コンピュータサイエンス基礎(2018)・講義資料

(星野担当)

1　クリーネの第一代数

自然数 $n, m \in \mathbb{N}$ に対して $n * m$ を

$$n * m = \mathrm{eval}(n, m)$$

と定義する．(ここではローマン体で eval と書いているが $eval$ のことである．あまり気にしないでほしい．) この 2 項演算 $n * m$ が付与された自然数の集合 $\mathbb{N}$ はクリーネの第一代数と呼ばれる．代数とは言っても数学でよく現れる群や環のような扱いやすい性質は持っていない．

練習問題 1. $n * m \neq m * n$ となる自然数 n, m を与えよ．

しばらくはクリーネの第一代数についての性質を標準形定理を元に議論していく．標準形定理とは以下の主張であった．

定理 1. どのような再帰的関数 $f\colon \mathbb{N} \to \mathbb{N}_\perp$ についてもある自然数 e が存在して，全ての $n \in \mathbb{N}$ で

$$f(n) = e * n$$

が成り立つ．

読みやすさのために以降では

$$n_0 * n_1 * n_2 * \cdots * n_k = (\cdots((n_0 * n_1) * n_2) \cdots) * n_k$$

という略記をする．この略記に意味を持たせるため $(-) * (-)$ を以下のようにして $\mathbb{N}_\perp$ 上の二項演算に拡張しておく．

$$\perp * n = \perp, \qquad n * \perp = \perp, \qquad \perp * \perp = \perp.$$

この拡張の元では

$$n_0 * n_1 * n_2 * \cdots * n_k = [\![\mathsf{eval}(\cdots \mathsf{eval}(\mathsf{eval}(n_0, n_1), n_2) \cdots, n_k)]\!]$$

が成り立つ．特に $k+1$ 項演算 $(-) * (-) * (-) * \cdots * (-)$ は帰納的関数である．

クリーネの第一代数の重要な性質である標準形定理の一般化と不動点演算子の存在を示す．まず再帰的関数は以下の意味でカリー化が可能である．

補題 1. 任意の再帰的関数 $f\colon \mathbb{N}^{k+1} \to \mathbb{N}_\perp$ について，ある全域再帰的関数 $g\colon \mathbb{N}^k \to \mathbb{N}$ が存在して，全ての $n_1, \ldots, n_k, m \in \mathbb{N}$ で

$$f(n_1, \ldots, n_k, m) = g(n_1, \ldots, n_k) * m$$

が成り立つ．

証明. 再帰的プログラム $\mathsf{f}\colon \mathbf{N}^{k+1} \Rightarrow \mathbf{N}$ を $[\![\mathsf{f}]\!] = f$ を満たすものとし，再帰的関数 $g\colon \mathbb{N}^{k+1} \to \mathbb{N}_\perp$ を

$$g(n_1, \ldots, n_k) = \ulcorner \mathsf{f}\, \mathsf{n}_1 \cdots \mathsf{n}_k \urcorner$$

と定義する．定義から明らかに g は全域で定義されており $f(n_1, \ldots, n_k, m) = g(n_1, \ldots, n_k) * m$ を満たす． □

この補題を用いることで標準形定理の一般化および不動点演算子が得られる．

定理 2. 再帰的関数 $f\colon \mathbb{N}^{k+1} \to \mathbb{N}_\perp$ に対しある $e \in \mathbb{N}$ が存在して，全ての $n_1, \ldots, n_{k+1} \in \mathbb{N}$ について

- $e * n_1 * n_2 * \cdots * n_k \neq \perp$
- $e * n_1 * n_2 * \cdots * n_{k+1} = f(n_1, \ldots, n_{k+1})$

が成り立つ．

これまでは自然数を表すのに n, m, k, l などの文字を使ってきたがこれからはそれに加えて下線付きのローマン体で書かれた単語も使うことにする．例えば $\underline{\mathrm{zero}}$ や $\underline{\mathrm{code}}$ などである．

定理 3 (不動点演算子)**.** 以下の性質を満たす $\underline{\mathrm{fix}} \in \mathbb{N}$ が存在する．

- 任意の $n \in \mathbb{N}$ に対し $\underline{\mathrm{fix}} * n \neq \perp$.
- 任意の $n, m \in \mathbb{N}$ に対し $\underline{\mathrm{fix}} * n * m = n * (\underline{\mathrm{fix}} * n) * m$.

証明. 定理 2 より自然数 $e \in \mathbb{N}$ で

- 全ての $n, m \in \mathbb{N}$ について $e * n * m \neq \perp$ かつ
- 全ての $n, m, l \in \mathbb{N}$ について $e * n * m * l = n * (m * m) * l$

をみたすものが存在する．次に標準形定理を使うことで自然数 $\underline{\mathrm{fix}} \in \mathbb{N}$ で

$$\underline{\mathrm{fix}} * n = (e * n) * (e * n) \neq \perp$$

を満たすものが得られる．自然数 e の取り方から任意の $m \in \mathbb{N}$ で

$$\begin{aligned} \underline{\mathrm{fix}} * n * m &= (e * n) * (e * n) * m \\ &= n * ((e * n) * (e * n)) * m \\ &= n * (\underline{\mathrm{fix}} * n) * m \end{aligned}$$

が成り立つ． □

この主張は任意の $n \in \mathbb{N}$ について

$$x \text{ 番目の再帰的関数} = n * x \text{ 番目の再帰的関数}$$

を満たす $x \in \mathbb{N}$ が存在することを言っている．大雑把には

$$n * (-) \colon \{ \text{ 再帰的関数 (のゲーデル数) の集合 } \} \to \{ \text{ 再帰的関数 (のゲーデル数) の集合 } \}$$

が不動点 (fixed point) を持つということである．

不動点演算子を使って Rice の定理を証明してみよう．自然数 e に対し再帰的関数 $e * (-) \colon \mathbb{N} \to \mathbb{N}_\perp$ を φ_e と書くことにする．

定理 4. 集合 F を $\mathrm{REC} = \{f \colon \mathbb{N} \to \mathbb{N}_\perp : f \text{ は再帰的関数 } \}$ の部分集合とする．このとき以下は同値：

