

q -類似の Coq による形式化

アドバイザー：Jacques Garrigue 教授

学籍番号：322101289

氏名：中村 薫

2023 年 1 月 8 日

目次

第 1 章	修士論文	4
1.1	q -類似	4
1.1.1	q -微分について成り立つ諸性質	4
1.1.2	無限和に関する話題	6
1.1.3	q -類似を考える利点	6
1.2	型付き λ 計算	6
1.2.1	λ 計算	6
1.2.2	$\lambda \rightarrow$ と $\lambda 2$	8
1.3	Coq	15
1.3.1	λ -cube と CIC	15
1.3.2	Coq の使い方	15
1.3.3	mathcomp の型	20
1.4	形式化	20
1.4.1	q -微分の定義	20
1.4.2	$(x - a)^n$ の q -類似	22
1.4.3	関数から多項式へ	29
1.4.4	q -Taylor 展開	32
第 2 章	少人数クラスまとめ	36
2.1	はじめに	36
2.2	型から型を作る	36
2.3	型の同型	37
2.4	Univalence axiom	38
2.5	関数の外延性	38
2.6	可縮, ファイバー	39
2.7	レトラクト	41
2.8	Univalence implies function extensionality	42
参考文献		45

序文

本論文の主結果は、 q -類似の初等的な結果を Coq によって形式化するものである。具体的には Victor Kac, Pokman Cheung の *Quantum Calculus* [1] の 4 章 (4.1) 式の q -Taylor 展開, 及びその系として得られる Gauss's binomial formula の形式化を目標としている。本論文での q -類似に関する定義や定理, 証明は [1] によるものだが, その形式化を行ったという点において独自性がある。形式化したコード全体は <https://github.com/nakamurakaoru/q-analogue> [2] にある。q_analogue.v が [1] の形式化をしたファイルであり,

q_tool.v は直接 q -類似に関係はしないが, 形式化をするために自分で用意した補題をまとめたファイルである.

q -類似は, 学部 4 年次に卒業研究のテーマとして扱ったものである. 実数パラメータ q , 実数上の関数 f に対して

$$D_q f(x) := \frac{f(qx) - f(x)}{(q-1)x}$$

で定義される q -微分を出発点とし, この q -微分に対してうまく振る舞い, かつ q を極限で 1 に近づけると通常の定義に一致するように数学の諸概念を一般化するものである. 例えば, 自然数 n について x^n を定義に沿って q -微分すると,

$$D_q x^n = \frac{(qx)^n - x^n}{(q-1)x} = \frac{q^n - 1}{q-1} x^{n-1}$$

となる. 通常の微分では, $(x^n)' = nx^{n-1}$ となることと比較して, n の q -類似 $[n]$ を

$$[n] = \frac{q^n - 1}{q - 1}$$

と定める. また, $(x-a)^n$ の q -類似は,

$$(x-a)_q^n := \begin{cases} 1 & (n=0) \\ (x-a)(x-qa) \cdots (x-q^{n-1}a) & (n \geq 1) \end{cases}$$

と定義することで, q -微分と自然数の q -類似に対してうまく振る舞う, つまり

$$D_q (x-a)_q^n = [n](x-a)_q^{n-1}$$

が成り立つ. 更に, 階乗と二項係数の q -類似をそれぞれ

$$[n]! := \begin{cases} 1 & (n=0) \\ [n] \times [n-1] \cdots [1] & (n \geq 1) \end{cases}$$

$$\begin{bmatrix} n \\ j \end{bmatrix} := \frac{[n]!}{[j]![n-j]!}$$

で定めれば, q -Taylor 展開, Gauss's binomial formula は以下のように書ける.

— q -Taylor 展開 —

$f(x)$ を N 次の実係数多項式とする. 任意の $c \in \mathbb{R}$ について

$$f(x) = \sum_{j=0}^N (Dq^j f)(c) \frac{(x-c)_q^j}{[j]!}$$

が成り立つ.

— Gauss's binomial formula —

$x, a \in \mathbb{R}, n \in \mathbb{Z}_{>0}$ について

$$(x+a)_q^n = \sum_{j=0}^n \begin{bmatrix} n \\ j \end{bmatrix} q^{\frac{j(j-1)}{2}} a^j x^{n-j}$$

が成り立つ.

この 2 つを形式化することが本論文の主目的である.

Coq とは, 定理証明支援系の 1 つであり, 数学的な証明が論理学の推論規則に沿って正しく書かれているかどうか判定するプログラムである. 人間がチェックすることが難しい複雑な証明でも正しさが保証され, また証明付きプログラミングにも応用される. Mizar, Isabelle/HOL 等他にも定理証明支援系は存在するが, 修士 1 年次後期に履修した授業で Coq の使い方を学んだため, 今回の形式化に利用した. Coq は型付き λ 計算という理論

に基づいている. この λ 計算については修士 1 年次に少人数クラスで学習した内容であるため, Coq との関係にも触れつつ本論文でも説明を加える. 今回の証明に関しては, Coq の標準ライブラリ [4] に加えて, 数学の証明のために整備されたライブラリ群である mathcomp [5] も用いている. 実際に Coq が用いられた有名な例として, 四色定理や Feit-Thompson の定理 (奇数位数定理) などがある.

今後の展望としては, まずはこれまでに形式化した q -類似の各概念が $q \rightarrow 1$ としたときに通常の数学の概念に一致することの形式化を行いたい. このためには, 現在開発中のライブラリである mathcomp analysis [6] を用いる必要がある. また, このライブラリを用いると無限和に関する形式化も可能であるため, Gauss's binomial formula を無限に拡張したものや, 無限和を用いて定義される指数関数, 三角関数の q -類似の形式化にも挑戦していきたい.

最後に構成について述べる. まず 1.1 節で q -類似の概要について説明し, 1.2 節で修士 1 年次に少人数クラスで学習した H.P.Barendregt の *Lambda Calculi with Types* [8] に沿って, 型付き λ 計算について述べる. 1.3 節は Coq の概要についてで, 特に 1.3.2 節で Coq や mathcomp の使い方を具体例を交えて説明するが, より詳細な情報については萩原 学/アフェルト・レナルドの *Coq/SSReflect/Mathcomp* [3] 等を参照のこと. これらの準備のもと, 1.4 節から本題の形式化に入る. [1] での定義, 定理を述べた後, その形式化を与え, 必要であれば形式化をするにあたっての注意点を述べることを繰り返すという流れである. 証明の方針等は基本的に [1] の通りであるが, 1.4.3 節では一部 [1] から離れ, 多項式として q -微分や q -二項式を定義しなおして形式化を行っている. これらの新たな定義が多項式に対してのものと定義を適用したものと一致していることの証明も行っている. また, 2 章で少人数クラスで学習した内容についてまとめている.

第 1 章

修士論文

1.1 q -類似

本節では, 形式化について説明する 1.4 節では詳細には立ち入らない q -類似の性質や, q -Taylor 展開以降で [1] で取り上げられている内容について述べる. 1.1.2, 1.1.3 節は無限和に関するものであるため形式化はできていない.

1.1.1 q -微分について成り立つ諸性質

通常の微分は線形作用素であり, また積の微分法や商の微分法などの性質が成り立つ. これらの性質は q -微分でもどうなるのであろうか. まずは線形性について見ていく. $a, b \in \mathbb{R}$, 実数上の関数 f, g について,

$$\begin{aligned} D_q(af(x) + bg(x)) &= \frac{(af(qx) + bf(qx)) - (af(x) + bf(x))}{(q-1)x} \\ &= \frac{af(qx) - af(x)}{(q-1)x} + \frac{bg(qx) - bg(x)}{(q-1)x} \\ &= a \frac{f(qx) - f(x)}{(q-1)x} + b \frac{g(qx) - g(x)}{(q-1)x} \\ &= aD_qf(x) + bD_qg(x) \end{aligned}$$

と計算できるので, 通常の微分と変わらない形で成り立つ ([1] p2). 一方で積の微分法はそのままの形では成り立たない. このことを確認するために, まず q -差分

$$d_qf(x) = f(qx) - f(x)$$

が積についてどう振る舞うかを観察する. 実数上の関数 f, g について, 実際に計算してみると, 以下のように計算できる.

$$\begin{aligned} d_q(f(x)g(x)) &= f(qx)g(qx) - f(x)g(x) \\ &= f(qx)g(qx) - f(qx)g(x) + f(qx)g(x) - f(x)g(x) \\ &= f(qx)d_qg(x) + g(x)d_qf(x) \end{aligned}$$

ここで, q -微分は q -差分を用いて

$$D_qf(x) = \frac{d_qf(x)}{d_qx}$$

と表せることを用いると ([1] では順番が逆で, 先に q -微分を定義している),

$$\begin{aligned} D_q(f(x)g(x)) &= \frac{d_q(f(x)g(x))}{(q-1)x} \\ &= \frac{f(qx)d_qg(x) + g(x)d_qf(x)}{(q-1)x} \\ &= f(qx)\frac{d_qg(x)}{(q-1)x} + g(x)\frac{d_qf(x)}{(q-1)x} \end{aligned}$$

となるので, 積の q -微分法は

$$D_q(f(x)g(x)) = f(qx)D_qg(x) + g(x)D_qf(x) \quad ([1] \text{ p3 (1.11)})$$

となる. また, $f(x)g(x) = g(x)f(x)$ から, f と g を入れ替えることで

$$D_q(f(x)g(x)) = f(x)D_qg(x) + g(qx)D_qf(x) \quad ([1] \text{ p3 (1.12)})$$

も得られる. 次に, 商の微分法については,

$$g(x) \cdot \frac{f(x)}{g(x)} = f(x)$$

という等式の両辺を q -微分し, 左辺について 1 つ目の積の q -微分法を適用することで

$$g(qx)D_q\left(\frac{f(x)}{g(x)}\right) + \frac{f(x)}{g(x)}D_qg(x) = D_qf(x)$$

となるので, 式変形することで商の q -微分法

$$D_q\left(\frac{f(x)}{g(x)}\right) = \frac{g(x)D_qf(x) - f(x)D_qg(x)}{g(x)g(qx)} \quad [1] \text{ p3 (1.13)}$$

が得られる. 2 つ目の積の q -微分法を適用すれば, もう一つの商の微分法

$$D_q\left(\frac{f(x)}{g(x)}\right) = \frac{g(qx)D_qf(x) - f(qx)D_qg(x)}{g(x)g(qx)} \quad [1] \text{ p3 (1.14)}$$

が得られる. 更に, 合成関数の q -微分法については一般的なルールは存在せず, $u = u(x) = \alpha x^\beta$ のときについてのみ

$$D_qf(u(x)) = (D_q^\beta f)(u(x)) \cdot D_qu(x) \quad ([1] \text{ p4 (1.15)})$$

が成り立つことが知られている (証明は [1] p3, p4 を参照).

