# コンピュータサイエンス基礎: 計算可能性（2020 年度）

照井一成（京都大学数理解析研究所）

本稿では, 計算可能性理論の基礎について解説する. 計算可能性理論の根本的な問いは, 「どんな関数が計算可能（でない）か」である. ここでいう「計算可能」とは, 詳しく言えば「コンピュータプログラムで計算できる」ことをいう. それゆえ計算可能な関数とは何かを厳密に定義するには, プログラミング言語, あるいはもっと抽象的なレベルで**計算モデル**を 1 つ定める必要がある. それにはチューリング機械, 再帰的関数定義などいろいろな流儀があるが, 本稿では簡単な（一階）**関数型プログラミング言語**を導入し, その中で書き下せる関数のことを「計算可能」な関数とみなすことにする（実際には「**再帰的関数**」という語を用いる）. プログラミング言語を 1 つ定めるというと恣意的に思われるかもしれないが, 必要十分な表現能力を持つ言語を選び, 計算時間やメモリ使用量といった物理的制約を無視する限り, 「計算可能」の定義は一致することが知られている. それゆえ普遍性は失われない.

ところで, ひとことで「計算可能」といっても, 計算の複雑さにはいろいろな度合いがある. 本稿ではいわゆる計算複雑性の問題（P とか NP とか）には立ち入らないが, 「計算の易しさ」を示す 1 つの目安として, **原始再帰的関数**のクラスを考える. これは再帰的関数の部分クラスであり, 大雑把にいって「プログラム実行前に, あらかじめ計算時間を（一定の関数族により）見積もることができる」ような関数の集まりである. 言い換えれば, 「いつかは必ず計算が終わるとわかっているが, いつ終わるかは実際に計算してみないとわからない」といったことがない関数たちである. 「再帰的」と「原始再帰的」の区別を考えることで, 計算の理解が深まるもの思われる.

以下第 1 章で簡単な集合論の準備を行い, 第 2 章で原始再帰的関数を導入する. そして第 3 章で再帰的関数全体を取り扱うことにする.

## 1　集合論の初歩

あらゆる数理科学において, 初歩的な集合論は必要不可欠な道具である. 本章ではまず 1.1 節で集合論の用語・記法についておさらいをし, 1.2 節以降で関数について基本事項を解説する. 目標は (i) 集合の大小を比較する方法（とくに同数性の概念）を学び, (ii) コンピュータ科学で重要な役割を果たす 2 つの同数性について理解し, (iii) 可算無限集合と非可算無限集合の区別ができるようになることである.

## 1.1 集合

集合に関する基本的な概念や性質については高校数学である程度学んでいることと思うので, ここでは本稿の理解に必要な用語・記法について簡単にまとめるにとどめる.

**集合**とは対象の集まりのことである. 典型的な集合には以下のようなものがある.

$$\begin{array}{rcll} \mathbb{B} & = & \{true, false\} & (\textbf{ブール値}\text{の集合}) \\ \mathbb{N} & = & \{0, 1, 2, \cdots\} & (\textbf{自然数}\text{の集合}) \\ \mathbb{R} & = & \{r \mid r \text{ は実数 }\} & (\textbf{実数}\text{の集合}) \end{array}$$

**0 も自然数とみなす**ことに注意. これはコンピュータサイエンスの慣習である. $true, false$ は単なる文字列である. 別に「真」と「偽」でも構わない. 2 つの異なる対象である限りなんでもよい.

「対象 $a$ は集合 $A$ の**要素**である」ことを

$$a \in A$$

と書く. 「$a$ は $A$ の元である」, 「$a$ は $A$ に属する」等の言い方もする. その否定は「$a$ は $A$ の要素でない」であり

$$a \notin A$$

と書く. たとえば $0 \in \mathbb{N},\ 0 \in \mathbb{R},\ 0 \notin \mathbb{B}$ である.

対象 $a_1, \ldots, a_k$ からなる集合を

$$\{a_1, \ldots, a_k\}$$

と書く. もちろん, $a_i \in \{a_1, \ldots, a_k\}$ が各 $1 \leq i \leq k$ について成り立つ.

要素を 1 つも含まない集合を**空集合**といい, $\emptyset$ と書く. どんな対象 $a$ についても $a \notin \emptyset$ である.

集合自体も対象と見なすことができる. それゆえ $\{\mathbb{B}, \mathbb{N}\}$ も集合である. このように「集合の集合」や「集合の集合の集合」等をいくらでも考えていくことができる.

**■集合の内包表記**　$\varphi(x)$ を自然数に関する性質とするとき, $\varphi(x)$ を満たす自然数全ての集合（$\varphi(x)$ の**外延**）を

$$\{x \in \mathbb{N} \mid \varphi(x)\}$$

と書く（「$\in \mathbb{N}$」が明らかなときには省略して $\{x \mid \varphi(x)\}$ と書いてしまうこともある）. たとえば

$$\{x \in \mathbb{N} \mid x \text{ は素数である }\}$$

は素数全体の集合を表す.

一般に, 集合 $A$ 上の性質 $\varphi(x)$ が与えらえたとき, $A$ の中で $\varphi(x)$ を満たす要素全体の集合

$$\{x \in A \mid \varphi(x)\}$$

が存在する. これを**包括原理**という. 当然ながら, どんな $a \in A$ についても

$$\varphi(a) \quad \Longleftrightarrow \quad a \in \{x \in A \mid \varphi(x)\} \tag{1}$$

が成り立つ. このことは具体例を挙げれば一目瞭然であろう.

$$n \text{ は素数である} \quad \Longleftrightarrow \quad n \in \{x \in \mathbb{N} \mid x \text{ は素数である }\}.$$

上の記法を用いるときには,「$x \in \mathbb{N}$」や「$x \in A$」のように対象の範囲を制限するのが重要である. もしも範囲を制限しなくてよいのであれば, 集合

$$R := \{x \mid x \notin x\} \tag{2}$$

を作ることができるだろう. これは「自分自身を要素として含まない集合」全体の集まりを表す. すると

$$R \in R \quad \overset{(2)}{\Longleftrightarrow} \quad R \in \{x \mid x \notin x\} \quad \overset{(1)}{\Longleftrightarrow} \quad R \notin R$$

が成り立ってしまう. これは矛盾である（**ラッセルのパラドックス**）.

**■集合の外延性**　全く同じ要素からなる 2 つの集合は等しい. すなわち

$$A = B \quad \Longleftrightarrow \quad \text{どんな } x \text{ についても } x \in A \text{ と } x \in B \text{ は同値である.} \tag{3}$$

これを**外延性の原理**という.

「集合 $A$ は集合 $B$ の**部分集合**である」ことを $A \subseteq B$ と書く（人によっては $A \subset B$ とも書く）. すなわち

$$A \subseteq B \quad \Longleftrightarrow \quad \text{どんな } x \text{ についても } x \in A \text{ ならば } x \in B \text{ である.} \tag{4}$$

(3), (4) より

$$A = B \quad \Longleftrightarrow \quad A \subseteq B \text{ かつ } B \subseteq A$$

が成り立つ.

**■基本的な演算**　外延性の原理により, 集合はその要素によって一意に定まる. それゆえ 1 つの集合 $A$ を定めるには, 任意の要素 $x$ について「$x \in A$」が成り立つための必要十分条件を与えればよい.

たとえば, 集合 $A$ と $B$ の**共通部分**, **和**, **差**はそれぞれ $A \cap B$, $A \cup B$, $A \setminus B$ と書かれるが, これらは次のように定めることができる.

$$\begin{aligned} x \in A \cap B &\iff x \in A \text{ かつ } x \in B \\ x \in A \cup B &\iff x \in A \text{ または } x \in B \\ x \in A \setminus B &\iff x \in A \text{ かつ } x \notin B. \end{aligned}$$

集合 $A$ の部分集合全体の集まりを**べき集合**といい, $\wp(A)$ と書く. つまり

$$X \in \wp(A) \quad \Longleftrightarrow \quad X \subseteq A.$$

たとえば $A = \{a, b, c\}$ のとき

$$\wp(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, A\}$$

である. 空集合 $\emptyset$ および $A$ 自体も $\wp(A)$ に属することに注意.

集合 $A$ と $B$ の**直積**を以下で定める.

$$A \times B \ = \ \{(a,b) \mid a \in A,\ b \in B\}.$$

ここで $(a,b)$ は $a$ と $b$ の組を表す. 同様にして, $A, B, C$ の直積は $A \times B \times C = \{(a,b,c) \mid a \in A,\ b \in B,\ c \in C\}$ である. 一般に $n$ 個の集合 $A_1, \ldots, A_n$ についてその直積 $A_1 \times \cdots \times A_n$ を考えることができる.

同じ集合を $n$ 回掛け合わせたもの, すなわち $\underbrace{A \times \cdots \times A}_{n}$ のことを $A^n$ と書く. たとえば

$$\begin{array}{rcl} A^0 & = & \{()\} \\ A^1 & = & \{(a) \mid a \in A\} \\ A^2 & = & A \times A \ = \ \{(a,b) \mid a,b \in A\} \\ A^3 & = & A \times A \times A \ = \ \{(a,b,c) \mid a,b,c \in A\} \end{array}$$

等々である. ここで () は**空列**（0 個の要素の組）を表す. たとえば実直線上の点は $\mathbb{R}$ の要素に対応し, 実平面の点は $\mathbb{R}^2$ の要素, 実空間の点は $\mathbb{R}^3$ の要素に対応する.

■**性質・関係**　すでに述べたように「素数である」という性質は集合 $\{x \in \mathbb{N} \mid x \text{ は素数である }\}$ と同一視できる. これは $\mathbb{N}$ の部分集合である. 一般に, 集合 $A$ の部分集合 $P \subseteq A$ のことを $A$ 上の**性質**という.

同様に「$n$ は $m$ の約数である」という関係は集合 $\{(n,m) \in \mathbb{N}^2 \mid n \text{ は } m \text{ の約数である }\}$ で表すことができる. これは $\mathbb{N}^2$ の部分集合である. 一般に集合 $A, B$ が与えられたとき, 部分集合 $R \subseteq A \times B$ のことを, $A, B$ 上の **2 項関係**（あるいは単に**関係**）という.

これはさらに 3 項関係, 4 項関係等へと一般化できる. 1 項関係とは性質のことに他ならない.

## 1.2　関数

集合 $A, B$ が与えらえれたとき, $A$ から $B$ への**関数**（あるいは**写像**）とは, 各 $a \in A$ に何らかの $b \in B$ を割り当てる対応のことである. $f$ が $A$ から $B$ への関数のとき,

$$f : A \to B$$

と書く. また, $A$ のことを $f$ の**定義域**といい, $B$ のことを**値域**という. たとえば $m, n \in \mathbb{N}$ について $h_1(m) := m^2$ とおき, $h_2(m,n) := m + n$ とすると,

$$h_1 : \mathbb{N} \to \mathbb{N}, \qquad h_2 : \mathbb{N}^2 \to \mathbb{N}$$

となる. 前者を **1 項関数**, 後者を **2 項関数**という. 一般に

$$f : A_1 \times \cdots \times A_n \to B$$

の形の関数を $n$ **項関数**という.

$A$ から $B$ への関数全体の集合を $A \to B$ または $B^A$ と書く. つまり

$$f : A \to B, \qquad f \in A \to B, \qquad f \in B^A$$

は全部同じことを表す.

**■単射・全射・全単射**　関数 $f : A \to B$ については次の定義が重要である.

$f$ は**単射**である　$\iff$　どんな $a_1, a_2 \in A$ についても, $f(a_1) = f(a_2)$ ならば $a_1 = a_2$.
$f$ は**全射**である　$\iff$　どんな $b \in B$ についても, $f(a) = b$ を満たす $a \in A$ が存在する.

単射かつ全射な関数を**全単射**という. $f$ が単射であるとは, 対偶をとれば $a_1 \neq a_2 \Longrightarrow f(a_1) \neq f(a_2)$ ということなので, 「異なる 2 点を異なる 2 点に写す」のが単射であるといってもよい.

たとえば実数上の関数 $f : \mathbb{R} \to \mathbb{R}$ について考えると,

- 1 次関数 $f(x) = 2x + 3$ は全単射である.
- 2 次関数 $f(x) = x^2$ は単射でも全射でもない（$f(1) = f(-1) = 1$, $f(x) = -1$ を満たす $x$ は存在しない）.
- 3 次関数 $f(x) = x^3 + x^2$ は単射でないが全射である（$f(0) = f(-1) = 0$, $f(x)$ はどんな実数値もとりうる）.
- 指数関数 $f(x) = e^x$ は単射だが全射ではない（$x \neq y$ ならば $e^x \neq e^y$, $e^x = -1$ を満たす $x \in \mathbb{R}$ は存在しない）.

**補足 1.1** 一般に連続関数 $f : \mathbb{R} \to \mathbb{R}$ について以下のことが成り立つ（どちらも中間値の定理による）.
- $f$ が単射であるための必要十分条件は, $f$ が狭義単調増加または狭義単調減少なことである.
- $f$ が全射であるための必要十分条件は, $f$ が上下に非有界なことである.

**■単射・全射・全単射の基本的な性質**　関数 $f : A \to B$ が与えられたとき, その**像**を

$$f[A] := \{f(a) \mid a \in A\} \subseteq B$$

により定める. 関数 $f : A \to B$ の値域は, $f[A]$ に狭めることができる. このとき $f : A \to f[A]$ は全射になる. とくに元の $f$ が単射ならば, 新たな $f : A \to f[A]$ は全単射である.

全単射 $f : A \to B$ が与えられたとき,

$$f^{-1}(b) := a \quad \iff \quad f(a) = b$$

により逆向きの全単射 $f^{-1} : B \to A$ を定めることができる. これを**逆関数**という.

単射と全射の関係は次のようになっている.

**命題 1.2**

$A$ を空でない集合, $B$ を任意の集合とすると

$A$ から $B$ への単射が存在する　$\iff$　$B$ から $A$ への全射が存在する.

**証明：** 単射 $f : A \to B$ が与えられたとき, 1 点 $a_0 \in A$ を勝手に選んで関数 $g : B \to A$ を

$$\begin{array}{rcll} g(b) & := & a & (f(a) = b \text{ を満たす } a \in A \text{ が存在するとき}) \\ & := & a_0 & (\text{それ以外のとき}) \end{array}$$

と定める. $f$ は単射なので, $f(a) = b$ を満たす $a$ が存在すればただ 1 つに定まる. よって上の定義はうまくいっている. 任意の $a \in A$ について $b := f(a)$ とおけば $g(b) = a$ なので, $g$ は全射である.

逆に全射 $g : B \to A$ が与えられたとき, 各 $a \in A$ について集合

$$g^{-1}(a) := \{b \in B \mid g(b) = a\}$$

は空でない. よって 1 点 $b \in g^{-1}(a)$ を勝手に選んで $f(a) := b$ とすれば, 関数 $f : A \to B$ が得られる. $a_1 \neq a_2$ なら $g^{-1}(a_1)$ と $g^{-1}(a_2)$ は交わらず, したがって $f(a_1) \neq f(a_2)$ なので, $f$ は単射である. □

**■関数の合成** 関数は組み合わせることができる. 関数 $f : A \to B$, $g : B \to C$ が与えられらたとき, $f$ と $g$ の**合成** $g \circ f : A \to C$ を

$$g \circ f(x) \ := \ g(f(x))$$

と定める. たとえば $f(x) = 2x + 3$, $g(x) = x^2$ ならば $g \circ f(x) = (2x + 3)^2$ である.

特別な関数として, 各集合 $A$ 上の**恒等関数** $id_A : A \to A$ が挙げられる. これは $id_A(x) := x$ により定められる全単射であり, 要するに「何もしない関数」である.

**命題 1.3**

$f : A \to B$, $g : B \to C$ とする.

1. $f, g$ が単射ならば $g \circ f$ も単射である.
2. $f, g$ が全射ならば $g \circ f$ も全射である.
3. $f, g$ が全単射ならば $g \circ f$ も全単射である.

証明は練習問題とする.

## 1.3 集合の比較

有限集合 $A, B$ が与えられたとき, 両者の大きさを比較するには要素の個数を数えればよい. たとえば $A = \{1, 2, 3\}$, $B = \{0.1, \sqrt{2}, \pi\}$ ならば, 両方とも 3 つの要素からなるので「同数」であるといえる. しかし $A, B$ が無限集合のときには（自然数だけでは）個数を数えることができない. そこで一般の集合を比較をするときには, 全単射の存在をもって「同数」であると定める.

まずは基礎となる定理を示す. やや込み入った議論なので, わからなければ証明は飛ばしてもよい.

**補題 1.4**

$A \subseteq B$ で単射 $g : B \to A$ があれば, 全単射 $h : B \to A$ が存在する.

**証明：** $C := \{g^n(x) \mid n \in \mathbb{N},\ x \in B \setminus A\}$ とおく．ただし $g^n(x) := g(g(\cdots g(x)\cdots))$（$x$ に $g$ を $n$ 回適用した結果）とする．とくに $g^0(x) = x$ である．集合 $C$ は $g$ の適用について閉じている．つまり，

$$b \in C \quad \Longrightarrow \quad g(b) \in C. \tag{5}$$

また, 任意の $b \in B$ について

$$b \notin C \quad \Longrightarrow \quad b \in A \tag{6}$$

も成り立つ．実際, もし $b \notin A$ ならば $b \in B \setminus A$, したがって $b = g^0(b) \in C$ となるが, これは仮定 $b \notin C$ に反する．

$B$ を定義域とする関数 $h$ を以下で定める．

$$\begin{array}{rcll} h(b) & := & g(b) & (b \in C \text{ のとき}) \\ & := & b & (b \notin C \text{ のとき}) \end{array}$$

$g$ の値域が $A$ であることと (6) より $h : B \to A$ である．あとはこの $h$ が全単射であることを示せばよい．

- 単射性：$B$ から相異なる要素 $b_1, b_2$ をとる．$b_1, b_2 \notin C$ の場合も $b_1, b_2 \in C$ の場合も $h(b_1) \neq h(b_2)$ である（$g$ の単射性より）．$b_1 \in C$, $b_2 \notin C$ のときは (5) より $h(b_1) = g(b_1) \in C$ であり, 一方で $h(b_2) = b_2 \notin C$ である．よって $h(b_1) \neq h(b_2)$ が成り立つ．
- 全射性：$a \in A$ とする．$a \notin C$ なら $h(a) = a$ である．$a \in C$ なら, $C$ の定義よりある $n \in \mathbb{N}$ と $b \in B \setminus A$ について $a = g^n(b)$ と書けるはずである．$b \notin A$ なので $a \neq b$, したがって $n > 0$ である．$c := g^{n-1}(b)$ とおけば $c \in C$ である．よって $h(c) = g(c) = g^n(b) = a$ が成り立つ．

□

**定理 1.5(カントール・ベルンシュタインの定理)**

単射 $f : A \to B$ と単射 $g : B \to A$ があれば, 全単射 $h : A \to B$ が存在する．

**証明：** $f$ の値域を像 $f[A]$ に制限することにより, $f$ は全単射 $f : A \to f[A]$ であるとしてよい．$f[A] \subseteq B$ であり, なおかつ命題 1.3 より $f \circ g : B \to f[A]$ は単射である．よって補題 1.4 より全単射 $h : B \to f[A]$ が存在する．$h$ の逆関数 $h^{-1} : f[A] \to B$ も全単射なので, 求める全単射 $h^{-1} \circ f : A \to B$ が得られる[*1]． □

**系 1.6**

単射 $f : A \to B$ と全射 $g : A \to B$ があれば, 全単射 $h : A \to B$ が存在する．

**証明：** $B = \emptyset$ のときは $A = \emptyset$ なのでよい．$B \neq \emptyset$ のときは命題 1.2 より単射 $g' : B \to A$ が得られるので定理が適用できる． □

[*1] 以上の証明は M. Hils and F. Loeser. *A First Journey through Logic.* AMS, 2019 より採録した．

■**集合の比較**　集合 $A, B$ の間に全単射 $f : A \to B$ があるとき, $A$ と $B$ は**同数**である（あるいは濃度が等しい）といい, 本稿では $A \simeq B$ と書く. また単射 $f : A \to B$ があるとき, $A$ の数は $B$ **以下**であるといい, $A \preceq B$ と書く. 最後に $A \preceq B$ かつ $A \not\simeq B$ のときは $A \prec B$ と書く.

次の命題により, 数の大小を比較するのと同じように集合の大小も比較できることがわかる.

**命題 1.7**

集合 $A, B, C$ について

1. $A \preceq A$.
2. $A \preceq B \preceq C$ ならば $A \preceq C$.
3. $A \preceq B$ かつ $B \preceq A$ ならば $A \simeq B$.

**証明：**　1 は恒等関数 $id_A : A \to A$ が全単射であることより, 2, 3 はそれぞれ命題 1.3, 定理 1.5 からしたがう. □

**補足 1.8** さらに追加すれば, 任意の集合 $A, B$ について $A \preceq B$ か $B \preceq A$ の少なくともどちらか一方が成り立つ. 以下証明の概略のみ述べておく（読み飛ばしてよい）.

2 項関係 $R \subseteq A \times B$ が**一対一**であるとは,

$$\begin{array}{lcl} (a, b_1), (a, b_2) \in R & \Longrightarrow & b_1 = b_2 \\ (a_1, b), (a_2, b) \in R & \Longrightarrow & a_1 = a_2 \end{array}$$

が成り立つことをいう. 集合 $\{R \subseteq A \times B \mid R \text{ は一対一関係 }\}$ は空でなく帰納的なので, ツォルンの補題により極大な一対一関係 $S \subseteq A \times B$ が存在する.

$$\begin{array}{lcll} \mathsf{dom}(S) & := & \{a \in A \mid (a, b) \in S \text{ を満たす } b \in B \text{ が存在する }\} & \subseteq A \\ \mathsf{ran}(S) & := & \{b \in B \mid (a, b) \in S \text{ を満たす } a \in A \text{ が存在する }\} & \subseteq B \end{array}$$

と定めて, 場合分けする.