1. $F = \emptyset$ または $F = \mathrm{REC}$
2. ある $e \in \mathbb{N}$ が存在して任意の $n \in \mathbb{N}$ で

$$\varphi_n \in F \iff e * n = 0$$
$$\varphi_n \notin F \iff e * n = 1.$$

決定可能な再帰的関数の集合は自明なものに限るという主張である．

証明. (1) $\Longrightarrow$ (2) は明らか．逆を示す．(2) が成り立っているとして (1) の否定から矛盾を導く．部分集合 F が非自明と仮定すると，標準形定理から $\varphi_{e_0} \in F$ と $\varphi_{e_1} \notin F$ を満たすある自然数 e_0 と e_1 が得られる．関数 $h \colon \mathbb{N} \to \mathbb{N}$ を

$$h(n) = \begin{cases} e_0, & (\varphi_n \notin F), \\ e_1, & (\varphi_n \in F) \end{cases}$$

と定義するとこれは仮定から帰納的であり，$h = \varphi_e$ となる $e \in \mathbb{N}$ が存在する．しかし，

$$\varphi_{\underline{\mathrm{fix}} * e} \in F \overset{\underline{\mathrm{fix}} \text{ の性質}}{\iff} \varphi_{e * (\underline{\mathrm{fix}} * e)} \in F \overset{h \text{ の定義}}{\iff} \varphi_{\underline{\mathrm{fix}} * e} \notin F$$

となり矛盾する． □

2 Lawvere の不動点定理

不動点演算子 $\underline{\mathrm{fix}}$ と Lawvere の不動点定理の関係について触れておく．Lawvere の不動点定理はある状況における関数の不動点の存在を主張する定理で，その証明はラッセルのパラドックスやカントールの対角線論法などの自己言及的な議論に共通する構造をうまく抜き出している．ここでは $\underline{\mathrm{fix}}$ の簡易版を導入し，Lawvere の不動点定理の証明における不動点の構成がこの $\underline{\mathrm{fix}}$ の簡易版の構成と本質的に同じものであることを観察する．

定理 5 (Lawvere の不動点定理)**.** 集合 A, B と関数 $\theta\colon A \times A \to B$ が以下を満たすとする：任意の関数 $f\colon A \to B$ に対してある $a_f \in A$ が存在して $f(-) = \theta(a_f, -)$．この時，どのような関数 $g\colon B \to B$ も不動点を持つ．

証明. 関数 $h\colon A \to B$ を $h(a) = g(\theta(a,a))$ と定義する．仮定からある a_h が存在して $h(-) = \theta(a_h, -)$ である．すると $b = \theta(a_h, a_h)$ とおけば

$$g(b) = g(\theta(a_h, a_h)) = h(a_h) = \theta(a_h, a_h) = b.$$

□

実際には B が二つ以上の要素を持ってしまうと不動点をもたない $g\colon B \to B$ が必ず存在するので，主張の条件を満たす θ が存在するのは B が一点集合の場合のみであり「不動点が存在する」という主張は紛らわしいかもしれない．このような主張にしたのは実際の Lawvere の不動点定理がより一般的な状況を考えていて，そこでは不動点が存在するという主張は意味を持っているからである．

Lawvere の不動点定理の証明と $\underline{\text{fix}}$ の関係をみるために $\underline{\text{fix}}$ よりも簡単な不動点の構成を与える．自然数 n を一つ固定し，$n * d = d$ を満たす自然数 d を構成しよう．まず定理 2 から全ての $m \in \mathbb{N}$ で

$$e * m = n * (m * m) \tag{1}$$

を満たす $e \in \mathbb{N}$ が存在することがわかる．すると自然数 d を

$$d = e * e \tag{2}$$

と定義すれば

$$d = e * e = n * (e * e) = n * d$$

となって $n * (-)$ の不動点 d が構成できた . . . と言いたいが実際には $e * e = \bot$ となっている可能性がある．ここでの d の定義は $\underline{\text{fix}}$ の構成とおおよそ同じであることはわかって頂けると思うので，運良く $d = e * e \neq \bot$ であったと仮定して話を進める．(ちなみに $\underline{\text{fix}}$ の構成がやや複雑なのは $e * e = \bot$ という状況を確実に排除するためである．定理 3 の最初の項目が $e * e \neq \bot$ に相当する．)

では Lawvere の不動点定理の証明とこの d の構成が本質的に同じものであることを見よう．そのために $A = A \Rightarrow B$ を満たす集合の存在を仮定する．実際にはこのような集合は自明な場合を除きあり得ないのだが,「集合」や「関数」といった概念を適切に別のものに取り替えることで数学的正当化は可能である．ここではとりあえず $A = A \Rightarrow B$ を満たす集合があるとして話を進める．容易に

$$\theta(a, a') = a(a')$$

で与えられる関数 $\theta\colon A \times A \to B$ が Lawvere の不動点定理の条件を満たしていることが確認できるので，任意の関数 $g\colon B \to B$ が不動点を持つことがわかる．大事なのは g の不動点の構成である．Lawvere の不動点定理の証明を見るとまず

$$h(a) = g(\theta(a,a)) = g(a(a)) \tag{3}$$

と関数 $h\colon A \to B$ を定義している．$h(-) = \theta(h, -)$ なので Lawvere の不動点定理で構成されている g の不動点 b は

$$b = \theta(h, h) = h(h) \tag{4}$$

で与えられる．雰囲気を出すために $x(y)$ を $x * y$ と書くことにすると h と b の定義は $h * a = g * (a * a)$ と $b = h * h$ と書き直せ，(3) と (1) が対応していて，(4) と (2) が対応していることがわかる．

3　不動点演算子を使った “プログラミング”

不動点演算子 $\underline{\text{fix}}$ の応用として $\underline{\text{fix}}$ を使った再帰的関数の構成を紹介する．この章では算術演算を $\mathbb{N}_\perp$ 上の演算に自然に拡張している．例えば掛け算 $x \cdot y$ は x と y のいずれかが $\perp$ である場合には $x \cdot y = \perp$ とする．