これらの性質はすべて形式化できており, D_q の線形性, 積の q -微分法の 1 つ目・2 つ目, 商の q -微分法の 1 つ目・2 つ目, 合成関数の q -微分法はそれぞれこの順に `Dq_is_linear`, `Dq_prod`, `Dq_prod'`, `Dq_quot`, `Dq_quot'`, `qchain` という補題名で [6] の `q-analogue.v` に登録してある.

また, 今回の主目的である q -Taylor 展開には利用しないが, 二項係数の q -類似について, 通常二項係数において Pascal の法則が成り立つことの q -類似版として, $n > j$ をみたす 1 以上の整数 n, j について

$$\begin{bmatrix} n \\ j \end{bmatrix} = \begin{bmatrix} n-1 \\ j-1 \end{bmatrix} + q^j \begin{bmatrix} n-1 \\ j \end{bmatrix} \quad ([1] \text{ p17 (6.2)})$$

が成り立つことが証明されている ([1] p18 参照). この補題も形式化できており, [2] の `q-analogue.v` の `q_pascal` という補題である.

1.1.2 無限和に関する話題

本節と次節の内容は形式化できていない. Gauss's binomial formula において, $x = 1$, $a = x$ とすれば

$$(1+x)_q^n = \sum_{j=0}^n q^{j(j-1)/2} \begin{bmatrix} n \\ j \end{bmatrix} x^j$$

となる. ここで $|q| < 1$ とすると,

$$\lim_{n \rightarrow \infty} \begin{bmatrix} n \\ j \end{bmatrix} = \lim_{n \rightarrow \infty} \frac{(1-q^n)(1-q^{n-1}) \cdots (1-q^{n-j+1})}{(1-q)(1-q^2) \cdots (1-q^j)} = \frac{1}{(1-q)(1-q^2) \cdots (1-q^j)}$$

となるので,

$$(1+x)_q^\infty = \sum_{j=0}^{\infty} q^{j(j-1)/2} \frac{x^j}{(1-q)(1-q^2) \cdots (1-q^j)}$$

無限積を無限和に書き換える式が得られる. この式は Euler により見つけられたため, Euler's first identity の意で [1] では E1 と呼ばれている.

1.1.3 q -類似を考える利点

あえてパラメータを増やす q -類似を考える利点の一つとしては, 証明が複雑な定理に対してより簡単な別証明を与えられる場合があることである. 例えば, Jacobi の三重積 ([1] p35 Theorem 11.1)

$z, q \in \mathbb{R}, |q| < 1$ として,

$$\sum_{n=-\infty}^{\infty} q^{n^2} z^n = \prod_{n=1}^{\infty} (1-q^{2n})(1+q^{2n-1}z)(1+q^{2n-1}z^{-1})$$

が成り立つ.

はその一例である. 楕円関数論の文脈で登場する恒等式であるが ([7] p144 (3.47) 等を参照), q -類似で得られる式

$$(1+x)_q^\infty = \sum_{j=0}^{\infty} q^{j(j-1)/2} \frac{x^j}{(1-q)(1-q^2) \cdots (1-q^j)} \quad ([1] \text{ p30 (9.3) 式})$$

$$\frac{1}{(1-x)_q^\infty} = \sum_{j=0}^{\infty} \frac{x^j}{(1-q)(1-q^2) \cdots (1-q^j)} \quad ([1] \text{ p30 (9.4) 式})$$

を用いることで簡単に証明できる.

1.2 型付き λ 計算

1.2.1 λ 計算

まず型のない λ 計算を定義する. 初めに, λ 計算がどのようなものなのかについての概要を説明し, その後厳密な定義に移る.

λ 計算の概要

λ 計算には, 抽象と適用の 2 つの基本的な操作がある. まず, 抽象については, 「式から関数を作る操作」と捉えることができる. M を λ 計算における式 (λ 計算においてはこれを λ 項と呼ぶ) だとすると,

$$\lambda x.M$$

で、「 x を変数とする関数」を表すことになる。例えば、 M が $x^2 + 3xy + 4$ という式であれば、

- $\lambda x.(x^2 + 3xy + 4)$
- $\lambda xy.(x^2 + 3xy + 4)$
- $\lambda z.(x^2 + 3xy + 4)$

はそれぞれ、1 つ目は $x \mapsto (x^2 + 3xy + 4)$ という x についての 2 次関数、2 つ目は $(x, y) \mapsto (x^2 + 3xy + 4)$ という x, y についての 2 変数関数を表す。3 つ目は、 $(x^2 + 3xy + 4)$ は変数 z を含まないので、定数関数を表すことになる。

もう 1 つの操作である適用は、2 つの λ 項 M と N を並べて、

$$MN$$

と書かれ、直観的には「関数 M に値 N を代入する」ことを示している。例えば、 M が $\lambda x.(3x + 2)$ 、 N が 4 であれば、

$$(\lambda x.(3x + 2)) 4 = 3 \cdot 4 + 2 (= 14)$$

となる。一般には、 $[x := N]$ で x に N を代入することを表すとして、

$$(\lambda x.M) N = M[x := N]$$

と書く。

λ 計算の定義

ここから実際の λ 計算の定義に入る。

Definition 1.2.1.1 ([8] Definition 2.1.1) $V = \{v, v', v'', \dots\}$ を無限個の変数の集合とする。 λ 項全体の集合 Λ を、以下のように帰納的に定義する。

$$\begin{aligned} x \in V & \implies x \in \Lambda \\ M, N \in \Lambda & \implies (MN) \in \Lambda \\ M \in \Lambda, x \in V & \implies (\lambda x.M) \in \Lambda \end{aligned}$$

抽象構文を用いると、以下のようにも書ける。

$$\begin{aligned} V &::= v \mid V' \\ \Lambda &::= V \mid (\Lambda\Lambda) \mid (\lambda V\Lambda) \end{aligned}$$

Example 1.2.1.2 $v, (vv'), \lambda v(vv'), ((\lambda v(vv''))v')$ などが λ 項の例である。

Remark 1.2.1.3 表記上のルールや省略を以下のように約束する。

1. x, y, z, \dots で任意の変数を、 M, N, L, \dots で任意の λ 項を表す。
2. $FM_1 \cdots M_n$ で $(\cdots((FM_1)M_2) \cdots M_n)$ を、 $\lambda x_1 \cdots x_n.M$ で $(\lambda x_1(\lambda x_2(\cdots(\lambda x_n(M)) \cdots)))$ を表す。
3. 最も外側の $()$ は書かない。

Definition 1.2.1.4 ([8] Notation 2.1.4, Definition 2.1.5) $M, N, P, Q \in \Lambda$, $x, y \in V$ とする.

1. M の自由変数の集合 $FV(M)$ を, 以下のように帰納的に定義する.

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) - \{x\} \end{aligned}$$

2. M に現れる変数のうち, 自由変数でないものを束縛変数と呼ぶ. また, M と N が等しいか, 束縛変数の名前をつけかえることで互いにうつりあうとき, $M \equiv N$ と書く.
3. M において x に N を代入した結果 $M[x := N]$ を, 以下のように帰納的に定義する.

$$\begin{aligned} y[x := N] &\equiv \begin{cases} N & x = y \\ y & x \neq y \end{cases} \\ (PQ)[x := N] &\equiv (P[x := N])(Q[x := N]) \\ (\lambda y.M)[x := N] &\equiv \begin{cases} \lambda y.(M[x := N]) & x \neq y \\ \lambda y.M & x = y \end{cases} \end{aligned}$$

Remark 1.2.1.5 ([8] Notation 2.1.4) 2. について, 例えば,

$$\begin{aligned} (\lambda x.x)z &\equiv (\lambda x.x)z \\ (\lambda x.x)z &\equiv (\lambda y.y)z \\ (\lambda x.x)z &\not\equiv (\lambda x.y)z \end{aligned}$$

となる.

Remark 1.2.1.6 束縛変数は, 自由変数とは異なる名前とする. 例えば, $y(\lambda y.xy)$ は $y(\lambda y'.xy')$ などに書き換える.

最後に, β -簡約を定義する.

Definition 1.2.1.7 ([8] Definition 2.3.2) Λ 上の二項関係 \rightarrow_β (1 ステップ β -簡約), \twoheadrightarrow_β (β -簡約) を, 以下のように帰納的に定義する.

1. (a) $(\lambda x.M)N \rightarrow_\beta M[x := N]$
(b) $M \rightarrow_\beta N \implies ZM \rightarrow_\beta ZN, MZ \rightarrow_\beta NZ, \lambda x.M \rightarrow_\beta \lambda x.N$
2. (a) $M \twoheadrightarrow_\beta M$
(b) $M \rightarrow_\beta N \implies M \twoheadrightarrow_\beta N$
(c) $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \implies M \twoheadrightarrow_\beta L$

Example 1.2.1.8 1. $(\lambda x.xyz)y \rightarrow_\beta yyz$ である.

2. $(\lambda x.xy)(\lambda z.z) \rightarrow_\beta (\lambda z.z)y \rightarrow_\beta y$ より, $(\lambda x.xy)(\lambda z.z) \twoheadrightarrow_\beta y$ である.