- $\mathsf{dom}(S) = A$ のとき,
$$f(a) := b \quad \Longleftrightarrow \quad (a, b) \in S$$
により定められる $f : A \to B$ は単射である. よって $A \preceq B$.
- $\mathsf{ran}(S) = B$ のとき, 上の場合と対称的に単射 $g : B \to A$ が存在する. よって $B \preceq A$.
- $\mathsf{dom}(S) \neq A, \mathsf{ran}(S) \neq B$ が同時に成り立つことは, $S$ の極大性より不可能.

## 1.4　特性関数とカリー化

コンピュータ科学や数理論理学で特に重要な役割を果たす同数性を 2 つ挙げておく.

集合 $A$ とその部分集合 $P \in \wp(A)$ が与えられたとき, 関数 $\chi_P : A \to \mathbb{B}$ を次のように定める.

$$\begin{array}{lcll} \chi_P(a) & := & \mathit{true} & (a \in P \text{ のとき}) \\ & := & \mathit{false} & (a \notin P \text{ のとき}) \end{array}$$

これを $P$ の**特性関数**という. たとえば $P \subseteq \mathbb{N}$ が素数全体の集合なら, $\chi_P$ は入力 $n \in \mathbb{N}$ に対し $n$ が素数かどうかを判定する関数となる.

**命題 1.9**

$A$ を集合とする. 部分集合 $P \in \wp(A)$ に対し特性関数 $\chi_P \in A \to \mathbb{B}$ を割り当てる対応 $\chi$ は全単射である. よって同数性

$$\wp(A) \ \simeq \ A \to \mathbb{B}$$

が成り立つ.

**証明：** 関数 $f : A \to \mathbb{B}$ が与えられたとき, $P := \{x \in A \mid f(x) = \mathit{true}\}$ と定めれば

$$\chi_P(a) = \mathit{true} \quad \Longleftrightarrow \quad a \in P \quad \Longleftrightarrow \quad f(a) = \mathit{true}$$

なので $\chi_P = f$ である. よって $\chi$ は全射である. また $P, Q \in \wp(A)$ について $\chi_P = \chi_Q$ ならば

$$a \in P \quad \Longleftrightarrow \quad \chi_P(a) = \mathit{true} \quad \Longleftrightarrow \quad \chi_Q(a) = \mathit{true} \quad \Longleftrightarrow \quad a \in Q$$

なので $P = Q$ である. よって $\chi$ は単射である. □

上の命題により, 性質や関係に関する事柄を関数の話に帰着させることができる. たとえば自然数の性質 $P \subseteq \mathbb{N}$ が「計算可能」であることを $\chi_P : \mathbb{N} \to \mathbb{B}$ が「計算可能」であることにより定める, といった具合である.

次に 2 項関数 $f : A \times B \to C$ を考える. これは次のようにすれば 1 項関数に直して考えることができる. 要素 $a \in A$ が与えられたとき, 関数 $f_a : B \to C$ を

$$f_a(x) \ := \ f(a, x)$$

と定める. このようにして $a \in A$ に対して $f_a \in B \to C$ を割り当てる対応を $f$ の**カリー化**といい $\mathit{curry}(f) : A \to (B \to C)$ と表す.

**命題 1.10**

$A, B, C$ を集合とする. 関数 $f : A \times B \to C$ に対して $\mathit{curry}(f) : A \to (B \to C)$ を割り当てる対応 $\mathit{curry}$ は全単射である. よって同数性

$$A \times B \to C \ \simeq \ A \to (B \to C)$$

が成り立つ.

**証明：** 関数 $g \in A \to (B \to C)$ が与えられたとき, 任意の $a \in A$ について $g(a) \in B \to C$ である. これを $g_a : B \to C$ と書くことにする. 関数 $f : A \times B \to C$ を $f(a, b) := g_a(b)$ により定めれば, $\mathit{curry}(f) = g$ となる. 実際, $f_a(x) = f(a, x) = g_a(x)$ なので任意の $a \in A$ について $f_a = g_a$, したがって $\mathit{curry}(f)$ と $g$ は同じ関数を表す. よって $\mathit{curry}$ は全射である.

次に, 2 本の関数 $f, g : A \times B \to C$ について $\mathit{curry}(f) = \mathit{curry}(g)$ とする. すなわち任意の $a \in A$ について $f_a = g_a$. すると $f(a, b) = f_a(b) = g_a(b) = g(a, b)$ であり, したがって $f = g$ である. よって $\mathit{curry}$ は単射である. □

カリー化を繰り返せば, 直積 $\times$ を用いなくても $\to$ だけで多項関数を表すことができる.

$$A_1 \times \cdots \times A_n \to B \ \simeq \ A_1 \to (A_2 \to \cdots (A_n \to B) \cdots).$$

次章以降ではこのトリックをひんぱんに用いる.

## 1.5 可算無限集合

$\mathbb{N}$ と同数の集合 $A$ のことを**可算無限集合**という. これはつまり全単射 $f : \mathbb{N} \to A$ が存在するということなので, $f(0) = a_0,\ f(1) = a_1,\ f(2) = a_2,\ \ldots$ とおけば, $A$ の要素は

$$a_0,\ a_1,\ a_2,\ a_3,\ \ldots$$

というように余すことなく列挙できる. "算え上げられる" から可算というのである. 有限集合と可算無限集合を合わせて**高々可算**な集合という.

たとえば $E := \{n \in \mathbb{N} \mid n \text{ は偶数 }\}$ は可算無限集合である. なぜなら関数 $f(x) := 2x$ が $\mathbb{N}$ から $E$ への全単射を与えるからである. 同様に, 素数全体の集合も可算無限である. なぜなら, 自然数 $n$ に対して $n$ 番目の素数を対応させることができるからである. 一般に, $\mathbb{N}$ のどんな無限部分集合も可算無限である.

実際, $\mathbb{N}$ は無限集合の中で最小である.

**補題 1.11**

任意の無限集合 $A$ について $\mathbb{N} \preceq A$ である. したがってもし単射 $f : A \to \mathbb{N}$ が存在すれば $A$ は可算無限である.

**証明：** 概略のみ述べる. $A$ の中から 1 つ要素 $a_0$ を選び $A_1 := A \setminus \{a_0\}$ とする. これも無限集合であるから, 1 つ要素 $a_1 \in A_1$ を選び, 無限集合 $A_2 := A_1 \setminus \{a_1\}$ をつくることができる. このプロセスを続ければ, 相異なる要素の列 $a_0, a_1, a_2, \ldots$ が得られるので, $f(n) := a_n$ とすれば, 単射 $f : \mathbb{N} \to A$ が得られる. よって $\mathbb{N} \preceq A$. 後半の主張は命題 1.7 の帰結である. □

さらに例を挙げていこう.

**命題 1.12**

以下の集合は可算無限である.

1. $\mathbb{N}$, $\mathbb{N}^2$, $\mathbb{N}^3$, $\ldots$
2. 自然数の有限列全体の集合 $\mathbb{N}^* := \mathbb{N}^0 \cup \mathbb{N}^1 \cup \mathbb{N}^2 \cup \mathbb{N}^3 \cup \cdots$
3. 整数の集合 $\mathbb{Z}$
4. 有理数の集合 $\mathbb{Q}$

**証明：**

1: たとえば $\mathbb{N}^3$ について考えよう. 関数 $f_3 : \mathbb{N}^3 \to \mathbb{N}$ を $f_3(a, b, c) := 2^a 3^b 5^c$ と定めれば, これは単射であ

る．なぜなら素因数分解の一意性より，$f_3(a,b,c) = f_3(a',b',c')$ ならば $a=a', b=b', c=c'$ だからである．よって補題 1.11 より $\mathbb{N}^3$ は可算無限．同様にして，任意の $m \geq 1$ について $\mathbb{N}^m$ は可算無限なことが示せる．

2: 上記により，任意の $m \in \mathbb{N}$ について単射 $f_m : \mathbb{N}^m \to \mathbb{N}$ が存在する．そこで $f : \mathbb{N}^* \to \mathbb{N}^2$ を $f(x) := (m, f_m(x))$ と定めれば単射が得られる（$m$ は列 $x$ の長さ）．実際，$x, y \in \mathbb{N}^*$ が与えられたとき，$f(x) = f(y)$ なら $x, y$ の長さは等しい．その長さを $m$ とすると，さらに $f_m(x) = f_m(y)$ が成り立つので，$f_m$ の単射性より $x = y$ がいえる．

こうして得られた $f : \mathbb{N}^* \to \mathbb{N}^2$ を $f_2 : \mathbb{N}^2 \to \mathbb{N}$ と合成すれば，単射 $f_2 \circ f : \mathbb{N}^* \to \mathbb{N}$ が得られる．よって補題 1.11 より $\mathbb{N}^*$ は可算無限である．

3: 関数 $f : \mathbb{Z} \to \mathbb{N}^2$ を

$$\begin{array}{rcll} f(x) & := & (0,x) & (x \geq 0 \text{ のとき}) \\ & := & (1,-x) & (x < 0 \text{ のとき}) \end{array}$$

と定めれば $f$ は単射である．あとは上と同様．

4: 有理数 $a \in \mathbb{Q}$ は $p \in \mathbb{Z}$ と $q \in \mathbb{N}$ を用いて $a = p/q$ と一意に表せることに注意（ただし $q > 0$, $p$ と $q$ は互いに素）．よって単射 $f : \mathbb{Q} \to \mathbb{Z} \times \mathbb{N}$ が自然に定義できる．すでに $\mathbb{Z} \simeq \mathbb{N}$ を示してあるので，あとは上と同様に議論すればよい． □

**補足 1.13** 以下により定められる関数 $f : \mathbb{N}^2 \to \mathbb{N}$ は全単射である．

$$f(m,n) := \frac{(m+n)(m+n+1)}{2} + m.$$

このことから $\mathbb{N}^2$ が可算無限であることを（補題 1.11 によらずとも）直接示すことができる．

**■対角線論法** 一方で可算でない無限集合（**非可算無限集合**）も存在する．これは次の一般定理の帰結である．

**定理 1.14**

任意の集合 $A$ について $A \prec \wp(A)$ が成り立つ．とくに $\mathbb{N} \prec \wp(\mathbb{N})$ であり，$\wp(\mathbb{N})$ は非可算無限集合である．

**証明：** 関数 $f : A \to \wp(A)$ を $f(x) := \{x\}$ と定めれば，これは単射である．よって $A \preceq \wp(A)$．

仮に $A \simeq \wp(A)$ とすると，全射 $g : A \to \wp(A)$ が存在するはずである．そこで

$$D := \{x \in A \mid x \notin g(x)\} \in \wp(A) \tag{7}$$

とおくと，$g$ の全射性よりある $d \in A$ が存在して $f(d) = D$ となる．すると

$$d \in D \overset{(7)}{\iff} d \in \{x \in A \mid x \notin g(x)\} \overset{(1)}{\iff} d \notin g(d) \iff d \notin D$$

となり矛盾が生じる．よって $A \not\simeq \wp(A)$． □

上の論法をカントールの**対角線論法**という．この定理により，

$$\mathbb{N} \prec \wp(\mathbb{N}) \prec \wp(\wp(\mathbb{N})) \prec \wp(\wp(\wp(\mathbb{N}))) \prec \cdots$$

という風にして, いくらでも大きな無限集合を作り出していくことができる.

**命題 1.15**

以下の集合は $\wp(\mathbb{N})$ と同数であり, したがって非可算無限である.

1. $\mathbb{N} \to \mathbb{B}$
2. $\mathbb{N} \to \mathbb{N}$
3. 自然数の無限列の集合 $\mathbb{N}^\omega$
4. 実数の集合 $\mathbb{R}$

**証明：**
1: 命題 1.9 より.
2: (i) 関数 $f : \mathbb{N} \to \mathbb{B}$ が与えられたとき, 出力の $true, false$ をそれぞれ $0, 1$ と同一視すれば, 関数 $f' : \mathbb{N} \to \mathbb{N}$ が得られる. 当然この対応は単射である. 一方で (ii) 関数 $f : \mathbb{N} \to \mathbb{N}$ に対してその**グラフ**

$$\{(m, n) \in \mathbb{N}^2 \mid f(m) = n\} \ \in \ \wp(\mathbb{N}^2)$$

を割り当てる対応は $\mathbb{N} \to \mathbb{N}$ から $\wp(\mathbb{N}^2)$ への単射である. 以上より

$$(\mathbb{N} \to \mathbb{B}) \overset{\text{(i)}}{\preceq} (\mathbb{N} \to \mathbb{N}) \overset{\text{(ii)}}{\preceq} \wp(\mathbb{N}^2) \overset{\text{命題 1.12}}{\simeq} \wp(\mathbb{N}) \overset{1}{\simeq} (\mathbb{N} \to \mathbb{B})$$

となり, 命題 1.7 より $(\mathbb{N} \to \mathbb{N}) \simeq \wp(\mathbb{N})$ が帰結する.

3: 関数 $f : \mathbb{N} \to \mathbb{N}$ に対して無限列 $(f(0), f(1), f(2), f(3), \cdots) \in \mathbb{N}^\omega$ を割り当てる対応は全単射である.

4: (i) どんな実数 $r \in \mathbb{R}$ も, それより大きい有理数たちの下限として一意に表せる.

$$r \ = \ \inf\{q \in \mathbb{Q} \mid r < q\}.$$

よって $r$ に $\{q \in \mathbb{Q} \mid r < q\} \in \wp(\mathbb{Q})$ を割り当てる対応は単射である. 一方で (ii) 自然数の集合 $P \in \wp(\mathbb{N})$ に対して次の性質を満たす実数 $r_P \in \mathbb{R}$ を一意に割り当てることができる.

> $r_P$ を（10 進法で）無限小数展開すると $0.a_0a_1a_2a_3\cdots$ の形であり, $n \in P$ ならば $a_n = 1$, $n \notin P$ ならば $a_n = 0$ である.

たとえば $P$ が素数全体の集合なら $r_P = 0.00110101000101\cdots$ である. 明らかにこの割り当ては $\wp(\mathbb{N})$ から $\mathbb{R}$ への単射になっている. 以上より

$$\mathbb{R} \overset{\text{(i)}}{\preceq} \wp(\mathbb{Q}) \overset{\text{命題 1.12}}{\simeq} \wp(\mathbb{N}) \overset{\text{(ii)}}{\preceq} \mathbb{R}.$$

よって $\mathbb{R} \simeq \wp(\mathbb{N})$ が帰結する. □

**補足 1.16** 数の集合について考えてみると, $\mathbb{Z}$ や $\mathbb{Q}$ は $\mathbb{N}$ と同数であり, $\mathbb{R}$ や無理数の集合 $\mathbb{R} \setminus \mathbb{Q}$ はそれらより真に大きい. ではその中間の大きさを持つ集合は存在するだろうか? つまり

$$\mathbb{N} \ \prec \ X \ \prec \ \mathbb{R}$$

を満たす集合 $X$ は存在するだろうか? 「存在しない」というのが**連続体仮説**である. 集合論の発展を促してきた仮説であるが, その成否は標準とされる集合論の公理系 $ZFC$ では決定できないことが知られている. 要は「どちらでもありうる」のである.

■「計算できない」関数の存在　コンピュータ科学で決定的に重要なのは次の事実である．今, 文字や記号を自然数で表すことにする．たとえば:

| $a$ | $b$ | $c$ | $\cdots$ | ( | ) | + | * | $\cdots$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | $\cdots$ | 27 | 28 | 29 | 30 | $\cdots$ |

すると, どんなプログラミング言語を考えても, そのプログラムは有限長の文字列に過ぎないから $\mathbb{N}^*$ の要素と同一視することができる．さて, 関数 $f : \mathbb{N} \to \mathbb{N}$ が「計算可能」ならば, それを計算するプログラム $\mathsf{f} \in \mathbb{N}^*$ が存在するはずである．しかも $f \neq g$ ならば $\mathsf{f} \neq \mathsf{g}$ である．それゆえ「計算可能」な関数は, 高々 $\mathbb{N}^*$ と同数しか存在しえない．つまり高々可算個である．一方で $\mathbb{N} \to \mathbb{N}$ は非可算無限集合であり, $\mathbb{N}^*$ より真に大きい．よって

**「計算可能」ではない関数 $f : \mathbb{N} \to \mathbb{N}$ が存在する**

ことになる．そうすると気になるのは, 具体的にいってどんな関数が計算できて, どんな関数が計算できないのかである．この問いがコンピュータ科学の出発点であったといってよい．

**練習問題**

1. 自然数上の関数 $f : \mathbb{N} \to \mathbb{N}$ で以下を満たすものの具体例を挙げよ．
   (i) $f$ は全単射である．
   (ii) $f$ は単射だが全射でない．
   (iii) $f$ は全射だが単射でない．
   (iv) $f$ は単射でも全射もない．
2. 補足 1.1 の主張を証明せよ．
3. 命題 1.3 を証明せよ．
4. 次のことを証明せよ．関数 $f : A \to B$ が全単射であるための必要十分条件は, ある $g : B \to A$ が存在して $g \circ f = id_A$ かつ $f \circ g = id_B$ が成り立つことである．
5. $A$ と $B$ が可算無限集合ならば $A \cup B$ も可算無限集合であることを証明せよ．
6. $\wp_{fin}(\mathbb{N}) := \{P \mid P \text{ は } \mathbb{N} \text{ の有限部分集合 }\}$ が可算無限集合であることを証明せよ．
7. 無理数の集合 $\mathbb{R} \setminus \mathbb{Q}$ が非可算無限集合であることを証明せよ．
8. $\mathbb{R} \prec (\mathbb{R} \to \mathbb{R})$ が成り立つを証明せよ．

# 2　原始再帰的関数

すでに述べたように, 計算可能な関数とはプログラムにより計算できる関数のことである. それゆえ, 計算可能な関数についてきちんと議論するには, 何らかのプログラミング言語を介する必要がある.

本章では, 簡単な一階関数型プログラミング言語を導入する. この言語で表現できるプログラムのことを**原始再帰的プログラム**といい, それにより計算できる関数のことを**原始再帰的関数**という. 強い制約のある言語なので, すべての計算可能な関数が表現できるわけではない. そのかわり, **停止性**という著しい性質を持つ. すなわち, この言語で書かれたプログラムを実行すると必ず有限ステップで停止する. しかも計算時間（ステップ数）をプログラム実行前にあらかじめ見積もることができる.

原始再帰的プログラミングにおいて鍵となるのは, **帰納的データ型**の概念である. どんなデータ構造が扱えるかはプログラミング言語ごとに異なるのだが, 帰納的なデータに制限し, その構造に沿ってプログラミングすることにより, 上記の停止性が達成できるというのがアイデアである.

本章の目標は, (i) 帰納的データ型と一階関数型について理解し, (ii) 原始再帰的関数の定義と性質を学び, (iii) その限界を知ることである.

## 2.1　式と計算規則

まずは, プログラムやデータが型を持たない無秩序な状況を考える. それにより型の重要性を際立たせるのが目的である.

プログラムを記述するのに以下の記号を用いる.

- **構成子**：true, false, 0, s.
- **識別子**：f, g, add, even?, ... 等々.
- **変数**：$x$, $y$, $z$, ... 等々.
- **カッコ**：(, ).

構成子は, ブール値や自然数などのデータを表すのに用いる. なお s は「+1」を表す記号である. 一方で識別子は, （部分）プログラムの名前を表すのに用いる. 単なる名前にすぎないので, 他の記号と重ならない限り好きな記号を用いてよい. 最後に変数は, 別の記号表現を代替するのに用いる. つまり変数には他の表現を代入できる.

■**式**　以下のように定義される記号表現を**式**（または**項**）という.

(i) 構成子, 識別子, 変数は式である.
(ii) $M$ と $N$ が式ならば, $(M\ N)$ も式である.
(iii) 以上により構成される記号表現のみが式である.

たとえば, 0 は式であり s も式である（(i) より）. よって (s 0) は式である（(ii) より）. 同様に (s $x$) も式である. よって (f (s 0)) も式であり,

$$((\mathsf{f}\ (\mathsf{s}\ 0))\ (\mathsf{s}\ x)) \tag{8}$$

も式である. 一方で () は式ではないし, f$(x, y)$ も式ではない（(iii) より）.

カッコが多くなると読みにくくなるので, 適宜省略表現を用いる. たとえば $((M_1\ M_2)\ M_3)$ は $(M_1\ M_2\ M_3)$ と省略してよい. また一番外側のカッコは省略してよい. よって (8) は

$$\mathsf{f}\ (\mathsf{s}\ 0)\ (\mathsf{s}\ x)$$

と書ける.

0 と s さえあればどんな自然数も記述することができる（s は「+1」を表すため）.

$$0,\quad \mathsf{s}\ 0,\quad \mathsf{s}\ (\mathsf{s}\ 0),\quad \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ 0)),\quad \ldots$$

これはつまり十進法ではなく, 一進法を用いて自然数を表記するということである. とはいえ数が大きくなると見にくくなるので,

$$0,\quad 1,\quad 2,\quad 3,\quad \ldots$$

といった省略表現も用いる.

