数理情報学 9,10 （計算と論理）*

木原 貴行
名古屋大学 情報学部・情報学研究科

最終更新日: 2024 年 2 月 15 日

目次

* 本講義ノートは，2023 年度秋期開講の名古屋大学情報学部における講義「数理情報学 9,10」の内容をまとめたものである．講義のページ： http://www.math.mi.i.nagoya-u.ac.jp/~kihara/teach.html

§1. 結合子計算

1.1. チャーチ・チューリングの提唱

■計算論前史: 計算という概念自体は，はるか昔の時代から人びとの頭の中にあった．現代では，問題を解く計算的手続きに対して**アルゴリズム** (*algorithm*) という言葉を用いるが，アルゴリズムという言葉の由来は 9 世紀の数学者アル＝フワーリズミーに遡るとされる．アルゴリズム概念自体は，さらに古代に遡り，たとえば紀元前 3 世紀以前にユークリッドの『原論』に記されたユークリッドの互除法は，2 つの自然数の最大公約数を求めるためのアルゴリズムである．

また，人類は古くから様々な計算を自動的に行うための機械の構築を試みていた．実際に計算を実行する機械の起源としては，諸説あるが，たとえば，紀元前 2 世紀頃には，天体の位置を計算する機械として，アンティキティラ島の機械が発見されている．より計算機らしい例としては，たとえば 1642 年にパスカル (Blaise Pascal 1623–1662) は足し算を実行する機械 Pascaline を発明したようである．より高度な計算として，1671 年にライプニッツ (Gottfried Leibniz 1646–1716) の発明した機械は，乗算，除算や平方根の計算などもできたという．ライプニッツが，記号計算の研究にかなり尽力していたことは広く知られている．このように，17 世紀頃から，大勢の研究者によって，様々な計算を行う機械が考案されていった．その中でも特筆すべき例として，1822 年にバベッジ (Charles Babbage 1791–1871) は多項式の値を計算する能力を持つ機械である階差機関 (difference engine) を設計したが，実際に完成させることはなかった．とにかく，このように 19 世紀以前においても，様々な計算機械が考案されてきたのである．しかし，これらのような，計算論前史における機械の多くは，特定の関数だけを計算することに特化している．

■汎用計算: その例外となるものとして，1830 年代からはバベッジは解析機関 (analytic engine) と呼ばれる機械の設計に没頭する．彼が亡くなる 1870 年代まで設計の改良が続けられたが，これもまた現実に完成することはなかった．解析機関は，計算論前史における典型的な機械と違い，パンチカードを用いて様々な計算を実現可能な，いわゆる汎用コンピュータの一種であった．ただし，解析機関でそれなりの種類の計算が実行できるからといって，ありとあらゆる種類の計算が実行できるかどうかは，少なくとも当時の感覚では，まったく分かったものではない．そもそも「ありとあらゆる計算」とは何であるか，バベッジの時代には，まったく想像の付くものではなかったと思われる．

もちろん，現代の計算論的知識を用いることによって，バベッジの解析機関で実行できるいくつかの種類の計算を複雑に組み合わせると，（計算時間やメモリの問題を気にしなければ）理論上は現代のコンピュータと同じ計算ができると示せる，と言われている．実際，これから本稿で見るように，ほんのわずかな種類の基本的な計算さえ実行できれば，それが汎用計算モデルとなることが示せるため，そうであってもまったく不思議なことではない．

現代のコンピュータは，一つの機械の内部で，プログラムを実行することによって，ありとあらゆる種類の計算を実行できる．このような

あらゆる計算を内部でシミュレートできる計算モデル（**万能マシン**）

の発見が，計算理論のはじまりであった．

しかし，そもそも「計算」とは何であろうか．「計算」という概念に形式的定義が与えられない限り，ある計算モデルが「ありとあらゆる種類の計算を実行できる」汎用計算モデルであるか否かを議論することはできない．計算理論の誕生において最も重要であったブレイクスルーは，「計算とは何か」が定式化されたこと，そして，その結果として「万能計算モデル（万能マシン）」の存在に厳密な数学的証明が与えられたこと，さらに「わずかな種類の計算から膨大な種類の計算を実行可能である」と証明されたことである．これらの偉大な発見は，バベッジよりもかなり後の時代のこととなる．

■計算の数学的定義に至るまで: 「計算」とは，大雑把に言えば，「有限的な操作からなる有限ステップのプロセス」である．数学においては，数学の発展に伴い，徐々に数学は無限を駆使し始めるようになり，19 世紀になると「非構成的」と呼ばれる論法が見られるようになった．このため，20 世紀に入った頃になると，数学において「有限のプロセス」と「無限のプロセス」 を分別する言及が徐々に現れ始める．その時代に提示された問題として，以下のようなものがある．

- (ヒルベルトの第 10 問題, 1900) 整数係数ディオファントス方程式が整数解を持つか否かを有限の操作で決定できるようなプロセスを考案せよ．
- (ヒルベルト-アッカーマンの決定問題, 1928) 一階述語論理において，与えられた言明の真偽を決定する機械的な手続きを見つけよ．

しかし，これらの問題が提示された時点では，「有限の操作」や「機械的な手続き」といった概念が何であるかについて，形式的な定義は与えられていなかった．したがって，これらの問題を厳密に解決するためには，まずは「有限の操作」や「機械的な手続き」といった概念の数学的定義を与えなければならない．

同様の問題は，数学基礎論の別の問題においても発生していた．ゲーデル (Kurt Gödel 1906–1978) は 1930 年にゲーデルの不完全性定理と呼ばれる定理を証明した．このうち第一不完全性定理は，ある種の前提条件を満たす数学的理論の不完全性を示すものであるが，その前提条件の一つは「何が公理であるかが有限的な手続きで定められている」というものであった．ゲーデル曰く，ゲーデルが不完全性定理の証明に必要な前提条件は

> 「どの記号列が体系内で扱われる論理式であるか」「どの論理式が公理であるか」を定める有限の手続きがあり，各「推論規則」も有限の手続きで与えられなければならない

というものである．ここでも「有限の手続き」という概念が出現する．上記のヒルベルトらの問題との違いは，ヒルベルトらの問題は漠然とした曖昧な問い掛けであったのに対して，ゲーデルの定理は厳密な数学的定理であるという点である．曖昧な問い掛けに定義は必要ないが，数学的定理には厳密な定義が必要である．ゲーデルの不完全性定理の証明には，上記の前提条件は外せない必須の条件であったため，「有限の手続き」とは何であるか，その範囲を定める必要があった．そこ

で，ゲーデルは，エルブラン (Jacques Herbrand 1908–1931) のアイデアに基づき，**一般再帰関数** (*general recursive function*) の概念を「有限の手続き」の定式化の一つとして用い，この前提の下で不完全性定理の厳密な証明を与えた．この意味で，一般再帰関数が「計算」の概念の最初の定義案とも言うことができ，実際に，後に与えられた「計算可能性の定義」と同値になることが示されることとなる．しかし，ゲーデルは，その段階では，一般再帰関数を「計算可能性の定義」とは考えていなかった．とはいえ，「計算」という日常用語と厳密な数学的定義を関連付けた最初期のアイデアであることは間違いない．

■チャーチ-チューリングの提唱: 「計算とは何か」の定義に至る大きなイベントは 1936 年に起きた．その年，上で述べたヒルベルト-アッカーマンの決定問題の否定的解決が 2 人の数学者によって独立に宣言されたのである．しかし，先に述べたように，ヒルベルト-アッカーマンの決定問題を数学的に解決するためには，「機械的な手続き」とは何であるかの数学的定義を与えられる必要がある．「機械的な手続き」の定義として，解決者のうち 1 人のチャーチ (Alonzo Church 1903–1995) は，**ラムダ計算** (*lambda calculus*) を考案し，もう 1 人のチューリング (Alan Turing 1912–1954) は，**チューリングマシン** (*Turing machine*) を考案した．チャーチとチューリングの 2 人は，彼ら独自の「計算可能性の定義」の用い，それぞれ独立にヒルベルト-アッカーマンの決定問題の否定的解決を宣言したのである．同年に，クリーネは，エルブランとゲーデルのアイデアに基づき **μ-再帰関数** (μ-*recursive function*) を導入し，またポストは彼が Formulation I と呼ぶ計算モデルを導入した．つまり，1936 年に，以下の 4 つの「計算可能性の定義」が提示されたことになる．

ラムダ計算	チューリングマシン
μ-再帰関数	Formulation I

すぐにクリーネは μ-再帰関数と λ-計算が計算モデルとして等価であることを示し，翌 1937 年に，チューリングは，チューリングマシンとそれらの計算モデルとの等価性を示した．事実，上記の 4 つの「計算可能性の定義」はいずれも同値であることが示されることとなる．また，1936 年に，チャーチは，これらの定義とアルゴリズム的計算可能性および形式算術による計算可能性と結び付ける議論を行い，これらを根拠に，これらの計算モデルによる計算可能性を「計算可能性の数学的定義」とすることを提唱した．

その後，現代に至るまで，幾百もの計算モデルと幾千ものプログラミング言語が世に溢れることとなる．そして，「これらの計算モデルによって計算可能なものは（計算速度などを気にしなければ）チューリングマシンによって計算可能なものと正確に一致する」ということが数学的に証明され，ならば，やはりこれが計算可能性の妥当な定義であろう，ということで決着が付いた．これが **チャーチ・チューリングの提唱** (*Church-Turing thesis*) である．

念のため，この計算史の中で，どこまでが《数学》でどこからが《提唱》なのかを明示しておこう．まず，

> 「1930 年代以降に人類が考案した幾千幾万の計算モデルの計算能力が（計算速度などを気にしなければ）等価である」

という点については，**数学的な証明が与えられている**部分であるから，ここは《**数学**》である．ちなみに 1930 年代以降と注記したのは，現代的な計算モデルの計算能力よりも真に弱い計算モデルも，過去には考案されてきたためである．1930 年代以降でも，弱い計算モデルが意図的に扱われることがあるが，とにかく，弱い計算モデルは無視することにする．さておき，その一方で，

> 「これまでに人類が考えついた計算モデルの計算能力はすべて等価なのだから，これが計算可能性の定義なのだろう」

という点については，《**提唱**》である．なぜなら，現時点までに人類が考えついた幾千幾万の計算モデルの計算能力がたまたま等価だった，というだけのことに過ぎないかもしれず，実は，人類はまだ完璧な計算モデルを考案できていないのかもしれない．そして，千年後あるいは十億年後の未来に，何らかの生命体が，既知の計算モデルを超越する計算能力を持つ機械を発明するかもしれない．まず有り得ないだろうが，全く有り得ないと断言できるわけではない，かもしれない．

とにかく，このように，世には幾千幾万の等価な計算モデルが存在している．すると，これらの等価な計算モデルに共通する何か，計算可能性の本質のようなものがあってよいはずである．そのような本質は，人類史において既に現れた計算モデルだけでなく，未だ見ぬ遠い未来の超越的な計算モデルにも（そのようなものが存在するかどうかはさておいて）共有されているものであろう．つまるところ，われわれは，ありとあらゆる計算モデルの共有する，計算可能性の根源となる普遍的原理を，何らかの単純な数学的構造として抽出したい．未来の超越的な計算モデルという夢想はさておくにせよ，現存する計算モデルの数学的本質を追求することに価値があるという点については疑いの余地はないであろう．

1.2. 結合子による計算

■結合子論理: 計算の本質を理解するために，まずは最も単純な計算モデルの分析から始めたい．1936 年にラムダ計算，チューリングマシン，μ-再帰関数，Formulation I という万能計算モデルが「計算可能性の定義」として提示されたと触れたが，実はそれに遡ること十数年，それよりも遥かに単純な万能計算モデルが考案されていた．それは，シェーンフィンケリ (Moses Shönfinkel; 1889–1942) によって 1920 年頃に導入された**結合子論理** (*combinatory logic*) である．ただし，シェーンフィンケリは，結合子論理が汎用計算モデルだとは考えていなかったため，バベッジの解析機関と同様，あくまで後の時代になって，万能計算モデルであると認識されたものである．しかし，極めて単純な汎用計算モデルであり，計算可能性の本質を突いたものであるから，まずは結合子計算の紹介をしよう．現代的な結合子の計算論は，たった 2 つの記号列書き換え操作のみからなる．

$$\mathtt{k}\,xy \to x \qquad\qquad \mathtt{s}\,xyz \to (xz)(yz)$$

もう少し丁寧に述べれば，結合子計算は，2 つのコマンド k を s を持っており，これらのコマンドは以下の動作を実行する．

- k: 右 2 つのデータ xy を読み込んで，x を取り出す．
- s: 右 3 つのデータ xyz を読み込んで，$(xz)(yz)$ に変換する．

ここで，記号列は左から順に読み込み，それに応じて括弧を省略する．つまり $xyz = (xy)z$ である．たとえば，$(xz)(yz)$ は $xz(yz)$ と略記してよい．この k や s は**結合子** (*combinator*) と呼ばれる．驚くべきことに，なんと，この 2 つの記号列操作の命令だけで，（計算時間やメモリの問題を気にしなければ）あらゆるコンピュータの計算をシミュレートできることが知られている．しかし，この k と s を用いて様々な計算をシミュレートできると言われても，すぐには感覚が掴めないと思うので，まずは k や s を組み合わせて，様々な記号列書き換え操作コマンドを表せることを見ていこう．ここで，記法に関する注意として，書き換え操作 $\to$ の有限回の適用を通常は $\to^*$ と表すことが多いが，記号が煩雑になるので，本稿では単に $\to$ と書いてしまうことにする．

例 1.1. まず，「何もしない」という操作を行うコマンドをどのように作るか考えてみよう．

$$?\,x \to x$$

答えを述べてしまえば，$? = \mathtt{skk}$ とすればよい．

$$\mathtt{skk}\,x \to \mathtt{k}x(\mathtt{k}x) \to x$$

ここで，各ステップで実行されるコマンドを赤字で，読み込まれる文字を青字で表している．この結合子計算の注意点としては，括弧で括られた文字列 $(\mathtt{k}x)$ は 1 セットであるとみなすという点である．つまり，上記の第 2 ステップの計算において，k が読み込む 1 つめのデータは x であり，2 つめのデータは $(\mathtt{k}x)$ というセットである．以後は，何もしないコマンドをしばしば i と略記する．

例 1.2. 次に「右 2 つのデータ xy を読み込んで，2 つめのデータ y を取り出す」という操作を行うコマンドをどのように作るか考えてみよう．

$$?\,xy \to y$$

答えを述べてしまえば，$? = \mathtt{sk}$ とすればよい．

$$\mathtt{sk}\,xy \to \mathtt{k}y(xy) \to y$$

例 1.3. ところで，xyz は $(xy)z$ の略記であったが，「$(xy)z$ を読み込んで $x(yz)$ を返す」という操作，つまり「結合順序を変える」コマンドをどのように作るか考えてみよう．

$$?\,xyz \to x(yz)$$

答えを述べてしまえば，$? = \mathtt{s(ks)k}$ とすればよい．

$$\mathtt{s(ks)k}\,xyz \to \mathtt{ks}x(\mathtt{k}x)yz \to \mathtt{s}(\mathtt{k}x)yz \to \mathtt{k}xz(yz) \to x(yz)$$

この結合子計算における注意点としては，コマンド s は右 3 つのデータ $(\mathtt{ks})\mathtt{k}x$ を読み込んだ段階で計算を実行するので，yz の部分は書き換え操作を行わず，そのまま次のステップへと持ち越される．その後のステップについても同様である．

リマーク. シェーンフィンケリは 1914 年から 1924 年までゲッティンゲンのヒルベルト (David Hilbert 1862-1943) の研究グループに属しており，結合子論理のアイデアはそこで 1920 年 12 月に発表されたものであるようだ．結合子論理に関するシェーンフィンケリの研究は，「数理論理学の構成単位について」という題で 1924 年に出版された．

■関数構成システム: 以上の例より，結合子 s と k を用いて様々な記号列書き換えコマンドが作れるという感覚は掴めたかもしれない．しかし，このままでは結合子計算はあくまで謎の記号列書き換えシステムであり，どういうアイデアが背後にあるのか，まったく明確ではない．このアイデアを理解するためには，「計算モデル」という概念に最低限要求される数学的性質とは何か，ということについて思索を巡らす必要がある．

まず，「計算モデル」というからには，基本的な関数は実装できてほしい．たとえば，各 $c \in C$ に対して，定数関数 $\mathtt{const}_c : X \to C$ を実装できるべきである．さらに言えば，$c \mapsto \mathtt{const}_c$ を実装できてほしい．つまり，$\mathtt{k}(c) = \mathtt{const}_c$ となる関数 $\mathtt{k} : C \to C^X$ を実装できる．これは以下のように表される．

$$\mathtt{k}(c)(x) = c \qquad \text{あるいは} \qquad \mathtt{k}\colon (c, x) \mapsto c$$

続いて，関数適用も実装できるのがよいだろう．関数のリスト $f = (f_t : X \to Y)_{t \in T}$ が与えられたとき，入力リスト $x = (x_t)_{t \in T}$, $x_t \in X_t$, に出力リスト $(f_t(x_t))_{t \in T}$ を対応させる関数を実装したい．これは，次の関数 $\mathtt{s} : [X \to Y]^T \to [X^T \to Y^T]$ が実装できるということである．

$$\mathtt{s}(f)(x)(t) = f_t(x_t) \qquad \text{あるいは} \qquad \mathtt{s}\colon (f, x, t) \mapsto f_t(x_t)$$

簡単のために f_t の始域と終域を統一させたが，実際には t 毎に異なる始域と終域を変えても構わない．このような関数適用のリストを，シェーンフィンケリは「関数融合」と呼んだ．ともあれ，我々の「計算モデル」に求める最低限の要件は以上であるとする．定数関数構成と関数融合における括弧を省略してまとめると，

$$\mathtt{k}\, cx \to c \qquad\qquad \mathtt{s}\, fxt \to ft(xt)$$

となっている．もちろん，これは結合子 s と k の動作そのものである．このように考えると，結合子計算は，「定数関数構成」と「関数融合」を組み合わせて，どんどん新しい関数を作っていく「関数構成システム」であると考えることができる．同時に，「様々な関数を有限文字列で表すシステム」であると思ってもよい．たとえば，例 1.4 より，恒等関数 $\mathrm{id}\colon x \mapsto x$ は skk という組み合わせで実現できる，というようなことである．

結合子計算を関数構成システムとみなすという観点の詳細に踏み込もう．結合子 k は，右 2 つのデータ c, x を読み込むコマンドであるが，右 1 つのデータ c を読み込んだ時点で，$\mathtt{const}_c$ という定数関数を得られており，2 つめデータ x はこの関数に対する入力である．結合子 s は，右 3 つのデータ f, x, t を読み込むコマンドであるが，右 2 つのデータ f, x を読み込んだ時点で，$(f_t(x_t))_{t \in T}$ というデータのリストは得られており，3 つめのデータ t はこのうちのどのデータを取り出すかの指定に過ぎない．

$$\mathtt{k}\, c\colon x \mapsto c \qquad\qquad \mathtt{s}\, fx\colon t \mapsto f_t(x_t)$$

一般的に，結合子計算における右 $n+1$ 個のデータを読み込むコマンド z は，右 n 個のデータを読み込んだ時点で，関数として完成していると考える．そして，$n+1$ 個めのデータがこの関数に対する入力であると考える．たとえば，例 1.4 のコマンドを i, 例 1.2 のコマンドを j, 例 1.3 のコマンドを b と略記して，関数を完成させるのに必要なデータを青色，関数への入力データを橙色で強調しよう．

$$\mathtt{i}\,x \to x \qquad \mathtt{j}\,zx \to x \qquad \mathtt{b}\,fgx \to f(gx)$$

これらのコマンドは，左から順に

恒等関数 $I\colon x \mapsto x$　　恒等関数の構成 $J_z\colon x \mapsto x$　　関数合成 $B_{f,g}\colon x \mapsto f(g(x))$

を表すものであると理解できる．このようにして，結合子計算は，単なる謎の記号列書き換えシステムではなく，「定数関数構成」と「関数融合」から様々な関数を構成していくシステム，関数構成を有限記号列によって表すシステムである，という側面があることが明確になった．

■自己言及: ところが，結合子計算を関数構成とみなした場合に，少し扱いが難しいものがある．それが，自己言及を行う種類のコマンドである．具体例を挙げよう．

例 1.4. 「入力データを複製する」という操作を行うコマンドをどのように作るか考えてみよう．

$$?\,x \to xx$$

答えを述べてしまえば，$? = \mathtt{sii}$ とすればよい．ここで，i は何もしないコマンド，つまり skk である．

$$\mathtt{sii}x \to \mathtt{i}x(\mathtt{i}x) \to xx$$

ここで，各ステップで実行されるコマンドを赤字で，読み込まれる文字を青字で表している．

この複製コマンドをそのまま関数として理解するのは難しい．無理やり関数として理解しようとすると，関数 x に x を入力するという操作 $x(x)$ となるが，これはあまり意味をなさない．ただし，関数として適切に理解する方法もあり，たとえば，「x という記号でコードされた関数に，記号 x を入力する」という解釈であれば意味を持つ．

1.3. 結合子完全性

■結合子計算の代数構造: ここからの目標は，結合子計算によって，あらゆるコンピュータの計算をシミュレートできるという実感を抱くことである．このための第一ステップとして，まずは結合子計算によって「あらゆる文字列書き換え」を実現できることを証明しよう．

結合子計算は，記号列のシステムであるが，記号列 x と y の結合 xy のことを「x と y の積」と考えることもできる．抽象代数では，このような積概念があるとき，単位元の存在や逆元の存在などを議論するが，結合子計算の世界においてはどうだろうか．まず，例 1.4 より，左単位元 i を持つ．つまり，

(左単位元)　ある $\mathtt{i}$ が存在して，任意の x に対して，$\mathtt{i}\,x \to x$ となる．

したがって，結合子計算の代数的構造には多少の秩序がありそうである．しかし，他の代数的振る舞いについては，あまり良くはない．たとえば，多くの代数構造において，結合律 $(xy)z = x(yz)$ は成り立つが，結合子計算の関数構成としての意味を考えれば，当然これは成り立たない．一応，証明のアイデアを与えておこう．

例 1.5. 結合子計算において，結合律は成立しない．

Proof (アイデア). 結合律が成立するならば，特に $\mathtt{kkk} = (\mathtt{kk})\mathtt{k}$ と $\mathtt{k}(\mathtt{kk})$ を同一視できる．これを否定するために，$\mathtt{k}(\mathtt{kk}) \to \mathtt{kkk}$ が成り立つと仮定する．まず，任意の x に対して，

$$\mathtt{k}(\mathtt{kk})x\mathtt{k} \to \mathtt{kkk} \to \mathtt{k}$$

であるが，一方で，仮定より

$$\mathtt{k}(\mathtt{kk})x\mathtt{k} \xrightarrow{\text{仮定}} \mathtt{kkk}x\mathtt{k} \to \mathtt{k}x\mathtt{k} \to x$$

よって，特に $x = \mathtt{s}$ とすれば，$\mathtt{s} = \mathtt{k}$ となるが，これは成り立たない． □

これを厳密な証明とするためには，結合子計算による記号列書き換えの結果が一意である（合流性）とかいった，そういうことを証明する必要がある．しかし，ここでは一旦それは置いておく．もう一つ，結合子計算においては可換性 $xy = yx$ も成立しない．これも結合子計算の関数構成としての側面から明らかであろう．

例 1.6. 結合子計算は非可換である．

Proof (アイデア). 可換ならば，特に $\mathtt{kii} = (\mathtt{ki})\mathtt{i}$ と $\mathtt{i}(\mathtt{ki})$ を同一視できる．これを否定するために，$\mathtt{kii} \to \mathtt{i}(\mathtt{ki})$ が成り立つと仮定する．まず，任意の x に対して，

$$\mathtt{kii}x \to \mathtt{i}x \to x$$

であるが，一方で，仮定より，

$$\mathtt{kii}x \xrightarrow{\text{仮定}} \mathtt{i}(\mathtt{ki})x \to \mathtt{ki}x \to \mathtt{i}$$

よって，特に $x = \mathtt{s}$ とすれば，$\mathtt{s} = \mathtt{i}$ となるが，これは成り立たない． □

上記と同様に，あくまで厳密な証明ではなく，現時点ではアイデアである．しかし，結合子計算において，結合律も可換性も成立しないであろうという直感は得られたと思われる．

ところで，環のような代数構造が与えられると，我々は多変数多項式を考えることができる．多変数多項式とは $6x^5y^2 + 3x^2y^4 + x^3 + 9$ のようなものである．しかし多項式を扱うには，和と積という 2 つの演算が必要であるが，我々の扱う代数には 1 つの演算しかないから，考えるものは単項式である．多変数単項式とは $4x^3y^2z^4$ のようなものであり，つまり $4xxxyyzzzz$ のことであ

る．このように多変数単項式を整理された形で書けるのは，積が結合的かつ可換であるときのみである．たとえば，文字列 $xy(yxz)yay(zx)x$ は結合律と可換性によって $axxxxyyyyzz$ あるいは $ax^4y^4z^2$ と整理される．

$$xy(yxz)yay(zx)x \xrightarrow{\text{結合律}} xyyxzyayzxx \xrightarrow{\text{可換性}} axxxxyyyyzz = ax^4y^4z^2$$

しかし，例 1.5 と 1.6 で見たように，結合子計算の代数は結合律も可換性も満たさない．つまり，我々の代数における多変数単項式（単に項と呼ぶ）は，$axy(yxz)yy(zx)x$ のように整理されていない文字列であり，つまり（括弧付き）有限記号列である．形式的には，項の概念は以下によって定義される．

定義 1.7. 結合子計算の**項** (*term*) を以下のように帰納的に定義する．

1. 変数 $x, y, z, \ldots$ は項である．
2. 結合子 $\mathtt{s}, \mathtt{k}$ は項である．
3. t と u が項ならば tu も項である．

以後は, 結合子計算の項のことを結合子項あるいは単に項と呼ぶ．バッカス-ナウア記法 (Backus-Naur form) というものを用いれば，結合子項の定義は,

$$t ::= x \mid \mathtt{s} \mid \mathtt{k} \mid tt \qquad (x \in \mathrm{Var})$$

と表示することもできる．ここで，Var は変数全体の集合である．このバッカス-ナウア記法は，「項 t とは，変数 x であるか，結合子 $\mathtt{s}$ であるか，結合子 $\mathtt{k}$ であるか，項 t_1, t_2 について t_1t_2 の形である」ということを意味する．ともあれ，どのようなものが項になるかは明らかであると思う．何らかの文字列に対して「項であることを示せ」と言われた場合に，どのように項であることを証明するか，についてだけ説明しておこう．

例 1.8. $xz(yz)$ は項である．

$$\dfrac{\dfrac{\dfrac{}{x \text{ は項}}\,(1) \quad \dfrac{}{z \text{ は項}}\,(1)}{xz \text{ は項}}\,(3) \quad \dfrac{\dfrac{}{y \text{ は項}}\,(1) \quad \dfrac{}{z \text{ は項}}\,(1)}{yz \text{ は項}}\,(3)}{xz(yz) \text{ は項}}\,(3)$$

上の樹形図の意味は，項の定義 (1) および (3) を用いて，木の葉（上）から順に，項であることを示していき，最終的に木の根（下）において $xz(yz)$ が項であるということが示されるという証明の流れを表す．上記のような樹形図を項の**構文木** (*syntax tree*) と呼ぶ．構文木は，しばしば上下逆向きに書かれることもある．

例 1.9. $\mathtt{skk}x$ は項である．

$$\dfrac{\dfrac{\dfrac{}{\mathtt{s}\text{ は項}}\,(2) \quad \dfrac{}{\mathtt{k}\text{ は項}}\,(2)}{\mathtt{sk}\text{ は項}}\,(3) \quad \dfrac{}{\mathtt{k}\text{ は項}}\,(2)}{\mathtt{skk}\text{ は項}}\,(3) \quad \dfrac{}{x\text{ は項}}\,(1)}{\mathtt{skk}x\text{ は項}}\,(3)$$

ところで，変数 $x_1, \ldots, x_n$ を含む項 $t(x_1, \ldots, x_n)$ は，記号列 $x_1 \ldots x_n$ を $t(x_1, \ldots, x_n)$ に書き換える動作だと考えることができる．現時点までに分かっていることとして，以下のいずれの項

$$t_{\mathtt{k}}(x, y) = x \qquad t_{\mathtt{s}}(x, y, z) = xz(yz) \qquad t_{\mathtt{i}}(x) = x$$
$$t_{\mathtt{j}}(x, y) = y \qquad t_{\mathtt{b}}(x, y, z) = x(yz) \qquad t_{\mathtt{m}}(x) = xx$$

を考えても，左から適切なコマンドをぶつけることで，項と同じ動作をシミュレートできるのであった．

$$\mathtt{k}\,xy \to t_{\mathtt{k}}(x, y) \qquad \mathtt{s}\,xyz \to t_{\mathtt{s}}(x, y, z) \qquad \mathtt{skk}\,x \to t_{\mathtt{i}}(x)$$
$$\mathtt{sk}\,xy \to t_{\mathtt{j}}(x, y) \qquad \mathtt{s(ks)k}\,xyz \to t_{\mathtt{b}}(x, y, z) \qquad \mathtt{sii}\,x \to t_{\mathtt{m}}(x)$$

一般に，どんな記号列書き換え操作 $t(x_1, \ldots, x_n)$ が与えられても，適切なコマンドの実行により，その動作をシミュレートできることを示そう．

定理 1.10 (結合子完全性)**.** $t(x_1, \ldots, x_n)$ を $x_1, \ldots, x_n$ 以外の変数を含まない項とする．このとき，変数を含まない項 $\mathtt{a}$ が存在して，以下が成立する．

$$\mathtt{a}\,x_1 \ldots x_n \to t(x_1, \ldots, x_n)$$

結合子完全性は，ある意味で，結合子計算によってラムダ計算をシミュレートできるということを述べるものである．これについて，まず，ラムダ記法に関する解説から始めよう．

1.4. ラムダ記法

■ラムダ記法とは: ラムダ記法 (lambda notation) の基本的なアイデアを述べると，ラムダ記法とは関数内の記号のどれが変数でどれが定数かを曖昧さなく明示化するための記法である．たとえば，$x^2 + 5y + 3$ という式があったとしよう．これだけを見ても，たとえば x は定数なのか変数しか判別が付かないし，以下の 4 つのパターンが考えられる．

1. x も y も定数である．
2. x は動く変数であるが，y は固定された定数である．
3. y は動く変数であるが，x は固定された定数である．
4. x も y も動く変数である．

標準的な数学的記法では，たとえば (2) の場合は $f(x) = x^2 + 5y + 3$ と書き，(3) の場合は $g(y) = x^2 + 5y + 3$ と書き，(4) の場合は $h(x, y) = x^2 + 5y + 3$ と書くことによって，どれが変数でどれが定数かを明示化することができる．ただし，この記法の欠点は，毎回新たな関数名を準備する必要があるという点である．通常の数学的議論では，この欠点は大きな問題とはならないが，関数による計算システムを考える場合には，大量の関数が出現する上に，どの記号が変数でどの記号が固定されているかも次々に移り変わっていくので，この欠点は致命的なものとなり得る．この欠点を解決するために，新しい関数名を導入せずに，どれが変数でどれが定数かを明示する方法を与えるものが，ラムダ記法である．具体的には，(2),(3),(4) の状況は，ラムダ記法においては，以下のように表される．

$$(2)\quad \lambda x.x^2 + 5y + 3 \qquad (3)\quad \lambda y.x^2 + 5y + 3 \qquad (4)\quad \lambda x.\lambda y.x^2 + 5y + 3$$

つまり，ラムダ記法とは，「λz」という記号[*1]で，「z を動かす」という宣言をするシステムである．もう少し正確には，関数 $t(x_1, x_2, \ldots, x_n)$ があったときに，記号 $\lambda x_i.t(x_1, \ldots, x_n)$ によって，t に出現する変数のうち x_i を動かすと宣言するということである．この方法で，新しい関数名を導入せずに，どれが変数でどれが定数かを明示することができる．

ところで，関数を考えるならば，変数への値の代入を表す記法が必要である．たとえば，数学の標準的記法においては，「関数 $f(x)$ の中の x に 7 を代入する」という記法は $f(7)$ として表される．「関数 $h(x, y)$ の x に 2 を代入し，y に 1 を代入する」という記法は $h(2, 1)$ として表される．それでは，ラムダ記法によって，これをどのように表すかというと，

$$(\lambda x.\lambda y.x^2 + 5y + 3)\ 2\ 1$$

のように，右に代入したい値を配置する．つまり，結合子計算のときと同様に，$\lambda x.\lambda y.x^2 + 5y + 3$ という 2 つの引数を持つ関数的プログラムが，右 2 つのデータを読み込んで，計算を実行すると考えてもよい．上では，どの引数にどの値が対応するか分かりやすくなるように色を塗っている．実際には，次のように，ラムダ記法の計算は左から順に実行されていく．

$$(\lambda x.\lambda y.x^2 + 5y + 3)\ 2\ 1 \to (\lambda y.2^2 + 5y + 3)\ 1 \to 2^2 + 5 \cdot 1 + 3 \to 12$$

より一般的に，以下のような式変形が成り立つと考えてよい．

$$\big(\lambda x_1.\lambda x_2.\ldots.\lambda x_n.t(x_1, x_2, \ldots, x_n)\big)\ a_1\ a_2\ \ldots\ a_n \longrightarrow t(a_1, a_2, \ldots, a_n)$$

厳密には，t とは何であるかなどを説明する必要があるが，現時点では，このような素朴ラムダ計算論さえ把握していれば十分である．もう少し詳細なラムダ計算の基礎理論は，第 2.1 節で展開する．省略記法として，複数の連続するラムダ $\lambda x_1.\lambda x_2.\ldots.\lambda x_n.t$ は，しばしば 1 つのラムダ $\lambda x_1 x_2 \ldots x_n.t$ にまとめてしまうことが多い．たとえば，$\lambda x.\lambda y.x^2 + 5y + 3$ は $\lambda xy.x^2 + 5y + 3$ と略記する．

[*1] チャーチのオリジナルの記法では，ラムダ λ ではなく，山型記号 ⋀ を用いて，⋀ $x.xy$ のように記述していたらしい．

以上が，ラムダ記法の基本である．これだけを聞くと，ラムダ記法とは，単なる自明な記法の話に過ぎないと思うかもしれない．実際，余談であるが，筆者は学生時代に初めてラムダ計算という概念を聞いて，てっきり数学的には特に中身のない単なる記法の話だと思い込み，長年ラムダ計算を勉強しなかった……というか，ラムダ計算に対して「勉強する」という要素が何か存在し得ることさえ全く気づいていなかった．ラムダ記法の話だけ聞いてしまうと，それがあまりにも自明すぎて，ラムダ計算に関する非自明な理論などというものがあり得る可能性について想像することすら無くなってしまう．しかし，驚くべきことに，このような自明なラムダ記法の話から，想像以上に深いラムダの数学的理論が展開されるのである．

1.5. 結合子完全性の証明

ラムダ記法の解説を終えたので，結合子完全性をラムダ記法の言葉で言い直そう．結合子完全性 (定理 1.10) の主張をじっくりと眺めると，結合子計算の項 t に対して，

$$\lambda x_1 \dots x_n.t(x_1,\dots,x_n) \text{ の働きをする項 } \mathtt{a} \text{ が必ず存在する}$$

ということを述べていると分かる．つまり，結合子完全性とは，結合子計算によってラムダ計算をシミュレートできるということを述べていると言うこともできる．

まず，1 変数の場合についてアイデアを述べよう．ラムダ項 $\lambda x.t(x)$ を組み上げるには，結合子項の定義より，以下のパターンを考えればよい．

$$\lambda x.x \qquad\qquad \lambda x.y \qquad\qquad \lambda x.tu$$

ここで，y は結合子または x と異なる変数であり，t と u は項である．もちろん，左のラムダ項は，結合子項 $\mathtt{i}$ によって表され．中央のラムダ項は，結合子項 $\mathtt{k}y$ によって表される．

右のラムダ項は，まず変数を明示化して，$\lambda x.t(x)u(x)$ と書いておこう．このとき，結合子完全性を帰納的に証明していたとすれば，それぞれ $\lambda x.t(x)$ と $\lambda x.u(x)$ の働きをする結合子項 $\mathtt{a}_t$ と $\mathtt{a}_u$ を得ているはずである．このとき，$t(x)u(x)$ は $\mathtt{a}_t u(\mathtt{a}_u x)$ を表されるが，これはもちろん $\mathtt{sa}_t\mathtt{a}_u x$ のことである．したがって，上記の 3 つのラムダ項は，以下の項と対応する．

$$\mathtt{i} \qquad\qquad \mathtt{k}y \qquad\qquad \lambda x.\mathtt{sa}_t\mathtt{a}_u x$$

以上より，結合子完全性の証明はほとんど明らかであると思うが，一応，形式的な証明を与えよう．上記のアイデアを用いて，結合子によってラムダを以下のようにシミュレートする．

定義 1.11. 結合子項 t と変数 x が与えられたとき，項 $\Lambda x.t$ を以下によって帰納的に定義する．

$$\begin{aligned} \Lambda x.x &\equiv \mathtt{i} \\ \Lambda x.y &\equiv \mathtt{k}y \qquad\qquad (y \neq x \text{ が変数または結合子のとき}) \\ \Lambda x.tu &\equiv \mathtt{s}(\Lambda x.t)(\Lambda x.u) \end{aligned}$$

$\Lambda x.t$ はラムダ項 $\lambda x.t$ と同じ働きをすることを期待される結合子項であるが，あくまでこれは結合子項であり，ラムダ項そのものではないので，λ ではなく Λ という記号を用いている．

例 1.12. $\Lambda x.xx$ を得るためには，xx の構文木を葉から順に定義 1.11 に従って変形していく．

$$\frac{x \quad x}{xx} \xrightarrow{\Lambda x.} \frac{\mathtt{i} \quad \mathtt{i}}{\mathtt{sii}}$$

まず, 葉の x が $\mathtt{i}$ に変形され, 2 つの $\mathtt{i}$ を結合する際に $\mathtt{s}$ が左に配置される．よって, $\Lambda x.xx = \mathtt{sii}$ を得る．これは例 1.4 の入力データの複製コマンドと同じものである．

これが $\lambda x.xx$ の働きをすることは，例 1.4 で既に見た通りであるが，構文木を見るとこれは分かりやすい．右に入力 x を配置して，根から順に計算していくと，以下のような形になっている．

$$\frac{\dfrac{x}{\mathtt{i}\,x}\uparrow \quad \dfrac{x}{\mathtt{i}\,x}\uparrow}{\mathtt{sii}\,x}\uparrow \qquad\qquad \mathtt{sii}x \to \mathtt{i}x(\mathtt{i}x) \to xx$$

例 1.13. $\Lambda x.yx(xz)$ を得る際も同様に，構文木を葉から順に変形していけばよい．

$$\frac{\dfrac{y \quad x}{yx} \quad \dfrac{x \quad z}{xz}}{yx(xz)} \xrightarrow{\Lambda x.} \frac{\dfrac{\mathtt{k}y \quad \mathtt{i}}{\mathtt{s}(\mathtt{k}y)\mathtt{i}} \quad \dfrac{\mathtt{i} \quad \mathtt{k}z}{\mathtt{si}(\mathtt{k}z)}}{\mathtt{s}(\mathtt{s}(\mathtt{k}y)\mathtt{i})(\mathtt{si}(\mathtt{k}z))}$$

これが $\lambda x.yx(xz)$ の働きをすることも，構文木の根の右に x を配置して，根に向かって計算を進めていくことで視覚化できる．

$$\frac{\dfrac{\dfrac{y}{\mathtt{k}y\,x}\uparrow \quad \dfrac{x}{\mathtt{i}\,x}\uparrow}{\mathtt{s}(\mathtt{k}y)\mathtt{i}\,x}\uparrow \quad \dfrac{\dfrac{x}{\mathtt{i}\,x}\uparrow \quad \dfrac{z}{\mathtt{k}z\,x}\uparrow}{\mathtt{si}(\mathtt{k}z)\,x}\uparrow}{\mathtt{s}(\mathtt{s}(\mathtt{k}y)\mathtt{i})(\mathtt{si}(\mathtt{k}z))\,x}$$

以後，$\Lambda x.(\Lambda y.(\Lambda z.t))$ などは $\Lambda xyz.t$ のように省略する．

例 1.14. $\Lambda xy.x$ を得るためには，まずは $\Lambda y.x$ を得てから，次に $\Lambda x.\Lambda y.x$ を得る．

$$x \xrightarrow{\Lambda y.} \mathtt{k}x \qquad\qquad \frac{\mathtt{k} \quad x}{\mathtt{k}x} \xrightarrow{\Lambda x.} \frac{\mathtt{kk} \quad \mathtt{i}}{\mathtt{s}(\mathtt{kk})\mathtt{i}}$$

一般的に，項 $\Lambda x.t$ に含まれる変数は，項 t に含まれる変数から x を除いたものである．よって，t が x のみを変数として含むならば，$\Lambda x.t$ は変数を含まない．

■構造帰納法: 結合子完全性の形式的な証明を与えるためには，結合子項上の帰納法を用いる．自然数上の数学的帰納法には慣れ親しんでいると思うが，まずは自然数上の帰納法のおさらいをしよう．任意の自然数 n に対して，性質 $P(n)$ を証明したいというとき，我々は以下のように証明を進めるのであった．

- (基礎ステップ) 性質 $P(0)$ が成立していることを証明する．

- (帰納ステップ) 性質 $P(k)$ が成立していると仮定して，$P(k+1)$ を証明する．

同じアイデアより，任意の結合子項 t に対して，性質 $P(t)$ を証明したいとき，以下のようなステップで証明を進める．

- (基礎ステップ) 性質 $P(x), P(\mathtt{s}), P(\mathtt{k})$ が成立していることを証明する．ここで，x は変数である．
- (帰納ステップ) 性質 $P(t)$ と $P(u)$ が成立していると仮定して，$P(tu)$ を証明する．

構文木を用いて説明すると，$x, \mathtt{s}, \mathtt{k}$ というのは，構文木における葉の部分である．構文木のノード tu は t と u へ分岐する．つまり，基礎ステップでは「木の葉の部分で P が成立している」ということを示し，帰納ステップでは「枝分かれの先すべてで P が成立しているならば，分岐点でも P が成立している」ということを示す．

一見すると，自然数上の数学的帰納法より複雑そうであるが，結合子項の場合は，自然数上の帰納法に還元可能である．たとえば，結合子項の複雑性を以下によって定義する．

- 変数 x および結合子 $\mathtt{s}, \mathtt{k}$ の複雑さは 0 とする．
- 項 tu の複雑さは t と u の複雑さの大きい方に 1 を足したものとする．

例 1.15. 項 $\mathtt{ss}(\mathtt{k}x)\mathtt{k}$ の複雑さは 3 である．ここで，$\mathtt{ss}(\mathtt{k}x)\mathtt{k}$ の括弧を省略せずに書くと，$((\mathtt{ss})(\mathtt{k}x))\mathtt{k}$ となっていることに注意する．以下，青字によって，複雑さを表している．

$$\dfrac{\dfrac{\dfrac{\mathtt{s}\quad 0 \qquad \mathtt{s}\quad 0}{\mathtt{ss}\quad 1} \qquad \dfrac{\mathtt{k}\quad 0 \qquad x\quad 0}{\mathtt{k}x\quad 1}}{(\mathtt{ss})(\mathtt{k}x)\quad 2} \qquad \mathtt{k}\quad 0}{((\mathtt{ss})(\mathtt{k}x))\mathtt{k}\quad 3}$$

この複雑さの概念を用いれば，項上の帰納法は，以下のように表すこともできる．

- (基礎ステップ) 複雑さ 0 のすべての項 t に対して，性質 $P(t)$ が成立していることを証明する．
- (帰納ステップ) 複雑さ k 以下のすべての項 t に対して性質 $P(t)$ が成立していると仮定して，複雑さ $k+1$ のすべての項 t に対して性質 $P(t)$ を証明する．

「複雑さ n 以下のすべての項 t で $P(t)$ が成立している」という性質を $Q(n)$ と書けば，上記の帰納法は，性質 Q に関する自然数上の数学的帰納法に過ぎないことが分かる．結合子項上の帰納法の場合は，このように自然数上の帰納法とみなせることが分かったが，ただし，より一般的な木上の帰納法は，必ずしも自然数上の帰納法に還元できるとは限らないことには注意しておく．

■結合子完全性の証明: それでは，結合子項 $\Lambda x.t$ が本物のラムダ項 $\lambda x.t$ のような働きをするということの厳密な証明を与えよう．以下，項 t が与えられたとき，$t[u/x]$ によって，t の変数 x に項 u を代入した結果を表す．

補題 1.16. t と u を結合子項とする．このとき，$(\Lambda x.t)u \to t[u/x]$ が成り立つ．

Proof. t の構造に関する帰納法による．$t = x$ のとき，

$$(\Lambda x.x)u = \mathtt{i}u \to u = x[u/x].$$

続いて，$t = y$ が x と異なる変数であるか，$\mathtt{s}$ または $\mathtt{k}$ であるとき，

$$(\Lambda x.y)u = \mathtt{k}yu \to y = y[u/x].$$

最後に，$t = t_1t_2$ である場合，帰納的に $(\Lambda x.t_1)u \to t_1[u/x]$ かつ $(\Lambda x.t_2)u \to t_2[u/x]$ が成り立っていると仮定する．このとき，

$$(\Lambda x.t_1t_2)u = \mathtt{s}(\Lambda x.t_1)(\Lambda x.t_2)u \to (\Lambda x.t_1)u((\Lambda x.t_2)u)$$
$$\xrightarrow{\text{帰納的仮定}} t_1[u/x]t_2[u/x] = (t_1t_2)[u/x] = t[u/x].$$

以上より，結合子項上の帰納法によって，求める性質が得られる． □

*Proof (*定理 *1.10).* 任意の $i \leqslant n$ に対して，以下を証明すれば十分である．

$$(\Lambda x_1x_2 \dots x_n.t)u_1u_2 \dots u_i \to \Lambda x_{i+1} \dots x_n.t[u_1/x_1] \dots [u_i/x_i]$$

これが成り立つ理由として，$\Lambda x_1x_2 \dots x_n.t = \Lambda x_1.(\Lambda x_2 \dots x_n.t)$ であるから，これに u_1 を右から掛けると，補題 1.16 より $(\Lambda x_1x_2 \dots x_n.t)u_1 = (\Lambda x_1.(\Lambda x_2 \dots x_n.t))u_1 \to \Lambda x_2 \dots x_n.t[u_1/x_1]$ となる．次に u_2 を右から掛け，同様の変形を繰り返せばよい．よって，$\mathtt{a} = \Lambda x_1x_2 \dots x_n.t$ が求める項である． □

§2. 型なしラムダ計算

2.1. ラムダ計算のための準備

■ラムダ前項: ラムダ計算もまた，結合子計算と同様に，記号操作のシステムである．ただし，結合子項とは少し異なるタイプの項を扱う．ラムダ計算において許容される項は，以下のラムダ前項 (λ-preterm) と呼ばれるものである．

1. 変数記号 x はラムダ前項である．
2. M が λ-前項ならば，$\lambda x.M$ はラムダ前項である．
3. M, N が λ-前項ならば，MN はラムダ前項である．

バッカス-ナウア記法を用いれば，変数記号の集合 Var に対して，ラムダ前項は以下のように定義される．

$$t ::= x \mid tt \mid \lambda x.t \qquad (x \in \mathrm{Var})$$

あるいは証明木の形式で，前項の定義を記述しておくことも便利である．

$$\frac{}{x \text{ は } \lambda\text{-前項}}\ (1) \qquad \frac{N \text{ は } \lambda\text{-前項}}{\lambda x.N \text{ は } \lambda\text{-前項}}\ (2) \qquad \frac{M \text{ は } \lambda\text{-前項} \qquad N \text{ は } \lambda\text{-前項}}{MN \text{ は } \lambda\text{-前項}}\ (3)$$

この証明木形式は，記号列がラムダ前項であることの形式的証明を与える際に便利である．

例 2.1. $\lambda x.(\lambda y.x(yz))$ がラムダ前項であることは以下のように確認できる．

$$\dfrac{\dfrac{\dfrac{}{x \text{ は } \lambda\text{-前項}}\ (1) \qquad \dfrac{\dfrac{}{y \text{ は } \lambda\text{-前項}}\ (1) \qquad \dfrac{}{z \text{ は } \lambda\text{-前項}}\ (1)}{yz \text{ は } \lambda\text{-前項}}\ (3)}{x(yz) \text{ は } \lambda\text{-前項}}\ (3)}{\lambda y.x(yz) \text{ は } \lambda\text{-前項}}\ (2)}{\lambda x.(\lambda y.x(yz)) \text{ は } \lambda\text{-前項}}\ (2)$$

例 2.2. $(\lambda x.xx)(\lambda x.xx)$ がラムダ前項であることは以下のように確認できる．

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x \text{ は } \lambda\text{-前項}}\ (1) \qquad \dfrac{}{x \text{ は } \lambda\text{-前項}}\ (1)}{xx \text{ は } \lambda\text{-前項}}\ (3)}{\lambda x.xx \text{ は } \lambda\text{-前項}}\ (2) \qquad \dfrac{\dfrac{\dfrac{}{x \text{ は } \lambda\text{-前項}}\ (1) \qquad \dfrac{}{x \text{ は } \lambda\text{-前項}}\ (1)}{xx \text{ は } \lambda\text{-前項}}\ (3)}{\lambda x.xx \text{ は } \lambda\text{-前項}}\ (2)}{(\lambda x.xx)(\lambda x.xx) \text{ は } \lambda\text{-前項}}\ (3)$$

この木は，前項としての構文木であると考えることもできる．上記の例から，構文木としての本質的な部分だけを抽出しておけば，以下のようになっている．

$$\dfrac{\dfrac{\dfrac{x \qquad \dfrac{y \qquad z}{yz}}{x(yz)}}{\lambda y.x(yz)}}{\lambda x.(\lambda y.x(yz))} \qquad\qquad \dfrac{\dfrac{x \qquad x}{xx}}{\lambda x.xx} \qquad \dfrac{\dfrac{x \qquad x}{xx}}{\lambda x.xx}}{(\lambda x.xx)(\lambda x.xx)}$$

構文木は上下逆に書くことも多いと思われるが，本稿では後の都合から，項を下から上に分解していく．この向きに関する伏線は後で回収される．

ラムダ前項の略記についてもここで言及しておく．結合子項と同様に，結合律が成り立たないため，積に関する括弧を一般には取り外すことはできないが，xyz は $(xy)z$ の略記として用いる．また，$\lambda x.MN$ は $\lambda x.(MN)$ の略記であり，$(\lambda x.M)N$ の略記ではない．λ 抽象に関しては，$\lambda x_1.\lambda x_2.\ldots\lambda x_n.M$ および $\lambda x_1 x_2 \ldots x_n.M$ は，

$$\lambda x_1.(\lambda x_2.(\ldots(\lambda x_{n-1}.(\lambda x_n.M))\ldots))$$

の略記として用いる．

ラムダ計算とは，一言で言えば，ラムダ前項の記号操作を行う計算モデルである．ここからは，記号列操作の形式的定義を厳密に書き下すことを試みる．ラムダ計算は様々なプログラミング言語の基礎理論であるから，可能な限りコンピュータにも読めるような厳密な理論展開を心がけることにしよう．

■自由変数の定義: ラムダ項 M の中に現れる変数 x のうち，$\lambda x.(\cdots x \cdots)$ のような形で，λ 抽象の内部に入っているものを**束縛変数** (*bounded variable*) と呼び，そうでないものを**自由変数** (*free variable*) と呼ぶ．形式的には，M に現れる自由変数全体の集合 $\mathrm{FV}(M)$ を以下のように帰納的に定義する．

$$\begin{aligned}\mathrm{FV}(x) &= \{x\}\\ \mathrm{FV}(\lambda x.P) &= \mathrm{FV}(P)\backslash\{x\}\\ \mathrm{FV}(PQ) &= \mathrm{FV}(P) \cup \mathrm{FV}(Q)\end{aligned}$$

項 M が自由変数を持たない, つまり $\mathrm{FV}(M) = \varnothing$ であるとき，M を**閉項** (*closed term*) と呼ぶ.