1.2.2 $\lambda \rightarrow$ と $\lambda 2$

このセクションでは, 単純型付きラムダ計算である $\lambda \rightarrow$ と, その拡張である $\lambda 2$ を導入する. $\lambda \rightarrow$ の定義の仕方には, Curry 流と Church 流の 2 通りがあるが, 本稿においては Curry 流を採用する.

$\lambda \rightarrow$

Definition 1.2.2.1 ([8] Definition 3.1.1) $\lambda \rightarrow$ の型の集合 $\text{Type}(\lambda \rightarrow)$ は, 以下のように帰納的に定義される.
 $\mathbb{T} = \text{Type}(\lambda \rightarrow)$ と書くことにする.

$$\alpha, \alpha', \dots \in \mathbb{T} \quad (\text{型変数})$$

$$\sigma, \tau \in \mathbb{T} \Rightarrow (\sigma \rightarrow \tau) \in \mathbb{T} \quad (\text{関数型})$$

抽象構文を用いれば,

$$\mathbb{T} ::= \mathbb{V} \mid \mathbb{V} \rightarrow \mathbb{V}$$

と書ける. ここで, \mathbb{V} は,

$$\mathbb{V} ::= \alpha \mid \mathbb{V}'$$

である.

Remark 1.2.2.2 ([8] Notation 3.1.2)

1. $\sigma_1, \dots, \sigma_n \in \mathbb{T}$ のとき,

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \dots \rightarrow \sigma_n$$

と書いた場合,

$$(\sigma_1 \rightarrow (\sigma_2 \rightarrow \dots \rightarrow (\sigma_{n-1} \rightarrow \sigma_n) \dots))$$

を表すものとする (右結合).

2. $\alpha, \beta, \gamma, \dots$ で型変数を表す.

Definition 1.2.2.3 ([8] Definition 3.1.3)($\lambda \rightarrow$ -Curry)

1. $M \in \Lambda, \sigma \in \mathbb{T}$ のとき, $M: \sigma$ の形を statement(文) という. このとき, 型 σ を文の predicate (述語), 項 M を文の subject (主語) という.
2. 変数 (項) を主語とする文を declaration (宣言) という.
3. 異なる変数を主語とする宣言の集合を basis (基底) という.

Definition 1.2.2.4 ([8] Definition 3.1.4) 文 $M: \sigma$ が基底 Γ から導出される ($\Gamma \vdash_{\lambda \rightarrow \text{Curry}} M: \sigma$ or $\Gamma \vdash_{\lambda \rightarrow} M: \sigma$ or $\Gamma \vdash M: \sigma$ と書く) のは, $\Gamma \vdash M: \sigma$ が以下のルールから導かれるときである.

導出規則 0

$$\begin{aligned} (x: \sigma) \in \Gamma &\Rightarrow \Gamma \vdash x: \sigma \\ \Gamma \vdash M: (\sigma \rightarrow \tau), \Gamma \vdash N: \sigma &\Rightarrow \Gamma \vdash (MN): \tau \\ \Gamma, x: \sigma \vdash M: \tau &\Rightarrow \Gamma \vdash (\lambda x.M): (\sigma \rightarrow \tau) \end{aligned}$$

ここで $\Gamma, x: \sigma$ は $\Gamma \cup x: \sigma$ のことである.

$\Gamma = \{x_1: \sigma_1, \dots, x_n: \sigma_n\}$ (または $\Gamma = \emptyset$) のとき, $\Gamma \vdash M: \sigma$ を $x_1: \sigma_1, \dots, x_n: \sigma_n \vdash M: \sigma$ (または $\vdash M: \sigma$) と書く. 導入規則は以下のようにも表される.

$$\begin{array}{ll}
 \text{(公理)} & \Gamma \vdash x: \sigma \quad (x: \sigma) \in \Gamma \text{ のとき} \\
 (\rightarrow \text{除去}) & \frac{\Gamma \vdash M: (\sigma \rightarrow \tau) \quad \Gamma \vdash N: \sigma}{\Gamma \vdash (MN): \tau} \\
 (\rightarrow \text{導入}) & \frac{\Gamma, x: \sigma \vdash M: \tau}{\Gamma \vdash (\lambda x.M): (\sigma \rightarrow \tau)}
 \end{array}$$

以下、本稿においては、導出規則 1 を用いる。

Example 1.2.2.5 1. λ 項に対してある型が付くことを示すには、導出規則を逆向きに使って辿る方法、つまり、

「示したい型付けから始めて、その導出のために必要となる前提を順に上に書き加えていくことを繰り返し、変数のみとなった時点で、その変数に対して矛盾なく型がついているかを確認する」

という手順が便利である。この手順を用いて、任意の $\sigma, \tau \in \mathbb{T}$ について、 $\vdash (\lambda xy.x): (\sigma \rightarrow \tau \rightarrow \sigma)$ であることを確認する。まず、 $\lambda xy.x$ の最も大きな構造は抽象である。抽象に対して型を割り当てる規則は \rightarrow 導入なので、

$$\frac{x: \sigma \vdash (\lambda y.x): (\tau \rightarrow \sigma)}{\vdash (\lambda xy.x): (\sigma \rightarrow \tau \rightarrow \sigma)}$$

というように、必要となる前提を書き加える。ここで、この書き加えた前提を見てみると、これも抽象の形をしているため、

$$\frac{\frac{x: \sigma, y: \tau \vdash x: \sigma}{x: \sigma \vdash (\lambda y.x): (\tau \rightarrow \sigma)}}{\vdash (\lambda xy.x): (\sigma \rightarrow \tau \rightarrow \sigma)}$$

というように、さらに前提を書き加える。すると、一番上の行が変数のみとなり、 x に対してともに σ が型付けられており、矛盾なく型がついていることがわかるので、 $\vdash (\lambda xy.x): (\sigma \rightarrow \tau \rightarrow \sigma)$ が示せたことになる。このように、必要な前提を書き加え続けることで得られたものを導出木と呼ぶ。

2. 任意の $\sigma, \tau \in \mathbb{T}$ について、 $\lambda xy.xy: (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$ であることは、

$$\frac{\frac{\frac{x: (\sigma \rightarrow \tau), y: \sigma \vdash x: (\sigma \rightarrow \tau) \quad x: (\sigma \rightarrow \tau), y: \sigma \vdash y: \sigma}{x: (\sigma \rightarrow \tau), y: \sigma \vdash xy: \tau}}{x: (\sigma \rightarrow \tau) \vdash \lambda y.xy: (\sigma \rightarrow \tau)}}{\vdash \lambda xy.xy: (\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)}$$

という導出木により確認できる。

3. $\lambda \rightarrow$ においては、 $\lambda x.xx$ には型が付かない。実際、

$$\Gamma \vdash \lambda x.xx: \sigma$$

と仮定すると、抽象の形であるから、 $\sigma = (\sigma_1 \rightarrow \sigma_2)$ となる σ_1, σ_2 が存在して、

$$\frac{\Gamma, x: \sigma_1 \vdash xx: \sigma_2}{\Gamma \vdash \lambda x.xx: \sigma_1 \rightarrow \sigma_2}$$

となるはずである。さらに、 xx は適用の形なので、ある τ が存在し、

$$\frac{\frac{\Gamma, x: \sigma_1 \vdash x: \tau \rightarrow \sigma_2 \quad \Gamma, x: \sigma_1 \vdash x: \tau}{\Gamma, x: \sigma_1 \vdash xx: \sigma_2}}{\Gamma \vdash \lambda x.xx: \sigma_1 \rightarrow \sigma_2}$$

となるはずである。しかし、 $\lambda \rightarrow$ において、同じ x に対して τ と $\tau \rightarrow \sigma_2$ の両方がつくことはありえない。よって、 $\lambda x.xx$ には型が付かない。

$\lambda \rightarrow$ の性質

$\lambda \rightarrow$ の型付けに関するいくつかの性質を述べる. 1 つ目の性質は, ある型付けを導出するためには, どれだけの基底が必要かを示すものである.

Definition 1.2.2.6 ([8] Definition 3.1.6) $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ を基底とする.

1. Γ を部分関数のように扱い, $\text{dom}(\Gamma) = \{x_1, \dots, x_n\}$, $\sigma_i = \Gamma(x_i)$ とする.
2. V_0 を変数の集合とする. このとき, $\Gamma \upharpoonright V_0 = \{x : \sigma \mid x \in V_0 \text{ かつ } \sigma = \Gamma(x)\}$ と定める.
3. $\sigma, \tau \in \mathbb{T}$ に対して, σ において α に τ を代入することを $\sigma[\alpha := \tau]$ と表す.

Proposition 1.2.2.7 ([8] Proposition 3.1.7)(Basis lemma for $\lambda \rightarrow$) Γ を基底とする.

1. $\Gamma' \supset \Gamma$ が基底のとき,

$$\Gamma \vdash M : \sigma \Rightarrow \Gamma' \vdash M : \sigma.$$

2. $\Gamma \vdash M : \sigma \Rightarrow FV(M) \subset \text{dom}(\Gamma)$.
3. $\Gamma \vdash M : \sigma \Rightarrow \Gamma \upharpoonright FV(M) \vdash M : \sigma$.

Proof. $M : \sigma$ の導出に関する帰納法により示す. ここで, 導出に関する帰納法とは, 導出規則が帰納的に定義されていることを用いた証明法であり, 型付けに関する証明においてよく使われるやり方である. Proposition 1.2.2.7 の証明の前に,

$$\Gamma \vdash M : \sigma \Rightarrow f(\Gamma, M, \sigma) \quad (\text{ここで, } f(\Gamma, M, \sigma) \text{ は } \Gamma \text{ と } M \text{ と } \sigma \text{ に関する何らかの主張})$$

という形の命題の, 導出に関する帰納法による証明の概略について述べておく.