階乗　階乗を計算する関数 $\text{fact}(n) = n \cdot (n-1) \cdots 2 \cdot 1$ が

$$(F(f))(n) = \begin{cases} 1, & (n = 0) \\ n \cdot f(n-1), & (n > 0) \end{cases}$$

で定義される関数 $F\colon (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ の不動点であることを利用する．定理 2 より自然数 $\underline{\text{fact0}}$ で任意の $r, n \in \mathbb{N}$ で $\underline{\text{fact0}} * k \neq \perp$ かつ

$$\underline{\text{fact0}} * r * n = \begin{cases} 1, & (n = 0) \\ n \cdot (r * (n-1)), & (n > 0) \end{cases}$$

を満たすものが存在する．自然数 $\underline{\text{fact}}$ を $\underline{\text{fact}} = \underline{\text{fix}} * \underline{\text{fact0}}$ と定義すれば任意の自然数 m について

$$\underline{\text{fact}} * 0 = \underline{\text{fact0}} * \underline{\text{fact}} * 0 = 1$$
$$\underline{\text{fact}} * (n+1) = \underline{\text{fact0}} * \underline{\text{fact}} * (n+1) = n \cdot (\underline{\text{fact}} * n)$$

が成り立つ．よってすべての $n \in \mathbb{N}$ で $\underline{\text{fact}} * n = \text{fact}(n)$ である．

最大公約数　2 変数の再帰的関数

$$\gcd(x, y) = x \text{ と } y \text{ の最大公約数}$$

も階乗の定義と同じアイデアで不動点演算子を利用して再定義できる．定理 2 より自然数 $\underline{\text{gcd0}} \in \mathbb{N}$ で任意の $r, n, m \in \mathbb{N}$ について

$$\underline{\text{gcd0}} * r * n \neq \perp,$$
$$\underline{\text{gcd0}} * r * n * m = \begin{cases} n, & (n = m) \\ r * m * n, & (n < m) \\ r * (n-m) * m, & (n > m) \end{cases}$$

を満たすものが存在する．自然数 $\underline{\mathrm{gcd}}$ を $\underline{\mathrm{fix}} * \underline{\mathrm{gcd0}}$ と定義すれば任意の $n, m \in \mathbb{N}$ について

$$\underline{\mathrm{gcd}} * n * m = \underline{\mathrm{gcd0}} * \underline{\mathrm{gcd}} * n * m = \begin{cases} n, & (n = m) \\ \underline{\mathrm{gcd}} * m * n, & (n < m) \\ \underline{\mathrm{gcd}} * (n - m) * m, & (n > m) \end{cases}$$

が成り立つことが $\underline{\mathrm{gcd0}}$ の定義から分かる．この $\underline{\mathrm{gcd}} * n * m$ が n と m の最大公約数となっていることは最大公約数を計算するプログラム gcd を定義した時と同じ議論により確認できる．

総和　全域再帰的プログラムのゲーデル数 n と自然数 m を受け取り総和

$$\sum_{i=0}^{m} n * i$$

を計算する再帰的関数を定義しよう．定理 2 から自然数 $\underline{\mathrm{sum0}} \in \mathbb{N}$ で任意の $r, n, m \in \mathbb{N}$ について $\underline{\mathrm{sum0}} * r * n \neq \bot$ かつ

$$\underline{\mathrm{sum0}} * r * n * m = \begin{cases} n * 0, & (m = 0) \\ n * m + r * e * (m - 1), & (m > 0) \end{cases}$$

を満たすものが存在する．自然数 $\underline{\mathrm{sum}}$ を $\underline{\mathrm{fix}} * \underline{\mathrm{sum0}}$ とすれば任意の全域再帰的プログラムのゲーデル数 n と自然数 m について

$$\underline{\mathrm{sum}} * n * m = \underline{\mathrm{sum0}} * \underline{\mathrm{sum}} * n * m = \begin{cases} n * 0, & (m = 0) \\ n * m + \underline{\mathrm{sum}} * n * (m - 1), & (m > 0) \end{cases}$$

が成り立つ．自然数 n が全域再帰的プログラムのゲーデル数であるので $\underline{\mathrm{sum}} * n * m \neq \bot$ であり，その値は $\sum_{i=0}^{m} n * i$ となる．

自己判定プログラム　実用性はないが以下のようなプログラムの存在が示せる．

命題 1. 再帰的プログラム $\mathsf{myself} : \mathbf{N} \Rightarrow \mathbf{N}$ で

$$\mathsf{myself\ n} = \begin{cases} 0, & (\mathsf{n} = \ulcorner \mathsf{myself} \urcorner) \\ 1, & (\mathsf{n} \neq \ulcorner \mathsf{myself} \urcorner) \end{cases}$$

を満たすものが存在する．

証明. 再帰的関数 eval の定義の詳細は述べられていなかったが，ここでは $\mathrm{eval}(e, n)$ は e が再帰的プログラムのゲーデル数でない場合には 0 を返すように定義されていると仮定する．これまでの議論は eval の具体的な定義には依存していないので例えば定理 2 はそのまま使える．定理 2 から自然数 $\underline{\mathrm{myself0}} \in \mathbb{N}$ で任意の自然数 $n, m \in \mathbb{N}$ について

$$\underline{\mathrm{myself0}} * n * m = \begin{cases} 0, & (n = m) \\ 1, & (n \neq m) \end{cases}$$

を満たすものが存在する．自然数 $\underline{\text{myself}}$ を $\underline{\text{fix}} * \underline{\text{myself0}}$ と定義すると，任意の $n \in \mathbb{N}$ で

$$\underline{\text{myself}} * n = \underline{\text{myself0}} * \underline{\text{myself}} * n = \begin{cases} 0, & (n = \underline{\text{myself}}) \\ 1, & (n \neq \underline{\text{myself}}) \end{cases} \tag{5}$$

が成り立つ．もし $\underline{\text{myself}}$ が再帰的プログラムのゲーデル数でないなら常に $\underline{\text{myself}} * n = 0$ であり (5) と矛盾する．よってある再帰的プログラム myself が存在して $\underline{\text{myself}} =$「myself」である．標準形定理から myself n は n が myself のゲーデル数の時に 0 を返し，それ以外の場合は 1 を返す． □