例 2.3. $\lambda x.(\lambda y.x(yz))$ と $x(\lambda x.xy)$ の自由変数の集合を求める過程を以下のように青字で表す.

$$\dfrac{\dfrac{x\ \cdots\{x\} \qquad \dfrac{y\ \cdots\{y\} \qquad z\ \cdots\{z\}}{yz\ \cdots\{y,z\}}}{x(yz)\ \cdots\{x,y,z\}}}{\dfrac{\lambda y.x(yz)\ \cdots\{x,z\}}{\lambda x.(\lambda y.x(yz))\ \cdots\{z\}}} \qquad\qquad \dfrac{x\ \cdots\{x\} \qquad \dfrac{\dfrac{x\ \cdots\{x\} \qquad y\ \cdots\{y\}}{xy\ \cdots\{x,y\}}}{\lambda x.xy\ \cdots\{y\}}}{x(\lambda x.xy)\ \cdots\{x,y\}}$$

したがって，$\mathrm{FV}(\lambda x.(\lambda y.x(yz))) = \{z\}$ であり，$\mathrm{FV}(x(\lambda x.xy)) = \{x, y\}$ である.

■代入の定義: 我々は，日頃から「代入」という概念を難なく用いている．しかし，代入の厳密な定義を書き下してみようとすると，実は意外と難しいことが分かる．「$\lambda y.xy$ に現れる x に zz を代入せよ」と言われたら，$\lambda y.zzy$ と即答できるが，「$\lambda z.xz$ に現れる x に zz を代入せよ」と言われたら，どのように答えるだろうか．普通に考えれば，$\lambda z.zzz$ であるような気がするが，それで正しいだろうか.

$$(\lambda y.xy)\left[\frac{zz}{x}\right] \longrightarrow \lambda y.zzy \qquad\qquad (\lambda z.xz)\left[\frac{zz}{x}\right] \overset{?}{\longrightarrow} \lambda z.zzz$$

ところで，関数としての意味を考えると，$\lambda y.xy$ と $\lambda z.xz$ は全く同じものである．それにもかかわらず，これらの「同じ関数」に対して「自由変数 x への項 zz の代入」という全く同じ操作を行った結果は，$\lambda y.zzy$ と $\lambda z.zzz$ という「別の関数」になってしまっている．これは何かの問題を生じてしまわないだろうか．やはり，右側の代入結果には議論の余地があると思われる.

このように，代入する項の自由変数の中に束縛変数が含まれる場合の代入の定義には一考を要する．この場合の代入の定義は一先ず保留にしておいて，先に，左側の代入のように，特に議論を要しない代入，つまり代入する項の自由変数の中に束縛変数が含まれない場合の代入の形式的定義を与えておこう．以後，λ-前項 M, N と変数記号 x に対して，「M の中の x への N の代入」のことを $M[\frac{N}{x}]$, $M[N/x]$ または $M[x := N]$ のように書く．代入する項 N の自由変数の中に束縛変数

が含まれない場合には，代入の定義は以下のようにして帰納的に定義できる．

$$\begin{aligned} x[\tfrac{N}{x}] &= N \\ y[\tfrac{N}{x}] &= y \qquad (y \neq x) \\ (PQ)[\tfrac{N}{x}] &= P[\tfrac{N}{x}]Q[\tfrac{N}{x}] \\ (\lambda y.P)[\tfrac{N}{x}] &= \lambda y.\left(P[\tfrac{N}{x}]\right) \qquad (y \notin \mathrm{FV}(N)) \end{aligned}$$

それでは，$y \in FV(N)$ の場合に代入の定義をどうするか，という問題について議論していこう．

■ **α-同値性:** さて，束縛変数の記号は変えてしまっても，項の「意味」に変化はないと考えられる．たとえば，$\lambda x.x$ と $\lambda y.y$ の「意味」は全く同じだと言ってよいだろう．もちろん，項の「意味」の定義はまだ与えていないから，これを記号操作によって定式化しなければならない．アイデアは自明であるが，その自明なアイデアを厳密に定式化できるか，という点は重要である．「束縛変数を取り替えても同じ」という概念を厳密に定義したものは，**α-同値**と呼ばれ，$=_\alpha$ と書かれる．

$$\begin{aligned} P &=_\alpha P \\ \lambda x.P &=_\alpha \lambda y.\left(P[\tfrac{y}{x}]\right) \qquad (y \notin \mathrm{FV}(P)) \end{aligned}$$

つまり，束縛変数 x の出現をすべて新しい変数記号 y に置き換えてしまっても同じということである．α-同値なラムダ前項を暗黙に同一視しているとき，ラムダ前項のことを**ラムダ項** (λ-*term*) と呼ぶ．形式的には，ラムダ項とは，ラムダ前項の α-同値類のことである．

ラムダ項を考えている場合には，束縛変数は常に別の記号に取り替えることができる．特に，束縛変数と自由変数との間で記号の衝突が一切ない項のみを考えて良い．たとえば，自由変数記号 $x, y, z, \ldots$ と束縛変数記号 $\dot{x}, \dot{y}, \dot{z}, \ldots$ を別々に用意することができる．この場合，ラムダ項の定義は以下のように補正してもよい．

$$\frac{}{x \text{ は } \lambda\text{-項}}\ (1) \qquad \frac{N \text{ は } \lambda\text{-項}}{\lambda\dot{x}.N[\tfrac{\dot{x}}{x}] \text{ は } \lambda\text{-項}}\ (2) \qquad \frac{M \text{ は } \lambda\text{-項} \qquad N \text{ は } \lambda\text{-項}}{MN \text{ は } \lambda\text{-項}}\ (3)$$

このようにすれば，自由変数と束縛変数の間に記号の衝突は決して発生せず，代入の定義において，そもそも $\dot{y} \in \mathrm{FV}(N)$ の場合を考える必要がない．これが代入の問題を解決する一つの方法である．たとえば，自由変数 z と束縛変数 $\dot{z}$ は全く無関係な別の記号であるから，代入のプロセスは以下のように行われる．

$$(\lambda\dot{y}.x\dot{y})\left[\frac{zz}{x}\right] \longrightarrow \lambda\dot{y}.zz\dot{y} \qquad\qquad (\lambda\dot{z}.x\dot{z})\left[\frac{zz}{x}\right] \longrightarrow \lambda\dot{z}.zz\dot{z} =_\alpha \lambda\dot{y}.zz\dot{y}$$

また，α-同値性の恩恵はもう一つある．通常のラムダ前項の場合，$\lambda x.((\lambda x.xx)yx)y$ のようなものもラムダ前項になっている．x が 2 回束縛されているのでややこしいが，色を付けて束縛関係を明示すれば，$\lambda x.((\lambda x.xx)yx)y$ となっている．同じ変数記号が複数回束縛されると可読性が低くなるので，「同じ変数記号が複数回束縛されることがない」ように，常に束縛変数を取り替えることができる．

$$\lambda x.((\lambda x.xx)yx)y =_\alpha \lambda\dot{u}.((\lambda\dot{v}.\dot{v}\dot{v})y\dot{u})y$$

以後は，このような束縛変数の取替トリックを用いていくが，ただし，記述が煩雑になるので，多くの場合には，束縛変数の上にドットを明示的に付けることはしない．しかし，常に脳内では，束縛変数の上には見えないドットが付いていると思って，注意深く代入を行っていこう．また，以後は α-同値の記号 $=_\alpha$ を単に $=$ と書いてしまう．

例 2.4. $\lambda x.\lambda y.x(yz)$ に $z := xyz$ を代入してみよう．

$$\lambda x.\lambda y.x(yz) = \lambda\dot{x}.\lambda\dot{y}.\dot{x}(\dot{y}z) \xrightarrow{[\frac{xyz}{z}]} \lambda\dot{x}.\lambda\dot{y}.\dot{x}(\dot{y}(xyz)) = \lambda u.\lambda v.u(v(xyz))$$

■ **β-簡約:** ラムダ計算の理論は，ラムダ項の記号操作に過ぎないが，この記号操作に「意味」を生み出す概念が，関数適用である．ラムダ計算の理論においては，関数 $\lambda x.P$ に対して $x = Q$ を入力する，という操作を $(\lambda x.P)Q$ で表す．そして，その結果はもちろん $P[\frac{Q}{x}]$ である．これを記号列操作として厳密に定式化するものが **β-簡約**である．

$$(\lambda x.P)Q \to_\beta P\left[\frac{Q}{x}\right]$$

さらに，部分項の β-簡約はそれを含む項の β-簡約へと持ち上げることができる．

$$\begin{array}{lcl} P \to_\beta P' & \Longrightarrow & \lambda x.P \to_\beta \lambda x.P' \\ P \to_\beta P' & \Longrightarrow & MP \to_\beta MP' \\ P \to_\beta P' & \Longrightarrow & PN \to_\beta P'N \end{array}$$

この β-簡約が，λ-計算に意味を与える根幹的な概念，つまり計算のプロセスである．仰々しい名前の概念であるが，ただ計算を 1 ステップ進めるだけであるから，恐れる必要はない．$(\lambda x.P)Q$ の形の項は，**β-可約部** (β-*redex*) と呼ばれる．構文木の中では，可約部は，以下のような形状をしている．

$$\dfrac{\dfrac{P}{\lambda x.P} \qquad Q}{(\lambda x.P)Q}$$

例 2.5. $(\lambda xy.xyx)(zz)w$ を β-簡約してみよう．

$$(\lambda x.\lambda y.xyx)(zz)w \to_\beta (\lambda y.zzy(zz))w \to_\beta zzw(zz)$$

例 2.6. $(\lambda xy.yx)yz$ を β-簡約してみよう．ここで，束縛変数と自由変数の記号の衝突に注意しなければならない．

$$(\lambda x.\lambda y.yx)yz = (\lambda\dot{x}.\lambda\dot{y}.\dot{y}\dot{x})yz \to_\beta (\lambda\dot{y}.\dot{y}y)z \to_\beta zy$$

計算プロセスが終了するということは，上記の例のように，これ以上記号変形できない状態になったとき，つまり $M \to_\beta N$ となる項 N が存在しない項 M に辿り着いたときである．そのような項 M は，**β-正規形** (β-*normal form*) であると言う．項 M から N へ β-簡約の有限ステップでたどり着けるとき，$M \twoheadrightarrow_\beta N$ と書かれる．つまり，

$$M \twoheadrightarrow_\beta N \iff M = M_0 \to_\beta M_1 \to_\beta \cdots \to_\beta M_n = N$$

項 M の β-正規形が存在するとき，M は**弱正規化可能** (*weakly normalizable*) であるという．項 M をどのように β-簡約していっても，有限ステップで β-正規形に辿り着くとき，M は**強正規化可能** (*strongly normalizable*) であるという．つまり，弱正規化可能な項とは「適切な順序で計算を進めれば，いつか計算が停止する項」であり，強正規化可能な項とは「どのような順序で計算を進めても，いつか計算が停止する項」である．通常のコンピュータの計算の場合と同様に，無限ループに陥って計算が停止しなくなることが有り得る．

例 2.7. 弱正規化可能でない項が存在する．つまり，β-簡約は有限ステップで止まるとは限らない．たとえば，

$$(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta \dots$$

弱正規化可能であるが強正規化可能でない項が存在する．たとえば，$c \neq y$ について $(\lambda y.c)((\lambda x.xx)(\lambda x.xx))$ を考えよう．このとき，β-簡約の列として，以下のようなものがある．

$$(\lambda y.c)((\lambda x.xx)(\lambda x.xx)) \to_\beta c$$

$$(\lambda y.c)((\lambda x.xx)(\lambda x.xx)) \to_\beta (\lambda y.c)((\lambda x.xx)(\lambda x.xx)) \to_\beta (\lambda y.c)((\lambda x.xx)(\lambda x.xx)) \to_\beta \dots$$

前者は 1 ステップで正規形に辿り着くが，後者は延々 $(\lambda x.xx)(\lambda x.xx)$ を β-簡約し続けるだけであり，無限に正規形に辿り着かない．

以後，β-簡約の記法 $\to_\beta$ および $\twoheadrightarrow_\beta$ は，単に $\to$ と書いてしまうことが多い．

2.2. ラムダ計算における直和と直積

結合子計算によってラムダ計算を取り扱えることは分かったから，ここからはラムダ計算の能力について明かしていこう．

■条件分岐: 計算における最も重要な成分の一つは「条件分岐」である．この条件分岐をどのようにしてラムダ計算の世界で実装しようか．最初の目標は，ブール値を持つ述語 φ が与えられたとき，それが真ならば s を返し，偽ならば t を返す「ラムダ項」を構成することである．

$$\texttt{if } \varphi \texttt{ istrue then } s \texttt{ else } t$$

この命令の基本的なアイデアは，s, t の情報が与えられたとき，真ならば s を取り出し，偽ならば t を取り出すことである．つまり，「真」とは受け取った情報のひとつめを返す処理であり，「偽」とは受け取った情報のふたつめを返す処理だと考える．

$$\texttt{true} = \lambda xy.x \qquad\qquad \texttt{false} = \lambda xy.y$$

これはあくまで `true` と `false` のラムダ計算の言語における実装例の 1 つであって，こうでなければならないというわけではない．ある概念を何らかの言語で実装したいというとき，実装方法は 1 つではない．ともあれ，この前提の下で，条件分岐をラムダ項として表すことができる．

命題 2.8 (条件分岐)**.** 次の条件を満たすラムダ項 cond が存在する．

$$\texttt{cond}\ \varphi\, s\, t = \begin{cases} s & \textit{if}\ \varphi = \texttt{true}, \\ t & \textit{if}\ \varphi = \texttt{false}. \end{cases}$$

Proof. 条件分岐とは，ブール値 φ と 2 つの項 s, t を受け取って，φ の真理値によって s, t に対する上記の処理を行う操作であるから，

$$\texttt{cond} := \lambda\varphi st.\varphi st$$

と定義すればよい．実際に目的の条件を満たしていることを確認すると，

$$\texttt{cond true}\ s\, t \to \texttt{true}\ s\, t \to s$$
$$\texttt{cond false}\ s\, t \to \texttt{false}\ s\, t \to t$$

となっているから，主張は示された． □

以後はこの条件分岐ラムダ項 cond を用いていく．ただ，cond と書いても若干意味が伝わりづらいので，$\texttt{cond}\ \varphi\, s\, t$ の代わりに

$$\texttt{if}\ \varphi\ \texttt{istrue then}\ s\ \texttt{else}\ t$$

という記法を用いる．重要なことは，ラムダ計算の言語において上記のような英文があるわけではなくて，あくまで cond というひとつのラムダ項として実装されている，という点である．

■ペアリング: 計算とは，何らかのデータを取り扱うものであるが，複数のデータを同時に取り扱いたいことも多い．万能計算モデルは，複数のデータのまとまりを 1 つのデータとして取り扱える，という特性を持つ．たとえば，すべてのデータを自然数としてコードする自然数上の計算モデルでは，自然数の有限列はひとつの自然数としてコードされる．なぜなら，自然数の有限列は有限文字列に過ぎないから，有限ビット列として表すことができ，有限ビット列は自然数の 2 進表記であると思えるから，結局，これは自然数なのである．他にも，すべてのデータを有限文字列として表す計算モデルがあるが，この場合は，状況はもっと単純である．複数のデータとは，複数の有限文字列 $\sigma_1, \sigma_2, \ldots, \sigma_n$ であるが，この $[\sigma_1, \sigma_2, \ldots, \sigma_n]$ 自体が有限文字列であるから，有限文字列の有限列は有限文字列で表されるということである．

それでは，ラムダ項についても同様の処理が可能であろうか．つまり，有限個のラムダ項をまとめてひとつのラムダ項にしてしまうことが可能だろうか．まずは 2 つのラムダ項 s, t について考えることにしよう．ラムダ項の対は, 文字列の対ほど単純ではない．たとえば, ラムダ項 s, t を文字列のように st とまとめてしまうと，st の適用によって別の項に変化してしまい，s と t の情報を復元することができないからである．たとえば，$s = \lambda xy.y$ と $t = z$ を繋げると $st = (\lambda xy.y)z \to \lambda y.y$ のように計算が進み，z の情報は完全に失われる．したがって，単なる文字列結合ではなく，別の方法を考えなければならない．

ともあれ，まずは 2 つのデータを 1 つのデータにまとめる**ペアリング** (*pairing*) の概念から説明しよう．形式的には，ペアリングとは，2 つのデータ a, b を 1 つのデータ $\langle a, b\rangle$ にまとめる処理である．ただし，$\langle a, b\rangle$ は元のデータ a, b の情報を保持していなければならない．つまり，$\langle a, b\rangle$ から a, b の情報を復元する何らかの処理 π_0, π_1 が存在する．

$$a \xleftarrow{\pi_0} \langle a, b\rangle \xrightarrow{\pi_1} b$$

それでは，実際にラムダ計算の言語において，この処理が実装可能であることを証明しよう．

命題 2.9 (ペアリング)**.** 以下の条件を満たすラムダ項 $\mathtt{pair}, \pi_0, \pi_1$ が存在する．

$$\mathtt{pair}\, ab \downarrow, \quad \pi_0(\mathtt{pair}\, ab) = a, \quad \pi_1(\mathtt{pair}\, ab) = b$$

まず，証明のアイデアを述べよう．ラムダ項 s, t の対 $\langle s, t\rangle$ の定義のアイデアは，「項 s と t の両方の情報を保持している何か」であり，これは裏を返せば「項 s と t の両方の情報を利用できる」「項 s と t を用いた好きな処理を実行できる」という性質によって特徴付けられる．したがって，「s, t の対の情報」と「s と t を受け取って何らかの処理を実行する全パターン」を対応付けることができる．これは「与えられた任意の処理 p に対して，それに s と t を適用した実行結果を見ること」と同じことであろう．つまり，$\lambda p.pst$ である．このアイデアの下で，対 $\langle s, t\rangle$ の定義は $\langle s, t\rangle := \lambda p.pst$ として与えるのが妥当であろう．いま，x, y から $\langle x, y\rangle$ を作る処理が対関数であり．これは $\mathtt{pair} := \lambda xyp.pxy$ である．また，「s, t を受け取って s を取り出す処理」は $p_0 := \lambda st.s$ である，つまり $\langle s, t\rangle p_0 = s$ であるので，$\pi_0 = \lambda x.xp_0$ を考えればよい．

*Proof (*命題 *2.9).* $\mathtt{pair} = \lambda xyp.pxy$, $\pi_0 = \lambda x.x(\lambda st.s)$, $\pi_1 = \lambda x.x(\lambda st.t)$ によって定義する．このとき，

$$\pi_0(\mathtt{pair}\, ab) = (\lambda x.x(\lambda st.s))(\lambda p.pab) = (\lambda p.pab)(\lambda st.s) = (\lambda st.s)ab = a,$$
$$\pi_1(\mathtt{pair}\, ab) = (\lambda x.x(\lambda st.t))(\lambda p.pab) = (\lambda p.pab)(\lambda st.t) = (\lambda st.t)ab = b.$$

よって，主張は示された． □

以後，$\mathtt{pair}\, ab$ のことを $\langle a, b\rangle$ と書くことにする．このとき，ラムダ項の任意の有限列 $(a_i)_{i\leqslant n}$ は，次のように 1 つのラムダ項としてコードできる．

$$\langle a_0, a_1, a_2, \ldots, a_n\rangle := \langle\langle\langle a_0, a_1\rangle, a_2\rangle, \ldots\rangle, a_n\rangle.$$

2.3. ラムダ計算における算術的演算

■自然数: さて，結合子計算やラムダ計算は，項に対する計算システムであるが，たとえばこれで自然数の四則演算であるとか，自然数の計算をどのように実装すればよいだろうか．しかし，そもそもラムダ項の定義の中に自然数なんて無かったから，まず，最初にラムダ計算の言語で自然数と

いう概念を実装する必要がある．何かの概念をラムダ項として実装するというのはどういうことかというと，その概念をどうにかして関数だと考える，ということである．自然数をどうにかして関数だと考えよう．たとえば 2022 という数は，どんな関数だと考えられるだろうか．

このために，自然数とは何であったか，という点に立ち返ろう．ペアノ (Giuseppe Peano 1858-1932) によれば，自然数とは，最小元 0 と後続関数 $+1$ を持つ概念で数学的帰納法を満たすものである．ところで，ヒルベルトが述べるように，幾何学を展開するにあたって，点を椅子，線を机，平面をビールジョッキと呼んでも差し支えない．同様に，自然数論を展開するにあたって，最小元と後続関数の名前は何でもよい．たとえば，最小元の名前は z とし，後続関数の名前は s としてもよい．

$$\mathbf{1} \xrightarrow{0} \mathbb{N} \xrightarrow{+1} \mathbb{N} \qquad\qquad \mathbf{1} \xrightarrow{z} M \xrightarrow{\mathrm{s}} M$$

ここで，$\mathbf{1}$ は一点集合 $\{*\}$ を意味し，関数 $\mathbf{1} \xrightarrow{x} X$ は，X から元 x を 1 つ選択する行為を意味する．したがって，自然数の構造とは 2 本の矢印 $\mathbf{1} \xrightarrow{0} \mathbb{N} \xrightarrow{+1} \mathbb{N}$ あるいは $\mathbf{1} \xrightarrow{z} M \xrightarrow{s} M$ であるが，このようなものであれば何でもいいというわけではない．最小元 0 を始点として後続関数 $+1$ から作られるものだけが自然数である．つまり，自然数の構造 $\mathbf{1} \xrightarrow{0} \mathbb{N} \xrightarrow{+1} \mathbb{N}$ は余計な元を持たないシステムであり，この最小性が数学的帰納法を保証している．

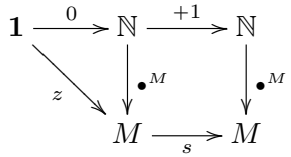
自然数のシステム $\mathbf{1} \xrightarrow{0} \mathbb{N} \xrightarrow{+1} \mathbb{N}$ の最小性より，どんな別の矢印システム $\mathbf{1} \xrightarrow{z} M \xrightarrow{s} M$ の内部にも入り込むことができる．言い換えれば，各自然数 $n \in \mathbb{N}$ の M における呼び名 n^M が存在することと言ってもよい．実際，この呼び名 n^M は具体的には以下のように与えられるべきだろう．

$$0^M = z, \qquad 1^M = sz, \qquad 2^M = s(sz), \qquad 3^M = s(s(sz)), \qquad 4^M = s(s(s(sz))), \ldots$$

形式的には，各自然数 $n \in \mathbb{N}$ の M における呼び名 $n^M \in M$ が存在して，

$$0^M = z \qquad\qquad (n+1)^M = s(n^M)$$

という条件を満たすということである．つまり，自然数 0 の M における呼び名は z であり，自然数 n の M における呼び名 n^M に後続関数の M における呼び名 s を適用すると，$n+1$ の呼び名 $(n+1)^M$ になっているということである．図式的には以下のように表す．



自然数に関するこのようなアイデアを用いて，自然数に対するラムダ項を与えよう．ここまでの議論から，自然数を関数とみなす余地は何かあっただろうか．アイデアは，

> 自然数の n とは，与えられた矢印システム $\mathbf{1} \xrightarrow{z} M \xrightarrow{s} M$ における n の呼び名 n^M を取り出す操作である

と考えることである．上の例では，0 の呼び名は z であり，1 の呼び名は sz であり，2 の呼び名は $s(sz)$ であり……というものであった．ここで，集合付きのシステム $\mathbf{1} \xrightarrow{z} M \xrightarrow{s} M$ で考えていたが，実際には矢印の情報 $\xrightarrow{z}\xrightarrow{s}$ だけあればよい．したがって，

> 自然数の n とは，2 つの矢印 $(\xrightarrow{s}, \xrightarrow{z})$ を入力として，対応する呼び名 $s^n z$ を出力する関数である．

ここで，$f^{n+1}x = f(f^n x)$ と定義する．以上のアイデアより，自然数 n に対応するラムダ項は以下のように定義できる．

定義 2.10 (自然数)**.** 各自然数 $n \in \mathbb{N}$ に対するラムダ項 $\underline{n}$ を以下のように定義する．

$$\underline{n} = \lambda sz.s^n z.$$

本稿のような自然数 n の実装 $\underline{n}$ は**チャーチ数項** (*Church numeral*) と呼ばれる．チャーチ数項以外にも，自然数をラムダ項としてコードする方法は無数にある．「自然数を実装できる」という点のみが重要なのであって，自然数の具体的な実装方法は何でもよい．文字列によって自然数をコードする方法がいくらでもあるように，ラムダ計算の言語で自然数をコードする方法は唯一ではない．

■自然数上の演算: チャーチ数項を利用して，ラムダ計算の内部で自然数の基本演算を取り扱うことができる．具体例をいくつか挙げよう．

例 2.11 (後続関数)**.** 与えられた自然数に 1 を加える関数，つまり後続関数を λ 項として表そう．定義より，

$$(\underline{n+1})sz \to s^{n+1}z = s(s^n z) \leftarrow s(\underline{n}sz)$$

であるから，$\mathtt{succ} = \lambda nsz.s(nsz)$ が後続関数を与える．実際に計算してみると，正しく後続関数になっている．

$$\mathtt{succ}\ \underline{n} \longrightarrow \lambda sz.s(\underline{n}sz) \longrightarrow \lambda sz.s(s^n z) = \underline{n+1}$$

ここから様々な関数を作れることを確認するために，ラムダの世界における n の呼び名 $\underline{n}$ の持つある性質に注目する．このラムダ項 $\underline{n}$ に関数 f と初期値 a を適用すると，a に f を n 回適用した結果になる．

$$\underline{n}fa = f^n a.$$

つまり，$\underline{n}fa$ は「n 回 f を a に適用する」と読む．これを利用して，様々な関数を作ってみよう．

例 2.12 (加法)**.** 加法 $x + y$ は，初期値 x に $+1$ を y 回適用した結果として表すことができる．「y 回 $+1$ を x に適用する」を表す項は $y\ \mathtt{succ}\ x$ である．したがって，加法は以下のようにラムダ項として実装できる．

$$\mathtt{add} = \lambda xy.y\ \mathtt{succ}\ x$$

実際に計算してみると，

$$\texttt{add}\ \underline{x}\underline{y} \longrightarrow \underline{y}\ \texttt{succ}\ \underline{x} \longrightarrow \texttt{succ}^{y}\underline{x} \longrightarrow \underline{x+y}$$

例 2.13 (乗法)**.** 乗法 $x \cdot y$ は，初期値 0 に $+x$ を y 回適用した結果として表すことができる．「y 回 $+x$ を 0 に適用する」を表す項は $y\ (\texttt{add}\ x)\ \underline{0}$ である．したがって，乗法は以下のようにラムダ項として実装できる．

$$\texttt{mult} = \lambda xy.y\ (\texttt{add}\ x)\ \underline{0}$$

実際に計算してみると，

$$\texttt{mult}\ \underline{x}\underline{y} \longrightarrow \underline{y}\ (\texttt{add}\ \underline{x})\ \underline{0} \longrightarrow (\texttt{add}\ x)^{y}\underline{0} \longrightarrow \underline{x \cdot y}$$

例 2.14 (冪乗)**.** 冪乗 x^y は，初期値 1 に $\times x$ を y 回適用した結果として表すことができる．「y 回 $\times x$ を 1 に適用する」を表す項は $y\ (\texttt{mult}\ x)\ \underline{1}$ である．したがって，冪乗は以下のようにラムダ項として実装できる．

$$\texttt{exp} = \lambda xy.y\ (\texttt{mult}\ x)\ \underline{1}$$

以上のように，様々な自然数上の基本演算はラムダ項によって実装できることが分かった．次は，自然数値に依存する条件分岐を実装してみよう．先ほど与えられたブール値が true か false かどうかの条件分岐をラムダ項で実装したが，その条件分岐プログラムを有効活用するには，何らかの式の真偽判定を行ってブール値を返す関数がなければならない．ここでは，与えられた値が 0 かどうかの判定を行うブール値関数を実装しよう．

例 2.15 (零判定)**.** 与えられた数 n が 0 かどうかの判定は，初期値 true に $\lambda x.\texttt{false}$ を n 回適用した結果として表すことができる．したがって，零判定は以下のようにラムダ項として実装できる．

$$\texttt{iszero} = \lambda n.n\ (\lambda x.\texttt{false})\ \texttt{true}$$

実際に計算してみると，

$$\begin{aligned}
&\texttt{iszero}\ \underline{0} \longrightarrow \underline{0}\ (\lambda x.\texttt{false})\ \texttt{true} \longrightarrow \texttt{true}\\
&\texttt{iszero}\ \underline{n+1} \longrightarrow \underline{n+1}\ (\lambda x.\texttt{false})\ \texttt{true} \longrightarrow (\lambda x.\texttt{false})^{n+1}\texttt{true} \longrightarrow \texttt{false}
\end{aligned}$$

例 2.16 (減法)**.** 次に，引き算をラムダ項として実装しよう．ただし，現時点では自然数を扱っているので，部分的引き算 (monus) を取り扱う．

$$x \dot{-} y = \max\{x - y, 0\}$$

このための第一ステップとして，$x \dot{-} 1$ を実装したいが，これが実はそんなに自明ではない．この準備として，次の性質を持つラムダ項 slow_succ を作ろう．

$$\texttt{slow\_succ}\ \langle \underline{n}, \underline{m} \rangle \longrightarrow \begin{cases} \langle \underline{1}, \underline{m} \rangle & \text{if } n = 0 \\ \langle \underline{n+1}, \underline{m+1} \rangle & \text{if } n > 0 \end{cases}$$

これについては，以下のように定義すればよい．

$$\texttt{slow\_succ} := \lambda x.\texttt{if iszero } \pi_0 x \texttt{ then } \langle \underline{1}, \pi_1 x\rangle \texttt{ else } \langle \texttt{succ } \pi_0 x, \texttt{succ } \pi_1 x\rangle$$

このとき，初期値 $\langle \underline{0}, \underline{0}\rangle$ に slow_succ を $n+1$ 回適用した計算結果を見てみよう．

$$\underline{n+1}\ \texttt{slow\_succ}\ \langle \underline{0}, \underline{0}\rangle \longrightarrow \texttt{slow\_succ}^{n+1}\ \langle \underline{0}, \underline{0}\rangle \longrightarrow \langle \underline{n+1}, \underline{n}\rangle$$

したがって，1 を引く演算は，この対の，右側を取り出せばよい．

$$\texttt{pred} := \lambda n.\pi_1(n\ \texttt{slow\_succ}\ \langle \underline{0}, \underline{0}\rangle)$$

実際に計算してみると，

$$\begin{aligned}
&\texttt{pred}\ \underline{0} \longrightarrow \pi_1(\underline{0}\ \texttt{slow_succ}\ \langle \underline{0}, \underline{0}\rangle) \longrightarrow \pi_1(\langle \underline{0}, \underline{0}\rangle) \longrightarrow \underline{0}\\
&\texttt{pred}\ \underline{n+1} \longrightarrow \pi_1(\texttt{slow_succ}^{n+1}\ \langle \underline{0}, \underline{0}\rangle) \longrightarrow \pi_1(\langle \underline{n+1}, \underline{n}\rangle) \longrightarrow \underline{n}
\end{aligned}$$

減法 $x \dot{-} y$ は，初期値 x に -1 を y 回適用した結果として表すことができる．「y 回 -1 を x に適用する」を表す項は y pred x である．したがって，減法は以下のようにラムダ項として実装できる．

$$\texttt{monus} = \lambda xy.y\ \texttt{pred}\ x$$

2.4. 再帰関数論

■自己言及と再帰: 次は，自己参照を持つプログラムを記述してみよう．自己参照を持つプログラム，これをラムダ計算で実装するアイデアとしては，不動点である．数学においては，関数 $f : X \to X$ の**不動点** (*fixed point*) とは，$f(x) = x$ なる $x \in X$ のことである．f の不動点のことを $\texttt{fix}\, f$ と書くとすると，$\texttt{fix}\, f = f(\texttt{fix}\, f)$ を満たす．数学には様々な不動点定理があり，ブラウワーの不動点定理や縮小写像に対するバナッハの不動点定理が代表例である．

同様に，ラムダ計算の世界にも，不動点定理がある．ラムダ計算の世界では，ただ不動点があるというだけでなく，関数 f を入力として，その関数 f の不動点を出力するようなラムダ項 fix を作ることができる．つまり，$\texttt{fix}\, f = f(\texttt{fix}\, f)$ となる項 fix であり，そのようなものは不動点コンビネータと呼ばれる．具体例をいくつか挙げよう．

例 2.17. カリーの Y-コンビネータは，以下のように構成される．

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

実際，$Yf = f(Yf)$ となることは次のように確認できる．

$$Yf \to (\lambda x.f(xx))(\lambda x.f(xx)) \to f((\lambda x.f(xx))(\lambda x.f(xx)))$$

しかし，上記の不動点コンビネータは $Yf \to f(Yf)$ であることを意味しない．$Yf \to f(Yf)$ となる強い不動点コンビネータの構成にはもう少し工夫が必要で，以下のように与えられる．

例 2.18. チューリングの Θ コンビネータは，以下のように構成される．

$$\Theta = (\lambda uf.f(uuf))(\lambda uf.f(uuf))$$

実際に確認してみよう．$U = \lambda uf.f(uuf)$ と書くことにすると，$\Theta = UU$ である．

$$\Theta f = UUf \longrightarrow_\beta (\lambda uf.f(uuf))Uf \longrightarrow_\beta f(UUf) = f(\Theta f).$$

不動点コンビネータの重要な応用は，自己参照である．

例 2.19. 自己参照の最も簡単な具体例を挙げると，何を入力としても「自分」を出力するプログラムがある．このプログラムを作るための下準備として，何を入力しても I を出力するプログラム $Q = \lambda x.I$ を書く．ここで，I はただのパラメータ，変数とする．このパラメータ I も動かすと宣言し，$\lambda I.Q$ を考えよう．これにチューリングの Θ-コンビネータを適用すると，$Q^\star := \Theta(\lambda I.Q)$ は $\lambda I.Q$ の不動点になっている．この計算を実行すると，

$$Q^\star = \Theta(\lambda I.Q) \longrightarrow (\lambda I.Q)(\Theta(\lambda I.Q)) = (\lambda I.Q)Q^\star = (\lambda I.\lambda x.I)Q^\star \longrightarrow \lambda x.Q^\star$$

したがって，$Q^\star$ は，何を入力しても $Q^\star$ 自身を出力するプログラムとなっている．

同様の理屈で，たとえば，I を 2 つ出力するプログラム $\lambda x.II$ に $\Theta(\lambda I.-)$ を適用すると，「自分」を 2 つ出力するプログラムになる．一般的に，I というパラメータを含むプログラムに $\Theta(\lambda I.-)$ を適用すると，I の部分が「自分」に置き換わったプログラムになる．まとめると，I をパラメータに持つプログラム Q があるならば，I を自分自身 $Q^\star$ のことだと解釈するプログラム $Q^\star$ がある．

命題 2.20 (再帰定理)**.** 任意のラムダ項 Q に対して，あるラムダ項 $Q^\star$ が存在して，次の性質を持つ．

$$Q^\star \longrightarrow Q[Q^\star/I]$$

Proof. $Q^\star = \Theta(\lambda I.Q)$ と定義する．このとき，

$$Q^\star = \Theta(\lambda I.Q) \longrightarrow (\lambda I.Q)(\Theta(\lambda I.Q)) = (\lambda I.Q)Q^\star = Q[Q^\star/I]$$

が成立している． □

解説. 再帰定理の直感的な説明を与えよう．固定した未知変数 I を使いながら，コンピュータ・プログラム $Q(I)$ を書いている，というシチュエーションを想定しよう．我々はその段階では I が何であるかは知らないが，とにかく I は自由に使えるので，プログラム内部に「プログラム I に n を入力した計算を実行せよ」などの命令を書き込むことができる．

さて，再帰定理 2.20 における $Q^\star$ を取ってきて，$Q(Q^\star)$ を考えよう．つまり，先ほど我々の書いたプログラムの中で I と書かれている部分を $Q^\star$ で上書きしたものが新しいプログラム $Q(Q^\star)$ である．すると，再帰定理より，プログラム $Q^\star$ を実行したものと $Q(Q^\star)$ を実行したものの計算結果は等しい．したがって，Q のプログラム内部に書かれている「プログラム I に n を入力した計算を実行せよ」という命令は「プログラム $Q^\star$ に n を入力した計算を実行せよ」という命令に等しい（図 1）．

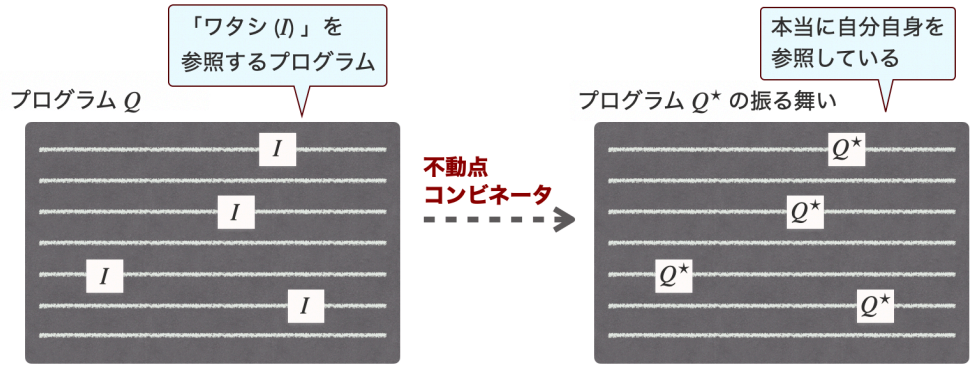


図 1　再帰定理

これが意味していることは何だろうか．我々はプログラム Q を書いている途中段階では，最終的な $Q^\star$ がどうなるかは知らないし，無限の自由度がある．それにも関わらず，我々が途中で書いた「プログラム I に n を入力した計算を実行せよ」と言う命令の意味は，常に「最終的な $Q^\star$ に n を入力した計算を実行せよ」を表す．つまり，我々はあたかも「最終的に書き上げる予定のプログラムが何であるか既に知っているかの如く」プログラムを記述することができるのである．

そういうわけで，以後，プログラム I すなわち『私』すなわち自己に言及したプログラムは自由に記述してよい．特に，例 2.19 のような自身のソースコードを出力するプログラムは，コンピュータ・プログラミングの文脈では**クワイン** (*Quine*) としてよく知られており，様々なプログラミング言語での実装例を見つけることができるだろう．

さて，再帰定理は数学的には一見単純であるが，それ故に強力である．基礎的なレベルから研究の最先端に至るまで，極めて広範な応用を持つ．計算可能性理論の入門的内容における最も重要な定理と言っても過言ではないだろう．ここでは，再帰定理の簡単な応用として，原始再帰法の実装を行う．原始再帰法とは，関数 g, h から次のような関数 f を作る操作である．

$$\begin{cases} f(0,x) = g(x) \\ f(n+1,x) = h(n,x,f(n,x)) \end{cases}$$

この関数 h を $\mathtt{rec}\, gh$ として表そう．この原始再帰作用素 $\mathtt{rec}$ は，不動点として実装できる．

命題 2.21 (原始再帰法)**.** 次の性質を持つラムダ項 $\mathtt{rec}$ が存在する．

$$\begin{aligned} \mathtt{rec}\, gh\underline{0} &\longrightarrow g \\ \mathtt{rec}\, gh\underline{n+1} &\longrightarrow h\underline{n}(\mathtt{rec}\, gh\underline{n}) \end{aligned}$$

Proof. 証明のアイデアを述べると，次のようなラムダ項 $\mathtt{rec}$ を構成すればよい．

$$\mathtt{rec}\ g\, h\, \underline{n} \longrightarrow \begin{cases} g & \text{if } n = 0 \\ h\ (\mathtt{pred}\ \underline{n})\ (\mathtt{rec}\ g\, h\, (\mathtt{pred}\ \underline{n})) & \text{if } n > 0 \end{cases}$$

注目点としては，項 rec の動作の中に rec が出てくるので自己参照を含んでいる．しかし，我々はすでに自己参照を含むプログラムの書き方を知っている．下準備として，rec の部分をワタシ I としたプログラムを書けばよい．

$$Q := \lambda ghn.\texttt{if iszero } n \texttt{ then } g \texttt{ else } h\ (\texttt{pred } \underline{n})\ (I\ g\,h\,(\texttt{pred } \underline{n}))$$

再帰定理 2.20 より，$\texttt{rec} := \Theta(\lambda I.Q)$ とすれば，

$$\texttt{rec}\ g\,h\,n \longrightarrow \texttt{if iszero } n \texttt{ then } g \texttt{ else } h\ (\texttt{pred } \underline{n})\ (\texttt{rec}\ g\,h\,(\texttt{pred } \underline{n}))$$

を得る．このとき，明らかに

$$\texttt{rec}\,g\,f\underline{0} = g, \qquad \texttt{rec}\,g\,f\underline{n+1} = f\underline{n}(\texttt{rec}\,g\,f\underline{n})$$

が導かれる．よって求める性質が示された． □

以上より，型なしラムダ計算で，原始再帰関数を実装することができた．原始再帰を用いて，繰り返し回数が事前にわかるループ命令を記述することができる．これによって，かなり多くのプログラムを記述できるが，これが全てではない．原始再帰では表せないプログラムというものがあり，それはたとえば繰り返し回数が事前に分からないループ命令である．つまり，いわゆる while ループがそれに該当する．具体例としては，何か条件 P があったときに，その条件を満たす自然数を探索するアルゴリズムなどである．型なしラムダ計算がチューリング完全である，通常のコンピュータと同性能である，と主張するには，もちろん while ループも実装できなければならない．

実際には，**μ-最小化** (μ-*minimization*) を実装できることを示せば十分である．μ-最小化とは，x を変数とする式 $P(x)$ が与えられたとき，$P(x)$ を満たす x が存在するならば，そのような最小の x を返す演算 $\mu x.P(x)$ である．ここで，そのような x が存在しないならば，$\mu x.P(x)$ は定義されない．ここでは，式 $P(y)$ が関数等式 $f(\bar{x}, y) = 0$ によって与えられている場合を考えよう．

定理 2.22 (μ-最小化)**.** 次のようなラムダ項 min が存在する*:* 任意のラムダ項 f および自然数 x に対して，もし $f\underline{x}\underline{y} \to \underline{0}$ となる自然数 y が存在するならば，そのような最小の y に対して，$\texttt{min}\,f\underline{x} \to \underline{y}$ となる．

Proof. アイデアとしては，次のようなアルゴリズム $Q^\star(y)$ を考えよう．これは，$f(\bar{x}, y) = 0$ がどうかを判定し，もしそうならば y を出力し，さもなくば $Q^\star(y+1)$ を呼び出す．最小値探索の際には，$Q^\star(0)$ を最初に実行して，計算が進むにつれて順々に $Q^\star(1), Q^\star(2), Q^\star(3), \ldots$ と呼び出す形になる．このアルゴリズムはもちろん，$Q^\star$ の自己参照を行っている．注目点としては，原始再帰法のときは自身の小さい入力値を参照していたが，μ-最小化においては自身の大きい入力値を参照している．ともあれ，まずは $Q^\star$ の自己参照の部分をワタシ I としたプログラムを書く．

$$Q := \lambda y.\texttt{if iszero } f\,\bar{x}\,y \texttt{ then } y \texttt{ else } I\,\underline{y+1}$$

再帰定理 2.20 より，$Q^\star := \Theta(\lambda I.Q)$ とすれば，

$$Q^\star y \simeq \texttt{if iszero } f\bar{x}y \texttt{ then } y \texttt{ else } Q^\star \underline{y+1}$$

となり，これはアイデアで述べたアルゴリズムの働きを行う．このとき，$\texttt{min} = \lambda f\bar{x}.Q^\star\underline{0}$ としよう．$\texttt{min}f\underline{x}$ の計算結果がどうなるかというと，$f\underline{x}\underline{y} \to \underline{0}$ となる自然数 y を 0 から順々に探索していって，そのような y が見つかったら計算を終え，y を出力する．ただし，もし $f\underline{x}\underline{y} \to \underline{0}$ となる自然数 y が存在しないのであれば，計算は停止せずに，無限に計算は続いていく．形式的には，数学的帰納法によって，min が求めるラムダ項であることを容易に示せる． □

こうして最小値探索がラムダ計算によって実装できる，ということがわかった．結論として，型なしラムダ計算で，コンピュータのありとあらゆる計算をシミュレートすることができる．つまり，型なしラムダ計算はチューリング完全な計算モデルである．ラムダ計算の利点は，理論的分析のしやすさと現実のプログラミングとの近さのバランスが優れているという点である．つまり，ラムダ計算の定義は，現実の多くのプログラミング言語と比較すると，非常にシンプルであり，これによって詳細な理論的分析が極めて容易である．その一方で，ラムダ計算における原始再帰法や最小値探索の実装などは，通常のコンピュータ・プログラムを組んでいく感覚とそこまで乖離しない．

2.5. ライスの定理

再帰定理のもうひとつの応用として，本節では，1953 年にヘンリー・ライス (Henry G. Rice) によって証明された**ライスの定理** (*Rice's theorem*) を紹介しよう．これは，次のような驚くべき主張をする定理である．

> 部分計算可能関数に関する非自明な性質 P が任意に与えられている．このとき，与えられたプログラム p の計算する関数が P を満たすか否かを判定するアルゴリズムは存在しない．

よく知られているチューリングの定理はあくまで「停止問題」というひとつの決定問題が計算不可能であることを示すものであるが，ライスの定理は一度で大量の決定問題の計算不可能性を導く．究極の計算不可能性定理の 1 つである．

プログラム p によって計算される $\mathbb{N}$ 上の部分関数を $\{\!\{p\}\!\}$ と書くことにしよう．ライスの定理によれば，たとえば，以下の判定問題はすべて計算不可能である．

$\{\!\{p\}\!\}$ は全域関数か？
$\{\!\{p\}\!\}$ は 2 値部分関数か？
$\{\!\{p\}\!\}$ の定義域は有限か？
$\{\!\{p\}\!\}$ は空関数か？

それでは，ライスの定理の正確な主張を述べよう．プログラム p と q が**外延的に等しい**とは，

$$(\forall n)\ \{\!\{p\}\!\}(n) \simeq \{\!\{q\}\!\}(n)$$

となることを意味する．プログラムに対する性質 A が**外延的**であるとは,

$$p \text{ と } q \text{ が外延的に等しい } \Longrightarrow [A(p) \iff A(q)]$$

性質 A が**自明**であるとは,「すべてのプログラムが性質 A を満たす」または「どのプログラムも性質 A を満たさない」のいずれか一方を満たすことを意味する．

定理 2.23 (ライスの定理)**.** A を非自明な外延的性質であるとする．このとき，与えられたプログラムが性質 A を満たすかどうかを判定するアルゴリズムは存在しない.

Proof. A を非自明な外延的性質とする．非自明なので，性質 A を満たすプログラム p と性質 A を満たさないプログラム q が存在する．いま，新たなプログラム r_e を次のように定義する．

- 入力 n:
- $\{\!\{e\}\!\}(I)$ の計算をシミュレートする．
- もし，この計算が有限時間で終わり，$\{\!\{e\}\!\}(I)\downarrow = 0$ であった場合:
  - 任意の入力 n に対して，$\{\!\{p\}\!\}(n)$ をシミュレートし，その計算結果を出力する．
- もし，この計算が有限時間で終わり，$\{\!\{e\}\!\}(I)\downarrow = 1$ であった場合:
  - 任意の入力 n に対して，$\{\!\{q\}\!\}(n)$ をシミュレートし，その計算結果を出力する．

再帰定理より，$I = r_e$ であるようにプログラム r_e を書き上げることができる．このようなプログラムは，現実に容易に書くことができ，次のような性質を満たす.

$$\{\!\{e\}\!\}(r_e)\downarrow = 0 \implies (\forall n)\ \{\!\{r_e\}\!\}(n) \simeq \{\!\{p\}\!\}(n) \implies A(r_e)$$
$$\{\!\{e\}\!\}(r_e)\downarrow = 1 \implies (\forall n)\ \{\!\{r_e\}\!\}(n) \simeq \{\!\{q\}\!\}(n) \implies \neg A(r_e)$$

さて，与えられたプログラムが性質 A を満たすかどうかを判定するアルゴリズムというものは,以下のようなプログラム e である.

$$\{\!\{e\}\!\}(n) = \begin{cases} 1 & \text{if } A(n) \\ 0 & \text{if } \neg A(n) \end{cases}$$

しかし，どんなプログラム e を取ってきても，$\{\!\{e\}\!\}(r_e)$ を考えると，判定に失敗していることが分かる．つまり，与えられたプログラムが性質 A を満たすかどうかを判定するアルゴリズムは存在しない. □

ライスの定理のラムダ計算版は，1960 年代にスコットとカリーによって独立に証明された．ラムダ項に関する性質 A が，β**-閉**とは,

$$M \equiv_\beta N \implies [A(M) \iff A(N)]$$

となることであり，性質 A が**自明**であるとは,「すべてのラムダ項が性質 A を満たす」または「どのラムダ項も性質 A を満たさない」のいずれか一方が成り立つことを意味する.