**■計算規則**　さて, 次のような式の変形規則を考えてみよう.

$$\begin{array}{lcl}
\mathsf{add}\ 0\ x & = & x \\
\mathsf{add}\ (\mathsf{s}\ y)\ x & = & \mathsf{s}\ (\mathsf{add}\ y\ x) \\
\mathsf{even?}\ 0 & = & \mathsf{true} \\
\mathsf{even?}\ (\mathsf{s}\ x) & = & \mathsf{odd?}\ x \\
\mathsf{odd?}\ 0 & = & \mathsf{false} \\
\mathsf{odd?}\ (\mathsf{s}\ x) & = & \mathsf{even?}\ x
\end{array}$$

これらはいずれも「等号の左辺を右辺で置き換えてよい」ことを表す. その際, 変数には適当な代入を行ってよい. これらの規則を用いれば, 次のような計算ができる.

$$\begin{array}{lcl}
\mathsf{add}\ 2\ 1 & = & \mathsf{add}\ (\mathsf{s}\ (\mathsf{s}\ 0))\ (\mathsf{s}\ 0) \\
& = & \mathsf{s}\ (\mathsf{add}\ (\mathsf{s}\ 0)\ (\mathsf{s}\ 0)) \\
& = & \mathsf{s}\ (\mathsf{s}\ (\mathsf{add}\ 0\ (\mathsf{s}\ 0))) \\
& = & \mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ 0)) \quad = \quad 3.
\end{array}$$

$$\begin{array}{lcl}
\mathsf{even?}\ (\mathsf{add}\ 2\ 1) & = & \cdots\ =\ \mathsf{even?}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ 0))) \\
& = & \mathsf{odd?}\ (\mathsf{s}\ (\mathsf{s}\ 0)) \\
& = & \mathsf{even?}\ (\mathsf{s}\ 0) \\
& = & \mathsf{odd?}\ 0 \\
& = & \mathsf{false}.
\end{array}$$

add や even?, odd? 等の識別子は単なる記号にすぎないが, 適切な計算規則を与えることにより足し算を表したり, 偶奇判定を表したりすることができる. このようにすればさまざまな関数が計算できる.

しかし問題もある．まず，不用意な計算規則を与えると，計算の途中で (s true) や (add even? false) のように意味不明な式が発生してしまう恐れがある．これを**型エラー**という．プログラミング経験のある人ならわかるように，型エラーは最もひんぱんに生じるエラーのひとつである．

また，計算が有限ステップで停止することが保証されていない．たとえば

$$\mathsf{diverge}\ x \ = \ \mathsf{diverge}\ (\mathsf{s}\ x)$$

のような計算規則を考えると

$$\mathsf{diverge}\ 0 \ = \ \mathsf{diverge}\ 1 \ = \ \mathsf{diverge}\ 2 \ = \ \cdots$$

となり，いつまでたっても計算は停止しない．このような無限ループ・暴走も回避しなければならない．

**補足 2.1** 上のように式の変形によって計算を行うシステムを**項書き換え系**という．四則演算でおなじみの方法であるが，それ以外にも微積分や微分方程式の解法なども（適切な構成子・識別子を導入し，ものすごく複雑な計算規則を与えることにより）表現できる．実際，どんな計算可能な関数であっても，適切な項書き換え系によって表現できることが知られている．

## 2.2 型

上の問題を回避するひとつの方法は，式に**型**（**タイプ**）を付与することである．一般に，式 $M$ が型 $A$ を持つことを

$$M : A$$

と書く．型 $A$ により $M$ がどんな対象を表すのか——それは自然数なのか? 関数なのか? 関数だとしたらどんな定義域と値域を持つのか? ——を表すのである．プログラムを書くときに型を付与することで，多くの型エラーを実行前に検出することができる．

型の中で最も基本的なのは**データ型**である．これは入力値，出力値などの型を表すものであり，たとえば整数型，文字列型，不動点小数型，リスト型等が挙げられる．本講義では，当面**自然数型 N** と**ブール型 B** のみを取り扱う．これらは**帰納的データ型**と呼ばれるものの典型例である．それぞれ集合 $\mathbb{N}$ と $\mathbb{B}$ に対応するが，自然数やブール値そのものではなく，それらを表す記号表現を分類するための手段であることに注意してほしい．

**■自然数型 N**　ひとまず集合論に戻って，自然数の集合 $\mathbb{N}$ について考える．この集合は，次のように定義することができる．

(i) 0 は自然数である．
(ii) $x$ が自然数ならば $x+1$ も自然数である．
(iii) 以上により構成されるもののみが自然数である．

この帰納的定義に沿って自然数を表現しようというのが, 自然数型 $\mathbf{N}$ の意図である. (i) と (ii) に対応して, 自然数型 $\mathbf{N}$ には 2 つの**構成子**が備わっている.

$$\mathsf{0} : \mathbf{N}, \qquad \mathsf{s} : \mathbf{N} \Rightarrow \mathbf{N}.$$

ただし $\mathbf{N} \Rightarrow \mathbf{N}$ は, 自然数を受け取ったら自然数を返すプログラムの型であり, (**N** から **N** への) **関数型**といわれる. すでに述べたように, $\mathsf{0}$ と $\mathsf{s}$ を用いれば次々に自然数を作り出すことができる. すなわち

$$\mathsf{0}, \quad \mathsf{s}\,\mathsf{0}, \quad \mathsf{s}\,(\mathsf{s}\,\mathsf{0}), \quad \mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,\mathsf{0})), \quad \ldots$$

等々である. (iii) により, どんな自然数もこのようにして作り出せることがわかる.

**■ブール型 B**　ブール値の集合 $\mathbb{B}$ も $\mathbb{N}$ と同様, 次のように定義することができる.

(i) *true* はブール値である.
(ii) *false* はブール値である.
(iii) 以上の 2 つのみがブール値である.

この集合を表現するために, ブール型 $\mathbf{B}$ には次の 2 つの構成子が備わっている.

$$\mathsf{true} : \mathbf{B}, \qquad \mathsf{false} : \mathbf{B}.$$

このように一定数の構成子により定義されるデータ型のことを**帰納的データ型**という.

**■一階型**　データ型やデータ上の関数型を合わせて**一階型**という. 厳密な定義は以下の通りである.

(i) データ型 $D$ は一階型である.
(ii) $D$ がデータ型で $A$ が一階型ならば $(D \Rightarrow A)$ は一階型である.
(iii) 以上により構成されるもののみが一階型である.

$\mathbb{N}$ や $\mathbb{B}$ の定義と同様, この定義もまた帰納的であることに注意. (iii) は大切な条件であるが, 決まりきった形をしているので今後は省略する.

たとえば $\mathbf{N}$ や $\mathbf{B}$ はデータ型なので一階型である. それゆえ $(\mathbf{N} \Rightarrow \mathbf{B})$ は一階型である. したがって $(\mathbf{N} \Rightarrow (\mathbf{N} \Rightarrow \mathbf{B}))$ や

$$(\mathbf{B} \Rightarrow (\mathbf{N} \Rightarrow (\mathbf{N} \Rightarrow \mathbf{B}))) \tag{9}$$

も一階型である. 先ほどと同様に, 一番外側のカッコは省略する. また $(D_1 \Rightarrow (D_2 \Rightarrow A))$ のかわりに $D_1 \Rightarrow D_2 \Rightarrow A$ と書く. すると (9) は $\mathbf{B} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B}$ と書ける. 以上の省略を認めれば, 一階型の一般形は

$$D_1 \Rightarrow \cdots D_n \Rightarrow D_0$$

となる（ただし $D_0, \ldots, D_n$ はデータ型）. 一階型 $(D \Rightarrow A)$ において $D$ は常にデータ型である点に注意. 「一階」型と呼ばれるのは, この制限のためである. ここで $D$ の代わりに関数型を用いてもよいことにすると $(\mathbf{N} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{N}$ のような**二階型**が得られる. 同様にすれば, 三階型, 四階型, より一般に**高階型**が得られるが, 本稿では扱わない.

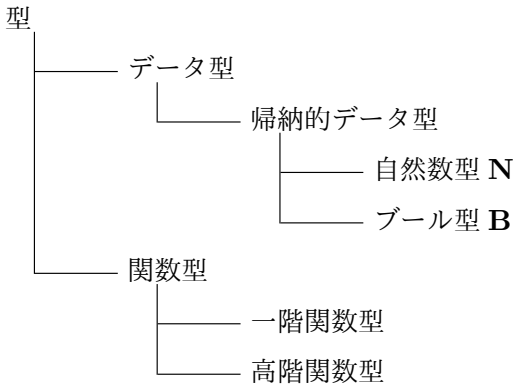


図 1　型の分類：本稿ではデータ型と一階関数型を合わせて一階型と呼ぶ.

例として一階型 $\mathbf{N} \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N})$ を考えよう. これは「自然数から『自然数上の関数』への関数型」を表す. つまり集合 $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ に対応する型であるが, 命題 1.10（カリー化）により同型性

$$\mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \quad \cong \quad \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

が成り立つ. それゆえ $M : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ なるプログラム $M$ は, 「2 つの自然数を受け取ったら 1 つの自然数を返す関数」を表すものと思ってよい.

たくさんの種類の型が出てきたので, 図 1 にまとめておく.

**■型の割り当て**　次に, 各式に型を割り当てる. 式とは, 構成子であるか, 識別子であるか, 変数であるか, $(M\ N)$ の形であるかのいずれかなので, それぞれについて型の割り当て方を考えればよい. 構成子の型はすでに定まっている.

$$\mathsf{0} : \mathbf{N}, \qquad \mathsf{s} : \mathbf{N} \Rightarrow \mathbf{N}, \qquad \mathsf{true} : \mathbf{B}, \qquad \mathsf{false} : \mathbf{B}.$$

変数の型は, 各変数を用いるときに**宣言**する.

$$x : A$$

の形の表現, あるいはその有限列

$$x_1 : A_1, \ldots, x_n : A_n \qquad （x_1, \ldots, x_n \text{ は互いに異なる変数}）$$

のことを変数の**型宣言**（**型環境**）という. 型宣言を $\Gamma$ や $\Delta$ を用いて表す. ここでは一階のプログラムのみを考えるので, **変数の型はすべてデータ型である**とする. つまり上の $A$ や $A_1, \ldots, A_n$ は全部データ型（ここでは $\mathbf{N}$ か $\mathbf{B}$ のどちらか）である.

識別子の型については後述する.

最後に $(M\ N)$ の形の式には次の規則を用いて型を割り当てる.

$$\frac{M : A \Rightarrow B \quad N : A}{(M\ N) : B} \tag{10}$$

この規則は「$M : A \Rightarrow B$ と $N : A$ が成り立つならば, $(M\ N) : B$ が成り立つ」ことを表す. たとえば, 識別子 $\mathsf{add}, \mathsf{even?}$ がそれぞれ

$$\mathsf{add} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}, \qquad \mathsf{even?} : \mathbf{N} \Rightarrow \mathbf{B}$$

という型を持つと仮定すれば, 型宣言 $x : \mathbf{N}$ のもとで次のような導出ができる.

$$\dfrac{\mathsf{even?} : \mathbf{N} \Rightarrow \mathbf{B} \qquad \dfrac{\dfrac{\mathsf{add} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \quad 0 : \mathbf{N}}{\mathsf{add}\ 0 : \mathbf{N} \Rightarrow \mathbf{N}} \quad \dfrac{\mathsf{s} : \mathbf{N} \Rightarrow \mathbf{N} \quad x : \mathbf{N}}{\mathsf{s}\ x : \mathbf{N}}}{\mathsf{add}\ 0\ (\mathsf{s}\ x) : \mathbf{N}}}{\mathsf{even?}\ (\mathsf{add}\ 0\ (\mathsf{s}\ x)) : \mathbf{B}}$$

つまり型宣言 $x : \mathbf{N}$ のもとで $\mathsf{even?}\ (\mathsf{add}\ 0\ (\mathsf{s}\ x)) : \mathbf{B}$ が成り立つ.

このように, 型宣言 $\Gamma$ のもとで $M : A$ が成り立つとき

$$\Gamma \rhd M : A$$

と書く.

大事な約束として, 今後は**型を持つ式のみを考える**ことにする. つまり $(\mathsf{s}\ \mathsf{true})$ や $(0\ \mathsf{s})$ のような型エラーを含む式は考えない.

**補足 2.2** 規則 (10) から型の部分のみを抽出すれば

$$\dfrac{A \Rightarrow B \quad A}{B}$$

という規則が得られる. 「$A \Rightarrow B$」を「$A$ ならば $B$」と読むことにすれば, これは「前提 $A \Rightarrow B$ と $A$ が成り立つならば結論 $B$ が成り立つ」という最も基本的な論理法則（**Modus Ponens**）に他ならない. このことは偶然の一致ではなく, 「型=命題」（および「式=証明」）という対応関係を広範囲で成り立たせることができる. そうすることで計算（プログラム）の理論と証明の理論が統一できる. これを**カリー・ハワード対応**という.

## 2.3 原始再帰的プログラム

いよいよ原始再帰的プログラムの定義に入る. 一般論を進める前に, まずは具体例をいくつか挙げておく.

| | | | | | |
|---|---|---|---|---|---|
| $\mathsf{id}$ | : | $D \Rightarrow D$ | $\mathsf{add2}$ | : | $\mathbf{N} \Rightarrow \mathbf{N}$ |
| $\mathsf{id}\ x$ | = | $x$ | $\mathsf{add2}\ x$ | = | $\mathsf{s}\ (\mathsf{s}\ x)$ |
| $\mathsf{proj}_1$ | : | $D_1 \Rightarrow D_2 \Rightarrow D_1$ | $\mathsf{ctrue}$ | : | $D \Rightarrow \mathbf{B}$ |
| $\mathsf{proj}_1\ x\ y$ | = | $x$ | $\mathsf{ctrue}\ x$ | = | $\mathsf{true}$ |
| $\mathsf{proj}_2$ | : | $D_1 \Rightarrow D_2 \Rightarrow D_2$ | $\mathsf{add4}$ | : | $\mathbf{N} \Rightarrow \mathbf{N}$ |
| $\mathsf{proj}_2\ x\ y$ | = | $y$ | $\mathsf{add4}\ x$ | = | $\mathsf{add2}\ (\mathsf{add2}\ x)$ |

左上は, 最も基本的な**恒等関数**である. これには id という名前（識別子）をつけてある. $D \Rightarrow D$ という型をみれば, id は型 $D$ の入力を受け取ったら同じ型の出力を返すことがわかる.

次に $\mathsf{proj}_1$, $\mathsf{proj}_2$ は**射影**を表す. 型 $D_1 \Rightarrow D_2 \Rightarrow D_i$ が示す通り, どちらも 2 つの引数をとり, $\mathsf{proj}_1$ は第 1 引数を, $\mathsf{proj}_2$ は第 2 引数を返す. $D_1$ と $D_2$ は同じ型であってもよい.

add2 は, 与えられた入力に 2 を足す関数である. s を使っているので, 引数 $x$ の型は $\mathbf{N}$ でなければならないし, 全体 $\mathsf{s}\ (\mathsf{s}\ x)$ の型も $\mathbf{N}$ でなければならない. よって識別子 add2 の型は $\mathbf{N} \Rightarrow \mathbf{N}$ となる.

ctrue は, 引数の値に関わらず true を返す**定数関数**である. このように, プログラム中で用いられない引数があってもよい.

最後に add4 は, add2 を 2 つ合成してつくった人為的なプログラムである. add2 の型は $\mathbf{N} \Rightarrow \mathbf{N}$ なので, 全体の型も $\mathbf{N} \Rightarrow \mathbf{N}$ となる. このように, すでに定義した関数を用いてどんどん新しい関数をつくっていくことができる.

関数は**帰納的に**定義することもできる.

$$
\begin{array}{lcllcl}
\mathsf{add} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} & \mathsf{and} & : & \mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{B} \\
\mathsf{add}\ 0\ x & = & x & \mathsf{and}\ \mathsf{true}\ x & = & x \\
\mathsf{add}\ (\mathsf{s}\ y)\ x & = & \mathsf{s}\ (\mathsf{add}\ y\ x) & \mathsf{and}\ \mathsf{false}\ x & = & \mathsf{false}
\end{array}
$$

$\mathbf{N}$ も $\mathbf{B}$ も 2 つの構成子を持つので, 2 つの場合に分けて定義を行っている. たとえば add の場合は, 第 1 引数が 0 の場合と $(\mathsf{s}\ y)$ の場合に分けて関数定義を行っている. ところで 3 行目では, $\mathsf{add}\ (\mathsf{s}\ y)\ x$ の値を定めるのに $\mathsf{add}\ y\ x$ の値を用いている. ある意味自己言及的だが, 第 1 引数の値が減っているため循環論には陥らない. このように帰納的データ型の構造に沿って関数を定めることを, 関数の**帰納的定義**, あるいは**原始再帰法**, **原始帰納法**などという.

**■関数の定義法**　ここから一般論に移る. すでに定義した関数から新しい関数を定義するための方法として, 次の 3 つを考える（一般論なのでわかりにくいかもしれないが, 次節で例たくさん挙げるので, それらを通して理解してほしい）.

1. 合成による定義

$$
\begin{array}{lcl}
\mathsf{f} & : & D_1 \Rightarrow \cdots D_n \Rightarrow D_0 \\
\mathsf{f}\ x_1\ \cdots\ x_n & = & M
\end{array}
$$

ただし式 $M$ は

$$x_1 : D_1,\ \ldots,\ x_n : D_n \rhd M : D_0$$

を満たすものとする. これを f **の定義**といい, $M$ を f **の定義式**という.

$M$ の中では, 構成子や変数 $x_1, \ldots, x_n$ に加え, すでに定義済みの識別子を用いることができる. $\mathsf{f}$ の型 $D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ は変数の型 $x_1 : D_1, \ldots, x_n : D_n$ および定義式の型 $M : D_0$ と対応する. 最初に例に挙げた $\mathsf{id}, \mathsf{proj}_1, \mathsf{proj}_2, \mathsf{add2}, \mathsf{ctrue}, \mathsf{add4}$ は全部このパターンに当てはまる.

以下では変数の列 $x_1 \ \cdots \ x_n$ を $\overline{x}$ と書き, 型 $D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ を $\overline{D} \Rightarrow D_0$ と書き, 型宣言 $x_1 : D_1, \ \ldots, \ x_n : D_n$ を $\Gamma$ と書く.

2. 原始再帰法による定義（$\mathbf{B}$）

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{B} \Rightarrow \overline{D} \Rightarrow D_0 \\ \mathsf{f}\ \mathsf{true}\ \overline{x} & = & M_{\mathsf{true}} \\ \mathsf{f}\ \mathsf{false}\ \overline{x} & = & M_{\mathsf{false}} \end{array}$$

ただし式 $M_{\mathsf{true}}, M_{\mathsf{false}}$ は

$$\Gamma \rhd M_{\mathsf{true}} : D_0, \qquad \Gamma \rhd M_{\mathsf{false}} : D_0$$

を満たすものとする. これを $\mathsf{f}$ **の定義**といい, $M_{\mathsf{true}}, M_{\mathsf{false}}$ を $\mathsf{f}$ **の定義式**という.

3. 原始再帰法による定義（$\mathbf{N}$）

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{N} \Rightarrow \overline{D} \Rightarrow D_0 \\ \mathsf{f}\ \mathsf{0}\ \overline{x} & = & M_{\mathsf{0}} \\ \mathsf{f}\ (\mathsf{s}\ y)\ \overline{x} & = & M_{\mathsf{s}} \end{array}$$

ただし $y$ は $x_1, \ldots, x_n$ とは異なる変数であり, 式 $M_{\mathsf{0}}, M_{\mathsf{s}}$ は

$$\Gamma \rhd M_{\mathsf{0}} : D_0, \qquad \Gamma, y : \mathbf{N} \rhd M_{\mathsf{s}} : D_0$$

を満たすものとする. また $M_{\mathsf{s}}$ の中で式 $(\mathsf{f}\ y\ \overline{x})$ を用いてもよい. これを $\mathsf{f}$ **の定義**といい, $M_{\mathsf{0}}, M_{\mathsf{s}}$ を $\mathsf{f}$ **の定義式**という.

これまでと違い, $M_{\mathsf{s}}$ の中では $x_1, \ldots, x_n$ に加えて変数 $y$ や式 $(\mathsf{f}\ y\ \overline{x})$ を用いることができる. これはつまり $(\mathsf{f}\ (\mathsf{s}\ y)\ \overline{x})$ の値を定めるのに $y$ や $(\mathsf{f}\ y\ \overline{x})$ の値を参照してもよいということである. とはいえ $(\mathsf{f}\ (\mathsf{s}\ y)\ \overline{x})$ や $(\mathsf{f}\ y\ (\mathsf{s}\ (\mathsf{s}\ y)))$ といった類似表現の値を参照することはできない. あくまでもたったひとつの形 $(\mathsf{f}\ y\ \overline{x})$ のみである. それゆえ

$$\begin{array}{lcl} \mathsf{add}\ \mathsf{0}\ x & = & x \\ \mathsf{add}\ (\mathsf{s}\ y)\ x & = & \mathsf{s}\ (\mathsf{add}\ y\ x) \end{array}$$

は認められるが,

$$\begin{array}{lcl} \mathsf{bad}\ \mathsf{0}\ x & = & x \\ \mathsf{bad}\ (\mathsf{s}\ y)\ x & = & \mathsf{s}\ (\mathsf{bad}\ y\ (\mathsf{s}\ x)) \end{array}$$

は認められない.

■**原始再帰的プログラム**　以上 3 種類の関数定義を繰り返したものがプログラムである. 正確に言えば, **原始再帰的プログラム** $P$ とは, 識別子 $\mathsf{f}_1, \ldots, \mathsf{f}_n$ の定義の列のことである. ただし $\mathsf{f}_1, \ldots, \mathsf{f}_n$ は互いに異なり, $\mathsf{f}_k$（$1 \leq k \leq n$）の定義式の中では $\mathsf{f}_1, \ldots, \mathsf{f}_{k-1}$ 以外の識別子を用いてはならない. つまり未定義の識別子を用いたり, 循環的な定義をしてはならないということである. たとえば

$$
\begin{array}{lcl}
\mathsf{even?} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{even?}\ 0 & = & \mathsf{true} \\
\mathsf{even?}\ (\mathsf{s}\ x) & = & \mathsf{odd?}\ x
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{odd?} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{odd?}\ 0 & = & \mathsf{false} \\
\mathsf{odd?}\ (\mathsf{s}\ x) & = & \mathsf{even?}\ x
\end{array}
$$

は原始再帰的プログラムではない. なぜならば識別子 even?, odd? の定義が相互に依存しあっているからである.

