コンピュータサイエンス基礎・講義資料（2019 年度）

照井一成
京都大学数理解析研究所
terui@kurims.kyoto-u.ac.jp
http://www.kurims.kyoto-u.ac.jp/∼terui

1　簡単な集合論の準備

1.1　集合

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。

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

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。

「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 の要素ではない」であり

$$a \notin A$$

と書く。たとえば

$$0 \in \mathbb{N}, \qquad 0 \in \mathbb{R}, \qquad 0 \notin \mathbb{B}$$

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。

$\varphi(x)$ 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

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

と書く（人によっては $\{x \in \mathbb{N} \mid \varphi(x)\}$ とも書く。あるいは「$\in \mathbb{N}$」が明らかなときには省略して $\{x : \varphi(x)\}$ と書いてしまうこともある）。たとえば

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

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

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

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

$$\varphi(a) \quad \Longleftrightarrow \quad a \in \{x \in A : \varphi(x)\}$$

が成り立つ。

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。もしも対象の範囲を制限しなくてよいのであれば、集合

$$R = \{x : x \notin x\}$$

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

$$R \in R \quad \Longleftrightarrow \quad R \in \{x : x \notin x\} \quad \Longleftrightarrow \quad R \notin R$$

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**）。

全く同じ要素からなる 2 つの集合は等しい。すなわち

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

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

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

当然のことながら

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

が成り立つ。

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**交わり**、**和**、**差**はそれぞれ $A \cap B$、$A \cup B$、$A - B$（あるいは $A \backslash B$）と書かれるが、これらは次のように定めることができる。

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

操作 $\cap$ と $\cup$ は次の性質を満たす。

$$\begin{aligned} C \subseteq A \cap B &\iff C \subseteq A \text{ かつ } C \subseteq B \\ A \cup B \subseteq C &\iff A \subseteq C \text{ かつ } B \subseteq C \end{aligned}$$

つまり半順序 $\subseteq$ に関して、$A \cap B$ と $A \cup B$ は A と B の下限と上限を与える。

集合 A の部分集合全体の集まりを**巾集合**といい、$\wp(A)$ と書く。つまり

$$X \in \wp(A) \iff X \subseteq A.$$

たとえば $A = \{a, b\}$ のとき

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

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。

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

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 $A \times B \times C = \{(a, b, c) : a \in A,\ b \in B,\ c \in C\}$ 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

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

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 $\mathbb{R}$ を実直線のことだと思えば、実平面の点は $\mathbb{R}^2$ の要素、（3 次元）実空間の点は $\mathbb{R}^3$ の要素で表すことができる。

この定義により、任意の $m, n \in \mathbb{N}$ について

$$A^{m+n} \;\cong\; A^m \times A^n$$

と見なすことができる。一般に $\cong$ は両辺が**同型**であること、すなわち同じ構造を持ち、同一視できることを表す記号である。ここでは何ら構造を持たない単なる集合を考えているので、$\cong$ は単に同等であること（後述）を表す。

1.2 性質・関係・関数

すでに述べたように「素数である」という性質は集合 $\{x \in \mathbb{N} : x \text{ は素数である }\}$ 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 $\{(n, m) \in \mathbb{N}^2 : n \text{ は } m \text{ の約数である }\}$ 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。

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。

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**を $g \circ f$ と書き、

$$g \circ f(x) \ := \ g(f(x)) \quad : A \to C$$

と定める（:= は左辺が右辺で定義されていることを表す記号）。上の例でいえば $h_1 \circ h_2(m, n) = (m + n)^2$ である。

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 $f : A \to B$ と $f \in A \to B$ と $f \in B^A$ は同じことを表す。

関数 $f : A \to B$ は 2 項関係

$$\{(x, y) : f(x) = y\} \ \subseteq \ A \times B$$

と同一視できる（なぜなら一方から他方が復元できるので）。これを f の**グラフ**という。一般に 2 項関係 $R \subseteq A \times B$ が関数のグラフとなるための必要十分条件は以下の通りである。

任意の $x \in A$ について $(x, y) \in R$ を満たす $y \in B$ がちょうど 1 つ存在する。

このように、1 項関数とは 2 項関係の特別な場合であると考えることができるので、

$$A \to B \ \subseteq \ \wp(A \times B)$$

が成り立つものと思ってよい。

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 $x, y \in A$ についても、$f(x) = f(y)$ ならば $x = y$.
f は**全射**である $\iff$ どんな $y \in B$ についても、$f(x) = y$ を満たす $x \in A$ 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 $x \neq y \Longrightarrow f(x) \neq f(y)$ 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 $f : \mathbb{R} \to \mathbb{R}$ を連続関数とすると、

- 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。

また $A \subseteq B$ のとき、写像 $\iota : A \to B$ を $\iota(a) := a$ により定めると単射が得られる。これを**包含写像**という。

次のことが成り立つ（証明には選択公理を使う）。

命題 1.1

A, A', B を集合とする（ただし $A \neq \emptyset$）。

$$\begin{array}{lcl} A \text{ から } B \text{ への単射が存在する} & \iff & B \text{ から } A \text{ への全射が存在する。} \\ A' \text{ から } B \text{ への全単射が存在する} & \iff & B \text{ から } A' \text{ への全単射が存在する。} \end{array}$$

では単射（あるいは全射）と全単射の関係はどうなっているだろうか？次の**カントール・ベルンシュタインの定理**が答えを与えてくれる。

定理 1.2

A から B への単射と B から A への単射があれば、A から B への全単射が存在する。

命題 1.1 に鑑みれば、上の定理は「A から B への単射と全射があれば、A から B への全単射が存在する」と言っても同じことである。

集合 A, B の間に全単射があるとき、A と B は**同等**であるといい、ここでは $A \cong B$ と書く。

命題 1.3

$\cong$ は**同値関係**である。すなわち A, B, C を集合とするとき、以下が成り立つ。

1. $A \cong A$（反射律）
2. $A \cong B$ ならば $B \cong A$（対称律）
3. $A \cong B$ かつ $B \cong C$ ならば $A \cong C$（推移律）