定理 2.24 (スコット-カリーの定理)**.** A を非自明かつ β-閉な性質であるとする．このとき，与えられたラムダ項が性質 A を満たすかどうかを判定するアルゴリズムは存在しない.

Proof. A を非自明な外延的性質とする．非自明なので，性質 A を満たすラムダ項 P と性質 A を満たさないラムダ項 Q が存在する．いま，与えられたラムダ項 E に対して，次のように新たなラムダ項 R_E を定義する．

$$R_E := \texttt{if } EI \texttt{ iszero then } P \texttt{ else } Q$$

再帰定理より，$R_E = I$ とすることができる．さて，与えられたプログラムが性質 A を満たすかどうかを判定するアルゴリズムというものは，以下のようなラムダ項 E である.

$$EN = \begin{cases} \underline{1} & \text{if } A(E) \\ \underline{0} & \text{if } \neg A(E) \end{cases}$$

しかし，どんなラムダ項 E を考えても，ER_E を考えると，判定に失敗していることが分かる．つまり，与えられたラムダ項が性質 A を満たすかどうかを判定するアルゴリズムは存在しない． □

ライスの定理の教えるところは，構文の枠を踏み越えて，意味を判断しようとすると，計算的にはうまくいかない.

■発展トピック: ライスの定理は，数学基礎論的観点からも少し興味深い点がある．数学基礎論と計算可能性の関連性が明確に現れたのは，1928 年のヒルベルトとアッカーマンの決定問題，つまり述語論理において論理式の恒真性を判定する方法が判定する機械的方法は存在することを問う問題であった．この問題を解決するために，そもそも「機械的方法」という概念とは何かを問う必要がある．そのために 1936 年にチューリングは現在チューリング・マシンとして知られる計算モデルを導入し，ヒルベルト-アッカーマンの決定問題を否定的に解決したのであった．つまり，述語論理において論理式の恒真性を判定する計算可能な方法は存在しない.

ライス的にヒルベルト-アッカーマンの問題を再考しよう．これは，「論理式の恒真性」という論理式の「意味論的性質」の判定を行うことを求める問題である．与えられた文字列が論理式を表すかどうか，含意を含むかどうか，などの構文論的性質であれば，容易に機械的判定ができるが，ヒルベルト-アッカーマンの問題は意味論的性質に踏み込んでしまった問題である．したがって，ライスの定理を学んだ後であれば，これが決定不可能問題であったというチューリングの定理は，さほど不思議ではないと思えるだろう．とはいえ，論理式に対して，ライスの定理を適用できるかというとそう簡単ではない．実際には，決定可能な公理系，つまり論理式の証明可能性の判定が可能な公理系はいくつも知られている.

ライスの定理は，万能チューリングマシンのような，ある程度，強力なシステムに対してしか成立しない．たとえば，チューリングマシンよりも能力の限定された計算モデルを考えることができるが，限定型計算モデルでは，一般的にはライスの定理は成り立つとは限らない．したがって，論理式に対しても，ある程度，強力なシステム上での意味論に対してしか，ライスの定理の類似物は

成立しない．論理式に対して，そのような強力なシステムの例としては，たとえばペアノ算術などがある．

ゲーデルは 1931 年に，ゲーデルの不完全性定理を証明した．ゲーデルは，これをプリンキピア・マテマティカおよびその関連体系に対するものとして提示したが，実際には，極めて多くの体系に対してゲーデルの不完全性定理は適用できる．たとえば，ペアノ算術などもその対象である．理論 T に対するゲーデルの第一不完全性定理は，T を含む無矛盾かつ完全な理論について，その理論における証明可能性の機械的判定方法は存在しないことを述べるものと言ってよい．このような T は，本質的決定不可能 (essentially undecidable) であると言われる．

実際には，ゲーデルの第一不完全性定理を少し補正すれば，「非自明な T-意味論的性質」はすべて計算可能な判定方法を持たないことを示すことができる．ここで，T-意味論的性質とは，T-証明可能性で同値なものを区別しない性質のことである．つまり以下を満たす性質 P である．

$$T \vdash \varphi \leftrightarrow \psi \quad \text{ならば} \quad (P(\varphi) \iff P(\psi))$$

定理 2.25 (ゲーデルの第一不完全性定理の系)**.** T をロビンソン算術を含み，無矛盾かつ再帰的公理化可能とし，P を非自明な T-意味論的性質であるとする．このとき，与えられた論理式が性質 P を満たすかどうかを判定するアルゴリズムは存在しない．

実際，以下の 2 条件は同値であることが知られている．

1. T は本質的決定不可能である．
2. 非自明な T-意味論的性質の判定はすべて計算不可能である．

したがって，上記に述べた論理式に対するライスの定理は，ゲーデルの第一不完全性定理と同値である．

§ 3. 単純型付きラムダ計算

3.1. 型付きラムダ計算のための準備

前節までで取り扱ったラムダ計算は，専門用語を用いれば，型なしラムダ計算 (untyped lambda calculus) というものである．『型』とは，あるデータの正体が何であるかを指し示す情報である．たとえば，この記号は整数を表す，この記号は文字列を表す，といったデータの種類を示すものである．型なしラムダ計算においては，ラムダ項が何かの関数を表していると考えるとしても，関数の定義域と値域が明示的に定まってはいない．この点が，通常の数学における関数の定義と若干様子が異なる．通常の数学では，関数に対して $f\colon X \to Y$ のような記号を用いる．これは，f は入力として X の元を受け付け，それに対して Y の元を出力するということを意味する．あるいは，f の入力の型は X であり，出力の型は Y であると言ってよい．さらに，f の型は $X \to Y$ である

と述べることができる．ラムダ計算においても，変数などに対する型を明示的に宣言するシステムがあり，それを型付きラムダ計算 (typed lambda calculus) と呼ぶ．

たとえば，足し算 + という演算を考えると，これは数を 2 つ受け取って，数を返す演算である．足し算という演算を整数上に制限すれば，数学においては，$+\colon \mathbb{Z}^2 \to \mathbb{Z}$ のように表す．この $\mathbb{Z}^2 \to \mathbb{Z}$ が + の型である．計算機科学においては，整数の型は `int` と書くことが多いので，計算機科学の記法を用いれば，

$$+\colon \texttt{int} \times \texttt{int} \to \texttt{int}$$

が + の型である．ただし，+ は整数上の足し算を表しているかもしれないし，有理数上の足し算を表しているかもしれないし，それは**コンテキストに依存する**．つまり，与えられた x, y について，$x + y$ の型が何であるかは，x と y の型を元にして推論される．

$$\frac{x\colon \texttt{int} \qquad y\colon \texttt{int}}{x + y\colon \texttt{int}} \qquad\qquad \frac{x\colon \texttt{rational} \qquad y\colon \texttt{rational}}{x + y\colon \texttt{rational}}$$

■カリー式単純型付きラムダ計算: **単純型付きラムダ計算** (*simply typed lambda calculus*) においては，基本型と関数型の組み合わせで作られる型を取り扱う．形式的には，型変数 $\alpha, \beta, \ldots$ たちと記号 $\to$ の組み合わせから作られる:

$$\sigma ::= \alpha \mid (\sigma \to \sigma)$$

略記に関する注意として，$\sigma \to \tau \to \rho$ と書いた場合には，$\sigma \to (\tau \to \rho)$ を表す．つまり，右結合的な表記を行う．

ラムダ項には，アプリオリには型は指定されていないが，変数の型を指定しておけば，項の型も定まる可能性がある．たとえば，x の型が σ で y の型が τ ならば，$\lambda x.y$ の型は $\sigma \to \tau$ であろう．このような，ラムダ項に含まれる変数の型指定のことを**コンテキスト** (*context*) あるいは**型環境**と呼ぶ．つまり，“$x\colon \sigma$” によって，変数 x の型を σ に指定することを意味するとすれば，コンテキストとは，変数の型指定の有限リストである．

$$\{x_1\colon \tau_1, \quad x_2\colon \tau_2, \quad \ldots, \quad x_n\colon \tau_n\}$$

我々の目標は，コンテキスト Γ が与えられた上で，ラムダ項 M の型 σ を推論することである．

■型の推論規則: ここから行う議論は，与えられたラムダ項の型を推論するというプロセスである．つまり，項の値の所属する範囲や，関数の始域や終域を求めたい．たとえば，通常の数学的関数に関しては，以下のような推論を行うことができる．

$$\frac{x \in X \implies f(x) \in Y}{f\colon X \to Y}\ (\to\texttt{Intro}) \qquad\qquad \frac{f\colon X \to Y \qquad x \in X}{f(x) \in Y}\ (\to\texttt{Elim})$$

左の (→Intro) 推論は，集合 X 内の入力 x に対して集合 Y 内に出力 $f(x)$ を持つならば，f は始域 X 終域 Y の関数とみなせるということである．右の (→Elim) 推論は，関数 f が始域 X 終域 Y であるとき，X の元 x を入力すると，Y の元 $f(x)$ が出力されるということである．

このアイデアをラムダ項の型推論へと持ち込もう．変数の型指定，つまりコンテキストを定めれば，他にも様々なラムダ項の型が定まる．具体的には，次の方法で型を推論していく．

$$\frac{}{\Gamma, x\colon \tau \vdash x\colon \tau}\ (\mathtt{Ax}) \qquad \frac{\Gamma, x\colon \sigma \vdash M\colon \tau}{\Gamma \vdash \lambda x.M\colon \sigma \to \tau}\ (\to\mathtt{Intro}) \qquad \frac{\Gamma \vdash M\colon \sigma \to \tau \qquad \Gamma \vdash N\colon \sigma}{\Gamma \vdash MN\colon \tau}\ (\to\mathtt{Elim})$$

$\Gamma \vdash M\colon \sigma$ の直感的な意味としては，「コンテキスト Γ の下で，項 M の型は σ であると推論できる」ということである．各推論を簡単に説明しておくと，以下のようなものである．

- (Ax) 変数 x の型が τ であるという宣言がコンテキストに含まれるならば「項 x の型は τ である」と推論する．
- ($\to$Intro) 「変数 x の型が σ という前提で，項 M の型が τ である」ならば「項 $\lambda x.M$ の型は $\sigma \to \tau$ である」と推論する．
- ($\to$Elim) 「項 M の型が $\sigma \to \tau$ であり，項 N の型が σ である」ならば「項 MN の型は τ である」と推論する．

定義 3.1. $\Gamma \vdash M\colon \sigma$ であるとき，Γ **で** M **は型** σ **を持つ**という．ラムダ項 M が**型付け可能**とは，あるコンテキスト Γ と型 σ が存在して，$\Gamma \vdash M\colon \sigma$ であることを指す．

λ-項 t が**型付け可能** (*typable*) とは，あるコンテキスト Γ において t の型が σ であることが従う．つまり，$\Gamma \vdash t\colon \sigma$ である．M の型推論は，M をどんどん部分項に解体していく M の構文木を考えればよい．葉は M に現れる変数が対応するが，ここに (Ax) で型が付けられる．

例 3.2. $\lambda x.\lambda y.x$ の構文木と型推論図を書いてみよう．

$$\cfrac{\cfrac{x}{\lambda y.x}}{\lambda x.\lambda y.x} \qquad \cfrac{\cfrac{\cfrac{}{x\colon \sigma,\ y\colon \tau \vdash x\colon \sigma}\ (\mathtt{Ax})}{x\colon \sigma \vdash \lambda y.x\colon \tau \to \sigma}\ (\to\mathtt{Intro})}{\vdash \lambda x.\lambda y.x\colon \sigma \to \tau \to \sigma}\ (\to\mathtt{Intro})$$

各推件において，右式が型付けを行いたいラムダ項，左式が自由変数のコンテキストとなっている．型推論の結果は一意ではないことに注意する．たとえば，型変数記号は別のものに置き換えてもよい．

$$\cfrac{\cfrac{\cfrac{}{x\colon \rho,\ y\colon \eta \vdash x\colon \rho}\ (\mathtt{Ax})}{x\colon \rho \vdash \lambda y.x\colon \eta \to \rho}\ (\to\mathtt{Intro})}{\vdash \lambda x.\lambda y.x\colon \rho \to \eta \to \rho}\ (\to\mathtt{Intro})$$

もちろん，最初からプログラム $\lambda x.\lambda y.x$ 中に束縛変数の型を宣言してあれば，型推論結果の曖昧性は排除される．たとえば，x が整数であり y が文字列であると最初から宣言してあるプログラム $\lambda x\colon \mathtt{int}.\lambda y\colon \mathtt{char}.x$ の型は，当然ながら $\mathtt{int} \to \mathtt{char} \to \mathtt{int}$ に一意に定まる．

例 3.3. $\lambda x.\lambda y.x(yz)$ の構文木は以下のように表すことができる．

$$\dfrac{\dfrac{\dfrac{x \qquad \dfrac{y \qquad z}{yz}}{x(yz)}}{\lambda y.x(yz)}}{\lambda x.\lambda y.x(yz)}$$

このとき，型推論図は以下のように与えることができる．

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x\colon \sigma \to \tau}\,\texttt{(Ax)} \qquad \dfrac{\dfrac{}{\Gamma \vdash y\colon \rho \to \sigma}\,\texttt{(Ax)} \qquad \dfrac{}{\Gamma \vdash z\colon \rho}\,\texttt{(Ax)}}{\Gamma \vdash yz\colon \sigma}\,\texttt{(→Elim)}}{\Gamma \vdash x(yz)\colon \tau}\,\texttt{(→Elim)}}{x\colon \sigma \to \tau,\ z\colon \rho \vdash \lambda y.x(yz)\colon (\rho \to \sigma) \to \tau}\,\texttt{(→Intro)}}{z\colon \rho \vdash \lambda x.\lambda y.x(yz)\colon (\sigma \to \tau) \to (\rho \to \sigma) \to \tau}\,\texttt{(→Intro)}$$

ここで，$\Gamma = \{x\colon \sigma \to \tau,\ y\colon \rho \to \sigma,\ z\colon \rho\}$ である．

型付け推論について，もう少し幅を取らない同値な省エネ記法がある．通常の型付け推論について，$\vdash$ の左側に同じような Γ が何度も出てきて，冗長である．実は，$\vdash$ 記号の右側だけ残して左側を消しても，曖昧さなく同値な推論図を表すことができる．上の例でも注目してもらいたいのは，$\vdash$ の左部の型付けは，推論木の葉 (Ax) の $\vdash$ の右部の型付けたちのリストとして発生しているという点である．したがって，最下部の結論の $\vdash$ の左部は，葉 (Ax) の $\vdash$ の右部の型付けを見ればよいのだが，実際は，葉 (Ax) の $\vdash$ の右部の型付けをしたものが，(→Intro) を行うたびに，$\vdash$ の左部から**排出**されて，打ち消されている．つまり，この排出の記法さえ上手く与えてやれば，$\vdash$ の右部だけしか書く必要はない．

具体的には，型推論を以下のように記述する．

$$x\colon \tau \qquad \dfrac{\begin{array}{c}[x\colon \sigma]\\ \vdots \\ M\colon \tau\end{array}}{\lambda x.M\colon \sigma \to \tau}\,\texttt{(→Intro)} \qquad \dfrac{M\colon \sigma \to \tau \qquad N\colon \sigma}{MN\colon \tau}\,\texttt{(→Elim)}$$

左は「x の型を τ とする」という宣言である．これはコンテキストに $x\colon \tau$ を加えるという宣言であると思ってもよい．右は説明するまでもないと思うので，中央について説明しよう．この括弧 $[x\colon \sigma]$ は，$x\colon \sigma$ を排出した，ということを意味する．記号的手続きとしては，(→Intro) 推論を行った場合，その推論の**上部**に現れる $x\colon \sigma$ には**好きなだけ括弧を付けてよい**，というものになる．括弧を付ける $x\colon \sigma$ の数は 0 個でもよい．ただし，曖昧さがないように，どこの型推論で排出を行ったかをラベル付けしておく．括弧で括られていない（排出されていない）葉がコンテキストである．

例 3.4. この記法による $\lambda x.\lambda y.x$ の型推論図を書いてみよう．

$$\dfrac{\dfrac{[x\colon \sigma]}{\lambda y.x\colon \tau \to \sigma}\,\texttt{(→Intro)}}{\lambda x.\lambda y.x\colon \sigma \to \tau \to \sigma}\,\texttt{(→Intro)}$$

説明をすると，最初の (→Intro) 推論では，0 個の $y\colon \tau$ を排出した．次の (→Intro) 推論では，1 個の $x\colon \sigma$ を排出した．これによって，$\vdash \lambda x.\lambda y.x\colon \sigma \to \tau \to \sigma$ が導かれる．

例 3.5. この記法による $\lambda x.\lambda y.x(yz)$ の型推論図を書いてみよう．

$$\dfrac{\dfrac{\dfrac{[x\colon \sigma \to \tau]^2 \qquad \dfrac{[y\colon \rho \to \sigma]^1 \qquad z\colon \rho}{yz\colon \sigma}\,(\to\mathtt{Elim})}{x(yz)\colon \tau}\,(\to\mathtt{Elim})}{\lambda y.x(yz)\colon (\rho \to \sigma) \to \tau}\,(\to\mathtt{Intro})^1}{\lambda x.\lambda y.x(yz)\colon (\sigma \to \tau) \to (\rho \to \sigma) \to \tau}\,(\to\mathtt{Intro})^2$$

排出されていない葉は $z\colon \rho$ のみであるから，$z\colon \rho \vdash \lambda x.\lambda y.x(yz)\colon (\sigma \to \tau) \to (\rho \to \sigma) \to \tau$ が導かれる．

例 3.6. $\lambda x.yxx$ の型推論図を書いてみよう．方法としては，まず構文木を書いてから，その形状に合わせて型を推論していく．

$$\dfrac{\dfrac{y \qquad x}{yx} \qquad x}{yxx}$$
$$\lambda x.yxx$$

$$\dfrac{\dfrac{\dfrac{y\colon \sigma \to \sigma \to \tau \qquad [x\colon \sigma]}{yx\colon \sigma \to \tau}\,(\to\mathtt{Elim}) \qquad [x\colon \sigma]}{yxx\colon \tau}\,(\to\mathtt{Elim})}{\lambda x.yxx\colon \sigma \to \tau}\,(\to\mathtt{Intro})$$

補題 3.7 (自由変数補題)**.** $\Gamma \vdash M\colon \sigma$ であるとき，M に含まれる自由変数はすべてコンテキスト Γ に入っている．

Proof. 型推論の複雑性に関する帰納法による．$\Gamma \vdash M\colon \sigma$ を導いた最後の推論に注目する．最後の推論が (Ax) である場合，

$$\dfrac{}{\Delta, x\colon \sigma \vdash x\colon \sigma}$$

ここで，$\Gamma = \Delta, x\colon \sigma$ かつ $M = x$ である．このとき，M の自由変数は x のみであり，これはコンテキスト Γ に含まれている．

最後の推論が (→Intro) であるとき，

$$\dfrac{\Gamma, x\colon \tau_1 \vdash P\colon \tau_2}{\Gamma \vdash \lambda x.P\colon \tau_1 \to \tau_2}$$

ここで $M = \lambda x.P$ である．このとき，$\mathrm{FV}(M) = \mathrm{FV}(P)\backslash\{x\}$ である．帰納的仮定より，$\mathrm{FV}(P) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$ であるから，$\mathrm{FV}(P) \subseteq \mathrm{dom}(\Gamma)$ が導かれる．

最後の推論が (→Elim) であるとき，

$$\dfrac{\Gamma \vdash P\colon \tau \to \sigma \qquad \Gamma \vdash Q\colon \tau}{\Gamma \vdash PQ\colon \sigma}$$

ここで $M = PQ$ であるから，$\mathrm{FV}(M) = \mathrm{FV}(P) \cup \mathrm{FV}(Q)$ である．帰納的仮定より，$\mathrm{FV}(P), \mathrm{FV}(Q) \subseteq \mathrm{dom}(\Gamma)$ であるから，$\mathrm{FV}(M) \subseteq \mathrm{dom}(\Gamma)$ が導かれる． □

さて，ある項 M に型 σ が付いた場合，実際には，M の部分項にも必ず型が付いている．つまり，たとえば，$M = \lambda x.P$ の形の場合には，$(\to \texttt{Intro})$ でしか型が付かないから，x と P にも型が付いていなければならない．同様に，$M = PQ$ の形の場合には，$(\to \texttt{Elim})$ でしか型が付かないから，P と Q の両方に型が付いているはずである．

次に，項を β-簡約をしても型は変化しないことを示そう．

補題 3.8. $\Gamma \vdash M : \sigma$ かつ $M \twoheadrightarrow_\beta N$ ならば $\Gamma \vdash N : \sigma$ である．

Proof. 一回の β-簡約では型が変化しない，つまり $\Gamma, x : \tau \vdash M : \sigma$ かつ $\Gamma \vdash N : \tau$ ならば $\Gamma \vdash M[\frac{N}{x}] : \sigma$ であることを示せば十分である．$M : \sigma$ の型推論は，M の構文木と同じ形状をしているので，型推論の葉の部分のいくつかに $x : \tau \vdash x : \tau$ が現れ得る．ここを $\Gamma \vdash N : \tau$ の型推論の木に置き換えて，元の型推論の部分の x をすべて N に置換しても，正しい型推論になっている．

$$\begin{array}{c} x : \tau \\ \vdots \\ M : \sigma \end{array} \qquad\qquad \begin{array}{c} \vdots \\ N : \tau \\ \vdots \\ M[N/x] : \sigma \end{array}$$

□

ラムダ項に型が付くなら，そのすべての部分項に型が付くというのは，先に見た通りである．したがって，M が部分項 $\lambda x.P$ を持つとき，$\lambda x.P$ には型 $\sigma \to \tau$ が付くが，このとき，x は型 σ 変数，P は型 τ 項であると考えられる．この場合，しばしば $\lambda x^\sigma.P^\tau$ あるいは $\lambda x : \sigma.P^\tau$ と書く．これは推論から得られる型であるが，逆に，事前に項の中で束縛変数の型指定を行ってしまうという方法があり，これがチャーチ式単純型付きラムダ計算である．つまり，チャーチ式においては，$\lambda x.P$ という項は用いず，$\lambda x^\sigma.P$ あるいは $\lambda x : \sigma.P$ という項を用いる．カリー式とチャーチ式の大きな違いは，カリー式の場合，$\lambda x.x$ に付く型は $\sigma \to \sigma$ や $\tau \to \tau$ など任意である

$$\frac{x : \sigma \vdash x : \sigma}{\vdash \lambda x.x : \sigma \to \sigma}$$

が，チャーチ式の場合は，項は $\lambda x^\tau.x$ のような形を取るため，この場合，型は $\tau \to \tau$ に確定する．実際，チャーチ式の場合，型が付くならば，必ずそれは一意に定まる．

3.2. 型推論アルゴリズム

ラムダ項の型の具体的な求め方について議論していこう．ここでは最も基本的なアルゴリズムのみを取り扱う．実際に，具体例を手に取って考えてみよう．

例 3.9. $\lambda x.\lambda y.yxx$ の型を推論したいとしよう．この場合，まずは項の構文木を書き，次に各部分

項の型を未知の型変数としておく．

$$\dfrac{\dfrac{\dfrac{y \qquad x}{yx} \qquad x}{yxx}}{\lambda y.yxx}}{\lambda x.\lambda y.yxx} \qquad \dfrac{\dfrac{\dfrac{[y\colon B] \qquad [x\colon A]}{yx\colon C}\ (\to\texttt{Elim}) \qquad [x\colon A]}{yxx\colon D}\ (\to\texttt{Elim})}{\lambda y.yxx\colon E}\ (\to\texttt{Intro})}{\lambda x.\lambda y.yxx\colon F}\ (\to\texttt{Intro})$$

ここで同じ変数には同じ型変数を与えておく必要がある．たとえば，ここでは変数 x の葉が 2 箇所現れるが，割り当てる型は同じ未知変数 A としておく．このとき，各型変数が満たすべき条件をリストアップすると，以下のようになる．

$$\begin{cases} B = A \to C \\ C = A \to D \\ E = B \to D \\ F = A \to E \end{cases}$$

後はこの型の連立方程式を解けばよい．具体的に代入を繰り返していくと，

$$\begin{cases} B = A \to C \\ C = A \to D \\ E = B \to D \\ F = A \to E \end{cases} \Longrightarrow \begin{cases} B = A \to A \to D \\ E = B \to D \\ F = A \to E \\ (C := A \to D) \end{cases} \Longrightarrow \begin{cases} E = (A \to A \to D) \to D \\ F = A \to E \\ (B := A \to A \to D) \\ (C := A \to D) \end{cases}$$

$$\Longrightarrow \begin{cases} F = A \to (A \to A \to D) \to D \\ (B := A \to A \to D) \\ (C := A \to D) \\ (E := (A \to A \to D) \to D) \end{cases}$$

ここで，各ステップで青字の代入操作を行っている．ともあれ，型の連立方程式が解けたので，型推論図に代入してみよう．

$$\dfrac{\dfrac{\dfrac{[y\colon A \to A \to D] \qquad [x\colon A]}{yx\colon A \to D}\ (\to\texttt{Elim}) \qquad [x\colon A]}{yxx\colon D}\ (\to\texttt{Elim})}{\lambda y.yxx\colon (A \to A \to D) \to D}\ (\to\texttt{Intro})}{\lambda x.\lambda y.yxx\colon A \to (A \to A \to D) \to D}\ (\to\texttt{Intro})$$

つまり，$\lambda x.\lambda y.yxx$ の型は $A \to (A \to A \to D) \to D$ である．

例 3.10. 次は $\lambda x.\lambda y.\lambda z.x(xy)(yz)$ の型推論を行う．先程と同様，まずは項の構文木を書く．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \quad \dfrac{x \quad y}{xy}}{x(xy)} \quad \dfrac{y \quad z}{yz}}{x(xy)(yz)}}{\lambda z.x(xy)(yz)}}{\lambda y.\lambda z.x(xy)(yz)}}{\lambda x.\lambda y.\lambda z.x(xy)(yz)}$$

次に各部分項の型を未知の型変数としておく．ただし，型推論に慣れてくると，たとえば (→Intro) の部分の型は明らかであるから，そこは先に代入してしまってもよい．

$$\dfrac{\dfrac{\dfrac{\dfrac{[x\colon A] \quad \dfrac{[x\colon A] \quad [y\colon B]}{xy\colon D}\,(\to\texttt{Elim})}{x(xy)\colon E} \quad \dfrac{[y\colon B] \quad [z\colon C]}{yz\colon F}\,(\to\texttt{Elim})}{x(xy)(yz)\colon G}\,(\to\texttt{Elim})}{\lambda z.x(xy)(yz)\colon C \to G}\,(\to\texttt{Intro})}{\lambda y.\lambda z.x(xy)(yz)\colon B \to C \to G}\,(\to\texttt{Intro})}{\lambda x.\lambda y.\lambda z.x(xy)(yz)\colon A \to B \to C \to G}\,(\to\texttt{Intro})$$

先程と同様，各型変数が満たすべき条件から型の連立方程式を構成し，それを解いていこう．今回は先程よりは少し異なった推論プロセスを挟む．

$$\begin{cases} A = B \to D \\ A = D \to E \\ B = C \to F \\ E = F \to G \end{cases} \Longrightarrow \begin{cases} B \to D = D \to E \\ B = C \to F \\ E = F \to G \\ (A := B \to D) \end{cases} \Longrightarrow \begin{cases} B = D \\ D = E \\ B = C \to F \\ E = F \to G \\ (A := B \to D) \end{cases}$$

ここで，赤字の部分に注目しよう．式 $B \to D = D \to E$ はどのような条件で成立するだろうか．一般的に，$X \to Y = Z \to W$ が成立する十分条件としては，前提と結論が等しい，つまり $X = Z$ かつ $Y = W$ という状況が考えられるだろう．我々は無数にあり得る解のひとつを求めたいだけなので，この十分条件だけを考えれば問題ない．つまり，上記のプロセスでは，式 $B \to D = D \to E$ を十分条件 $B = D$ かつ $D = E$ に変形したということである．それでは，式変形を進めよう．

$$\begin{cases} D = E \\ D = C \to F \\ E = F \to G \\ (A := D \to D) \\ (B := D) \end{cases} \Longrightarrow \begin{cases} E = C \to F \\ E = F \to G \\ (A := E \to E) \\ (B := E) \\ (D := E) \end{cases} \Longrightarrow \begin{cases} C \to F = F \to G \\ (A := E \to E) \\ (B := E) \\ (D := E) \\ (E := C \to F) \end{cases}$$

$$\Longrightarrow \begin{cases} C = F \\ F = G \\ (A := E \to E) \\ (B := E) \\ (D := E) \\ (E := C \to F) \end{cases} \Longrightarrow \begin{cases} C = G \\ (A := E \to E) \\ (B := E) \\ (D := E) \\ (E := C \to G) \\ (F := G) \end{cases} \Longrightarrow \begin{cases} (A := E \to E) \\ (B := E) \\ (C := G) \\ (D := E) \\ (E := G \to G) \\ (F := G) \end{cases}$$

最後に $E := G \to G$ を代入することで，$\lambda x.\lambda y.\lambda z.x(xy)(yz)$ の型 $((G \to G) \to (G \to G)) \to (G \to G) \to G \to G$ を得た．

以上，具体的なラムダ項に対する型推論を行った．そのプロセスは，型に関する連立方程式を解くというものであり，今回の例ではいずれも連立方程式を容易に解くことができた．しかし，一般的に，何らかのプログラムに対応するラムダ項から生成される型の連立方程式は，極めて巨大なサイズになることが容易に想像できる．そのような巨大な連立方程式でも，一般的に解を求める方法はあるのだろうか．

単純型付きラムダ計算の場合は，容易な解法があり，それが単一化 (unification) のアルゴリズムである．歴史的には，単一化アルゴリズムの萌芽的概念は，1930 年のジャック・エルブラン (Jacques Herbrand) による，いわゆるエルブランの定理の証明において現れていた．これは，数理論理学における述語論理の充足性問題に関する定理であるが，1960 年前後から自動定理証明というものが研究されるようになり，1960 年代にロビンソン (John Alan Robinson) が単一化アルゴリズムを大きなトピックとして取り上げ，形式的な研究の対象となった．元々の単一化アルゴリズムは論理式に対するアルゴリズムであるが，ここで扱うものは，型の方程式に対する単一化アルゴリズムである．後で解説するが，論理式の概念と型の概念には強い類似性があるので，論理式に対するアルゴリズムを型に対するアルゴリズムとして再利用可能なのは不思議なことではない．

単一化アルゴリズムは「自明な式の除去」「代入」「含意に関する式の分解」の 3 つのプロセスからなる．

- （自明な式の除去）連立方程式中の $A = A$ は除去可能．
- （代入）連立方程式中に現れる $X = A$ を選択し，他のすべての式の X 中に A を代入する．ただし，型変数 X が A に含まれない場合のみである．
- （含意の分解）連立方程式中に現れる $X \to Y = Z \to W$ を $X = Z$ と $Y = W$ の 2 つの式に分解する．

単純型付きラムダ計算において，型の連立方程式は，もし解を持つならば，この 3 つの操作の繰

り返しで解くことができる．もしこの 3 つの操作を適用できないまま，何らかの式が残った場合には，その連立方程式は解を持たないということである．型付け不可能なラムダ項から得られた連立方程式は解を持たないことに注意する．

例 3.11. $\lambda x.xx$ には型が付かないことを示そう．これまでと同様，構文木を書き，各部分項を未知の型変数とする．

$$\dfrac{\dfrac{x \qquad x}{xx}}{\lambda x.xx} \qquad\qquad \dfrac{\dfrac{x\colon A \qquad x\colon A}{xx\colon B}\ (\to\texttt{Elim})}{\lambda x.xx\colon A \to B}\ (\to\texttt{Intro})$$

このとき，各型変数が満たすべき式は $A = A \to B$ のみである．当然，自明な式の除去と願意の分解は適用できない．また，型変数 A が型 $A \to B$ の中に出現するので，代入も適用できない．したがって，どの操作も適用できないまま式が残ってしまう．これは，$\lambda x.xx$ が型付け不可能ということである．

例 3.12. $(\lambda x.xy)(zy)$ が型を持つようなコンテキストを見つけ，そのコンテキストの下での $(\lambda x.xy)(zy)$ の型を求めてみよう．これまでと同様，構文木を書き，各部分項を未知の型変数としておこう．

$$\dfrac{\dfrac{\dfrac{x \qquad y}{xy}}{\lambda x.xy} \qquad \dfrac{z \qquad y}{zy}}{(\lambda x.xy)(zy)} \qquad\qquad \dfrac{\dfrac{\dfrac{[x\colon A] \qquad y\colon B}{xy\colon D}\ (\to\texttt{Elim})}{\lambda x.xy\colon A \to D}\ (\to\texttt{Intro}) \qquad \dfrac{z\colon C \qquad y\colon B}{zy\colon E}\ (\to\texttt{Elim})}{(\lambda x.xy)(zy)\colon F}\ (\to\texttt{Elim})$$

単一化アルゴリズムによって，型の連立方程式を解くと，以下を得る．

$$\dfrac{\dfrac{\dfrac{[x\colon B \to F] \qquad y\colon B}{xy\colon F}\ (\to\texttt{Elim})}{\lambda x.xy\colon (B \to F) \to F}\ (\to\texttt{Intro}) \qquad \dfrac{z\colon B \to B \to F \qquad y\colon B}{zy\colon B \to F}\ (\to\texttt{Elim})}{(\lambda x.xy)(zy)\colon F}\ (\to\texttt{Elim})$$

つまり，コンテキスト $y\colon B$ および $z\colon B \to B \to F$ の下で，$(\lambda x.xy)(zy)$ の型は F である．記号的に書けば，$y\colon B,\ z\colon B \to B \to F \vdash (\lambda x.xy)(zy)\colon F$ ということである．

3.3. 型付きラムダ項の正規化定理

ラムダ項に型を付けるということは，そのラムダ項の高階関数としての正体が分かるということである．これは大きなメリットであるが，型付けの価値はそれだけではない．実を言えば，ラムダ項に単純型が与えられるということは，そのラムダ項の計算が有限ステップで停止することの保証となっている．一般に，ラムダ項には $(\lambda x.xx)(\lambda x.xx)$ のように正規形を持たない，すなわち計算が無限ループに陥るものがあった．これに対して，単純型付きラムダ項は必ず正規形を持つのである．これが型付けという行為の重要な応用のひとつである．

ラムダ項を高階関数としてみなせることと，計算が有限ステップで停止することに如何なる関係があるかについてのアイデアをまず軽く述べよう．基本的なアイデアは，単純型付きラムダ計算の本質的に唯一の計算ステップは関数適用であるという点である．いま，ラムダ項に型が付き，高

階関数 f とみなせるとしよう．関数 f に入力 x_0 を与えた結果 $f(x_0)$ がまだ関数ということはある．さらに入力 x_1 を与えた結果 $f(x_0)(x_1)$ がまだ関数ということもある．これを繰り返し，有限ステップ $f(x_0)(x_1)(x_2)\dots(x_n)$ で関数ではなく，具体的な値になり，計算が終了することを期待したい．

アイデアとしては，関数適用をする度に，関数適用を行う部分の型の「ランク」がどんどん下がっていくというものである．もし型に自然数や順序数[*2]でランク付けることができるならば，「ランク」が無限に下降することはない．型の「ランク」が無限に下降しないということは，計算は有限ステップで停止することを保証できるということである．

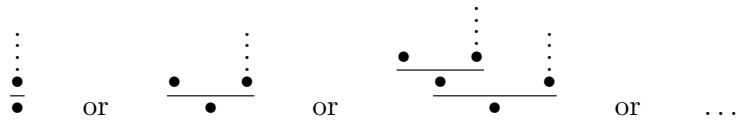
この議論を厳密に行うために，ラムダ項 M の正規形について分析しよう．構文木を分析すると，葉は変数，2 分岐ノードは適用，1 分岐だが葉でないノードは抽象である．

$$\frac{\dfrac{P}{\lambda x.P}\qquad Q}{(\lambda x.P)Q}$$

右側は，構文木の形状だけを強調して描いたものである．正規形の場合，$(\lambda x.P)Q$ の形の部分項を含まない．つまり，正規形であることと「抽象ノードは 2 分岐ノードの左側には決して現れない」ということは同値である．

命題 3.13. ラムダ項が正規形ならば，$\lambda x.N$ または xN の形である．ここで N は正規形である．

Proof. 根が葉ならば，$M = x$ の形であり，根が 1 分岐ならば，$\lambda x.N$ の形であり，N も正規形である．根が 2 分岐ならば，左側に抽象ノードは現れないので，葉または 2 分岐ノードである．これがずっと続くので，左端のパスには決して抽象ノードは現れない．つまり，左端のパスの終点は葉，つまり変数ノードである．



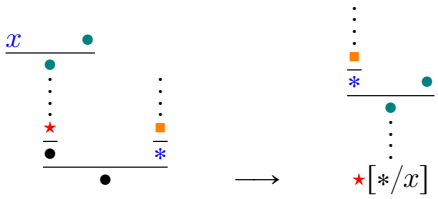
根から左端の葉に至るルートはすべて 2 分岐であるから，項は適用の繰り返し $xN_1N_2\dots N_k$ の形で表される．したがって，正規形は $\lambda x.N$ の形であるか，xN の形である． □

β-簡約 $(\lambda x.M)N \to M[\frac{N}{x}]$ の手続きを構文木を用いて分析すると，2 分岐ノード v の左側に 1 分岐ノードが現れたとき，この 2 分岐ノード v と左側の 1 分岐ノード（下図の • 部分）および右側の部分木 T_N（下図の $*$ 部分）を消して，x でラベル付けられた葉をすべて T_N で置き換える操作

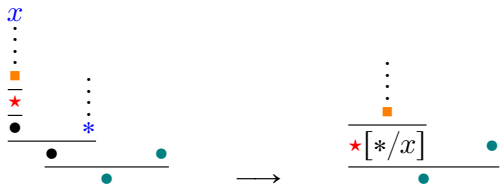
[*2] 順序数 (ordinal) とは，任意の下降列が有限で停止するという性質を持つ全順序（整列順序）の順序型である．このように計算が有限ステップで停止することの保証などに用いられる．

である．抽象的に書き表せば，

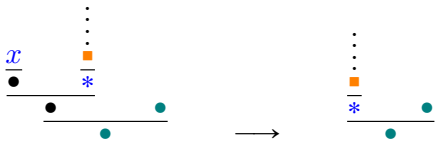
この簡約が，新たな可約部を作ることができる．まず，葉だった部分に新たな可約部ができる場合である．



もう一つは，簡約した下の部分に可約部ができる場合である．



これにはもう一つのパターンがある．



もう少し丁寧に，この議論を書き下そう．左側の 1 分岐ノードの上の木を T_M と呼ぶとしよう．つまり，可約部 $(\lambda x.M)N$ の構文木を見たとき，2 分岐ノード v がこの外側の適用を表し，T_N が N の構文木，T_M が M の構文木である．もし，これによって新しく可約部ができるのは，以下の場合である．

1. （上に新たに可約部ができる場合）x でラベル付けられた葉が 2 分岐ノードの左側であり，T_N の根が 1 分岐である．つまり，M は xP の形の部分項を含み，N は $\lambda y.Q$ という形である:

$$(\lambda x.\_\_(xP)\_\_)(\lambda y.Q) \quad \to \quad \_\_(\lambda y.Q)P\_\_$$

2. （削除部の上下で新たに可約部ができる場合）v の下が 2 分岐で，v はその左のノードであり，T_M の根が 1 分岐である．つまり，M は $\lambda y.P$ の形である．

$$(\lambda x.\lambda y.P)NQ \quad \to \quad (\lambda y.P[\tfrac{N}{x}])Q$$

3.（削除部の上下で新たに可約部ができる場合 2）v の下が 2 分岐で，v はその左のノードであり，T_M の根と葉が等しく，T_N の根が 1 分岐である．つまり，$M = x$ であり，$N = \lambda y.P$ という形である．

$$(\lambda x.x)(\lambda y.P)Q \quad \to \quad (\lambda y.P)Q$$

単純型 σ の複雑性 $|\sigma|$ を次によって導入する．型変数 α の複雑性は $|\alpha| = 0$ であり，$\sigma \to \tau$ の複雑性は $|\sigma \to \tau| = 1 + \max\{|\sigma|, |\tau|\}$ とする．

定理 3.14 (弱正規化可能性)**.** チャーチ式ラムダ項が型付け可能，つまり $\Gamma \vdash M : \sigma$ ならば，正規形へ辿り着く β-簡約の有限列

$$M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots \to_\beta M_n \in \mathrm{NF}_\beta$$

が存在する．

Proof. 項の中の簡約可能部 $(\lambda x.R)P$ に注目する．前に述べたように，M が型付けされているならば，M の部分項 R も型付けされている．チャーチ式であるから変数 x にも型が付いており，それを明示すると $(\lambda x^\tau.R^\rho)P$ であり，その型の複雑性 $|\tau \to \rho|$ を計算できる．項 M について，

$$|M| = \max\{M \text{ の可約部の型の複雑性 }\}$$

を考える．さらに，M の可約部のうち型複雑性 $|M|$ のものの個数 n_M を考える．このとき，値 $\omega|M| + n_M$ を考えよう．M の可約部のうち最も型複雑性が高いものを簡約すると，この値が下がることを示す．

ある可約部を簡約 $(\lambda x.R)P \to R[\frac{P}{x}]$ したとき，新たな可約部が生まれることは有り得る．これについては，次の 3 つのパターンがある．

$$(\lambda x.\_\_xP\_\_)(\lambda y.Q) \quad \to \quad \_\_(\lambda y.Q)P\_\_$$

$$(\lambda x.\lambda y.R)PQ \quad \to \quad (\lambda y.R[\tfrac{P}{x}])Q$$

$$(\lambda x.x)(\lambda y.P)Q \quad \to \quad (\lambda y.P)Q$$

型を明示的に書いて，型複雑性を計算すると，まずひとつめについては，簡約前は

$$(\lambda x^{\rho\to\mu}.\_\_xP^\rho\_\_)(\lambda y^\rho.Q^\mu)$$

であるから，型 $(\rho \to \mu) \to \tau$ という型である．一方，新しくできた簡約 $(\lambda y^\rho.Q^\mu)P$ の方は $\rho \to \mu$ であるから，型複雑性は下がっている．

ふたつめについて，簡約前は $(\lambda x^\tau.\lambda y^\rho.R^\mu)P^\tau$ であるから，型は $\tau \to (\rho \to \mu)$ である．一方，新しい可約部は $(\lambda y^\rho.R[\frac{P}{x}]^\mu)Q$ であるから，型は $\rho \to \mu$ であるから，型複雑性は下がっている．

最後に，3 番目については，簡約前は $(\lambda x^{\rho\to\mu}.x)(\lambda y^\rho.P^\mu)$ であるから，型 $(\rho \to \mu) \to (\rho \to \mu)$ であり，新しい可約部は $(\lambda y^\rho.P^\mu)Q$ であるから，型 $\rho \to \mu$ であり，型複雑性は下がっている． □

この定理の証明が述べていることは，型複雑性の高い β-可約部から順に簡約していけば，いつかは正規形に辿り着く，というものである．しかし，実際には，いかなる順に簡約していっても，正規形に辿り着くことが示される．

§4. カリー-ハワード対応

4.1. 命題論理の自然演繹

関数の記法 $f\colon A \to B$ と論理学における「ならば」の記法 $A \to B$ 共に，同じ矢印記法 $\to$ が用いられる．この 2 種類の矢印に何らかの関連性はあるだろうか．ラムダ計算における型推論は，大雑把に言えば，ラムダ項がどのような関数かを求める手続きであったが，これを論理学的な証明だと思うと，実は，関数記号 $f\colon A \to B$ と含意記号 $A \to B$ に形式的な関連性があることが分かるのである．

これを理解するためには，まずは論理学について学ばなければならない．このために，**命題論理** (*propositional logic*) を導入しよう．命題論理式とは，命題変数と論理結合子 $\wedge, \vee, \to$ から構成される式である．もちろん，これらの論理結合子はそれぞれ「かつ」「または」「ならば」を意図している．

$$A ::= p \mid A \wedge A \mid A \vee A \mid A \to A$$

通常は，命題論理においては否定 $\neg$ も扱うが，論理の簡易版から順に扱っていこう．「かつ」「または」「ならば」だけを扱って「否定」を扱わない論理は，最小論理 (minimal logic) と呼ばれる．

形式論理の文脈では，「A を仮定して B を導く」ということをしばしば $A \vdash B$ のような記号で表す．

$$A \vdash B\colon \quad A \text{ を仮定して } B \text{ を導く}$$

たとえば，A を仮定すれば A を導けるから，$A \vdash A$ である．論理的推論の形式化には様々なものがあるが，本稿では，ゲンツェンによる**自然演繹** (*natural deduction*) の体系と実質的に同値な体系を採用しよう．自然演繹では，各論理記号に対して，それぞれ**導入規則** (`Intro`) と**除去規則** (`Elim`) という 2 つの推論規則が割り当てられる．誤解を恐れずに言えば，それぞれ「その結論をどう導くか」「その仮定をどう用いるか」に関する規則だと思うと良い．

1. 導入規則: それが結論にあるとき，それをどのように演繹するか．
2. 除去規則: それが前提にある（あるいは導出済である）とき，それをどのように活用するか．

■含意 $\to$ の推論規則: 含意記号 $\to$ に対しては，以下の推論を行うのは妥当であろう．

$$\frac{A \vdash B}{\vdash A \to B}\ (\to\texttt{Intro}) \qquad\qquad \frac{\vdash A \to B \qquad \vdash A}{\vdash B}\ (\to\texttt{Elim})$$

含意 $\to$ の導入規則は，「A を仮定して B を導ける」ならば「前提なしで $A \to B$ を導ける」と

述べている．含意 $\to$ の除去規則は，いわゆるモーダス・ポーネンス (modus ponens) と呼ばれるもので，「A ならば B」であり「A である」ならば「B である」と述べている．

■連言 $\wedge$ の推論規則: 連言記号 $\wedge$ に対して，以下の推論は妥当であると考えられる．

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}\ (\wedge\texttt{Intro}) \qquad\qquad \frac{\vdash A_0 \wedge A_1}{\vdash A_i}\ (\wedge\texttt{Elim})$$

連言 $\wedge$ の導入規則は，「A と B を両方とも導ける」とき，「$A \wedge B$ を導ける」ということを述べている．連言 $\wedge$ の除去規則は，「$A_0 \wedge A_1$ を導ける」ならば「A_0 も A_1 も導ける」ということを述べている．

■選言 $\vee$ の推論規則: 選言記号 $\vee$ に対して，以下の推論は妥当であると考えられる．

$$\frac{\vdash A_i}{\vdash A_0 \vee A_1}\ (\vee\texttt{Intro}) \qquad\qquad \frac{A \vdash C \qquad B \vdash C}{A \vee B \vdash C}\ (\vee\texttt{Elim})$$

選言 $\vee$ の導入規則は，「A_0 を導ける」または「A_1 を導ける」のどちらかが成り立っているとき，「$A_0 \vee A_1$ を導ける」ということを述べている．選言 $\vee$ の除去規則は若干難しいが，下式「$A \vee B$ を仮定して C を導く」ことが我々の目標であり，仮定 $A \vee B$ を前提として利用してよい．したがって，A または B のいずれかは成り立っているはずなので，「A が成り立っている場合には……」「B が成り立っている場合には……」という場合分けを行うのが通例であろう．どちらの場合にせよ C を導けた場合には，「$A \vee B$ を仮定して C を導けた」ということである．

まとめると，場合分けによって「A が成り立つ場合には C も成り立つ」「B が成り立つ場合には C も成り立つ」ということが両方とも示せたならば，「$A \vee B$ を仮定して C を導ける」と結論付けることができる．

以上が命題論理記号 $\to, \wedge, \vee$ に対する推論規則である．命題論理記号には他に否定記号 $\neg$ もあるが，形式論理においては「A が否定される」ということは「A を仮定したら矛盾が導かれる」と同じものであるとみなされる[*3]．

$$\neg A \quad \equiv \quad A \to \bot$$

本稿でも，この流儀を採用する．ここで，$\bot$ は矛盾を表す記号である．

■矛盾記号 $\bot$ と否定記号 $\neg$: ここまででは，$\bot$ に関する規則を与えていないため，$\bot$ は好きな命題に置き換えることができるから，一般的には「矛盾」と一切関係ない．記号 $\bot$ が矛盾を表すと述べる際には，「矛盾」という概念を特徴付ける規則を考える必要がある．矛盾とは，通常は，「A の肯定と否定の両方が導かれてしまうこと」つまり，$A \wedge \neg A$ のことであると理解することが多い．したがって，矛盾記号 $\bot$ は以下の性質を満たすべきであろう．

$$\bot \quad \leftrightarrow \quad (A \wedge \neg A)$$

[*3] ただし，すべての論理において，否定概念がこのように解釈されるわけではなく，そうでない論理は，部分最小論理 (subminimal logic) において扱われる．

ただし，ここで注意点として，$\neg A$ の中に $\bot$ という記号が含まれるので，これを $\bot$ の定義と考えると，定義が循環してしまう．この問題を解決するため，もう少し矛盾 $A \wedge \neg A$ の性質に注目してみよう．

矛盾という概念は，論理学史の様々な場面で大きなトピックとして扱われてきた．たとえば，学術的理論が矛盾を孕んでいないかどうか，というのは大きな問題である．なぜなら，矛盾した理論には，以下で述べるように，致命的な欠陥があるためである．矛盾した理論を用いると，どのようなことが起きてしまうか，ということを分析すると，たとえば，以下のように，矛盾した理論を用いるとありとあらゆるものを否定できる．

命題 4.1 (否定爆発律)**.** 任意の論理式 A, B に対して，以下は証明可能である．

$$(A \wedge \neg A) \to \neg B.$$

ところで，通常の古典論理においては，二重否定除去 $\neg\neg A \to A$ と呼ばれる規則が成立する．否定爆発律を $B \equiv \neg C$ に対して適用したものと，二重否定除去を組み合わせれば，$(A \wedge \neg A) \to C$ を導くことができる．つまり，矛盾からは何でも導くことができる．

$$\frac{\vdash \neg\neg A}{\vdash A}\ (\neg\neg\texttt{Elim}) \qquad\qquad \frac{\vdash \bot}{\vdash A}\ (\bot\texttt{Elim})$$

かくして，矛盾した理論は，あらゆる論理式を演繹してしまうので，全く無価値な理論であると言える．ともあれ，矛盾 $\bot$ に関する規則は，除去規則「矛盾からすべてが導かれる」によって与えられる．この規則は，しばしば爆発律とも呼ばれる．

■推論規則のリスト: 上記の例では，導出関係 $A \vdash B$ を扱ったが，より一般的に，有限個の論理式を前提とする導出関係 $A_0, A_1, \ldots, A_n \vdash B$ をここでは取り扱う．これは，「$A_0, \ldots, A_n$ が全て成立すると仮定すれば，式 B を証明できる」あるいは「公理系 $\{A_0, \ldots, A_n\}$ の下で B が証明可能である」を意味する．以後は，有限個の仮定 $H_0, \ldots, H_n$ を $\bar{H}$ のような記法を用いて表す．

まず，当然ながら，$A \vdash A$ は自明に導出可能である．より一般的に，各 $i \leqslant n$ について，以下が成立する．

$$A_0, \ldots, A_n \vdash A_i$$

つづいて，命題論理記号 $\wedge, \vee, \to$ に対する推論規則を定義する．自然演繹では，各論理記号に対して，それぞれ**導入規則** (Intro) と**除去規則** (Eilm) という 2 つの推論規則が以下のように割り当てられる．

定義 4.2. 各命題論理記号に対して，以下のように 2 つずつ推論規則を割り当てる．

$$\frac{\bar{H}, A \vdash B}{\bar{H} \vdash A \to B}\ (\to\texttt{Intro}) \qquad\qquad \frac{\bar{H} \vdash A \to B \qquad \bar{H}' \vdash A}{\bar{H}, \bar{H}' \vdash B}\ (\to\texttt{Elim})$$

$$\frac{\bar{H} \vdash A \qquad \bar{H}' \vdash B}{\bar{H}, \bar{H}' \vdash A \wedge B}\ (\wedge\texttt{Intro}) \qquad \frac{\bar{H} \vdash A_0 \wedge A_1}{\bar{H} \vdash A_i}\ (\wedge\texttt{Elim})$$

$$\frac{\bar{H} \vdash A_i}{\bar{H} \vdash A_0 \vee A_1}\ (\vee\texttt{Intro}) \qquad \frac{\bar{H}, A \vdash C \qquad \bar{H}', B \vdash C}{\bar{H}, \bar{H}', A \vee B \vdash C}\ (\vee\texttt{Elim})$$

各推論規則は，上式を導出できている場合，下式も導出できることを意味する．各規則の意味について，既に説明した通りであるが，慣れないうちは一旦 $\bar{H}$ と $\bar{H}'$ を取り除いて考えると理解しやすい．また，各導出関係の前提部分は，仮説の有限集合であり，たとえば列 $\bar{H}, \bar{H}'$ は和集合 $\bar{H} \cup \bar{H}'$ と同一視してよいものとする．

■自然演繹の縦書き法: 自然演繹の具体的な証明を何度か記述すると分かるが，自然演繹も導出関係 $\vdash$ の左の仮定部分が証明図の中で何度も繰り返し現れ，かなり冗長となる．このため，基本的には，自然演繹に関して以下の別記法を用いる．アイデアは，導出関係 $A \vdash B$ を 90 度回転させて，縦向きにするというアイデアである．

$$A \vdash B \qquad \rightsquigarrow \qquad \begin{array}{c} A \\ \vdots \\ B \end{array}$$

つまり，仮定 A は上へと持ち上げられ，結論 B はそのまま下に配置される．自然演繹の証明図は基本的には下から構成するものであり，常に「仮定」と「ゴール」を意識しておくと良い．

■含意 $\rightarrow$ の推論規則 (縦書き): この縦書き法に基づくと，まず「ならば $\rightarrow$」に関する規則は以下のように書き直せる．

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \rightarrow B}\ (\rightarrow\texttt{Intro}) \qquad \frac{A \rightarrow B \qquad A}{B}\ (\rightarrow\texttt{Elim})$$

「$\rightarrow$ 導入規則」の図の意味について説明しよう．結論 $A \rightarrow B$ を証明するためには，A を仮定して B を導出すればよい．つまり，ゴール $A \rightarrow B$ がゴール B へと変更され，仮定 $[A]$ という手札が手に入った．証明が進むにつれ，視点が下から上へと徐々に上がっていくという点に注意しよう．「$\rightarrow$ 除去規則」については，略記でもほとんど変化は無いので，説明するまでもないだろう．

■連言 $\wedge$ の推論規則 (縦書き): つづいて，「かつ $\wedge$」に関する規則は，縦書き法では以下のように書き直せる．

$$\frac{A \qquad B}{A \wedge B}\ (\wedge\texttt{Intro}) \qquad \frac{A_0 \wedge A_1}{A_i}\ (\wedge\texttt{Elim})$$

これも説明する必要はないと思われる．

■選言 $\vee$ の推論規則 (縦書き): 最後に,「または $\vee$」に関する規則は,縦書き法では以下のように書き直せる.

$$\frac{A_i}{A_0 \vee A_1}\ (\vee\texttt{Intro}) \qquad\qquad \frac{\begin{array}{ccc} & [A] & [B] \\ A \vee B & \vdots & \vdots \\ & C & C \end{array}}{C}\ (\vee\texttt{Elim})$$

「$\vee$ 除去規則」はかなり非直感的なので,詳細に説明しよう.まず,「$\vee$ 除去規則」はこの図だけを静的に見ても全く意味が分からない.証明図の背後には,「証明する」という動的な行為が伴っていることを意識する必要がある.つまり,我々は,証明の過程の中で証明図を徐々に構築しており,各推論規則は,証明図の構築を 1 ステップ進めるプロセスなのである.推論規則を動的に見ると,「$\vee$ 除去規則」とは,以下のように,左図を右図に変化させる規則である.

$$\begin{array}{c} A \vee B \\ \vdots \\ C \end{array} \quad \rightsquigarrow \quad \frac{\begin{array}{ccc} & [A] & [B] \\ A \vee B & \vdots & \vdots \\ & C & C \end{array}}{C}\ (\vee\texttt{Elim})$$

つまり,左図では,我々は仮定 $A \vee B$ の下で結論 C を導こうと試みていた.このとき「$\vee$ 除去規則」は「仮定 $A \vee B$ を用いる」という宣言である.仮定 $A \vee B$ より,A または B のどちらかが成立するので,「A が成立する場合」と「B が成立する場合」で場合分けが行われる.これが証明図における中央と右の分岐である.