- base case : 前提がない場合

$\Gamma \vdash M : \sigma$ が前提なく導出されるのは, $M \equiv x$, つまり M が変数 1 つのみのときである. この場合は, $(x : \sigma \in \Gamma)$ のみを仮定して, $f(\Gamma, x, \sigma)$ が成り立つことを示すことになる.

- inductive case : 前提がある場合

前提をもとにある型が付くことが導出されるのは, $M \equiv M_1 M_2$ もしくは $M \equiv \lambda x. M_1$ のときである. このときは, 前提について主張が成り立つことを仮定して証明を進める. 例えば, $M \equiv M_1 M_2$ のときであれば, $\Gamma \vdash M : \sigma$ は, ある τ について, $\Gamma \vdash M_1 : \tau \rightarrow \sigma$ と $\Gamma \vdash M_2 : \tau$ の 2 つの前提から導出されるので,

$$\begin{aligned} \Gamma \vdash M_1 : \tau \rightarrow \sigma &\Rightarrow f(\Gamma, M_1, (\tau \rightarrow \sigma)) \\ \Gamma \vdash M_2 : \tau &\Rightarrow f(\Gamma, M_2, \tau) \end{aligned}$$

の 2 つを仮定して, $\Gamma \vdash (M_1 M_2) : \sigma \Rightarrow f(\Gamma, (M_1 M_2), \sigma)$ を示すことになる. 同様に, $M \equiv \lambda x. M_1$ の場合, $\Gamma \vdash \lambda x. M_1 : \sigma$ は, ある σ_1, σ_2 について, $\sigma = (\sigma_1 \rightarrow \sigma_2)$ であり, $\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2$ から導出されるので,

$$\Gamma, x : \sigma_1 \vdash M_1 : \sigma_2 \Rightarrow f(\Gamma, x : \sigma_1, M_1, \sigma_2)$$

を仮定して, $\Gamma \vdash (\lambda x. M_1) : (\sigma_1 \rightarrow \sigma_2) \Rightarrow f(\Gamma, (\lambda x. M_1), (\sigma_1 \rightarrow \sigma_2))$ を示すことになる.

まとめると, M がどの形なのかで場合分けし, それぞれの導出の前提に対して成立を仮定して, 証明をするということである. 以下, 実際の Proposition 1.2.2.7 の証明にうつる.

1. (a) $M : \sigma = x : \sigma$ のとき

$x : \sigma \in \Gamma$ より $x : \sigma \in \Gamma'$ から $\Gamma' \vdash x : \sigma$ である.

- (b) $M : \sigma = (M_1 M_2) : \sigma$ のとき

$\Gamma \vdash (M_1 M_2) : \sigma$ は, ある τ について $\Gamma \vdash M_1 : (\tau \rightarrow \sigma)$ と $\Gamma \vdash M_2 : \tau$ から導出される. $\Gamma \vdash M_1 : (\tau \rightarrow \sigma) \Rightarrow \Gamma' \vdash M_1 : (\tau \rightarrow \sigma)$ かつ $\Gamma \vdash M_1 : (\tau \rightarrow \sigma) \Rightarrow \Gamma' \vdash M_2 : \tau$ と仮定すると, \rightarrow 除去規則より $\Gamma' \vdash (M_1 M_2) : \sigma$ となる.

(c) $M: \sigma = (\lambda x.M_1): (\sigma_1 \rightarrow \sigma_2)$ のとき

$\Gamma \vdash (\lambda x.M_1): (\sigma_1 \rightarrow \sigma_2)$ は $\Gamma, x: \sigma_1 \vdash M_1: \sigma_2$ から導出される. $x \notin \text{dom}(\Gamma')$ としてよく, このとき, $\Gamma', x: \sigma_1 \supset \Gamma, x: \sigma_1$ である. $\Gamma, x: \sigma_1 \vdash M: \sigma_2 \Rightarrow \Gamma', x: \sigma_1 \vdash M: \sigma_2$ と仮定すると, \rightarrow 導入規則より $\Gamma' \vdash (\lambda x.M_1): (\sigma_1 \rightarrow \sigma_2)$ となる.

2. (a) $M: \sigma = x: \sigma$ のとき, $FV(M) = x$ よりすぐにわかる.

(b) $M: \sigma = (M_1 M_2): \sigma$ のとき

帰納法の仮定から, $FV(M_1) \subset \text{dom}(\Gamma)$ かつ $FV(M_2) \subset \text{dom}(\Gamma)$ が成り立つので,

$$(FV(M_1) \cup FV(M_2)) \subset \text{dom}(\Gamma)$$

となる. $FV(M_1 M_2) = FV(M_1) \cup FV(M_2)$ より, $FV(M_1 M_2) \subset \text{dom} \Gamma$ がわかる.

(c) $M: \sigma = (\lambda x.M_1): (\sigma_1 \rightarrow \sigma_2)$ のとき

帰納法の仮定より, $FV(M_1) \subset \text{dom}(\Gamma, x: \sigma)$ である. また,

$$\begin{aligned} \text{dom}(\Gamma, x: \sigma) &= \text{dom}(\Gamma) \cup \{x\} \\ FV(\lambda x.M_1) &= FV(M_1) - \{x\} \end{aligned}$$

より, $FV(\lambda x.M_1) \subset \text{dom}(\Gamma)$ がわかる.

3. (a) $M: \sigma = x: \sigma$ のとき

$FV(M) = x$ から, $\Gamma \upharpoonright FV(M) = x: \sigma$ となる. $(x: \sigma) \vdash (x: \sigma)$ より成り立つ.

(b) $M: \sigma = (M_1 M_2): \sigma$ のとき

帰納法の仮定から, ある τ が存在し, $\Gamma \upharpoonright FV(M_1) \vdash M_1: \tau \rightarrow \sigma$, $\Gamma \upharpoonright FV(M_2) \vdash M_2: \tau$ となる. よって, 1. より, $\Gamma \upharpoonright FV(M_1 M_2) \vdash M_1: \tau \rightarrow \sigma$, $\Gamma \upharpoonright FV(M_1 M_2) \vdash M_2: \tau$ がわかる. したがって, $\Gamma \upharpoonright FV(M) \vdash M: \sigma$ となる.

(c) $M: \sigma = \lambda x.M_1: (\sigma_1 \rightarrow \sigma_2)$ のとき

帰納法の仮定から, $(\Gamma, x: \sigma_1) \upharpoonright FV(M_1) \vdash M_1: \sigma_2$ となる. ここで, $(\Gamma, x: \sigma) \upharpoonright FV(M_1) = \Gamma \upharpoonright FV(M_1) \cup (x: \sigma) \upharpoonright FV(M_1)$ であり,

$$\begin{aligned} \Gamma \upharpoonright (FV(M_1) - \{x\}) &\supset \Gamma \upharpoonright FV(M_1) \\ x: \sigma &\supset (x: \sigma) \upharpoonright FV(M_1) \end{aligned}$$

から, 1. より $x: \sigma, \Gamma \upharpoonright (FV(M_1) - \{x\}) \vdash M_1: \sigma_2$ となる. よって,

$$\Gamma \upharpoonright FV(M_1) \vdash \lambda x.M_1: (\sigma_1 \rightarrow \sigma_2)$$

がわかる.

□

2 つ目の性質は、特定の形の項がどのように型付けされるかを分析するものである. これは、ある種の項が型を持たないことを示すのに特に有用である.

Proposition 1.2.2.8 ([8] Proposition 3.1.8)(Generation lemma for $\lambda \rightarrow$) 以下が成り立つ.

1. $\Gamma \vdash x: \sigma \Rightarrow (x: \sigma) \in \Gamma$
2. $\Gamma \vdash MN: \tau \Rightarrow \exists \sigma [\Gamma \vdash M: (\sigma \rightarrow \tau) \text{ かつ } \Gamma \vdash N: \sigma]$
3. $\Gamma \vdash (\lambda x.M): \rho \Rightarrow \exists \sigma, \tau [\Gamma, x: \sigma \vdash M: \tau \text{ かつ } \rho \equiv (\sigma \rightarrow \tau)]$

Proof. 導出規則より従う.

□

Proposition 1.2.2.9 ([8] Proposition 3.1.9) (Subterm lemma for $\lambda \rightarrow$) M' を M の部分項とする. このとき, ある Γ' と σ' が存在して,

$$\Gamma \vdash M : \sigma \Rightarrow \Gamma' \vdash M' : \sigma'$$

をみたす. つまり, M が型を持つとき, その任意の部分項も型を持つ.

Proof. $\Gamma \vdash M : \sigma$ の導出木を考えれば, 各部分項についての導出木も得られることからわかる. \square

Remark 1.2.2.10 この命題の対偶から, 部分項に型が付かなければ, 全体にも型が付かないことがわかる. このことを用いれば, 例えば, $\lambda \rightarrow$ において $\lambda x.xx$ に型が付かないことから, $(\lambda x.xx)(yz)$ や, $\lambda y.(\lambda x.xx)$ には型がつかない.

Proposition 1.2.2.11 ([8] Proposition 3.1.10) (Substitution lemma for $\lambda \rightarrow$) 以下が成り立つ.

1. $\Gamma \vdash M : \sigma \Rightarrow \Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.
2. $\Gamma, x : \sigma \vdash M : \tau$ かつ $\Gamma \vdash N : \sigma$ とする. このとき, $\Gamma \vdash M[x := N] : \tau$.

Proof. 1. $M : \sigma$ の導出に関する帰納法により示される.
2. $\Gamma, x : \sigma \vdash M : \tau$ の導出のステップ数に関する帰納法により示される. \square

次の結果は, ラムダ項がある型を持つということが, 簡約について閉じていることを示す.

Proposition 1.2.2.12 [8] Proposition 3.1.11 (Subject reduction theorem for $\lambda \rightarrow$) $M \rightarrow_\beta M'$ とする. このとき,

$$\Gamma \vdash M : \sigma \Rightarrow \Gamma \vdash M' : \sigma.$$

Proof. Prop 1.2.2.8, Prop 1.2.2.11 を用いて, 簡約のステップ数に関する帰納法により示される.