重要な同等性を 2 つ挙げておく。集合 A とその部分集合 P 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。

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

これを P の**特性関数**という。

命題 1.4

A を集合とする。部分集合 $P \in \wp(A)$ に対し特性関数 $\chi_P \in A \to \mathbb{B}$ を割り当てる対応は全単射であり、同等性

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

を定める。

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 $curry(f) : A \to (B \to C)$ と表す。

命題 1.5

A, B, C を集合とする。関数 $f : A \times B \to C$ に対して $curry(f) : A \to (B \to C)$ を割り当てる対応は全単射であり、同等性

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

を定める。

1.3 可算無限集合

次に、無限集合の大きさについて考える。$\mathbb{N}$ と同等な集合 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（「可算」という語は曖昧で、人によって「可算無限」のことも「高々可算」のことも表す）。

命題 1.6

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. 偶数全体の集合
4. 素数全体の集合
5. 整数の集合 $\mathbb{Z}$
6. 有理数の集合 $\mathbb{Q}$

たとえば $\mathbb{N} \cong \mathbb{N}^2$ をいうには、関数

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

が $\mathbb{N}^2$ から $\mathbb{N}$ への全単射であることを示せばよい。

一方で可算でない無限集合（**非可算集合**）も存在する。たとえば：

命題 1.7

$\wp(\mathbb{N})$ は非可算集合である。

証明はカントールの**対角線論法**による。仮に $\wp(\mathbb{N}) \cong \mathbb{N}$ とすると、全単射 $f : \mathbb{N} \to \wp(\mathbb{N})$ が存在するはずである。そこで

$$D := \{x \in \mathbb{N} : x \notin f(x)\} \subseteq \mathbb{N}$$

とおくと、f の全射性よりある $d \in \mathbb{N}$ が存在して $D = f(d)$ となる。すると

$$d \in D \iff d \in \{x \in \mathbb{N} : x \notin f(x)\} \iff d \notin f(d) \iff d \notin D$$

となり矛盾が生じる。

命題 1.8

以下の集合は $\wp(\mathbb{N})$ と同等である。

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 $\mathbb{R}$

一般に、集合 A よりも $\wp(A)$ のほうが必ず大きくなる。よって

$$\mathbb{N}, \quad \wp(\mathbb{N}), \quad \wp(\wp(\mathbb{N})), \quad \wp(\wp(\wp(\mathbb{N}))), \quad \ldots$$

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。

数の集合について考えてみると、$\mathbb{Z}$ や $\mathbb{Q}$ は $\mathbb{N}$ と同等であり、$\mathbb{R}$ や無理数の集合 $\mathbb{R} - \mathbb{Q}$ 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？すなわち

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

なる集合 X で $\mathbb{N}$ とも $\mathbb{R}$ とも同等でないようなもの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。

コンピュータサイエンスで決定的に重要なのは次の事実である。今、英数字を ASCII コードと同一視すれば、どんなプログラムも自然数の有限列と同一視することができる。すなわちどんなプログラミング言語を考えても、プログラム全体の集合は $\mathbb{N}^*$ の部分集合と同一視できるので、高々可算である。一方で $\mathbb{N} \to \mathbb{N}$ は非可算集合であり、$\mathbb{N}^*$ より真に大きい。よって

プログラムでは決して表せない（計算できない）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プログラムで計算できて、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サイエンスの出発点であったといってよい。

2　原始再帰的関数

2.1　型

プログラムのエラーを防ぐには、**型（タイプ）**を意識するのが重要である。一般に、表現 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。プログラムに入力値を与えたり、複数のプログラムを合成したりする際に型を意識することにより、多くの**型エラー**をプログラム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、不動小数点型、（種々の）レコード型等が挙げられる。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.　さて、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、（$\mathbf{N}$ から $\mathbf{N}$ への）**関数型**といわれる。s は与えられた数に 1 を加える操作に相当し、**後者関数**、**後続者関数**などと呼ばれる。

(iii) により、$\mathsf{0}$ と s さえあればどんな自然数も記述することができる。

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

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、

$$\mathsf{0}, \quad \mathsf{1}, \quad \mathsf{2}, \quad \mathsf{3}, \quad \ldots$$

という風に十進法も用いる。これらは省略表現に過ぎないことに注意。

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。

上の帰納的定義は、次の **BNF 文法**を用いて簡潔に表すことができる。

$$\begin{array}{lcl} D & ::= & \mathbf{N} \mid \mathbf{B} \\ A & ::= & D \mid (D \Rightarrow A). \end{array}$$

この文法により導出できる表現 A が一階型である。今後カッコは省略して、$(D_1 \Rightarrow (D_2 \Rightarrow A))$ 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。

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 1.5 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。

2.2 原始再帰的プログラム

次に簡単な一階の**関数型プログラミング言語**を考える。関数型言語においては、プログラミングとは関数を定義することに他ならない。以下の素材を用いて関数を定義していくことにする。

- **構成子**：0, s, true, false
- **識別子**：f_0, f_1, f_2, ...
- **変数**：x_0, x_1, x_2, ...