1. 中央の分岐は「A が成立する場合」なので,仮定 $[A]$ が手札として手に入る.
2. 右の分岐は「B が成立する場合」なので,仮定 $[B]$ が手札として手に入る.
3. 場合分けのいずれの場合でも C を示すことができればよいので,どちらの分岐においてもゴールは C である.
4. 左の分岐については,ここで仮定 $A \vee B$ を使用したという宣言だと思っても差し支えはない.仮定 $A \vee B$ を証明したければ,その分岐の先を進んでもよい.しかし,$A \vee B$ を仮定として認めるのであれば,その先にそれ以上進む必要はない.

4.2. ラムダ項の型付け = 自然演繹の証明

ラムダ項の型付けは,ラムダ項が如何なる高階関数かを推論する手続きであった.我々がこれから見るものは,どのような高階関数かということを推論するという行為が,命題を証明するという行為に対応しているということである.

例 4.3. たとえば $\lambda xyz.xz(yz)$ というラムダ項を考えよう.一応,$xz(yz)$ が $(xz)(yz)$ の略記だと

いうことを思い出せば，この構文木は次のような形になっている．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x \quad z}{xz} \quad \dfrac{y \quad z}{yz}}{xz(yz)}}{\lambda z.xz(yz)}}{\lambda yz.xz(yz)}}{\lambda xyz.xz(yz)}$$

以前のように，ラムダ項 $\lambda xyz.xz(yz)$ の構文木に沿って型推論をしていけば，これはたとえば次のように型付け可能である．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x\colon A \to B \to C]^3 \quad [z\colon A]^1}{xz\colon B \to C}\ (\to\mathtt{Elim}) \quad \dfrac{[y\colon A \to B]^2 \quad [z\colon A]^1}{yz\colon B}\ (\to\mathtt{Elim})}{xz(yz)\colon C}\ (\to\mathtt{Elim})}{\lambda z.xz(yz)\colon A \to C}\ (\to\mathtt{Intro}^1)}{\lambda yz.xz(yz)\colon (A \to B) \to A \to C}\ (\to\mathtt{Intro}^2)}{\lambda xyz.xz(yz)\colon (A \to B \to C) \to (A \to B) \to A \to C}\ (\to\mathtt{Intro}^3)$$

したがって，ラムダ項 $\lambda xyz.xz(yz)$ の型は $(A \to B \to C) \to (A \to B) \to A \to C$ であると推論できた．さて，この型推論図からラムダ項の部分を削除してみよう．すると，以下のように，直観主義論理における論理式 $(A \to B \to C) \to (A \to B) \to A \to C$ の証明を得る．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A \to B \to C]^3 \quad [A]^1}{B \to C}\ (\to\mathtt{Elim}) \quad \dfrac{[A \to B]^2 \quad [A]^1}{B}\ (\to\mathtt{Elim})}{C}\ (\to\mathtt{Elim})}{A \to C}\ (\to\mathtt{Intro}^1)}{(A \to B) \to A \to C}\ (\to\mathtt{Intro}^2)}{(A \to B \to C) \to (A \to B) \to A \to C}\ (\to\mathtt{Intro}^3)$$

つまり，ラムダ項の型推論図からラムダ項を消去して，型の部分だけに注目すると，命題の証明図になっているのである．型推論の文脈では $(A \to B \to C) \to (A \to B) \to A \to C$ は型であるが，ラムダ項を忘れると $(A \to B \to C) \to (A \to B) \to A \to C$ は論理式になっている．型の文脈では $\to$ は関数の矢印であるが，命題の文脈では $\to$ は「ならば」の矢印である．関数の矢印の話が，何故か「ならば」の矢印の話になっていた！　ラムダ項の型推論をしていたと思っていたら，命題の証明が行われていた……というのは驚くべきことのように思うかもしれないが，原理は単純である．

■型推論から自然演繹へ: まず，型推論規則について思い出そう．コンテキスト Γ が与えられたとき，$|\Gamma| = \{\sigma \mid (x\colon \sigma) \in \Gamma\}$ とする．ラムダ項の型付け規則から項の部分を取り除くと，以下のようになっている．

$$\dfrac{}{|\Gamma|, \tau \vdash \tau}\ (\mathtt{Ax}) \qquad \dfrac{|\Gamma|, \sigma \vdash \tau}{|\Gamma| \vdash \sigma \to \tau}\ (\to\mathtt{Intro}) \qquad \dfrac{|\Gamma| \vdash \sigma \to \tau \quad |\Gamma| \vdash \sigma}{|\Gamma| \vdash \tau}\ (\to\mathtt{Elim})$$

これは直観主義命題論理の規則とまったく同じものである．したがって，ラムダ項の型付けの意味で $\Gamma \vdash M\colon \sigma$ が成り立つならば，直観主義命題論理の推論の意味で $|\Gamma| \vdash \sigma$ が成立する．実際，ラムダ項 M の構文木が $|\Gamma| \vdash \sigma$ の証明図の骨格を与える．このようなものは具体例を幾つも見てみるのがよい．

例 4.4. ラムダ項 $\lambda xy.yxx$ の構文木に従って，以下のように型推論できる．

$$\dfrac{\dfrac{\dfrac{\dfrac{[y\colon A \to A \to B]^2 \qquad [x\colon A]^1}{yx\colon A \to B}\,(\to\mathtt{Elim}) \qquad [x\colon A]^1}{yxx\colon B}\,(\to\mathtt{Elim})}{\lambda y.yxx\colon (A \to A \to B) \to B}\,(\to\mathtt{Intro}^2)}{\lambda x.\lambda y.yxx\colon A \to (A \to A \to B) \to B}\,(\to\mathtt{Intro}^1)$$

したがって，ラムダ項 $\lambda xy.yxx$ の型は $A \to (A \to A \to B) \to B$ であると推論できたが，ここからラムダ項の部分を消去すると，命題 $A \to (A \to A \to B) \to B$ の証明図を得る．

$$\dfrac{\dfrac{\dfrac{\dfrac{[A \to A \to B]^2 \qquad [A]^1}{A \to B}\,(\to\mathtt{Elim}) \qquad [A]^1}{B}\,(\to\mathtt{Elim})}{(A \to A \to B) \to B}\,(\to\mathtt{Intro}^2)}{A \to (A \to A \to B) \to B}\,(\to\mathtt{Intro}^1)$$

■自然演繹から型推論へ: 逆に，直観主義命題論理の自然演繹における証明が与えられていたとする．たとえば，まず命題 $(B \to C) \to (A \to B) \to A \to C$ の証明を行ってみよう．

$$\dfrac{\dfrac{\dfrac{\dfrac{[B \to C]^1 \qquad \dfrac{[A \to B]^2 \qquad [A]^3}{B}\,(\to\mathtt{Elim})}{C}\,(\to\mathtt{Elim})}{A \to C}\,(\to\mathtt{Intro}^3)}{(A \to B) \to A \to C}\,(\to\mathtt{Intro}^2)}{(B \to C) \to (A \to B) \to A \to C}\,(\to\mathtt{Intro}^1)$$

この証明図の各葉，すなわち仮定部分に変数を割り当てる．たとえば，仮定 1 の $B \to C$ には変数 x, 仮定 2 の $A \to B$ には変数 y, 仮定 3 の A には変数 z を割り当てよう．そうすると，ラムダ項の構成規則にしたがって，以下のように自動的にラムダ項が組み上がっていく．

$$\dfrac{\dfrac{\dfrac{\dfrac{[x\colon B \to C]^1 \qquad \dfrac{[y\colon A \to B]^2 \qquad [z\colon A]^3}{yz\colon B}\,(\to\mathtt{Elim})}{x(yz)\colon C}\,(\to\mathtt{Elim})}{\lambda z.x(yz)\colon A \to C}\,(\to\mathtt{Intro}^3)}{\lambda yz.x(yz)\colon (A \to B) \to A \to C}\,(\to\mathtt{Intro}^2)}{\lambda xyz.x(yz)\colon (B \to C) \to (A \to B) \to A \to C}\,(\to\mathtt{Intro}^1)$$

つまり，除去規則の部分は関数適用を行い，導入規則の部分はそのタグに対応する変数に対するラムダ抽象を行う．このようにして，証明図からラムダ項を組み上げることができる．今回の場合

は，命題 $(B \to C) \to (A \to B) \to A \to C$ の証明図からラムダ項 $\lambda xyz.x(yz)$ を得た．さて，観点を変えると，上記の証明図は，ラムダ項 $\lambda xyz.x(yz)$ の型が $(B \to C) \to (A \to B) \to A \to C$ であることを示す型推論図であると見ることもできる．つまり，証明図だと思っていたものが，今度は型推論図になっている．

ともあれ，証明図からラムダ項を常に得られるというのは重要なポイントである．このラムダ項を証明図の名前であると考えよう．証明図は二次元的で巨大になるが，ラムダ項はコンパクトである．たとえば上記の命題 $(B \to C) \to (A \to B) \to A \to C$ の証明図は中々の紙幅を取るが，ラムダ項で表せば $\lambda xyz.x(yz)$ である．つまり，命題 $(B \to C) \to (A \to B) \to A \to C$ の証明は $\lambda xyz.x(yz)$ であると一言で述べることができる．

■否定: 「ならば」に関する推論だけではあまり面白くないので，もう少し意味のある命題に関する推論を行ってみよう．ただし，「かつ」や「または」に関する推論だと少し難しくなるので，ここでは「否定」を取り扱う．命題 A について

「否定 $\neg A$ が成り立つ」とは「A を仮定すると矛盾する」

ということである．記号的には，否定 $\neg A$ は $A \to \bot$ の略記として定義される．ここで，$\bot$ は矛盾を意味する．このように考えると，「否定」に関する推論は「ならば」に関する推論を流用できるので，新たな推論規則を説明する必要はない．つまり，「ならば」に関する規則の特殊な場合として「否定」に関する推論を得られる．

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ (\to\texttt{Intro}) \qquad\qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ (\to\texttt{Elim})$$

あるいは，縦書き記法を用いると，以下のように書き直せる．

$$\frac{\begin{array}{c}[A]\\ \vdots\\ \bot\end{array}}{\neg A}\ (\to\texttt{Intro}) \qquad\qquad \frac{\neg A \qquad A}{\bot}\ (\to\texttt{Elim})$$

ラムダ項まで明示的に書くと，否定に関する推論とは，以下のようなものである．

$$\frac{\begin{array}{c}[x\colon A]\\ \vdots\\ M\colon \bot\end{array}}{\lambda x.M\colon \neg A}\ (\to\texttt{Intro}) \qquad\qquad \frac{M\colon \neg A \qquad N\colon A}{MN\colon \bot}\ (\to\texttt{Elim})$$

例 4.5. まずは，二重否定の付加 $A \to \neg\neg A$ の証明は以下によってなされる．

$$\dfrac{\dfrac{\dfrac{[\neg A]^2 \qquad [A]^1}{\bot}\ (\to\ \texttt{Elim})}{\neg\neg A}\ (\to\ \texttt{Intro})^2}{A \to \neg\neg A}\ (\to\ \texttt{Intro})^1 \qquad\qquad \dfrac{\dfrac{\dfrac{[x\colon \neg A]^2 \qquad [y\colon A]^1}{xy\colon \bot}\ (\to\ \texttt{Elim})}{\lambda x.xy\colon \neg\neg A}\ (\to\ \texttt{Intro})^2}{\lambda y.\lambda x.xy\colon A \to \neg\neg A}\ (\to\ \texttt{Intro})^1$$

以上より，二重否定の付加 $A \to \neg\neg A$ の証明は $\lambda y.\lambda x.xy$ である．実は，上の証明は $\bot$ の性質を一切用いていないので，$\bot$ を任意の B に置換しても同じ証明が成立する．つまり，$\lambda y.\lambda x.xy$ は $A \to ((A \to B) \to B)$ の証明にもなっている．これはラムダ項 $\lambda y.\lambda x.xy$ の型を $A \to ((A \to B) \to B)$ であると推論しているということでもある．

例 4.6. 三重否定の除去 $\neg\neg\neg A \to \neg A$ は以下によって証明される．

$$\dfrac{\dfrac{[\neg\neg\neg A]^1 \qquad \dfrac{\dfrac{[\neg A]^3 \qquad [A]^2}{\bot}\,(\to\mathtt{Elim})}{\neg\neg A}\,(\to\mathtt{Intro})^3}{\bot}\,(\to\mathtt{Elim})}{\dfrac{\neg A}{\neg\neg\neg A \to \neg A}\,(\to\mathtt{Intro})^1}\,(\to\mathtt{Intro})^2$$

$$\dfrac{\dfrac{[x\colon \neg\neg\neg A]^1 \qquad \dfrac{\dfrac{[z\colon \neg A]^3 \qquad [y\colon A]^2}{zy\colon \bot}\,(\to\mathtt{Elim})}{\lambda z.zy\colon \neg\neg A}\,(\to\mathtt{Intro})^3}{x(\lambda z.zy)\colon \bot}\,(\to\mathtt{Elim})}{\dfrac{\lambda y.x(\lambda z.zy)\colon \neg A}{\lambda x.\lambda y.x(\lambda z.zy)\colon \neg\neg\neg A \to \neg A}\,(\to\mathtt{Intro})^1}\,(\to\mathtt{Intro})^2$$

以上より，三重否定の除去 $\neg\neg\neg A \to \neg A$ の証明は $\lambda x.\lambda y.x(\lambda z.zy)$ である．先ほどのように，この証明は $\bot$ の性質を一切用いていないので，$\bot$ は別の命題変数 B に置き換えても成立する．つまり，$\lambda x.\lambda y.x(\lambda z.zy)$ は $(((A \to B) \to B) \to B) \to A \to B$ の証明でもある．これはラムダ項 $\lambda x.\lambda y.x(\lambda z.zy)$ の型を $(((A \to B) \to B) \to B) \to A \to B$ であると推論しているということでもある．

例 4.7. 対偶の片向き $(A \to B) \to (\neg B \to \neg A)$ を示してみよう．

$$\dfrac{\dfrac{[\neg B]^2 \qquad \dfrac{[A \to B]^1 \qquad [A]^3}{B}\,(\to\mathtt{E})}{\bot}\,(\to\mathtt{E})}{\dfrac{\neg A}{\neg B \to \neg A}\,(\to\mathtt{I})^2}\,(\to\mathtt{I})^3}{(A \to B) \to (\neg B \to \neg A)}\,(\to\mathtt{I})^1$$

$$\dfrac{\dfrac{[y\colon \neg B]^2 \qquad \dfrac{[x\colon A \to B]^1 \qquad [z\colon A]^3}{xz\colon B}\,(\to\mathtt{E})}{y(xz)\colon \bot}\,(\to\mathtt{E})}{\dfrac{\lambda z.y(xz)\colon \neg A}{\lambda y.\lambda z.y(xz)\colon \neg B \to \neg A}\,(\to\mathtt{I})^2}\,(\to\mathtt{I})^3}{\lambda x.\lambda y.\lambda z.y(xz)\colon (A \to B) \to (\neg B \to \neg A)}\,(\to\mathtt{I})^1$$

以上より，三重否定の除去 $(A \to B) \to (\neg B \to \neg A)$ の証明は $\lambda x.\lambda y.\lambda z.y(xz)$ である．先ほどのように，この証明は $\bot$ の性質を一切用いていないので，$\bot$ は別の命題変数 C に置き換えても成立する．つまり，$\lambda x.\lambda y.\lambda z.y(xz)$ は $(A \to B) \to ((B \to C) \to A \to C)$ の証明でもある．これはラムダ項 $\lambda x.\lambda y.\lambda z.y(xz)$ の型を $(A \to B) \to ((B \to C) \to A \to C)$ であると推論しているということでもある．

例 4.8. 次は，$(\neg\neg(A \to B)) \to (\neg\neg A \to \neg\neg B)$ の証明をしてみよう．

$$\dfrac{\dfrac{\dfrac{[\neg\neg(A \to B)]^{1} \qquad \dfrac{\dfrac{\dfrac{[\neg\neg A]^{2} \qquad \dfrac{\dfrac{\dfrac{[\neg B]^{3} \qquad \dfrac{[A \to B]^{4} \qquad [A]^{5}}{B}\,(\to\text{Elim})}{\bot}\,(\to\text{Elim})}{\neg A}\,(\to\text{Intro})^{5}}{\bot}\,(\to\text{Elim})}{\neg(A \to B)}\,(\to\text{Intro})^{4}}{\bot}\,(\to\text{Elim})}{\neg\neg B}\,(\to\text{Intro})^{3}}{(\neg\neg A \to \neg\neg B)}\,(\to\text{Intro})^{2}}{(\neg\neg(A \to B)) \to (\neg\neg A \to \neg\neg B)}\,(\to\text{Intro})^{1}$$

証明に対応するラムダ項だけ明示すると，以下のようになっている．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[v]^{1} \qquad \dfrac{\dfrac{\dfrac{[w]^{2} \qquad \dfrac{\dfrac{\dfrac{[z]^{3} \qquad \dfrac{[y]^{4} \qquad [x]^{5}}{yx}\,(\to\text{Elim})}{z(yx)}\,(\to\text{Elim})}{\lambda x.z(yx)}\,(\to\text{Intro})^{5}}{w(\lambda x.z(yx))}\,(\to\text{Elim})}{\lambda y.w(\lambda x.z(yx))}\,(\to\text{Intro})^{4}}{v(\lambda y.w(\lambda x.z(yx)))}\,(\to\text{Elim})}{\lambda z.v(\lambda y.w(\lambda x.z(yx)))}\,(\to\text{Intro})^{3}}{\lambda w.\lambda z.v(\lambda y.w(\lambda x.z(yx)))}\,(\to\text{Intro})^{2}}{\lambda v.\lambda w.\lambda z.v(\lambda y.w(\lambda x.z(yx)))}\,(\to\text{Intro})^{1}$$

4.3. カリー-ハワード対応と BHK 解釈

■カリー-ハワード対応: 第 4.2 節の内容をまとめると，以下のような対応があることがなんとなく理解できたと思われる．

ラムダ項 $\approx$ 証明　　型 $\approx$ 命題

これがいわゆる**カリー-ハワード対応** (*Curry-Howard correspondence*) と呼ばれるもの（の最も基本的な形式）である．カリー-ハワード対応により，「型は命題である」「ラムダ項は証明である」などと言うことができる．この考え方は，「**型** = **命題** (*propositions-as-types*)」や「**プログラム** = **証明** (*proofs-as-programs*)」というスローガンで表現されることも多い．

実際に，論理結合子を「ならば」のみに制限した場合のカリー-ハワード対応の形式的な証明を与えよう．以下，論理式の列 Γ について，$\Gamma' = \{x_\varphi : \varphi \mid \varphi \in \Gamma\}$ とする．

定理 4.9 (カリー-ハワード対応)**.**

1. $\Gamma \vdash M : \varphi$ ならば，$|\Gamma| \vdash \varphi$ である．

2. もし $\Gamma \vdash \varphi$ ならば，ある λ-項 M が存在して，$\Gamma' \vdash M : \varphi$ となる．

Proof. (1) については，上に述べたように，各型付け規則からラムダ項の部分を削除したものが推論規則に一致するから，型付け推論図からラムダ項を削除すれば，証明図になる．

(2) については，(→ `Intro`) において新しい変数記号を導入する必要があるかもしれないので，その場合だけ注意すればよい．一応，証明の方針を説明するために，(`Ax`), (→`Elim`), (→`Intro`) について証明を与える．証明は証明図の複雑さに関する帰納法による．

1. (`Ax`) $\Gamma, \varphi \vdash \varphi$ である場合を考える．もし $\varphi \in \Gamma$ ならば，$(x_\varphi : \varphi) \in \Gamma'$ なので，$\Gamma' \vdash x_\varphi : \varphi$ が従う．もし $\varphi \notin \Gamma$ ならば，新しい変数 x_φ を用意して，$\Gamma', x_\varphi : \varphi \vdash x_\varphi : \varphi$ が導かれる．
2. (→`Elim`) 次の推論によって導かれるとする．

$$\frac{\Gamma \vdash \psi \to \varphi \qquad \Gamma \vdash \psi}{\Gamma \vdash \varphi}$$

   このときは，上式に対して帰納的仮定を用いると，$\Gamma' \vdash M : \psi \to \varphi$ および $\Gamma' \vdash N : \psi$ となる項 M, N が存在する．したがって，型付け規則の定義より，$\Gamma' \vdash MN : \varphi$ が導かれる．
3. (→`Intro`) 次の推論によって導かれるとする．

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}$$

   $\varphi \in \Gamma$ の場合には，上式に対して帰納的仮定を用いると，$\Gamma' \vdash M : \psi$ となる項 M が存在する．既に $(x_\varphi : \varphi) \in \Gamma'$ であるが，x_φ とは別に新しい変数 x を用意しよう．弱化より，$\Gamma', x : \varphi \vdash M : \psi$ は成立する．したがって，型付け規則の定義より，$\Gamma' \vdash \lambda x.M : \varphi \to \psi$ が成り立つ．
   $\varphi \notin \Gamma$ の場合には，上式に対して帰納的仮定を用いると，$\Gamma', x_\varphi : \varphi \vdash M : \psi$ となる項 M が存在する．このとき，型付け規則の定義より，$\Gamma' \vdash \lambda x_\varphi.M : \varphi \to \psi$ が成り立つ．

以上より，定理は示された． □

■ **BHK 解釈:** カリー-ハワード対応には，その前身となる幾つかのアイデアがある．根幹となるアイデアは，論理式の証明をラムダ項，つまり，論理式の正しさを計算によって保証するという行為である．歴史的には，

- 論理式の正しさをその保証付きで考える，というアイデアの源流は，**ブラウワー-ハイティング-コルモゴロフ解釈** (*Brower-Heyting-Kolmogorov interpretation*) あるいは略して **BHK 解釈**と呼ばれるものである．
  - たとえば，1920 年代のコルモゴロフは，「命題」を「問題」(propositions-as-problems)，「証明」を「問題を解くプロセス」と解釈することを提案した．

- 論理式の正しさを計算によって保証する，というアイデアは，1945 年のクリーネによる**実現可能性** (*realizability*) が初出である．ラムダ計算の文脈では，証明の保証として，型なしラムダ項を用いるものが，クリーネ実現可能性である．
- 型付き項によって証明の保証を与える形に修正したものが，1959 年のクライゼルによる**修正実現可能性** (*modified realizability*) であり，これが証明に型付きラムダ項を割り当てるというアイデアの源流のひとつである．

ここで正しさの保証というものは，たとえば $A \vee B$ が真ならば，どちらが正しいかの証拠を持ってくることを要求するものである．たとえば，RH を現代数学における最大の未解決問題のひとつであるリーマン予想を表すものとする．われわれは普段，古典論理を用いているから，特に排中律より，リーマン予想は正しいか正しくないかのいずれかである，つまり $\mathsf{RH} \vee \neg\mathsf{RH}$ である．しかし，RH と $\neg$RH のどちらが正しいかを断言することは，リーマン予想を解決することと同等である．また，もしリーマン予想が ZFC 集合論から独立であったとしたら，状況はますますややこしくなりそうである．つまるところ，$A \vee B$ の正しさの証拠を見つけてくるというのは，単に $A \vee B$ が正しいと主張することより難しそうである．

まず，このアイデアの原点である，BHK 解釈について説明しよう．BHK 解釈のアイデアは，以下のように帰納的に説明できる．

1. A が原始論理式ならば，A の証拠とは A の証明である．
2. $A \wedge B$ の証拠とは，A の証拠と B の証拠の対である．
3. $A \vee B$ の証拠とは，どちらの式が正しいかの言及 i と，正しい側の式の証拠 p の対 $\langle i, p\rangle$ である．より正確には，$i = 0$ ならば p は A の証拠であり，$i = 1$ ならば p は B の証拠である．
4. $A \to B$ の証拠とは，次を満たす関数 f である．もし x が A の証拠ならば，$f(x)$ は B の証拠である．
5. $\exists x A(x)$ の証拠とは，対 $\langle a, p\rangle$ である．ここで，p は $A(a)$ の証拠である．
6. $\forall x A(x)$ の証拠とは，次を満たす関数 f である．量化領域内の任意の x について，$f(x)$ は $A(x)$ の証拠である．

項目 (4) を考えれば，含意 $A \to B$ と関数 $A \to B$ の関係は明らかであろう．具体的には，$[\![A]\!]$ を論理式 A の正しさの証拠全体の集合とすれば，BHK 解釈は，含意 $A \to B$ と関数 $f\colon [\![A]\!] \to [\![B]\!]$ を対応させている．実際には，上記項目 (4) 内の関数のクラスをある程度制限するのが標準的である．たとえば，クリーネ実現可能性においては，部分計算可能関数 f（つまり，型なしラムダ項）を考え，クライゼルの修正実現可能性においては，型付きラムダ項を考える．このように，証明のラムダ項とは，BHK 解釈に従って，証明の各ステップに命題の正しさの証拠を付随させていくものだと考えることができる．

4.4. 命題結合子に対するカリー-ハワード対応

完全なカリー-ハワード対応を与えるためには，「ならば」以外の論理結合子，つまり「かつ」や「または」がラムダ計算において如何なる意味を持つかについて分析することにしよう．

■かつ: 前に見たように，含意 $\to$ の導入と除去の規則はそれぞれラムダ抽象と適用に対応しているが，この対応は BHK 解釈から予測可能である．BHK 解釈に従って，$A \wedge B$ の証拠とは何であるかと考えると，A と B の証拠の対である．したがって，$A \wedge B$ の証拠を知っていれば，その射影を取ることによって，A の証拠以上の議論に基づき，BHK 解釈のアイデアに従って $\wedge$ の導入規則および除去規則に証拠を付加してみよう．

$$\frac{\begin{array}{cc}\vdots & \vdots \\ M\colon A_0 & N\colon A_1\end{array}}{\langle M, N\rangle\colon A_0 \wedge A_1}\ (\wedge\texttt{Intro}) \qquad \frac{\begin{array}{c}\vdots \\ M\colon A_0 \wedge A_1\end{array}}{\pi_i M\colon A_i}\ (\wedge\texttt{Elim})$$

証拠の集合上の関数として考えると，導入規則は対関数 $\langle\cdot,\cdot\rangle\colon [\![A_0]\!] \times [\![A_1]\!] \to [\![A_0 \wedge A_1]\!]$ であり，除去規則は射影 $\pi_i\colon [\![A_0]\!] \times [\![A_1]\!] \to [\![A_i]\!]$．このアイデアに基づいて考えると，「かつ」に対応するものは集合の直積である．

命題 2.9 で見たように，型なしラムダ計算において既にペアリングと射影の概念は実装した．つまり，項 $\texttt{pair}, \pi_0, \pi_1$ が存在して，

$$\pi_0(\texttt{pair}\, ab) = a, \qquad \pi_0(\texttt{pair}\, ab) = b$$

となる．項 $\texttt{pair}\, ab$ のことは $\langle a, b\rangle$ と略記したことを思い出そう．このペアリングと射影に対して型を与えよう．ここからは，$\texttt{pair}$ や π_0, π_1 の具体的な実装方法に依存しない議論を行いたい．このため，抽象的に，次のように，項の構成で $\texttt{pair}, \pi_0, \pi_1$ の利用を認めることにしよう．

$$M ::= x \mid MM \mid \lambda x.M \mid \langle M, M\rangle \mid \pi_0 M \mid \pi_1 M$$

ここで，通常に β-簡約に加えて，次のような簡約も可能であるとする．

$$\pi_0\langle M, N\rangle \to M, \qquad \pi_1\langle M, N\rangle \to M$$

これはまだ型なし項であるが，この型付けについて考えよう．通常の数学では，元 $x \in X$ と $y \in Y$ の対 $\langle x, y\rangle$ は直積 $X \times Y$ に属すのであった．したがって，M が型 σ で N が型 τ であれば，$\langle M, N\rangle$ の型は $\sigma \times \tau$ であると考えるのが妥当であろう．このように，ここからは新たな種類の型 $\sigma \times \tau$ を構成するための型コンストラクタ $\times$ を取り扱う．つまり，現在，認めている型とは，以下のようなものである．

$$\sigma ::= \alpha \mid \sigma \to \sigma \mid \sigma \times \sigma$$

新たな項に関する型付け規則は，以下によって与えられる．

$$\frac{\Gamma \vdash M\colon \sigma \qquad \Gamma \vdash N\colon \tau}{\Gamma \vdash \langle M, N\rangle\colon \sigma \times \tau}\ (\times\texttt{Intro}) \qquad \frac{\Gamma \vdash L\colon \sigma_0 \times \sigma_1}{\Gamma \vdash \pi_i L\colon \sigma_i}\ (\times\texttt{Elim})$$

この型付け規則からラムダ項を取り除いて，$\times$ を $\wedge$ に置き換えれば，自然演繹における $\wedge$ の推論規則である．ここからは，$\sigma \times \tau$ のことを $\sigma \wedge \tau$ とも書くことにする．このとき，もし $\Gamma \vdash M : \sigma$ ならば，型付けの推論図からラムダ項を取り除くと，$|\Gamma| \vdash \sigma$ の証明図になり，M の構文木は $|\Gamma| \vdash \sigma$ の証明の骨格を与える．逆に，自然演繹 IPC($\to, \wedge$) における $\Gamma \vdash \sigma$ の証明から，先ほどと同様にして，$\Gamma' \vdash M : \sigma$ となるラムダ項 M を得られる．

例 4.10. 矛盾律 $((A \to B) \wedge (A \to \neg B)) \to \neg A$ を証明しよう．

$$\dfrac{\dfrac{\dfrac{\dfrac{[(A \to B) \wedge (A \to \neg B)]^1}{A \to \neg B}\,(\wedge\mathtt{E}) \qquad [A]^2}{\neg B}\,(\to\mathtt{E}) \qquad \dfrac{\dfrac{[(A \to B) \wedge (A \to \neg B)]^1}{A \to B}\,(\wedge\mathtt{E}) \qquad [A]^2}{B}\,(\to\mathtt{E})}{\bot}\,(\to\mathtt{E})}{\neg A}\,(\to\mathtt{I})^2}{((A \to B) \wedge (A \to \neg B)) \to \neg A}\,(\to\mathtt{I})^1$$

この証明に対応するラムダ項の部分だけ明示すると，以下のようになっている．

$$\dfrac{\dfrac{\dfrac{\dfrac{[x]^1}{\pi_1 x}\,(\wedge\mathtt{Elim}) \qquad [y]^2}{\pi_1 xy}\,(\to\mathtt{Elim}) \qquad \dfrac{\dfrac{[x]^1}{\pi_0 x}\,(\wedge\mathtt{Elim}) \qquad [y]^2}{\pi_0 xy}\,(\to\mathtt{Elim})}{\pi_1 xy(\pi_0 xy)}\,(\to\mathtt{Elim})}{\lambda y.\pi_1 xy(\pi_0 xy)}\,(\to\mathtt{Intro})^2}{\lambda x.\lambda y.\pi_1 xy(\pi_0 xy)}\,(\to\mathtt{Intro})^1$$

以上より，矛盾律 $((A \to B) \wedge (A \to \neg B)) \to \neg A$ の証明は $\lambda x.\lambda y.\pi_1 xy(\pi_0 xy)$ である．この証明は $\bot$ の性質を一切用いていないので，$\bot$ は別の命題変数 C に置き換えても成立する．特に，これはラムダ項 $\lambda x.\lambda y.\pi_1 xy(\pi_0 xy)$ の型 $((A \to B) \times (A \to B \to C)) \to A \to C$ の推論にもなっている．

例 4.11. $(\neg\neg(A \wedge B)) \to (\neg\neg A \wedge \neg\neg B)$ を証明しよう．

$$\dfrac{\dfrac{\dfrac{[\neg\neg(A \wedge B)]^1 \qquad \dfrac{\dfrac{[\neg A]^2 \qquad \dfrac{[A \wedge B]^4}{A}\,(\wedge\mathtt{Elim})}{\bot}\,(\to\mathtt{Elim})}{\neg(A \wedge B)}\,(\to\mathtt{Intro})^4}{\bot}\,(\to\mathtt{Elim})}{\neg\neg A}\,(\to\mathtt{Intro})^2 \qquad \dfrac{\dfrac{\dfrac{[\neg\neg(A \wedge B)]^1 \qquad \dfrac{\dfrac{[\neg B]^3 \qquad \dfrac{[A \wedge B]^5}{B}\,(\wedge\mathtt{Elim})}{\bot}\,(\to\mathtt{Elim})}{\neg(A \wedge B)}\,(\to\mathtt{Intro})^5}{\bot}\,(\to\mathtt{Elim})}{\neg\neg B}\,(\to\mathtt{Intro})^3}{\neg\neg A \wedge \neg\neg B}\,(\wedge\mathtt{Intro})}{(\neg\neg(A \wedge B)) \to (\neg\neg A \wedge \neg\neg B)}\,(\to\mathtt{Intro})^1$$

この証明に対応するラムダ項の部分だけ明示すると，以下のようになっている．

$$\dfrac{\dfrac{\dfrac{\dfrac{[z]^1 \qquad \dfrac{\dfrac{[x]^2 \qquad \dfrac{[u]^4}{\pi_0 u}\,(\wedge\mathtt{Elim})}{x(\pi_0 u)}\,(\to\mathtt{Elim})}{\lambda u.x(\pi_0 u)}\,(\to\mathtt{Intro})^4}{z(\lambda u.x(\pi_0 u))}\,(\to\mathtt{Elim})}{\lambda x.z(\lambda u.x(\pi_0 u))}\,(\to\mathtt{Intro})^2 \qquad \dfrac{\dfrac{\dfrac{[z]^1 \qquad \dfrac{\dfrac{[y]^3 \qquad \dfrac{[v]^5}{\pi_1 v}\,(\wedge\mathtt{Elim})}{y(\pi_1 v)}\,(\to\mathtt{Elim})}{\lambda v.y(\pi_1 v)}\,(\to\mathtt{Intro})^5}{z(\lambda v.y(\pi_1 v))}\,(\to\mathtt{Elim})}{\lambda y.z(\lambda v.y(\pi_1 v))}\,(\to\mathtt{Intro})^3}{\langle \lambda x.z(\lambda u.x(\pi_0 u)), \lambda y.z(\lambda v.y(\pi_1 v))\rangle}\,(\wedge\mathtt{Intro})}{\lambda z.\langle \lambda x.z(\lambda u.x(\pi_0 u)), \lambda y.z(\lambda v.y(\pi_1 v))\rangle}\,(\to\mathtt{Intro})^1$$

以上より，$(\neg\neg(A \wedge B)) \to (\neg\neg A \wedge \neg\neg B)$ の証明は $\lambda z.\langle \lambda x.z(\lambda u.x(\pi_0 u)), \lambda y.z(\lambda v.y(\pi_1 v))\rangle$ である．この証明は $\bot$ の性質を一切用いていないので，$\bot$ は別の命題変数 C に置き換えても成立する．特に，これは上記のラムダ項の型 $(((A \wedge B) \to C) \to C) \to (((A \to C) \to C) \wedge ((B \to C) \to C))$ の推論にもなっている．

■型付き条件分岐: 計算という概念の最も基本的な構成要素のひとつは条件分岐である．命題 2.8 および例 2.15 より，型なしラムダ計算において，条件分岐を実装できたのであった．我々が実装した条件分岐の第一パターンは，与えられた論理式の真偽に基づく場合分けである．

$$\texttt{if } \varphi \texttt{ istrue then } x \texttt{ else } y \quad \longrightarrow \quad \begin{cases} x & \text{if } \varphi \text{ is } \texttt{true} \\ y & \text{if } \varphi \text{ is } \texttt{false} \end{cases}$$

計算プロセスとしては，φ を入力として x または y を返すものである．もし，x, y の型が共に σ なのであれば，この計算プロセスの型は $\texttt{bool} \to \sigma$ と考えるのが妥当であろう．条件分岐の第二パターンは，自然数の零判定に基づく場合分けである．

$$\texttt{if } \underline{n} \texttt{ iszero then } c \texttt{ else } f\underline{n} \quad \longrightarrow \quad \begin{cases} c & \text{if } n = 0 \\ f\underline{n} & \text{if } n > 0 \end{cases}$$

これもまた計算プロセスとしては，n を入力として c または fn を返すものである．もし，c, fn の型が共に σ なのであれば，この計算プロセスの型は $\texttt{nat} \to \sigma$ と考えるのが妥当であろう．

さて，ここで注目すべき点として，第一パターンは，$\texttt{true} \mapsto x$ および $\texttt{false} \mapsto y$ という 2 つの関数（定数）の組合せであるという点である．それぞれの関数の型は，$\{\texttt{true}\} \to X$ および $\{\texttt{false}\} \to X$ である．

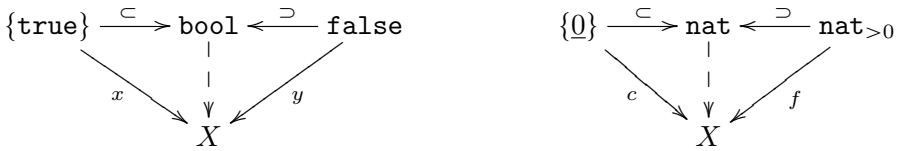
$$\{\texttt{true}\} \to X \quad + \quad \{\texttt{false}\} \to X \qquad \rightsquigarrow \qquad \texttt{bool} \to X$$

ブール値を真偽の 2 値集合だとみなすと，$\texttt{bool} = \{\texttt{true}\} \cup \{\texttt{false}\}$ であるから，これは始域の非交叉和 (disjoint union) を取る操作であると考えることもできる..