4 計算可能関数の連続性

前章では $\underline{\text{fix}}$ を使った “プログラミング” をいくつか紹介した．そこでは $\underline{\text{fix}}$ の性質である

$$\underline{\text{fix}} * n * m = n * (\underline{\text{fix}} * n) * m$$

を使うことで $\underline{\text{fix}}$ を用いて定義された再帰的関数の計算ができた．しかし場合によってはこの等式だけからは再帰的関数の計算結果が分からない場合もある．例えばすべての $n \in \mathbb{N}$ について $\underline{\text{id}} * n = n$ を満たす $\underline{\text{id}} \in \mathbb{N}$ については

$$\underline{\text{fix}} * \underline{\text{id}} * n = \underline{\text{id}} * (\underline{\text{fix}} * \underline{\text{id}}) * n = \underline{\text{fix}} * \underline{\text{id}} * n$$

となりこれだけでは $\underline{\text{fix}} * \underline{\text{id}} * n$ が未定義なのかどうかすら分からない．この章の目標は

- “計算可能な高階関数” が単調でありさらに連続であることを示す．
- 連続性から最小不動点の存在を示す．
- 最小不動点の最小性から再帰的関数の計算結果を知ることができることを理解する．

である．

4.1 計算可能な高階関数

高階関数とは関数の集合を定義域に持つ関数のことであった．例えば

$$\text{sum}(f\colon \mathbb{N} \to \mathbb{N}_\perp) = f(0) + f(1) + f(2)$$

は $\mathbb{N} \to \mathbb{N}_\perp$ から $\mathbb{N}_\perp$ への高階関数である．集合 REC は

$$\text{REC} = \{f\colon \mathbb{N} \to \mathbb{N}_\perp : f \text{ は再帰的 }\}$$

と定義されていた．ここでは REC 上の高階関数の計算可能性について考える．（いちいち「高階」関数と普通の関数を区別して呼ぶと面倒なのでこれからはどちらも単に関数と呼ぶ．）いろいろな計算可能性の定義が考えられるが，ここでは次の定義を採用する．

定義 1. 関数 $F\colon \mathrm{REC} \to \mathrm{REC}$ についてある自然数 $e \in \mathbb{N}$ が存在しすべての $n \in \mathbb{N}$ で

$$F(\varphi_n) = \varphi_{e*n}$$

が成り立つとき F は**計算可能**であるという．またこのとき F は e によって**実現される**という．

この定義は再帰的関数 $f \in \mathrm{REC}$ を計算する再帰的プログラムのゲーデル数から $F(f)$ を計算する再帰的プログラムのゲーデル数を計算する再帰的プログラムがあるときに F が計算可能であるということをいっている．多くの場合，なんとなく計算できそうと感じる関数 $F\colon \mathrm{REC} \to \mathrm{REC}$ は計算可能である．計算可能でない関数の例としては

$$\mathrm{Halt}(u)(n) = \begin{cases} 1, & (u(0) = \bot), \\ 0, & (u(0) \neq \bot) \end{cases}$$

で定義される $\mathrm{Halt}\colon \mathrm{REC} \to \mathrm{REC}$ が挙げられる．Halt が計算可能でないのは停止性問題が決定可能でないからである．また e はプログラムのゲーデル数を参照して計算を行うが，

$$\varphi_n = \varphi_m \implies \varphi_{e*n} = \varphi_{e*m}$$

を満たしていないといけない．

以降の目標は REC 上の計算可能な関数は全て単調かつ極限を保つことを示すことである．この主張の意味を説明するためにまず集合 REC 上の二項関係 $\leq$ を

$$f \leq g \iff \text{全ての } n \in \mathbb{N} \text{ について, } f(n) \neq \bot \text{なら } f(n) = g(n)$$

と定義する．再帰的関数 $f, g \in \mathrm{REC}$ が $f \leq g$ かつ $f \neq g$ を満たす時には $f < g$ と書き，また $f \leq g$ ではない時には $f \not\leq g$ と書くことにする．自然数の間の大小関係とは異なり $f \not\leq g$ だからといって $g < f$ とは限らない．また $f \leq g$ は各点ごとの自然数の大小比較ではないことに注意．例えば

$$\mathrm{succ}(n) = n + 1, \qquad \mathrm{id}(n) = n$$

の 2 つの再帰的関数については $\mathrm{id} \not\leq \mathrm{succ}$ である．直感的には $f \leq g$ は g のほうが f よりも多くの情報を持っていることを意味している．実際 f や g から情報を取り出すには $f(0), f(1), \ldots$ や $g(0), g(1), \ldots$ などを計算するしかないが (本当だろうか？) $f \leq g$ の時には各 $n \in \mathbb{N}$ で，

- $f(n)$ は未定義だが $g(n)$ は定義されている
- $f(n)$，$g(n)$ ともに定義され $f(n) = g(n)$

の 2 パターンしかなく f から得られる情報 (計算結果) は必ず g からも得られる．「$f(n)$ は未定義」ということ自体も情報のように思えるがそうではない．停止性問題の決定不可能性によって f を計算するプログラムのゲーデル数から「$f(n)$ は未定義」かどうかの情報を抽出するプログラムは存在しないからである．

命題 2. 二項関係 $\leq$ は REC の上の半順序である．つまり

- すべての $f \in \mathrm{REC}$ で $f \leq f$.
- すべての $f, g \in \mathrm{REC}$ で $f \leq g$ かつ $g \leq f$ なら $f = g$.
- すべての $f, g, h \in \mathrm{REC}$ で $f \leq g$ かつ $g \leq h$ なら $f \leq h$.