プログラム $P$ に現れる最後の識別子 $\mathsf{f}_n$ をプログラム全体の名前だと思い, $P$ のことを**プログラム** $\mathsf{f}_n$ と呼ぶこともある.

## 2.4　原始再帰的プログラムの表現力

ここでは原始再帰的プログラムによりどんな関数を表せるか, 例を挙げて見ていくことにする. まずはブール値に関する例を挙げる.

1. ブール値に関するもの

$$
\begin{array}{lcl}
\mathsf{and} & : & \mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{B} \\
\mathsf{and\ true}\ x & = & x \\
\mathsf{and\ false}\ x & = & \mathsf{false}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{not} & : & \mathbf{B} \Rightarrow \mathbf{B} \\
\mathsf{not\ true} & = & \mathsf{false} \\
\mathsf{not\ false} & = & \mathsf{true}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{or} & : & \mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{B} \\
\mathsf{or\ true}\ x & = & \mathsf{true} \\
\mathsf{or\ false}\ x & = & x
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{if} & : & \mathbf{B} \Rightarrow A \Rightarrow A \Rightarrow A \\
\mathsf{if\ true}\ x\ y & = & x \\
\mathsf{if\ false}\ x\ y & = & y
\end{array}
$$

これらはすべて原始再帰法（$\mathbf{B}$）の形式にのっとっている. and, or の定義が以下の**真理値表**を反映していることを確認してほしい.

| $y$ | $x$ | and $y$ $x$ | or $y$ $x$ |
|---|---|---|---|
| false | false | false | false |
| false | true | false | true |
| true | false | false | true |
| true | true | true | true |

if は今後頻繁に用いるので, 読みやすくするために以下の記法を用いる.

$$\mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 \quad := \quad \mathsf{if}\ M_1\ M_2\ M_3$$

次に自然数に関する例を挙げる.

2. 自然数に関するもの

$$
\begin{array}{lcl}
\mathsf{mul} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{mul}\ 0\ x & = & 0 \\
\mathsf{mul}\ (\mathsf{s}\ y)\ x & = & \mathsf{add}\ x\ (\mathsf{mul}\ y\ x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{exp} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{exp}\ 0\ x & = & 1 \\
\mathsf{exp}\ (\mathsf{s}\ y)\ x & = & \mathsf{mul}\ x\ (\mathsf{exp}\ y\ x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{fact} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{fact}\ 0 & = & 1 \\
\mathsf{fact}\ (\mathsf{s}\ y) & = & \mathsf{mul}\ (\mathsf{s}\ y)\ (\mathsf{fact}\ y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{pred} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{pred}\ 0 & = & 0 \\
\mathsf{pred}\ (\mathsf{s}\ y) & = & y
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{sub} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{sub}\ 0\ x & = & x \\
\mathsf{sub}\ (\mathsf{s}\ y)\ x & = & \mathsf{pred}\ (\mathsf{sub}\ y\ x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{zero?} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{zero?}\ 0 & = & \mathsf{true} \\
\mathsf{zero?}\ (\mathsf{s}\ y) & = & \mathsf{false}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{leq} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{leq}\ x\ y & = & \mathsf{zero?}\ (\mathsf{sub}\ y\ x)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{eq} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{eq}\ x\ y & = & \mathsf{and}\ (\mathsf{leq}\ x\ y)\ (\mathsf{leq}\ y\ x)
\end{array}
$$

mul, exp, fact はそれぞれ掛け算 $y \times x$, べき乗 $y^x$, 階乗 $y!$ を計算するプログラムである. たとえば $2 \times 2$ の計算は次のようになる.

$$
\begin{array}{rcl}
\mathsf{mul}\ 2\ 2 & = & \mathsf{mul}\ (\mathsf{s}\ (\mathsf{s}\ 0))\ 2 \\
& = & \mathsf{add}\ 2\ (\mathsf{mul}\ (\mathsf{s}\ 0)\ 2) \\
& = & \mathsf{add}\ 2\ (\mathsf{add}\ 2\ (\mathsf{mul}\ 0\ 2)) \\
& = & \mathsf{add}\ 2\ (\mathsf{add}\ 2\ 0) \\
& = & \cdots\ =\ 4.
\end{array}
$$

pred は入力から 1 を引くプログラム（ただし $\mathsf{pred}\ 0 = 0$）であり, sub はその繰り返しだから引き算を表す. 正確には「第 2 引数 $x$ − 第 1 引数 $y$」である（ただし $y > x$ のとき $\mathsf{sub}\ y\ x = 0$）. zero? は入力が 0 かどうかを判定する. $\mathsf{leq}\ x\ y = \mathsf{true}$ となるのは, $x \leq y$ が成り立つときである. たとえば :

$$
\begin{array}{rcl}
\mathsf{leq}\ 5\ 3 & = & \mathsf{zero?}\ (\mathsf{sub}\ 3\ 5) \\
& = & \cdots\ =\ \mathsf{zero?}\ (\mathsf{s}\ (\mathsf{s}\ 0)) \\
& = & \mathsf{false}.
\end{array}
$$

最後に $\mathsf{eq}\ x\ y = \mathsf{true}$ となるのは, $x = y$ が成り立つときである. 以上を順次確認してほしい.

**■有界量化**　プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{B}$ と自然数 $n$ が与えられたとき,

$$
\text{ある } k < n \text{ について } \mathsf{f}\ \mathsf{k} = \mathsf{true}
$$

が成り立つかどうかを判定するにはどうすればよいだろうか？それには次のようなプログラムを考えればよい.

$$
\begin{array}{lcl}
\exists^b\mathsf{f} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\exists^b\mathsf{f}\ 0 & = & \mathsf{false} \\
\exists^b\mathsf{f}\ (\mathsf{s}\ y) & = & \mathsf{or}\ (\mathsf{f}\ y)\ (\exists^b\mathsf{f}\ y)
\end{array}
$$

そうすれば, たとえば

$$\begin{array}{rcl} \exists^b\mathsf{f}\ 3 & = & \mathsf{or}\ (\mathsf{f}\ 2)\ (\exists^b\mathsf{f}\ 2) \\ & = & \mathsf{or}\ (\mathsf{f}\ 2)\ (\mathsf{or}\ (\mathsf{f}\ 1)\ (\exists^b\mathsf{f}\ 1)) \\ & = & \mathsf{or}\ (\mathsf{f}\ 2)\ (\mathsf{or}\ (\mathsf{f}\ 1)\ (\mathsf{or}\ (\mathsf{f}\ 0)\ (\exists^b\mathsf{f}\ 0))) \\ & = & \mathsf{or}\ (\mathsf{f}\ 2)\ (\mathsf{or}\ (\mathsf{f}\ 1)\ (\mathsf{or}\ (\mathsf{f}\ 0)\ \mathsf{false})) \end{array}$$

となるので, 結局

$$\begin{array}{rcl} \exists^b\mathsf{f}\ \mathsf{n} = \mathsf{true} & \Longleftrightarrow & \mathsf{f\ n\text{-}1} = \mathsf{true} \text{ または } \mathsf{f\ n\text{-}2} = \mathsf{true} \text{ または } \cdots \text{ または } \mathsf{f}\ 0 = \mathsf{true} \\ & \Longleftrightarrow & \text{ある } k < n \text{ について } \mathsf{f}\ \mathsf{k} = \mathsf{true} \end{array}$$

となって意図通りの結果が得られる.

同様にすれば

$$\text{すべての } k < n \text{ について } \mathsf{f}\ \mathsf{k} = \mathsf{true}$$

かどうかを調べるプログラム $\forall^b\mathsf{f}$ も定義することができる. これらを**有界量化**という.

これまでに定義した関数を組み合わせれば, 自然数に関する大抵の性質は判定できる. たとえば:

$x$ は $y$ の約数かどうかを判定するプログラム div

$$\begin{array}{lcl} \mathsf{meq} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{meq}\ y\ x\ z & = & \mathsf{eq}\ (\mathsf{mul}\ y\ x)\ z \\ & & \\ \exists^b\mathsf{meq} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \exists^b\mathsf{meq}\ 0\ x\ z & = & \mathsf{false} \\ \exists^b\mathsf{meq}\ (\mathsf{s}\ y)\ x\ z & = & \mathsf{or}\ (\mathsf{meq}\ y\ x\ z)\ (\exists^b\mathsf{meq}\ y\ x\ z) \\ & & \\ \mathsf{div} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{div}\ x\ y & = & \exists^b\mathsf{meq}\ (\mathsf{s}\ y)\ x\ y \end{array}$$

このコーディングが正しいことを見るには,

$$\begin{array}{lcl} \mathsf{meq}\ y\ x\ z = \mathsf{true} & \Longleftrightarrow & yx = z \\ \exists^b\mathsf{meq}\ y_0\ x\ z = \mathsf{true} & \Longleftrightarrow & \text{ある } y < y_0 \text{ について } yx = z \\ \mathsf{div}\ x\ y_0 = \mathsf{true} & \Longleftrightarrow & \text{ある } y < y_0 + 1 \text{ について } yx = y_0 \end{array}$$

となることを順次確かめればよい.

同様にすれば素数判定をする原始再帰的プログラムも書くことができる.

$x$ が素数かどうかを判定するプログラム prime

$$
\begin{array}{lcl}
\mathsf{prm} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{prm}\ y\ x & = & \mathsf{or}\ (\mathsf{not}\ (\mathsf{div}\ y\ x))\ (\mathsf{eq}\ y\ 1) \\
 & & \\
\forall^b\mathsf{prm} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\
\forall^b\mathsf{prm}\ 0\ x & = & \mathsf{true} \\
\forall^b\mathsf{prm}\ (\mathsf{s}\ y)\ x & = & \mathsf{and}\ (\mathsf{prm}\ y\ x)\ (\forall^b\mathsf{prm}\ y\ x) \\
 & & \\
\mathsf{prime} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{prime}\ x & = & \mathsf{and}\ (\mathsf{leq}\ 2\ x)\ (\forall^b\mathsf{prm}\ x\ x)
\end{array}
$$

確かめるべきことは以下の通りである.

$$
\begin{array}{lcl}
\mathsf{prm}\ y\ x = \mathsf{true} & \Longleftrightarrow & y \text{ は } x \text{ の約数でないかまたは } 1 \\
 & & (\text{つまり } y \text{ は } 1 \text{ 以外の } x \text{ の約数ではない}) \\
\forall^b\mathsf{prm}\ y_0\ x = \mathsf{true} & \Longleftrightarrow & \text{すべての } y < y_0 \text{ について } y \text{ は } 1 \text{ 以外の } x \text{ の約数ではない} \\
 & & (\text{つまり } x \text{ は } y_0 \text{ 未満で } 1 \text{ 以外の約数を持たない}) \\
\mathsf{prime}\ x = \mathsf{true} & \Longleftrightarrow & x \text{ は } 2 \text{ 以上で } x, 1 \text{ 以外の約数を持たない.}
\end{array}
$$

**補足 2.3** 有界量化は, **非有界**の量化「ある $k \in \mathbb{N}$ について $\mathsf{f}\ \mathsf{k} = \mathsf{true}$」「すべての $k \in \mathbb{N}$ について $\mathsf{f}\ \mathsf{k} = \mathsf{true}$」とは本質的に異なる. 後者は一般に原始再帰的でない（それどころか計算可能ではない）.

## 2.5 原始再帰的プログラムの停止性

すでに述べたように, 原始再帰的プログラム $P$ とは相異なる識別子 $\mathsf{f}_1, \ldots, \mathsf{f}_n$ の定義の列のことである（重要なのは最後の識別子 $\mathsf{f}_n$ である. これを適切に定義するために段階を踏んで $\mathsf{f}_1, \ldots, \mathsf{f}_{n-1}$ を定義するのである. そのため $\mathsf{f}_n$ のことをプログラム $P$ 全体の名前としても用いる）. そして原始再帰的プログラムの実行は, 定義に従って式変形を行うことで進められる. その際, どんな入力値を与えても計算は必ず有限ステップで停止し, 出力値が得られる. つまり無限ループに陥ったり計算が発散してしまうことはない. このことをプログラムの**停止性**という. 以下では, 原始再帰的プログラムの停止性についてもう少しだけ詳しく見ていくことにする.

ブール値 $\mathsf{true}, \mathsf{false}$ と数項を合わせて**値**または**値式**と呼ぶ.

$$\mathsf{true}, \quad \mathsf{false}, \quad \mathsf{0}, \quad \mathsf{1}, \quad \mathsf{2}, \quad \cdots$$

すなわち値式とは, 構成子のみを用いて作られる式で, 型エラーを含まないもののことである.

**定理 2.4(原始再帰的プログラムの停止性)**

$\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ を原始再帰的プログラムとする. 値式 $\mathsf{v}_1 : D_1, \cdots, \mathsf{v}_n : D_n$ が与えられたとき, 式変形を有限回繰り返せば必ず何らかの値式 $\mathsf{w} : D_0$ に到達する.

$$\mathsf{f}\ \mathsf{v}_1\ \cdots\ \mathsf{v}_n\ =\ \cdots\ =\ \cdots\ =\ \mathsf{w}.$$

**証明：** アイデアのみ述べる. 仮定により識別子 $\mathsf{f}_1, \ldots, \mathsf{f}_n$ の定義の列が存在し, $\mathsf{f} = \mathsf{f}_n$ である. 各 $1 \leq i \leq n$ について $\mathsf{f}_i$ が停止性を満たすことを, $i$ についての帰納法で証明する.

1. $\mathsf{f}_i$ が合成により定義されているとき. 一般論は大変なので, 具体例について考える. たとえば $i = 5$ として $\mathsf{f}_5$ の定義が以下のようであるとする.

$$\begin{array}{lcl} \mathsf{f}_5 & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{f}_5\ x\ y & = & \mathsf{f}_4\ (\mathsf{f}_3\ x\ y)\ (\mathsf{s}\ (\mathsf{f}_1\ x)) \end{array}$$

帰納法の仮定により $\mathsf{f}_1, \mathsf{f}_3, \mathsf{f}_4$ は停止性を満たすとしてよい. よって値式 $\mathsf{m}, \mathsf{n}$ が与えられたとき, ある値式 $\mathsf{v}_1, \mathsf{v}_3, \mathsf{v}_4$ が存在して以下が成り立つ.

$$\mathsf{f}_5\ \mathsf{m}\ \mathsf{n}\ =\ \mathsf{f}_4\ (\mathsf{f}_3\ \mathsf{m}\ \mathsf{n})\ (\mathsf{s}\ (\mathsf{f}_1\ \mathsf{m}))\ = \cdots =\ \mathsf{f}_4\ \mathsf{v}_3\ (\mathsf{s}\ (\mathsf{f}_1\ \mathsf{m}))\ = \cdots =\ \mathsf{f}_4\ \mathsf{v}_3\ (\mathsf{s}\ \mathsf{v}_1)\ = \cdots =\ \mathsf{v}_4.$$

（$\mathsf{s}\ \mathsf{v}_1$ も値式であることに注意. ）よって $\mathsf{f}_5$ は停止性を満たす.

2. $\mathsf{f}_i$ が原始再帰法（$\mathbf{N}$）により定義されているとき. ふたたび $i = 5$ として $\mathsf{f}_5$ の定義が以下のようであるとする.

$$\begin{array}{lcl} \mathsf{f}_5 & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{f}_5\ \mathsf{0}\ x & = & \mathsf{f}_2\ x \\ \mathsf{f}_5\ (\mathsf{s}\ y)\ x & = & \mathsf{f}_4\ y\ (\mathsf{f}_5\ y\ x) \end{array}$$

帰納法の仮定により $\mathsf{f}_2, \mathsf{f}_4$ は停止性を満たすとしてよい. $\mathsf{m}, \mathsf{n}$ を値式とするとき, ある値 $\mathsf{v}_m$ が存在して $\mathsf{f}_5\ \mathsf{m}\ \mathsf{n} = \cdots = \mathsf{v}_m$ となることを, $m$ についての帰納法で示す. $m = 0$ のときは, $\mathsf{f}_2$ の停止性より

$$\mathsf{f}_5\ \mathsf{m}\ \mathsf{n}\ =\ \mathsf{f}_5\ \mathsf{0}\ \mathsf{n}\ =\ \mathsf{f}_2\ \mathsf{n}\ =\ \cdots\ =\ \mathsf{v}_0$$

を満たす値式 $\mathsf{v}_0$ が存在する.

$m = m' + 1$ のときは, 帰納法の仮定により $\mathsf{f}_5\ \mathsf{m}'\ \mathsf{n} = \cdots = \mathsf{v}_{m'}$ を満たす値式 $\mathsf{v}_{m'}$ が存在する. よって $\mathsf{f}_4$ の停止性より

$$\mathsf{f}_5\ \mathsf{m}\ \mathsf{n}\ =\ \mathsf{f}_5\ (\mathsf{s}\ \mathsf{m}')\ \mathsf{n}\ =\ \mathsf{f}_4\ \mathsf{m}'\ (\mathsf{f}_5\ \mathsf{m}'\ \mathsf{n})\ = \cdots =\ \mathsf{f}_4\ \mathsf{m}'\ \mathsf{v}_{m'}\ = \cdots =\ \mathsf{v}_m$$

を満たす値式 $\mathsf{v}_m$ が存在する. ゆえに $\mathsf{f}_5$ は停止性を満たす.

3. $\mathsf{f}_i$ が原始再帰法（$\mathbf{B}$）により定義されているとき. 省略. □

**■原始再帰的関数**　停止性が示されたので, 原始再帰的プログラムに関数を割り当てることができる. それにはプログラミング言語の諸概念を集合論の言語へと引き戻せばよい. まず, データ型 $D$ に対して集合 $[\![D]\!]$ を以下のように定める.

$$[\![\mathbf{N}]\!] := \mathbb{N}, \qquad [\![\mathbf{B}]\!] := \mathbb{B}.$$

ここで値式 $\mathsf{n} : \mathbf{N}$ と自然数 $n \in \mathbb{N}$ が一対一に対応することに注意する. 同様に値式 $\mathsf{true}, \mathsf{false} : \mathbf{B}$ とブール値 $\mathit{true}, \mathit{false} \in \mathbb{B}$ も一対一に対応する.

原始再帰的プログラム $\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ に対しては, 関数

$$[\![\mathsf{f}]\!] : [\![D_1]\!] \times \cdots \times [\![D_n]\!] \to [\![D_0]\!]$$

を以下のように定める. 任意の $v_1 \in [\![D_1]\!], \cdots, v_n \in [\![D_n]\!]$, $w \in [\![D_0]\!]$ について

$$[\![\mathsf{f}]\!](v_1, \ldots, v_n) = w \quad \Longleftrightarrow \quad \mathsf{f}\ \mathsf{v}_1 \cdots \mathsf{v}_n = \cdots = \mathsf{w}.$$

プログラム $\mathsf{f}$ の停止性により, 任意の $v_1, \ldots, v_n$ に対して $[\![\mathsf{f}]\!](v_1, \ldots, v_n) = w$ を満たす $w$ が（ただ 1 つ）存在する. よって $[\![\mathsf{f}]\!]$ は確かに関数である.

このようにして原始再帰的プログラム $\mathsf{f}$ から得られる関数 $[\![\mathsf{f}]\!]$ のことを**原始再帰的関数**という.

たとえば既出のプログラム $\mathsf{add} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ に対しては, 自然数の足し算

$$[\![\mathsf{add}]\!] : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

が対応する. よって足し算は原始再帰的である. 掛け算, べき乗等についても同様である.

さて, 値域の型が $\mathbf{B}$ であるような原始再帰的プログラム $\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow \mathbf{B}$ について考えてみる. これには原始再帰的関数 $[\![\mathsf{f}]\!] : [\![D_1]\!] \times \cdots \times [\![D_n]\!] \to \mathbb{B}$ が対応するのであるが, 命題 1.9 を用いれば, さらに $n$ 項関係 $P \subseteq [\![D_1]\!] \times \cdots \times [\![D_n]\!]$ が得られ $\chi_P = [\![\mathsf{f}]\!]$ が成り立つ. このようにして原始再帰的プログラムから得られる集合・関係のことを**原始再帰的集合・原始再帰的関係**という.

たとえばプログラム $\mathsf{prime} : \mathbf{N} \Rightarrow \mathbf{B}$ に対しては, 素数判定関数 $[\![\mathsf{prime}]\!] : \mathbb{N} \to \mathbb{B}$ が対応する. そして同数性 $\mathbb{N} \to \mathbb{B} \simeq \wp(\mathbb{N})$ により, 集合 $\mathit{Prime} \subseteq \mathbb{N}$ が得られる. これは素数全体の集合に他ならない. よって素数全体の集合は原始再帰的であるといえる.

（文字列としての）プログラムと（集合としての）関数の区別に注意. 実際, $[\![\mathsf{f}_1]\!] = [\![\mathsf{f}_2]\!]$ であっても $\mathsf{f}_1 = \mathsf{f}_2$ であるとは限らない. それゆえ同じ関数を表すのであっても, 速いプログラム・遅いプログラムがあり, また構造が明確なプログラムもあれば, 不明確なプログラムもある（俗に**スパゲッティ・コード**と呼ばれる）.