第二パターンについては，$\underline{0} \mapsto c$ および f の組合せである．それぞれの関数の型は，$\{\underline{0}\} \to X$ および $\texttt{nat}_{>0} \to X$ である．

$$\underline{0} \to X \quad + \quad \texttt{nat}_{>0} \to X \qquad \rightsquigarrow \qquad \texttt{nat} \to X$$

上記と同様に，型を集合だとみなすならば，$\texttt{nat} = \underline{0} \cup \texttt{nat}_{>0}$ であるから，これも始域の非交叉和を取っている．上記の構成をもう少し図式的に書くならば，以下のようなものになるだろう．



以上が，条件分岐の型付けのアイデアであり，鍵となる概念は，非交叉和である．ラムダ計算において非交叉和に対応する概念は，直和型 $A + B$ と呼ばれるものである．これは型 A のデータと

型 B のデータを両方格納できる型のようなもので，たとえば，整数の型 int と実数の型 real を両方格納できる直和型 int + real などを考えることができる．ここで，型 $A + B$ 項 L には，実際に L がどちらの型の項なのかを表すタグが付いているものと考える．つまり，型 $A + B$ 項は正確には $\langle i, L\rangle$ の形であり，$i = 0$ ならば L は型 A 項であり，さもなくば L は型 B 項であることを表すものである．

通常の数学においては，集合の非交差和を作るために，しばしば $X \sqcup Y = (\{0\} \times X) \cup (\{1\} \times Y)$ を用いる．このとき，2 種類の自明な包含写像 $\mathtt{in}_0 \colon X \to \{0\} \times X$ と $\mathtt{in}_1 \colon Y \to \{1\} \times Y$ がある．ラムダ項の文脈では，たとえば，

$$\mathtt{in}_0\, L := \langle 0, L\rangle, \qquad \mathtt{in}_1\, L := \langle 1, L\rangle$$

として定義することができる．

さて，ここまでで我々が見た条件分岐は，入力の型が bool または nat であったが，ここからはより一般的な条件分岐を行いたい．条件分岐のアイデアは 2 つの関数の組合せである．つまり，条件分岐によって，型 $\sigma \to \rho$ 項 M あるいは型 $\tau \to \rho$ 項 N のいずれかの計算を行うというタイプのものである．ここまでのアイデアを応用すれば，その結果となる計算の型は $\sigma + \tau \to \rho$ となるであろうと予期できる．

もう少し形式的には，型付き条件分岐は，型 $\sigma \to \rho$ 項 $\lambda x.M$ と型 $\tau \to \rho$ 項 $\lambda y.N$ から構成される．型 $\sigma + \tau$ の項 $\langle i, L\rangle$ が入力されたとき，$i = 0$ ならば L は型 σ なので，これを $\lambda x.M$ に入力でき，さもなくば L は型 τ なので，これを $\lambda y.N$ に入力できる．つまり，項 $\langle i, L\rangle, M, N$ と変数 x, y が与えられているとき，

$$\mathtt{Cases}(\langle i, L\rangle; x.M; y.N) := \mathtt{if}\ i\ \mathtt{iszero}\ \mathtt{then}\ M[L/x]\ \mathtt{else}\ N[L/y]$$
$$\longrightarrow \begin{cases} M[\frac{L}{x}] & \text{if } i = 0 \\ N[\frac{L}{y}] & \text{otherwise} \end{cases}$$

を扱う．と，定義より，次の簡約が成り立つことは容易に分かる．

$$\mathtt{Cases}(\mathtt{in}_0\, L; x.M; y.N) \longrightarrow M\left[\tfrac{L}{x}\right]$$
$$\mathtt{Cases}(\mathtt{in}_1\, L; x.M; y.N) \longrightarrow N\left[\tfrac{L}{y}\right]$$

若干，人工的に感じるかもしれないが，とりあえずこれが必要なものである．つまり，項の構成において，$\mathtt{in}_0, \mathtt{in}_1, \mathtt{Cases}$ を利用可能にしよう．

$$M ::= x \mid MM \mid \lambda x.M \mid \langle M, M\rangle \mid \pi_0 M \mid \pi_1 M \mid \mathtt{in}_0\, M \mid \mathtt{in}_1\, M \mid \mathtt{Cases}(M; x.M; x.M)$$

Cases と $\mathtt{in}_0, \mathtt{in}_1$ に関する簡約規則は，すぐ上で与えた 2 つの規則のみである．また，新たな種類の型 $\sigma + \tau$ を構成するための型コンストラクタ $+$ を取り扱う．つまり，現在，認めている型とは，以下のようなものである．

$$\sigma ::= \alpha \mid \sigma \to \sigma \mid \sigma \times \sigma \mid \sigma + \sigma$$

新たな項に対する型付け規則は，以下によって与えられる．

$$\frac{\Gamma \vdash M : \sigma_i}{\Gamma \vdash \mathtt{in}_i\, M : \sigma_0 + \sigma_1}\ (+\mathtt{Intro}) \qquad \frac{\Gamma \vdash L : \sigma + \tau \qquad \Gamma, x : \sigma \vdash M : \rho \qquad \Gamma, y : \tau \vdash N : \rho}{\Gamma \vdash \mathtt{Cases}(L; x.M; y.M) : \rho}\ (+\mathtt{Elim})$$

これによって，カリー-ハワード対応は「または」に対して拡張される．

■または: 各命題に対応付けるべき証拠の概念は，BHK 解釈に基づく．「または」の導入規則 ($\vee$Intro) は「または」の証明に関する規則であるから，BHK 解釈を考えれば，$A \vee B$ のどちらが正しいかの情報を証拠として付随させなければならない．「または」の除去規則 ($\vee$Elim) を実現する項については，一見では明らかではないので，一旦？とおいておくことにする．

$$\frac{\begin{array}{c}\vdots\\ M : A_k\end{array}}{\mathtt{in}_k M : A_0 \vee A_1}\ (\vee\mathtt{Intro}) \qquad \frac{\begin{array}{ccc} & [x : A] & [y : B] \\ & \vdots & \vdots \\ L : A \vee B & M : C & N : C \end{array}}{? : C}\ (\vee\mathtt{Elim})$$

($\vee$Elim) 中の？を求めよう．まず，$\vee$ の BHK 解釈によれば，$A \vee B$ の証拠 L には，どちらが正しいかの情報が付随しており，それは $\pi_0 L$ によって取り出せる．さらに，$\pi_1 L$ は，正しい側の正しさの証拠を与えるものであった．したがって，A または B の正しさの証拠は以下のプロセスによって抽出できる．

$$\frac{L : A \vee B}{\mathtt{if}\ \pi_0 L\ \mathtt{iszero\ then}\ \pi_1 L : A\ \mathtt{else}\ \pi_1 L : B}$$

また，($\vee$Elim) の上式に注目すれば，$\lambda x.M$ が $A \to C$ の証拠となっており，$\lambda y.N$ が $B \to C$ の証拠になっていることが分かる．したがって，$\pi_1 L$ が A の証拠であるとき，$M[\pi_1 L/x]$ は C の証拠であり，$\pi_1 L$ が B の証拠であるとき，$N[\pi_1 L/x]$ は C の証拠である．

$$\begin{array}{c}\pi_1 L : A \\ \vdots \\ M[\pi_1 L/x] : C\end{array} \qquad \begin{array}{c}\pi_1 L : B \\ \vdots \\ N[\pi_1 L/y] : C\end{array}$$

以上より，いずれにせよ C の証拠を得ることができる．つまり，$\vee$ の除去規則に対応する項は以下であることが導かれた．

$$\frac{\begin{array}{ccc} & [x : A] & [y : B] \\ & \vdots & \vdots \\ L : A \vee B & M : C & N : C \end{array}}{\mathtt{if}\ \pi_0 L\ \mathtt{iszero\ then}\ M[\pi_1 L/x]\ \mathtt{else}\ N[\pi_1 L/y] : C}\ (\vee\mathtt{Elim})$$

以後，以下の略記を用いる．

$$\mathtt{Cases}(L; x.M; y.N) \qu:= \quad \mathtt{if}\ \pi_0 L\ \mathtt{iszero\ then}\ M[\pi_1 L/x]\ \mathtt{else}\ N[\pi_1 L/y].$$

略記を用いると，以下のように記述できる．

$$\frac{\begin{array}{ccc} & [x : A] & [y : B] \\ & \vdots & \vdots \\ L : A \vee B & M : C & N : C \end{array}}{\mathtt{Cases}(L;\ x.M;\ y.N) : C}\ (\vee\mathtt{Elim})$$

これによって，「または」の推論規則と条件分岐の型付け規則の間の対応が与えられた．

例 4.12. 排中律の二重否定 $\neg\neg(A \vee \neg A)$ を証明しよう．

$$\dfrac{\dfrac{[\neg(A \vee \neg A)]^1 \qquad \dfrac{\dfrac{[A]^2}{A \vee \neg A}\,(\vee\mathtt{Intro})}{\bot}\,(\to\mathtt{Elim})}{\neg A}\,(\to\mathtt{Intro})^2 \qquad \dfrac{\neg A}{A \vee \neg A}\,(\vee\mathtt{Intro})}{\dfrac{\bot}{\neg\neg(A \vee \neg A)}\,(\to\mathtt{Intro})^1}\,(\to\mathtt{Elim})$$

証明図にラムダ項を割り当てると，以下を得る．

$$\dfrac{\dfrac{[x\colon \neg(A \vee \neg A)]^1 \qquad \dfrac{\dfrac{[a\colon A]^2}{\mathtt{in}_0 a\colon A \vee \neg A}\,(\vee\mathtt{Intro})}{x(\mathtt{in}_0 a)\colon \bot}\,(\to\mathtt{Elim})}{\lambda a.x(\mathtt{in}_0 a)\colon \neg A}\,(\to\mathtt{Intro})^2 \qquad \dfrac{\lambda a.x(\mathtt{in}_0 a)\colon \neg A}{\mathtt{in}_1(\lambda a.x(\mathtt{in}_0 a))\colon A \vee \neg A}\,(\vee\mathtt{Intro})}{\dfrac{x(\mathtt{in}_1(\lambda a.x(\mathtt{in}_0 a)))\colon \bot}{\lambda x.x(\mathtt{in}_1(\lambda a.x(\mathtt{in}_0 a)))\colon \neg\neg(A \vee \neg A)}\,(\to\mathtt{Intro})^1}\,(\to\mathtt{Elim})$$

以上より，排中律の二重否定 $\neg\neg(A \vee \neg A)$ の証明は $\lambda x.x(\mathtt{in}_1(\lambda a.x(\mathtt{in}_0 a)))$ である．この証明は $\bot$ の性質を一切用いていないので，$\bot$ は別の命題変数 B に置き換えても成立する．特に，これはラムダ項 $\lambda x.x(\mathtt{in}_1(\lambda a.x(\mathtt{in}_0 a)))$ の型 $((A \vee (A \to B)) \to B) \to B$ の推論にもなっている．

例 4.13. ド・モルガンの法則の片向き $(\neg A \vee \neg B) \to \neg(A \wedge B)$ を示そう．

$$\dfrac{\dfrac{[\neg A \vee \neg B]^1 \qquad \dfrac{[\neg A]^3 \qquad \dfrac{[A \wedge B]^2}{A}\,(\wedge\mathtt{Elim})}{\bot}\,(\to\mathtt{Elim}) \qquad \dfrac{[\neg B]^3 \qquad \dfrac{[A \wedge B]^2}{B}\,(\wedge\mathtt{Intro})}{\bot}\,(\to\mathtt{Elim})}{\bot}\,(\vee\mathtt{Elim})^3}{\dfrac{\bot}{\neg(A \wedge B)}\,(\to\mathtt{Intro})^2}\,(\to\mathtt{Intro})^1$$

この証明に対応するラムダ項の部分だけ明示すると，以下のようになっている．

$$\dfrac{\dfrac{\dfrac{[x]^1 \qquad \dfrac{[u]^3 \qquad \dfrac{[y]^2}{\pi_0 y}\,(\wedge\mathtt{Elim})}{u(\pi_0 y)}\,(\to\mathtt{Elim}) \qquad \dfrac{[v]^3 \qquad \dfrac{[y]^2}{\pi_1 y}\,(\wedge\mathtt{Intro})}{v(\pi_1 y)}\,(\to\mathtt{Elim})}{\mathtt{Cases}(x; u.u(\pi_0 y); v.v(\pi_1 y))}\,(\vee\mathtt{Elim})^3}{\lambda y.\mathtt{Cases}(x; u.u(\pi_0 y); v.v(\pi_1 y))}\,(\to\mathtt{Intro})^2}{\lambda x.\lambda y.\mathtt{Cases}(x; u.u(\pi_0 y); v.v(\pi_1 y))}\,(\to\mathtt{Intro})^1$$

以上より，$(\neg A \vee \neg B) \to \neg(A \wedge B)$ の証明は $\lambda x.\lambda y.\mathtt{Cases}(x; u.u(\pi_0 y); v.v(\pi_1 y))$ である．この証明は $\bot$ の性質を一切用いていないので，$\bot$ は別の命題変数 C に置き換えても成立する．特に，これはラムダ項 $\lambda x.\lambda y.\mathtt{Cases}(x; u.u(\pi_0 y); v.v(\pi_1 y))$ の型 $((A \to C) \vee (B \to C)) \to (A \wedge B) \to C$ の推論にもなっている．

演習問題 4.14. $(\neg A \wedge \neg B) \to \neg(A \vee B)$ の証明を表す項を書き下せ．

例 4.15. 分配法則の片向き $A \vee (B \wedge C) \to ((A \vee B) \wedge (A \vee C))$ を示そう．

$$\dfrac{\dfrac{[A \vee (B \wedge C)]^1 \qquad \dfrac{[A]^2}{A \vee B}\,(\vee\mathtt{I}) \qquad \dfrac{\dfrac{[B \wedge C]^2}{B}\,(\wedge\mathtt{E})}{A \vee B}\,(\vee\mathtt{I})}{A \vee B}\,(\vee\mathtt{E})^2 \qquad \dfrac{[A \vee (B \wedge C)]^1 \qquad \dfrac{[A]^3}{A \vee C}\,(\vee\mathtt{I}) \qquad \dfrac{\dfrac{[B \wedge C]^3}{C}\,(\wedge\mathtt{E})}{A \vee C}\,(\vee\mathtt{I})}{A \vee C}\,(\vee\mathtt{E})^3}{(A \vee B) \wedge (A \vee C)}\,(\wedge\mathtt{I})}{A \vee (B \wedge C) \to ((A \vee B) \wedge (A \vee C))}\,(\to\mathtt{I})^1$$

この証明に対応するラムダ項の部分だけ明示すると，以下のようになっている．

$$\dfrac{\dfrac{\dfrac{[z]^1 \qquad \dfrac{[x]^2}{\mathtt{in}_0 x}\,(\vee\mathrm{I}) \qquad \dfrac{\dfrac{[y]^2}{\pi_0 y}\,(\wedge\mathrm{E})}{\mathtt{in}_1 \pi_0 y}\,(\vee\mathrm{I})}{\mathtt{Cases}(z; x.\mathtt{in}_0 x; y.\mathtt{in}_1 \pi_0 y)}\,(\vee\mathrm{E})^2 \qquad \dfrac{[z]^1 \qquad \dfrac{[u]^3}{\mathtt{in}_0 u}\,(\vee\mathrm{I}) \qquad \dfrac{\dfrac{[v]^3}{\pi_0 v}\,(\wedge\mathrm{E})}{\mathtt{in}_1 \pi_0 v}\,(\vee\mathrm{I})}{\mathtt{Cases}(z; u.\mathtt{in}_0 u; v.\mathtt{in}_1 \pi_0 v)}\,(\vee\mathrm{E})^3}{\langle \mathtt{Cases}(z; x.\mathtt{in}_0 x; y.\mathtt{in}_1 \pi_0 y), \mathtt{Cases}(z; u.\mathtt{in}_0 u; v.\mathtt{in}_1 \pi_0 v)\rangle}\,(\wedge\mathrm{I})}{\lambda z.\langle \mathtt{Cases}(z; x.\mathtt{in}_0 x; y.\mathtt{in}_1 \pi_0 y), \mathtt{Cases}(z; u.\mathtt{in}_0 u; v.\mathtt{in}_1 \pi_0 v)\rangle}\,(\to\mathrm{I})^1$$

以上より，$A \vee (B \wedge C) \to ((A \vee B) \wedge (A \vee C))$ の証明は上記のラムダ項によって表される．同時に，これは上記ラムダ項の型 $A \vee (B \wedge C) \to ((A \vee B) \wedge (A \vee C))$ の推論にもなっている．

本稿では，命題論理に対するカリー-ハワード対応しか扱わないが，述語論理にも拡張可能である．

4.5. ラムダ計算の健全性定理と正規化定理

カリー-ハワード対応と，弱正規化を応用すると，直観主義命題論理の自然演繹の無矛盾性を示すことができる．もちろん，直観主義論理の自然演繹の無矛盾性は他の方法でも容易に証明することができるが，この方法の興味深い点は，ラムダ計算と弱正規化可能性と論理体系の無矛盾性に対応があることが明示される，という点であろう．

命題 4.16. 直観主義命題論理の自然演繹は無矛盾である．

Proof. 自然演繹の体系から矛盾が証明できたと仮定する．つまり，$\vdash \bot$ とする．ここで，$\bot$ は任意の命題変数である．カリー-ハワード対応より，ある λ-項 M の型が $\bot$ となる．つまり，$\vdash M : \bot$ である．弱正規化可能性定理より，M を β-簡約していくと，正規形に辿り着く．つまり，正規 λ-項 N で，$M \twoheadrightarrow_\beta N$ となるものが存在する．補題 3.8 より，β-簡約は型を変えないので，$\vdash N : \bot$ となる．

命題 3.13 より，正規形ならば，$xN_1 \ldots N_m$ または $\lambda x^\sigma.N'$ という形である．補題 3.7 より，コンテキストなしで型付け可能ならば，項は自由変数を含み得ない．よって，$\lambda x^\sigma.N'$ という形だが，この型は $\sigma \to \tau$ である．一方，$\bot$ はジェネリックな型変数であるから，$\sigma \to \tau$ という型にはなり得ない． □

前に見たように，ラムダ項には $(\lambda x.xx)(\lambda x.xx)$ などのように正規形を持たないものがあった．正規形を持たない項は，停止しない計算に相当する．定理 3.14 では，型付け可能なラムダ項は弱正規化可能であることを見たが，実際には，強正規化可能である．つまり，

型付けとは，計算の停止性（計算が無限ループに陥らないこと）を保証する行為

のひとつであると思える．ここでは，この強正規化可能性定理を証明することを目的とするが，まず強正規化可能性に関する基本的な性質を分析しよう．以後，SN_β によって，強正規化可能なラムダ項全体を表す．

補題 4.17.

1. $M_1, \ldots, M_n \in \mathrm{SN}_\beta$ ならば $xM_1 \ldots M_n \in \mathrm{SN}_\beta$.
2. $M_0\left[\frac{M_1}{x}\right] M_2 \ldots M_n \in \mathrm{SN}_\beta$ ならば $(\lambda x.M_0)M_1M_2 \ldots M_n \in \mathrm{SN}_\beta$.

Proof. (1) まず β-可約部を図的に見ると，2 分岐ノードの左手側に 1 分岐ノードがあるときに発生するのであった．しかし，$xM_1 \ldots M_n$ の構文木を見ると，

$$\dfrac{\dfrac{\dfrac{\dfrac{x \qquad M_1}{\vdots \qquad\quad M_{n-2}}}{xM_1 \ldots M_{n-2} \qquad M_{n-1}}}{xM_1 \ldots M_{n-1} \qquad\qquad M_n}}{xM_1 \ldots M_n}$$

という形状であるから，β-可約部は常に $M_1, \ldots, M_n$ のいずれかの上の部分にしか発生しない．よって，$M_1 \ldots M_n$ が強正規化可能ならば，$xM_1 \ldots M_n$ も強正規化可能である．また，x も強正規化可能であるから，$n = 0$ においても主張は成り立つことに注意する．

(2) $M_0[\frac{M_1}{x}]M_2 \ldots M_n$ が強正規化可能，つまり有限回の β-簡約の後に正規形 N に辿り着くならば，

$$(\lambda x.M_0)M_1 \ldots M_n \to_\beta M_0\left[\tfrac{M_1}{x}\right] M_2 \ldots M_n \twoheadrightarrow_\beta N$$

であるから，$(\lambda x.M_0)M_1 \ldots M_n$ も強正規化可能である． □

例 4.18. $M, N \in \mathrm{SN}_\beta$ であったとしても，$MN \in \mathrm{SN}_\beta$ であるとは限らない．たとえば，$M = N = \lambda x.xx$ を考えよ．

いま，任意の項 M と型変数 α について，

$$\mathrm{i} \models M : \alpha \iff \text{“}M \text{ は強正規化可能”}$$

と定義する．この定義は変数記号 α には依存しないことに注意する．さらに，一般の単純型については，帰納的に，対応する関数型の項たちを考える:

$$\mathrm{i} \models M : \sigma \to \tau \iff \text{“任意の } N \text{ について，} \mathrm{i} \models N : \sigma \text{ ならば，} \mathrm{i} \models MN : \tau\text{”}$$

このとき，補題 4.17 は，次のように拡張できる．

補題 4.19.

1. $M_1, \ldots, M_n \in \mathrm{SN}_\beta$ ならば $\mathrm{i} \models xM_1 \ldots M_n \colon \sigma$.
2. $\mathrm{i} \models M \colon \sigma$ ならば，M は強正規化可能である．
3. $M_0\left[\frac{M_1}{x}\right] M_2 \ldots M_n \in \mathrm{SN}_\beta$ であるとき，

$$\mathrm{i} \models M_0\left[\frac{M_1}{x}\right] M_2 \ldots M_n \colon \sigma \implies \mathrm{i} \models (\lambda x.M_0)M_1M_2 \ldots M_n \colon \sigma.$$

Proof. 型の複雑性に関する帰納法による．σ が型変数の場合は，補題 4.17 から主張は従うので，$\sigma = \tau \to \rho$ の場合のみ考えればよい．

(1) いま，$M_1, \ldots, M_n$ は強正規化可能であるとする．$\mathrm{i} \models N \colon \tau$ となる項 N を取る．(2) に対する帰納的仮定より，N は強正規化可能である．したがって，帰納的仮定より，$\mathrm{i} \models xM_1 \ldots M_nN \colon \rho$ は示されている．よって，$\mathrm{i} \models xM_1 \ldots M_n \colon \tau \to \rho$ である．ここで上の議論は $n = 0$ のときにも適用できることに注意する．つまり，$\mathrm{i} \models x \colon \tau \to \rho$ である．

(2) 型 $\sigma = \tau \to \rho$ を考え，$\mathrm{i} \models M \colon \tau \to \rho$ となる項 M を取る．このとき，$N = x$ とする．(1) に対する帰納的仮定より，$\mathrm{i} \models x \colon \tau$ である．したがって，定義より，$\mathrm{i} \models Mx \colon \rho$ であるから，帰納的仮定より Mx は強正規化可能である．M の中の無限ステップの簡約が存在するならば，Mx の中にも無限ステップの簡約が存在するが，これは強正規化可能性に矛盾する．したがって，M も強正規化可能である．

(3) $\mathrm{i} \models M_0\left[\frac{M_1}{x}\right] M_2 \ldots M_n \colon \tau \to \rho$ を仮定する．$\mathrm{i} \models N \colon \tau$ となる項 N について，定義より，$\mathrm{i} \models M_0\left[\frac{M_1}{x}\right] M_2 \ldots M_nN \colon \rho$ となる．... [ここは証明をもっと丁寧に書く．TTFP 参照]... よって，帰納的仮定より，$\mathrm{i} \models (\lambda x.M_0)M_1M_2 \ldots M_nN \colon \rho$ となる．したがって，定義より $\mathrm{i} \models (\lambda x.M_0)M_1M_2 \ldots M_n \colon \tau \to \rho$ である． □

以下，Var を変数の集合，Lambda をラムダ項の集合とする．$\mathrm{FV}(M)$ で項 M に現れる自由変数全体の集合を意味していたことを思い出そう．

定義 4.20. 付値 (*valuation*) とは，写像 $\rho \colon \mathrm{Var} \to \mathrm{Lambda}$ である．コンテキスト Γ が与えられているとき，

$$\rho \models \Gamma \iff \text{“任意の } (x \colon \sigma) \in \Gamma \text{ について，} \mathrm{i} \models \rho(x) \colon \sigma\text{”}$$

と定義する．ラムダ項 M について，$\mathrm{FV}(M) = \{x_1, x_2, \ldots, x_n\}$ であるとき，

$$\rho \models M \colon \sigma \iff \mathrm{i} \models M\left[\frac{\rho(x_1)}{x_1} \ldots \frac{\rho(x_n)}{x_n}\right] \colon \sigma$$

$$\Gamma \models M \colon \sigma \iff \text{“任意の付値 } \rho \text{ について，} \rho \models \Gamma \text{ ならば } \rho \models M \colon \sigma\text{”}$$

と定義する．

以後，$M\left[\frac{\rho(x_1)}{x_1}\dots\frac{\rho(x_n)}{x_n}\right]$ を $[\![M]\!]_\rho$ と略記する．それでは，次の健全性定理を証明しよう．

定理 4.21 (健全性定理)**.** $\Gamma \vdash M\colon \sigma$ ならば $\Gamma \models M\colon \sigma$ である．

Proof. 型付け $\Gamma \vdash M\colon \sigma$ の導出に関する帰納法による．

1. (Ax) まず，規則 (Ax) から導出されている場合，つまり $\Gamma = \Delta, x\colon \sigma$ かつ $M = x$ のときを考える．
$$\frac{}{\Delta, x\colon \sigma \vdash x\colon \sigma}\ (\mathrm{Ax})$$
もし $\rho \models \Gamma$ ならば，$\mathrm{i} \models \rho(x)\colon \sigma$ であるが，$[\![M]\!]_\rho = x[\frac{\rho(x)}{x}] = \rho(x)$ なので，$\rho \models M\colon \sigma$ である．よって，$\Gamma \models M\colon \sigma$ が示された．
2. (→E) 型付けの最後の規則が次である場合，
$$\frac{\Gamma \vdash M\colon \sigma \to \tau \qquad \Gamma \vdash N\colon \sigma}{\Gamma \vdash MN\colon \tau}\ (\to\mathrm{E})$$
$\rho \models \Gamma$ を仮定する．帰納的仮定より，$\Gamma \models M\colon \sigma \to \tau$ かつ $\Gamma \models N\colon \sigma$ であるから，$\rho \models M\colon \sigma \to \tau$ かつ $\rho \models N\colon \sigma$ である．よって，$\mathrm{i} \models [\![M]\!]_\rho\colon \sigma \to \tau$ かつ $\mathrm{i} \models [\![N]\!]_\rho\colon \sigma$ である．定義より，$\mathrm{i} \models [\![M]\!]_\rho[\![N]\!]_\rho\colon \tau$ が導かれる．ここで，代入の定義より，
$$[\![M]\!]_\rho[\![N]\!]_\rho = M\left[\frac{\rho(x_1)}{x_1}\dots\frac{\rho(x_n)}{x_n}\right]N\left[\frac{\rho(x_1)}{x_1}\dots\frac{\rho(x_n)}{x_n}\right] = (MN)\left[\frac{\rho(x_1)}{x_1}\dots\frac{\rho(x_n)}{x_n}\right] = [\![MN]\!]_\rho$$
となる．したがって，$\mathrm{i} \models [\![MN]\!]_\rho\colon \tau$ であり，つまり $\rho \models MN\colon \tau$ である．よって，$\Gamma \models MN\colon \tau$ が示された．
3. (→I) 型付けの最後の規則が次である場合，
$$\frac{\Gamma, x\colon \sigma \vdash M\colon \tau}{\Gamma \vdash \lambda x.M\colon \sigma \to \tau}\ (\to\mathrm{I})$$
$\rho \models \Gamma$ を仮定する．下式を $\models$ に変えるために，$\mathrm{i} \models N\colon \sigma$ となる項 N が与えられていると仮定する．このとき，$\rho[\frac{N}{x}] \models \Gamma, x\colon \sigma$ である．帰納的仮定より，$\Gamma, x\colon \sigma \models M\colon \tau$ であるから，$\rho[\frac{N}{x}] \models M\colon \tau$ である．$\rho' = \rho[\frac{N}{x}]$ と略記することにすれば，$\mathrm{i} \models [\![M]\!]_{\rho'}\colon \tau$ である．いま，
$$\begin{aligned}[\![\lambda x.M]\!]_\rho N &= (\lambda x.M)\left[\frac{\rho(x_1)}{x_1}\dots\frac{\rho(x_n)}{x_n}\right]N \\ &\to_\beta M\left[\frac{\rho(x_1)}{x_1}\dots\frac{\rho(x_n)}{x_n}, \frac{N}{x}\right] = [\![M]\!]_{\rho'}\end{aligned}$$
いま，$\mathrm{i} \models N\colon \sigma$ かつ $\mathrm{i} \models [\![M]\!]_{\rho'}\colon \tau$ であるから，補題 4.19 より，$\mathrm{i} \models [\![\lambda x.M]\!]_\rho N\colon \tau$ が導かれる．したがって，$\mathrm{i} \models [\![\lambda x.M]\!]_\rho\colon \sigma \to \tau$ である．つまり，$\rho \models \lambda x, M\colon \sigma \to \tau$ であるから，$\Gamma \models \lambda x.M\colon \sigma \to \tau$ が導かれた．

□

系 4.22. $\Gamma \vdash M : \sigma$ ならば，M は強正規化可能である．

Proof. $\Gamma \vdash M : \sigma$ ならば，健全性定理 4.21 より，$\Gamma \models M : \sigma$ である．補題 4.19 (1) の $n = 0$ の場合を用いれば，$\mathrm{i} \models \Gamma$ が従う．ここで，i は恒等付値である．よって，定義より，$\mathrm{i} \models M : \sigma$ である．したがって，補題 4.19 (2) より M が強正規化可能であることが導かれる． □

§5. 原始再帰法

5.1. 単純型付きラムダ計算における自然数論

純粋な単純型付きラムダ計算は，必ず計算の停止性が保証されるという点では良いが，素のままではあまりにも計算能力が低すぎる．たとえば，プログラミング言語においてループ命令は本質的な要素であるが，型なしラムダ計算においては，原始再帰法（命題 2.21）すなわち「`for` ループ（繰り返し回数が事前に分かるループ命令）」を実装するにしても，最小値探索すなわち「`while` ループ（繰り返し回数が事前に分からないループ命令）」を実装するにしても，不動点コンビネータを必要とした．しかし，不動点コンビネータは，型付け不可能な項を生み出し得る概念であるため，単純型付きラムダ計算の枠組みには入らない．したがって，単純型付きラムダ計算は，ある意味で「ループ命令を用いない計算論」なのである．このために，プレーンな単純型付きラムダ計算はあまりにも計算能力が低すぎる．

とはいえ，単純型付きラムダ計算で自然数上の関数をどの程度計算できるかを理解しておいて損は無いであろう．自然数 n はチャーチ数項 $\underline{n} = \lambda sz.s^n z$ として実装されていたことを思い出そう．s の型を $o \to o$ とし z の型を o とすれば，$\underline{n}$ には $\mathtt{nat} := (o \to o) \to o \to o$ という型が付く．

例 5.1. 足し算を実装するためには，以下の式に注目しよう．

$$s^{m+n} z = s^m (s^n z) = \underline{m} s (\underline{n} s z)$$

よって，$\underline{m+n} = \lambda sz.s^{m+n} z = \lambda sz.\underline{m} s (\underline{n} s z)$ である．以上より，加法は以下のように実装すればよい．

$$\mathtt{add} = \lambda mnsz.ms(nsz)$$

$\mathtt{add}$ の型が $\mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$ であることは容易に確認できる．

例 5.2. 掛け算については，$s^{mn} z = (s^m)^n z$ という分解を考えると，以下の式を得る．

$$\underline{n}(\underline{m} s) z \to (\underline{m} s)^n z \to \underbrace{\underline{m} s (\underline{m} s (\dots \underline{m} s}_{n \text{ times}} z) \dots) \to \underbrace{\underline{m} s (\underline{m} s (\dots \underline{m} s}_{n-1 \text{ times}} (s^m z)) \dots) \to s^{mn} z$$

よって，$\underline{m \cdot n} = \lambda sz.\underline{n}(\underline{m}s)z$ である．以上より，乗法は以下のように実装すればよい．

$$\texttt{mult} = \lambda mnsz.n(ms)z$$

mult の型が nat → nat → nat であることは容易に確認できる．

例 5.3. 条件分岐については，以下の式に注目しよう．

$$\underline{n}(\lambda x.b)a \to \begin{cases} a & \text{if } n = 0 \\ b & \text{otherwise} \end{cases}$$

ここで，x は b に現れない変数である．以上より，条件分岐は以下のように実装できる．

$$\texttt{if } n \texttt{ iszero then } a \texttt{ else } b \quad = \quad n(\lambda x.b)a$$

項 n, a, b, x の型がいずれも nat であれば，上記の式の型は nat である．

以上より，単純型付きラムダ計算においても，多項式と条件分岐程度は取り扱えるようである．しかし，実を言えば，これがプレーンな単純型付きラムダ計算の限界である．多項式と条件分岐の組合せで表すことのできる関数は**拡張多項式** (*extended polynomial*) と呼ばれ，これが単純型付きラムダ計算で記述できる関数と対応することが知られている．プログラミング言語的な観点からは，拡張多項式とは，以下の操作のみで記述できる関数である．

```
c:=0;   c:=1;   c:=a+b;   c:=a*b;   if n iszero then a else b;
```

さて，単純型付きラムダ計算の大きなメリットの一つは，計算の停止性保証である．しかし，単純型付きラムダ計算の枠組の外にありつつ，計算の停止性保証がされるような手続きがある．それはたとえば，「for ループ」であり，原始再帰法である．我々のこれからの目標は，原始再帰法を型付きラムダ計算の枠組の中に取り込むことであるが，その前に原始再帰法の復習をしていこう．

5.2. 原始再帰法と LOOP 言語

■原始再帰法とは何か: 自然数の足し算および掛け算のような基本的な演算から，**原始再帰法** (*primitive recursion*)[*4]と呼ばれる操作によって，様々な自然数上の関数を構成できることを本節では見ていこう．

このアイデアを説明するために，人類が「新しい演算」を創造していく過程をシミュレートしよう[*5]．まず，自然数 x が与えられたとき，「x の次」である数が何であるかを知っているとしよう．

[*4] Primitive recursion は伝統的には，原始再帰でなく**原始帰納**と訳されることがある．しかし，再帰理論やその周辺の理論では，inductive と recursive が全く別概念として登場し，たとえば，「recursive ではないが inductive であるような集合が存在する」「Π^1_1-transfinite induction は Π^1_1-transfinite recursion を導かない」というような定理が成立する．したがって，inductive と recursive には異なる訳語を割り当てる必要があるが，ここでは inductive を**帰納**と訳し，recursive を**再帰**と訳す流儀を採用する．

[*5] 本稿の記述は現実の数学史に沿っているとは限らない．

すると，「x の次の次」や「x の次の次の次」などを考えることができる．しかし，いくつも「の次」という文字を書くのは億劫なので，x の y 個次の数を $x+y$ と書くことにしよう．こうして，人類は，「足す」という演算を生み出した．

$$x+y = x \text{ の } \underbrace{\text{次の次の次} \ldots \text{ の次の次}}_{y \text{ 個}}$$

このように「足す」という演算を知った人類は，$x+x+x$ や $x+x+x+x+x$ のように x を何度も足す，という演算が有用であることに次第に気づき始める．これを簡潔に表すために，人類は「掛ける」という演算を次のように定義した．

$$x \cdot y = \underbrace{x+x+\cdots+x+x}_{y \text{ 個}}$$

そして，「掛ける」という演算を知った人類は，$x \cdot x \cdot x$ のように x を何度も掛ける，という演算の有用性に気づく．そして「累乗」という演算を次のように定義した．

$$x^y = \underbrace{x \cdot x \cdot \cdots \cdot x \cdot x}_{y \text{ 個}}$$

すると，自然に x^{x^x} や $x^{x^{x^x}}$ のようなものを考える人も現れる．上方向にたくさん添字が付くのは見づらいので，x^y のことを今後は $x \uparrow y$ と書くこととしよう．たとえば，x^{x^x} は $x \uparrow (x \uparrow x)$ であり，$x^{x^{x^x}}$ は $x \uparrow (x \uparrow (x \uparrow x))$ である．また，以下，括弧は省略し，これらの演算は右から順に適用するものとする．さて，累乗でも飽き足らない一部の人類は，「テトレーション」という演算を編み出した．

$$x \uparrow\uparrow y = \underbrace{x \uparrow x \uparrow \ldots \uparrow x \uparrow x}_{y \text{ 個}}$$

飽くなき人類は，更なる演算「ペンテーション」を定義する．

$$x \uparrow\uparrow\uparrow y = \underbrace{x \uparrow\uparrow x \uparrow\uparrow \ldots \uparrow\uparrow x \uparrow\uparrow x}_{y \text{ 個}}$$

より一般に，クヌースの矢印記法というものは以下によって定義される．

$$x \uparrow^{n+1} y = \underbrace{x \uparrow^n x \uparrow^n \ldots \uparrow^n x \uparrow^n x}_{y \text{ 個}}$$

このような再帰的な関数構成を数学的に抽象化したものが，原始再帰法と呼ばれる概念である．

さて，ここまで，何かの演算 $x \diamond y$ を元に，人類が新たな演算 $x \star y$ を創造する過程を見てきた．これらの過程が共有するものとは何であろうか．それは以下の性質である．

$$x \star y = \underbrace{x \diamond x \diamond \cdots \diamond x \diamond x}_{y \text{ 個}}$$

実際に，この値 $x \star y$ を計算する場合には，$x \star 2 = x \diamond x$ を求め，$x \star 3 = x \diamond x \diamond x = x \diamond (x \star 2)$ を求め，$x \star 4 = x \diamond x \diamond x \diamond x = x \diamond (x \star 3)$ を求め，... という手続きを行うこととなるだろう．たとえば，掛け算以降の演算の定義を少し書き直せば，次のようにして定義されていることがわかる．

$$\begin{cases} x \star 1 = x, \\ x \star (y+1) = x \diamond (x \star y). \end{cases}$$

上の定義では曖昧であるが，$x \star 0$ の場合も定義しておくのが自然である．たとえば，$x \cdot 0 = 0$ であるし，$x \uparrow 0 = x^0 = 1$ である．

演習問題 5.4. 以下のようにして，$x \uparrow^{n+1} y$ を定義する．

$$\begin{cases} x \uparrow^{n+1} 0 = 1, \\ x \uparrow^{n+1} (y+1) = x \uparrow^{n} (x \uparrow^{n+1} y). \end{cases}$$

このとき，$x \uparrow^{n+1} 1 = x$ であることを示せ．

さて，ここまでは演算の形式で書いてきたが，これらを関数として見直すこととする．つまり，今までのことを，関数 $h(x,y) = x \diamond y$ から新たな関数 $f(x,y) = x \star y$ を創造する過程を考えてきたものとしよう．また，$g(x) = x \star 0$ は与えられているものとする．これまでの内容を言い直せば，g と h からの新たな関数 f は以下のように生み出された．

$$\begin{cases} f(x,0) = g(x), \\ f(x,y+1) = h(x, f(x,y)). \end{cases}$$

それでは，原始再帰法の厳密な定義に入ろう．上では，f は 2 変数関数であったが，より多変数であってよい．

定義 5.5 (原始再帰法)**.** 関数 $g : \mathbb{N}^n \to \mathbb{N}$ と $h : \mathbb{N}^{n+2} \to \mathbb{N}$ が与えられているとする．さらに，$f : \mathbb{N}^{n+1} \to \mathbb{N}$ が，次のように定義されるとしよう: 任意の $\bar{x} \in \mathbb{N}^n$ と $y \in \mathbb{N}$ について，

$$\begin{cases} f(\bar{x}, 0) = g(\bar{x}), \\ f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y)). \end{cases}$$

このとき，f は g と h から**原始再帰法** (*primitive recursion*) によって定義されるという．

意味をわかりやすくするために，$f_y = f(\bar{x}, y)$ および $h_y(\bar{x}, z) = h(\bar{x}, y, z)$ として，さらに $\bar{x}$ を省略すれば，

$$f_n = h_n(h_{n-1}(\dots h_1(h_0(g))\dots))$$

というような関数の繰り返し適用による新たな関数の構成を意味している．本節の導入において，「の次」を初期関数として，原始再帰法を有限回だけ用いて，数多くの関数を構成してきた．このような関数は，原始再帰関数と呼ばれる関数の一種である．より正確には，原始再帰関数の概念を以下のように導入しよう．

定義 5.6 (原始再帰関数)**.** 以下のように，**原始再帰関数** (*primitive recursive function*) を帰納的に定義する．

1. 以下によって定義される後続関数 $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$, 零関数 $\mathsf{zero}^n : \mathbb{N}^n \to \mathbb{N}$ および射影関数 $\mathsf{proj}_i^n : \mathbb{N}^n \to \mathbb{N}$ は原始再帰関数である．
$$\mathsf{succ}(x) = x + 1, \quad \mathsf{zero}^n(x_1, \ldots, x_n) = 0, \quad \mathsf{proj}_i^n(x_1, \ldots, x_n) = x_i.$$
2. 原始再帰関数たちの合成は原始再帰関数である．つまり，$h : \mathbb{N}^m \to \mathbb{N}$ と $g_1, \ldots, g_m : \mathbb{N}^n \to \mathbb{N}$ が原始再帰的ならば，以下のように定義される関数 $f : \mathbb{N}^n \to \mathbb{N}$ もまた原始再帰的である．
$$f(\bar{x}) = h(g_1(\bar{x}), \ldots, g_m(\bar{x})).$$
3. 原始再帰関数 g, h から原始再帰法によって定義される関数は原始再帰的である．

以後，後続関数，零関数，射影関数のことを**初期関数** (*initial function*) と呼ぶ．つまり，原始再帰関数全体の族とは，初期関数を含み合成と原始再帰法で閉じた最小の関数族として与えられる．

例 5.7. 和 $(x, y) \mapsto x + y$, 積 $(x, y) \mapsto x \cdot y$, 冪乗 $(x, y) \mapsto x^y$, 第 n 矢印演算 $(x, y) \mapsto x \uparrow^n y$ はいずれも原始再帰的関数である．

歴史. 本節の導入のような関数構成を，定義 5.5 のような原始再帰法として抽象化した歴史上最初の人物が誰であるかは不明瞭である．漸化式なども再帰式の一種であるから，フィボナッチ数列くらいまで歴史は遡るのかもしれない．1861 年のヘルマン・グラスマン (Hermann Grassmann; 1809–1877) は高校数学の教科書を執筆する過程で，足し算と掛け算の再帰的定義に注目した．1889 年のジュゼッペ・ペアノ (Giuseppe Peano; 1858–1932) もまた，足し算と掛け算の性質を数学的帰納法に還元するために，これらの再帰的定義を強調した．しかし，彼らはそもそも足し算と掛け算しか扱っていないため，原始再帰法の使用者と呼ぶのは厳しいだろう．1888 年のリヒャルト・デデキント (Richard Dedekind; 1831–1916) は，一般の関数の繰り返し定義 $f^{(n)}(x)$ の理論的考察を行った．このため，デデキントを原始再帰法の祖と呼ぶことは可能である．しかし，デデキントの再帰は，原始再帰法 $h_n(h_{n-1}(\ldots(h_0(g))))$ の特殊ケースに過ぎない．現代的な原始再帰法は，1923 年頃のトアルフ・スコーレム (Thoralf Skolem; 1887–1963) らによって初めて徹底的に研究されることとなった．

■プログラミング言語 LOOP: 原始再帰関数は常に全域関数であり，つまりは無限ループに陥ることはない．原始再帰関数は極めて単純なプログラミング言語で記述することができる．

定義 5.8. プログラミング言語 LOOP は以下の 3 種類の命令からなる．

1. x`:=0` ····· 変数 x に数 0 を代入する．
2. x`++` ····· 変数 x に格納された値を 1 増加させる．
3. `loop` x `do` P `end` ····· プログラム P を x 回ループする．

バッカス-ナウア記法を用いれば，LOOP プログラムの構文規則は以下のものである．

$$P ::= \ x\texttt{:=0} \mid x\texttt{++} \mid \texttt{loop}\ x\ \texttt{do}\ P\ \texttt{end} \mid P; P$$

このプログラミング言語 LOOP は 1967 年にリッチー (Dennis Ritchie) が原始再帰関数の特徴付けのために考案したものである[*6]．ループ命令のループ回数 x は事前に指定されているので，無限ループは決して発生しない．つまり，言語 LOOP における計算は必ず有限ステップで停止する．

興味深いことに，この言語 LOOP による計算可能性が正確に原始再帰関数を特徴付けるのである．原始再帰関数の定義よりも単純で，なおかつ，その本質が（事前に繰り返し回数の分かる）ループであるということを浮き彫りにする，エレガントな言語である．この言語 LOOP を用いたプログラムを幾つか記述してみよう．

例 5.9. 代入命令 $x\texttt{:=}y$ は，以下のように記述できる．

$$x\texttt{:=}y \quad \equiv \quad x\texttt{:=0;}\ \texttt{loop}\ y\ \texttt{do}\ x\texttt{++}\ \texttt{end;}$$

以後は，入力用の特別な変数 $\mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \ldots$ と出力用の特別な変数 y があるとしよう．入力読み込み命令 read と出力命令 print のようなものがあるとしてもよい．暗黙に入出力命令があるとすれば，入力変数と出力変数はプログラム毎に異なるものに変えてもよい．

例 5.10. 基本関数を LOOP プログラムによって記述してみよう．

$$\begin{aligned} \mathsf{zero} &\equiv \mathtt{y:=0;} \\ \mathsf{succ} &\equiv \mathtt{y:=x}_1\mathtt{;\ y++;} \\ \mathsf{proj}^n_k &\equiv \mathtt{y:=x}_k\mathtt{;} \end{aligned}$$

例 5.11. LOOP プログラムに慣れるため，具体的な原始再帰関数の記述を行ってみよう．

$$\begin{aligned} \mathtt{add} &\equiv \mathtt{y:=x}_1\mathtt{;\ loop\ x}_2\mathtt{\ do\ y++\ end;} \\ \mathtt{pred} &\equiv \mathtt{y:=0;\ z:=0;\ loop\ x}_1\mathtt{\ do\ y:=z;\ z++\ end;} \end{aligned}$$

後者の pred は入力の値を 1 減算する演算である．ただし，入力が 0 の場合には，出力も 0 である．他の例としては，条件分岐命令 $\texttt{if}\ \mathtt{x}_1\ \texttt{iszero then}\ P\ \texttt{else}\ Q$ は以下のように記述できる．

```
a:=1; b:=0;   /* a,b は P,Q に現れない変数 */
loop x1 do a:=0; b:=1 end;
loop a do P end;
loop b do Q end;
```

[*6] 実際にはメイヤー (Albert Meyer) との共著論文として発表されているが，この共著論文はこれ以外のトピックも含むものであり，後にメイヤーは LOOP プログラムに関してはリッチーによる発案であると述べている．ちなみにリッチーは，C 言語や UNIX などを開発した人物でもある．

例 5.12. もちろん，既に構成した関数をプログラム中に含めてもよい．たとえば，関数 f を既に LOOP プログラム F によって記述していたとしよう．代入命令 $z := f(a_1, \ldots, a_n)$ を以下のように記述できる．

$$\mathtt{z{:=}}f(\mathtt{a}_1, \ldots, \mathtt{a}_n) \quad \equiv \quad \mathtt{x}'_1 \mathtt{:=} \mathtt{a}_1;\ \ldots;\ \mathtt{x}'_n \mathtt{:=} \mathtt{a}_n;\ F';\ \mathtt{z{:=}y}'$$

ここで，プログラム F' は，F に出現するすべての変数 $\mathtt{x}$ を新しい変数 $\mathtt{x}'$ に置き換えたものである．この補正が必要な理由としては，プログラム P 中にプログラム F を組み込んだとき，P で利用していた変数の値が F の計算中で書き換わってしまうかもしれない．これを回避するために，上記の補正によって

$$P \text{ と } F' \text{ は共通の変数を計算に用いない}$$

という保証がされたのである．この補正によって，F' の入力変数は $\mathtt{x}'_1, \ldots, \mathtt{x}'_n$ となり，出力変数は $\mathtt{y}'$ となっているが，前述のようにこれは問題を引き起こさない．たとえば，LOOP プログラムには常に read, print 命令が隠れているという設定にすれば，この扱いは容易である．

$$\mathtt{read}\ \mathtt{x}'_1\mathtt{x}'_2 \ldots \mathtt{x}'_n;\ F';\ \mathtt{print}\ \mathtt{y}';$$

このアイデアを用いれば，たとえば以下のような関数を言語 LOOP において実装できる．

$$\begin{aligned}
\mathtt{mult} &\equiv \mathtt{y{:=}}0;\ \mathtt{loop}\ \mathtt{x}_2\ \mathtt{do}\ \mathtt{y{:=}add(y,x}_1)\ \mathtt{end};\\
\mathtt{exp} &\equiv \mathtt{y{:=}}1;\ \mathtt{loop}\ \mathtt{x}_2\ \mathtt{do}\ \mathtt{y{:=}mult(y,x}_1)\ \mathtt{end};\\
\mathtt{monus} &\equiv \mathtt{y{:=}x}_1;\ \mathtt{loop}\ \mathtt{x}_2\ \mathtt{do}\ \mathtt{y{:=}pred(y)}\ \mathtt{end};
\end{aligned}$$

例 5.13 (原始再帰法)**.** LOOP-計算可能関数 $g\colon \mathbb{N}^n \to \mathbb{N}$ と $h\colon \mathbb{N}^{n+2} \to \mathbb{N}$ から原始再帰法によって関数 $f\colon \mathbb{N}^{n+1} \to \mathbb{N}$ が構成されているとしよう．関数 g, h はそれぞれプログラム G, H によって記述されていると仮定する．ここで，G と H に共通の変数が出現しないことも仮定する．入出力変数を明示するために，read, print を用いた形式の記述を行う．

$$\begin{aligned}
g &\equiv \mathtt{read}\ x_1x_2 \ldots x_n;\ G;\ \mathtt{print}\ y;\\
h &\equiv \mathtt{read}\ x'_1x'_2 \ldots x'_{n+2};\ H;\ \mathtt{print}\ y';
\end{aligned}$$

このとき，f は以下の LOOP プログラムによって記述できる．

$$\begin{aligned}
&\mathtt{read}\ x''_1x''_2 \ldots x''_{n+1};\ x_1 \mathtt{:=} x''_1;\ \ldots;\ x_n \mathtt{:=} x''_n;\\
&\quad /*\ x''_1, \ldots, x''_{n+1} \text{ は } G, H \text{ に現れない変数 } */\\
&G;\ t \mathtt{:=} 0; \quad /*\ t \text{ は } H \text{ に現れない変数 } */\\
&\mathtt{loop}\ x''_{n+1}\ \mathtt{do}\ x'_1 \mathtt{:=} x''_1;\ \ldots;\ x'_n \mathtt{:=} x''_n;\ x'_{n+1} \mathtt{:=} y;\ x'_{n+2} \mathtt{:=} t;\\
&H;\ t\mathtt{++};\ y \mathtt{:=} y'\ \mathtt{end};\ \mathtt{print}\ y;
\end{aligned}$$

```
read x''_1 x''_2 ... x''_{n+1};
  /* x''_1, ..., x''_{n+1} は G, H に現れない変数 */
x_1 := x''_1; ...; x_n := x''_n;
G; t := 0;    /* t は H に現れない変数 */
loop x''_{n+1} do
x'_1 := x''_1; ...; x'_n := x''_n; x'_{n+1} := y; x'_{n+2} := t;
H; t++; y := y' end;
print y;
```

以上の議論より，リッチーの定理は容易に導かれる．

定理 5.14. 自然数上の関数が LOOP-計算可能であることと原始再帰的であることは同値である．

もちろんリッチーの定理の証明は容易であるが，言語 LOOP を発案したという点が重要である．更に言えば，あくまでこれはリッチーの研究の第一ステップに過ぎず，たとえば，リッチーは他にもループのネストの深さによる原始再帰関数の分類を行い，これがグジェゴルチク階層と呼ばれるものと対応することを示している．

5.3. 初等関数とグジェゴルチク階層

原始再帰関数の中にも，構成が簡単なものから難しいものまで様々な関数がある．この原始再帰の内部構造を理解するために，原始再帰関数の階層構造を考察しよう．その基点となる関数のクラスとして，初等関数という概念を導入する．初等関数の族とは，初期関数と加法，部分的減法を含み，合成と総和，総乗で閉じている最小の関数族である．より正確には，以下のように定義される．

定義 5.15. 以下のように，**初等関数** (*elementary function*) を帰納的に定義する．

1. 初期関数，加法 $x+y$, 部分的減法 $x \dot{-} y$ は初等関数である．
2. 初等関数たちの合成は初等関数である．
3. 初等関数から総和, 総乗によって定義される関数は初等関数である．つまり, $p : \mathbb{N}^{n+1} \to \mathbb{N}$ が初等関数ならば，以下のように定義される関数 $f, g : \mathbb{N}^{n+1} \to \mathbb{N}$ もまた初等関数である．

$$f(\bar{x}, y) = \sum_{i=0}^{y-1} p(\bar{x}, i) \qquad g(\bar{x}, y) = \prod_{i=0}^{y-1} p(\bar{x}, i).$$

例 5.16. 任意の $n \in \mathbb{N}$ について，関数 $f(x) = x \uparrow\uparrow n$ は初等関数である．なぜなら，総乗を用い

れば，指数関数 x^y が初等関数であることは明らかで，$x \uparrow\uparrow n$ は指数関数の n 回合成として表されるためである．

また，例**??**より，任意の初等関数は原始再帰的である．一方，$f(x) = x \uparrow\uparrow x$ は初等関数ではない．よって，初等関数でないような原始再帰的関数が存在する:

$$\text{初等関数全体の族} \subsetneq \text{原始再帰的関数全体の族}.$$

したがって，初等関数の構成は，ある制限された原始再帰と考えることができる．これをより明快に理解するために，以下の概念を導入しよう．

定義 5.17 (有界原始再帰法)**.** 関数 $g : \mathbb{N}^n \to \mathbb{N}$, $h : \mathbb{N}^{n+2} \to \mathbb{N}$ および $t : \mathbb{N}^{n+1} \to \mathbb{N}$ が与えられているとする．さらに，$f : \mathbb{N}^{n+1} \to \mathbb{N}$ が，次のように定義されるとしよう: 任意の $\bar{x} \in \mathbb{N}^n$ と $y \in \mathbb{N}$ について，

$$\begin{cases} f(\bar{x}, 0) = g(\bar{x}), \\ f(\bar{x}, y+1) = h(\bar{x}, y, f(\bar{x}, y)), \\ f(\bar{x}, y) \leqslant t(\bar{x}, y) \end{cases}$$

このとき，f は g, h, t から**有界原始再帰法** (*bounded primitive recursion*) によって定義されるという．

初等関数は，有界原始再帰法によって特徴づけることができる．このために，まず，素数判定などは初等関数によって行うことができ，したがって列のコーディングなども初等関数によって行うことができることに注意する．以下の定理の証明の前半の議論は，原始再帰によるコーディングの議論に慣れていないと難しいので，飛ばしてしまっても構わない．

定理 5.18. 初等関数全体の族は，以下を満たす最小の関数族と正確に一致する．

- 初期関数と指数関数 x^y を含む．
- 合成と有界原始再帰法で閉じている．

Proof. 初等関数全体の族が有界原始再帰法で閉じていることを示す．f が初等関数 g, h, t から有界原始再帰法で定義されていると仮定する．まず

$$\begin{aligned} m = \langle f(\bar{x}, 0), \ldots, f(\bar{x}, y) \rangle \iff\ & lh(m) = y+1 \ \wedge\ (m)_0 = g(\bar{x}) \\ & \wedge\ (\forall i < y)\ (m)_{i+1} = h(\bar{x}, i, (m)_i) \end{aligned} \tag{1}$$

