.3은 gcd 4 2의 값 계산 과정을 보여준다. Y 결합자의 존재를 가정하는 경우 대부분의 저자는 편의를 위해 함수의 재귀적 "정의"를 허용한다.

$$\begin{aligned}\texttt{gcd24} &\equiv \texttt{Y}f24\\ &\equiv ((\lambda h.(\lambda x.h(xx))(\lambda x.h(xx)))f)24\\ &\rightarrow_\beta ((\lambda x.f(xx))(\lambda x.f(xx)))24\\ &\equiv (kk)24,\ \text{where}\ k \equiv \lambda x.f(xx)\\ &\rightarrow_\beta (f(kk))24\\ &\equiv ((\lambda g.\lambda a.\lambda b.(\texttt{if}(=ab)a(\texttt{if}(>ab)(g(-ab)b)(g(-ba)a))))(kk))24\\ &\rightarrow_\beta (\lambda a.\lambda b.(\texttt{if}(=ab)a(\texttt{if}(>ab)((kk)(-ab)b)((kk)(-ba)a))))24\\ &\rightarrow^*_\beta \texttt{if}(=24)2(\texttt{if}(>24)((kk)(-(24)4)((kk)(-(42)2))\\ &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)(=24)2(\texttt{if}(>24)((kk)(-(24)4)((\texttt{kk})(-(42)2))\\ &\rightarrow^*_\beta (=24)2(\texttt{if}(>24)((kk)(-(24)4)((kk)(-(42)2))\\ &\rightarrow_\delta F2(\texttt{if}(>24)((kk)(-(24)4)((kk)(-(42)2))\\ &\equiv (\lambda x.\lambda y.y)2(\texttt{if}(>24)((kk)(-(24)4)((kk)(-(42)2))\\ &\rightarrow^*_\beta \texttt{if}(>24)((kk)(-(24)4)((kk)(-(42)2)\\ &\rightarrow \ldots\\ &\rightarrow (kk)(-(42)2\\ &\equiv ((\lambda x.\texttt{f}(xx))k)(-(42)2\\ &\rightarrow_\beta (f(kk))(-42)2\\ &\equiv ((\lambda g.\lambda a.\lambda b.(\texttt{if}(=ab)a(\texttt{if}(>ab)(g(-ab)b)(g(-ba)\texttt{a}))))(kk))(-42)2\\ &\rightarrow_\beta (\lambda a.\lambda b.(\texttt{if}(=ab)a(\texttt{if}(>ab)((kk)(-ab)b)((kk)(-ba)a))))(-42)2\end{aligned}$$

그림 10.3 | 재귀적 람다 수식의 값 계산. 책에서 설명한 것과 같이 gcd는 최대공약수에 대한 재귀적 표준 정의의 베타 추상화 f에 적용된 고정점 결합자 Y로 정의된다. 명확히 말하면 Y는 λh.(λx.h(x x)) (λx.h(x x))이며, f는 λg.λa. λb.(if (=ab) a (if (〉ab) (g(−ab) b) (g(−ba) a)))이다. 간결성을 위해 equal, greater_than, minus 대신 =, 〉, ?를 사용했다. 값 계산은 정규 순서로 수행했다.(이어짐)

$\rightarrow {}^{*}_{\beta}$ if(=(−42)2)(−(42)(if(>(−(42)2)((*kk*)(−((−42)2)2)((*kk*)(−(2(−42)) (−42)))

$\equiv$ (λ*c*.λ*t*.λ*e*.*c t e*)

(=(−42)2)(−(42)(if(>(−(42)2)((*kk*)(−((−42)2)2)((*kk*)(−(2(−42))(−42)))

$\rightarrow {}^{*}_{\beta}$ (=(−42)2)(−(42)(if(>(−(42)2)((*kk*)(−((−42)2)2)((*kk*)(−(2(−42))(−42)))

$\rightarrow_{\delta}$ (=22)(−(42)(if(>(−(42)2)((*kk*)(−((−42)2)2)((*kk*)(−(2(−42))(−42)))

$\rightarrow_{\delta}$ *T*(−42)(if(>(−(42)2)((*kk*)(−((−42)2)2)((*kk*)(−(2(−42))(−42)))

$\equiv$ (λ*x*.λ*y*.*x*)(−42)(if(>(−(42)2)((*kk*)(−((−42)2)2)((*kk*)(−(2(−42))(−42)))

$\rightarrow {}^{*}_{\beta}$ (−42)

$\rightarrow_{\delta}$ 2

그림 10.3 | 재귀적 람다 수식의 값 계산

【10.6.3】 구조체

예 10.64

람다 연산 리스트 연산자

숫자와 진리 값을 만들기 위해 함수를 사용하는 것과 마찬가지로 값을 구조체 안에 캡슐화하기 위해 함수를 사용할 수도 있다. 명확성을 위해 스킴 용어를 사용해서 다음과 같이 간단한 리스트 처리 함수들을 정의할 수 있다.

cons $\equiv$ λ*a*.λ*d*.λ*x*.*xad*
car $\equiv$ λ*l*.*l* select_first
cdr $\equiv$ λ*l*.*l* select_second
nil $\equiv$ λ*x*.*T*
null? $\equiv$ λ*l*.*l*(λ*x*.λ*y*.*F*)

여기서 select_first와 select_second는 각기 함수 λ*x*.λ*y*.*x*와 λ*x*.λ*y*.*y*로 참과 거짓을 나타내기 위해 사용한다.