## 2.6 原始再帰的関数の限界

次にどれくらい大きな関数が原始再帰的プログラムで表せるかを調べる. そのために, プログラムの列 $\mathsf{ack}_0$, $\mathsf{ack}_1$, $\mathsf{ack}_2$, ... を次のように定義する（いずれも型は $\mathbf{N} \Rightarrow \mathbf{N}$ である）.

$$
\begin{array}{lcl}
\mathsf{ack}_0\ y & = & \mathsf{s}\ y \\
\\
\mathsf{ack}_1\ \mathsf{0} & = & \mathsf{ack}_0\ \mathsf{1} \\
\mathsf{ack}_1\ (\mathsf{s}\ y) & = & \mathsf{ack}_0\ (\mathsf{ack}_1\ y) \\
\\
\mathsf{ack}_2\ \mathsf{0} & = & \mathsf{ack}_1\ \mathsf{1} \\
\mathsf{ack}_2\ (\mathsf{s}\ y) & = & \mathsf{ack}_1\ (\mathsf{ack}_2\ y) \\
\\
\vdots & & \vdots \\
\\
\mathsf{ack}_{n+1}\ \mathsf{0} & = & \mathsf{ack}_n\ \mathsf{1} \\
\mathsf{ack}_{n+1}\ (\mathsf{s}\ y) & = & \mathsf{ack}_n\ (\mathsf{ack}_{n+1}\ y)
\end{array}
$$

そして

$$ack_n \quad := \quad [\![\mathsf{ack}_n]\!] : \mathbb{N} \to \mathbb{N}$$

とおけば, 原始再帰的関数の列 $ack_0, ack_1, ack_2, \cdots$ が得られる.

具体的にどんな関数か調べてみると,

$$ack_0(y) = y + 1, \qquad ack_1(y) = y + 2, \qquad ack_2(y) = 2y + 3$$

となり, 一般に

$$ack_{n+1}(y) \ = \ \underbrace{ack_n(ack_n(\cdots ack_n}_{y+1\ 回}(1)\cdots))$$

が成り立つ. 実際, たとえば $\mathsf{ack}_3\ \mathsf{3}$ を計算してみると

$$
\begin{array}{rcl}
\mathsf{ack}_3\ \mathsf{3} & = & \mathsf{ack}_2\ (\mathsf{ack}_3\ \mathsf{2}) \\
& = & \mathsf{ack}_2\ (\mathsf{ack}_2\ (\mathsf{ack}_3\ \mathsf{1})) \\
& = & \mathsf{ack}_2\ (\mathsf{ack}_2\ (\mathsf{ack}_2\ (\mathsf{ack}_3\ \mathsf{0}))) \\
& = & \mathsf{ack}_2\ (\mathsf{ack}_2\ (\mathsf{ack}_2\ (\mathsf{ack}_2\ \mathsf{1})))
\end{array}
$$

となり, 初期値 1 に対して $\mathsf{ack}_2$ が $3 + 1 = 4$ 回繰り返されていることがわかる.

このように「繰り返し」が繰り返されるため, $n = 3$ 以降 $ack_n(y)$ の値は急激に大きくなる. たとえば $ack_3(y) \approx 2^{y+1}$ 程度であり,

$$ack_4(y) \ \approx \ 2^{2^{\cdot^{\cdot^{2}}}} \quad (2\ のべき乗を\ y+1\ 回繰り返したもの)$$

程度である. したがって $ack_4(4)$ の値は $2^{65536}$ を超える.

この関数列を用いれば, 原始再帰的関数の大きさに上限を与えることができる.

**定理 2.5**

どんな原始再帰的関数 $f : \mathbb{N}^k \to \mathbb{N}$ についてもある $n \in \mathbb{N}$ が存在し,

$$f(x_1, \ldots, x_k) \ < \ ack_n(\max(x_1, \ldots, x_k)) \qquad (x_1, \ldots, x_k \in \mathbb{N})$$

が成り立つ.

証明は少し面倒なので, 省略する.

このことから原始再帰的ではない関数が即座に得られる.

**定理 2.6**

**アッカーマン関数**

$$Ack(x, y) \ := \ ack_x(y) \ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

は原始再帰的ではない.

**証明：** もしも $Ack$ が原始再帰的だとしたら, 定理 2.5 により $Ack(x, y) \ < \ ack_n(\max(x, y))$ を満たす $n \in \mathbb{N}$ が存在するはずである. しかし $x = y = n$ とおけば

$$Ack(n, n) \ < \ ack_n(\max(n, n)) \ = \ ack_n(n) \ = \ Ack(n, n)$$

となり, 矛盾する. □

しかし, それでも $Ack$ は直感的にいって「計算可能」である. なぜなら入力値 $n, m$ が与えられたら, 原始再帰的プログラム $\mathsf{ack}_n$ を起動し, $\mathsf{ack}_n$ m の値を計算すれば $Ack(n, m)$ の値が得られるからである. 以上から得られる教訓は,

**原始再帰的プログラムだけでは, 計算可能な関数全部を表すことはできない**

ということである. ではどのように拡張したら計算可能関数全体が捉えられるのだろうか? それが問題である.

なお, ここではきちんと議論しないが大雑把にいって次のことが成り立つ.

関数 $f : \mathbb{N} \to \mathbb{N}$ が原始再帰的であるとは, $f$ を計算するコンピュータプログラム f と自然数 $n$ が存在して, (f m) の計算時間を $ack_n(m)$ で抑えられることに他ならない（プログラミング言語はなんでもよい）.

ひらたくいえば, 原始再帰的関数とは計算時間があらかじめ（$ack_n$ で）見積もれるような関数のことなのである. ゆえに自然数上の関数に限っていえば, かなり多くの関数が原始再帰的になる.

## 2.7 直積型とリスト型

本稿では, ここまで帰納的データ型として $\mathbf{N}$ と $\mathbf{B}$ のみを考えてきたが, さらに帰納的データ型を追加すると何が起こるのかを見ておきたい. 具体例として直積型とリスト型を考える.

まず最初に復習であるが, 自然数型 $\mathbf{N}$ は,

$$\mathsf{0} : \mathbf{N}, \qquad \mathsf{s} : \mathbf{N} \Rightarrow \mathbf{N}$$

という 2 つの構成子により定められていたことを思い出してほしい. これと対応して, プログラミングの際には 0 と s に場合分けして原始再帰法を用いることができるのであった. たとえば

$$\begin{array}{lcl} \mathsf{add} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{add}\ \mathsf{0}\ x & = & x \\ \mathsf{add}\ (\mathsf{s}\ y)\ x & = & \mathsf{s}(\mathsf{add}\ y\ x) \end{array}$$

等である. つまり, 原始再帰法の形は構成子の形によって決まる.

以下では $\mathbf{N}$ や $\mathbf{B}$ を使って新しいデータ型を作り出していく.

**■直積型** $A_1, A_2$ を型とするとき, **直積型** $A_1 \times A_2$ を単一の構成子

$$\mathsf{pair} : A_1 \Rightarrow A_2 \Rightarrow A_1 \times A_2$$

により定める. すると $M_1 : A_1$, $M_2 : A_2$ のとき $(\mathsf{pair}\ M_1\ M_2) : A_1 \times A_2$ となる. $(\mathsf{pair}\ M_1\ M_2)$ を省略して

$$\langle M_1, M_2 \rangle$$

とも書く. 直積型に対応する原始再帰法は次の通り. ただし以下では変数の列 $x_1\ \cdots\ x_n$ を $\overline{x}$ と書き, 型 $D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ を $\overline{D} \Rightarrow D_0$ と書き, 型宣言 $x_1 : D_1,\ \ldots,\ x_n : D_n$ を $\Gamma$ と書くことにする.

原始再帰法（直積型）

$$\begin{array}{lcl} \mathsf{f} & : & A_1 \times A_2 \Rightarrow \overline{D} \Rightarrow D_0 \\ \mathsf{f}\ \langle y_1, y_2 \rangle\ \overline{x} & = & M_{\mathsf{pair}} \end{array}$$

ただし $M_{\mathsf{pair}}$ は

$$\Gamma, y_1 : A_1, y_2 : A_2 \rhd M_{\mathsf{pair}} : D_0$$

を満たすものとする.

すなわち定義式 $M_{\mathsf{pair}}$ の中では $x_1, \ldots, x_n$ に加えて変数 $y_1, y_2$ を用いてもよい.

構成子 pair を用いれば, 2 つのデータの対（つい）を作ることができる. たとえば

$$\langle \mathsf{3}, \mathsf{5} \rangle : \mathbf{N} \times \mathbf{N}, \qquad \langle \mathsf{4}, \mathsf{true} \rangle : \mathbf{N} \times \mathbf{B}$$

などである. 対から一方を取り出すには, 原始再帰法により得られるプログラム

$$\begin{array}{lcl} \mathsf{pr}_i & : & A_1 \times A_2 \Rightarrow A_i \\ \mathsf{pr}_i\ \langle y_1, y_2 \rangle & = & y_i \end{array}$$

を用いればよい（$i = 1, 2$）.

直積型を用いた具体例を挙げておく. 関数

$$\begin{array}{lcl} \mathit{fib}(0) & := & 1 \\ \mathit{fib}(1) & := & 1 \\ \mathit{fib}(y+2) & := & \mathit{fib}(y+1) + \mathit{fib}(y) \end{array}$$

により生成される数列 $1, 1, 2, 3, 5, 8, \ldots$ を**フィボナッチ数列**という. 上の定義は一見原始再帰法のスタイルにのっとっているように見えるが, $\mathit{fib}(y+2)$ を定義するのに 2 つ前の値 $\mathit{fib}(y)$ を参照しているため, そのままでは原始再帰的プログラムで表すのが難しい. こんなときに役立つのが直積である.

自然数の対 $\langle m, n \rangle \in \mathbb{N} \times \mathbb{N}$ に対して操作 $\langle m, n \rangle \mapsto \langle n, m+n \rangle$ を考える. $\langle 0, 1 \rangle$ から出発してこの操作を繰り返すと

$$\langle 0,1 \rangle \ \mapsto\ \langle 1,1 \rangle \ \mapsto\ \langle 1,2 \rangle \ \mapsto\ \langle 2,3 \rangle \ \mapsto\ \langle 3,5 \rangle \ \mapsto\ \cdots$$

となるので $n$ 回繰り返して第 2 成分をとれば $\mathit{fib}(n)$ が得られる. このアイデアに基づいて原始再帰的プログラムを書くと次のようになる.

$y$ 番目のフィボナッチ数を出力するプログラム fib

$$\begin{array}{lcl} \mathsf{add}' & : & \mathbf{N} \times \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{add}'\ \langle y_1, y_2 \rangle & = & \mathsf{add}\ y_1\ y_2 \\ & & \\ \mathsf{fib}' & : & \mathbf{N} \Rightarrow \mathbf{N} \times \mathbf{N} \\ \mathsf{fib}'\ 0 & = & \langle 0, 1 \rangle \\ \mathsf{fib}'\ (\mathsf{s}\ y) & = & \langle \mathsf{pr}_2(\mathsf{fib}'\ y),\ \mathsf{add}'(\mathsf{fib}'\ y) \rangle \\ & & \\ \mathsf{fib} & : & \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{fib}\ y & = & \mathsf{pr}_2(\mathsf{fib}'\ y) \end{array}$$

ここで $\mathsf{add}'$ は $\mathsf{add} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ を逆カリー化したものであり, $\mathsf{fib}'$ が操作 $\langle m, n \rangle \mapsto \langle n, m+n \rangle$ の繰り返しを実現している. 最後に第 2 成分をとるのが fib である. このプログラムにより関数 $\mathit{fib} : \mathbb{N} \to \mathbb{N}$ が計算できる.

**■リスト型**　$A$ を型とするとき, 型 $A$ の**リスト型** $\mathbf{L}[A]$ を 2 つの構成子により定める.

$$\mathsf{nil} : \mathbf{L}[A], \qquad \mathsf{cons} : A \Rightarrow \mathbf{L}[A] \Rightarrow \mathbf{L}[A].$$

たとえば以下の項は全部型 $\mathbf{L}[\mathbf{N}]$ を持つ.

$$\mathsf{nil}, \qquad \mathsf{cons}\ 5\ \mathsf{nil}, \qquad \mathsf{cons}\ 3\ (\mathsf{cons}\ 5\ \mathsf{nil}), \qquad \mathsf{cons}\ 8\ (\mathsf{cons}\ 3\ (\mathsf{cons}\ 5\ \mathsf{nil})).$$

これらを省略して以下のようにも書く.

$$[], \qquad [5], \qquad [3,5], \qquad [8,3,5].$$

このように, 自然数の有限列は型 $\mathbf{L}[\mathbf{N}]$ のデータとして表現できる.

念のため $\mathsf{cons}\ 3\ (\mathsf{cons}\ 5\ \mathsf{nil})$ が型 $\mathbf{L}[\mathbf{N}]$ を持つことを確かめておこう.

$$\dfrac{\dfrac{\mathsf{cons} : \mathbf{N} \Rightarrow \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{L}[\mathbf{N}] \quad 3 : \mathbf{N}}{\mathsf{cons}\ 3 : \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{L}[\mathbf{N}]} \qquad \dfrac{\dfrac{\mathsf{cons} : \mathbf{N} \Rightarrow \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{L}[\mathbf{N}] \quad 5 : \mathbf{N}}{\mathsf{cons}\ 5 : \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{L}[\mathbf{N}]} \qquad \mathsf{nil} : \mathbf{L}[\mathbf{N}]}{(\mathsf{cons}\ 5\ \mathsf{nil}) : \mathbf{L}[\mathbf{N}]}}{\mathsf{cons}\ 3\ (\mathsf{cons}\ 5\ \mathsf{nil}) : \mathbf{L}[\mathbf{N}]}$$

リスト型の原始再帰法は次の通り.

原始再帰法（リスト型）

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{L}[A] \Rightarrow \overline{D} \Rightarrow D_0 \\ \mathsf{f}\ \mathsf{nil}\ \overline{x} & = & M_{\mathsf{nil}} \\ \mathsf{f}\ (\mathsf{cons}\ a\ y)\ \overline{x} & = & M_{\mathsf{cons}} \end{array}$$

ただし $M_{\mathsf{nil}}, M_{\mathsf{cons}}$ は

$$\Gamma \rhd M_{\mathsf{nil}} : D_0, \qquad \Gamma, a : A,\ y : \mathbf{L}[A] \rhd M_{\mathsf{cons}} : D_0$$

を満たすものとする. また $M_{\mathsf{cons}}$ の中では $(\mathsf{f}\ y\ \overline{x})$ が部分式として用いられていてもよい.

つまり定義式 $M_{\mathsf{cons}}$ の中では $x_1, \ldots, x_n$ に加えて変数 $a, y$ や部分式 $(\mathsf{f}\ y\ \overline{x})$ を用いてもよい.

リスト型を用いれば, プログラミングの幅が大きく広がる.

リスト型を用いたプログラムの例

$$
\begin{array}{lcl}
\mathsf{append} & : & \mathbf{L}[A] \Rightarrow \mathbf{L}[A] \Rightarrow \mathbf{L}[A] \\
\mathsf{append\ nil}\ x & = & x \\
\mathsf{append}\ (\mathsf{cons}\ a\ y)\ x & = & \mathsf{cons}\ a\ (\mathsf{append}\ y\ x) \\
 & & \\
\mathsf{filter0} & : & \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{L}[\mathbf{N}] \\
\mathsf{filter0\ nil} & = & \mathsf{nil} \\
\mathsf{filter0}\ (\mathsf{cons}\ a\ y) & = & \mathsf{if}\ (\mathsf{zero?}\ a)\ \mathsf{then}\ (\mathsf{filter0}\ y)\ \mathsf{else}\ (\mathsf{cons}\ a\ (\mathsf{filter0}\ y)) \\
 & & \\
\mathsf{insert} & : & \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{N} \Rightarrow \mathbf{L}[\mathbf{N}] \\
\mathsf{insert\ nil}\ b & = & \mathsf{cons}\ b\ \mathsf{nil} \\
\mathsf{insert}\ (\mathsf{cons}\ a\ y)\ b & = & \mathsf{if}\ (\mathsf{leq}\ b\ a)\ \mathsf{then}\ (\mathsf{cons}\ b\ (\mathsf{cons}\ a\ y))\ \mathsf{else}\ (\mathsf{cons}\ a\ (\mathsf{insert}\ y\ b)) \\
 & & \\
\mathsf{sort} & : & \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{L}[\mathbf{N}] \\
\mathsf{sort\ nil} & = & \mathsf{nil} \\
\mathsf{sort}\ (\mathsf{cons}\ a\ y) & = & \mathsf{insert}\ (\mathsf{sort}\ y)\ a
\end{array}
$$

append は 2 つのリストを結合するプログラムである. たとえば

$$
\begin{array}{rcl}
\mathsf{append}\ [1\ 2]\ [3\ 4] & = & \mathsf{append}\ (\mathsf{cons}\ 1\ (\mathsf{cons}\ 2\ \mathsf{nil}))\ [3\ 4] \\
 & = & \mathsf{cons}\ 1\ (\mathsf{append}\ (\mathsf{cons}\ 2\ \mathsf{nil})\ [3\ 4]) \\
 & = & \mathsf{cons}\ 1\ (\mathsf{cons}\ 2\ (\mathsf{append\ nil}\ [3\ 4])) \\
 & = & \mathsf{cons}\ 1\ (\mathsf{cons}\ 2\ [3\ 4]) \\
 & = & [1\ 2\ 3\ 4].
\end{array}
$$

filter0 は与えられたリストから 0 を取り除くプログラムである. たとえば

$$\mathsf{filter0}\ [8\ 0\ 4\ 0]\ =\ [8\ 4]$$

となる（確かめよ）.

insert は, リスト $l$ と自然数 $a$ が与えられたとき, $l$ を左から順に見ていき $a \leq b$ を満たす要素 $b$ が見つかったらその前に $a$ を挿入するプログラムである（そのような $b$ がない場合は $a$ をリストの最後に追加する）. たとえば

$$\mathsf{insert}\ [1\ 2\ 4]\ 3\ =\ [1\ 2\ 3\ 4]$$

となる（確かめよ）.

最後に sort は, 与えられたリストを昇順に並び替えるプログラムである. たとえば

$$\mathsf{sort}[3\ 2\ 4\ 1] =\ [1\ 2\ 3\ 4]$$

となる（確かめよ）. このアルゴリズムは挿入の繰り返しによりソーティングを行うため, 一般に**挿入ソート**と呼ばれる.

**■原始再帰的プログラムの拡張**　$\mathbf{N}$, $\mathbf{B}$ に加えて直積型 $\mathbf{N} \times \mathbf{N}$ やリスト型 $\mathbf{L}[\mathbf{N}]$ もデータ型と見なす. すると, プログラムを書く際に $\mathbf{N} \times \mathbf{N}$ や $\mathbf{L}[\mathbf{N}]$ についての原始再帰法も用いることができる. そうやって書くことができるプログラムのことを一時的に**拡張原始再帰的プログラム**と呼ぶことにしよう.

ここでは証明しないが, 拡張原始再帰的プログラムも常に停止性を満たす. それゆえ集合論的な関数として解釈することができる. まず, 直積型とリスト型に次のような集合を割り当てる.

$$[\![A \times B]\!] := [\![A]\!] \times [\![B]\!], \qquad [\![\mathbf{L}[A]]\!] := [\![A]\!]^*.$$

たとえば $[\![\mathbf{N} \times \mathbf{N}]\!] = \mathbb{N} \times \mathbb{N}$ であり, $[\![\mathbf{L}[\mathbf{N}]]\!] = [\![\mathbf{N}]\!]^* = \mathbb{N}^*$ である.

拡張原始再帰的プログラム $\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ に対しては, 関数 $[\![\mathsf{f}]\!] : [\![D_1]\!] \times \cdots \times [\![D_n]\!] \to [\![D_0]\!]$ を前と同様に割り当てる. つまり任意の $v_1 \in [\![D_1]\!], \cdots, v_n \in [\![D_n]\!]$, $w \in [\![D_0]\!]$ について

$$[\![\mathsf{f}]\!](v_1, \ldots, v_n) = w \quad \Longleftrightarrow \quad \mathsf{f}\ \mathsf{v}_1 \cdots \mathsf{v}_n = \cdots = \mathsf{w}$$

を満たすように $[\![\mathsf{f}]\!]$ を定める. このようにして得られる関数を**拡張原始再帰的関数**と呼ぶことにしよう.

こうして原始再帰的関数の守備範囲を直積やリストへと拡大したわけだが, それでも自然数やブール値上に話を制限すれば, そこに新たな関数が付け加わるわけではない.

**命題 2.7**

関数 $f : \mathbb{N} \to \mathbb{N}$ が原始再帰的関数であることと, 拡張原始再帰的関数であることは一致する.

たとえばフィボナッチ数を返す関数 $\mathit{fib} : \mathbb{N} \to \mathbb{N}$ が拡張原始再帰的であることはすでにわかっているが, これは原始再帰的でもある. つまり直積を用いずともプログラム可能である. なぜなら同数性

$$\mathbb{N} \times \mathbb{N} \simeq \mathbb{N}$$

が成り立ち, しかも補足 1.13 で言及した全単射

$$f(m,n) \ := \ \frac{(m+n)(m+n+1)}{2} + m \quad : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

(およびその逆) が原始再帰的だからである. それゆえ直積型 $\mathbf{N} \times \mathbf{N}$ の代わりに $\mathbf{N}$ を用いて, 構成子 $\mathsf{pair} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \times \mathbf{N}$ の代わりに上の $f$ を計算するプログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ を用いればよい. ただし原始再帰法 (直積型) の翻訳は少々複雑なので, ここでは割愛する.

上の命題があるので, 直積やリストを用いたからといってアッカーマン関数がプログラム可能になるわけではない. それゆえ自然数上で原始再帰的関数よりも多くの関数を表現するには, 単にデータ型を加えるのみでなく, プログラムの構成法を実質的に拡張しなければならない.