- ステップ数が 0, つまり $M = M'$ のとき, $\Gamma \vdash M : \sigma$ からすぐにわかる
- ステップ数が n 以下のとき成立すると仮定し, $n+1$ のとき成立することを示す.

$$\underbrace{M \rightarrow_\beta M''}_{n \text{ ステップ}} \rightarrow_\beta M'$$

をみたす M'' について, $\Gamma \vdash M : \sigma \Rightarrow \Gamma \vdash M'' : \sigma$ を仮定して, $\Gamma \vdash M'' : \sigma \Rightarrow \Gamma \vdash M' : \sigma$ を示す. $M'' \rightarrow_\beta M'$ の簡約がどのようにされるかで場合分けする.

1. $M'' \equiv (\lambda x.P)Q$ かつ $M' \equiv P[x := Q]$ のとき,

$$\Gamma \vdash (\lambda x.P)Q : \sigma$$

とすると, Proposition 1.2.2.8 より, ある τ が存在して,

$$\Gamma \vdash (\lambda x.P) : (\tau \rightarrow \sigma) \text{ かつ } \Gamma \vdash Q : \tau$$

をみたす. 更に, もう一度 Proposition 1.2.2.8 から,

$$\Gamma, x : \tau \vdash P : \sigma \text{ かつ } \Gamma \vdash Q : \tau$$

となる. ここで, Proposition 1.2.2.11 を用いれば,

$$\Gamma \vdash P[x := Q] : \sigma$$

がわかる.

2. $M'' \equiv ZP, M' \equiv ZQ$ のとき,

$$\Gamma \vdash ZP: \sigma$$

とすると, Proposition 1.2.2.8 より, ある τ が存在して,

$$\Gamma \vdash Z: \tau \rightarrow \sigma \text{ かつ } \Gamma \vdash P: \tau$$

をみたす. ここで, \rightarrow_β の定義から, $P \rightarrow_\beta Q$ でなくてはならず, この簡約は 1 ステップであるから, 帰納法の仮定から $\Gamma \vdash Q: \tau$ となる. よって,

$$\Gamma \vdash ZQ: \sigma$$

がわかる.

3. $M'' \equiv PZ, M' \equiv QZ$ のとき, 2 のときと同様, Proposition 3.1.8 と帰納法の仮定から $\Gamma \vdash P: \tau \rightarrow \sigma$ かつ $\Gamma \vdash Z: \tau$ および $\Gamma \vdash Q: \tau \rightarrow \sigma$ が得られることからわかる.

4. $M'' \equiv \lambda x.P, M' \equiv \lambda x.Q$ のとき, 2, 3 のときと同様, Proposition 3.1.8 と帰納法の仮定から, ある τ, ρ が存在して, $\Gamma, x: \tau \vdash P: \rho$ かつ $\sigma = (\tau \rightarrow \rho)$ および $\Gamma, x: \tau \vdash Q: \rho$ が得られる. よって,

$$\Gamma \vdash \lambda x.Q: (\tau \rightarrow \rho)(= \sigma)$$

が得られる.

□

$\lambda 2$

$\lambda \rightarrow$ の拡張の 1 つとして, $\lambda 2$ を考える. こちらも Curry 流と Church 流の 2 通りがあるが, 本稿では $\lambda \rightarrow$ と同じく Curry 流の定義を採用する.

Definition 1.2.2.13 ([8] Definition 4.1.5) $\lambda 2$ の型の集合 $\text{Type}(\lambda 2)$ を \mathbb{T} と書く. このとき, \mathbb{T} は以下のように帰納的に定義される.

$$\mathbb{T} = \mathbb{V} \mid \mathbb{T} \rightarrow \mathbb{T} \mid \forall \mathbb{V} \mathbb{T}$$

Remark 1.2.2.14 ([8] Notation 4.1.6)

1. $\forall \alpha_1 \cdots \alpha_n. \sigma$ は, $(\forall \alpha_1 (\forall \alpha_2 \cdots (\forall \alpha_n (\sigma)) \cdots))$ を表す.
2. \forall の方が \rightarrow よりも結合が強い, すなわち, $\forall \alpha \sigma \rightarrow \tau$ は $(\forall \alpha \sigma) \rightarrow \tau$ のことであり, $\forall \alpha. \sigma \rightarrow \tau$ は $\forall \alpha (\sigma \rightarrow \tau)$ のことである.

Definition 1.2.2.15 $\lambda 2$ の型導出規則を, 以下のように定める.

(公理)	$\Gamma \vdash x: \sigma$	$(x: \sigma)$ のとき
(\rightarrow 除去)	$\frac{\Gamma \vdash M: (\sigma \rightarrow \tau) \quad \Gamma \vdash N: \sigma}{\Gamma \vdash (MN): \tau}$	
(\rightarrow 導入)	$\frac{\Gamma, x: \sigma \vdash M: \tau}{\Gamma \vdash (\lambda x.M): (\sigma \rightarrow \tau)}$	
(\forall 除去)	$\frac{\Gamma \vdash M: (\forall \alpha. \sigma)}{\Gamma \vdash M: (\sigma[\alpha := \tau])}$	
(\forall 導入)	$\frac{\Gamma \vdash M: \sigma}{\Gamma \vdash M: (\forall \alpha. \sigma)}$	$(\alpha \notin FV(\Gamma))$

Example 1.2.2.16 ([8] Example 4.1.8) $\lambda 2$ での型付けの例として, 以下のようなものがある.

$$\begin{aligned} &\vdash (\lambda x.x): (\forall \alpha. \alpha \rightarrow \alpha) \\ &\vdash (\lambda xy.y): (\forall \alpha \beta. \alpha \rightarrow \beta \rightarrow \beta) \\ &\vdash (\lambda x.xx): (\forall \alpha \alpha) \rightarrow (\forall \alpha \alpha) \end{aligned}$$

ここで, 3 つ目の例については,

$$\frac{\frac{x: \forall \alpha \alpha \vdash x: \forall \alpha \alpha \quad x: \forall \alpha \alpha \vdash x: \forall \alpha \alpha}{x: \forall \alpha \alpha \vdash x: \beta \rightarrow \alpha} \quad \frac{x: \forall \alpha \alpha \vdash x: \forall \alpha \alpha}{x: \forall \alpha \alpha \vdash x: \beta}}{\frac{x: \forall \alpha \alpha \vdash xx: \alpha}{x: \forall \alpha \alpha \vdash xx: \forall \alpha \alpha}} \vdash (\lambda x.xx): (\forall \alpha \alpha) \rightarrow (\forall \alpha \alpha)$$

という導出木により確認できる. 型に \forall があることにより, $\lambda \rightarrow$ では型が付けられないような項についても, $\lambda 2$ では型が付けられる場合がある.

$\lambda 2$ についても, Basis lemma, Subterm lemma, Substitution lemma が成り立つことが知られている ([8] Proposition 4.1.17, Proposition 4.1.18, Proposition 4.1.19 参照). 一方で, Generation lemma は成り立たない. では, Subject reduction lemma はどうであろうか?

1.3 Coq

1.3.1 λ -cube と CIC

1.3.2 Coq の使い方

この節では Coq のコマンドとタクティックの使い方について述べる. ここでは, タクティックは証明の中でコンテキスト (変数や仮定) やゴール (証明すべき主張) を変形させるもの, コマンドはタクティック以外のものとして扱う. まず, よく使うコマンドについて説明する.

Require Import ライブラリを読み込むためのコマンドである. **From mathcomp Require Import ssreflect** であれば, ライブラリ群 mathcomp から ssreflect を読み込んでいる.

Section / End Section [セクション名], **End** [セクション名] でセクションを作ることができ, そのセクション内共通のコンテキストを宣言できる.

Variable Variable [変数]: [型] で, 特定の型を持つ変数を宣言できる. 例えば

Variable n: nat

で, 変数 n が自然数型 **nat** の要素であることを表している. **Section/End** コマンドと組み合わせることで, **End** まで同じ意味で扱われ, **End** 以降は効力を失う. 同時に複数の変数を宣言することもできる. その場合は

Variables [変数] [変数] \dots [変数]: [型]

と書く (ただし, Coq にとっては **Variable** と **Variables** に違いは無い).

Hypothesis Hypothesis [仮定名]: [仮定] で仮定を置くことができる. **Variable** 同様, **Section/End** と組み合わせることで, セクション内共通の仮定を置くことができる.

Definition 新たに関数を定義するためのコマンドで,

Definition [関数名] ([引数]: [引数の型]): [関数の型] := [関数の定義式]

という形で用いる.

Definition dq (f : R \rightarrow R) x := f (q * x) - f x.

であれば, `dq` が定義の名前, `f, x` が引数, $R \rightarrow R$ が `f` の型であり, $f (q * x) = f x$ が `dq` を定義する式である. また, `x` と `dq` そのものの型は推論できるため省略できる.

Fixpoint 再起関数を定義するためのコマンドで,

`Fixpoint` [関数名] ([引数]: [引数の型]): [関数の型] := [定義中の関数を含む定義式]

と書く. 停止しない関数を認めてしまうと矛盾が生じるため, 停止性が保証されていない関数を定義することはできない.

Lemma 補題を宣言するためのコマンドで,

`Lemma` [補題名] ([引数]: [引数の型]): [補題の主張]

という形である.

`Lemma` `Dq_pow n x : x ≠ 0 → Dq (fun x ⇒ x ^ n) x = qnat n * x ^ (n - 1).`

であれば, `Dq_pow` が補題名, `n, x` が引数, `:`以降が補題の主張である.

`Lemma` の代わりに `Theorem`, `Corollary` 等でも同じ機能をもつ.

Proof/Qed `Proof` は `Lemma` の後に書いて補題の主張と証明を分ける (実際には省略可能で, 人間の見やすさのために書いている). 証明を完了させて `Qed` を書くことで `Coq` に補題を登録することができ, 他の補題の証明に使えるようになる.