識別子とは、関数につける名前のことである。ただし f_i というのは名前として無味乾燥すぎるので、実際には add, zero などのわかりやすい記号も用いる。変数についても同様で、x_i だけでなく y, z, y', y'' なども臨機応変に用いる。

簡単なプログラムの例. 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。

id	:	$D \Rightarrow D$	add2	:	$\mathbf{N} \Rightarrow \mathbf{N}$
id x	=	x	add2 x	=	s(s x)
proj_1	:	$D_1 \Rightarrow D_2 \Rightarrow D_1$	ctrue	:	$D \Rightarrow \mathbf{B}$
$\mathsf{proj}_1\ x\ y$	=	x	ctrue x	=	true
proj_2	:	$D_1 \Rightarrow D_2 \Rightarrow D_2$	add4	:	$\mathbf{N} \Rightarrow \mathbf{N}$
$\mathsf{proj}_2\ x\ y$	=	y	add4 x	=	add2(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。$D \Rightarrow D$ という形をみれば、id は型 D の入力を受け取ったら同じ型の出力を返す関数であることがわかる。2 行目はカッコを用いて $\mathsf{id}(x) = x$ と書いたほうがわかりやすいかもしれないが、ここではなるべくカッコを省略して書くことにする。

次に proj_1, proj_2 は**射影関数**である。型 $D_1 \Rightarrow D_2 \Rightarrow D_i$ が示す通り、どちらも 2 つの引数をとる。D_1 と D_2 は同じ型であってもよい。

右上は、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。右中は、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。

最後に右下は 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。

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

慣れるまで読みにくいかもしれないが、$\mathsf{add}\ 0\ x$ は $\mathsf{add}(0, x)$ のことであり、$\mathsf{and}\ \mathsf{true}\ x$ は $\mathsf{and}(\mathsf{true}, x)$ のことである。

$\mathbf{N}$ 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、第一引数が 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、第一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。

なお add が足し算を表すことは、次のように確かめることができる。

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

式の定義. ここから一般論に移る。まず、**式**（あるいは**項**）を次の BNF 文法により定める。

$$M, N \ ::= \ x \mid \mathsf{f} \mid \mathsf{c} \mid (M\ N).$$

ただし x は変数、f は識別子、c は構成子（ここでは $0, \mathsf{s}, \mathsf{true}, \mathsf{false}$ のどれか）を表す。この文法により、たとえば

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

が式であることがわかる。不要なカッコは省略し、$\mathsf{add}\ (\mathsf{s}\ (\mathsf{s}\ x))\ (\mathsf{s}\ y)$ のようにも書く。一般に、複数の式を $M\ N_1\ \cdots\ N_n$ と並べて書いたら、（左に寄せてカッコをつけて）

$$(\cdots((M\ N_1)\ N_2)\cdots N_n)$$

のことを表す。

式の型. 次に、各式に型を割り当てる。上の BNF 文法によれば、式には 4 種類あることがわかるので、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。

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

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 Γ や Δ 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 MN の形の式（正確には $(M\ N)$）には次の規則を用いて型を割り当てる。

$$\frac{M : A \Rightarrow B \quad N : A}{MN : B}$$

この規則は「$M : A \Rightarrow B$ と $N : A$ が成り立つならば、$MN : B$ が成り立つ」ことを表す。たとえば型宣言 $x : \mathbf{N}, y : \mathbf{B}$ のもとで次のような導出ができる（$\mathsf{add2} : \mathbf{N} \Rightarrow \mathbf{N}$ と $\mathsf{proj}_1 : \mathbf{N} \Rightarrow \mathbf{B} \Rightarrow \mathbf{N}$ を仮定する）。

$$\dfrac{\mathsf{add2} : \mathbf{N} \Rightarrow \mathbf{N} \qquad \dfrac{\dfrac{\mathsf{proj}_1 : \mathbf{N} \Rightarrow \mathbf{B} \Rightarrow \mathbf{N} \quad x : \mathbf{N}}{\mathsf{proj}_1\ x : \mathbf{B} \Rightarrow \mathbf{N}} \qquad y : \mathbf{B}}{\mathsf{proj}_1\ x\ y : \mathbf{N}}}{\mathsf{add2}(\mathsf{proj}_1\ x\ y) : \mathbf{N}}$$

つまり型宣言 $x : \mathbf{N}, y : \mathbf{B}$ のもとで $\mathsf{add2}(\mathsf{proj}_1\ x\ y) : \mathbf{N}$ が成り立つ。このように、型宣言 Γ 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 $(\mathsf{0}\ \mathsf{s})$ のような**型エラー**を含む式は考えない。

関数の定義法. 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。

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 f 以外の識別子を用いることができる。f の型 $\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0$ 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 Γ 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 $M_{\mathsf{true}}, M_{\mathsf{false}}$ 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 f の**定義**といい、$M_{\mathsf{true}}, M_{\mathsf{false}}$ を 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、$M_{\mathsf{0}}, M_{\mathsf{s}}$ 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_{s} の中で式 $(\mathsf{f}\ y\ \overline{x}) : D_0$ を用いてもよい。これを f の**定義**といい、$M_{\mathsf{0}}, M_{\mathsf{s}}$ を f の**定義式**という。

これまでと違い、M_{s} の中では $x_1, \ldots, x_n$ に加えて変数 y や部分式 $(\mathsf{f}\ y\ \overline{x})$ 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。もちろん $(\mathsf{f}\ (\mathsf{s}\ y)\ \overline{x})$ や $(\mathsf{f}\ y\ (\mathsf{s}\ \mathsf{s}\ y))$ の値を参照することはできない。

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、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。ここで 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。ただし例外として、f_k を原始再帰法（$\mathbf{N}$）により定義する場合には、定義式 M_{s} の中で $(\mathsf{f}_k\ y\ \overline{x})$ を用いてよい。これならば循環定義にならないからである。最後の識別子 f_n をプログラム全体の名前だと思い、プログラム P のことを**プログラム** f_n ともいう。

原始再帰的関数． 先ほど定義したプログラム $\mathsf{add} : \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N}$ は、自然数上の足し算を表す。これは集合 $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ の要素であり、具体的には 2 項関数 $f(x, y) := x + y$ をカリー化したものである。一般に、一階型 A に対して集合 $[\![A]\!]$ を次のように帰納的に割り当てる：