であり，上で述べたように，(1) の右式の真偽判定は初等関数によって行うことができる．また，$f(\bar{x}, y) \leqslant t(\bar{x}, y)$ であるから，$t^+(\bar{x}, y) = \sum_{i \leqslant y} t(\bar{x}, i)$ とすると $f(\bar{x}, i) \leqslant t^+(\bar{x}, y)$ である．よって，$q(\bar{x}, y) = (p_y^{t^+(\bar{x}, y)})^{y+1}$ とすれば $\langle f(\bar{x}, 0), \ldots, f(\bar{x}, y) \rangle \leqslant q(\bar{x}, y)$ であることが分かる．いま，q は明らかに初等関数である．

以上より，$f(\bar{x}, y)$ は (1) の条件を満たす $m \leqslant q(\bar{x}, y)$ について $(m)_y$ として得られる．より正確には，

$$h(\bar{x}, y, m) = 1 \iff h(\bar{x}, y, m) \dot{+} 0 \iff m \text{ は (1) の条件を満たす}$$

と h を定義すると，h は初等関数であり，

$$f(\bar{x}, y) = \sum_{m \leqslant q(\bar{x}, y)} h(\bar{x}, m) \cdot (m)_y$$

が成立する．よって，f は初等関数である．

逆方向については，総和と総乗が有界原始再帰法によって定義できることを示せばよい．例**??**において総和と総乗が原始再帰法によって定義できることを示したので，これらの有界性をみればよい．これについては，

$$\sum_{z \leqslant y} p(\bar{x}, z) \leqslant (y+1) \cdot \max_{z \leqslant y} p(\bar{x}, z),$$
$$\prod_{z \leqslant y} p(\bar{x}, z) \leqslant (\max_{z \leqslant y} p(\bar{x}, z))^{y+1}$$

であるから，$q(\bar{x}, y) = \max_{z \leqslant y} p(\bar{x}, z)$ が有界原始再帰法によって構成できることを示せばよい．命題**??**で原始再帰的に構成した場合分け関数の特殊なケースである次の関数

$$\texttt{[if } z \texttt{ is zero then } x \texttt{ else } y\texttt{]} = \begin{cases} x & \text{if } z = 0, \\ y & \text{if } z \dot{+} 0. \end{cases}$$

は $x + y$ で有界であるから，有界原始再帰法によって定義できる．$\{p(\bar{z})\}_{z \leqslant y}$ の中から最大値を達成する z を探索する関数 p^* を次の原始再帰法によって定義する．

$$\begin{aligned} p^*(\bar{x}, 0) &= 0, \\ p^*(\bar{x}, z+1) &= \begin{cases} p^*(\bar{x}, z) & \text{if } p(\bar{x}, z+1) \leqslant p(\bar{x}, p^*(\bar{x}, z)) \\ z+1 & \text{otherwise.} \end{cases} \\ &= \texttt{[if } p(\bar{x}, z+1) \dot{-} p(\bar{x}, p^*(\bar{x}, z)) \texttt{ is zero then } p^*(\bar{x}, z) \texttt{ else } z+1\texttt{]}. \end{aligned}$$

明らかに $p^*(\bar{x}, z) \leqslant z$ であるから，有界原始再帰法によって p^* を構成できる．このとき，明らかに $\max_{z \leqslant y} p(\bar{x}, z) = p(\bar{x}, p^*(\bar{x}, y))$ であるから，定理は示された． □

有界原始再帰で初等関数を構成できることを示した．原始再帰関数にどのようなものがあるかを理解するために，原始再帰関数を構成に必要な原始再帰法の利用回数によって分類し，その各レベルがどのような関数を含むかを分析しよう．具体的には，次の関数族を考える．

$$\mathcal{PR}_n = \text{初等関数を始点とし，高々 } n \text{ 回の原始再帰法の適用で構成できる関数族.}$$

より正確には，以下のように定義する．

定義 5.19. 各 $n \in \mathbb{N}$ について，原始再帰関数の族 $\mathcal{PR}_n$ を以下のように帰納的に定義する．

1. $\mathcal{PR}_0$ を初等関数全体の族とする．
2. $\mathcal{PR}_{n+1}$ を $\mathcal{PR}_n$ から任意回の合成と高々 1 回の原始再帰法の適用で構成できる関数全体の族とする．

例 5.20. 演習問題 5.4 で定義を与えたクヌースの矢印表記 $\uparrow^n$ について，定義より指数関数 $\uparrow^1$ は初等関数である．よって，$\uparrow^{n+1}$ は $\mathcal{PR}_n$ に属す．一方，$\uparrow^{n+2}$ は $\mathcal{PR}_n$ に属さない (補題 5.22)．

以後，$\mathcal{PR}_n$ に属す関数を，**階数 n の原始再帰関数**と呼ぶこととする．$\mathcal{PR}_n$ の性質を調べるために，次のような初等関数の相対化を考えよう．

定義 5.21. 関数 $g : \mathbb{N} \to \mathbb{N}$ が与えられているとする．このとき，以下のように，**g-初等関数** (g-*elementary function*) を帰納的に定義する．

1. 初期関数，加法 $x + y$, 部分的減法 $x \dot{-} y$, および g は g-初等関数である．
2. g-初等関数たちの合成は g-初等関数である．
3. g-初等関数から総和，総乗によって定義される関数は g-初等関数である．

g-初等関数全体の族を $\mathcal{E}(g)$ と書こう．ここで主に取り扱うものは $\mathcal{E}(\uparrow^n)$ である．この階層は，**グジェゴルチク階層** (*Grzegorczyk hierarchy*) と呼ばれる．まず，グジェゴルチク階層が巨大関数の階層を与えることを示そう．以下，$\uparrow^n$ (x) によって $x \uparrow^n x$ を表す．関数 $f, g : \mathbb{N} \to \mathbb{N}$ について，g が f を**支配** (*dominate*) するとは，次が成立することである．

$$(\exists d \in \mathbb{N})(\forall x \geqslant d)\ f(x) \leqslant g(x).$$

補題 5.22. $n \geqslant 1$ とする．任意の関数 $f \in \mathcal{E}(\uparrow^n)$ について，$\uparrow^n$ のある定数 c 回合成 $(\uparrow^n)^{(c)}$ が f を支配する．

Proof. 証明はさほど難しくないので, 興味があったら, たとえば Odifreddi [**?**, Theorem VIII.8.10] を参考にしてほしい． □

定理 5.23. 任意の $n \in \mathbb{N}$ について，$\mathcal{PR}_n = \mathcal{E}(\uparrow^{n+1})$ が成立する．つまり，

$$\text{階数 } n \text{ の原始再帰関数} = (\uparrow^{n+1})\text{-初等関数}.$$

Proof. まず，$\uparrow^{n+1} \in \mathcal{PR}_n$ であるから，$\mathcal{E}(\uparrow^{n+1}) \subseteq \mathcal{PR}_n$ であることは明らかである．逆向きの包含関係を帰納法により証明する．$\mathcal{PR}_n = \mathcal{E}(\uparrow^{n+1})$ は既に示されていると仮定する．

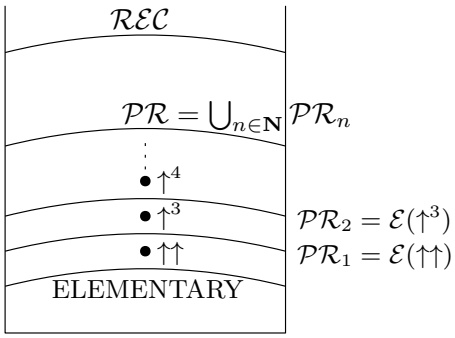


図 2　原始再帰関数のグジェゴルチク階層

$\mathcal{PR}_{n+1} = \mathcal{E}(\uparrow^{n+2})$ を示すためには，$\mathcal{E}(\uparrow^{n+1})$ の関数から 1 回の原始再帰法の適用で定義できる関数が $\mathcal{E}(\uparrow^{n+2})$ に属すことを示せばよい．$g, h \in \mathcal{E}(\uparrow^{n+1})$ から原始再帰法によって f が構成されていると仮定する．f の構成過程をコードする関数 t を次によって定義する．

$$t(\langle \bar{x}, n, z\rangle) = \langle \bar{x}, n+1, h(\bar{x}, n, z)\rangle.$$

このとき，

$$t^{(y)}(\langle \bar{x}, 0, g(\bar{x})\rangle) = \langle \bar{x}, y, f(\bar{x}, y)\rangle.$$

が成立するから，$(x, y) \mapsto t^{(y)}(x)$ から合成を用いて f を定義できる．これより，関数 $(x, y) \mapsto t^{(y)}(x)$ が $\mathcal{E}(\uparrow^{n+2})$ に属すことを示せばよい．定理 5.18 と同様にして，$\mathcal{E}(\uparrow^{n+2})$ が有界原始再帰法で閉じていることは示せるので，ある $s \in \mathcal{E}(\uparrow^{n+2})$ が存在して，$t^{(y)}(x) \leqslant s(x, y)$ であることを示せば十分である．補題 5.22 より，ある定数 c が存在して，十分大きな x について，$t(x) \leqslant (\uparrow^{n+1})^{(c)}(x)$ が成立する．よって，十分大きな x, y について，

$$t^{(y)}(x) \leqslant (\uparrow^{n+1})^{(c+y)}(x) \leqslant \uparrow^{n+2} (x + y).$$

よって，$t^{(y)}(x)$ は $\mathcal{E}(\uparrow^{n+2})$ の関数で抑えられることが示された．以上より，$\mathcal{PR}_{n+1} = \mathcal{E}(\uparrow^{n+2})$ が結論付けられる．　□

以上のようにして，原始再帰関数のクラスは，構成に必要な原始再帰法の利用回数，あるいはクヌースの矢印表記によって階層分けすることができる (図 2)．

$$\mathcal{E} = \mathcal{PR}_0 \subsetneq \mathcal{E}(\uparrow\uparrow) = \mathcal{PR}_1 \subsetneq \cdots \subsetneq \mathcal{E}(\uparrow^{n+1}) = \mathcal{PR}_n \subsetneq \mathcal{E}(\uparrow^{n+2}) = \mathcal{PR}_{n+1} \subsetneq \ldots$$
$$\cdots \subsetneq \mathcal{PR} = \bigcup_{n\in\mathbb{N}} \mathcal{PR}_n \subsetneq \cdots \subsetneq \mathcal{REC}.$$

ここで，$\mathcal{REC}$ は計算可能関数全体のクラスを表す．

5.4. 原始再帰法のシステム T_0

ラムダ計算の文脈で原始再帰関数を記述する簡便なシステム T_0 を準備しておく．システム T_0 の各項は，自然数または自然数上の関数を表す．以下，$\mathtt{N}$ によって自然数の型を表すとする．また，

$\mathtt{N}^n \to \mathtt{N}$ は自然数上の n 変数関数の型であり，$\mathtt{N}$ と $\mathtt{N}^0 \to \mathtt{N}$ は同一視する．

定義 5.24. システム T_0 の項は，以下のように帰納的に定義される．

1. 可算個の変数 x が用意されており，これは型 $\mathtt{N}$ 項である．
2. 記号 $\underline{0}$ は型 $\mathtt{N}$ 項であり，記号 $\mathtt{succ}$ は型 $\mathtt{N} \to \mathtt{N}$ 項である．
3. s が型 $\mathtt{N}^{n+1} \to \mathtt{N}$ 項で t が型 $\mathtt{N}$ 項ならば，st は型 $\mathtt{N}^n \to \mathtt{N}$ 項である．
4. x が変数であり t が型 $\mathtt{N}^n \to \mathtt{N}$ 項ならば，$\lambda x.t$ は型 $\mathtt{N}^{n+1} \to \mathtt{N}$ 項である．
5. s が型 $\mathtt{N}^n \to \mathtt{N}$ 項で t が型 $\mathtt{N}^{n+2} \to \mathtt{N}$ 項ならば，$\mathtt{rec}\ s\ t$ は型 $\mathtt{N}^{n+1} \to \mathtt{N}$ 項である．

以上の項の定義をまとめると，次のように表すこともできる．

$$\frac{}{x\colon \mathtt{N}} \qquad \frac{}{\underline{0}\colon \mathtt{N}} \qquad \frac{}{\mathtt{succ}\colon \mathtt{N} \to \mathtt{N}}$$

$$\frac{s\colon \mathtt{N}^{n+1} \to \mathtt{N} \qquad t\colon \mathtt{N}}{st\colon \mathtt{N}^n \to \mathtt{N}}\ (\to\mathtt{E}) \qquad \frac{\begin{array}{c}[x\colon \mathtt{N}]\\ \vdots \\ t\colon \mathtt{N}^n \to \mathtt{N}\end{array}}{\lambda x.t\colon \mathtt{N}^{n+1} \to \mathtt{N}}\ (\to\mathtt{I}) \qquad \frac{s\colon \mathtt{N}^n \to \mathtt{N} \qquad t\colon \mathtt{N}^{n+2} \to \mathtt{N}}{\mathtt{rec}\ s\ t\colon \mathtt{N}^{n+1} \to \mathtt{N}}\ (\mathtt{R})$$

以下，自由変数を含まない項を**閉項** (*closed*) と呼ぶ．型 $\mathtt{N}^n \to \mathtt{N}$ 閉項 t は，自然数上の n 変数関数 $[\![t]\!]\colon \mathbb{N}^n \to \mathbb{N}$ を表す．記号 $\underline{0}$ は自然数 0 を表す項であり，$\mathtt{succ}$ は後続関数を表す項である．つまり，

$$[\![\underline{0}]\!] = 0 \qquad\qquad [\![\mathtt{succ}]\!](n) = n + 1.$$

次に，st は関数適用を意味する．直感的には，$n+1$ 変数関数 $s(x_0, x_1, \ldots, x_n)$ の最初の変数に自然数 t を代入すると n 変数関数 $(x_1, \ldots, x_n) \mapsto s(t, x_1, \ldots, x_n)$ になるということである．具体的には，もし s と t が閉項ならば，

$$[\![st]\!](x_1, \ldots, x_n) = [\![s]\!]([\![t]\!], x_1, \ldots, x_n).$$

たとえば，$\mathtt{succ}\ \underline{0}$ は，関数 $\mathtt{succ}$ に 0 を入力した結果である．つまり，$\mathtt{succ}\ \underline{0}$ は，自然数 1 を表す項であり，以後はこれを $\underline{1}$ と略記する．同様に，$\mathtt{succ}(\mathtt{succ}\ \underline{0})$ は，自然数 2 を表す項であり，これを $\underline{2}$ と略記する．一般に，以下は，自然数 n を表す項である．

$$\underline{n} := \underbrace{\mathtt{succ}(\mathtt{succ}(\ldots(\mathtt{succ}}_{n \text{ times}}\ \underline{0})\ldots))$$

このとき，$\underline{n}$ のことを**数項** (*numeral*) と呼ぶ．つづいて，$\lambda x.t$ は λ **抽象** (*lambda abstraction*) と呼ばれる操作である．たとえば，項 $t(x, y)$ が複数の変数記号 x, y を含んでいる場合，x と y は固定されていると考える．しかし，$\lambda x.t(x, y)$ のように書いたときには，x は動く変数であり y は固定した変数と考える．したがって，$t = t(x, y)$ の場合，$(\lambda x.t)s$ は $t(s, y)$ を表す．より厳密には，

$$[\![\lambda x.t]\!](n, x_1, \ldots, x_k) = [\![t[\underline{n}/x]]\!](x_1, \ldots, x_k)$$

ここで，$t[s/x]$ は，項 t に出現する全ての変数記号 x に項 s を代入した結果を意味する．

最後に，$\mathtt{rec}$ は原始再帰を表す操作である．

$$\begin{cases} [\![\mathtt{rec}\ s\ t]\!](0,\bar{x}) = [\![s]\!](\bar{x}), \\ [\![\mathtt{rec}\ s\ t]\!](n+1,\bar{x}) = [\![t]\!](n,[\![\mathtt{rec}\ s\ t]\!](n,\bar{x}),\bar{x}). \end{cases}$$

これが，項が意味する自然数上の関数を考える意味論的アプローチである．

注意. 通常のラムダ計算の型推論規則に従えば，$(\to \mathtt{I})$ における $\lambda x.t$ の型は $\mathtt{N}^{n+1} \to \mathtt{N}$ ではなく $\mathtt{N} \to (\mathtt{N}^n \to \mathtt{N})$ であり，$(\to \mathtt{E})$ における s の型は $\mathtt{N}^{n+1} \to \mathtt{N}$ ではなく $\mathtt{N} \to (\mathtt{N}^n \to \mathtt{N})$ であるべきだと考えるかもしれない．しかし，システム T_0 では自然数上の関数しか扱えないので，以下の同一視

$$\mathtt{N}^{n+1} \to \mathtt{N} \quad \simeq \quad \mathtt{N} \to (\mathtt{N}^{\mathtt{n}} \to \mathtt{N})$$

を経由して，高階関数を自然数上の関数へと落としている．このように $\mathtt{X} \times \mathtt{Y} \to \mathtt{Z}$ と $\mathtt{X} \to (\mathtt{Y} \to \mathtt{Z})$ を対応付ける操作はカリー化と呼ばれる．

項に自然数上の関数を対応させる手続きは，項の記号変形操作とみなすこともできる．計算的観点からは，項の変形過程を計算の状態遷移として見るということである．具体的には，T の項に関する以下の状態遷移関係を導入する．

定義 5.25. 項上の簡約関係 $\longrightarrow$ を次によって定義する．

1.（適用） $t \longrightarrow t'$ かつ $s \longrightarrow s'$ ならば，$ts \longrightarrow t's'$ である．
2.（λ-抽象） $t \longrightarrow t'$ ならば，$\lambda x.t \longrightarrow \lambda x.t'$ である．
3.（代入） $(\lambda x.t)s \longrightarrow t[s/x]$ である．
4.（原始再帰） $\mathtt{rec}\ s\ t\ (\mathtt{succ}\ u) \longrightarrow t\ u\ (\mathtt{rec}\ s\ t\ u)$

例 5.26. 足し算 $x+y$ はたとえば次のような項で表すことができる．

$$\mathtt{add} := \lambda xy.\mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ y$$

原子再帰コマンド $\mathtt{rec}$ の意味を思い出せば，上記の式は，x, y が入力されたとき，「初期値 x, 第 k ステップで $(\lambda n.\mathtt{succ})\underline{k}$ を適用する計算を y ステップ進める」ことに対応する．ここで，$(\lambda n.\mathtt{succ})\underline{k}$ は，k に関わらず $\mathtt{succ}$ であるから，「初期値 x に対して $\mathtt{succ}$ を y 回適用する」とい

う計算が行われる．実際，$5+2$ を表す項 $\mathtt{add}\ \underline{5}\ \underline{2}$ を計算してみよう．

$$\begin{aligned}
\mathtt{add}\ \underline{5}\ \underline{2} &= \mathtt{rec}\ \underline{5}\ (\lambda n.\mathtt{succ})\ \underline{2} \\
&\longrightarrow (\lambda n.\mathtt{succ})\ \underline{1}\ (\mathtt{add}\ \underline{5}\ \underline{1}) \\
&\longrightarrow \mathtt{succ}\ (\mathtt{add}\ \underline{5}\ \underline{1}) \\
&\longrightarrow \mathtt{succ}\ ((\lambda n.\mathtt{succ})\ \underline{0}\ (\mathtt{add}\ \underline{5}\ \underline{0})) \\
&\longrightarrow \mathtt{succ}\ (\mathtt{succ}\ (\mathtt{add}\ \underline{5}\ \underline{0})) \\
&\longrightarrow \mathtt{succ}\ (\mathtt{succ}\ \underline{5}) = \underline{7}
\end{aligned}$$

ここまで来ると，簡約関係 $\longrightarrow$ を適用可能な部分が無いので，これ以上は計算は進まない．したがって，$\mathtt{add}\ \underline{5}\ \underline{2}$ の計算結果は $\underline{7}$ である．

また，以下のようにして，項 $\mathtt{add}$ の型が $\mathtt{N}^2 \to \mathtt{N}$ であることを確認できる．

$$\dfrac{\dfrac{\dfrac{[x\colon \mathtt{N}]^1 \qquad \dfrac{\mathtt{succ}\colon \mathtt{N}\to\mathtt{N}}{\lambda n.\mathtt{succ}\colon \mathtt{N}^2\to\mathtt{N}}\ (\to\mathtt{I}^3)}{\mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\colon \mathtt{N}\to\mathtt{N}}\ (\mathtt{R}) \qquad [y\colon \mathtt{N}]^2}{\mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ y\colon \mathtt{N}}\ (\to\mathtt{E})}{\lambda y.\mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ y\colon \mathtt{N}\to\mathtt{N}}\ (\to\mathtt{I}^2)}{\mathtt{add} = \lambda xy.\mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ y\colon \mathtt{N}^2\to\mathtt{N}}\ (\to\mathtt{I}^1)$$

計算のゴールは，与えられた項を，これ以上計算が進まない状態まで持ち込むことである．項 t が**正規**であるとは，$t \longrightarrow s$ となる項 s が存在しないことを意味する．つまり，正規項とは，計算の完了した項である．

さて，$\longrightarrow$ は計算を 1 ステップ進める操作だったが，有限ステップ進める操作を表す記法も導入しておくと便利である．具体的には，$\longrightarrow$ の生成する前順序を $\xrightarrow{*}$ と書く．正確には，項上の簡約関係 $\xrightarrow{*}$ は次によって与えられる．

1. （適用） $t \xrightarrow{*} t'$ かつ $s \xrightarrow{*} s'$ ならば，$ts \xrightarrow{*} t's'$ である．
2. （λ-抽象） $t \xrightarrow{*} t'$ ならば，$\lambda x.t \xrightarrow{*} \lambda x.t'$ である．
3. （代入） $(\lambda x.t)s \xrightarrow{*} t[s/x]$ である．
4. （原始再帰） $\mathtt{rec}\ s\ t\ (\mathtt{succ}\ u) \xrightarrow{*} t\ u\ (\mathtt{rec}\ s\ t\ u)$
5. （反射律） $t \xrightarrow{*} t$ である．
6. （推移律） $r \xrightarrow{*} s \xrightarrow{*} t$ ならば，$r \xrightarrow{*} t$ である．

5.5. 具体的な関数の項表現

定義だけを眺めていても，項システム T_0 の感覚がわからないと思うので，具体的な原始再帰関数を項によって表現してみることにしよう．

例 5.27. 零関数 $\lambda x_0.\lambda x_1.\ldots.\lambda x_n.\underline{0}$ は T_0 の項である．射影関数 $\lambda x_0.\lambda x_1.\ldots.\lambda x_n.x_i$ は T_0 の項である．

後続関数と原始再帰は関数合成（適用）は，最初から T_0 の言語に含まれているので，これよりあらゆる原始再帰関数は T_0 の項として表せることが分かる．実際，原始再帰関数と T_0 の項で表される関数は同一である．

$$\text{原始再帰関数} = \text{システム } T_0 \text{ の項で表現可能な関数}$$

システム T_0 は「原始再帰関数を記述するためのプログラミング言語」のようなものだと思ってもよい．とはいえ，ただの原始再帰関数を扱うだけならば，数学的に直接扱っても，項として扱ってもそんなに大差はない．項として扱うメリットが現れるのは，高階関数の原始再帰を扱う段階になってからである．しかし，それまでに原始再帰関数の項表示に慣れておいても損はないであろう．

例 5.28. 足し算 $x + y$ はたとえば次のような T_0-項を用いて表すことができる．

$$\mathtt{add} := \lambda x.(\mathtt{rec}\ x\ (\lambda n.\mathtt{succ}))$$

この項の意味としては「x を初期値として succ を繰り返し適用するプロセス」であるから，たしかに足し算を定義していそうである．これを丁寧に確認していくために，まず以下の簡約が成り立つことに注意しよう．

$$\mathtt{add}\ x\ \underline{n} \xrightarrow{*} \mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ \underline{n}.$$

簡約は逆行できないので，簡便のために以下の略記を利用する．

$$\mathtt{add}[x, n] := \mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ \underline{n}.$$

もちろん，意味を考えれば $[\![\mathtt{add}]\!](x, n) = [\![\mathtt{add}\ x\ \underline{n}]\!] = [\![\mathtt{add}[x, n]]\!]$ である．このとき，計算は次のように進んでいく．

$$\begin{aligned}
\mathtt{add}[x, 0] &\xrightarrow{*} \mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ \underline{0} \xrightarrow{*} x \\
\mathtt{add}[x, n+1] &\xrightarrow{*} \mathtt{rec}\ x\ (\lambda n.\mathtt{succ})\ \underline{n+1} \\
&\xrightarrow{*} (\lambda n.\mathtt{succ})\ \underline{n}\ (\mathtt{add}[x, n]) \xrightarrow{*} \mathtt{succ}\ (\mathtt{add}[x, n])
\end{aligned}$$

つまり，この項 add の解釈を考えれば，$[\![\mathtt{add}]\!](x, 0) = x$ かつ $[\![\mathtt{add}]\!](x, n+1) = [\![\mathtt{add}]\!](x, n) + 1$ という条件を満たすから，$[\![\mathtt{add}]\!](x, n) = x + n$ であることが導かれる．

注意. ちなみに add は型 $\mathtt{N}^2 \to \mathtt{N}$ 項であるから，任意の数項 $\underline{n}$ に対して，add $\underline{n}$ は型 $\mathtt{N} \to \mathtt{N}$ 項である．このとき，$[\![\mathtt{add}\ \underline{n}]\!](m) = [\![\mathtt{add}]\!](n, m) = n + m$ である．つまり，add $\underline{n}$ は関数 $x \mapsto n + x$ を表す項である．

ところで，型なしラムダ計算においては，チャーチ数項 $\underline{n}$ に f, a を適用すると，「n 回 f を a に適用せよ」というコマンドになるのであった．原子再帰子 rec を用いれば，このチャーチ数項の

振る舞いを模倣できる．実際，足し算のような単独の関数の繰り返しによる定義を導入する場合には，rec を用いるよりも，チャーチ数項に類似の振る舞いをする以下の反復適用子 irec を導入した方が簡単そうである．

$$\begin{cases} \mathtt{irec}\ s\ t\ \underline{0} \longrightarrow^{*} s, \\ \mathtt{irec}\ s\ t\ \underline{n+1} \longrightarrow^{*} t\ (\mathtt{irec}\ s\ t\ \underline{n}). \end{cases}$$

具体的には，t に現れない変数記号 x を取り，次のように定義すればよいように思える．

$$\mathtt{irec} := \lambda st.\mathtt{rec}\ s\ (\lambda x.t)$$

しかし，残念ながら，話はそう単純ではない．システム T_0 では型 N 項に対する λ-抽象しか行うことができないが，t は一般的には型 N 項ではないためである．その場合，irec は T_0-項ではない．とはいえ，$\mathtt{rec}\ s\ (\lambda x.t)$ 自体は T_0-項であるという点に注目しよう．よって，irec という項を考えるのではなく，項の略記として $\mathtt{irec}\ s\ t$ という記法を用いるのであれば，特に問題は起きない．

$$\mathtt{irec}\ s\ t := \mathtt{rec}\ s\ (\lambda x.t)$$

以後はしばしばこの略記を用いるが，irec 自体は T_0-項ではないという点は常に頭の片隅に置いておこう．こうすれば，たとえば，$\mathtt{add} = \lambda xy.(\mathtt{irec}\ x\ \mathtt{succ}\ y)$ が成立する．チャーチ数項と適用順が少し異なるが，チャーチ数項の節で学んだように，これを利用すると，掛け算，指数などを容易に定義できる．

例 5.29. 掛け算 $x \cdot y$ はたとえば次のような T_0-項で表すことができる．

$$\mathtt{mult} := \lambda x.(\mathtt{irec}\ \underline{0}\ (\mathtt{add}\ x))$$

この項の意味としては「$\underline{0}$ を初期値として $\mathtt{add}\ x$ を繰り返し適用するプロセス」である．項 $\mathtt{add}\ x$ が $y \mapsto x + y$ を意味していたことから，たしかに掛け算を定義していそうである．先程と同様に，まず以下の簡約が成り立つことに注意しよう．

$$\mathtt{mult}\ x\ \underline{n} \longrightarrow^{*} \mathtt{irec}\ \underline{0}\ (\mathtt{add}\ x)\ \underline{n}.$$

簡約は逆行できないので，簡便のために以下の略記を利用する．

$$\mathtt{mult}[x, n] := \mathtt{irec}\ \underline{0}\ (\mathtt{add}\ x)\ \underline{n}.$$

もちろん，意味を考えれば $[\![\mathtt{mult}]\!](x, n) = [\![\mathtt{mult}[x, n]]\!]$ である．このとき，計算は次のように進んでいく．

$$\begin{aligned} \mathtt{mult}[x, 0] &\longrightarrow^{*} \mathtt{irec}\ \underline{0}\ (\mathtt{add}\ x)\ \underline{0} \longrightarrow^{*} \underline{0} \\ \mathtt{mult}[x, n+1] &\longrightarrow^{*} \mathtt{irec}\ \underline{0}\ (\mathtt{add}\ x)\ \underline{n+1} \longrightarrow^{*} \mathtt{add}\ x\ (\mathtt{mult}\ [x, n]) \end{aligned}$$

つまり，項 $\mathtt{mult}$ の解釈を考えれば，$[\![\mathtt{mult}]\!](x,0)=0$ かつ $[\![\mathtt{mult}]\!](x,n+1)=x+[\![\mathtt{mult}]\!](x,n)$ という条件を満たすから，$[\![\mathtt{mult}]\!](x,n)=x\cdot n$ であることが導かれる．

例 5.30. 指数関数 x^y はたとえば次のような T_0-項で表すことができる．

$$\mathtt{exp} := \lambda x.(\mathtt{irec}\ \underline{1}\ (\mathtt{mult}\ x))$$

この項の意味としては「$\underline{1}$ を初期値として $\mathtt{mult}\ x$ を繰り返し適用するプロセス」である．項 $\mathtt{mult}\ x$ が $y \mapsto x\cdot y$ を定義していたことから，たしかに指数関数を定義していそうである．先程と同様に，まず以下の簡約が成り立つことに注意しよう．

$$\mathtt{exp}\ x\ \underline{n} \longrightarrow^{*} \mathtt{irec}\ \underline{1}\ (\mathtt{exp}\ x)\ \underline{n}.$$

簡約は逆行できないので，簡便のために以下の略記を利用する．

$$\mathtt{exp}[x,n] := \mathtt{irec}\ \underline{1}\ (\mathtt{exp}\ x)\ \underline{n}.$$

このとき，以下の計算遷移が行われる．

$$\begin{aligned}\mathtt{exp}[x,0] &\longrightarrow^{*} \mathtt{irec}\ \underline{1}\ (\mathtt{mult}\ x)\ \underline{0} \longrightarrow^{*} \underline{1}\\ \mathtt{exp}[x,n+1] &\longrightarrow^{*} \mathtt{irec}\ \underline{1}\ (\mathtt{mult}\ x)\ \underline{n+1} \longrightarrow^{*} \mathtt{mult}\ x\ (\mathtt{exp}[x,n])\end{aligned}$$

つまり，項 $\mathtt{exp}$ の解釈を考えれば，$[\![\mathtt{exp}]\!](x,0)=1$ かつ $[\![\mathtt{exp}]\!](x,n+1)=x\cdot[\![\mathtt{exp}]\!](x,n)$ という条件を満たすから，$[\![\mathtt{exp}]\!](x,n)=x^n$ であることが導かれる．

例 5.31. T_0-項 $\mathtt{p}$ が関数 $p\colon \mathtt{N}^{k+1}\to\mathtt{N}$ を表現していると仮定する．このとき，総和 $\sum_{z=0}^{y-1} p(\bar{x},z)$ はたとえば次のような T_0-項で表すことができる．

$$\mathtt{sum\ p} := \lambda x_1,\ldots,x_k.\mathtt{rec}\ \underline{0}\ (\lambda y.\mathtt{add}\ (\mathtt{p}\ x_1\ \ldots\ x_k\ y))$$

簡単のために $k=1$ の場合のみを考える．この場合，この項の意味としては，「$\underline{0}$ を初期値として，第 y ステップで $\mathtt{add}\ (\mathtt{p}\ x\ y)$ を実行するプロセス」である．ここで初めて，同じ関数の繰り返し適用ではなく，ステップ毎に異なる関数を適用する手続きが現れた．しかし，理屈はこれまでと同様である．簡便のために以下の略記を利用する．

$$\mathtt{sum\ p}\ [x,n] := \mathtt{rec}\ \underline{0}\ (\lambda y.\mathtt{add}\ (\mathtt{p}\ x\ y))\ \underline{n}.$$

基底ステップは自明なので，後続ステップのみ書くと，次の計算遷移が行われる．

$$\begin{aligned}\mathtt{sum\ p}\ [x,n+1] &\longrightarrow^{*} \mathtt{rec}\ \underline{0}\ (\lambda y.\mathtt{add}\ (\mathtt{p}\ x\ y))\ \underline{n+1}\\ &\longrightarrow^{*} (\lambda y.\mathtt{add}\ (\mathtt{p}\ x\ y))\ \underline{n}\ (\mathtt{sum\ p}\ [x,n]) \longrightarrow^{*} \mathtt{add}\ (\mathtt{p}\ x\ \underline{n})\ (\mathtt{sum\ p}\ [x,n])\end{aligned}$$

項 sum p の解釈を考えれば，$[\![\texttt{sum p}]\!](x,0)=0$ かつ $[\![\texttt{sum p}]\!](x,n+1)=p(x,n)+[\![\texttt{sum p}]\!](x,n)$ という条件を満たすから，$[\![\texttt{sum p}]\!](x,n)=\sum_{i=0}^{n-1}p(x,i)$ であることが導かれる．

例 5.32. T_0-項 p が関数 $p\colon \mathtt{N}^{k+1}\to\mathtt{N}$ を表現していると仮定する．このとき，総乗 $\prod_{z=0}^{y-1}p(\bar{x},z)$ はたとえば次のような T_0-項で表すことができる．

$$\texttt{prod p} := \lambda x_1,\ldots,x_k.\texttt{rec}\ \underline{1}\ (\lambda y.\texttt{mult}\ (\texttt{p}\ x_1\ \ldots\ x_k\ y))$$

簡単のために $k=1$ の場合のみ述べると，この項の意味としては，「$\underline{1}$ を初期値として，第 y ステップで mult (p x y) を実行するプロセス」である．これまでと同じような略記を用いると，基底ステップは自明なので，後続ステップのみ書くと，次の計算遷移が行われる．

$$\begin{aligned}\texttt{prod p}\ [x,n+1] &\xrightarrow{*} \texttt{rec}\ \underline{0}\ (\lambda y.\texttt{mult}\ (\texttt{p}\ x\ y))\ \underline{n+1}\\ &\xrightarrow{*} (\lambda y.\texttt{mult}\ (\texttt{p}\ x\ y))\ \underline{n}\ (\texttt{prod p}\ [x,n]) \xrightarrow{*} \texttt{mult}\ (\texttt{p}\ x\ \underline{n})\ (\texttt{prod p}\ [x,n])\end{aligned}$$

項 prod p を解釈すると，$[\![\texttt{prod p}]\!](x,0)=0$ かつ $[\![\texttt{prod p}]\!](x,n+1)=p(x,n)\cdot[\![\texttt{prod p}]\!](x,n)$ という条件を満たすから，$[\![\texttt{prod p}]\!](x,n)=\prod_{i=0}^{n-1}p(x,i)$ であることが導かれる．

演習問題 5.33. 階乗を T_0 の項を用いて表わせ．

例 5.34. 直前値 $x\dot{-}1$ は次の T_0-項で表すことができる．

$$\texttt{pred} := \texttt{rec}\ \underline{0}\ (\lambda xy.x)$$

これは再帰としての意味を考えずに，定義に従って，計算手続きを分析するのがよい．基底ステップは自明なので，後続ステップのみ書くと，次の計算遷移が行われる．

$$\texttt{pred}\ \underline{n+1} \xrightarrow{*} (\lambda xy.x)\ \underline{n}\ (\texttt{pred}\ \underline{n}) \xrightarrow{*} \underline{n}$$

つまり，$[\![\texttt{pred}]\!](0)=0$ かつ $[\![\texttt{pred}]\!](n+1)=n$ が成立する．

例 5.35. 部分的減算 $x\dot{-}y$ は次の T_0-項で表すことができる．

$$\texttt{monus} := \lambda x.(\texttt{irec}\ x\ \texttt{pred})$$

この項を説明する前に，ひとつ余談であるが，演算 $x\dot{-}n:=\max\{x-n,0\}$ にはマイナス (minus) ならぬ**モーナス** (*monus*) なる名前が付いているらしい，ということを最近知った．これがこの項の名前の由来である．この項の意味としては，「x を初期値として pred を繰り返し適用していくプ

ロセス」である．基底ステップは自明なので，後続ステップのみ書くと，次の計算遷移が行われる．

$$\mathtt{monus}\ x\ \underline{n+1} \xrightarrow{*} \mathtt{pred}\ (\mathtt{monus}\ x\ \underline{n})$$

つまり，$[\![\mathtt{monus}]\!](x,n) = [\![\mathtt{pred}]\!]^n(x) = x \dot{-} n$ が成立する．

例 5.36. 符号関数 sgn は次の T_0-項で表すことができる．

$$\mathtt{sgn} := \mathtt{irec}\ \underline{0}\ (\lambda x.\underline{1})$$

この項も，再帰としての意味を考えずに，定義に従って，計算手続きを分析するのがよい．このとき，次の計算遷移が行われる．

$$\begin{cases} \mathtt{sgn}\ \underline{0} \xrightarrow{*} \underline{0} \\ \mathtt{sgn}\ \underline{n+1} \xrightarrow{*} (\lambda x.\underline{1})\ (\mathtt{sgn}\ \underline{n}) \xrightarrow{*} \underline{1} \end{cases}$$

つまり，$n = 0$ ならば $[\![\mathtt{sgn}]\!](n) = 0$ であり，$n > 0$ ならば $[\![\mathtt{sgn}]\!](n) = 1$ となる．また，

$$\mathtt{neg\_sgn} := \lambda x.(\mathtt{monus}\ \underline{1}\ (\mathtt{sgn}\ x))$$

として定義する．この解釈を考えると，$[\![\mathtt{neg\_sgn}]\!](x) = 1 \dot{-} [\![\mathtt{sgn}]\!](x)$ である．この項については，$n = 0$ ならば $[\![\mathtt{neg\_sgn}]\!](n) = 0$ であり，$n > 0$ ならば $[\![\mathtt{neg\_sgn}]\!](n) = 1$ となる．この関数は，場合分けのベースとして用いられる．

例 5.37 (場合分け)**.** 自然数 i, x, y に対して，$i = 0$ ならば x を出力し，$i > 0$ ならば y を出力するという場合分けは，以下の T_0-項として表すことができる．

$$\mathtt{cond} := \lambda ixy.\Big(\mathtt{add}\ \big(\mathtt{mult}\ x\ (\mathtt{neg\_sgn}\ i)\big)\ \big(\mathtt{mult}\ y\ (\mathtt{sgn}\ i)\big)\Big)$$

この項が場合分けを表していることは，実際に，$\mathtt{cond}\ i\ x\ y$ を計算してみればよい．

$$\begin{aligned} \mathtt{cond}\ \underline{0}\ x\ y &\xrightarrow{*} \mathtt{add}\ \big(\mathtt{mult}\ x\ (\mathtt{neg\_sgn}\ \underline{0})\big)\ \big(\mathtt{mult}\ y\ (\mathtt{sgn}\ \underline{0})\big) \\ &\xrightarrow{*} \mathtt{add}\ (\mathtt{mult}\ x\ \underline{1})\ (\mathtt{mult}\ y\ \underline{0}) \\ &\xrightarrow{*} \mathtt{add}\ x\ \underline{0} \xrightarrow{*} x \end{aligned}$$

さらに，y が数項 $\underline{n}$ の場合には，

$$\begin{aligned} \mathtt{cond}\ \underline{1}\ x\ y &\xrightarrow{*} \mathtt{add}\ \big(\mathtt{mult}\ x\ (\mathtt{neg\_sgn}\ \underline{1})\big)\ \big(\mathtt{mult}\ y\ (\mathtt{sgn}\ \underline{1})\big) \\ &\xrightarrow{*} \mathtt{add}\ (\mathtt{mult}\ x\ \underline{0})\ (\mathtt{mult}\ y\ \underline{1}) \\ &\xrightarrow{*} \mathtt{add}\ \underline{0}\ y \xrightarrow{*} y \end{aligned}$$

よって，この項 cond は，入力 i, x, y に対して，$i = 0$ ならば x を返し，$i > 0$ ならば y を返す関数を表す．より一般的に，入力 x に対して，$p(x) = 0$ ならば $g(x)$ を出力し，$p(x) > 0$ ならば $h(x)$ を出力する関数を作りたいとしよう．このときは，$\lambda x.(\mathtt{cond}\ (p\ x)\ (g\ x)\ (h\ x))$ を考えればよい．

このように，システム T_0 において，条件分岐を作ることができるため，T_0 の項だけを用いて，プログラミングのかなりの部分は達成できる．実際，システム T_0, つまり原始再帰関数のプログラミング言語的な特徴付けを与えられることができる．基本的なブール演算，場合分け，for ループを利用可能なプログラミングである．

演習問題 5.38. フィボナッチ数列を T_0 の項を用いて表わせ．

演習問題 5.39. 素数判定アルゴリズムを T_0 の項を用いて表わせ．

§6. 高階原始再帰

6.1. ゲーデルのシステム T

ゲーデルのシステム T (*Gödel's system* T) とは，高階原始再帰汎関数を厳密に記述する型付き項のシステムである．自然数の型を N と書く．N はシステム T の型であり，σ, τ が T の型ならば $\sigma \to \tau$ も T の型である．

$$\sigma ::= \mathtt{N} \mid \sigma \to \sigma$$

型 N のことをしばしば 0 と書き，型 $\mathtt{n} \to 0$ のことを $\mathtt{n} + 1$ と書く．以後，型 $\sigma \to (\tau \to \rho)$ を $\sigma \to \tau \to \rho$ を略記する．また，$\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n \to \tau$ のことをしばしば $\sigma_0 \times \cdots \times \sigma_n \to \tau$ と略記する．また，$\sigma \times \cdots \times \sigma \to \tau$ のことを $\sigma^n \to \tau$ のように略記する．

定義 6.1. システム T の項は，以下のように帰納的に定義される．

1. 型 σ 毎に，可算個の変数 x^σ が用意されており，これは型 σ 項である．
2. 記号 $\underline{0}$ は型 N 項であり，記号 succ は型 $\mathtt{N} \to \mathtt{N}$ 項である．
3. s が型 $\sigma \to \tau$ 項で t が型 σ 項ならば，st は型 τ 項である．
4. x^σ が型 σ 変数であり t が型 τ 項ならば，$\lambda x^\sigma.t$ は型 $\sigma \to \tau$ 項である．
5. s が型 σ 項で t が型 $\mathtt{N} \to \sigma \to \sigma$ 項ならば，$\mathtt{rec}^\sigma\ s\ t$ は型 $\mathtt{N} \to \sigma$ 項である．

以上の型付き項の定義をまとめると，次のように表すこともできる．

$$\frac{}{x^\sigma : \sigma} \qquad \frac{}{\underline{0} : \mathtt{N}} \qquad \frac{}{\mathtt{succ} : \mathtt{N} \to \mathtt{N}}$$

$$\frac{s\colon \sigma \to \tau \qquad t\colon \sigma}{st\colon \tau}\ (\to\mathtt{E}) \qquad \frac{\begin{array}{c}[x^\sigma\colon \sigma]\\ \vdots \\ t\colon \tau\end{array}}{\lambda x^\sigma.t\colon \sigma \to \tau}\ (\to\mathtt{I}) \qquad \frac{s\colon \sigma \qquad t\colon \mathtt{N} \to \sigma \to \sigma}{\mathtt{rec}^\sigma\ s\ t\colon \mathtt{N} \to \sigma}\ (\mathtt{R})$$

ここで，$(\to\mathtt{E})$, $(\to\mathtt{I})$, $(\mathtt{R})$ などは各定義に名前を付けているだけである．変数記号 x^σ の肩に型 σ を付けているが，この σ は省略することが多い．また，変数記号を最初に導入するときだけ x^σ と型を明示し，その後の出現では x のように型を省略することもある．λ 抽象に対しても，同様の省略を行う．また，$\mathtt{rec}^\sigma$ のこともしばしば $\mathtt{rec}$ と略記する．

システム T_0 のときと同様に，記号 $\underline{0}$ は自然数 0 を表す項であり，$\mathtt{succ}$ は後続関数を表す項である．同じく，st は関数適用，$\lambda x.t$ は λ 抽象を表し，$\mathtt{rec}$ は（高階）原始再帰を表す操作である．したがって，システム T の λ 抽象と原始再帰を $\lambda x^{\mathtt{N}}.t$ と $\mathtt{rec}_{\mathtt{N}^n \to \mathtt{N}}$ に制限したものが，システム T_0 である．T_0 のときと同様に，T の項に関する以下の状態遷移関係を導入する．

定義 6.2. 項上の簡約関係 $\xrightarrow{*}$ を次によって定義する．

1. （反射律） $t \xrightarrow{*} t$ である．
2. （推移律） $r \xrightarrow{*} s \xrightarrow{*} t$ ならば，$r \xrightarrow{*} t$ である．
3. （適用） $t \xrightarrow{*} t'$ かつ $s \xrightarrow{*} s'$ ならば，$ts \xrightarrow{*} t's'$ である．
4. （λ-抽象） $t \xrightarrow{*} t'$ ならば，$\lambda x.t \xrightarrow{*} \lambda x.t'$ である．
5. （代入） $(\lambda x.t)s \xrightarrow{*} t[s/x]$ である．
6. （原始再帰） $\mathtt{rec}\ s\ t\ (\mathtt{succ}\ u) \xrightarrow{*} t\ u\ (\mathtt{rec}\ s\ t\ u)$

例 6.3. 例 5.31 においては総和を表す T_0-項 $\mathtt{sum\ p}$ を構成したが，システム T では，この $\mathtt{p}$ も変数にした項 $\mathtt{sum}$ を定義することができる．

$$\mathtt{sum} := \lambda p^{\mathtt{N}^2 \to \mathtt{N}}.\lambda x.\mathtt{rec}^{\mathtt{N}}\ \underline{0}\ (\lambda y.\mathtt{add}\ (p\ x\ y))$$

ここで，$p^{\mathtt{N}^2 \to \mathtt{N}}$ という記法は，p が型 $\mathtt{N}^2 \to \mathtt{N}$ 変数であることを示唆していることに注意する．このとき，$\mathtt{sum}\ p\ x\ n$ は $\sum_{i=0}^{n-1} p(x,i)$ の計算を行う．

6.2. アッカーマン関数

ゲーデルのシステム T は，システム T_0 と比較すると，どれくらい強力だろうか．これを確認するために，和や積や指数の定義に戻ろう．

$$\begin{aligned}\mathtt{add} &= \lambda x.(\mathtt{irec}\ x\ \mathtt{succ})\\ \mathtt{mult} &= \lambda x.(\mathtt{irec}\ \underline{0}\ (\mathtt{add}\ x))\\ \mathtt{exp} &= \lambda x.(\mathtt{irec}\ \underline{1}\ (\mathtt{mult}\ x))\end{aligned}$$

なんとなく共通のパターンがあることは，予想が付くだろう．$\mathtt{Ack}_0 = \mathtt{mult}$ として，$\mathtt{Ack}_{n+1}$ を次のように定義する．

$$\mathtt{Ack}_{n+1} \;:=\; \lambda x.\big(\mathtt{irec}\ \underline{1}\ (\mathtt{Ack}_n\ x)\big)$$

もちろん $\mathtt{Ack}_1 = \mathtt{exp}$ であり，$\mathtt{Ack}_2$ はテトレーションである．クヌースの矢印表記を用いれば，一般に $\mathtt{Ack}_n\ x\ y$ は $x \uparrow^n y$ と同じものである．実際，

$$\begin{aligned}[\![\mathtt{Ack}_{n+1}\ x\ y]\!] &= [\![\mathtt{irec}\ \underline{1}\ (\mathtt{Ack}_n\ x)\ y]\!] \\ &= [\![\mathtt{irec}]\!](1, x \uparrow^n -, y) = \underbrace{x \uparrow^n x \uparrow^n x \uparrow^n \ldots \uparrow^n x \uparrow^n}_{y \text{ times}} 1.\end{aligned}$$

さて，ダフィット・ヒルベルト (David Hilbert; 1862–1943) が 1925 年に言及したように，拡張された原始再帰法を用いれば，与えられた 2 項演算 $*$ の累積を行う汎関数を定義できる．つまり，上記の関数構成は，次のような項を用いて表すことができる．

$$\mathtt{Ack\_iter} \;:=\; \lambda f x.\big(\mathtt{irec}\ \underline{1}\ (f\ x)\big)$$

ここで，f の型は $\mathtt{N}^2 \to \mathtt{N}$ であることに注意する．この辺りになると $\mathtt{Ack\_iter}$ の型がぱっと見では分かりづらい，形式的に型を推論してみよう．

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[f\colon \mathtt{N}\to\mathtt{N}\to\mathtt{N}]^1 \qquad [x\colon \mathtt{N}]^2}{f\ x\colon \mathtt{N}\to\mathtt{N}}\ (\to\mathrm{E}) \qquad \underline{1}\colon \mathtt{N}}{\lambda n.(f\ x)\colon \mathtt{N}\to\mathtt{N}\to\mathtt{N}}\ (\to\mathrm{I})}{\mathtt{irec}\ \underline{1}\ (f\ x)\ =\ \mathtt{rec}\ \underline{1}\ (\lambda n.(f\ x))\colon \mathtt{N}\to\mathtt{N}}\ (\mathrm{R})}{\lambda x.(\mathtt{irec}\ \underline{1}\ (f\ x))\colon \mathtt{N}\to\mathtt{N}\to\mathtt{N}}\ (\to\mathrm{I})^2}{\mathtt{Ack\_iter} = \lambda f x.(\mathtt{irec}\ \underline{1}\ (f\ x))\colon (\mathtt{N}\to\mathtt{N}\to\mathtt{N})\to\mathtt{N}\to\mathtt{N}\to\mathtt{N}}\ (\to\mathrm{I})^1$$

したがって，$\mathtt{Ack\_iter}$ の型は $(\mathtt{N}\to\mathtt{N}\to\mathtt{N})\to\mathtt{N}\to\mathtt{N}\to\mathtt{N}$ なのであるが，これでは何なのか分かりにくいので，省略記法を使ってまとめる．$\mathtt{N}\to\mathtt{N}\to\mathtt{N}$ が $\mathtt{N}^2\to\mathtt{N}$ と略記できることに注意すれば，$(\mathtt{N}^2\to\mathtt{N})\to\mathtt{N}^2\to\mathtt{N}$ となる．つまり，2 項演算を受け取って 2 項演算を返す関数である．

$$[\![\mathtt{Ack\_iter}\ \star]\!](x, y) = [\![\mathtt{irec}\ \underline{1}\ (\star\ x)\ y]\!] = \underbrace{x \star x \star x \star \cdots \star x \star}_{y \text{ times}} 1.$$

ここで，$\star\ x\ z$ を $x \star z$ のように書いていることに注意する．この高階関数を用いて，1925 年，ヒルベルトは矢印演算の階層を以下のように高階関数を用いて表した．

$$\begin{aligned}[\![\mathtt{Ack}_0\ a\ b]\!] &= a \cdot b \\ [\![\mathtt{Ack}_1\ a\ b]\!] &= [\![\mathtt{Ack\_iter}\ \mathtt{Ack}_0\ a\ b]\!] = a^b \\ [\![\mathtt{Ack}_2\ a\ b]\!] &= [\![\mathtt{Ack\_iter}\ \mathtt{Ack}_1\ a\ b]\!] = a \uparrow\uparrow b \\ &\vdots \\ [\![\mathtt{Ack}_{n+1}\ a\ b]\!] &= [\![\mathtt{Ack\_iter}\ \mathtt{Ack}_n\ a\ b]\!] = a \uparrow^{n+1} b\end{aligned}$$

ヒルベルトが注目したことは，この演算階層を，型 2 汎関数 Ack_iter を用いたある種の原始再帰によって定義できるということであった具体的には，関数 Ack を次のように定義しよう．

$$\texttt{Ack} \;:=\; \texttt{irec mult Ack\_iter}$$

この関数の型も分かりにくいが，次のように型を推論できる．

$$\dfrac{\texttt{mult}\colon \mathtt{N}^2 \to \mathtt{N} \qquad \dfrac{\texttt{Ack\_iter}\colon (\mathtt{N}^2 \to \mathtt{N}) \to \mathtt{N}^2 \to \mathtt{N}}{\lambda z.\texttt{Ack\_iter}\colon \mathtt{N} \to (\mathtt{N}^2 \to \mathtt{N}) \to \mathtt{N}^2 \to \mathtt{N}}\ (\to\mathrm{I})}{\texttt{Ack} = \texttt{rec mult}\ \lambda z.\texttt{Ack\_iter}\colon \mathtt{N} \to \mathtt{N}^2 \to \mathtt{N}}\ (\mathrm{R})$$

そういうわけで，型は $\mathtt{N} \to \mathtt{N}^2 \to \mathtt{N}$ であるが，前半をまとめてしまえば，$\mathtt{N}^3 \to \mathtt{N}$ ということでもある．つまり，Ack の型は $\mathtt{N}^3 \to \mathtt{N}$ である．このとき，$\texttt{Ack}\ \underline{0} = \texttt{mult}$ であり，後続ステップに関しては，

$$\begin{aligned}\texttt{Ack}\ \underline{n+1} &\xrightarrow{*} \texttt{Ack\_iter}\ (\texttt{Ack}\ \underline{n}) = (\lambda f x.\big(\texttt{irec}\ \underline{1}\ (f\ x)\big)\ (\texttt{Ack}\ \underline{n}) \\ &\xrightarrow{*} \texttt{irec}\ \underline{1}\ (\texttt{Ack}\ \underline{n}\ x)\end{aligned}$$

したがって，$\texttt{Ack}\ \underline{n} = \texttt{Ack}_n$ である．この関数 Ack はヒルベルトとアッカーマンによるオリジナルの**アッカーマン関数** (*Ackermann function*) の定義（の添字を少しずらしたもの）である．つまり，オリジナルのアッカーマン関数は，後続，加法，乗法，指数，テトレーション，クヌースの矢印表記の各レベル，という 2 項演算の階層を与えるものである．現在アッカーマン関数としてよく知られているものは，後の時代に改変が加えられたものであるが，それよりも，オリジナルのアッカーマン関数 Ack の方が遥かに自然な定義である．

さて，通常の原始再帰関数の定義においては，型 $\sigma = \mathtt{N}^n$ に対する原始再帰しか用いられていなかったのだが，アッカーマン関数 Ack の定義において，はじめてこの枠外の原始再帰が用いられている．2 項演算の型 $\sigma = \mathtt{N}^2 \to \mathtt{N}$ に対する原始再帰である．つまり，ヒルベルトは，

アッカーマン関数 Ack は型 2 汎関数を用いた原始再帰法によって定義できる

ということを示したのである．そして，ヒルベルトは，関数 Ack は型 1 汎関数しか用いない原始再帰法によっては定義できないと予想した．つまり，現代的な用語を用いれば，Ack は原始再帰的関数ではないと予想したのである．ヒルベルトの目的は，「高い型を持つ汎関数を利用すれば，低い型を持つ汎関数しか利用しないよりも，真に多くの $\mathbb{N}$ 上の関数を定義できる」ということを示すことであった．このようにして，再帰的定義のために必要な汎関数の型のランクによって，$\mathbb{N}$ 上の関数たちを分類しようとしたのである．1927 年頃に，ヒルベルトの問題に解決を与え，Ack が原始再帰的でないことを示したのが，ヒルベルトの 2 人の弟子，ヴィルヘルム・アッカーマン (Wilhelm Ackermann; 1896–1962) とガブリエル・スーダン (Gabriel Sudan; 1899–1977) であり，このために A はアッカーマン関数と呼ばれることとなった．

定理 6.4 (スーダン, アッカーマン)**.** アッカーマン関数 $\mathtt{Ack}\colon \mathbb{N}^3 \to \mathbb{N}$ は原始再帰的ではない.