예 10.65

리스트 연산자들의 내용

이러한 정의를 사용하면 다음을 알 수 있다.

car(cons *A B*) $\equiv$ (λ*l*.*l* select_first)(cons *A B*)
$\rightarrow_{\beta}$ (cons *A B*) select_first
$\equiv$ ((λ*a*.λ*d*.λ*x*.*xad*)*A B*) select_first
$\rightarrow {}^{*}_{\beta}$ (λ*x*.*xA B*) select_first
$\rightarrow_{\beta}$ select_first *A B*
$\equiv$ (λ*x*.λ*y*.*x*)*A B*
$\rightarrow {}^{*}_{\beta}$ *A*

cdr(cons *A B*) $\equiv$ (λ*l*.*l* select_second)(cons *A B*)
$\rightarrow_{\beta}$ (cons *A B*) select_second
$\equiv$ ((λ*a*.λ*d*.λ*x*.*xad*)*A B*) select_second

$$\rightarrow^{*}_{\beta} (\lambda x.xA\ B)\ \texttt{select\_second}$$
$$\rightarrow_{\beta} \texttt{select\_second}\ A\ B$$
$$\equiv (\lambda x.\lambda y.y)\ A\ B$$
$$\rightarrow^{*}_{\beta} B$$

$$\texttt{null?}\ \texttt{nil} \equiv (\lambda l.l(\lambda x.\lambda y.\texttt{select\_second}))\ \texttt{nil}$$
$$\rightarrow_{\beta} \texttt{nil}(\lambda x.\lambda y.\texttt{select\_second})$$
$$\equiv (\lambda x.\texttt{select\_first})(\lambda x.\lambda y.\texttt{select\_second})$$
$$\rightarrow_{\beta} \texttt{select\_first}$$
$$\equiv T$$

$$\texttt{null?}\ (\texttt{cons A B}) \equiv (\lambda l.l(\lambda x.\lambda y.\texttt{select\_second}))(\texttt{cons}\ A\ B)$$
$$\rightarrow_{\beta} (\texttt{cons}\ A\ B)(\lambda x.\lambda y.\texttt{select\_second})$$
$$\equiv ((\lambda a.\lambda d.\lambda x.xad)\ A\ B)(\lambda x.\lambda y.\texttt{select\_second})$$
$$\rightarrow^{*}_{\beta} (\lambda x.xA\ B)(\lambda x.\lambda y.\texttt{select\_second})$$
$$\rightarrow_{\beta} (\lambda x.\lambda y.\texttt{select\_second})\ A\ B$$
$$\rightarrow^{*}_{\beta} \texttt{select\_second}$$
$$\equiv F$$

예 10.66
람다 수식의 내포

모든 람다 추상화는 하나의 인자를 가지기 때문에 람다 수식은 본질적으로 커링된다. 일반적으로 람다 추상화를 내포시킴으로써 다중 인자 함수의 효과를 얻는다.

$$\texttt{compose} \equiv \lambda f.\lambda g.\lambda x.f(g\ x)$$

이 수식은 다음과 같이 묶일 수 있다.

$$\lambda f.(\lambda g.(\lambda x.(f(gx))))$$

일반적으로 compose는 인자로 두 개의 함수를 취해 세 번째 함수를 결과로 반환하는 함수로 생각할 수 있다. 그러나 compose는 간단하게 세 개의 인자 f, g, x를 가지는 함수로 생각할 수도 있다. 물론 공식적으로 compose는 하나의 인자를 가지는 함수로 이 인자는 하나의 인자를 가지는 함수로 값 계산되며 차례로 해당 인자는 하나의 인자를 가지는 함수로 값 계산된다.

예 10.67
쌍 인자와 커링

원하는 경우에는 구조-생성 함수를 사용해 (단일) 인자가 하나의 쌍인 커링되지 않은 버전의 compose를 정의할 수 있다.

$$\texttt{paired\_compose} \equiv \lambda p.\lambda x.(\texttt{car}\ p)((\texttt{cdr}\ p)\ x)$$

인자 짝짓기를 일반 규칙으로 간주하면 10.5절에서 스킴으로 했던 것과 마찬가지로 단일 인자 버전을 재생산하는 curry 함수를 작성할 수 있다.

$$\texttt{curry} \equiv \lambda f\ .\lambda a.\lambda b.f(\texttt{cons}\ ab)$$

✓ 확인문제

22. 부분 함수와 전체 함수 사이의 차이는 무엇인가? 이러한 차이는 왜 중요한가?

23. 함수 공간 A → B의 의미는 무엇인가?

24. 베타 축약, 알파 변환, 에타 축약, 델타 축약을 정의하라.

25. 람다 연산의 베타 축약은 하스켈과 같은 비엄격 프로그래밍 언어에서 사용하는 인자의 게으른 값 계산과 어떻게 다른가?

26. 람다 수식을 사용해서 어떻게 논리형 값과 제어 흐름을 표현할 수 있는지 설명하라.

27. 베타 추상화는 무엇인가?

28. Y 결합자란 무엇인가? Y 결합자가 가진 유용한 특징은 무엇인가?

29. 리스트와 같이 구조화된 값을 표현하는 데 람다 수식을 어떻게 사용할 수 있는지 설명하라.

30. 처치–로써 정리는 무엇인가?