次に, タクティックについて述べる. よく使われるタクティックは `move`, `apply`, `rewrite` の 3 つである.

move `move=> H` でゴールの前提に `H` という名前をつけてコンテキストに移動する. また `move: H` で補題 `H` もしくはコンテキストに存在する `H` をゴールの前提に移す.

apply 補題 `lem` が $P1 \rightarrow P2$ という形で, ゴールが `P2` のとき, `apply lem` でゴールを `P1` に変える. コンテキストの仮定 `H` が $P1 \rightarrow P2$ であれば `apply H` で同じことができる.

rewrite `def` が定義のとき, `rewrite /def` でゴールに出現している `def` を展開する.

また, 補題 `lem` が $A = B$ という形のとき, `rewrite lem` でゴールに出現する `A` を `B` に書き換える (ただし, `lem` が $H \rightarrow (A = B)$ という形であるとき, `H` がゴールに追加される). 更に, `rewrite lem in H` で, コンテキストの `H` に出現する `A` を `B` に書き換える. `apply` と同じく, 使いたい等式が仮定にある場合も同じように使える. 更に, `rewrite` は繰り返し回数や適用箇所を指定できる.

`rewrite !lem` `lem` による書き換えを可能な限り繰り返す. ただし場合によっては繰り返しが終わらない場合があることに注意 ($x + y$ を $y + x$ に書き換える補題を使う場合など).

`rewrite n!lem` `lem` による書き換えを `n` 回限り繰り返す.

`rewrite ?lem` `lem` による書き換えを 0 回または 1 回行う. 直前のタクティックでゴールが増える場合に特に有効.

`rewrite -lem` 逆向きに `lem` による書き換えを行う. つまり, `lem` が $A = B$ のとき, ゴールの `B` を `A` に書き換える.

`rewrite {n}lem` `lem` で書き換えられる場所のうち `n` 番目のみを書き換える.

`rewrite (_ : A = B)` ゴールの `A` を `B` に書き換える. $A = B$ がゴールに追加される. この書き方の場合は補題を引数にとらない.

以下, 2 つの具体例を用いて `Proof` 内でのタクティックの使い方を説明する.

Example 1.3.2.1 モーダスポーネンス

命題 P, Q について, $P \implies Q$ かつ P であれば, Q が成り立つということを `Coq` で証明する. まずこの主張を形式化すると以下の通り.

From mathcomp Require Import ssreflect.

`Lemma` `modus_ponens (P Q : Prop) : (P → Q) ∧ P → Q.`

Prop とは Coq において命題全体を表す型であり, \wedge は「かつ」を表している.

このとき, Coq のゴールエリア (コンテキストとゴールが表示される画面) は以下の通りである.

```
1 subgoal
P, Q : Prop
-----
(P → Q) ∧ P → Q
```

---の上がコンテキスト, 下がゴールである. 1 行目の 1 subgoal はゴールが 1 つであることを示しており, 用いるタクティックによってはゴールが増えることもある. 命題 P_1, P_2, P_3 について $P_1 \wedge P_2 \rightarrow P_3$ と $P_1 \rightarrow P_2 \rightarrow P_3$ は同じ意味であり, この書き換えは `move=> []` で行える. 実行すると以下のようにゴールエリアが変化する.

```
1 subgoal
P, Q : Prop
-----
(P → Q) → P → Q
```

ゴールに前提 $(P \rightarrow Q)$ があるため, `move=> pq` で `pq` という名前をつけてコンテキストに移動する.

```
1 subgoal
P, Q : Prop
pq : P → Q
-----
P → Q
```

まだ前提 P があるため, `move=> p` で `p` と名付けてコンテキストに移動する.

```
1 subgoal
P, Q : Prop
pq : P → Q
p : P
-----
Q
```

ゴールが Q であり, コンテキストに $P \rightarrow Q$ という仮定 `pq` があるので, `apply pq` でゴールを P に書き換える.

```
1 subgoal
P, Q : Prop
pq : P → Q
p : P
-----
P
```

ここまで来ると, ゴールが P であり, コンテキストに P があるため, `by []` で証明を終了する. `done` も `by []` と

同じ意味を持つ.

No more subgoals.

ゴールエリアに **No more subgoals.** と表示されれば証明は終了であり, **Qed** を書くことで補題として登録されることになる.

以上をまとめると次のようになる.

Lemma modus_ponens (P Q : **Prop**) : (P → Q) ∧ P → Q.

Proof.

move⇒ [].

move⇒ pq.

move⇒ p.

apply pq.

by [].

Qed.

説明のため細かく 1 行ずつ書いたが, 複数の **move** はまとめられること, あるタクティックによりゴールが自明なもの (コンテキストに存在する, $A = A$ である, 計算から簡単に示されるなど) になる場合はそのタクティックの前に **by** をつけることで証明を終了させられることを用いれば, 次のように短くすることができる.

Lemma modus_ponens (P Q : **Prop**) : (P → Q) ∧ P → Q.

Proof.

move⇒ [] pq p.

by **apply** pq.

Qed.

Coq による証明は, Curry-Howard 同型と呼ばれる,

命題 ↔ 型

証明 ↔ 型に要素が存在する

という対応関係に基づいている. また, 論理演算子についても, 以下のような対応がある.

P ならば Q $P \rightarrow Q$

P かつ Q $P \times Q$

P または Q $P + Q$

この同型をもとに上記の証明をもう一度考えてみると, $P \rightarrow Q$ と P という型に要素が存在することから, Q という型の要素を構成すればよいということである.

まず, 前提の要素それぞれに pq, p と名前をつける. これがプログラム中の **move**⇒ [] pq p のことである. ここで, $P \rightarrow Q$ という型は, 入力する値の型が P , 出力する値の型が Q であるような関数の型であるため, P の要素 p に pq を適用することで, Q の要素を構成することができる. この関数適用がプログラム中の **apply** pq のことである.

Example 1.3.2.2 代入計算

自然数 m, n について,

$$m = 0 \implies n + m = n$$

という簡単な代入に関する計算を証明してみる. まず主張を形式化する.

Lemma substitution m n : m = 0 → n + m = n.

このとき、ゴールエリアは以下の通りである。

```
1 subgoal
m, n : nat
-----
m = 0 → n + m = n
```

まず $m = 0$ という前提を `move=> Hm` で `Hm` という名前をつけてコンテキストに移動する。

```
1 subgoal
m, n : nat
Hm : m = 0
-----
n + m = 0
```

仮定に $m = 0$ という等式があるため、`rewrite Hm` でゴールの m を 0 に書き換える。

```
1 subgoal
m, n : nat
Hm : m = 0
-----
n + 0 = n
```

次に、 $n + 0$ を n に書き換えたい。mathcomp の `ssrnat` に `addn0` という、
`forall n : nat, n + 0 = n`
に対応する補題があるため、`rewrite addn0` を実行する。

```
1 subgoal
m, n : nat
Hm : m = 0
-----
n = n
```

このとき、ゴールは同じもの同士の等号であるため、自明に成り立つ、つまり `by []` (もしくは `done`) で終了する。

```
No more subgoals.
```

この証明をまとめると以下の通りである。

Lemma `substitution m n : m = 0 → n + m = n.`

Proof.

```
move=> Hm.
rewrite Hm.
rewrite addn0.
by [].
```

Qed.

複数の `rewrite` がまとめられることと `by` の使い方から、より短く次のように書ける。

```
Lemma substitution m n : m = 0 → n + m = n.
```

```
Proof.
```

```
  move⇒ Hm.
```

```
  by rewrite Hm addn0.
```

```
Qed.
```

更に、前提が $A = B$ の形であるとき、`move⇒` でゴールの A を B に書き換えられること、`;` で異なるタクティックをつながられることを用いれば以下のように書くこともできる。

```
Lemma substitution m n : m = 0 → n + m = n.
```

```
Proof. by move⇒; by rewrite addn0. Qed.
```

Remark 1.3.2.3 正確には、`addn0` は

```
right_id 0 addn
```

という補題であり、`addn` は自然数同士の加法である。`right_id` の定義は

```
fun (S T : Type) (e : T) (op : S → T → S) ⇒ ∀ x : S, op x e = x
```

であり、単位元を右から作用させても元のままであるということを一般的に定義している。`addn0` は `right_id` の `e` に `0` を、`op` に `addn` を入れたものであるので、

```
∀ x : nat, addn x 0 = x
```

となる。

このように、Coq での証明は、ゴールを自明な形になるまで繰り返し書き変えていくやり方が基本である。

1.3.3 mathcomp の型

今回は `mathcomp` の `ssrnum` にある `rcfType` を用いる。`rcfType` とは、real closed field、つまり実閉体全体からなる型であり、この型の要素を実数として用いることにする。`mathcomp` はある型を構成するために他の型を用いているため、ヒエラルキー (階層構造) がある。通常の数学において体が環の性質を、環が群の性質を引き継ぐように、`mathcomp` でもより一般の型の性質を引き継いでいる。`rcfType` も多くの性質を引き継いでおり、特に環の性質、`ringType` を持っていることが重要である。実際、形式化で利用するライブラリの補題の多くは `ringType` に対するものである。

自然数型 `nat` .+1 は `nat` での後続数を表す

整数型 `int`

1.4 形式化

この定義を形式化するためまずは実数の形式化を考える。

1.4.1 q -微分の定義

様々な q -類似を考えるにあたって、まずは微分の q -類似から始める。以下、 q を 1 でない実数とする。

Definition 1.4.1.1 ([1] p1 (1.1), p2 (1.5)) 関数 $f : \mathbb{R} \rightarrow \mathbb{R}$ に対して, $f(x)$ の q 差分 $d_q f(x)$ を,

$$d_q f(x) := f(qx) - f(x)$$

と定める. 更に, $f(x)$ の q 微分を $D_q f(x)$ を,

$$D_q f(x) := \frac{d_q f(x)}{d_q x} = \frac{f(qx) - f(x)}{(q-1)x}$$

と定める.

`rcfType` を使って q -微分の形式化を以下のように行う.

From mathcomp Require `Import` all_ssreflect all_algebra.