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

すると任意の原始再帰的プログラム $\mathsf{f} : A$ は、集合 $[\![A]\!]$ の要素を表す。これを $[\![\mathsf{f}]\!]$ と書く。すなわち

$$\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0 \quad \text{ならば} \quad [\![\mathsf{f}]\!] \in [\![D_1]\!] \to \cdots [\![D_n]\!] \to [\![D_0]\!]$$

である。命題 1.5 によれば、$[\![\mathsf{f}]\!]$ は $[\![D_1]\!] \times \cdots \times [\![D_n]\!] \to [\![D_0]\!]$ の要素である n 項関数とみなすこともできる。何らかの原始再帰的プログラム f により表される（集合論的な意味での）関数 $[\![\mathsf{f}]\!]$ を**原始再帰的関数**という。

（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。$[\![\mathsf{f}_1]\!] = [\![\mathsf{f}_2]\!]$ 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）。

2.3 原始再帰的プログラムの表現力

ここではどのような関数が原始再帰的プログラムにより表せるかを調べていく。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

and	:	$\mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{B}$	not	:	$\mathbf{B} \Rightarrow \mathbf{B}$
and true x	=	x	not true	=	false
and false x	=	false	not false	=	true
or	:	$\mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{B}$	if	:	$\mathbf{B} \Rightarrow A \Rightarrow A \Rightarrow A$
or true x	=	true	if true $x\ y$	=	x
or false x	=	x	if false $x\ y$	=	y

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。

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

次に自然数に関する例を挙げる。

2. 自然数に関するもの

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

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

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

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

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

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

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

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

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したらよいか？それには次のようなプログラムを考えればよい。

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

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

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

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

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

このコーディングが正しいことを見るには、

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

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

$$\begin{array}{lcl}
\mathsf{prm} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{prm}\ y\ x & = & \mathsf{or}\ \mathsf{neg}(\mathsf{div}\ y\ x)\ (\mathsf{eq}\ y\ 1) \\
 & & \\
\forall^b\mathsf{prm} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\
\forall^b\mathsf{prm}\ 0\ x & = & \mathsf{true} \\
\forall^b\mathsf{prm}\ (\mathsf{s}\ y)\ x & = & \mathsf{and}\ (\mathsf{prm}\ y\ x)\ (\forall^b\mathsf{prm}\ y\ x) \\
 & & \\
\mathsf{prime} & : & \mathbf{N} \Rightarrow \mathbf{B} \\
\mathsf{prime}\ x & = & \mathsf{and}\ (\mathsf{leq}\ 2\ x)\ (\forall^b\mathsf{prm}\ x\ x)
\end{array}$$

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）。

2.4 原始再帰的関数の大きさ

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 $\mathsf{ack}_0, \mathsf{ack}_1, \mathsf{ack}_2, \ldots$ を次のように帰納的に定義する。

$$\begin{array}{lcl}
\mathsf{ack}_0 & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{ack}_0\ y & = & \mathsf{s}\ y
\end{array}
\qquad\qquad
\begin{array}{lcl}
\mathsf{ack}_{n+1} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{ack}_{n+1}\ 0 & = & \mathsf{ack}_n\ 1 \\
\mathsf{ack}_{n+1}\ (\mathsf{s}\ y) & = & \mathsf{ack}_n\ (\mathsf{ack}_{n+1}\ y)
\end{array}$$

また、

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

とおく。各 ack_n は原始再帰的関数であり、

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

となることが容易に確かめられる。$n = 3$ 以降 $ack_n(y)$ の値は急激に大きくなる。だいたいの大きさを見積もると $ack_3(y) \approx 2^y$ 程度である。

この関数列を用いれば、（自然数上の）原始再帰的関数に上限を与えられる。

定理 2.1

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

$$f(m) \ < \ ack_n(m) \quad (\text{すべての } m \in \mathbb{N} \text{ について})$$

が成り立つ。

原始再帰的関数の限界. 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 2.2

アッカーマン関数

$$Ack(n, m) \quad := \quad ack_n(m) : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

は原始再帰的ではない。

上の定理が成り立つことは簡単に確かめられる。実際、Ack が原始再帰的だとしたら、$Ack'(m) := ack_m(m)$ も原始再帰的である。それゆえ定理 2.1 により、ある $n \in \mathbb{N}$ が存在し、

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

となるが、これは矛盾である。

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 ack_n を起動し、ack_n m の値を計算すれば出力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 k が存在して、(f n) の計算時間を $ack_k(n)$ 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）。

練習問題 2.3

1. 入力値 n に対して n よりも大きな最小の素数を返す原始再帰的プログラム nextprime : $\mathbf{N} \Rightarrow \mathbf{N}$ を定義せよ。

2. 入力値 n に対して n 番目の素数を返す原始再帰的プログラム nthprime : $\mathbf{N} \Rightarrow \mathbf{N}$ を定義せよ。

2.5 直積型とリスト型

自然数全体を表す型 $\mathbf{N}$ は、

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

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。たとえば

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

により自然数上の足し算 $[\![\mathsf{add}]\!] : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ を定めることができる。

これまでは $\mathbf{N}$ と $\mathbf{B}$ に話を制限してきたが、これらを使って新しい帰納的データ型を作り出すこともできる。ここでは重要な例を 2 つ挙げておく。以前と同様、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 Γ と書く。