# 3　再帰的関数

原始再帰法によるプログラム定義の例

$$\begin{array}{lcl} \mathsf{add} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{add}\ 0\ x & = & x \\ \mathsf{add}\ (\mathsf{s}\ y)\ x & = & \mathsf{s}\ (\mathsf{add}\ y\ x) \end{array}$$

をもう一度見てみよう．3 行目では関数 add を定義するのに右辺で add そのものを用いているが, 第一引数の値が減少しているので, 計算は循環せず必ず停止する．同じことがすべての原始再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ について言える（定理 2.4）．プログラム構成法が原始再帰法という絶対安全なもののみに制限されているので, どんなプログラムも停止性を満たすのである．

しかしこのようなアプローチでは, すべての「計算可能な」関数を捉えることはできない．それが前章の結論であった．より一般的な関数を得るには, この制限をいったん取り払う必要がある．すなわち, すべての入力値について計算が停止するとは限らないような状況をあえて考えるのである．

以下では一般再帰法の考え方を説明し, （全域）再帰的関数のクラスを定義する．これが「計算可能な」関数全体を定めるという主張があり, 一般に広く受け入れられている（チャーチの提唱）．その後, 関数のみでなく集合についても「計算可能」であることをきちんと定め, 「計算不能」（決定不能）な集合が存在することを示す．

## 3.1　一般再帰法

次のようなプログラム構成法を**再帰法**, または**一般再帰法**という．

一般再帰法による定義

$$\begin{array}{lcl} \mathsf{f} & : & D_1 \Rightarrow \cdots D_n \Rightarrow D_0 \\ \mathsf{f}\ x_1\ \cdots\ x_n & = & M_{\mathsf{f}} \end{array}$$

ただし $M_{\mathsf{f}}$ は

$$x_1 : D_1,\ \ldots,\ x_n : D_n \rhd M_{\mathsf{f}} : D_0$$

を満たすものとする．なお, $M_{\mathsf{f}}$ **の中で識別子 f が用いられていても構わない**．$M_{\mathsf{f}}$ を f **の定義式**という．

再帰法の具体例として最大公約数を計算するプログラムを考える．

最大公約数を計算するプログラム

$$
\begin{array}{lcl}
\mathsf{gcd} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{gcd}\ x\ y & = & \mathsf{if}\ x = y\ \mathsf{then}\ x\ \mathsf{else} \\
 & & \quad \mathsf{if}\ x < y\ \mathsf{then}\ (\mathsf{gcd}\ y\ x)\ \mathsf{else}\ (\mathsf{gcd}\ (x - y)\ y)
\end{array}
$$

ただし見やすくするために以下の記法を用いている：

$$
\begin{array}{lcl}
x = y & := & \mathsf{eq}\ x\ y \\
x < y & := & \mathsf{and}\ (\mathsf{leq}\ x\ y)\ (\mathsf{not}(\mathsf{eq}\ x\ y)) \\
x - y & := & \mathsf{sub}\ y\ x
\end{array}
$$

この例では gcd $x\ y$ の値を gcd $y\ x$ と gcd $(x - y)\ y$ を用いて定義している（関数の**再帰呼び出し**）. これは原始再帰法とは異なるが, gcd が呼び出される度に「第一引数 + 第二引数 $\times 2$」が減少するので, 計算は必ず停止することがわかる（ただし $x = 0$ または $y = 0$ の場合を除く）.

一方で次も再帰法の一例である.

$$
\begin{array}{lcl}
\mathsf{diverge} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{diverge}\ x & = & \mathsf{diverge}\ (\mathsf{s}\ x)
\end{array}
$$

だがこのプログラムを実行すると

$$\mathsf{diverge}\ 0\ =\ \mathsf{diverge}\ 1\ =\ \mathsf{diverge}\ 2\ =\ \cdots$$

となり計算は停止しない.

最後にもう 1 つ例を挙げる. 「どんな自然数 $n \geq 1$ から出発しても, 偶数なら $n/2$ でおきかえ, 奇数なら $3n + 1$ でおきかえていけば, いつかは必ず 1 に到達する」という予想がある（**コラッツの予想**または**角谷の予想**）. たとえば

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1.$$

そこで次のプログラムを考える（適当な省略表現を用いる）.

$3n + 1$ 関数を計算するプログラム

$$
\begin{array}{lcl}
\mathsf{half} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{half}\ x & = & \mathsf{if}\ (\mathsf{zero}\ x)\ \mathsf{then}\ 0\ \mathsf{else}\ \mathsf{s}\ (\mathsf{half}\ (x - 2)) \\
 & & \\
\mathsf{even} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{even}\ x & = & \mathsf{if}\ x = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else} \\
 & & \quad \mathsf{if}\ x = 1\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ (\mathsf{even}\ (x - 2)) \\
 & & \\
\mathsf{colatz} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{colatz}\ x & = & \mathsf{if}\ (x = 0\ \mathsf{or}\ x = 1)\ \mathsf{then}\ \mathsf{true}\ \mathsf{else} \\
 & & \quad \mathsf{if}\ (\mathsf{even}\ x)\ \mathsf{then}\ \mathsf{colatz}\ (\mathsf{half}\ x)\ \mathsf{else}\ \mathsf{colatz}\ (3 \cdot x + 1)
\end{array}
$$

このプログラムは少なくとも $2^{60}$ までの入力値について停止することがわかっているが, 全ての入力値について停止するかどうかは未解決である.

このように再帰法は原始再帰法よりもフレキシブルだが, 計算が停止しない危険性がある. 計算の停止を保証するには別途論証を与えなければならない.

**■再帰的プログラム**　ここで再帰的プログラムの正確な定義を与えておきたい. 話をなるべく簡単にするために, 次の 2 点に注意する.

まず, 既出の「合成による定義」は再帰法の特別な場合にすぎない. すなわち, 識別子 f が定義式の中に出てこない場合である. ゆえに「合成による定義」はないものと思ってよい.

次に, 原始再帰法は,

$$\mathsf{if} : \mathbf{B} \Rightarrow D \Rightarrow D \Rightarrow D, \qquad \mathsf{zero?} : \mathbf{N} \Rightarrow \mathbf{B}, \qquad \mathsf{pred} : \mathbf{N} \Rightarrow \mathbf{N}$$

さえあれば, 再帰法によりシミュレートできる. 実際, 原始再帰法による定義

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{N} \Rightarrow \overline{D} \Rightarrow D_0 \\ \mathsf{f}\ 0\ \overline{x} & = & M_0 \\ \mathsf{f}\ (\mathsf{s}\ y)\ \overline{x} & = & M_{\mathsf{s}}[\cdots(\mathsf{f}\ y\ \overline{x})\cdots] \end{array}$$

は

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{N} \Rightarrow \overline{D} \Rightarrow D_0 \\ \mathsf{f}\ y\ \overline{x} & = & \mathsf{if}\ (\mathsf{zero?}\ y)\ \mathsf{then}\ M_0\ \mathsf{else}\ M_{\mathsf{s}}[\cdots(\mathsf{f}\ (\mathsf{pred}\ y)\ \overline{x})\cdots] \end{array}$$

としても同じことだからである.

以上の準備のもとで, 正確な定義を与える. **再帰的プログラム** $P$ とは, 互いに異なる識別子 $\mathsf{f}_1, \ldots, \mathsf{f}_n$ の定義の列のことである. ただし各 $\mathsf{f}_k$（$1 \leq k \leq n$）は再帰法により定義されており, 定義式 $M_{\mathsf{f}_1}, \ldots, M_{\mathsf{f}_n}$ の中では if, zero?, pred と $\mathsf{f}_1, \ldots, \mathsf{f}_n$ 以外の識別子を用いてはならない. 型情報を省略すれば, 再帰的プログラム $P$ は

$$\mathsf{f}_1\ \overline{x}_1 = M_{\mathsf{f}_1}, \quad \mathsf{f}_2\ \overline{x}_2 = M_{\mathsf{f}_2}, \quad \cdots, \quad \mathsf{f}_n\ \overline{x}_n = M_{\mathsf{f}_n}$$

という形をしている. 最後の識別子 $\mathsf{f} = \mathsf{f}_n$ を $P$ の**主たる識別子**と呼ぶ. これをプログラム全体の名前だと思って, $P$ のことを**プログラム** f とも呼ぶ.

**■再帰的プログラムの実行法**　次に, プログラムの実行手順について考える. プログラムを実行するには識別子を定義式で書き換えてゆけばよいのだが, どんな順序で書き換えるかによって結果は変わってくる. 実際, ある順序によれば計算は停止するのだが, 別の順序では停止しないといった事態が起こりうる（とはいえ停止する場合出力値はただ 1 つである）. たとえば再帰的プログラム

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{f}\ y & = & \mathsf{if\ false\ then\ diverge}\ y\ \mathsf{else}\ y \end{array}$$

について考えると, if の第一引数が false なので自然に書き換えれば

$$\begin{array}{lcl} \mathsf{f}\ 0 & = & \mathsf{if\ false\ then\ diverge}\ 0\ \mathsf{else}\ 0 \\ & = & 0 \end{array}$$

となり計算が停止する. しかし, then 節の中身を書き換えることに固執すれば

$$\begin{aligned} \mathsf{f}\ 0 &= \text{if false then diverge 0 else 0} \\ &= \text{if false then diverge 1 else 0} \\ &= \text{if false then diverge 2 else 0} \\ &= \cdots \end{aligned}$$

となり計算は停止しない.

書き換え順序（**評価戦略**）には, **名前呼び**（call-by-name）, **値呼び**（call-by-value）, **必要呼び**（call-by-need）などがあるが, ここでは値呼び戦略を採用することにする.

いまデータ型は $\mathbf{B}$, $\mathbf{N}$ のみとする. 以下の式を**値式**または単に**値**というのであった.

$$\mathsf{true} : \mathbf{B}, \qquad \mathsf{false} : \mathbf{B}, \qquad \mathsf{n} : \mathbf{N} \quad (n \in \mathbb{N})$$

値式を記号 $\mathsf{v}, \mathsf{w}$ 等により表す.

次の再帰的プログラム $P$ を考える（型情報は省略）.

$$\mathsf{f}_1\ \overline{x}_1 = M_{\mathsf{f}_1}, \quad \mathsf{f}_2\ \overline{x}_2 = M_{\mathsf{f}_2}, \quad \cdots, \quad \mathsf{f}_n\ \overline{x}_n = M_{\mathsf{f}_n}.$$

このプログラムに対して, 次の書き換え規則 $\longrightarrow_P$ を与える.

$$\begin{array}{lll} \mathsf{f}_i\ \mathsf{v}_1\ \cdots\ \mathsf{v}_k & \longrightarrow_P & M_{\mathsf{f}_i}[x_1 := \mathsf{v}_1, \ldots, x_k := \mathsf{v}_k] \\ \mathsf{if\ true}\ M\ N & \longrightarrow_P & M \\ \mathsf{if\ false}\ M\ N & \longrightarrow_P & N \\ \mathsf{zero?}\ 0 & \longrightarrow_P & \mathsf{true} \\ \mathsf{zero?}\ \mathsf{s}(\mathsf{n}) & \longrightarrow_P & \mathsf{false} \\ \mathsf{pred}\ 0 & \longrightarrow_P & 0 \\ \mathsf{pred}\ \mathsf{s}(\mathsf{n}) & \longrightarrow_P & \mathsf{n} \end{array}$$

ただし $1 \leq i \leq n$ であり, $[x_1 := \mathsf{v}_1, \ldots, x_k := \mathsf{v}_k]$ は「変数 $x_1, \ldots, x_k$ に値式 $\mathsf{v}_1, \ldots, \mathsf{v}_k$ を代入する」操作を表す.

書き換えが行われるのは, 引数が値式のときに限ることに注意（それゆえ「値呼び」なのである）. もっとも if の第二, 第三引数は例外であり, $M, N$ は値式である必要はない. 書き換えは, 式の内部で自由に行ってよい. ただし if の第二, 第三引数を書き換えてはならない. いわば $M, N$ は第一引数により “ロック” された状態にある.

さて, プログラム f と値式の列 $\overline{\mathsf{v}} = \mathsf{v}_1, \ldots, \mathsf{v}_k$ を考える. 式 $\mathsf{f}\ \overline{\mathsf{v}}$ に何度か書き換えを行うことで値式 w に到達するとき,

$$\mathsf{f}\ \overline{\mathsf{v}} = \mathsf{w}, \qquad \mathsf{f}\ \overline{\mathsf{v}} \Downarrow \mathsf{w}, \qquad \mathsf{f}\ \overline{\mathsf{v}} \Downarrow$$

等と書く. 最後の表現は出力 w が重要でないときに用いる記法である. また $\mathsf{f}\ \overline{\mathsf{v}}$ に何度書き換えを行っても値式に到達しないとき,

$$\mathsf{f}\ \overline{\mathsf{v}} \Uparrow$$

と書く. （適切な型をもつ）どのような値式の列 $\overline{\mathsf{v}}$ についても $\mathsf{f}\ \overline{\mathsf{v}} \Downarrow$ となるとき, プログラム f は**全域再帰的**であるという. 「全域」というのは, すべての入力値に対して出力値が定まっているとい

う意味合いである．これは「停止性を満たす」というのと同じことであるが, 以下で定義する「全域再帰的関数」と用語を合わせておく．

どんな原始再帰的プログラムも全域再帰的である（定理 2.4）．diverge は再帰的プログラムであるが, 全域再帰的プログラムではない．colatz が全域再帰的プログラムかどうかは未解決問題である．

**■再帰的関数**　次に再帰的プログラムに集合論的解釈を与える．まず特別な要素 $\perp$ を用意し, 集合 $A$ が与えられたとき, $A_\perp := A \cup \{\perp\}$ と定める（ただし $\perp \notin A$）．直感的にいって $\perp$ は「未定義」を表す．そして関数 $f : A \to B_\perp$ は $A$ から $B$ への**部分関数**を表すものと考える．すなわち $a \in A$ について $f(a) = \perp$ のときには, $f(a)$ の値は「未定義」と考えるのである．

再帰的プログラム $\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ に対して部分関数 $[\![\mathsf{f}]\!] : [\![D_1]\!] \times \cdots \times [\![D_n]\!] \to [\![D_0]\!]_\perp$ を次のように定める．

$$\begin{array}{rcll} [\![\mathsf{f}]\!](v_1, \cdots, v_n) & := & w & (\mathsf{f}\ \mathsf{v}_1\ \cdots\ \mathsf{v}_n = \mathsf{w}\ \text{のとき}) \\ & := & \perp & (\mathsf{f}\ \mathsf{v}_1\ \cdots\ \mathsf{v}_n \Uparrow \text{のとき}) \end{array}$$

たとえばプログラム $\mathsf{gcd} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ の場合 $[\![\mathsf{gcd}]\!] : \mathbb{N} \times \mathbb{N} \to \mathbb{N}_\perp$ は次のようになる．

$$\begin{array}{rcll} [\![\mathsf{gcd}]\!](m, n) & = & k & (k\ \text{は}\ m, n\ \text{の最大公約数}) \\ [\![\mathsf{gcd}]\!](m, n) & = & \perp & (m = 0\ \text{または}\ n = 0\ \text{のとき}) \end{array}$$

f を再帰的プログラムとするとき, 部分関数 $f = [\![\mathsf{f}]\!]$ を**再帰的関数**という．とくに f が全域再帰的プログラムのときには, 決して $f(\bar{a}) = \perp$ とならない．つまり $f : [\![D_1]\!] \times [\![D_n]\!] \to [\![D_0]\!]$ となる．このとき $f$ を**全域再帰的関数**という．

**■原始再帰的関数と全域再帰的関数**　2.6 節で挙げたアッカーマン関数 $Ack : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ に戻ろう．$Ack(x, y) := ack_x(y)$ であり, $ack_x$ は 2.6 節冒頭の原始再帰的プログラムにより定義されたのであった．これらをまとめれば, 関数 $Ack$ は直接定義することができる．

$$\begin{array}{rcl} Ack(0, y) & = & y + 1 \\ Ack(x + 1, 0) & = & Ack(x, 1) \\ Ack(x + 1, y + 1) & = & Ack(x, Ack(x + 1, y)) \end{array}$$

この定義を再帰的プログラムとして書き起こすと次が得られる．

$$\begin{array}{lcl} \mathsf{Ack} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{Ack}\ x\ y & = & \mathsf{if}\ (\mathsf{zero?}\ x)\ \mathsf{then}\ (\mathsf{s}\ y)\ \mathsf{else} \\ & & \quad \mathsf{if}\ (\mathsf{zero?}\ y)\ \mathsf{then}\ (\mathsf{Ack}\ (\mathsf{pred}\ x)\ 1)\ \mathsf{else}\ (\mathsf{Ack}\ (\mathsf{pred}\ x)\ (\mathsf{Ack}\ x\ (\mathsf{pred}\ y))) \end{array}$$

**命題 3.1**

プログラム Ack は全域再帰的である。したがって $Ack$ は全域再帰的関数である。

**証明：**　任意の $(m, n) \in \mathbb{N}^2$ について Ack m n $\Downarrow$ となることを $(m, n)$ に関する二重帰納法で証明する。

- $m = 0$ のときは明らか。

- $m \neq 0, n = 0$ のとき、Ack m 0 $= \cdots =$ Ack m$-$1 1 である。第 1 引数 $m$ についての帰納法の仮定より後者の計算は停止する。よって Ack m 0 $\Downarrow$.
- $m \neq 0, n \neq 0$ のとき、

  $$\mathsf{Ack\ m\ n} = \cdots = \mathsf{Ack\ (pred\ m)\ (Ack\ m\ (pred\ n))} = \cdots = \mathsf{Ack\ m{-}1\ (Ack\ m\ n{-}1)}$$

  である。第 2 引数 $n$ についての帰納法の仮定により Ack m n$-$1 $\Downarrow$ k. 第 1 引数 $m$ についての帰納法の仮定により Ack m$-$1 k $\Downarrow$ （一般に $k > n$ となるが、$m$ の値が減っているので、帰納法の仮定を適用できる）. よって Ack m n $\Downarrow$ が成り立つ。

□

よって原始再帰的ではないが全域再帰的な関数が存在する。

**定理 3.2**

（原始再帰的関数全体の集合） $\subsetneq$ （全域再帰的関数全体の集合）
$\subsetneq$ （再帰的関数全体の集合）

## 3.2 再帰的関数の自己解釈

再帰的プログラムはある種の自己言及性を持つ. それは, 再帰的プログラムの実行方法それ自体が再帰的プログラムで記述できるという点である. 本節では, このような再帰的プログラムの自己解釈について簡単に説明し, 重要な標準形定理を紹介する.

式 $M$ とは一定の記号からなる列であり, 各記号は 1 つの自然数で表せる. それゆえ $M$ は $\mathbb{N}^*$ の要素として表せる. よって同数性

$$\mathbb{N} \quad \simeq \quad \mathbb{N}^*$$

（命題 1.12）を用いれば, どんな式 $M$ も 1 つの自然数で符号化できることになる. 以下ではそのような（都合のよい）符号化の方法を 1 つ固定する. その上で式 $M$ を表す自然数のことを $M$ の**ゲーデル数**といい, $\ulcorner M \urcorner \in \mathbb{N}$ と書く. また自然数 $m = \ulcorner M \urcorner$ を表す値式 m のことを $\mathbb{\ulcorner} M \mathbb{\urcorner} : \mathbf{N}$ と書く. 同様にすれば, プログラム $P$（再帰的定義の有限列）に対してもゲーデル数 $\ulcorner P \urcorner \in \mathbb{N}$ や値式 $\mathbb{\ulcorner}\mathsf{f}\mathbb{\urcorner} : \mathbf{N}$ を定めることができる.