`Import` GRing.

`Section` q_analogue.

Local Open Scope ring_scope.

`Variables` (R : rcfType) (q : R).

`Hypothesis` Hq : q - 1 ≠ 0.

`Notation` "f // g" := (fun x => f x / g x) (at level 40).

`Definition` dq (f : R → R) x := f (q * x) - f x.

`Definition` Dq f := dq f // dq id.

5 行目の `Variables` (R : rcfType) (q : R). は, R が `rcfType` の要素であること, q が R の要素であることを表している. `rcfType` は実閉体全体を表す型であるため, R は具体的な型ではなく, 実閉体としての性質をもつ抽象的な型として扱っている.

Remark 1.4.1.2 f が微分可能であるとき,

$$\lim_{q \rightarrow 1} D_q f(x) = \frac{d}{dx} f(x)$$

が成り立つが, 本論文においては極限操作に関しての形式化は扱わない.

次に, x^n ($n \in \mathbb{Z}_{\geq 0}$) を q -微分した際にうまく振る舞うように自然数の q -類似を定義する.

Definition 1.4.1.3 ([1] p2 (1.9)) $n \in \mathbb{Z}_{\geq 0}$ に対して, n の q -類似 $[n]$ を,

$$[n] := \frac{q^n - 1}{q - 1}$$

と定義する.

この $[n]$ に対して, $(x^n)' = nx^{n-1}$ の q -類似が成り立つ.

Proposition 1.4.1.4 ([1] p2 Example (1.7)) $n \in \mathbb{Z}_{>0}$ について,

$$D_q x^n = [n] x^{n-1}$$

が成り立つ.

Proof. 定義に従って計算すればよく,

$$D_q x^n = \frac{(qx)^n - x^n}{(q-1)x} = \frac{q^n - 1}{q-1} x^{n-1} = [n] x^{n-1}$$

□

この定義と補題の形式化は以下のとおりである.

Definition `qnat n : R := (q ^ n - 1) / (q - 1).`

Lemma `Dq_pow n x : x ≠ 0 → Dq (fun x => x ^ n) x = qnat n * x ^ (n - 1).`

Proof.

```
move => Hx.
rewrite /Dq /dq /qnat.
rewrite -{4}(mulr x) -mulrBl expfzM1 -add_div; last by apply mulf_neq0.
rewrite [in x ^ n](_ : n = (n - 1) + 1) //; last by rewrite subrK.
rewrite expfzDr ?expfzDr ?mulrA -?mulNr ?red_frac_r ?add_div //.
rewrite -{2}[x ^ (n - 1)]mulr -mulrBl mulrC mulrA.
by rewrite [in (q - 1)^-1 * (q ^ n - 1)] mulrC.
```

Qed.

証明で用いている補題について, 例えば `mulr` は

`∀ R : ringType, right_id 1 *%R`

という補題であり, 任意の `ringType` を型にもつ `R` について, `R` 上の単位元の性質を表している. ここで, 補題の `R` の型は `rcfType` ではなく `ringType` であるが, 前述の通り `rcfType` は `ringType` の性質を持っているため, 今回の形式化に用いている `rcfType` 型の `R` についてもこの補題を使うことができる. 補題名 `mulr` の `r` は `ringType` に対する補題であることを表しており, `mulrBl`, `subrK`, `mulrC` などと同じである.

また `red_frac_r` は,

`∀ x y z : R, z ≠ 0 → (x * z) / (y * z) = x / y`

という自分で用意した補題である. この補題の本質は $z/z = 1$ という約分計算であり, `mathcomp` の `ssralg` の補題

`divff : ∀ (F : fieldType) (x : F), x ≠ 0 → x / x = 1`

を用いているため $z \neq 0$ という仮定が必要になる. よって, `Dq_pow` にも $x \neq 0$ という前提を加えている. 今後も補題を形式化するにあたって, その証明の中で約分を行う際には 0 でないという前提を付け加えることになる.

Remark 1.4.1.5 `qnat` という名前であるが, 実際には `n` の型は `nat` ではなく `int` にしている. また, `Dq_of_pow` の `n` の型は `int` であるため, より一般化した形で形式化になっている.

[1] では証明は 1 行で終わっているが, 形式化する場合には何倍もかかっている. これは, 積の交換法則や指数法則などの, 通常の数学では当たり前なことが自動では計算されず, `rewrite mulrC` や `rewrite expfzDr` というように `rewrite` での書き換えを明示的に行わなければならないからである. 一般に, もとの数学の証明と比べてその形式化の方が長くなる.

1.4.2 $(x - a)^n$ の q -類似

続いて $(x - a)^n$ の q -類似を定義し, その性質を調べる.

Definition 1.4.2.1 ([1] p8 Definition (3.4)) $x, a \in \mathbb{R}, n \in \mathbb{Z}_{\geq 0}$ に対して, $(x - a)^n$ の q -類似 $(x - a)_q^n$ を,

$$(x - a)_q^n = \begin{cases} 1 & \text{if } n = 0 \\ (x - a)(x - qa) \cdots (x - q^{n-1}a) & \text{if } n \geq 1 \end{cases}$$

と定義する.

Proposition 1.4.2.2 $n \in \mathbb{Z}_{>0}$ に対し,

$$D_q(x - a)_q^n = [n](x - a)_q^{n-1}$$

が成り立つ.

Proof. n についての帰納法により示される. □

まず, $(x - a)_q^n$ の定義を形式化する.

```
Fixpoint qbinom_pos a n x := match n with
| 0 => 1
| n0.+1 => (qbinom_pos a n0 x) * (x - q ^ n0 * a)
end.
```

Fixpoint を用いて再帰的な定義をしており, **match** を使って n が 0 かどうかで場合分けしている. 再帰で呼び出す際に引数 n が真に小さくなっているため停止性が保証されている. 補題の証明については以下の通り.

Theorem $Dq_qbinom_pos\ a\ n\ x : x \neq 0 \rightarrow$

$Dq\ (qbinom_pos\ a\ n.+1)\ x =$
 $qnat\ n.+1 * qbinom_pos\ a\ n\ x.$

Proof.

```
move=> Hx.
elim: n => [|n IH].
- rewrite /Dq /dq /qbinom_pos /qnat.
  rewrite !mulr mulr1 exprIz.
  rewrite opprB subrKA !divff //.
  by rewrite denom_is_nonzero.
- rewrite (_ : Dq (qbinom_pos a n.+2) x =
    Dq ((qbinom_pos a n.+1) **
      (fun x => (x - q ^ (n.+1) * a))) x) //.
  rewrite Dq_prod' //.
  rewrite [Dq (+%R^~ (- (q ^ n.+1 * a))) x]/Dq /dq.
  rewrite opprB subrKA divff //; last by apply denom_is_nonzero.
  rewrite mulr1 exprSz.
  rewrite -[q * q ^ n * a]mulrA -(mulrBr q) IH.
  rewrite -[q * (x - q ^ n * a) * (qnat n.+1 * qbinom_pos a n x)]mulrA.
  rewrite [(x - q ^ n * a) * (qnat n.+1 * qbinom_pos a n x)]mulrC.
  rewrite -[qnat n.+1 * qbinom_pos a n x * (x - q ^ n * a)]mulrA.
  rewrite (_ : qbinom_pos a n x * (x - q ^ n * a) = qbinom_pos a n.+1 x) //.
  rewrite mulrA -{1}(mulr (qbinom_pos a n.+1 x)).
  by rewrite -mulrDl -qnat_cat1.
```

Qed.

ここで **elim: n** は n の帰納法に対応している.

指数法則については, 一般には $(x - a)^{m+n} \neq (x - a)_q^m (x - a)_q^n$ であり, 以下のようなになる.

Proposition 1.4.2.3 ([1] p8 (3.6)) $x, a \in \mathbb{R}, m, n \in \mathbb{Z}_{>0}$ について,

$$(x - a)_q^{m+n} = (x - a)_q^m (x - q^m a)_q^n$$

が成り立つ.

Proof.

$$\begin{aligned}
(x-a)_q^{m+n} &= (x-a)(x-qa) \cdots (x-q^{m-1}a) \times (x-q^ma)(x-q^{m+1}a) \cdots (x-q^{m+n-1}a) \\
&= (x-a)(x-qa) \cdots (x-q^{m-1}a) \times (x-q^ma)(x-q(q^ma)) \cdots (x-q^{n-1}(q^ma)) \\
&= (x-a)_q^m (x-q^ma)_q^n
\end{aligned}$$

より成立する. □

この形式化は次のとおりである.

Lemma `qbinom_pos_explaw` `x a m n :`

```
qbinom_pos a (m + n) x =
  qbinom_pos a m x * qbinom_pos (q ^ m * a) n x.
```

Proof.

```
elim: n.
- by rewrite addn0 /= mulr1.
- elim => [!|n - IH].
  + by rewrite addnS /= addn0 expr0z !mulr1.
  + rewrite addnS [LHS]/= IH /= !mulrA.
    by rewrite -[q ^ n.+1 * q ^ m] expfz_n0addr // addnC.
```

Qed.

[1] の証明では単に式変形しているが, 形式化の証明では `m, n` に関する帰納法を用いている. これは `qbinom_pos` が再帰的に定義されているため, m 項と n 項で分けるよりも最後の 1 項を取り出す方が簡単であり, 帰納法と相性が良いからである.

この指数法則を用いて, $(x-a)_q^n$ の n を負の数に拡張する. まず, [1] の定義は

Definition 1.4.2.4 ([1] p9 (3.7)) $x, a \in \mathbb{R}, l \in \mathbb{Z}_{>0}$ とする. このとき,

$$(x-a)_q^{-l} := \frac{1}{(x-q^{-l}a)_q^l}$$

と定める.