そういうわけで，アッカーマン関数を原始再帰によって定義するためには，必ず高階関数を使わなければならない．自然数上の関数を定義するために，高階関数を経由しなければならない，というのが，高階関数の計算論の興味深いところである．

注意. 現在，アッカーマン関数として巷で最も広く知られているものは，上で定義した関数 $\mathtt{Ack}$ ではなく，ロザ・ペーター (Rózsa Péter; 1905–1977) による別の関数である．ペーターは，アッカーマン関数の定義から高階汎関数の概念を除去し，その定義を単純化した．ペーターによるアッカーマン関数 $A\colon \mathbb{N}^2 \to \mathbb{N}$ の定義は以下によって与えられる．

$$\begin{aligned} A(0,x) &= x+1, \\ A(n+1,0) &= A(n,1), \\ A(n+1,x+1) &= A(n,A(n+1,x)). \end{aligned}$$

ペーターのアッカーマン関数が元のアッカーマン関数と異なるということについては，それなりに認知されており，ペーターの関数 A は 2 変数関数であり，元の関数 $\mathtt{Ack}$ は 3 変数関数であるという点が違いとして強調されることがあるようである．ただ，アッカーマン関数の定義が 2 変数か 3 変数かという点は，とてつもなくどうでもいいことである．真に強調すべき点は，ヒルベルトのアッカーマン関数（アッカーマンが原論文で用いた関数）は「**型 2 原始再帰**」によって定義され，ペーターのアッカーマン関数は「**型 1 多重再帰**」によって定義される，という点であろう．

6.3. 急増加階層と順序数

この発想を推し進めて，さらに複雑な関数を高階原始再帰によって定義していこう．ただし，アッカーマン関数は 2 項演算の累積の階層であったが，2 項演算だと議論が複雑になるので，ここからは 1 変数版の関数累積を考えたい．つまり，以下のような関数階層を考える．

$$\begin{aligned} \mathbf{f}_0(x) &= x+1, \\ \mathbf{f}_{n+1}(x) &= \underbrace{\mathbf{f}_n \circ \mathbf{f}_n \circ \cdots \circ \mathbf{f}_n}_{x \text{ 個}}(x), \\ \mathbf{f}_\omega(x) &= \mathbf{f}_x(x). \end{aligned}$$

この発想は，**急増加階層** (*fast-growing hierarchy*) というものに拡張される．たとえば，各 $n \leqslant \omega$ について，以下のような関数 $\mathbf{f}_{\omega+n}\colon \mathbb{N} \to \mathbb{N}$ を定義できることは明らかであろう．

$$\begin{aligned} \mathbf{f}_{\omega+n+1}(x) &= \underbrace{\mathbf{f}_{\omega+n} \circ \mathbf{f}_{\omega+n} \circ \cdots \circ \mathbf{f}_{\omega+n}}_{x \text{ 個}}(x), \\ \mathbf{f}_{\omega+\omega}(x) &= \mathbf{f}_{\omega+x}(x). \end{aligned}$$

これを繰り返して，可算順序数 α に対して，急増加関数 $\mathbf{f}_\alpha\colon \mathbb{N} \to \mathbb{N}$ を定義していこう，というものが急増加階層の発想である．ただし，これはこの言葉通りに都合よく行くわけではない．実際

には，可算順序数 α ではなく**可算整礎木** $T \subseteq \mathbb{N}^*$ に対して，急増加関数 $\mathbf{f}_T \colon \mathbb{N} \to \mathbb{N}$ を定義できるだけである，という点に注意しておこう．たとえば，十分小さい順序数 α, たとえば $\alpha < \varepsilon_0$ については，基本列 $(\alpha[n])_{n\in\mathbb{N}}$ というものを割り当てる標準的な方法があり，ここから可算整礎木 $T(\alpha)$ が定義されるから，$\mathbf{f}_\alpha = \mathbf{f}_{T(\alpha)}$ の標準的な定義がある．しかし，順序数 α に対して，急増加関数 $\mathbf{f}_\alpha$ が定義されるというわけではない．

■順序数 ε_0 の表現: この議論をもう少し丁寧に進めていこう．いま，言語 $\mathcal{L}_\varepsilon = \{0, +, \mathtt{exp}\}$ は定数記号 0, 2 変数関数 $\#$ および 1 変数関数記号 exp からなるとする．$\mathcal{L}_\varepsilon$-項は，以下のように帰納的に定義される．

- 定数記号 0 は $\mathcal{L}_\varepsilon$-項である．
- s, t が $\mathcal{L}_\varepsilon$-項ならば，$s\#t$ および $\mathtt{exp}(t)$ は $\mathcal{L}_\varepsilon$-項である．

項 $s\#t$ のことは s と t の自然和 (natural sum) と呼び，$\mathtt{exp}(t)$ のことはしばしば ω^t と書く．$\mathcal{L}_\varepsilon$-項の間の順序 $\leqslant$ を以下のように定義する，

$$x \leqslant x\#y \qquad x\#y \leqslant y\#x \qquad x \leqslant \mathtt{exp}(x)$$

$$\frac{x \leqslant y \qquad x' \leqslant y'}{x\#x' \leqslant y\#y'} \qquad \frac{x_1, \ldots, x_n < y}{\mathtt{exp}(x_1)\# \ldots \#\mathtt{exp}(x_n) < \mathtt{exp}(y)}$$

[TODO]

■順序数 ε_0 の別表現: 集合 D 上の**多重集合** (*multiset*) とは，関数 $A \colon D \to \mathbb{N}$ である．ここで，$A(x) = n$ は，この多重集合が元 x を n 個持つことを意味する．本稿では有限多重集合のみを考える．つまり，$A(x) \neq 0$ となる $x \in D$ は有限個しか存在しない．以下，元 a を k 個含むこと $a \otimes k$ と書くことにすれば，有限多重集合は，

$$\{a_0 \otimes k_0, a_1 \otimes k_1, \ldots, a_\ell \otimes k_\ell\}$$

のように表すことができる．ここで，各 k_i は自然数である．以下，$\mathrm{Mul}(D)$ によって，D 上の有限多重集合全体を表す．多重集合の和は，元の数を足し合わせることによって定義される．

集合 D 上に前順序 $\leqslant_D$ が与えられているとき，以下の条件

$$A \leqslant'_D A \cup B \qquad \frac{A \leqslant'_D B \qquad A' \leqslant'_D B'}{A \cup A' \leqslant'_D B \cup B'} \qquad \frac{a_1, \ldots, a_n <_D b}{\{a_1, \ldots, a_n\} <'_D \{b\}}$$

によって得られる関係 $\leqslant'_D$ から，$\mathrm{Mul}(D)$ 上の前順序を生成できる．これを D 上の**多重集合順序** (*multiset ordering*) と呼ぶ．D 上に全順序が与えられているならば，有限多重集合は必ず大きい順に並べるとする．つまり，$a_0 > a_1 > \cdots > a_\ell$ を仮定する．

もし D が順序数の集合ならば，D 上の多重集合順序は，次のように特徴づけることができる．

$$\{\alpha_0 \otimes k_0, \ldots, \alpha_\ell \otimes k_\ell\} \leqslant'_D \{\beta_0 \otimes j_0, \ldots, \beta_\ell \otimes j_\ell\}$$
$$\iff \omega^{\alpha_0} \cdot k_0 + \cdots + \omega^{\alpha_\ell} \cdot k_\ell \leqslant \omega^{\beta_0} \cdot j_0 + \cdots + \omega^{\beta_\ell} \cdot j_\ell.$$

よって，D_0 を自然数の順序 ω とすれば，$D_1 := \mathrm{Mul}(D_0)$ の順序型は ω^ω, $D_2 := \mathrm{Mul}(D_1)$ の順序型は ω^{ω^ω} といったことが成立する．$D_{n+1} := \mathrm{Mul}(D_n)$ とし，$E := \bigcup_{n\in\mathbb{N}} D_n$ の順序型を ε_0 と書く．

$$\varepsilon_0 = \sup_{n<\omega} \underbrace{\omega^{\omega^{\cdot^{\cdot^{\omega}}}}}_{n \text{ times}}$$

もし $\alpha \in E$ ならば，ある k について $\alpha \in D_k$ である．$D_0 = \mathrm{Mul}(\{\varnothing\})$ とすることで，E の全ての元は有限多重集合だと思うことができる．つまり，$\{\varnothing \otimes n\}$ が自然数 n を表す．$\leqslant'_D$ の定義の一番左より，空集合 $\varnothing$ が E の最小元であるから，順序数 0 を表す．よって，以後は $\varnothing$ のことを 0 と表す．同様に，以後は $\alpha = \{\alpha_0 \otimes k_0, \ldots, \alpha_\ell \otimes k_\ell\} \in E$ のことを

$$\omega^{\alpha_0} \cdot k_0 + \cdots + \omega^{\alpha_\ell} \cdot k_\ell$$

と表す．以後，しばしば $\alpha \in E$ のことを $\alpha < \varepsilon_0$ と略記する．もし $\alpha > 0$ ならば，$n \in \mathbb{N}$ に対して，$\alpha[n]$ を以下によって帰納的に定義する．

1. α が後続順序数ならば，$\alpha[n] = \alpha - 1$ とする．
2. α_ℓ が後続順序数ならば，$\alpha[n] = \omega^{\alpha_0} \cdot k_0 + \cdots + \omega^{\alpha_\ell} \cdot (k_\ell - 1) + \omega^{\alpha_\ell - 1} \cdot n$ とする．
3. α_ℓ が極限順序数ならば，$\alpha[n] = \omega^{\alpha_0} \cdot k_0 + \cdots + \omega^{\alpha_\ell} \cdot (k_\ell - 1) + \omega^{\alpha_\ell[n]}$ とする．

このとき，$(\alpha[n])_{n\in\mathbb{N}}$ のことを α の**基本列** (*fundamental sequence*) と呼ぶ．さて，それでは関数階層の話に戻ろう．自然数上の関数の階層 $(\mathbf{f}_\alpha)_{\alpha<\varepsilon_0}$ を以下によって定義する．

定義 6.5. 各 $\alpha < \varepsilon_0$ に対して，関数 $\mathbf{f}_\alpha \colon \mathbb{N} \to \mathbb{N}$ を以下のように帰納的に定義する．

$$\mathbf{f}_0(x) = x + 1 \qquad \mathbf{f}_{\alpha+1}(x) = \mathbf{f}_\alpha^{(x)}(x) \qquad \mathbf{f}_\eta(x) = \mathbf{f}_{\eta[x]}(x) \quad (\eta \text{ limit})$$

この関数階層 $(\mathbf{f}_\alpha)_{\alpha<\varepsilon_0}$ を**急増加階層** (*fast growing hierarchy*) と呼ぶ．

もちろん，$\omega \leqslant \alpha < \varepsilon_0$ に対して，$\mathbf{f}_\alpha$ の定義は原始再帰法によるものではない．関数の定義に順序数を伴うので，これは超限再帰の一種である．しかし，順序数の整礎性より，定義に従って $\mathtt{f}_\alpha(x)$ の値を求める手続きは有限ステップで停止する．たとえば，$\mathbf{f}_{\omega^2}(5)$ を計算する過程では，まず $\mathbf{f}_{\omega\cdot 5}(5)$ を読み込んで，次に $\mathbf{f}_{\omega\cdot 4+5}(5)$ を読み込んで，$\mathbf{f}^{(5)}_{\omega\cdot 4+4}(5)$ を読み込んで……と計算が進んでいくが，添字の順序数はどんどん下がっていくので，この計算は必ず有限ステップで終了するのである．

観察 1. 任意の順序数 $\alpha < \varepsilon_0$ に対して，関数 $\mathbf{f}_\alpha \colon \mathbb{N} \to \mathbb{N}$ は計算可能である．

ここで，関数が計算可能であるとは，それがチューリング・マシン等の計算モデルで有限ステップで計算できることを意味する．このように，

順序数（の整礎性）は，様々な手続きの有限性を保証するための重要な道具

である．一応注意しておけば，上記の観察に関して，ε_0 であるという点は重要ではない．ε_0 より大きな順序数に対しても，基本列のシステムを適切に与えた場合，対応する関数は計算可能関数になる．具体的には，計算可能な整礎木 T に対して，$\mathbf{f}_T$ は計算可能関数であると言える．

6.4. 急増加階層の項表現

急増加階層は，ゲーデルのシステム T の強さを測る指標として有用である．まず，関数階層 $(\mathbf{f}_n)_{n<\omega}$ が T-項によって表現できることは容易に分かるだろう．具体的には，次の変換

$$(n, f, x) \mapsto f^{(n)}(x)$$

を表す項は，$\mathtt{iter} := \lambda nfx.\mathtt{irec}\ x\ f\ n$ である．このとき，$\mathbf{f}_0$ は $\mathtt{fast}[0] := \mathtt{succ}$ によって表され，$\mathbf{f}_{n+1}$ は以下の項によって表される．

$$\mathtt{fast}_{n+1} := \lambda x.(\mathtt{iter}\ x\ \mathtt{fast}_n\ x)$$

一応，丁寧に確認すれば，

$$[\![\mathtt{fast}_{n+1}]\!](x) = [\![\mathtt{iter}\ x\ \mathtt{fast}_n\ x]\!] = [\![\mathtt{fast}_n]\!]^{(x)}(x)$$

となっている．実際，ω 未満のランクの急増加関数であれば，T_0-項によって表すことができる．しかし，$\mathtt{f}_\omega$ の段階でアッカーマン関数レベルの複雑さになり，もう T_0-項によって表すことができない．とはいえ，アッカーマン関数を高階原始再帰のシステム T の項によって表すことができたことから，$\mathtt{f}_\omega$ を T-項で表せることは予期できるだろう．それでは，どのレベルの急増加関数まで，T の項として表現することができるだろうか．実を言うと，興味深いことに，ε_0 までのランクの急増加階層はすべて，ゲーデルの T によって表すことができるのである．

まず，ω 未満のランクの急増加関数の構成は，以下の項を用いて表すこともできる．

$$\mathtt{iter}^2 := \lambda fx.(\mathtt{iter}\ x\ f\ x)$$

肩の 2 は，型 1 関数 $f\colon \mathtt{N} \to \mathtt{N}$ を入力としている型 2 汎関数であることを示唆している．型を明示的に書けば，$\mathtt{iter}^2\colon (\mathtt{N} \to \mathtt{N}) \to \mathtt{N} \to \mathtt{N}$ あるいは $\mathtt{iter}^2\colon (\mathtt{N} \to \mathtt{N}) \times \mathtt{N} \to \mathtt{N}$ である．この項は関数 $(f, x) \mapsto f^{(x)}(x)$ を表していることに注意する．このとき，

$$\mathtt{fast}_n = \mathtt{iter}\ n\ \mathtt{iter}^2\ \mathtt{fast}_0$$

が成立している．この項の意味を考えれば，

$$\begin{aligned}[\![\mathtt{fast}_n]\!](x) &= [\![\mathtt{iter}^2]\!]^{(n)}([\![\mathtt{fast}_0]\!])(x) \\ &= [\![\mathtt{iter}^2]\!]^{(n)}(\mathbf{f}_0)(x) = \mathbf{f}_n(x)\end{aligned}$$

となっていることが分かる．したがって，$\mathbf{f}_\omega$ は次の項で表されることが分かる．

$$\mathtt{fast}_\omega = \lambda x.(\mathtt{iter}\ x\ \mathtt{iter}^2\ \mathtt{fast}_0\ x)$$

したがって，急増加階層の低いランクであれば，以下のように有限的な項で表すことができる．

$$\mathtt{fast}_{\omega+n} = \mathtt{iter}\ n\ \mathtt{iter}^2\ \mathtt{fast}_\omega$$

$$\mathtt{fast}_{\omega\cdot 2} = \lambda x.(\mathtt{iter}\ x\ \mathtt{iter}^2\ \mathtt{fast}_\omega\ x)$$

$$\mathtt{fast}_{\omega\cdot(n+1)} = \lambda x.(\mathtt{iter}\ x\ \mathtt{iter}^2\ \mathtt{fast}_{\omega\cdot n}\ x)$$

しかし，これはもう少しすっきりと表すことができる．まず，$\mathtt{fast}_{\omega\cdot n}$ の定義に注目して，次の項を考えよう．

$$\begin{cases} \mathtt{fast}_0^2 := \mathtt{iter}^2 \\ \mathtt{fast}_{n+1}^2 := \lambda f x.(\mathtt{iter}\ x\ \mathtt{fast}_n^2\ f\ x) \end{cases}$$

ここで $\mathtt{fast}_n^2$ の型は $(\mathtt{N} \to \mathtt{N}) \to \mathtt{N} \to \mathtt{N}$ である．つまり，関数を入力して関数を出力する汎関数であると考えると分かりやすい．この項を利用すると，急増加関数を表す項は，次のようにも表記することができる．

$$\mathtt{fast}_1^2\ \mathtt{fast}_0 \xrightarrow{*} \mathtt{fast}_\omega$$

$$\mathtt{fast}_1^2\ \mathtt{fast}_{\omega\cdot n} \xrightarrow{*} \mathtt{fast}_{\omega\cdot(n+1)}$$

したがって，$\mathbf{f}_{\omega\cdot m}$ および $\mathbf{f}_{\omega\cdot m+n}$ は次のような項として表現できる．

$$\mathtt{iter}\ m\ \mathtt{fast}_1^2\ \mathtt{fast}_0 \xrightarrow{*} \mathtt{fast}_{\omega\cdot m}$$

$$\mathtt{iter}\ n\ \mathtt{fast}_0^2\ (\mathtt{iter}\ m\ \mathtt{fast}_1^2\ \mathtt{fast}_0) \xrightarrow{*} \mathtt{fast}_{\omega\cdot m+n}$$

前者に注目すれば，

$$\mathtt{fast}_2^2\ \mathtt{fast}_0 \xrightarrow{*} \lambda x.(\mathtt{iter}\ x\ \mathtt{fast}_1^2\ \mathtt{fast}_0\ x) = \mathtt{fast}_{\omega^2}$$

であることが分かる．これを繰り返すと，

$$\mathtt{fast}_n^2\ \mathtt{fast}_0 \xrightarrow{*} \mathtt{fast}_{\omega^n}$$

を得る．さて，ここまでの項は，型レベル 2 以下の項しか出現していないことに注意しよう．つまり，急増加階層 $(\mathbf{f}_\alpha)_{\alpha<\omega^\omega}$ は，型レベル 2 以下の T-項を用いて表現できる．

この先に進むには，$\mathtt{fast}_n^2$ の定義に注目する必要がある．この構成は，以下のような型 3 累積関数を表す項として置き換えることができる．

$$\mathtt{iter}^3 := \lambda F f x.(\mathtt{iter}\ x\ F\ f\ x)$$

つまり，$\mathtt{iter}^3$ は $(F, f, x) \mapsto F^{(x)}(f, x)$ を表す項である．これを $\mathtt{fast}_n^2$ の定義と比較すれば，$\mathtt{iter}^3\ \mathtt{fast}_n^2 \xrightarrow{*} \mathtt{fast}_{n+1}^2$ であるから，以下が成立していることが分かる．

$$\mathtt{iter}\ n\ \mathtt{iter}^3 \xrightarrow{*} \mathtt{fast}_n^2$$

よって，$\mathbf{f}_{\omega^n}$ は $\mathtt{fast}_n^2\ \mathtt{fast}_0$ によって表されていたことから，$\mathbf{f}_{\omega^\omega}$ は次の項によって表すことができる．

$$\mathtt{fast}_{\omega^\omega} := \lambda x.(\mathtt{iter}\ x\ \mathtt{iter}^3\ \mathtt{fast}_0\ x)$$

このような感じで続けていけば，急増加階層のかなりのレベルを T の項で書けるという予想が付くであろう．鍵となる概念は，$\mathtt{iter}^n$ である．これは，$n \geqslant 2$ について，次のように定義される．

$$\mathtt{iter}^n := \lambda x_n x_{n-1} x_{n-2} \ldots x_0.(\mathtt{iter}\ x_0\ x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_2\ x_1\ x_0)$$

ここで，x_k は型 $\mathtt{k}$ 変数である．具体的に，項ではなく関数を用いてこの概念を整理してみよう．

定義 6.6. 各自然数 $n \geqslant 2$ と順序数 $\alpha < \varepsilon$ に対して，高階関数 $\mathbf{f}_\alpha^n$ を以下のように帰納的に定義する．

$$\begin{aligned}
\mathbf{f}_0^n(x_n, x_{n-1}, \ldots, x_1, x_0) &= x_n^{(x_0)}(x_{n-1}, \ldots, x_1, x_0) \\
\mathbf{f}_{\alpha+1}^n(x_n, x_{n-1}, \ldots, x_1, x_0) &= (\mathbf{f}_\alpha^n)^{(x_0)}(x_n, x_{n-1}, \ldots, x_1, x_0) \\
\mathbf{f}_\eta^n(x_n, x_{n-1}, \ldots, x_1, x_0) &= \mathbf{f}_{\eta[x_0]}^n(x_n, x_{n-1}, \ldots, x_1, x_0) \qquad (\text{if } \eta \text{ is limit})
\end{aligned}$$

以下，$\mathbf{f}_\alpha^1 = \mathbf{f}_\alpha$ とする．いま，順序数 $\alpha < \varepsilon_0$ は，多重集合 $\{\alpha_0 \otimes n_0, \alpha_1 \otimes n_1, \ldots, \alpha_k \otimes n_k\}$ として表されていたから，つまり $\omega^{\alpha_0} \cdot n_0 + \omega^{\alpha_1} \cdot n_1 + \cdots + \omega^{\alpha_k} \cdot n_k$ の形で書くことができる．ここで，$\alpha_0 > \alpha_1 > \cdots > \alpha_k$ である．最後の項の指数 α_k のことを $\mathrm{last}(\alpha)$ と書くことにしよう．いま，$\alpha < \varepsilon_0$ ならば $\alpha_i < \alpha$ であることに注意する．

定理 6.7. 任意の順序数 α, β に対して，もし $\mathrm{last}(\beta) \geqslant \alpha$ ならば，以下が成立する．

$$\mathbf{f}_\alpha^{n+1}(\mathbf{f}_\beta^n) = \mathbf{f}_{\beta+\omega^\alpha}^n.$$

Proof. まず，$\alpha = 0$ の場合には，定義より以下が成立する．

$$\mathbf{f}_0^{n+1}(\mathbf{f}_\beta^n, x_{n-1}, \ldots, x_0) = (\mathbf{f}_\beta^n)^{(x_0)}(x_{n-1}, \ldots, x_0) = \mathbf{f}_{\beta+1}^n(x_{n-1}, \ldots, x_0).$$

もし α が後続順序数 $\alpha^- + 1$ の場合には，以下が成り立つ．

$$\begin{aligned}
\mathbf{f}_\alpha^{n+1}(\mathbf{f}_\beta^n, x_{n-1}, \ldots, x_0) &= (\mathbf{f}_{\alpha^-}^{n+1})^{(x_0)}(\mathbf{f}_\beta^n, x_{n-1}, \ldots, x_0) && (\text{by definition}) \\
&= \mathbf{f}_{\beta+\omega^{\alpha^-}\cdot x_0}^n(x_{n-1}, \ldots, x_0) && (\text{by induction hypothesis}) \\
&= \mathbf{f}_{(\beta+\omega^\alpha)[x_0]}^n(x_{n-1}, \ldots, x_0) && (\because\ \mathrm{last}(\beta) \geqslant \alpha) \\
&= \mathbf{f}_{\beta+\omega^\alpha}^n(x_{n-1}, \ldots, x_0) && (\text{by definition})
\end{aligned}$$

もし α が極限順序数の場合には，以下が成り立つ．

$$\begin{aligned}
\mathbf{f}_{\alpha}^{n+1}(\mathbf{f}_{\beta}^{n}, x_{n-1}, \ldots, x_0) &= \mathbf{f}_{\alpha[x_0]}^{n+1}(\mathbf{f}_{\beta}^{n}, x_{n-1}, \ldots, x_0) && \text{(by definition)} \\
&= \mathbf{f}_{\beta+\omega^{\alpha[x_0]}}^{n}(x_{n-1}, \ldots, x_0) && \text{(by induction hypothesis)} \\
&= \mathbf{f}_{(\beta+\omega^{\alpha})[x_0]}^{n}(x_{n-1}, \ldots, x_0) && (\because\ \mathrm{last}(\beta) \geqslant \alpha) \\
&= \mathbf{f}_{\beta+\omega^{\alpha}}^{n}(x_{n-1}, \ldots, x_0) && \text{(by definition)}
\end{aligned}$$

以上より定理は示された． □

$\mathbf{f}_0^n$ は項 $\mathtt{iter}^n$ によって表されるから，以下の結論が得られる．

系 6.8. 任意の $\alpha < \varepsilon_0$ に対して，$\mathbf{f}_\alpha$ はシステム T の項によって表現可能である．

Proof. 多重集合の階層 $E = \bigcup_{m\in\mathbb{N}} D_m$ が ε_0 を表していたことを思い出そう．帰納的に，各 $\alpha \in D_m$ および $n \in \mathbb{N}$ に対して，$\mathbf{f}_\alpha^n$ がシステム T の項で表現可能であることを示す．$m = 0$ の場合は，$\alpha < \omega$ であるから，$\mathbf{f}_\alpha^n$ が T-項として表せることは既に知っている．$m > 0$ の場合，$\alpha \in D_m$ は D_{m-1} 上の多重集合であるから，α は多重集合 $\{\alpha_0 \otimes k_0, \ldots, \alpha_\ell \otimes k_\ell\}$ として書くことができる．ここで，各 $i \leqslant \ell$ について $\alpha_i \in D_{m-1}$ である．これが $\alpha = \sum_{i\leqslant\ell} \omega^{\alpha_i} \cdot k_i$ を意味していることを思い出せば，定理 6.7 より，以下を得る．

$$\mathbf{f}_\alpha^n = (\mathbf{f}_{\alpha_\ell}^{n+1})^{(k_\ell)} \circ (\mathbf{f}_{\alpha_{\ell-1}}^{n+1})^{(k_{\ell-1})} \circ \cdots \circ (\mathbf{f}_{\alpha_1}^{n+1})^{(k_1)} \circ (\mathbf{f}_{\alpha_0}^{n+1})^{(k_0)}(\mathbf{f}_0^n)$$

各 $i \leqslant \ell$ について $\alpha_i \in D_{m-1}$ であるから，帰納的仮定より $\mathbf{f}_{\alpha_i}^{n+1}$ は T-項として表すことができる．よって，$\mathbf{f}_\alpha^n$ は T-項として表すことができる． □

実際，急増加階層 $(\mathbf{f}_\alpha)_{\alpha<\varepsilon_0}$ は，数項 $\underline{n}$, 項 $\mathtt{succ}$ と $\mathtt{iter}$, $(\mathtt{iter}^n)_{n\geqslant 2}$ のみを用いて表現することができる．たとえば，急増加関数 $\mathbf{f}_{\omega^3\cdot 2+\omega\cdot 4+5}$ を考えると，上の定理より，

$$\begin{aligned}
\mathbf{f}_{\omega^3\cdot 2+\omega\cdot 4+5} &= (\mathbf{f}_0^2)^{(5)}(\mathbf{f}_{\omega^3\cdot 2+\omega\cdot 4}) \\
&= (\mathbf{f}_0^2)^{(5)} \circ (\mathbf{f}_1^2)^{(4)}(f_{\omega^3\cdot 2}) \\
&= (\mathbf{f}_0^2)^{(5)} \circ (\mathbf{f}_1^2)^{(4)} \circ (\mathbf{f}_3^2)^{(2)}(\mathbf{f}_0) \\
&= (\mathbf{f}_0^2)^{(5)} \circ (\mathbf{f}_0^3(\mathbf{f}_0^2))^{(4)} \circ ((\mathbf{f}_0^3)^{(3)}(\mathbf{f}_0^2))^{(2)}(\mathbf{f}_0)
\end{aligned}$$

したがって，$\mathbf{f}_{\omega^3\cdot 2+\omega\cdot 4+5}$ を表す項は以下によって与えられる．

$$\mathtt{iter}\ \underline{5}\ \mathtt{iter}^2\ (\mathtt{iter}\ \underline{4}\ (\mathtt{iter}^3\ \mathtt{iter}^2)\ (\mathtt{iter}\ \underline{2}\ (\mathtt{iter}\ \underline{3}\ \mathtt{iter}^3\ \mathtt{iter}^2)\ \mathtt{succ}))$$

もう一つ例を挙げておけば，たとえば $\omega^{\omega^{3\cdot 2+4}} + \omega^{\omega^2}\cdot 7 + 6$ ランクの急増加関数を考えると，上

の定理より，

$$\begin{aligned}
\mathbf{f}_{\omega^{\omega^3\cdot 2+4}+\omega^{\omega^2}\cdot 7+6} &= (\mathbf{f}_0^2)^{(6)}(\mathbf{f}_{\omega^{\omega^3\cdot 2+4}+\omega^{\omega^2}\cdot 7}) \\
&= (\mathbf{f}_0^2)^{(6)} \circ (\mathbf{f}_{\omega^2}^2)^{(7)}(\mathbf{f}_{\omega^{\omega^3\cdot 2+4}}) \\
&= (\mathbf{f}_0^2)^{(6)} \circ (\mathbf{f}_{\omega^2}^2)^{(7)} \circ \mathbf{f}_{\omega^3\cdot 2+4}^2(\mathbf{f}_0) \\
&= (\mathbf{f}_0^2)^{(6)} \circ \left(\mathbf{f}_2^3(\mathbf{f}_0^2)\right)^{(7)} \circ \left((\mathbf{f}_0^3)^{(4)} \circ (\mathbf{f}_3^3)^{(2)}(\mathbf{f}_0^2)\right)(\mathbf{f}_0) \\
&= (\mathbf{f}_0^2)^{(6)} \circ \left((\mathbf{f}_0^4)^{(2)}(\mathbf{f}_0^3)(\mathbf{f}_0^2)\right)^{(7)} \circ \left((\mathbf{f}_0^3)^{(4)} \circ \left((\mathbf{f}_0^4)^{(3)}(\mathbf{f}_0^3)\right)^{(2)}(\mathbf{f}_0^2)\right)(\mathbf{f}_0)
\end{aligned}$$

項で表すと長くなるから具体的には書かないが，これを T の項で表せることは明らかであろう．さて，もう一つ注目する点として，関数階層 $(\mathbf{f}_n)_{n<\omega}$ は型レベル 1 以下の T-項，すなわち T_0-項によって表すことができ，関数階層 $(\mathbf{f}_\alpha)_{\alpha<\omega^\omega}$ は型レベル 2 以下の T-項によって表すことができた．この観点をもう少し精密化していこう．

定義 6.9. 型 σ のレベルを以下の方法で帰納的に定義する．

$$\mathsf{lev}(\mathtt{N}) = 0, \qquad \mathsf{lev}(\sigma \to \tau) = \max\{\mathsf{lev}(\sigma)+1, \mathsf{lev}(\tau)\}.$$

型 σ の項 t のレベルは，$\mathsf{lev}(t) = \mathsf{lev}(\sigma)$ によって定義される．

例 6.10. 項 $\mathtt{iter}^n$ の型レベルは n である．

多重集合の階層 $(D_m)_{m\in\mathbb{N}}$ について，D_m の順序型を $\omega \uparrow\uparrow (m+1)$ と書く．原始再帰的に $\omega \uparrow\uparrow 0 = 1$ かつ $\omega \uparrow\uparrow (n+1) = \omega^{\omega\uparrow\uparrow n}$ と定義してもよい．このとき，以下が成立する．

系 6.11. 任意の自然数 $n > 0$ と順序数 $\alpha < \omega \uparrow\uparrow n$ に対して，$\mathbf{f}_\alpha$ は型レベル n 以下の T-項によって表現可能である．

Proof. 証明は系 6.8 とほとんど同様である．帰納的に，$\alpha \in D_m$ と $n \in \mathbb{N}$ に対して，$\mathbf{f}_\alpha^n$ が型レベル $n+m$ 以下の T-項によって表現可能であることを示す．ただし，$\mathbf{f}_\alpha^1 = f_\alpha$ であるとする．$m = 0$ の場合は，$\alpha < \omega$ であるから，$\mathbf{f}_\alpha^n$ は型レベル n 以下の T-項で表せる．$m > 0$ の場合は，系 6.8 の証明のようにして，$\mathbf{f}_\alpha^n$ は $(\mathbf{f}_{\alpha_i}^{n+1})_{i\leqslant\ell}$ を用いて表すことができる．また，$\alpha_i \in D_{m-1}$ なので，帰納的仮定より，各 $\mathbf{f}_{\alpha_i}^{n+1}$ は型レベル $n+m$ 以下の項を用いて表される．よって，$\mathbf{f}_\alpha^n$ は型レベル $n+m$ 以下の項によって表される．いま，$\alpha < \omega \uparrow\uparrow n$ ならば $\alpha \in D_{n-1}$ であるから，$\mathbf{f}_\alpha = \mathbf{f}_\alpha^1$ は型レベル $1+(n-1) = n$ 以下の T-項によって表現される． □

歴史. 歴史的には，これは 1925 年のミュンスター会議においてヒルベルトの報告したプロジェクトと関係しているようだ．つまり，第 1 ステップとして「自然数上の関数の構成から超限再帰を除去して，高階汎関数を用いた有限的な再帰に置き換える」ということ，そして，第 2 ステップとして「再帰的定義に必要な型のランクによって自然数上の関数を分類する」というものである．ただし，ヒルベルトは，有限型の汎関数に留まらず，超限順序数型の高階汎関数を主な考察対象としている．どうやら，超限型の汎関数による再帰まで考えることによって，実質的に，自然数上のすべての関数の分類が可能であると考えていたようである．このように

自然数上の関数とそれを再帰的定義するのに必要な型という順序数を対応付けることによって，自然数上の関数の個数を数え上げる—それが当時ヒルベルトの思い描いていた「連続体仮説」を解くための戦略であった．

6.5. β-正規化定理

ゲーデルのシステム T の項をプログラムだと思うことにすると，このプログラムを実行した結果というのは，簡約 $s \longrightarrow t$ を行っていく手続きである．つまり，システム T における計算とは，ラムダ抽象と原始再帰を削除していく手続きである．さて，我々はシステム T における計算が無限ループに陥らないことを示したい．無限ループに陥らない，ということは簡約を続けていけば，いつか正規形に辿り着く，ということである．なぜ正規形に辿り着くと嬉しいだろうか．

自然数上の関数を表すプログラムは，型 $\mathtt{N}^k \to \mathtt{N}$ 項 t である．我々は，このプログラム t に k 個の数項 $\underline{n_1}, \ldots, \underline{n_k}$ を入力する．このとき，項 $t\ \underline{n_1}\ \ldots\ \underline{n_k}$ は型 $\mathtt{N}$ 閉項であるが，これがどんな自然数値であるかは項の形から一目瞭然というわけではない．この自然数値を求めるためには，それを数項の形に変形する必要がある．次の命題は，この数項の形に変形するためには，正規形にすればよいということを述べる．

命題 6.12. 型 $\mathtt{N}$ 正規閉項は，数項である．

Proof. 項 t を型 $\mathtt{N}$ 正規閉項とする．項 t の構成で，λ 抽象または原始再帰の導入を含むと仮定して矛盾を導く．まず，項 t の部分項が λ 抽象または原始再帰の導入直後の項であれば，それは関数型の項である．しかし，関数型の項 s を型 $\mathtt{N}$ 項に変える項構成は関数適用 sz しか存在しない．関数型の項の直後の項構成は，関数適用，λ-抽象または原始再帰しかないから，関数適用に辿り着くまで，$\lambda x.s'$ または $\mathtt{rec}\ s'\ s''$ の形を保ち続ける．したがって，この形の項 s の右に型 $\mathtt{N}$ 項 z が配置される．この項 z が自由変数 x を含む場合には，t が閉項であるという仮定から，sz の外で変数 x を束縛する λ-抽象が導入されるはずであるから，同じ議論を行う．よって，関数適用 sz の型 $\mathtt{N}$ 項 z は自由変数を含まないと仮定できる．つまり，z は型 $\mathtt{N}$ 正規閉項で，t の真部分項であるから，帰納的仮定より，z は数項である．$s = \lambda x.s'$ の形の場合には，$sz = (\lambda x.s')z$ は可約である．同様に，$s = \mathtt{rec}\ s'\ s''$ の形の場合も，$z = \underline{n}$ の形であるから，$sz = (\mathtt{rec}\ s'\ s'')\underline{n}$ は可約である．どちらにせよ，sz は可約であるから，t は正規ではない． □

それでは，システム T のプログラムは決して無限ループに陥らずに有限ステップで計算が終了する，つまり**正規化可能**であることを示したい．このために，最も注意しなければならないのは $\mathtt{rec}$ の計算手続きである．項 $\mathtt{rec}\ g\ h\ \underline{n+1}$ の計算がどのように実行されるかを思い出せば，

$$\begin{aligned}\mathtt{rec}\ g\ h\ \underline{n+1} &\longrightarrow h\ \underline{n}\ (\mathtt{rec}\ g\ h\ \underline{n}) \\ &\longrightarrow h\ \underline{n}\ (h\ \underline{n-1}\ (\mathtt{rec}\ g\ h\ \underline{n-1})) \\ &\xrightarrow{*} h\ \underline{n}\ (h\ \underline{n-1}\ (\ldots h\ \underline{1}\ (h\ \underline{0}\ g))\ldots)\end{aligned}$$

となる．とりあえずここまでの手続きは自動的に認めてしまっていいとしよう．ここから先の分析

においては，$\mathtt{rec}\ f\ g$ をそのまま扱うのではなく，これを列 $\langle \mathtt{rec}\ f\ g\ \underline{n} \rangle_{n\in\mathbb{N}}$ あるいはその簡約後の結果

$$\mathtt{rec}^\star\ g\ h := \langle g, h\underline{0}g,\ h\underline{1}(h\underline{0}g),\ h\underline{2}(h\underline{1}(h\underline{0}g)), \dots, h\underline{n}(h\underline{n-1}(\dots h\underline{1}(h\underline{0}g))\dots), \dots \rangle$$

を扱った方が話が容易になる．つまり，これは原始再帰法を次のような無限ルールへと分解したものと思ってもよい．

$$\frac{g \qquad h}{\mathtt{rec}\ g\ h} \qquad \rightsquigarrow \qquad \frac{g \quad h\underline{0}g \quad h\underline{1}(h\underline{0}g) \quad h\underline{2}(h\underline{1}(h\underline{0}g)) \quad \dots}{\mathtt{rec}^\star\ g\ h}$$

ここで，$\mathtt{rec}^\star\ g\ h$ は，列 $\langle g, h\underline{0}g, h\underline{1}(h\underline{0}g), h\underline{2}(h\underline{1}(h\underline{0}g)), \dots \rangle$ を表すものとする．項の構文木は有限木であるが，この分解を経ると無限木になる．しかし，無限木であっても，**整礎木** (*well-founded tree*) である．以上のように，項 t に出現する $\mathtt{rec}$ を $\mathtt{rec}^\star$ に置き換えたものを $t^\star$ と書く．この $t^\star$ の構文木を t の $\star$-構文木と呼ぶことにしよう．

例 6.13. 足し算 $x + y$ について，$\mathtt{s} := \lambda n.\mathtt{succ}$ とすれば，$\mathtt{add} = \lambda x.\mathtt{rec}\ x\ \mathtt{s}$ と表せたことに注意する．まず，$\mathtt{add}$ の構文木は次のように得られる．

$$\dfrac{\dfrac{[x\colon \mathtt{N}] \qquad \dfrac{\mathtt{succ}\colon \mathtt{N}\to\mathtt{N}}{\mathtt{s} = \lambda n.\mathtt{succ}\colon \mathtt{N}\to\mathtt{N}\to\mathtt{N}}\ (\to\mathrm{I})}{\mathtt{rec}\ x\ \mathtt{s}\colon \mathtt{N}\to\mathtt{N}}\ (\mathrm{R})}{\lambda x.(\mathtt{rec}\ x\ \mathtt{s})\colon \mathtt{N}\to\mathtt{N}\to\mathtt{N}}\ (\to\mathrm{I}) \qquad\qquad \dfrac{x \qquad \dfrac{\mathtt{succ}}{\mathtt{s} = \lambda n.\mathtt{succ}}}{\dfrac{\mathtt{rec}\ x\ \mathtt{s}}{\lambda x.(\mathtt{rec}\ x\ \mathtt{s})}}$$

左側は，項の型を推論する木である．右側は，左の木から型の情報を消去したものであり，これは項の構文木である．他方，$\mathtt{add}$ の $\star$-構文木は以下のように与えられる．

$$\dfrac{x \quad \dfrac{\mathtt{s} \quad \underline{0}}{\mathtt{s}\underline{0}} \quad x}{\mathtt{s}\underline{0}x} \quad \dfrac{\mathtt{s} \quad \underline{1} \quad \dfrac{\mathtt{s} \quad \underline{0}}{\mathtt{s}\underline{0}} \quad x}{\mathtt{s}\underline{1}(\mathtt{s}\underline{0}x)} \quad \dfrac{\mathtt{s} \quad \underline{2} \quad \dfrac{\mathtt{s} \quad \underline{1} \quad \dfrac{\mathtt{s} \quad \underline{0}}{\mathtt{s}\underline{0}} \quad x}{\mathtt{s}\underline{1}(\mathtt{s}\underline{0}x)}}{\mathtt{s}\underline{2}(\mathtt{s}\underline{1}(\mathtt{s}\underline{0}x))} \quad \dots}{\dfrac{\mathtt{rec}^\star\ x\ \mathtt{s}}{\lambda x.\mathtt{rec}^\star\ x\ \mathtt{s}}}$$

正確に言えば，$\mathtt{s}$ や $\underline{1}, \underline{2}$ などはもう少し分解できるが，省略することにする．

このような $\star$-構文に基づいて，簡約を行うことにする．この手続きは，全ての $\mathtt{rec}$ の出現を除去して，列 $\langle p_k \rangle$ の形に変形する．したがって，いま，項の構成規則は以下のように変化している．

$$\frac{}{x^\sigma\colon \sigma} \qquad \frac{}{\underline{0}\colon \mathtt{N}} \qquad \frac{}{\mathtt{succ}\colon \mathtt{N}\to\mathtt{N}}$$

$$\frac{s\colon \sigma\to\tau \qquad t\colon \sigma}{st\colon \tau}\ (\to\mathrm{E}) \qquad\qquad \frac{\begin{array}{c}[x^\sigma\colon \sigma]\\ \vdots \\ t\colon \tau\end{array}}{\lambda x^\sigma.t\colon \sigma\to\tau}\ (\to\mathrm{I})$$

$$\frac{p_0 : \sigma \quad p_1 : \sigma \quad \dots \quad p_n : \sigma \quad p_{n+1} : \sigma \quad \dots}{\langle p_k \rangle_{k \in \mathbb{N}} : \mathtt{N} \to \sigma} \ (\mathrm{R}^\star)$$

この新たなタイプの項を $\star$-項と呼ぶことにしよう．このとき，$\star$-項の簡約を定義する必要がある．

定義 6.14. $\star$-項上の簡約関係 $\xrightarrow{*}$ を次によって定義する．

1. $t \xrightarrow{*} t$ である．
2. $r \xrightarrow{*} s \xrightarrow{*} t$ ならば，$r \xrightarrow{*} t$ である．
3. $t \xrightarrow{*} t'$ かつ $s \xrightarrow{*} s'$ ならば，$ts \xrightarrow{*} t's'$ である．
4. $t \xrightarrow{*} t'$ ならば，$\lambda x.t \xrightarrow{*} \lambda x.t'$ である．
5. 任意の $i \in \mathbb{N}$ について $t_i \xrightarrow{*} t'_i$ ならば，$\langle t_i \rangle_{i \in \mathbb{N}} \xrightarrow{*} \langle t'_i \rangle_{i \in \mathbb{N}}$ である．
6. $(\lambda x.t)s \xrightarrow{*} t[s/x]$ である．