**補足 3.3** 具体的には, たとえば表

| 0 | s | true | false | ( | ) | $\mathsf{f}_0$ | $x_0$ | $\mathsf{f}_1$ | $x_1$ | $\cdots$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | $\cdots$ |

を用いる. 識別子（プログラムの名前）としては $\mathsf{f}_0, \mathsf{f}_1, \mathsf{f}_2, \ldots$ しか用いないことにし, 変数としては $x_0, x_1, x_2, \ldots$ しか用いないことにすれば, 必要な記号はすべて上の表の中に含まれている. それゆえどんな式も自然数の列で表せる. たとえば値式 2（つまり s (s 0)）について考えてみると,

$$\mathsf{s\ (s\ 0)} \ \mapsto\ (2, 5, 2, 1, 6) \ \in \mathbb{N}^*$$

となる. 次に素数を小さい順に並べて $2, 3, 5, 7, 11, \cdots$ とし, 肩にそれぞれの自然数をのせて掛け合わせれば,

$$\ulcorner 2 \urcorner \ = \ 2^2 3^5 7^2 11^1 13^6 \ = \ 2528803849572 \in \mathbb{N}$$

が得られる．これが 2 のゲーデル数であり, この数に相当する値式が $\ulcorner 2 \urcorner$ である．つまり

$$\ulcorner 2 \urcorner \;=\; 2528803849572 \;=\; \underbrace{\mathsf{s}\,(\mathsf{s}\,(\cdots(\mathsf{s}}_{2528803849572}\;\;0)\cdots)) \;:\mathbf{N}.$$

$2 = \mathsf{s}\ (\mathsf{s}\ 0)$ とは似ても似つかない巨大な式であるが, それでも素数・べき乗・掛け算の組み合わせで得られる値である．よって原始再帰的プログラム $\mathsf{nat2gn} : \mathbf{N} \Rightarrow \mathbf{N}$ をうまく定めれば,

$$\mathsf{nat2gn}\ \mathsf{n} \;=\; \ulcorner \mathsf{n} \urcorner$$

がすべての $n$ について成り立つようにすることができる．

なおプログラム（再帰的定義の有限列）を自然にコード化するには, 上の表を適当に拡充する必要がある．たとえば等号 $=$ や空白文字を表す数も導入する必要があるだろう．型情報も加えるなら,「:」や「⇒」を表す数も必要である．

さて, プログラム $P$ を定めれば, 式の書き換え規則 $\longrightarrow_P$ が定まる．この規則による 1 ステップの書き換えは簡単な記号操作だから, 次を満たす原始再帰的プログラム $\mathsf{step} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ が存在するはずである（実際の構成は煩雑なので省略する）．

$$\mathsf{step}\ \ulcorner P \urcorner\ \ulcorner M \urcorner = \ulcorner N \urcorner \quad\Longleftrightarrow\quad M \longrightarrow_P N.$$

また, 与えられた式（のゲーデル数）が値式かどうかを判定する原始再帰的プログラム $\mathsf{value} : \mathbf{N} \Rightarrow \mathbf{B}$ と,（型 $\mathbf{N}$ の）値式のゲーデル数 $\ulcorner \mathsf{n} \urcorner$ から値式 n 自体を復元する原始再帰的プログラム $\mathsf{gn2nat} : \mathbf{N} \Rightarrow \mathbf{N}$ が自然に構成できる．

$$\begin{array}{l} \mathsf{value}\ \ulcorner M \urcorner = \mathsf{true} \quad\Longleftrightarrow\quad M \text{ は値式} \\ \mathsf{gn2nat}\ \ulcorner \mathsf{n} \urcorner = \mathsf{n} \end{array}$$

あとひとつ, プログラム $P$ の主たる識別子を f とする．このとき $P$ のゲーデル数と値式 m が与えられたら, 式 f m のゲーデル数を返す原始再帰的プログラム $\mathsf{init} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ が存在する．

$$\mathsf{init}\ \ulcorner P \urcorner\ \mathsf{m} = \ulcorner \mathsf{f}\ \mathsf{m} \urcorner$$

これらのプログラムを組み合わせれば, 次のような再帰的プログラム eval を定義することができる．

$$\begin{array}{lcl} \mathsf{eval}' & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{eval}'\ x\ y & = & \mathsf{if}\ (\mathsf{value}\ y)\ \mathsf{then}\ (\mathsf{gn2nat}\ y)\ \mathsf{else}\ \mathsf{eval}'\ x\ (\mathsf{step}\ x\ y) \\ & & \\ \mathsf{eval} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{eval}\ x\ z & = & \mathsf{eval}'\ x\ (\mathsf{init}\ x\ z) \end{array}$$

プログラム eval は, どんな再帰的プログラム $P$ の動作もシミュレートできる**万能**再帰的プログラムである．このことを見るために, $M$ が値式ではなく, $M \longrightarrow_P M'$ だと仮定する．このとき

$$\begin{array}{rcl} \mathsf{eval}'\ \ulcorner \mathsf{P} \urcorner\ \ulcorner M \urcorner & = & \mathsf{if}\ (\mathsf{value}\ \ulcorner M \urcorner)\ \mathsf{then}\ \cdots\cdots\ \mathsf{else}\ \cdots \\ & = & \mathsf{if}\ \mathsf{false}\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots \\ & = & \mathsf{eval}'\ \ulcorner P \urcorner\ (\mathsf{step}\ \ulcorner P \urcorner\ \ulcorner M \urcorner) \\ & = & \mathsf{eval}'\ \ulcorner P \urcorner\ \ulcorner M' \urcorner \end{array}$$

となる（見やすくするため適宜省略してある）．一方で n を値式とすると

$$\begin{array}{rcl} \mathsf{eval}'\ulcorner P\urcorner\ulcorner \mathsf{n}\urcorner & = & \mathsf{if}\ (\mathsf{value}\ulcorner \mathsf{n}\urcorner)\ \mathsf{then}\ \cdots\ \mathsf{else}\ \cdots \\ & = & \mathsf{if\ true\ then}\ \cdots \mathsf{else}\ \cdots \\ & = & \mathsf{gn2nat}\ulcorner \mathsf{n}\urcorner \\ & = & \mathsf{n} \end{array}$$

が得られる．したがって, もし入力値 m に対して f m の実行過程が

$$\mathsf{f\ m}\ \longrightarrow_P\ M\ \longrightarrow_P \cdots\ \longrightarrow_P \mathsf{n}$$

のようだったとすると, eval $\ulcorner P\urcorner$ m の計算結果は次のようになる（f は $P$ の主たる識別子）．

$$\begin{array}{rcl} \mathsf{eval}\ulcorner P\urcorner\ \mathsf{m} & = & \mathsf{eval}'\ulcorner P\urcorner\ (\mathsf{init}\ulcorner P\urcorner\ \mathsf{m}) \\ & = & \mathsf{eval}'\ulcorner P\urcorner\ulcorner \mathsf{f\ m}\urcorner \\ & = & \mathsf{eval}'\ulcorner P\urcorner\ulcorner M\urcorner \\ & = & \mathsf{eval}'\ulcorner P\urcorner\ulcorner \mathsf{n}\urcorner \\ & = & \mathsf{n}. \end{array}$$

つまり eval $\ulcorner P\urcorner$ m の出力は f m の出力と一致する．一方で f m の計算が止まらないときは, eval $\ulcorner P\urcorner$ m の計算も止まらない．以上をまとめておこう．

**定理 3.4**(eval の万能性)

eval は型 $\mathbf{N} \Rightarrow \mathbf{N}$ をもつどんな再帰的プログラム $P$ もシミュレートできる．つまり以下の同値性が成り立つ（f は $P$ の主たる識別子）．

$$\begin{array}{lcl} \mathsf{eval}\ulcorner P\urcorner\ \mathsf{m} = \mathsf{n} & \Longleftrightarrow & \mathsf{f\ m} = \mathsf{n} \\ \mathsf{eval}\ulcorner P\urcorner\ \mathsf{m} \Uparrow & \Longleftrightarrow & \mathsf{f\ m} \Uparrow \end{array}$$

同じことを再帰的関数について述べるために,

$$eval := [\![\mathsf{eval}]\!] : \mathbb{N} \to \mathbb{N} \to \mathbb{N}_\perp$$

とおく．

**定理 3.5**(標準形定理)

どんな再帰的関数 $f : \mathbb{N} \to \mathbb{N}_\perp$ についてもある自然数 $e$ が存在し,

$$f(m) = eval(e, m)$$

が全ての $m \in \mathbb{N}$ について成り立つ．

**証明：** $f$ は再帰的関数なので, $f = [\![\mathsf{f}]\!]$ を満たす再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ が存在する．そこで（プログラム f を改めて $P$ と書くことにし）$e := \ulcorner P\urcorner$ とすれば,

$$\begin{array}{lcl} eval(\ulcorner P\urcorner, m) = n & \Longleftrightarrow & \mathsf{eval}\ulcorner P\urcorner\ \mathsf{m} = \mathsf{n} \\ & \Longleftrightarrow & \mathsf{f\ m} = \mathsf{n} \\ & \Longleftrightarrow & f(m) = n. \end{array}$$

$eval(\ulcorner P \urcorner, m) = \bot \iff f(m) = \bot$ も同様に示せる. □

上では型 $\mathbf{N} \Rightarrow \mathbf{N}$（関数集合 $\mathbb{N} \to \mathbb{N}_\bot$）について標準形定理を述べたが, 同様のことは任意の一階型について成り立つ.

この定理の帰結として, どんな再帰的関数もただ 1 回だけしか一般再帰法を用いずにプログラム可能なことがわかる（合成, 原始再帰法は自由に使ってよいものとする）. 実際, step, value, gn2nat, init はいずれも原始再帰的プログラムである. 一般再帰法は $\mathsf{eval}'$ を定義するときにしか用いられていない.

よって原始再帰的関数について以下の特徴づけが得られる.

**定理 3.6**

$f : \mathbb{N} \to \mathbb{N}_\bot$ を再帰的関数とする. $f$ が原始再帰的関数となるための必要十分条件は, ある自然数 $n$ が存在して, $f(m)$ を計算するのに必要なステップ数が $ack_n(m)$ で抑えられることである（$m$ は任意の自然数）.

**証明：** 原始再帰的関数を計算するのに必要なステップ数がある $ack_n$ で抑えられることについては省略する（参考：定理 2.5）.

逆に $f : \mathbb{N} \to \mathbb{N}_\bot$ を再帰的関数とし, $f$ を計算する再帰的プログラムを $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ とする. また $\mathsf{f}\ \mathsf{m}$ の計算ステップ数が $ack_n(m)$ で抑えられると仮定する. このとき eval を改変し, 次の原始再帰的プログラムを構成する.

$$
\begin{array}{lcl}
\mathsf{stepv} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{stepv}\ x\ y & = & \mathsf{if}\ (\mathsf{value}\ y)\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{step}\ x\ y
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{eval}'_{pr} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{eval}'_{pr}\ 0\ x\ y & = & \mathsf{init}\ x\ y \\
\mathsf{eval}'_{pr}\ (\mathsf{s}\ a)\ x\ y & = & \mathsf{stepv}\ x\ (\mathsf{eval}'_{pr}\ a\ x\ y)
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{eval}_{pr} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{eval}_{pr}\ a\ x\ y & = & \mathsf{gn2nat}\ (\mathsf{eval}'_{pr}\ a\ x\ y)
\end{array}
$$

少し考えればわかるように, 十分大きな $k \in \mathbb{N}$ を与えれば $\mathsf{eval}_{pr}\ \mathsf{k}\ x\ z$ の挙動は $\mathsf{eval}\ x\ z$ と一致する. それゆえ

$$
\begin{array}{lcl}
\mathsf{f}_{pr} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{f}_{pr}\ x & = & \mathsf{eval}_{pr}\ (\mathsf{ack}_n\ x)\ \ulcorner \mathsf{f} \urcorner\ x
\end{array}
$$

と置くことで

$$\mathsf{f}_{pr}\ \mathsf{m} \ = \ \mathsf{eval}_{pr}\ (\mathsf{ack}_n\ \mathsf{m})\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m}$$

の計算結果は $\mathsf{eval}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m}$, したがって元の $\mathsf{f}\ \mathsf{m}$ の計算結果と等しくなる. すなわち $[\![\mathsf{f}_{pr}]\!] = [\![\mathsf{f}]\!] = f$. よって $f$ は原始再帰的関数である. □

**■チャーチの提唱** 上のアイデアは, 再帰的プログラム以外にもさまざまな計算モデルに対して適用できる. たとえば実物のコンピュータ上の計算だって基本的には簡単なステップの繰り返しであるから, 上の step のような原始再帰的プログラムを繰り返すことで実現できる. それゆえ再帰的関

数として表すことができる. 人間が行う手計算ですらそうである. 計算の “本質” は, あらかじめ決められた規則に従って基本的なステップを繰り返すことにあるからである. ゆえに, およそ「**計算可能**」な関数なら, 必ず再帰的関数として表現できるだろうと予想できる. そこで

**計算可能であるとは再帰的であることに他ならない**

というテーゼが立てられる. これを**チャーチの提唱**という（文脈によっては「計算可能＝全域再帰的」と考えることもよくある）. 1930 年代に立てられて以降, このテーゼに対する反例は見つかっていない. たとえば量子コンピュータを用いようとも, 新たな関数が「計算可能」になるわけではない（ごく一部の関数の計算が高速化されるだけである）. 長く受け入れられてきたため, 今ではこのテーゼ自体が「計算可能」の “定義” として認められているといっても過言ではない.

## 3.3 再帰的可算集合と決定可能集合

これまでは主に「関数」の話をしてきたが, ここで「集合」の事柄に話題を転じる.

$k \geq 1$ とし, 再帰的プログラム $\mathsf{f} : \underbrace{\mathbf{N} \Rightarrow \cdots \mathbf{N} \Rightarrow}_{k} \mathbf{B}$ が与えられたとする. このとき集合 $R_{\mathsf{f}} \subseteq \mathbb{N}^k$ を

$$(n_1, \ldots, n_k) \in R_{\mathsf{f}} \quad \Longleftrightarrow \quad \mathsf{f}\ \mathsf{n}_1\ \cdots\ \mathsf{n}_k = \mathsf{true}$$

により定める. このような集合を**再帰的可算集合**（または**半決定可能集合**）という.

簡単のため $k = 1$ とし, $R = R_{\mathsf{f}} \subseteq \mathbb{N}$ を再帰的可算集合とする. このとき, もしも $n \in R$ が事実として成り立つならば, $\mathsf{f}\ \mathsf{n}$ の計算は停止して $\mathsf{f}\ \mathsf{n} = \mathsf{true}$ となるはずなので, 有限の時間内で $n \in R$ であることが確かめられる. 一方で $n \notin R$ のときには $\mathsf{f}\ \mathsf{n}$ の計算が停止するとは限らないので, 有限の時間内で $n \notin R$ であると確かめらるとは限らない. このように再帰的可算集合には, 計算の停止・非停止に関して非対称性がある.

一方で $\mathsf{f}$ が全域再帰的な場合, すなわちどんな $n \in \mathbb{N}$ についても $\mathsf{f}\ \mathsf{n} \Downarrow$ となる場合を考えよう. このときには $n \in \mathbb{N}$ が与えられたとき, $n \in R$ か $n \notin R$ かは必ず有限の時間内に決定することができる.

一般に, 集合 $R \subseteq \mathbb{N}^k$ が**決定可能**である（または**再帰的**である）とは, 「全域」再帰的プログラム $\mathsf{f} : \underbrace{\mathbf{N} \Rightarrow \cdots \mathbf{N} \Rightarrow}_{k} \mathbf{B}$ が存在して $R = R_{\mathsf{f}}$ となることをいう. 定義より明らかに

$$\text{(決定可能集合全体)} \quad \subseteq \quad \text{(再帰的可算集合全体)}$$

となる.

たとえば

$$\mathit{Prime} = \{n \in \mathbb{N} \mid n \text{ は素数 }\}$$

とすると, $\mathit{Prime}$ は決定可能である（原始再帰的プログラム $\mathsf{prime} : \mathbf{N} \Rightarrow \mathbf{B}$ があるため）.

次の集合も決定可能である.

$$\mathit{DS1} = \{(a, b, c) \in \mathbb{N}^3 \mid \text{方程式 } ax + by = c \text{ は整数解を持つ }\}$$

なぜならば, $a \neq 0$, $b \neq 0$ のとき $ax + by = c$ が整数解 $(x, y) \in \mathbb{Z}^2$ を持つのは, $a$ と $b$ の最大公約数が $c$ の約数のときに限られるからである. したがってプログラム

$$\begin{array}{lcl} \mathsf{ds1} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{ds1}\ a\ b\ c & = & \mathsf{if}\ (\mathsf{zero?}\ a)\ \mathsf{then}\ (\mathsf{div}\ b\ c)\ \mathsf{else} \\ & & \quad \mathsf{if}\ (\mathsf{zero?}\ b)\ \mathsf{then}\ (\mathsf{div}\ a\ c)\ \mathsf{else}\ \mathsf{div}\ (\mathsf{gcd}\ a\ b)\ c \end{array}$$

を考えれば, 任意の $(a, b, c) \in \mathbb{N}$ について

$$\mathsf{ds1\ a\ b\ c} = \mathsf{true} \quad \Longleftrightarrow \quad (a, b, c) \in \mathit{DS1}$$

が成り立つ.

これを一般化しよう.

$$5x^3yz^2 - 9xy^7 + 12z^3$$

のように整数を係数とする多変数多項式のことを**ディオファントス多項式**という. ディオファントス多項式 $p = p(x_1, \ldots, x_n)$ は文字列で表せるから, ゲーデル数 $\ulcorner p \urcorner \in \mathbb{N}$ で表すことができる. そこで次の集合を考える.

$$DS = \{ \ulcorner p \urcorner \in \mathbb{N} \mid p(x_1, \ldots, x_n) = 0 \text{ は整数解を持つ} \}$$

この集合は再帰的可算である. すなわち, 次を満たす再帰的プログラム $\mathsf{ds} : \mathbf{N} \Rightarrow \mathbf{B}$ が存在する.

$$\mathsf{ds}\ \ulcorner p \urcorner = \mathsf{true} \quad \Longleftrightarrow \quad p(x_1, \ldots, x_n) = 0 \text{ は整数解を持つ.}$$

簡単のため $p(x)$ を 1 変数のディオファントス方程式とする. このときプログラム ds は, 整数

$$n = 0, \pm 1, \pm 2, \pm 3, \cdots$$

のそれぞれについて, $p(n) = 0$ が成り立つかどうかを調べていく. もし成り立つなら true を出力して停止し, 成り立たないなら次の整数へと進む.

このプログラムは, 整数解が存在しなければ永遠に停止しないので全域的ではない. 実際, 後で述べるとおり $DS$ を計算する全域的なプログラムは存在しない. すなわち $DS$ は決定不能である.

「再帰的可算」とは recursively enumerable の訳であるが, この語は「再帰的枚挙可能」と訳されることのほうが多い. 名前の由来は次の定理にある.

**定理 3.7**

空でない集合 $R \subseteq \mathbb{N}$ が再帰的可算であることと, 次のような全域再帰的関数 $f : \mathbb{N} \to \mathbb{N}$ が存在することは一致する.

$$n \in R \quad \Longleftrightarrow \quad \text{ある } m \in \mathbb{N} \text{ について } f(m) = n.$$

つまり空でない再帰的可算集合 $R$ は全域再帰的な $f$ を用いて

$$R = \{ f(0), f(1), f(2), \ldots \}$$

と枚挙できるということである（ただし昇順とは限らないし、同じ要素が複数回現れる可能性がある）.

**証明：** 集合 $R$ が全域再帰的関数 $f : \mathbb{N} \to \mathbb{N}$ を用いて上のように表されているとする. 定義により $f$ を計算する全域再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ が存在する. このとき新たなプログラム g を以下のように構成する.

$$\begin{array}{lcl} \mathsf{g}' & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{g}'\ x\ y & = & \mathsf{if}\ (\mathsf{f}\ x) = y\ \mathsf{then\ true\ else}\ \mathsf{g}'\ (\mathsf{s}\ x)\ y \end{array}$$

$$\begin{array}{lcl} \mathsf{g} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{g}\ y & = & \mathsf{g}'\ 0\ y \end{array}$$

すると

$$\begin{array}{rcl} n \in R & \Longleftrightarrow & \text{ある } m \in \mathbb{N} \text{ について } f(m) = n \\ & \Longleftrightarrow & \text{ある } m \in \mathbb{N} \text{ について } \mathsf{f\ m} = \mathsf{n} \\ & \Longleftrightarrow & \mathsf{g\ n} = \mathsf{true} \end{array}$$

となるので $R$ は再帰的可算である.

逆方向についてはアイデアのみ述べる. $R$ は再帰的可算なので, ある再帰的プログラム $\mathsf{g} : \mathbf{N} \Rightarrow \mathbf{B}$ が存在し, $n \in R \Longleftrightarrow \mathsf{g\ n} = \mathsf{true}$ となる. $R$ は空ではないので, 要素 $n_0 \in R$ をひとつ選んでおく.

次の作業を行う再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ を構成すればよい. f は

$$\mathsf{g\ 0}, \quad \mathsf{g\ 1}, \quad \mathsf{g\ 2}, \quad \mathsf{g\ 3}, \quad \cdots$$

のそれぞれの計算を並行的にシミュレートする. そして $m$ ステップ目で g n のシミュレーションが完了し, $\mathsf{g\ n} = \mathsf{true}$ がわかったら $\mathsf{f\ m} = \mathsf{n}$ を出力する. さもなければ $\mathsf{f\ m} = \mathsf{n}_0$ を出力する.

f は高々有限ステップしかシミュレーションを行わないので停止性を満たす. また $\mathsf{g\ n} = \mathsf{true}$ である限り, 必ず何ステップ目かでそのシミュレーションが行われるので,

$$n \in R \quad \Longleftrightarrow \quad \mathsf{g\ n} = \mathsf{true} \quad \Longleftrightarrow \quad \text{ある } m \in \mathbb{N} \text{ について } \mathsf{f\ m} = \mathsf{n}$$

が成り立つ. □

再帰的可算集合と決定可能集合の特徴を表す定理を紹介しておこう. $k+1$ 項関係 $R \subseteq \mathbb{N}^{k+1}$ が与えられたとき,

$$\exists R = \{(n_1, \ldots, n_k) \in \mathbb{N}^k \mid \text{ある } n_0 \in \mathbb{N} \text{ について } (n_0, n_1, \ldots, n_k) \in R\}$$

と定義する.

**定理 3.8**

$R \subseteq \mathbb{N}^{k+1}$ が再帰的可算ならば, $\exists R \subseteq \mathbb{N}^k$ も再帰的可算である.
$R \subseteq \mathbb{N}^k$ が決定可能ならば, その補集合 $\mathbb{N}^k \backslash R$ も決定可能である.

**証明：** 第二の主張のみ示す。簡単のため $k = 1$ とする. 仮定により $R$ を計算する全域再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{B}$ が与えられている. そこで

$$\begin{array}{lcl} \mathsf{notf} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{notf}\ x & = & \mathsf{not}\ (\mathsf{f}\ x) \end{array}$$

とすれば, 任意の $n \in \mathbb{N}$ について

$$\mathsf{notf\ n} = \mathsf{true} \quad \Longleftrightarrow \quad \mathsf{f\ n} = \mathsf{false} \quad \Longleftrightarrow \quad n \notin R$$

が成り立つ. □

再帰的可算集合と決定可能集合の関係は次の定理によく表れている.

**定理 3.9**

$R \subseteq \mathbb{N}^k$ が決定可能であることの必要十分条件は, $R$ とその補集合 $\mathbb{N}^k \backslash R$ がともに再帰的可算なことである.

**証明：** 簡単のため $k = 1$ とする.

$R$ が決定可能ならば, 定理 3.8 によりその補集合も決定可能である. 決定可能 $\Rightarrow$ 再帰的可算は定義より明らか.