が成り立つ．

4.2 計算可能な関数の連続性

4.2.1 インフォーマルな説明

これから計算可能な関数の単調性と連続性について議論していくが，まずはなぜ計算可能な関数が単調性と連続性を持つのかの直感的な説明をする．関数 $F\colon \mathrm{REC} \to \mathrm{REC}$ が計算可能であり，$e \in \mathbb{N}$ により実現されるとする．つまり

$$F(\varphi_n)(m) = e * n * m$$

が全ての $n, m \in \mathbb{N}$ で成り立つとする．自然数 e は同じ再帰的関数を計算するプログラムのゲーデル数は同じ再帰的関数を計算するプログラムのゲーデル数に移さないといけないので e ができそうなことは受け取るゲーデル数 n と自然数 m に応じて

$$\varphi_n(k_1) = l_1, \quad \ldots, \quad \varphi_n(k_j) = l_j$$

といった入出力の関係を調べこれらから $e*n*m$ の値を計算することであろう．結果 $e*n*m$ を計算する際に参照する入出力の関係が有限個であり無限個ではないのは $e * n * m$ が定義されるならその計算に要する計算ステップ数は有限だからである．有限ステップで無限個の入出力の関係を調べることはできない (だろう)．また停止性問題の決定不可能性からもプログラムをうまく調べれば無限個の入出力の関係を得られるなどありえないことがわかる．

本当にここでの考察が正しいのかは自明ではないがとりあえずこの考察から計算可能な関数の単調性と連続性が従うことをを説明しよう．まずは単調性について述べる．再帰的関数 φ_n, φ_m が $\varphi_n \leq \varphi_m$ を満たすとしよう．すると e はそれぞれのを計算するプログラムのゲーデル数 n と m から計算に必要な情報を抽出し，その情報によって $F(\varphi_n)$ と $F(\varphi_m)$ を計算するプログラムのゲーデル数を構成するのだから $F(\varphi_n)$ の持つ情報は $F(\varphi_m)$ の持つ情報よりも少ないだろう．つまり

$$f \leq g \implies F(f) \leq F(g)$$

が全ての $f, g \in \mathrm{REC}$ で成り立つ．この順序を保つという性質は**単調性**と呼ばれる．次に連続性について述べる．再帰的関数の上昇列 $f_1 \leq f_2 \leq f_3 \leq \cdots$ が与えられた時

$$f(n) = \begin{cases} m, & (\text{ある } i \in \mathbb{N} \text{ で } f_i(n) = m), \\ \bot, & (\text{その他}) \end{cases}$$

で与えられる関数 $f\colon \mathbb{N} \to \mathbb{N}_\perp$ が再帰的であったとすると f は上昇列 $f_1 \leq f_2 \leq f_3 \leq \cdots$ の「極限」であると捉えることができる．この f を F で写した結果は F の計算可能性から

$$F(f)(n) = \begin{cases} m, & (\text{ある } i \in \mathbb{N} \text{ で } F(f_i)(n) = m), \\ \perp, & (\text{その他}) \end{cases}$$

を満たすと考えられる．なぜなら $e \in \mathbb{N}$ が再帰的関数 f を計算するプログラムのゲーデル数から抽出した情報は

$$f(k_1) = l_1, \quad \ldots, \quad f(k_j) = l_j$$

という有限個の入出力の組であり (あると考えられ)，これらの情報は十分大きな i を選べば f_i から入手可能だからである．この「極限」を保つ性質が**連続性**と呼ばれる．

4.2.2 単調性と連続性の証明

単調性と連続性の証明を与えよう．証明はこれまでの議論を正しく行うのではなく背理法を用いる．例えば単調性の証明では単調性が成り立たないとしてそこから停止問題が決定可能であることを導く．まずは単調性について示す．

命題 3. 任意の計算可能な $F\colon \mathrm{REC} \to \mathrm{REC}$ は単調である．つまり全ての $f, g \in \mathrm{REC}$ について $f \leq g$ なら $F(f) \leq F(g)$ が成り立つ．

証明. $f \leq g$ となる $f, g \in \mathrm{REC}$ が与えられたとする．$f = g$ なら $F(f) = F(g)$ が明らかに成り立つ．$f \neq g$ である時には $F(f) \not\leq F(g)$ から矛盾を導く．仮定 $F(f) \not\leq F(g)$ から $F(f)(n) \neq \perp$ かつ $F(g)(n) \neq F(f)(n)$ を満たす $n \in \mathbb{N}$ が存在する．関数 $G\colon \mathrm{REC} \to \mathrm{REC}$ を

$$G(u)(n) = \begin{cases} f(n), & (f(n) \neq \perp) \\ g(n), & (u(0) \neq \perp \text{ かつ } g(n) \neq \perp) \\ \perp, & (\text{その他}) \end{cases}$$

と定めると $u(0) = \perp$ の場合には $G(u) = f$ であり，$u(0) \neq \perp$ の場合には $G(u) = g$ である．よって以下が成り立つ

$$u(0) = \perp \iff G(u) = f \iff F(G(u))(n) \neq \perp$$

特にある $e \in \mathbb{N}$ が存在して全ての $n \in \mathbb{N}$ で

$$\varphi_n(0) = \perp \iff e * n = 0$$

が成り立つ．(なぜか？) この e を使い以下の $e' \in \mathbb{N}$ を構成することは容易である．

$$\varphi_n(0) = \perp \iff e' * n = 0$$
$$\varphi_n(0) \neq \perp \iff e' * n = 1$$

この e' の存在は停止性問題の決定不可能性に矛盾する． □

次に計算可能関数の連続性について述べるために最小上界を定義する．

定義 2. 上昇列 $f_1 \leq f_2 \leq \cdots \in \mathrm{REC}$ の最小上界とは以下を満たす $f \in \mathrm{REC}$ のことである．

- 全ての $n \in \mathbb{N}$ で $f_n \leq f$
- 全ての $n \in \mathbb{N}$ で $f_n \leq g$ を満たす任意の $g \in \mathrm{REC}$ について $f \leq g$

上昇列 $f_1 \leq f_2 \leq \cdots$ の最小上界が存在する場合それを $\bigvee_{n\in\mathbb{N}} f_n$ や単に $\bigvee f_n$ などと書く．

練習問題 2. 最小上界が存在しない上昇列 $f_1 \leq f_2 \leq \cdots \in \mathrm{REC}$ を与え，実際に最小上界が存在しないことを証明せよ．

練習問題 3. 上昇列 $f_1 \leq f_2 \leq \cdots \in \mathrm{REC}$ の最小上界は存在するならただひとつであることを示せ．