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。これを省略して

$$\langle M_1, M_2 \rangle \ := \ \mathsf{pair}\ M_1\ M_2$$

とも書く。直積型に対応する原始再帰法は次の通り。

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_{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_{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）をつくることができる。たとえば $\langle 3, 5 \rangle : \mathbf{N} \times \mathbf{N}$ など。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上げておく。

y 番目のフィボナッチ数を出力するプログラム fib

$$
\begin{array}{lcl}
\mathsf{add}' & : & \mathbf{N} \times \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{add}'\ \langle y_1, y_2 \rangle & = & \mathsf{add}\ y_1\ y_2 \\
 & & \\
\mathsf{fib}' & : & \mathbf{N} \Rightarrow \mathbf{N} \times \mathbf{N} \\
\mathsf{fib}'\ 0 & = & \langle 0, 1 \rangle \\
\mathsf{fib}'\ (\mathsf{s}\ y) & = & \langle \mathsf{pr}_2(\mathsf{fib}'\ y),\ \mathsf{add}'(\mathsf{fib}'\ y) \rangle \\
 & & \\
\mathsf{fib} & : & \mathbf{N} \Rightarrow \mathbf{N} \\
\mathsf{fib}\ y & = & \mathsf{pr}_1(\mathsf{fib}'\ y)
\end{array}
$$

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。

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

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_{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_{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

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

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ことができる。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。

さて、一階型の解釈は直積型やリスト型へと次のように拡張することができる。

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

それゆえ

$$\mathsf{f} : D_1 \Rightarrow \cdots D_n \Rightarrow D_0 \quad \text{ならば} \quad [\![\mathsf{f}]\!] \in [\![D_1]\!] \to \cdots [\![D_n]\!] \to [\![D_0]\!]$$

となるように拡張原始再帰的プログラムを集合論的に解釈することができる。これを**拡張原始再帰的関数**と呼ぶことにしよう。こうすれば原始再帰的関数の守備範囲を直積やリストへと拡大することができる。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 2.4

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。

とくにフィボナッチ数を返す関数 $fib : \mathbb{N} \to \mathbb{N}$ は直積やリストを用いずともプログラム可能であるし、逆に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する必要がある。

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。すなわち、どんな入力値 $n \in \mathbb{N}$ を与えても、f n の計算は必ず停止する。このことを f は**全域的**であるという。

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考えるのである。

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_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_f **の中で識別子 f が用いられていても構わない**。M_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。

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

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 $\mathsf{gcd}\ x\ y$ の値を $\mathsf{gcd}\ y\ x$ と $\mathsf{gcd}\ (x - y)\ y$ 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 × 2」が減少するので、計算は必ず停止することがわかる（$x = 0$ 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**）。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

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

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（再帰のための識別子 f が定義式の中に出てこない場合）にすぎない。

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、

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

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 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_{f_k} 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。最後の識別子に着目して、P のことを**プログラム** f_n 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）。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**）といい、V, W 等により表す。

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

n は $\mathsf{s}(\mathsf{s} \cdots (\mathsf{s}\ \mathsf{0}) \cdots)$ の形の式の省略表現であることに注意。

次の再帰的プログラム $\mathsf{f} = \mathsf{f}_n$ 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 $\longrightarrow_{\mathsf{f}}$ を与える。

$$\begin{array}{lcl}
\mathsf{f}_i\ V_1\ \cdots\ V_k & \longrightarrow_{\mathsf{f}} & M_{\mathsf{f}_i}[x_1 := V_1, \ldots, x_k := V_k] \\
\mathsf{if\ true}\ M\ N & \longrightarrow_{\mathsf{f}} & M \\
\mathsf{if\ false}\ M\ N & \longrightarrow_{\mathsf{f}} & N \\
\mathsf{zero\ 0} & \longrightarrow_{\mathsf{f}} & \mathsf{true} \\
\mathsf{zero\ s(n)} & \longrightarrow_{\mathsf{f}} & \mathsf{false} \\
\mathsf{pred\ 0} & \longrightarrow_{\mathsf{f}} & \mathsf{0} \\
\mathsf{pred\ s(n)} & \longrightarrow_{\mathsf{f}} & \mathsf{n}
\end{array}$$

ただし $1 \leq i \leq n$ であり、$[x_1 := V_1, \ldots, x_k := V_k]$ は「変数 $x_1, \ldots, x_n$ に値式 $V_1, \ldots, V_k$ 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、第三引数は書き換えを行ってはならない。いわば M, N は第一引数により “ロック” された状態にある。

いま、プログラム f と値式（の列）$\overline{V} = V_1, \ldots, V_k$ を考える。式 $\mathsf{f}\ \overline{V}$ に何度か書き換えを行うことで値式 W に到達するとき、

$$\mathsf{f}\ \overline{V} = W, \qquad \mathsf{f}\ \overline{V} \Downarrow W, \qquad \mathsf{f}\ \overline{V} \Downarrow$$

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

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

と書く。（適切な型をもつ）どのような値式の列 $\overline{V}$ についても $\mathsf{f}\ \overline{V} \Downarrow$ となるとき、プログラム f は**全域再帰的**であるという。

どんな原始再帰的プログラムも全域再帰的である。diverge はもちろん全域再帰的ではない。colatz が全域再帰的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 $\bot$ を用意し、集合 X が与えられたとき、$X_\bot := X \cup \{\bot\}$ と定める（ただし $\bot \notin X$）。直感的にいって $\bot$ は「未定義」を表す。そして関数 $f : X \to Y_\bot$ は X から Y への**部分関数**を表すものと考える。すなわち $a \in X$ について $f(a) = \bot$ のときには、$f(a)$ の値は「未定義」と考えるのである。

自然数上の再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ に対して部分関数 $[\![\mathsf{f}]\!] \in \mathbb{N} \to \mathbb{N}_\bot$ を次のように定める。

$$\begin{array}{rcl} [\![\mathsf{f}]\!](m) & := & n \quad (\mathsf{f}\ \mathsf{m} = \mathsf{n}\ \text{のとき}) \\ & := & \bot \quad (\mathsf{f}\ \mathsf{m} \Uparrow\ \text{のとき}) \end{array}$$

同様にして、任意の一階型の再帰的プログラム $\mathsf{f} : \overline{D} \Rightarrow D_0$ に対して部分関数 $[\![\mathsf{f}]\!] \in \overline{[\![D]\!]} \to [\![D_0]\!]_\bot$ を定めることができる。

f を再帰的プログラムとするとき、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 $f(\overline{a}) = \bot$ とならない。このとき f を**全域再帰的関数**という（文脈により「再帰的」を「部分再帰的」といい、「全域再帰的」を「再帰的」ということもあるので、他の文献を読むときには注意が必要）。