逆方向を示すには, 定理 3.7 が便利である. （$R \neq \emptyset$ を仮定して）$R$ と $\mathbb{N}\backslash R$ を枚挙する全域再帰的プログラムをそれぞれ $\mathsf{f}, \mathsf{g}$ とする. すなわち

$$\begin{array}{rcl} n \in R & \Longleftrightarrow & \text{ある } m \in \mathbb{N} \text{ について } \mathsf{f}\ \mathsf{m} = \mathsf{n} \\ n \notin R & \Longleftrightarrow & \text{ある } m \in \mathbb{N} \text{ について } \mathsf{g}\ \mathsf{m} = \mathsf{n} \end{array}$$

このとき次のプログラムを考える.

$$\begin{array}{lcl} \mathsf{h}' & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{h}'\ x\ y & = & \mathsf{if}\ (\mathsf{f}\ x = y)\ \mathsf{then\ true\ else} \\ & & \quad \mathsf{if}\ (\mathsf{g}\ x = y)\ \mathsf{then\ false\ else}\ (\mathsf{h}'\ (\mathsf{s}\ x)\ y) \end{array}$$

$$\begin{array}{lcl} \mathsf{h} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{h}\ y & = & \mathsf{h}'\ 0\ y \end{array}$$

すると

$$n \in R \quad \Longleftrightarrow \quad \mathsf{h}\ \mathsf{n} = \mathsf{true}$$

であり, なおかつ $\mathsf{h}$ は全域再帰的である. 実際, どんな $n \in \mathbb{N}$ についても $n \in R$ または $n \notin R$ なので, $\mathsf{f}\ \mathsf{m} = \mathsf{n}$ または $\mathsf{g}\ \mathsf{m} = \mathsf{n}$ となる $m \in \mathbb{N}$ が必ず存在する. しかも, $\mathsf{f}$ も $\mathsf{g}$ も全域再帰的であるから, その計算は必ず停止する. ゆえに $\mathsf{h}\ y = \mathsf{h}'\ 0\ y$ の計算は停止し, true か false を返す. □

具体例として集合 $DS$ について考えよう. すでに述べたとおり, この集合は再帰的可算である. そこで上の定理によれば, もし次のようなプログラム $\mathsf{notds} : \mathbf{N} \Rightarrow \mathbf{B}$ が存在すれば, $DS$ が決定可能だと言えることになる.

$$\mathsf{notds}\ \ulcorner p \urcorner = \mathsf{true} \quad \Longleftrightarrow \quad p(x_1, \ldots, x_n) = 0 \text{ は整数解を持たない.}$$

しかしそのような再帰的プログラムは存在しないことがわかっている.

プログラミングの際には, このような事態がしばしば起こる. ある問いが与えられたとき, それを半決定するプログラム（たとえば ds）は容易に書けるのだが, その補集合を半決定するプログラム（たとえば notds）が書けないといった事態である. 後者が書けるかどうか, それが再帰的可算と決定可能を分ける境目なのである.

## 3.4 決定不能集合

本節では, 今まで準備してきた事柄を用いて「計算できない」（すなわち決定不能な）集合の存在を示す.

再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{B}$ と自然数 $m \in \mathbb{N}$ が与えられたとき, $\mathsf{f}\ \mathsf{m}$ の計算結果は

$$\mathsf{f}\ \mathsf{m} = \mathsf{true}, \qquad \mathsf{f}\ \mathsf{m} = \mathsf{false}, \qquad \mathsf{f}\ \mathsf{m} \Uparrow$$

の 3 通りである. 前 2 つの場合 $\mathsf{f}\ \mathsf{m} \Downarrow$ と書くのであった.

以下で「計算できない」集合の具体例を与える（チューリングの**停止問題**）.

**定理 3.10(停止問題の決定不能性)**

次の集合 $Halt \subseteq \mathbb{N}^2$ は再帰的可算ではあるが決定可能ではない.

$$Halt = \{(\ulcorner \mathsf{f} \urcorner, m) \in \mathbb{N}^2 \mid \mathsf{f} \text{ は型 } \mathbf{N} \Rightarrow \mathbf{B} \text{ の再帰的プログラムで } \mathsf{f}\ \mathsf{m} \Downarrow\}.$$

ここで $\ulcorner \mathsf{f} \urcorner$ は再帰的プログラム f のゲーデル数を表す.

**証明：** まず再帰的可算であることを示すために, 前節の要領で次の性質を満たす再帰的プログラム $\mathsf{evalb} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B}$ を構成する（eval との違いは出力型が $\mathbf{N}$ でなく $\mathbf{B}$ な点のみである）.

$$\begin{array}{lcl} \mathsf{evalb}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m} = \mathsf{true} & \Longleftrightarrow & \mathsf{f}\ \mathsf{m} = \mathsf{true} \\ \mathsf{evalb}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m} = \mathsf{false} & \Longleftrightarrow & \mathsf{f}\ \mathsf{m} = \mathsf{false} \\ \mathsf{evalb}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m} \Uparrow & \Longleftrightarrow & \mathsf{f}\ \mathsf{m} \Uparrow \end{array}$$

これを用いて

$$\begin{array}{lcl} \mathsf{halt} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{halt}\ x\ y & = & \mathsf{if}\ (\mathsf{evalb}\ x\ y)\ \mathsf{then\ true\ else\ true} \end{array}$$

とすれば計算の停止を半決定することができる（最後の true は誤植ではない）.

$$\mathsf{halt}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m} = \mathsf{true} \quad \Longleftrightarrow \quad \mathsf{evalb}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m} \Downarrow \quad \Longleftrightarrow \quad \mathsf{f}\ \mathsf{m} \Downarrow \quad \Longleftrightarrow \quad (\ulcorner \mathsf{f} \urcorner, m) \in Halt.$$

ゆえに $Halt$ は再帰的可算である.

次に $Halt$ が決定可能でないことを対角線論法により示す. 仮に $Halt$ が決定可能だとすると, 次のような「**全域**」再帰的プログラム $\mathsf{haltd} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B}$ が存在するはずである.

$$\begin{array}{lcll} \mathsf{haltd}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{m} & = & \mathsf{true} & (\mathsf{f}\ \mathsf{m} \Downarrow \text{のとき}) \\ & = & \mathsf{false} & (\mathsf{f}\ \mathsf{m} \Uparrow \text{のとき}) \end{array}$$

ここから矛盾を導くために, 次のプログラム diag を考える（diagonal=対角線）.

$$\begin{array}{lcl} \mathsf{diag} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{diag}\ x & = & \mathsf{if}\ (\mathsf{haltd}\ x\ x)\ \mathsf{then}\ (\mathsf{diverge}\ x)\ \mathsf{else\ true} \end{array}$$

定義により, プログラム diag は停止しないか true を返すかのどちらかである（false は返さない）. ところが,

$$\begin{array}{rcl} \mathsf{diag}\ \ulcorner \mathsf{diag} \urcorner = \mathsf{true} & \Longleftrightarrow & \mathsf{haltd}\ \ulcorner \mathsf{diag} \urcorner\ \ulcorner \mathsf{diag} \urcorner = \mathsf{false} \\ & \Longleftrightarrow & \mathsf{diag}\ \ulcorner \mathsf{diag} \urcorner \Uparrow . \end{array}$$

つまり $\mathsf{diag}\ \ulcorner \mathsf{diag} \urcorner$ の計算は, 停止することと停止しないことが同値である. これは矛盾である. □

上の定理を取り掛かりとして, さまざまな集合 $R$ の決定不能性を示していくことができる. 基本的な方針は**還元法**である. すなわち, 仮に $R$ が決定可能ならば $Halt$ も決定可能となることを示す. そうすれば定理 3.10 により $R$ は決定不能であることが従う.

**定理 3.11**

集合

$$Halt0 = \{\ulcorner \mathsf{f} \urcorner \mid \mathsf{f} \text{ は型 } \mathbf{N} \Rightarrow \mathbf{B} \text{ の再帰的プログラムで } \mathsf{f}\ 0 \Downarrow\}$$

は再帰的可算だが, 決定可能ではない.

**証明：** 再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{B}$ と自然数 $n \in \mathbb{N}$ が与えられたとき, 再帰的プログラム

$$\begin{array}{lcl} \mathsf{fn} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{fn}\ x & = & \mathsf{f}\ \mathsf{n} \end{array}$$

を構成する. すると明らかに

$$\mathsf{fn}\ 0 \Downarrow \iff \mathsf{f}\ \mathsf{n} \Downarrow$$

が成り立つ. しかもこの構成自体は原始再帰的である. すなわち原始再帰的関数 $\mathsf{inst} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ で

$$\mathsf{inst}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{n} = \ulcorner \mathsf{fn} \urcorner$$

を満たすものが存在する.

以下, 還元法により *Halt0* が決定不能であることを示す. 仮に *Halt0* が決定可能だとすると, ある**全域**再帰的プログラム halt0 が存在し,

$$\begin{array}{rcll} \mathsf{halt0}\ \ulcorner \mathsf{f} \urcorner & = & \mathsf{true} & (\mathsf{f}\ 0 \Downarrow \text{ のとき}) \\ & = & \mathsf{false} & (\mathsf{f}\ 0 \Uparrow \text{ のとき}) \end{array}$$

となるはずである. そこでプログラム

$$\begin{array}{lcl} \mathsf{haltd} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{haltd}\ x\ y & = & \mathsf{halt0}\ (\mathsf{inst}\ x\ y) \end{array}$$

を考えると, これは**全域**再帰的であり, しかも

$$\begin{array}{rcl} \mathsf{haltd}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{n} = \mathsf{true} & \iff & \mathsf{halt0}\ (\mathsf{inst}\ \ulcorner \mathsf{f} \urcorner\ \mathsf{n}) = \mathsf{true} \\ & \iff & \mathsf{halt0}\ \ulcorner \mathsf{fn} \urcorner = \mathsf{true} \\ & \iff & \mathsf{fn}\ 0 \Downarrow \\ & \iff & \mathsf{f}\ \mathsf{n} \Downarrow \\ & \iff & (\ulcorner \mathsf{f} \urcorner, n) \in Halt \end{array}$$

を満たす. これは集合 *Halt* が決定可能なことを示しており, 定理 3.10 と矛盾する. □

上の *Halt0* は, 入力値 0 に対して停止するプログラム（のゲーデル数）全体の集合である. これが決定不能だというのであるが, もちろん入力値を 1 に変えても決定不能なままであるし, 「ある入力値について」または「すべての入力値について」停止するプログラム全体を考えても決定不能なことは変わらない.

実をいうと, この手の集合はほとんど全部決定不能になることが知られている.

**定理 3.12(ライスの定理)**

$F$ を再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{B}$ の一部からなる集合とする．ただし $F$ は空でなく, $F$ に含まれない再帰的プログラムが少なくとも 1 つ存在する．さらに $\mathsf{f} \in F$ で $[\![\mathsf{f}]\!] = [\![\mathsf{g}]\!]$ ならば $\mathsf{g}$ も $F$ に含まれるとする．
このとき集合

$$R_F \ := \ \{\ulcorner \mathsf{f} \urcorner \in \mathbb{N} \mid \mathsf{f} \in F\}$$

は決定不能である．

証明はそれほど難しくないが, ここでは省略する．

## 3.5 ヒルベルトの第 10 問題

最後にあと 2 つだけ決定不能集合の例を挙げよう．集合

$$DS = \{\ulcorner p \urcorner \in \mathbb{N} \mid p(x_1, \ldots, x_n) = 0 \text{ は整数解を持つ } \}$$

を少し改変して

$$DS' = \{\ulcorner p \urcorner \in \mathbb{N} \mid p(x_1, \ldots, x_n) = 0 \text{ は自然数解を持つ } \}$$

とする．この 2 つの集合はある意味等価である．

**補題 3.13**

$DS$ が決定可能なことと $DS'$ が決定可能なことは一致する．

**証明：** どんな整数も 2 つの自然数の差として表せる．よって多項式 $p(x_1, \ldots, x_n)$ が与えられたとき,

$$q(y_1, \ldots, y_n, z_1, \ldots, z_n) \ := \ p(y_1 - z_1, \ldots, y_n - z_n)$$

と定めれば,

$$p \text{ は整数解を持つ} \quad \Longleftrightarrow \quad q \text{ は自然数解を持つ}$$

となる．すなわち $\ulcorner p \urcorner \in DS \Longleftrightarrow \ulcorner q \urcorner \in DS'$.

一方, どんな自然数 $n$ も 4 つの整数 $a, b, c, d$ を用いて $n = a^2 + b^2 + c^2 + d^2$ と表せる（ラグランジュの**四平方定理**）．よって多項式 $p(x)$（簡単のため 1 変数とする）が与えられたとき,

$$q(x, y, z, w) \ := \ p(x^2 + y^2 + z^2 + w^2)$$

と定めれば,

$$p \text{ は自然数解を持つ} \quad \Longleftrightarrow \quad q \text{ は整数解を持つ}$$

となる．すなわち $\ulcorner p \urcorner \in DS' \Longleftrightarrow \ulcorner q \urcorner \in DS$.

以上の式変形は簡単な原始再帰的プログラムにより実行できる．よって一方の決定可能性から他方の決定可能性が従う． □

さて, ディオファントス多項式 $p(x, \vec{y})$ が与えられたとき, 集合

$$\{n \in \mathbb{N} \mid p(n, \vec{y}) = 0 \text{ は自然数解を持つ } \}$$

は再帰的可算であることに注意する．実際, 足し算, 掛け算, 等号はいずれも原始再帰的なので, 自然数の組 $(n, \vec{m})$ が与えられたときに $p(n, \vec{m}) = 0$ が成り立つかどうかを判定する原始再帰的プログラムが存在する．あとは定理 3.8 を用いればよい．

これの反対を主張するのが次の定理である（証明は難解である）．

**定理 3.14(MRDP 定理)**

どんな再帰的可算集合 $R \subseteq \mathbb{N}$ についてもあるディオファントス多項式 $p_R(x, \vec{y})$ が存在し,

$$n \in R \quad \Longleftrightarrow \quad p_R(n, \vec{y}) = 0 \text{ は自然数解を持つ}$$

が全ての $n \in \mathbb{N}$ について成り立つ．

つまり, ディオファントス多項式は一種のプログラミング言語と見なせるということである．ちなみに定理の名前は証明に貢献した数学者たち Martin Davis, Yuri Matiyasevich, Hilary Putnam, Julia Robinson の名字に由来する．

この定理により *Halt0* を *DS* に還元することができる．

**定理 3.15(ヒルベルト第 10 問題の否定的解決)**

集合 *DS* は再帰的可算であるが決定可能ではない．

**証明：** 再帰的可算であることはすでに見た．決定不能であることは還元法により示すことができる．仮に *DS* が決定可能だとすると, 補題 3.13 により $DS'$ も決定可能なので全域再帰的プログラム $\mathsf{dsd} : \mathbf{N} \Rightarrow \mathbf{B}$ が存在し,

$$\begin{array}{rcll} \mathsf{dsd}\ \ulcorner p(\vec{x}) \urcorner & = & \mathsf{true} & (p(\vec{x}) = 0 \text{ は自然数解を持つ}) \\ & = & \mathsf{false} & (p(\vec{x}) = 0 \text{ は自然数解を持たない}) \end{array}$$

が成り立つはずである．一方, 定理 3.11 により集合 $\mathit{Halt0} \subseteq \mathbb{N}$ は再帰的可算である．それゆえ定理 3.14 により, 対応するディオファントス多項式 $p_{Halt0}(x, \vec{y})$ が存在する．次の性質を満たす原始再帰的プログラム $\mathsf{p} : \mathbf{N} \Rightarrow \mathbf{N}$ は容易に構成することができる．

$$\mathsf{p}\ \mathsf{n} = \ulcorner p_{Halt0}(n, \vec{y}) \urcorner.$$

そこでプログラム

$$\begin{array}{lcl} \mathsf{halt0d} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{halt0d}\ x & = & \mathsf{dsd}(\mathsf{p}\ x) \end{array}$$

を考えると, これは全域再帰的であり, しかも

$$\begin{array}{lcl} \mathsf{halt0d}\ \mathsf{n} = \mathsf{true} & \Longleftrightarrow & \mathsf{dsd}(\mathsf{p}\ \mathsf{n}) = \mathsf{true} \\ & \Longleftrightarrow & \mathsf{dsd}\ \ulcorner p_{Halt0}(n, \vec{y}) \urcorner = \mathsf{true} \\ & \Longleftrightarrow & p_{Halt0}(n, \vec{y}) = 0 \text{ は自然数解を持つ} \\ & \Longleftrightarrow & n \in \mathit{Halt0} \end{array}$$

となる．これは *Halt0* が決定可能であることを示しており, 定理 3.11 と矛盾する． □

1900 年にヒルベルトが提示した 23 問題のうち, 第 10 番目は「与えられたディオファントス方程式が整数解を持つかどうかを判定する一般的手続きを与えよ」というものであった．1970 年に証明が完了した MRDP 定理により, 第 10 問題に否定的解決がもたらされたのである．

代表的な決定不能問題をもう 1 つ挙げておく.

**定理 3.16(ヒルベルトの決定問題の否定的解決)**

集合

$$\mathit{Valid} \ := \ \{ \ulcorner \varphi \urcorner \in \mathbb{N} \mid \varphi \text{ は論理的に正しい一階論理式 } \}$$

は再帰的可算であるが決定不能である.

この問題も同じくヒルベルトに由来するものであり, チャーチとチューリングにより 1936 年に否定的解決がもたらされた. この手の事柄に触発されて, チューリングは “Computing Machine”（いわゆる**チューリング機械**）という数理的概念を考案した. ここから（現代的な意味での）コンピュータの歴史が始まったのである.

# 4 まとめ

計算可能な関数とは, ひらたくいえばコンピュータプログラムにより計算できる関数のことである. 本稿では, 簡単な一階関数型プログラム言語を考え, 計算可能な関数の分類を行った. まとめると次のようになる（ただし $f : \mathbb{N} \to \mathbb{N}_\perp$ なる関数のみを考える）.

$$\begin{array}{rl} \text{（原始再帰的関数全体）} & \subsetneq \ \text{（全域再帰的関数全体）} \\ & \subsetneq \ \text{（再帰的関数全体）} \\ & \subsetneq \ \mathbb{N} \to \mathbb{N}_\perp \end{array}$$

また, 同等性 $\mathbb{N} \to \mathbb{B} \cong \wp(\mathbb{N})$（命題 1.9）に基づいて関数 $f : \mathbb{N} \to \mathbb{B}$ に対応する集合 $R \in \wp(\mathbb{N})$ を考えると, 全域再帰的関数には決定可能集合が, 再帰的関数には再帰的可算集合が対応する. 大小関係は以下の通りである（ただし $\mathbb{N}$ の部分集合のみを考える）.

$$\text{（決定可能集合全体）} \ \subsetneq \ \text{（再帰的可算集合全体）} \ \subsetneq \ \wp(\mathbb{N})$$

「計算可能＝（全域）再帰的」というチャーチの提唱に鑑みれば, これらはみな, 個別のハードウェアやプログラミング言語に依存しない普遍的な事柄とみなすことができる. とくに決定可能・決定不能の区別はハードウェアやプログラミング言語に依存しない. たとえばディオファントス方程式の可解性が決定不能というのは, 単に我々のプログラミング言語で判定プログラムを書けないというだけではなく, およそプログラミング言語と呼ばれうるどんな言語を用いようとも決して判定することができない, そういう絶対不能性を意味するのである.

原始再帰的プログラムは, プログラムの形からして全域的であることが一目瞭然である. 一方で再帰的プログラムにはそのような保証がないので, 必要に応じて, 各プログラム f ごとに全域性を「証明」しなければならない. そこで重要になるのが, 全域性を証明するための系統的かつ半自動的な方法論である（**プログラム停止解析**）.

ゲーデルの**不完全性定理**が絡んでくるのもここである. 自然数上の再帰的プログラム f が全域的

であるとは

$$\text{すべての } n \in \mathbb{N} \text{ について } \mathsf{f}\ \mathsf{n} \Downarrow$$

ということであるが, これはゲーデル数を用いれば, 初等数論の文で表すことができる. 第一不完全性定理によれば, どんな（$\mathbf{Q}$ の再帰的拡大で無矛盾な）公理系 $T$ をとっても, 真なのに $T$ からは証明できない文が存在する. とくに, どんな公理系 $T$ をとっても, 全域性が証明できない再帰的プログラム $\mathsf{f}$ が存在する. すなわち, 完全に系統的で自動的な全域性証明の方法などありえないのである.

もちろん, だからといってプログラムの解析をあきらめる理由にはならない. むしろ逆で, 理論的解決がありえないからこそ立ち向かうのが, コンピュータサイエンスの本懐であるといえる. あと 3 点補足して, 本稿を終えることにする.

- 非可算集合にさまざまな度合いがあるように, 決定不能集合にもさまざまな度合いがある. 本稿では計算「可能」関数や決定「可能」集合に力点をおき, 決定「不能」の度合いについてはまったく論じなかった. 一方で伝統的な**再帰理論**（計算可能性理論）は, むしろ決定不能な集合たちが織り成すユニバースの構造解析に力を入れている. 数学基礎論の一分野だが, 近年では**逆数学**, **アルゴリズム的ランダムネス**, **計算可能解析学**等の理論と結びついて, 新たな展開を見せている.
- 本稿では, 与えられた関数が計算可能かどうか, 与えられた集合が決定可能かどうかについて論じたが, 現代的な観点からいってむしろ重要なのは, どの程度の時間で, どの程度のメモリを用いれば計算可能かである（**計算複雑性理論**）. 一見実用的な問題設定に見えるが, 理論的にも大変興味深い. 時間や空間に一定の制約を設けることで, 計算の本質を突く不思議な事象が多く表れるからである. 100 万ドルの賞金のかかった $\mathbf{P} \neq \mathbf{NP}$ **予想**も計算複雑性に関する未解決問題である. 本稿で計算理論に興味を持たれた方は, ぜひ計算複雑性の勉強にも取り組んでみてほしい.
- 本講義では一階のプログラムしか扱わなかったが, 関数型プログラミングが本領を発揮するのは, むしろ**高階プログラム**においてである. 高階プログラミングは, 実用的・美的観点から重要であるばかりではなく, 理論面でも多くの興味深い問いを投げかけてくる. **構成的論理**との対応が見えてくるのも高階に入ってからである. これらの点については, 本講義の続き（室谷先生, 長谷川先生）にご期待いただきたい.