命題 4. F を REC 上の関数とする．上昇列 $f_1 \leq f_2 \leq \cdots \in \mathrm{REC}$ の最小上界が存在する場合 $Ff_1 \leq Ff_2 \leq \cdots \in \mathrm{REC}$ の最小上界も存在して $F\left(\bigvee_{n\in\mathbb{N}} f_n\right) = \bigvee_{n\in\mathbb{N}} Ff_n$ が成り立つ．

証明. $\bigvee_{n\in\mathbb{N}} f_n$ を f と書くことにする．$Ff_n \leq F(f)$ は F の単調性から従う．$g \in \mathrm{REC}$ で全ての $n \in \mathbb{N}$ で $Ff_n \leq g$ となるものが与えられたとする．各 $m \in \mathbb{N}$ で $F(f)(m) \neq \bot$ の時に $g(m) = F(f)(m)$ が成り立つことを示す．$F(f_n)(m) \neq \bot$ となる $n \in \mathbb{N}$ が存在するなら $Ff_n \leq F(f)$ および $Ff_n \leq g$ から

$$g(m) = F(f)(m)$$

が従う．後は全ての $n \in \mathbb{N}$ で $F(f_n)(m) = \bot$ であるとして $F(f)(m) = \bot$ を示せば十分である．証明は $F(f)(m) \neq \bot$ を仮定して矛盾を導く．まず $u \in \mathrm{REC}$ に対し $\tilde{u} \in \mathrm{REC}$ を以下で定義する．

$$\tilde{u}(k) = \begin{cases} f(k), & (\text{全ての } n < k \text{ で } u(n) \neq \bot), \\ \bot, & (\text{その他}), \end{cases}$$

どのような $u \in \mathrm{REC}$ についても $u(k) = \bot$ となる $k \in \mathbb{N}$ が存在するなら $\tilde{u} \leq f_{n'}$ となる $n' \in \mathbb{N}$ を見つけられる．よって，そのような u については常に $F(\tilde{u})(m) = \bot$ である．一方で全ての $k \in \mathbb{N}$ で $u(k) \neq \bot$ なら $F(\tilde{u})(m) = F(f)(m) \neq \bot$ である．今，与えられた再帰的プログラム $\mathsf{f}\colon \mathbf{N} \Rightarrow \mathbf{N}$ に対し $u \in \mathrm{REC}$ を

$$u(n) = \begin{cases} 0, & (\mathsf{f}\,0 \text{ の実行が } n \text{ ステップでは停止しない}), \\ \bot, & (\mathsf{f}\,0 \text{ の実行が } n \text{ ステップ以内に停止する}) \end{cases}$$

と定義するとこれは再帰的関数である．u の定義から

$$\mathsf{f}\,0 \text{ の実行が停止しない} \iff F(\tilde{u})(m) \neq \bot$$

がわかる．右辺の値は f のゲーデル数から計算可能である．つまり全ての再帰的プログラム f について

$$\mathsf{f}\,0 \text{ の実行が停止しない} \iff e * \ulcorner \mathsf{f} \urcorner \neq \bot$$

を満たすある $e \in \mathbb{N}$ が存在する．この e を使い以下の $e' \in \mathbb{N}$ を構成することは容易である．

$$e' * n = \begin{cases} 0, & (n \text{ はある再帰的プログラム } \mathsf{f} \text{ のゲーデル数であり，} \mathsf{f}\,0 \Uparrow) \\ 1, & (n \text{ はある再帰的プログラム } \mathsf{f} \text{ のゲーデル数であり，} \mathsf{f}\,0 \Downarrow) \\ 2, & (\text{その他}) \end{cases}$$

この e' の存在は停止性問題の決定不可能性に矛盾する． □

4.3 不動点定理

連続性の帰結である以下の定理を証明する．

定理 6. 関数 $F\colon \mathrm{REC} \to \mathrm{REC}$ が計算可能なら F は最小不動点を持つ．つまり $F^\dagger \in \mathrm{REC}$ で

- $F(F^\dagger) = F^\dagger$
- $F(f) = f$ なら $F^\dagger \leq f$

を満たすものが存在する．

証明. まずは最小不動点の構成を与える．再帰的関数 $b \in \mathrm{REC}$ を

$$b(n) = \bot$$

と定義する．これは REC の最小元である．つまりすべての $f \in \mathrm{REC}$ について $b \leq f$ を満たす．特に $b \leq F(b)$ であり，F の単調性から

$$b \leq F(b) \leq F(F(b)) \leq F(F(F(b))) \leq \cdots$$

という上昇列を得る．この上昇列は最小上界 $F^\dagger \in \mathrm{REC}$ を持つ (なぜか?)．すると F の連続性から

$$F(F^\dagger) = \bigvee_{n \in \mathbb{N}} F^{n+1}(b) = F^\dagger$$

となる．つぎに最小性であるが，$F(f) = f$ を満たす $f \in \mathrm{REC}$ が与えられたとすると $b \leq f$ から帰納的に

$$F^n(b) \leq f$$

を示すことができ，最小上界の性質から $F^\dagger \leq f$ を得る． □

次の事実は証明はしない．

定理 7. 関数 $F\colon \mathrm{REC} \to \mathrm{REC}$ が計算可能であり，$e \in \mathbb{N}$ が F を実現するなら F の最小不動点 $F^\dagger$ は $\varphi_{\underline{\mathrm{fix}} * e}$ と等しい．

この定理からすべての $n \in \mathbb{N}$ について $\underline{\mathrm{id}} * n = n$ を満たす $\underline{\mathrm{id}} \in \mathbb{N}$ については

$$\underline{\mathrm{fix}} * \underline{\mathrm{id}} * n = \bot$$

であることが分かる．実際，$\underline{\mathrm{id}}$ は恒等関数 $\mathrm{id}\colon \mathrm{REC} \to \mathrm{REC}$ を実現しているので，id の最小不動点が b であることから $\underline{\mathrm{fix}} * \underline{\mathrm{id}} * n = \bot$ が得られる．