原始再帰的関数と再帰的関数. 前章で挙げたアッカーマン関数 $Ack : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ を計算するには、次の再帰的プログラムを考えればよい。

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

このプログラムは、どんな入力が与えられても必ず計算が停止する。実際、定義式内で Ack が呼び出されるときには、第一引数と第二引数の対が辞書式順序で必ず減少しているからである。よって $Ack = [\![\mathsf{Ack}]\!]$ は全域再帰的関数である。

定理 3.1

（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）

3.1 再帰的関数の自己解釈

再帰的プログラミングは強力なプログラミング手法であり、どれくらい強力かというと、再帰的プログラムの実行方法それ自体を再帰的プログラムで記述できるほどである。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する。

各式 M は有限の文字列であり、各文字は 1 つの自然数で表せる。それゆえ同等性

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

（命題 1.6）を用いれば、どんな式も 1 つの自然数で符号化することができる。以下ではそのような符号化の方法を 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 $\urcorner M \urcorner : \mathbf{N}$ と書く。同様にすれば、プログラム f（再帰的定義の有限列）に対しても**ゲーデル数** $\ulcorner \mathsf{f} \urcorner \in \mathbb{N}$ や値式 $\urcorner \mathsf{f} \urcorner : \mathbf{N}$ を定めることができる。

さて、プログラム f を定めれば、式の書き換え規則 $\longrightarrow_{\mathsf{f}}$ 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）。

$$\mathsf{step}\, \urcorner \mathsf{f} \urcorner\, \ulcorner M \urcorner = \ulcorner N \urcorner \quad \Longleftrightarrow \quad M \longrightarrow_{\mathsf{f}} N.$$

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 $\mathsf{val2nat} : \mathbf{N} \Rightarrow \mathbf{N}$ が自然に構成できる。

$$\begin{array}{l} \mathsf{value}\, \ulcorner M \urcorner = \mathsf{true} \quad \Longleftrightarrow \quad M \text{ は値式} \\ \mathsf{val2nat}\, \ulcorner \mathsf{n} \urcorner = \mathsf{n} \end{array}$$

最後に、プログラム f のゲーデル数と値式 m が与えられたら、式 $\mathsf{f}\ \mathsf{m}$ 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。

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

これらのプログラムが与えられれば、次のように再帰的プログラム eval を定義することができる。

$$\begin{array}{lcl} \mathsf{eval}' & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{eval}'\ x\ y & = & \mathsf{if}\ (\mathsf{value}\ y)\ \mathsf{then}\ (\mathsf{val2nat}\ y)\ \mathsf{else}\ \mathsf{eval}'\ x\ (\mathsf{step}\ x\ y) \\ \\ \mathsf{eval} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{eval}\ x\ z & = & \mathsf{eval}'\ x\ (\mathsf{init}\ x\ z) \end{array}$$

プログラム eval は、どんな再帰的プログラム f 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。

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

このプログラムが表す再帰的関数を

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

とおく。以上の準備のもとで再帰的関数の**標準形定理**を述べることができる。

定理 3.2

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。

実際 f は再帰的関数なので、f を表す再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ が存在する。そこで $e = \ulcorner \mathsf{f} \urcorner$ とすれば、

$$\begin{aligned} eval(\ulcorner \mathsf{f} \urcorner, m) = n &\iff \mathsf{eval}\, \ulcorner \mathsf{f} \urcorner\, \mathsf{m} = \mathsf{n} \\ &\iff \mathsf{f}\, \mathsf{m} = \mathsf{n} \\ &\iff f(m) = n. \end{aligned}$$

$eval(\ulcorner \mathsf{f} \urcorner, m) = \perp \iff f(m) = \perp$ も同様に示せる。

上では型 $\mathbf{N} \Rightarrow \mathbf{N}$（関数集合 $\mathbb{N} \to \mathbb{N}_\perp$）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）。

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（逆にこのことをもって “計算可能” を定義してもよい）。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、このテーゼはコンピュータ科学者の間で広く受け入れられている。

3.2 再帰的枚挙可能集合と決定可能集合

$k \geq 1$ 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**再帰的枚挙可能集合**（または**半決定可能集合**）という。

簡単のため $k = 1$ とし、$R = R_{\mathsf{f}}$ を再帰的枚挙可能集合とする。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。このように再帰的枚挙可能集合には、計算の停止・非停止に関して非対称性がある。

一方で 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

$$\text{(決定可能集合全体)} \quad \subseteq \quad \text{(再帰的枚挙可能集合全体)}$$

となる。

たとえば

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

とすると、$Prime$ 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。

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

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。

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

この集合は再帰的枚挙可能である。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。整数解が存在しなければこのプログラムは永遠に停止しないので、全域的ではない。

一方、後で述べるとおり DS は決定可能ではない。

再帰的枚挙可能という名前の由来は次の定理にある。

定理 3.3

空でない集合 $R \subseteq \mathbb{N}$ が再帰的枚挙可能であることと、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。

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

つまり再帰的枚挙可能集合 R は全域再帰的な f を用いて

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

と枚挙できるということである（ただし同じ要素が複数回現れる可能性がある）。

再帰的枚挙可能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、

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

と定義する。

定理 3.4

$R \subseteq \mathbb{N}^{k+1}$ が再帰的枚挙可能ならば、$\exists R \subseteq \mathbb{N}^k$ も再帰的枚挙可能である。
$R \subseteq \mathbb{N}^k$ が決定可能ならば、$\mathbb{N}^k - R$ も決定可能である。