それでは，まず項から λ 抽象の可約部を削除する手続きに考えよう．項 t の簡約レベルは，t に現れる部分項 $(\lambda x.t)s$ について，$\lambda x.t$ の型レベルの最大値とする．

$$\mathsf{redlev}(t) = \sup\{\mathsf{lev}(\lambda x.t) : (\lambda x.t)s \text{ は } t \text{ の部分項 }\}.$$

T の項 t から得られる無限項 $t^\star$ の簡約レベルは元のものに等しいことに注意する．さて，項から λ 抽象の可約部を削除する手続きとは，簡約レベルを 0 にする手続きである．簡約レベル 0 の項を β-正規形と呼ぶことにする．項が β-**正規化可能**であるとは，β-正規形に簡約可能であることを意味する．

まず，β-簡約の 1 ステップによる簡約レベルの変化に関して，以下を示す．

補題 6.15. 任意の項 s, t および変数記号 x に対して，以下が成立する．

$$\mathsf{redlev}\big(t[s/x]\big) \leqslant \max\big\{\mathsf{redlev}(t), \mathsf{redlev}(s), \mathsf{lev}(s)\big\}$$

Proof. t が定数記号か関数記号の場合は簡約レベルは 0 なので明らかである．t が変数記号の場合，$t \neq x$ ならば $t[s/x] = t$ であるから，簡約レベルは 0 である．$t = x$ の場合は $t[s/x] = s$ なので，$\mathsf{redlev}(t[s/x]) = \mathsf{redlev}(s)$ となり，目的の条件を満たしている．

次に，$t = \lambda y.t'$ の形であると仮定する．このとき，$t[s/x] = \lambda y.t'[s/x]$ であり，t と t' の簡約レベルは等しく，同様に $t[s/x]$ と $t'[s/x]$ の簡約レベルも等しいことに注意する．よって，帰納的仮定より，以下のようにして目的の条件を満たしていることが分かる．

$$\begin{aligned}\mathsf{redlev}(t[s/x]) = \mathsf{redlev}(t'[s/x]) &\leqslant \max\big\{\mathsf{redlev}(t'), \mathsf{redlev}(s), \mathsf{lev}(s)\big\} \\ &= \max\big\{\mathsf{redlev}(t), \mathsf{redlev}(s), \mathsf{lev}(s)\big\}.\end{aligned}$$

つづいて，$t = \langle t_k \rangle_{k\in\mathbb{N}}$ の場合には，$t[s/x] = \langle t_k[s/x] \rangle_{k\in\mathbb{N}}$ であることに注意する．このとき，定義より $\mathtt{redlev}(t) = \sup_{k\in\mathbb{N}} \mathtt{redlev}(t_k)$ であるから，帰納的仮定より，以下を得る．

$$\begin{aligned}\mathtt{redlev}(t[s/x]) = \sup_{k\in\mathbb{N}} \mathtt{redlev}(t_k[s/x]) &\leqslant \sup_{k\in\mathbb{N}} \max\big\{\mathsf{redlev}(t_k), \mathsf{redlev}(s), \mathsf{lev}(s)\big\} \\ &= \max\big\{\mathsf{redlev}(t), \mathsf{redlev}(s), \mathsf{lev}(s)\big\}.\end{aligned}$$

最後に $t = t't''$ であると仮定する．このとき，$\mathsf{redlev}(t) \geqslant \mathsf{redlev}(t'), \mathsf{redlev}(t'')$ である．$t' = \lambda y.u$ の場合，$t = (\lambda y.u)t''$ なので，t の簡約レベルは $\lambda y.u$ の型レベル以上である．また，x に s を代入可能である場合，x と s は同じ型でなければならない．よって，

$$\mathsf{redlev}(t) \geqslant \mathsf{lev}(\lambda y.u) = \mathsf{lev}(\lambda y.u[s/x])$$

となる．したがって，簡約レベルの定義と帰納的仮定より，

$$\begin{aligned}\mathsf{redlev}(t[s/x]) &= \mathsf{redlev}\big((\lambda y.u[s/x])t''[s/x]\big) \\ &\leqslant \max\big\{\mathsf{lev}(\lambda y.u[s/x]), \mathtt{redlev}(u[s/x]), \mathtt{redlev}(t''[s/x])\big\} \\ &\leqslant \max\{\mathsf{redlev}(t), \mathsf{redlev}(u), \mathsf{redlev}(t''), \mathsf{redlev}(s), \mathsf{lev}(s)\} \\ &\leqslant \max\{\mathsf{redlev}(t), \mathsf{redlev}(s), \mathsf{lev}(s)\}\end{aligned}$$

を得る．つづいて，$t' = x$ かつ $s = (\lambda y.s')$ だった場合には，$t[s/x] = (\lambda y.s')t''[s/x]$ である．よって，簡約レベルの定義と帰納的仮定より，

$$\begin{aligned}\mathsf{redlev}(t[s/x]) &\leqslant \max\{\mathsf{lev}(\lambda y.s'), \mathsf{redlev}(t''[s/x])\} \\ &\leqslant \max\{\mathsf{redlev}(t''), \mathsf{redlev}(s), \mathsf{lev}(s)\}\end{aligned}$$

となり，主張は示された．t' と t'' がそれ以外のパターンの場合は自明である． □

これを利用して，項から λ 抽象の可約部を除去することができる．

定理 6.16. システム T の項は β-正規化可能である．

Proof. 与えられた項 t について，$0 < \mathsf{redlev}(t) < \infty$ ならば，$t \xrightarrow{*} t'$ かつ $\mathsf{redlev}(t') < \mathsf{redlev}(t)$ となる項 t' を見つける．t が変数記号 x, 定数記号 $\underline{0}$, 関数記号 $\mathtt{succ}$ の場合には，t の簡約レベルは 0 なので，そうでないことを仮定する．

まず，$t = \lambda x.t'$ の形の場合，t の簡約レベルと t' の簡約レベルは等しい．帰納的仮定より，t' の簡約レベルは 0 であるか，さもなくば $t' \xrightarrow{*} t''$ となる簡約レベルの低い項 t'' を見つけられる．後者の場合は，簡約関係の定義より，$t = \lambda x.t' \xrightarrow{*} \lambda x.t''$ であるから，項 $\lambda x.t''$ が求めるものである．

次に，$t = \langle t_n \rangle_{n\in\mathbb{N}}$ の場合を考えよう．t の簡約レベルは有限の値 r であるから，t_n たちの簡約レベルも r 以下である．帰納的仮定より，t_n の簡約レベルは 0 であるか，$t_n \xrightarrow{*} t'_n$ となる項 t'_n

で簡約レベルが $r-1$ 以下となるものが存在する．前者の場合には，$t'_n = t_n$ とすれば，$\langle t'_n \rangle_{n\in\mathbb{N}}$ の簡約レベルは $r-1$ 以下である．また，簡約関係の定義より，$\langle t_n \rangle_{n\in\mathbb{N}} \xrightarrow{*} \langle t'_n \rangle_{n\in\mathbb{N}}$ である．

最後に，$t = t's$ の場合を考える．$t' = \lambda x.u$ の形でないならば，t の簡約レベルは，u と s のそれの最大値であり，帰納的仮定を用いて，上と同様の方法で簡約レベルを下げることができる．$t' = \lambda x.u$ の形の場合を考える．帰納的仮定より，$u \xrightarrow{*} u'$ で，簡約レベルが u より真に低いか 0 であるような項 u' が存在する．同様に，$s \xrightarrow{*} s'$ で，簡約レベルが s より真に低いか 0 であるような項 s' が存在する．このとき，簡約関係の定義より，

$$t = (\lambda x.u)s \xrightarrow{*} (\lambda x.u')s' \xrightarrow{*} u'[s'/x].$$

このとき，補題 6.15 より，

$$\mathsf{redlev}\big(u'[s'/x]\big) \leqslant \max\big\{\mathsf{redlev}(u'), \mathsf{redlev}(s'), \mathsf{lev}(s')\big\}$$

である．まず，u' の簡約レベルは u のそれより真に低く，t の簡約レベルは u と s のそれの最大値であったから，$\mathsf{redlev}(u') < \mathsf{redlev}(t)$ を得る．同様に，s' の簡約レベルは s のそれより真に低いから，先程と同じ議論によって $\mathsf{redlev}(s') < \mathsf{redlev}(t)$ を得る．次に，s' の型が σ ならば，$\lambda x.u'$ の型は $\sigma \to \tau$ の形でなければならない．型レベルの定義より，後者の方が型レベルが高い，つまり $\mathsf{lev}(s') < \mathsf{lev}(\lambda x.u')$ となる．いま，t は $(\lambda x.u)s$ という項であるから，簡約レベルの定義と簡約が型を変えないことから，$\mathsf{lev}(\lambda x.u') = \mathsf{lev}(\lambda x.u) \leqslant \mathsf{redlev}(t)$ である．以上より，

$$\max\big\{\mathsf{redlev}(u'), \mathsf{redlev}(s'), \mathsf{lev}(s')\big\} < \mathsf{redlev}(t)$$

であるから，$t \xrightarrow{*} u'[s'/x]$ かつ $u'[s'/x]$ の簡約レベルは t から真に下がっている．

システム T の項 t の簡約レベルは有限であり，上記の手続きによって，簡約レベルは真に下がり続けるから，いつか簡約レベル 0 の項に辿り着く．つまり，$t \xrightarrow{*} t'$ となる項 t' で，簡約レベルが 0 であるものが存在する． □

しかし，この手続きはまだ簡約レベルを 0 に落としただけであり，計算は終了していない．つまり，型 $\mathtt{N}$ 閉 $\star$-項が β-正規になったからといって，数項からはまだ程遠い．ここから，さらに数項になるまで簡約してやる必要がある．

6.6. システム T の計算能力と ε_0

項の $\star$-構文木は無限木をなすが，整礎木であるから，そのランクを計算することができる．整礎木 $T \subseteq \omega^{<\omega}$ のランクは，次のように帰納的に定義できる．葉 $\rho \in T$ のランク $\mathsf{rank}_T(\rho)$ は 0 であり，葉でないノード $\sigma \in T$ のランク $\mathsf{rank}_T(\sigma)$ は，

$$\mathsf{rank}_T(\sigma) = \sup\{\mathsf{rank}_T(\sigma^\frown n) : \sigma^\frown n \in T\}$$

によって定義される．そして，T のランクは，T の根のランクによって与えられる．項 t のランクも $\star$-構文木のランクによって定義するが，これを具体的に次のように定義できる．

定義 6.17. $\star$-項 t のランク $\mathsf{rank}(t)$ を次のように帰納的に定義する．

1. t が変数記号 x, 定数記号 $\underline{0}$ または関数記号 $\mathtt{succ}$ ならば，t のランクは 0 である．
2. $t = ss'$ ならば，$\mathsf{rank}(t) = \max\{\mathsf{rank}(s), \mathsf{rank}(s')\} + 1$ である．
3. $t = \lambda x.t'$ ならば，$\mathsf{rank}(t) = \mathsf{rank}(t') + 1$ である．
4. $t = \langle t_n \rangle_{n\in\mathbb{N}}$ ならば，$\mathsf{rank}(t) = \sup_{n\in\mathbb{N}}(\mathsf{rank}(t_n) + 1)$ である．

まず，簡約前の T-項のランクはあまり高くならないことを確認する．

補題 6.18. t がシステム T の項ならば，$\mathsf{rank}(t^\star) < \omega^2$ である．

Proof. 帰納法による．t が変数記号，定数記号，関数記号ならば $t^\star = t$ なので，主張は明らかである．$t = ss'$ の場合，$t^\star = s^\star s'^\star$ である．帰納的仮定より，$s^\star$ と $s'^\star$ のランクは ω^2 未満であり，$t^\star$ のランクはその最大値に 1 を足したものであるから，これは ω^2 未満である．$t = \lambda x.t'$ の場合も，$t^\star = \lambda x.t'^\star$ であるから，先程と同様に帰納的仮定を利用すればよい．$t = \mathtt{rec}\ g\ h$ の場合，$t = \mathtt{rec}^\star\ g^\star\ h^\star = \langle g^\star, h^\star\underline{0}g^\star, h^\star\underline{1}(h^\star\underline{0}g^\star), h^\star\underline{2}(h^\star\underline{1}(h^\star\underline{0}g^\star)), \dots \rangle$ である．この列の第 n 番目は $g^\star, h^\star, \underline{0}, \underline{1}, \dots, \underline{n}$ を適当な順番で関数適用だけを用いて組み合わせたものである．したがって，

$$\mathsf{rank}(\mathtt{rec}^\star\ g^\star\ h^\star) = \sup_{n\in\mathbb{N}}\big(\max\{\mathsf{rank}(g^\star), \mathsf{rank}(h^\star)\} + n\big) = \max\{\mathsf{rank}(g^\star), \mathsf{rank}(h^\star)\} + \omega$$

であることを容易に確認できる．帰納的仮定より，$g^\star$ と $h^\star$ のランクは ω^2 未満なので，ある n について，$\omega \cdot n$ 未満である．以上より，

$$\mathsf{rank}(\mathtt{rec}^\star\ g^\star\ h^\star) = \max\{\mathsf{rank}(g^\star), \mathsf{rank}(h^\star)\} + \omega < \omega \cdot n + \omega = \omega \cdot (n+1) < \omega^2.$$

よって，主張は示された． □

項の変数部分に別の項の代入した後のランクは次のように見積もることができる．

補題 6.19. $\star$-項 t, t' および変数記号 x について，以下の不等式が成立する．

$$\mathsf{rank}\big(t[t'/x]\big) \leqslant \mathsf{rank}(t') + \mathsf{rank}(t).$$

Proof. 項 t の構文木 T の葉のうち，ラベル x の付いた部分をすべて項 t' の構文木に置換したものが, 項 $t[t'/x]$ の構文木 T' である．項のランクは構文木のランクと等しいので, $\mathsf{rank}(t) = \mathsf{rank}_T(\varnothing)$ かつ $\mathsf{rank}(t[t'/x]) = \mathsf{rank}_{T'}(\varnothing)$ であることに注意する．構文木 T 上の帰納法によって，各 σ に対して，$\mathsf{rank}_{T'}(\sigma) \leqslant \mathsf{rank}(t') + \mathsf{rank}_T(\sigma)$ を示す．木 T の葉 ρ のうち，ラベル x の部分は，T' にお

いては t' のランクになる．つまり，$\mathsf{rank}_{T'}(\rho) = \mathsf{rank}(t')$ である．それ以外の T の葉 ρ については，T' でも葉であるから，$\mathsf{rank}_{T'}(\rho) = 0$ が分かる．

いま，葉でないノード $\sigma \in T$ を取る．このとき，帰納的仮定より，

$$\begin{aligned}\mathsf{rank}_{T'}(\sigma) &= \sup\left\{\mathsf{rank}_{T'}(\sigma^\frown n) + 1 : \sigma^\frown n \in T'\right\}\\ &\leqslant \sup\left\{\mathsf{rank}(t') + \mathsf{rank}_T(\sigma^\frown n) + 1 : \sigma^\frown n \in T'\right\}\\ &= \mathsf{rank}(t') + \sup\left\{\mathsf{rank}_T(\sigma^\frown n) + 1 : \sigma^\frown n \in T'\right\}\\ &= \mathsf{rank}(t') + \mathsf{rank}_T(\sigma).\end{aligned}$$

よって，帰納法によって，$\mathsf{rank}_{T'}(\varnothing) \leqslant \mathsf{rank}(t') + \mathsf{rank}_T(\varnothing)$ を得る．項のランクと構文木のランクは等しいから，これは $\mathsf{rank}(t[t'/x]) \leqslant \mathsf{rank}(t') + \mathsf{rank}(t)$ を導く． □

正規化の手続きであまりランクが持ち上がらないことを確認しよう．

補題 6.20. $\star$-項 t の簡約レベルが有限ならば，$t \xrightarrow{*} t_N$ となる β-正規 $\star$-項 t_N が存在して，t_N のランクは，t のランクの次の ε 数未満である．つまり，

$$\mathsf{rank}(t_N) < \omega^{\omega^{\cdot^{\cdot^{\cdot^{\omega^{\mathsf{rank}(t)}}}}}}$$

Proof. 定理 6.16 の簡約レベルを落とす各ステップ $t \xrightarrow{*}_\beta \tilde{t}$ において，$\mathsf{rank}(\tilde{t}) \leqslant \omega^{\mathsf{rank}}(t)$ であることを示す．t が変数記号，定数記号，関数記号の場合には，$\tilde{t} = t$ なので問題ない．$t = \lambda x.t'$ の場合には，$\tilde{t} = \lambda x.\tilde{t'}$ によって与えられていた．帰納的仮定より，

$$\mathsf{rank}(\tilde{t}) = \mathsf{rank}(\lambda x.\tilde{t'}) = \mathsf{rank}(\tilde{t'}) + 1 \leqslant \omega^{\mathsf{rank}(t')} + 1 \leqslant \omega^{\mathsf{rank}(t)}.$$

$t = \langle t_n \rangle_{n\in\mathbb{N}}$ の場合には，$\tilde{t} = \langle \tilde{t_n} \rangle_{n\in\mathbb{N}}$ によって与えられていた．このとき，帰納的仮定より，

$$\mathsf{rank}(\tilde{t}) = \sup_{n\in\mathbb{N}}(\mathsf{rank}(\tilde{t_n}) + 1) \leqslant \sup_{n\in\mathbb{N}}(\omega^{\mathsf{rank}(t_n)} + 1) \leqslant \omega^{\mathsf{rank}(t)}.$$

$t = t's$ の場合，$t' = \lambda x.u$ の形でないならば，同様の議論による．$t' = \lambda x.u$ の形の場合には，$\tilde{t} = \tilde{u}[\tilde{s}/x]$ であった．このとき，t のランクは $\max\{\mathsf{rank}(u) + 1, \mathsf{rank}(s)\} + 1$ であることに注意する．補題 6.19 および帰納的仮定より，

$$\begin{aligned}\mathsf{rank}(\tilde{t}) \leqslant \mathsf{rank}(\tilde{s}) + \mathsf{rank}(\tilde{u}) &\leqslant \omega^{\mathsf{rank}(s)} + \omega^{\mathsf{rank}(u)}\\ &\leqslant \omega^{\max\{\mathsf{rank}(u),\mathsf{rank}(s)\}} \cdot 2 \leqslant \omega^{\mathsf{rank}(t)}.\end{aligned}$$

以上より，主張は示された．β-正規化の手続きは，この議論を有限回繰り返すだけであるから，目的の主張を得る． □

補題 6.18 より，T-項のランクは ω^2 未満であるから，補題 6.20 より，その β-正規化のランクは ε_0 未満である．

定理 6.21. β-正規な型 $\mathtt{N}$ 閉 $\star$-項 t に対して，$t \xrightarrow{*} \underline{m}$ となる数項 $\underline{m}$ が存在する．

Proof. β-正規であるから，特にラムダ抽象も含まないことに注意する．項のランク上の帰納法によって，以下を同時に示していく．

1. 項 t の型が $\mathtt{N}$ ならば，$t \xrightarrow{*} \underline{m}$ となる数項 $\underline{m}$ が存在する．
2. $t = ss'$ の形ならば，$\mathsf{rank}(t') < \mathsf{rank}(t)$ かつ $t \xrightarrow{*} t'$ となる項 t' が存在する．

基底ステップについては，定数記号と関数記号の場合があるが，関数記号は型 $\mathtt{N}$ でも適用の形でもないから，考える必要はない．項 t が定数記号 $\underline{0}$ の場合は，t 自身が数項なので問題ない．

帰納ステップについては，項 t は列か適用の形であるが，列 $\langle t_n \rangle$ の型は $\mathtt{N} \to \sigma$ の形であり，(1),(2) どちらの前提にも当てはまらないから考える必要はない．よって，$t = ss'$ の形のみを考えればよい．項がラムダ抽象を含まない場合，入力の型は $\mathtt{N}$ しかありえないので，s' の型は $\mathtt{N}$ である．s' の方が t よりランクが低いので，帰納的仮定 (1) より，$s' \xrightarrow{*} \underline{m}$ となる数項 $\underline{m}$ が存在する．

さて，s は関数記号か，$\langle s_n \rangle_{n\in\omega}$ の形であるか，ふたたび適用の形である．関数記号 $s = \mathsf{succ}$ の場合には，$t = \mathsf{succ}\ s' \xrightarrow{*} \mathsf{succ}\ \underline{m}$ となるから，これは数項であり，性質 (1) が満たされる．$s = \langle s_n \rangle_{n\in\omega}$ の場合には，

$$t = \langle s_n \rangle s' \xrightarrow{*} \langle s_n \rangle \underline{m} \xrightarrow{*} s_m$$

であるが，

$$\mathsf{rank}(s_m) < \sup_n(\mathsf{rank}(s_n) + 1) = \mathsf{rank}(s) < \mathsf{rank}(t)$$

であるから，性質 (2) が満たされる．また，もし t の型が $\mathtt{N}$ だった場合には，s_m の型も $\mathtt{N}$ であり，これは t よりランクが低いから，帰納的仮定 (1) より，$t \xrightarrow{*} s_m \xrightarrow{*} \underline{k}$ となる数項 $\underline{k}$ が存在し，性質 (1) も満たされる．

項 s が再び適用 uu' の形の場合，s の方が t よりランクが低いので，帰納的仮定 (2) より，s よりランクの低い項 r で，$s \xrightarrow{*} r$ となるものが存在する．このとき，$t \xrightarrow{*} rs'$ であり，

$$\mathsf{rank}(rs') = \max\{\mathsf{rank}(r), \mathsf{rank}(s')\} + 1 < \max\{\mathsf{rank}(s), \mathsf{rank}(s')\} + 1 < \mathsf{rank}(t)$$

であるから，性質 (2) が満たされる．また，もし t の型が $\mathtt{N}$ だった場合には，rs' の型も $\mathtt{N}$ であり，これは t よりランクが低いから，帰納的仮定 (1) より，$t \xrightarrow{*} rs' \xrightarrow{*} \underline{k}$ となる数項 $\underline{k}$ が存在し，性質 (1) も満たされる． □

以上より，型 $\mathtt{N}^n \to \mathtt{N}$ の T-閉項 t が与えられたとき，それが必ずある関数 $[\![t]\!] \colon \mathbb{N}^n \to \mathbb{N}$ を定義することが，ε_0 上の整礎性を用いて示された．もう少し細かいことを言えば，上記の証明を分析すると，T-項によって表現される関数 $f \colon \mathbb{N} \to \mathbb{N}$ は，ε_0 未満の整礎再帰によって計算可能であることも分かる．

§7. 形式算術の解釈

前節までではゲーデルのシステム T という高階原始再帰のシステムを取り扱ったが，システム T という強力な計算システムが必要不可欠となる場面は現れるのだろうか．現実的な計算能力としては，原始再帰関数，つまりシステム T_0 程度で十分だと思う人も多いかもしれない．システム T が登場した歴史的な経緯を述べると，これは，有限型原始再帰の計算の停止性によって一階ペアノ算術の無矛盾性を保証するゲーデルのダイアレクティカ解釈にある．日常的な計算ならさておき，一階ペアノ算術の無矛盾性を保証するほどの大仕事となると，システム T_0 では不足で，システム T のフルパワーが必要になるということである．

単純型付きラムダ計算と直観主義命題論理の対応を思い出すと，たとえば証明中で $\to$ 導入が行われるたびに，対応するラムダ項の型は高階へと上がっていく．したがって，直観主義命題論理を解釈する段階で既に高階の概念が出現していると言える．システム T_0 や T がプレーンな単純型付きラムダ計算と異なる点は，自然数の型を持つこと，そして原始再帰法を利用可能なことである．自然数の型を用いて，自然数上の論理式について議論することが可能になり，そして原始再帰法によって数学的帰納法を解釈することが可能になる．

高階原始再帰算術のシステムによって形式自然数論を解釈する方法としては，ゲーデルのダイアレクティカ解釈とクライゼルの修正実現可能性 (modified realizability) が有名である．歴史的には，ゲーデルのダイアレクティカ解釈の方が古いが，クライゼルの修正実現可能性の方が単純なので，まずはクライゼルの修正実現可能性から説明しよう．ちなみに前者の名の由来としては，ゲーデルがこの研究報告をダイアレクティカ (Dialectica) という雑誌に出版したためであり，後者の名の由来は，その前段階として，型なしラムダ計算による形式自然数論の解釈であるクリーネの実現可能性解釈というものがあり，それを修正したものだからである．

7.1. クライゼルの実現可能性

算術の言語は，以下からなる．

定数記号 $0, 1$　　2 変数関係記号 $+, \cdot$　　2 変数関係記号 $\leqslant$

つまり，バッカス-ナウア記法を用いれば，まず，算術の項は，以下によって構成される．

$$t ::= x \mid 0 \mid 1 \mid t + t \mid t \cdot t$$

ここで，x は変数記号である．さらに，算術の論理式は，以下によって構成される．

$$\varphi ::= (t = t) \mid (t \leqslant t) \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \to \varphi \mid \forall x\varphi \mid \exists x\varphi$$

ここで，量化領域は自然数であるとする．クライゼルの修正実現可能性は，システム T の項 p と算術の論理式 φ の間の関係 $p \Vdash \varphi$ である．θ が原子論理式の場合，つまり $s = t$, $s \leqslant t$, $\bot$ の場合には，θ が真であるときは常に $p \Vdash \theta$ であり，さもなくば $p \Vdash \theta$ は成り立たないとする．原子論

理式以外の場合には，以下のように帰納的に定義される．

$$\begin{aligned}
\langle p, q\rangle \Vdash \varphi \wedge \psi &\iff (p \Vdash \varphi) \text{ and } (q \Vdash \psi)\\
\langle i, p\rangle \Vdash \varphi \vee \psi &\iff \texttt{if } i = 0 \texttt{ then } p \Vdash \varphi \texttt{ else } p \Vdash \psi\\
p \Vdash \varphi \to \psi &\iff (\forall a)\,[a \Vdash \varphi \implies pa \Vdash \psi]\\
p \Vdash \forall x\varphi(x) &\iff (\forall n\colon \mathtt{N})\, pn \Vdash \varphi(n)\\
\langle n, p\rangle \Vdash \exists x\varphi(x) &\iff p \Vdash \varphi(n)
\end{aligned}$$

直観主義論理のカリーハワード対応によって，直観主義論理で φ 証明可能であるならば，$p \Vdash \varphi$ となる項 p が存在することは予期できる．しかし，我々はいま，自然数論について考えている．ペアノは，自然数の本質とは数学的帰納法にあると考えた．数学的帰納法は，以下によって与えられる．

$$[\varphi(0) \wedge \forall k(\varphi(k) \to \varphi(k+1))] \to \forall n\varphi(n)$$

数学的帰納法の実現可能性は，以下によって説明できる．まず，前提部分の実現 $\langle p, q\rangle$ が与えられているとする．このとき，

$$p \Vdash \varphi(0) \qquad\qquad qk \Vdash \varphi(k) \to \varphi(k+1)$$

後者については，特に $a \Vdash \varphi(k)$ ならば $qka \Vdash \varphi(k+1)$ である．したがって，$\forall n\varphi(n)$ の実現プロセスは，初期値 p に $q0, q1, q2, \ldots$ を順に適用していく原始再帰に他ならない．実際，$\mathtt{rec}\ p\ q\ \underline{n}$ の計算プロセスを思い出すと以下のようになっていた．

$$\mathtt{rec}\ p\ q\,\underline{n+1} \longrightarrow q\underline{n}(\mathtt{rec}\ p\ q\ \underline{n}) \longrightarrow \ldots \longrightarrow q\underline{n}(q\underline{n-1}(\ldots q\underline{1}(q\underline{0}p)\ldots))$$

したがって，論理学的には，原始再帰子 rec は，まさに数学的帰納法を実現するものという役割を持つ．

$$\frac{p \Vdash \varphi(0) \qquad q \Vdash \forall k(\varphi(k) \to \varphi(k+1))}{\mathtt{rec}\ p\ q \Vdash \forall n\varphi(n)}$$

7.2. ダイアレクティカ解釈

ゲーデルのダイアレクティカ解釈は，含意 $\to$ の解釈を除けば，クライゼルの修正実現可能性とほとんど同じである．ただし，根本的な大きな違いは，算術の論理式 φ が与えられたとき，これを帰納的に，有限型汎関数パラメータの列 $\bar{g}, \bar{x}$ を含む**量化なし**論理式 $\varphi_D(\bar{g} \mid \bar{x})$ に変形するという点である．ここで，$\bar{g}$ と $\bar{x}$ の間の縦線は，パラメータの列 $\bar{g}$ と $\bar{x}$ の役割が異なることを強調するためだけのものであり，特に数学的な意味はない．

このアイデアを順を追って説明していこう．実現可能性とは，論理式 φ の正しさの証拠を具体的に記述することである．いま，φ の証拠を求める問題のことを「φ-実現問題」と呼ぶことにしよう．もし $u \Vdash \varphi$ であれば，u のことを φ-実現問題の解と呼ぶ．たとえば，全称量化について，$\forall x\varphi(x)$-実現問題とは，「与えられた n に対して，$\varphi(n)$ の証拠を与えよ」という問題である．この部分問題「$\varphi(n)$ の証拠を与えよ」を $\forall x\varphi(x)$-実現問題の「第 n 題」と呼ぶことにする．論理式中の量化が増

えると，第 $n_1, \ldots, n_k$ 題のような概念も出てくる．したがって，いま，各論理式 φ の実現問題は，有限個のパラメータ $\bar{u} = u_1, \ldots, u_k$ を伴っていると考える．このとき，「u が論理式 φ の実現問題の第 $\bar{u}$ 題の解である」ということを $u \Vdash \varphi[\bar{u}]$ と書くことにしよう．一旦，含意 $\to$ 以外に関して，帰納的定義を書き下しておくと，以下のようになる．

$$\begin{aligned}
\langle p, q\rangle \Vdash (\varphi \wedge \psi)[\bar{u}, \bar{v}] &\iff (p \Vdash \varphi[\bar{u}]) \text{ and } (q \Vdash \psi[\bar{v}]) \\
\langle i, p\rangle \Vdash (\varphi \vee \psi)[\bar{u}, \bar{v}] &\iff \texttt{if } i = 0 \texttt{ then } p \Vdash \varphi[\bar{u}] \texttt{ else } p \Vdash \psi[\bar{v}] \\
p \Vdash (\forall x \varphi(x))[a, \bar{u}] &\iff pa \Vdash \varphi(a)[\bar{u}] \\
\langle n, p\rangle \Vdash (\exists x \varphi(x))[\bar{u}] &\iff p \Vdash \varphi(n)[\bar{u}]
\end{aligned}$$

実際に φ-実現問題を解く際には，すべての題 $\bar{u}$ に対する部分問題 $\varphi[\bar{u}]$ の解 g を与えなければならない．つまり，$g\bar{u} \Vdash \varphi[\bar{u}]$ となっている必要がある．このアイデアを元に，論理式 φ_D を以下によって定義する．

$$\varphi_D(g \mid \bar{u}) \quad \equiv \quad g\bar{u} \Vdash \varphi[\bar{u}]$$

現時点では，この定義に量化記号が現れていないことに注意しよう．さて，含意 $\to$ の解釈も与えなければならないが，この定義が若干複雑なので，まずは否定 $\neg$ の解釈を与えることにする．アイデアとしては，$(\neg\varphi)$-実現問題は，「φ が実現不可能であることを示せ」という問題だと考えることである．つまり，「φ を実現する試み g が与えられたとき，その g の誤りを指摘せよ」ということである．誤りを指摘するというのは「g では解けない題 $\bar{u}$ を具体的に与える」ということである．以上をまとめると，形式的には，否定 $\neg\varphi$ の実現問題は以下によって定義される．

$$\bar{u} \Vdash (\neg\varphi)[g] \iff g\bar{u} \nVdash \varphi[\bar{u}]$$

それでは，本題となる含意 $\varphi \to \psi$ の解釈である．まずは，「φ-実現問題を解けるならば ψ-実現問題も解ける」というものを思いつくであろう．これを正確に定義するには，$\varphi \to \psi$-実現問題の第 $\bar{u}$ 題を考える必要があるが，これは「φ-実現問題の解法を知っていれば，$\psi[\bar{u}]$ を解ける」と考えるのが適切だろうか．このプロセスは，自身 Pro と相手 Opp の以下の対話を考えるとわかりやすい．

Opp: ψ-実現問題の第 $\bar{u}$ 題 $\psi[\bar{u}]$ を解くように指示する．
Pro: φ-実現問題の第 $\bar{v}$ 題 $\varphi[\bar{v}]$ の解を要求する．
Opp: $\varphi[\bar{v}]$ の解 p を提示する．
Pro: 以上の情報を参考にして，$\psi[\bar{u}]$ の解 q を提示する．

このプロセスは，形式的には，$r_-\colon \bar{u} \mapsto \bar{v}$ と $r_+\colon (\bar{u}, p) \mapsto q$ の対として表すことができる．したがって，この素朴なアイデアを基にして $\varphi \to \psi$ の実現可能性を定義すれば，以下のようになる．

$$\text{(案)} \quad \langle r_-, r_+\rangle \Vdash (\varphi \to \psi)[p, \bar{u}] \iff (p \Vdash \varphi[r_-\bar{u}] \text{ implies } r_+\langle \bar{u}, p\rangle \Vdash \psi[\bar{u}])$$

しかし，この定義にはいくつか問題がある．たとえば，明らかに正しい式 $\varphi \to \varphi \wedge \varphi$ を実現できない．なぜなら，$\varphi \wedge \varphi$-実現問題の第 $(\bar{u}, \bar{v})$ 題を解くためには，φ-実現問題の第 $\bar{u}$ 題と第 $\bar{v}$

題の両方を解く必要があるが，Opp に対しては 1 つの題に対する解しか要求できない．もちろん，$\varphi \to \psi$ の実現可能性の定義で，複数の題に対する解を要求できるように補正すれば，$\varphi \to \varphi \wedge \varphi$ については解決するが，結局，式 $\varphi \to \forall x\varphi$ においても同様の問題が発生し，どれだけの解を要求してもよいとすべきか明らかではない．

このため，ゲーデルは別の解決法を提示した．まず「φ-実現問題を解けるならば ψ-実現問題も解ける」という主張を解体すると，「φ-実現問題を解くための戦略 g が提示されたら，ψ-実現問題を解くための戦略 h を考案することができ，もし g が正しい解法ならば h も正しい解法である」という文に書き換えることができる．さらに，文の後半の対偶を取ると「h が誤っているならば，g も誤っている」あるいは「h で解けない題を提示されれば，g で解けない題を提示できる」と言ってもよい．自身 Pro と相手 Opp の以下の対話を考えるとわかりやすい．

Opp: φ-実現問題を解くための戦略 g を提示する．
Pro: その情報を参考にして，ψ-実現問題を解くための戦略 h を提示する．
Opp: h で解けない題 $\bar{u}$ を提示する．
Pro: その情報を参考にして，g で解けない題 $\bar{v}$ を提示する．

このプロセスは，形式的には，$r_-\colon g \mapsto h$ と $r_+\colon (g, \bar{u}) \mapsto \bar{v}$ の対として表すことができる．したがって，このアイデアを基にして $\varphi \to \psi$ の実現可能性を定義すれば，以下のようになる．

$$\langle r_-, r_+ \rangle \Vdash (\varphi \to \psi)[g, \bar{u}] \iff (r_- g\bar{u} \Vdash \psi[\bar{u}] \text{ implies } g\bar{u} \Vdash \varphi[r_+\langle g, \bar{u}\rangle])$$

事前に導入していた否定の実現問題を使って言い換えれば，

$$\langle r_-, r_+ \rangle \Vdash (\varphi \to \psi)[g, \bar{u}] \iff (\bar{u} \Vdash (\neg\psi)[r_- g] \text{ implies } r_+(g, \bar{u}) \Vdash (\neg\varphi)[\bar{u}])$$

ということである．あるいは，同値であるが，$\varphi_D(g \mid r_+\langle g, \bar{u}\rangle)$ ならば $\psi_D(r_- g \mid \bar{u})$ としても表すことができる．

定義 7.1 (ゲーデル)**.** 算術の論理式 φ に対し，有限型の汎関数の列を自由変数として持つ量化なし論理式 φ_D を以下のように定義する．まず，φ が原始論理式ならば，$\varphi_D \equiv \varphi$ とする．もし φ が原始論理式でないならば，φ_D を以下のように帰納的に定義する．

$$\begin{aligned}
(\varphi \wedge \psi)_D(\bar{g}, \bar{h} \mid \bar{x}, \bar{y}) &\equiv \varphi_D(\bar{g} \mid \bar{x}) \wedge \psi_D(\bar{h} \mid \bar{y}) \\
(\varphi \vee \psi)_D(i, \bar{g}, \bar{h} \mid \bar{x}, \bar{y}) &\equiv [i = 0 \wedge \varphi_D(\bar{g} \mid \bar{x})] \vee [i = 1 \wedge \psi_D(\bar{h} \mid \bar{y})] \\
(\forall z\varphi(z))_D(f \mid a, \bar{x}) &\equiv \varphi(a)_D(f(a) \mid \bar{x}) \\
(\exists z\varphi(z))_D(a, \bar{g} \mid \bar{x}) &\equiv \varphi(a)_D(\bar{g} \mid \bar{x}) \\
(\varphi \to \psi)_D(h, k \mid \bar{g}, \bar{x}) &\equiv [\varphi_D(\bar{g} \mid h(\bar{g}, \bar{x})) \to \psi_D(k(\bar{g}) \mid \bar{x})]
\end{aligned}$$

このようにして，φ から量化なし論理式 $\varphi_D(\bar{g} \mid \bar{x})$ を作る．高階汎関数の列 $\bar{g}$ が任意の $\bar{x}$ に対して $\varphi_D(\bar{g} \mid \bar{x})$ を満たすとき，$\bar{g}$ によって φ の正しさが保証されていると考える．このように，

高階汎関数を用いた φ の正しさの保証を表す論理式

$$\varphi^D \equiv \exists \bar{g} \forall \bar{x} \varphi_D(\bar{g} \mid \bar{x})$$

を φ の**ダイアレクティカ解釈** (*Dialectica interpretation*) と呼ぶ.

包含関係の解釈として，ただ対偶を取って補正をしただけという印象を覚えるかもしれないが，不思議とこれで上手くいく．いくつかの具体例を見ていこう．

例 7.2. $\varphi \wedge \psi \to \varphi$ を解くための戦略は以下によって与えられる．

Opp: $\varphi \wedge \psi$ に対する戦略 $\alpha = (\alpha_0, \alpha_1)$ を提示する．
Pro: このうち φ に対する戦略 α_0 をシミュレートする．
Opp: 戦略 α_0 で解けない φ-題 u_0 を提示する．
Pro: 適当に u_1 を選んで $\varphi \wedge \psi$ の第 (u_0, u_1) 題を提示すれば，これは戦略 α では解けない．

例 7.3. $\varphi \to \varphi \vee \psi$ を解くための戦略は以下によって与えられる．

Opp: φ に対する戦略 α を提示する．
Pro: このとき，$\varphi \vee \psi$ に対する戦略として，$\alpha' = (0, \alpha)$ を提示する．
Opp: 戦略 α' で解けない $\varphi \vee \psi$-題 (u, v) を提示する．
Pro: このとき，φ の第 u 題を提示すれば，これは戦略 α では解けない．

例 7.4. $\varphi \to \varphi \wedge \varphi$ を解くための戦略は以下によって与えられる．

Opp: φ に対する戦略 α を提示する．
Pro: このとき，$\varphi \wedge \varphi$ に対する戦略として，$\alpha' = (\alpha, \alpha)$ を提示する．
Opp: 戦略 α' で解けない $\varphi \wedge \varphi$-題 (u, v) を提示する．
Pro: 戦略 α では φ の第 u 題または第 v 題を解けないので，解けない方の題を提示する．

例 7.5. $\varphi \vee \varphi \to \varphi$ を解くための戦略は以下によって与えられる．

Opp: $\varphi \vee \varphi$ に対する戦略 $\alpha = (i, \alpha')$ を提示する．
Pro: このとき，φ に対する戦略として，α' を提示する．
Opp: 戦略 α' で解けない φ-題 u を提示する．
Pro: このとき，$\varphi \vee \varphi$ の第 u 題を提示すれば，これは戦略 α では解けない．

例 7.6. $\varphi \wedge (\varphi \to \psi) \to \psi$ を解くための戦略は以下によって与えられる．

Opp: $\varphi \wedge (\varphi \to \psi)$ に対する戦略 $\alpha = (\beta, (\gamma_-, \gamma_+))$ を提示する．つまり，β が φ に対する戦略であり，$\gamma = (\gamma_-, \gamma_+)$ は $\varphi \to \psi$ に対する戦略で，以下の動作を目指す．
Opp′: φ に対する戦略 δ を提示する．

Pro′: このとき，ψ に対する戦略として，$\gamma_-(\delta)$ を提示する．

Opp′: 戦略 $\gamma_-(\delta)$ で解けない ψ-題 w を提示する．

Pro′: このとき，φ の第 $\gamma_+(\delta, w)$ 題を提示すれば，これは戦略 δ では解けない．

Pro: このとき，ψ に対する戦略として，上記の部分戦略を利用して，$\gamma_-(\beta)$ を提示する．

Opp: 戦略 $\gamma_-(\beta)$ で解けない ψ-題 u を提示する．

Pro: このとき，δ は φ の第 $\gamma_+(\beta, u)$ 題を解けない，または γ は $\varphi \to \psi$ の第 (β, u) 題を解けない．

(∵) もし後者が成り立たないとすると，γ が $\varphi \to \psi$ の第 (β, u) 題を解けるということは，Opp′ の手が順に β, u であるとき，Pro′ の戦略 γ によって勝利できるということである．Opp の手 β と u の性質より，これらは Opp′ の手としても認められるので，これはつまり Pro′ の戦略 γ の性質より，δ が φ の第 $\gamma_+(\beta, u)$ 題を解けないということである．

したがって，α は $\varphi \wedge (\varphi \to \psi)$ の第 $(\gamma_+(\beta, u), (\beta, u))$ 題を解けない．

例 7.7 (数学的帰納法のダイアレクティカ解釈)**.** 上記のようにして，複雑な量化の入れ子構造を持ち得る算術的論理式は，有限型汎関数を変数として持つ**量化なし**論理式に翻訳される．この解釈の下で，数学的帰納法の正当性は，有限型の原始再帰に還元されるということを見てみよう．数学的帰納法は，形式的には以下の論理式によって表される．

$$[\varphi(0) \wedge \forall n(\varphi(n) \to \varphi(n+1))] \to \forall n \varphi(n).$$

論理式 φ は大量の量化記号を含み得ることに注意する．いま，数学的帰納法の前提の正しさがダイアレクティカ解釈によって保証されていると仮定する．つまり，ある $\bar{f}$ について，任意の $\bar{x}$ で $(\varphi(0))_D(\bar{f} \mid \bar{x})$ が成立しており，さらに，ある h, k について，任意の $\bar{y}$ で以下が成立している．

$$(\varphi(n))_D(\bar{g} \mid h(n, \bar{g}, \bar{y})) \to (\varphi(n+1))_D(k(n, \bar{g}) \mid \bar{y}).$$

以下のような高階原始再帰で，$(\forall n \varphi(n))_D$ の解を見つけられる．

$$\begin{cases} K(0) = \bar{f} \\ K(n+1) = k(n, K(n)) \end{cases}$$

これが実際に $(\forall n \varphi(n))_D$ の解となっていることを，**量化なし**論理式に対する数学的帰納法によって保証しよう．いま，K の定義より，

$$(\varphi(0))_D(K(0) \mid \bar{x}) \ \wedge \ \big[(\varphi(n))_D(K(n) \mid h(n, K(n), \bar{y})) \to (\varphi(n+1))_D(K(n+1) \mid \bar{y})\big]$$

が成立している．与えられた $m \in \mathbb{N}$ について，高階原始再帰によって以下のように H を定義する．

$$\begin{cases} H(0, \bar{y}) = \bar{y} \\ H(n+1, \bar{y}) = h(m \dot{-} n, K(m \dot{-} n), H(n, \bar{y})) \end{cases}$$

これを利用すると，**量化なし**論理式に対する数学的帰納法によって，式 $(\varphi(m))_D(K(m) \mid \bar{y})$ の正しさを $(\varphi(0))_D(K(0) \mid H(m, \bar{y}))$ の正しさに還元できる．後者の式は，前提によって保証されて

いるから，以上により数学的帰納法の帰結のダイアレクティカ解釈 $(\forall n\varphi(n))_D$ が正しいことが確認された．

以上より，数学的帰納法の正当性が有限型原始再帰関数によって保証されることが確認できた．より一般に，直観主義論理上の一階算術として知られる**ハイティング算術** (*Heyting arithmetic*) がある算術的文を証明するならば，ダイアレクティカ解釈より，有限型原始再帰関数がその正しさを保証してくれることを証明できる．これは，無矛盾性証明の観点からは，重要な帰結を持つ．もしハイティング算術が矛盾するならば，ハイティング算術による矛盾の証明の正しさを有限型原始再帰関数が保証してしまう．これを言い換えれば，ハイティング算術が矛盾するならば，有限型原始再帰算術の体系 T も矛盾している，ということに他ならない．つまり，ゲーデルのダイアレクティカ解釈の帰結として，

有限型原始再帰の体系 T が無矛盾ならば，ハイティング算術も無矛盾である

という定理が得られる．一方で，ゲーデル-ゲンツェンの二重否定解釈というものを用いると，ペアノ算術の無矛盾性をハイティング算術の無矛盾性へと容易に還元できる．したがって，上の定理は，実際には，「有限型原始再帰の体系 T が無矛盾ならば，ペアノ算術も無矛盾である」ということを導く．

ペアノ算術やハイティング算術が公理として認めている**数学的帰納法**は，とてつもなく複雑な量化の入れ子構造を伴う論理式に対しても適用可能であった．たった 1 つの量化記号ですら，計算不可能な関数や述語を容易に定義できてしまうし，複数の量化記号を用いると想像を絶するような状況が起こるかもしれない．このように，とてつもなく複雑な量化を伴った数学的帰納法の無矛盾性を，ダイアレクティカ解釈によって，有限型の高階原始再帰（あるいは現代的な観点からは，原始的な高階プログラミングと呼べるものかもしれない）という，量化のない世界に還元した．

さらに，第**??**節の議論によれば，有限型の原始再帰は，ε_0 未満の順序型の整礎原始再帰によってシミュレートできる．ここで注意すれば，順序数とは，整礎性というある種の有限性を持つ順序構造（の同値類）に過ぎず，そういう意味では ε_0 などは自然数概念を超えるものではないとも言える．具体的には，ε_0 未満の順序型は，あくまで $\mathbb{N}$ 上の辞書式順序に過ぎない．つまり，ε_0 未満の順序型は，極めて単純な算術的論理式によって定義できる $\mathbb{N}$ 上の（原始再帰的）関係である．また，有限型の原始再帰を ε_0 未満の順序型の整礎原始再帰へ還元する過程において，本質的な量化は一切現れない．つまるところ，ペアノ算術という極めて複雑な体系の無矛盾性は，最終的に，本質的な量化を伴わない ε_0 上の整礎帰納法という比較的単純な操作によって保証されるのである．

$$\boxed{\text{ペアノ算術の無矛盾性}} \Longleftarrow \boxed{\text{有限型原始再帰算術 } T \text{ の無矛盾性}} \Longleftarrow \boxed{\varepsilon_0 \text{ の整礎性}}$$

量化記号という概念が如何に複雑怪奇なものであるか，ということをペアノ算術を階層化することによっても理解できる．ペアノ算術は，任意の算術的論理式に対する数学的帰納法を公理として保有していた．算術的論理式のうち，量化記号を高々 n 個しか用いず，その先頭の量化記号が $\exists$ である式を Σ_n-論理式と呼ぶ．体系 $\mathrm{I}\Sigma_n$ とは，ペアノ算術における数学的帰納法の公理を Σ_n-論理

式に制限したものである．

$$\mathrm{I}\Sigma_1 \subset \mathrm{I}\Sigma_2 \subset \mathrm{I}\Sigma_3 \subset \cdots \subset \mathrm{I}\Sigma_n \subset \mathrm{I}\Sigma_{n+1} \subset \cdots \subset \text{ペアノ算術}$$

体系 $\mathrm{I}\Sigma_n$ の無矛盾性証明のためには，$\omega \uparrow\uparrow (n+1)$ の整礎性があれば十分であることが知られている．一方で，$\mathrm{I}\Sigma_n$ は，$\omega \uparrow\uparrow (n+1)$ 未満の任意の順序数の整礎性を証明できる．したがって，体系 $\mathrm{I}\Sigma_{n+1}$ によって $\mathrm{I}\Sigma_n$ の無矛盾性を証明できる．ここでゲーデルの第 2 不完全性定理より，$\mathrm{I}\Sigma_n$ が無矛盾ならば，$\mathrm{I}\Sigma_n$ によって $\mathrm{I}\Sigma_n$ の無矛盾性は証明できないことに注意しよう．これによって，$(n+1)$ 量化論理式に対する数学的帰納法の体系 $\mathrm{I}\Sigma_{n+1}$ が n 量化論理式に対する数学的帰納法の体系 $\mathrm{I}\Sigma_n$ より遥かに強いことが理解できる．このようにして，数学的帰納法を適用する式に現れる量化記号の数によって，その複雑さは著しく変化するのである．

参考文献

[1] 大堀 淳, プログラミング言語の基礎理論, 共立出版, 新装版, 2019.

[2] 田中 一之 (編), ゲーデルと 20 世紀の論理学 3 –不完全性定理と算術の体系–, 東京大学出版会, 2007.

[3] 照井 一成, コンピュータは数学者になれるのか? –数学基礎論から証明とプログラムの理論へ–, 青土社, 2015.

[4] 萩谷 昌己, 西崎 真也, 論理と計算のしくみ, 岩波書店, 2007.

[5] 藤田 憲悦, 数理パズルで楽しく学べる論理学, コロナ社, 2022.

[6] J. Roger Hindley, and Jonathan P. Seldin, Lambda-Calculus and Combinators: An Introduction, Cambridge University Press, 2008.

[7] P. Odifreddi, Classical Recursion Theory, Volume II, North Holland, 1999.

[8] H. E. Rose, Subrecursion: Functions and Hierarchies, Clarendon Pr, 1984.

[9] Morten Heine Sørensen, Pawel Urzyczyn, Lectures on the Curry-Howard Isomorphism, Elsevier Science, 2006.