5 高階プログラミング

5.1 再帰的プログラムからの拡張

再帰的プログラムを使って出来ない自然数の計算は他のどのプログラミング言語を使っても不可能である (と強く信じられている)。だからといって何でもかんでも再帰的プログラムを使えば良いという話にはならない。プログラミングのしやすさやプログラムの読みやすさもプログラミング言語の大事な尺度だからである。

コラッツ予想の変種に興味があるとしよう。例えば n が奇数の場合には n を $2n+4$ でおきかえるコラッツ予想の様子を調べるためにはプログラム $\mathsf{colatz} : \mathbf{N} \Rightarrow \mathbf{B}$ の定義を少し変更して

$$\mathsf{colatz}'\ x = \mathsf{if}\ (x = 0\ \mathsf{or}\ x = 1)\ \mathsf{then\ true\ else}$$
$$\mathsf{if}\ (\mathsf{even}\ x)\ \mathsf{then}\ \mathsf{colatz}'\ (\mathsf{half}\ x)\ \mathsf{else}\ \mathsf{colatz}'\ (2x+4)$$

とすればよい。しかし他にも色々なコラッツ予想の変種を試す度にいちいち似たようなプログラムを書くのは面倒であるし、うっかり間違ったプログラムを書いてしまう可能性も出てくる。高階プログラミングの利点の一つは「同様な機能を持ったプログラム」をそのパラメータに着目してコンパクトに生成できることである。コラッツ予想の場合では n が偶数の場合と奇数の場合のそれぞれでどのような置き換えをするのかをパラメータ化して

$$\begin{array}{lcl} \mathsf{colatz}'' & : & (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{colatz}''\ f\ g\ x & = & \mathsf{if}\ (x = 0\ \mathsf{or}\ x = 1)\ \mathsf{then\ true\ else} \\ & & \quad \mathsf{if}\ (\mathsf{even}\ x)\ \mathsf{then}\ \mathsf{colatz}''\ f\ g\ x\ \mathsf{else}\ \mathsf{colatz}''\ f\ g\ x \end{array}$$

とし、興味のあるパラメータ f と g を選んで $\mathsf{colatz}''\ f\ g\ x$ を実行すればよい。こうすればいちいち新しくプログラムを書く手間が省け間違ったプログラムを書いてしまう可能性も多少は低くなるだろう。またうまくプログラムをパラメータ化することは読みやすいプログラムを書く上で重要である。

以下、この章では再帰的プログラムを高階プログラミングができるように拡張する。といってもこれまでの関数定義に現れる型についての制限を外すだけである。例えば高階版原始再帰法による定義（$\mathbf{B}$）

$$\begin{array}{lcl} \mathsf{f} & : & \mathbf{B} \Rightarrow A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow A_0 \\ \mathsf{f\ true}\ \vec{x} & = & M_{\mathsf{true}} \\ \mathsf{f\ false}\ \vec{x} & = & M_{\mathsf{false}} \end{array}$$

では $A_0, A_1, A_2, \ldots, A_n$ は任意の型でよいとする．つまり $A_1 = \mathbf{N} \Rightarrow \mathbf{N}$ などが許容されることになる．他の定義法についても同様である．プログラムの実行法については値式の定義に手を加えること以外は同じである。値式の定義は次のように変更する。

- これまで値式とされていたものは全て値式である。
- 識別子 f の定義が

  $$\mathsf{f}\ x_1\ \cdots\ x_n = \ldots$$

  という形をしていた場合、任意の $m < n$ と任意の値式 $V_1, \ldots, V_m$ について $\mathsf{f}\ V_1\ \cdots\ V_m$ は値式である。

例えば add 3 や add などは値式であるが、add 3 2 は値式ではない。識別子自体が値式となるのはその識別子が引数を伴って定義されている場合のみである。つまり

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

で定義される diverge は値式ではない。このプログラムは

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

と形は似ているが diverge′ は値式であり別物である。

練習問題 4.
直積型とリスト型についての原始再帰法はどのように変更するべきか？

5.2 高階プログラムの例

簡単な例

$$\begin{array}{lcl} \mathsf{comp} & : & (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow A \Rightarrow C \\ \mathsf{comp}\ f\ g\ x & = & g\ (f\ x) \end{array}$$

は関数の組を受け取ってそれらの合成を返す関数を定義している。

$$\begin{array}{lcl} \mathsf{eval} & : & \mathbf{N} \Rightarrow (\mathbf{N} \Rightarrow A) \Rightarrow A \\ \mathsf{eval}\ x\ f & = & f\ x \end{array}$$

は自然数 n を受け取ると次に受け取った関数の n での値を返す関数である。自然数上の関数の組を受け取りそれらを各点ごとに足して得られる関数を返す関数は

$$\begin{array}{lcl} \mathsf{add2} & : & (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{add2}\ f\ g\ x & = & \mathsf{add}\ (f\ x)\ (g\ x) \end{array}$$

とすれば得られる。add2 を関数 f と g および自然数 x を受け取り $f(x)+g(x)$ を返す関数と言うこともできる。

パラメータ化　以下は条件 p と自然数 x を受け取って x 以上で条件 p を満たす最小の自然数を返す関数である。

$$\begin{array}{lcl} \mathsf{next} & : & (\mathbf{N} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{N} \Rightarrow \mathbf{N} \\ \mathsf{next}\ p\ n & = & \mathsf{if}\ p\ n\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{next}\ p\ (\mathsf{s}\ n) \end{array}$$

条件 p の選び方によって色々な結果が得られる。例えば next prime x を実行すれば x 以上の最小の素数が得られ、next even x を実行すれば x の偶奇に応じて x もしくは $x+1$ が返ってくる。

コラッツ予想の場合であれば n が偶数の場合と奇数の場合のそれぞれでどのような置き換えをするのかをパラメータ化した

$$\begin{array}{lcl} \mathsf{gcolatz} & : & (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N} \Rightarrow \mathbf{B} \\ \mathsf{gcolatz}\ f\ g\ x & = & \mathsf{if}\ (x = 0\ \mathsf{or}\ x = 1)\ \mathsf{then\ true\ else} \\ & & \quad \mathsf{if}\ (\mathsf{even}\ x)\ \mathsf{then\ gcolatz}\ f\ g\ x\ \mathsf{else\ gcolatz}\ f\ g\ x \end{array}$$