再帰的枚挙可能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 3.5

$R \subseteq \mathbb{N}^k$ が決定可能であることと、R と $\mathbb{N}^k - R$ がともに再帰的枚挙可能であることは一致する。

決定可能 $\Rightarrow$ 再帰的枚挙可能は明らか。逆方向を示すには、定理 3.3 が便利である。R と $\mathbb{N} - R$ を枚挙する全域再帰的プログラムをそれぞれ f, g とする。すなわち

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

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 $R = R_{\mathsf{h}}$ であり、なおかつ h は全域再帰的である。実際、どんな $n \in \mathbb{N}$ についても f m = n または g m = n となる $m \in \mathbb{N}$ が必ず存在する。しかも、f も 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 h y = 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、この集合は再帰的枚挙可能である。そこで上の定理によれば、もし次のようなプログラム dsnot : $\mathbf{N} \Rightarrow \mathbf{B}$ が存在すれば、DS が決定可能だと言えることになる。

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

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 dsnot）が書けないといった事態である。後者が書けるかどうか、それが再帰的枚挙可能と決定可能を分ける境目なのである。

3.3 決定不能集合

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

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

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

次の集合は再帰的枚挙可能だが決定可能ではない集合の具体例である。

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

定理 3.6

集合 Halt は再帰的枚挙可能ではあるが決定可能ではない。

証明は以下の通りである。まず再帰的枚挙可能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 evalb : $\mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{B}$ を構成する。

$$\begin{aligned} \mathsf{evalb}\,\ulcorner \mathsf{f} \urcorner\ \mathsf{m} = \mathsf{true} &\iff \mathsf{f\ m} = \mathsf{true} \\ \mathsf{evalb}\,\ulcorner \mathsf{f} \urcorner\ \mathsf{m} = \mathsf{false} &\iff \mathsf{f\ m} = \mathsf{false} \\ \mathsf{evalb}\,\ulcorner \mathsf{f} \urcorner\ \mathsf{m} \Uparrow &\iff \mathsf{f\ m} \Uparrow \end{aligned}$$

これを用いて

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

とすれば計算の停止を半決定することができる。

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

ゆえに $Halt$ は再帰的枚挙可能である。

次に $Halt$ が決定可能でないことを背理法により示す。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。

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

ここから矛盾を導くために、次のプログラム diag を考える。

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

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、

$$\begin{aligned} \mathsf{diag}\,\ulcorner \mathsf{diag} \urcorner = \mathsf{true} &\iff \mathsf{haltd}\,\ulcorner \mathsf{diag} \urcorner\,\ulcorner \mathsf{diag} \urcorner = \mathsf{false} \\ &\iff \mathsf{diag}\,\ulcorner \mathsf{diag} \urcorner \Uparrow . \end{aligned}$$

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。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 3.6 により R は決定不能であることが従う。

定理 3.7

集合

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

は再帰的枚挙可能だが、決定可能ではない。

再帰的枚挙可能性は明らかなので、決定不能性を証明しよう。まず再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{B}$ と自然数 $n \in \mathbb{N}$ が与えられたとき、再帰的プログラム

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

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、

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

となる。そこでプログラム

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

を考えると、これは**全域**再帰的であり、しかも

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

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 3.6 と矛盾する。

最後にもう 1 つだけ還元法の例を挙げよう。まず、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

$$\{n \in \mathbb{N} : p(n, \vec{y}) = 0 \text{ は整数解を持つ } \}$$

は再帰的枚挙可能であることに注意する。実際、（自然数を使って整数を符号化すれば）整数の足し算、掛け算、等号はいずれも原始再帰的なので、整数の組 $(n, \vec{m})$ 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 3.4（の整数版）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 3.8

どんな再帰的枚挙可能集合 $R \subseteq \mathbb{N}$ に対してもあるディオファントス多項式 $p_R(x, \vec{y})$ が存在し、

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

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。この定理により $\mathit{Halt0}$ を DS に還元することができる。

定理 3.9

集合 DS は再帰的枚挙可能であるが決定可能ではない。

再帰的枚挙可能であることはすでに見た。決定不能性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、**全域**再帰的なプログラム $\mathsf{dsd} : \mathbf{N} \Rightarrow \mathbf{B}$ が存在し、

$$\begin{array}{rcl} \mathsf{dsd}\ulcorner p(\vec{x})\urcorner & = & \mathsf{true} \quad (p(\vec{x}) = 0 \text{ は整数解を持つ}) \\ & = & \mathsf{false} \quad (p(\vec{x}) = 0 \text{ は整数解を持たない}) \end{array}$$

が成り立つ。さて、定理 3.7 により集合 $\mathit{Halt0} \subseteq \mathbb{N}$ は再帰的枚挙可能である。それゆえ定理 3.8 により、対応するディオファントス多項式 $p_{\mathit{Halt0}}(x, \vec{y})$ 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。

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

そこでプログラム

$$\begin{array}{lcl} \mathsf{halt0'} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{halt0'}\ x & = & \mathsf{dsd}(\mathsf{p}\ x) \end{array}$$

を考えると、これは**全域**再帰的であり、しかも

$$\begin{array}{rcl} \mathsf{halt0'\ n} = \mathsf{true} & \Longleftrightarrow & \mathsf{dsd}(\mathsf{p\ n}) = \mathsf{true} \\ & \Longleftrightarrow & \mathsf{dsd}\ulcorner p_{Halt0}(n, \vec{y})\urcorner = \mathsf{true} \\ & \Longleftrightarrow & p_{Halt0}(n, \vec{y}) = 0 \text{ は整数解を持つ} \\ & \Longleftrightarrow & n \in \mathit{Halt0} \end{array}$$

となる。これは $\mathit{Halt0}$ が決定可能であることを示しているが、定理 3.7 と矛盾する。

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表せる関数のことである。本講義では、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）。