という定義ができる。実際に使う場合は

$$\mathsf{gcolatz}\ (\mathsf{add}\ 3)\ (\mathsf{mul}\ 5)\ 1$$

などを実行すればよい。値式の定義を拡張したのはこのような高階プログラミングに対応するためである。これまでの値式の定義では add 3 や mul 5 は値式ではなかったが、そうすると gcolatz の展開が行われず何も計算されなくなってしまう。

アッカーマン関数　高階関数と原始再帰法を組み合わせると一般再帰法を使わなくてもアッカーマン関数を与えることが出来る。アッカーマン関数を展開していくと

$$\mathrm{ack}_{n+1}(m) = \overbrace{\mathrm{ack}_n(\mathrm{ack}_n(\cdots \mathrm{ack}_n}^{m+1}(1)\cdots)) \tag{6}$$

となることに注意する。まず

$$\begin{array}{lcl} \mathsf{itr} & : & (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N} \Rightarrow (\mathbf{N} \Rightarrow \mathbf{N}) \\ \mathsf{itr}\ u\ \mathsf{zero} & = & u\ 1 \\ \mathsf{itr}\ u\ (\mathsf{suc}\ y) & = & u\ (\mathsf{itr}\ u\ y) \end{array}$$

というプログラムを準備しておく。この itr は $u\colon \mathbb{N} \to \mathbb{N}$ と $n \in \mathbb{N}$ を受け取ると

$$\mathrm{itr}\ u\ n = \overbrace{u(u(\cdots u}^{n+1}(1)\cdots))$$

を計算する。すると (6) より、アッカーマン関数は

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

で与えられることがわかる。ただし全域再帰的関数が全て高階プログラムと原始再帰法を組み合わせて与えられるわけではない。

6　高階プログラミング言語の表現力

再帰的関数　高階プログラミング言語は今まで考えてきたプログラミング言語を含むので明らかに全ての再帰的関数 $f\colon \mathbb{N} \to \mathbb{N}_\perp$ に対し f を計算する高階プログラムが存在する. 逆にある高階プログラム $\mathsf{f}\colon \mathbf{N} \Rightarrow \mathbf{N}$ に対して $f\colon \mathbb{N} \to \mathbb{N}_\perp$ を

$$f(n) = \begin{cases} m & (\mathsf{f\ n} \Downarrow \mathsf{m}) \\ \perp & (\mathsf{f\ n} \Uparrow) \end{cases}$$

と定義すると，これは再帰的関数である．というのも高階プログラミングに移行したとは言ってもプログラム自体は依然ただの文字列であり，プログラムの実行は文字列の書き換えとして与えられているので高階プログラミング言語のプログラムのゲーデル数を受け取り，その実行結果を返すプログラムは高階型を用いなくても書けるからである．

並列実行　次の性質を満たす高階プログラム $\mathsf{p}\colon (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$ は存在しないことが知られている：任意の高階プログラム $\mathsf{f}\colon \mathbf{N} \Rightarrow \mathbf{N}$ について

- もし $\mathsf{f}\ 0 \Downarrow$ なら $\mathsf{p}\ \mathsf{f} \Downarrow 0$
- もし $\mathsf{f}\ 1 \Downarrow$ なら $\mathsf{p}\ \mathsf{f} \Downarrow 0$
- もし $\mathsf{f}\ 0 \Uparrow$ かつ $\mathsf{f}\ 1 \Uparrow$ なら $\mathsf{p}\ \mathsf{f} \Uparrow$

このような例を挙げたのは，高階プログラミング言語で表せない計算が必ずしも「計算不可能な」関数とは限らないことを示すためである．実際，高階プログラミング言語以下のように拡張することが可能である．

1. 2 引数の構成子 or を追加する．
2. M, N の型が $\mathbf{N}$ のときに $\mathrm{or}(M, N)$ の型を $\mathbf{N}$ とすることにする．
3. 高階プログラムの実行手続きに以下の規則を追加する：
   - $\mathsf{or}(\mathsf{n}, M) \longrightarrow_{\mathsf{f}} 0$
   - $\mathsf{or}(M, \mathsf{n}) \longrightarrow_{\mathsf{f}} 0$

プログラム p を $\mathsf{p}\ f = \mathsf{or}(f\ 0, f\ 1)$ と定義すればこの p が上の性質を満たすことは直接確認できる．

別の拡張　他にも高階プログラミング言語では表現できない計算が存在する．次のような高階プログラム $\mathsf{g}\colon ((\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$ が存在しないことが知られている：任意の $\mathsf{f}\colon (\mathbf{N} \Rightarrow \mathbf{N}) \Rightarrow \mathbf{N}$ について

- もし全ての $\mathsf{d} \in D$ で $\mathsf{f}\ \mathsf{d} \Downarrow 0$ なら $\mathsf{g}\ \mathsf{f} \Downarrow 0$
- もし全ての $\mathsf{d} \in D$ で $\mathsf{f}\ \mathsf{d} \Uparrow$ かつ全ての $\mathsf{c} \in C$ で $\mathsf{f}\ \mathsf{c} \Downarrow 0$ なら $\mathsf{g}\ \mathsf{f} \Downarrow 1$
- それ以外の場合は $\mathsf{g}\ \mathsf{f} \Uparrow$

ここで D と C は

$$D = \{\mathsf{d}\colon \mathbf{N} \Rightarrow \mathbf{N} : \mathsf{d}\ 0 \Uparrow\}$$
$$C = \{\mathsf{c}\colon \mathbf{N} \Rightarrow \mathbf{N} : \mathsf{c}\ 0 \Downarrow 0\}$$

で与えられる高階プログラムの集合である．この g も高階プログラミング言語に適切な拡張を施すことでプログラム可能になることが知られている．

しかし，高階プログラミング言語を上の p と g が共にプログラム可能になるように拡張することはできないことも知られている．というのも p と g が共にプログラム可能な場合これらを組合わせてプログラムの停止判定を行えてしまうからである．このことはチャーチの提唱の高階版が存在しないことを意味している．