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

また、同等性 $\mathbb{N} \to \mathbb{B} \cong \wp(\mathbb{N})$（命題 1.4）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枚挙可能集合が対応する。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）。

$$\text{（決定可能集合全体）} \quad \subsetneq \quad \text{（再帰的枚挙可能集合全体）} \quad \subsetneq \quad \wp(\mathbb{N})$$

「計算可能関数＝（全域）再帰的関数」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。また、決定可能・決定不能の区別も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概念である。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

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

ということであるが、これはゲーデル数を用いれば、初等数論の文で表現可能である。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 T をとっても、全域性が証明できない再帰的プログラム 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停止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。本講義では計算「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。

- 本講義では、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、計算可能性・決定可能性は大前提とした上で、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。本講義で計算理論に興味を持った方は、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、本講義の続きにご期待いただきたい。

練習問題 4.1

1. 集合 $\mathbb{R} \cup \{\infty, -\infty\}$ が $\mathbb{R}$ と同等であることを示せ（両者間に全単射があることを示せ）。

2. 自然数の有限列が与えられたらその平均値を返す関数 $mean : \mathbb{N}^* \to \mathbb{N}$

$$mean(n_1, \ldots, n_k) = \frac{1}{k}(n_1 + \cdots + n_k), \qquad mean() = 0$$

   が原始再帰的関数であることを示せ（小数点以下切り捨て）。

3. 次の集合 $ZZ \subseteq \mathbb{N}$ が決定不能であることを示せ。

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

解答.

1. 包含写像 $i : \mathbb{R} \to \mathbb{R} \cup \{\infty, -\infty\}$ は単射である。また、写像 $f : \mathbb{R} \cup \{\infty, -\infty\} \to \mathbb{R}$ を

$$\begin{aligned} f(x) &= \arctan x \quad (x \in \mathbb{R} \text{ のとき}) \\ f(-\infty) &= -\pi/2 \\ f(\infty) &= \pi/2 \end{aligned}$$

により定めれば、これが単射になることは明らか（確かめよ）。両方向に単射があるので、カントール・ベルンシュタインの定理により $\mathbb{R}$ と $\mathbb{R} \cup \{\infty, -\infty\}$ は同等である。

2. たとえば次の原始再帰的プログラムを考えればよい（適宜省略表現を用いる）。

$$\begin{array}{lcl} \mathsf{length} & : & \mathbf{L}[A] \Rightarrow \mathbf{N} \\ \mathsf{length\ nil} & = & 0 \\ \mathsf{length}\ (\mathsf{cons}\ a\ y) & = & (\mathsf{length}\ y) + 1 \\ \\ \mathsf{sum} & : & \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{N} \\ \mathsf{sum\ nil} & = & 0 \\ \mathsf{sum}\ (\mathsf{cons}\ a\ y) & = & (\mathsf{sum}\ y) + a \\ \\ \mathsf{div}' & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{div}'\ 0\ y\ z & = & z \\ \mathsf{div}'\ (\mathsf{s}\ x)\ y\ z & = & \mathsf{if}\ (z - x) \cdot y > z\ \mathsf{then}\ (z - x - 1)\ \mathsf{else}\ (\mathsf{div}'\ x\ y\ z) \\ \\ \mathsf{div} & : & \mathbf{N} \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{div}\ y\ z & = & \mathsf{if}\ y = 0\ \mathsf{then}\ 0\ \mathsf{else}\ (\mathsf{div}'\ z\ y\ z) \\ \\ \mathsf{mean} & : & \mathbf{L}[\mathbf{N}] \Rightarrow \mathbf{N} \\ \mathsf{mean}\ x & = & \mathsf{div}\ (\mathsf{length}\ x)\ (\mathsf{sum}\ x) \end{array}$$

3. *Halt0* を *ZZ* に還元する。まず再帰的プログラム $\mathsf{f} : \mathbf{N} \Rightarrow \mathbf{N}$ が与えられたとき、再帰的プログラム

$$\begin{array}{lcl} \mathsf{zf} & : & \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{zf}\ x & = & (\mathsf{f}\ x) \cdot 0 \end{array}$$

を構成する。すると明らかに

$$\mathsf{zf}\ 0 = 0 \quad \Longleftrightarrow \quad \mathsf{f}\ 0 \Downarrow$$

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 $\mathsf{z} : \mathbf{N} \Rightarrow \mathbf{N}$ で

$$\mathsf{z}\ulcorner\mathsf{f}\urcorner = \ulcorner\mathsf{zf}\urcorner$$

を満たすものが存在する。

仮に *ZZ* が決定可能だとすると、全域再帰的プログラム $\mathsf{zz} : \mathbf{N} \Rightarrow \mathbf{B}$ が存在して

$$\begin{array}{lcl} \mathsf{zz}\ulcorner\mathsf{f}\urcorner & = & \mathsf{true} \quad (\mathsf{f}\ 0 = 0 \text{ のとき}) \\ & = & \mathsf{false} \quad (\text{それ以外のとき}) \end{array}$$

となるはずである。そこでプログラム

$$\begin{array}{lcl} \mathsf{halt0''} & : & \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{halt0''}\ x & = & \mathsf{zz}(\mathsf{z}\ x) \end{array}$$

を考えればこれは全域再帰的であり、

$$\begin{array}{lcl} \mathsf{halt0''}\ulcorner\mathsf{f}\urcorner = \mathsf{true} & \Longleftrightarrow & \mathsf{zz}(\mathsf{z}\ulcorner\mathsf{f}\urcorner) = \mathsf{true} \\ & \Longleftrightarrow & \mathsf{zz}\ulcorner\mathsf{zf}\urcorner = \mathsf{true} \\ & \Longleftrightarrow & \mathsf{zf}\ 0 = 0 \\ & \Longleftrightarrow & \mathsf{f}\ 0 \Downarrow \end{array}$$

が成り立つ。これは集合 *Halt0* が決定可能なことを示しており、定理 3.7 と矛盾する。