であり, この形式化は,

Definition `qbinom_neg` `a n x := 1 / qbinom_pos (q ^ ((Negz n) + 1) * a) n x.`

となる. ここで, `Negz n` とは `Negz n = - n.+1` をみたすものであるので, `(Negz n) + 1` は `-n` となり, [1] の定義と一致する. また `int` は

`Variant int : Set := Posz : nat → int | Negz : nat → int.`

のように定義されている. よって, `int` は 0 以上か負かで場合分けできるため, `n: int` に対して `qbinom_pos` の定義を以下のように整数に拡張する.

Definition `qbinom` `a n x :=`

```
match n with
| Posz n0 => qbinom_pos a n0 x
| Negz n0 => qbinom_neg a n0.+1 x
end.
```

整数に拡張した $(x-a)_q^n$ についても, 指数法則と q -微分はうまく振る舞う. まず指数法則について見ていく.

Proposition 1.4.2.5 ([1] p10 Proposition 3.2) $m, n \in \mathbb{Z}$ について, Proposition 1.4.2.3 は成り立つ, つまり

$$(x-a)_q^{m+n} = (x-a)_q^m (x-q^ma)_q^n$$

が成り立つ.

Proof. m, n の正負で場合分けして示す. $m > 0$ かつ $n > 0$ の場合はすでに示しており, $m = n = 0$ の場合は定義からすぐにわかる. その他の場合について, まず $m < 0$ かつ $n \geq 0$ の場合, $m = -m'$ とおくと

$$\begin{aligned}
(x-a)_q^m (x-q^m)_q^n &= (x-a)_q^{-m'} (x-q^{-m'} a)_q^n \\
&= \frac{(x-q^{-m'} a)_q^n}{(x-q^{-m'} a)_q^{m'}} \\
&= \begin{cases} (x-q^{m'} (q^{-m'} a))_q^{n-m'} & n \geq m' \\ \frac{1}{(x-q^n (q^{-m'} a))_q^{m'-n}} & n < m' \end{cases} \\
&= (x-a)_q^{n-m'} \\
&= (x-a)_q^{n+m}
\end{aligned}$$

というように, n と m' の大小で場合分けすることで示せる. 次に, $m \geq 0$ かつ $n < 0$ の場合, $n = -n'$ として,

$$\begin{aligned}
(x-a)_q^m (x-q^m)_q^n &= (x-a)_q^m (x-q^m a)_q^{-n'} \\
&= \begin{cases} \frac{(x-a)_q^{m-n'} (x-q^{m-n'} a)_q^{n'}}{(x-q^{m-n'} a)_q^{n'}} & m \geq n' \\ \frac{(x-a)_q^m}{(x-q^{m-n'} a)_q^{n'-m} (x-q^{n'-m} (q^{m-n'} a))} & m < n' \end{cases} \\
&= \begin{cases} (x-a)_q^{m-n'} & m \geq n' \\ \frac{1}{(x-q^{m-n'} a)_q^{n'-m}} & m < n' \end{cases} \\
&= (x-a)_q^{m-n'} = (x-a)_q^{m+n}
\end{aligned}$$

となる. 最後に, $m < 0$ かつ $n < 0$ のとき, $m = -m', n = -n'$ として,

$$\begin{aligned}
(x-a)_q^m (x-q^m)_q^n &= (x-a)_q^{-m'} (x-q^{-m'} a)_q^{-n'} \\
&= \frac{1}{(x-q^{-m'} a)_q^{m'} (x-q^{-n'-m'} a)_q^{n'}} \\
&= \frac{1}{(x-q^{-n'-m'} a)_q^{n'} (x-q^{n'} (q^{-m'-n'} a))_q^{m'}} \\
&= \frac{1}{(x-q^{-n'-m'} a)_q^{n'+m'}} \\
&= (x-a)_q^{-m'-n'} \\
&= (x-a)_q^{m'+n'}
\end{aligned}$$

となる. □

この補題を形式化すると次のようになる.

Theorem `qbinom_explaw` $a \ m \ n \ x : q \neq 0 \rightarrow$

`qbinom_denom` $a \ m \ x \neq 0 \rightarrow$

`qbinom_denom` $(q^m * a) \ n \ x \neq 0 \rightarrow$

`qbinom` $a \ (m+n) \ x = \text{qbinom } a \ m \ x * \text{qbinom } (q^m * a) \ n \ x.$

Proof.

`move` \Rightarrow `Hq0`.

`case`: $m \Rightarrow m \text{ Hm}.$

- `case`: $n \Rightarrow n \text{ Hn}.$

+ by `apply qbinom_pos_explaw`.

+ `rewrite qbinom_exp_pos_neg` `//`.

by `rewrite addrC expfzDr` `// -mulrA`.

- `case`: $n \Rightarrow n \text{ Hn}.$

+ by `rewrite qbinom_exp_neg_pos`.

+ by `apply qbinom_exp_neg_neg`.

Qed.

証明の構造としては、まず `case: m` で m が 0 以上か負かの場合分けを行い、更にそれぞれの場合について `case: n` で n の場合分けを行っている。ここで、前提の `qbinom_denom` の定義は

```
Definition qbinom_denom a n x :=
  match n with
  | Posz n0 => 1
  | Negz n0 => qbinom_pos (q ^ Negz n0 * a) n0.+1 x
  end.
```

であり、2つの前提は補題の右辺に出現する項の分母が 0 にならないということである。証明中に使われている補題のうち、`qbinom_exp_pos_neg`, `qbinom_exp_neg_pos`, `qbinom_exp_neg_neg` はそれぞれ $m \geq 0$ かつ $n < 0$, $m < 0$ かつ $n \geq 0$, $m < 0$ かつ $n < 0$ のときの証明の形式化であり、例えば `qbinom_exp_pos_neg` については以下の通り。

```
Lemma qbinom_exp_pos_neg a (m n : nat) x : q ≠ 0 →
  qbinom_pos (q ^ (Posz m + Negz n) * a) n.+1 x ≠ 0 →
  qbinom a (Posz m + Negz n) x = qbinom a m x * qbinom (q ^ m * a) (Negz n) x.
```

Proof.

```
move => Hq0 Hqbinommn.
case Hmn : (Posz m + Negz n) => [l|l] /=.
- rewrite /qbinom_neg mul1r.
  rewrite (_ : qbinom_pos a m x = qbinom_pos a (l + n.+1) x).
  rewrite qbinom_pos_explaw.
  have → : q ^ (Negz n.+1 + 1) * (q ^ m * a) = q ^ l * a.
    by rewrite mulrA -expfzDr // -addn1 Negz_addK addrC Hmn.
  rewrite -{2}(mul1r (qbinom_pos (q ^ l * a) n.+1 x)) red_frac_r.
    by rewrite divr1.
  by rewrite -Hmn.
apply Negz_transp in Hmn.
apply (eq_int_to_nat R) in Hmn.
by rewrite Hmn.
- rewrite /qbinom_neg.
  have Hmn' : n.+1 = (l.+1 + m)%N.
  move /Negz_transp /esym in Hmn.
  rewrite addrC in Hmn.
  move /Negz_transp /(eq_int_to_nat R) in Hmn.
  by rewrite addnC in Hmn.
  rewrite (_ : qbinom_pos (q ^ (Negz n.+1 + 1) * (q ^ m * a)) n.+1 x
    = qbinom_pos (q ^ (Negz n.+1 + 1) * (q ^ m * a))
      (l.+1 + m) x).
  rewrite qbinom_pos_explaw.
  have → : q ^ (Negz n.+1 + 1) * (q ^ m * a) =
    q ^ (Negz l.+1 + 1) * a.
    by rewrite mulrA -expfzDr // !NegzS addrC Hmn.
  have → : q ^ l.+1 * (q ^ (Negz l.+1 + 1) * a) = a.
    by rewrite mulrA -expfzDr // NegzS NegzK expr0z mul1r.
  rewrite mulrA.
  rewrite [qbinom_pos (q ^ (Negz l.+1 + 1) * a) l.+1 x *
    qbinom_pos a m x]mulrC.
  rewrite red_frac_l //.
  have → : a = q ^ l.+1 * (q ^ (Posz m + Negz n) * a) => //.
```

```

    by rewrite mulrA -expfzDr // Hmn NegzK expr0z mulr.
  apply qbinom_exp_non0r.
  rewrite -Hmn' //.
  by rewrite Hmn'.

```

Qed.

この証明についての注目点としては,

- [1] では m と n' の大小で場合分けをしていたが, 形式化では,

$$\text{case Hmn : (Posz m + Negz n) } \Rightarrow [1|1] \quad /=.$$
 として, $m - n'$ の値を 1 とおき, 1 が 0 以上かどうかで場合分けをしている
- Coq では $A = B$ という等式はどの型の上でのものなのかが区別されている. `eq_int_to_nat` という補題は `int` 上の等式を `nat` 上の等式に写している.

などが挙げられる.

次に q -微分について見ていく.

Proposition 1.4.2.6 ([1] p10 Proposition 3.3) $n \in \mathbb{Z}$ について,

$$D_q(x - a)_q^n = [n](x - a)_q^{n-1}$$

が成り立つ. ただし, n が整数の場合にも, 自然数のときと同様, $[n]$ の定義は

$$\frac{q^n - 1}{q - 1}$$

である.

Proof. $n > 0$ のときは Proposition 1.4.2.2 であり, $n = 0$ のときは $[0] = 0$ からすぐにわかる. $n < 0$ のときは, Definition 1.4.2.4 と, 商の微分公式の q -類似版である

$$D_q \left(\frac{f(x)}{g(x)} \right) = \frac{g(x)D_q f(x) - f(x)D_q g(x)}{g(x)g(qx)} \quad ([1] \text{ p3 (1.13)})$$

及び Proposition 1.4.2.2 を用いて示される. □

[1] と同じ方針で証明する. まず, $n = 0$ のときは次の通り.

Lemma `Dq_qbinomn0 a x :`

`Dq (qbinom a 0) x = qnat 0 * qbinom a (- 1) x.`

Proof. `by rewrite Dq_const qnat0 mul0r. Qed.`

ここで, `Dq_const` は

Lemma `Dq_const x c : Dq (fun x => c) x = 0.`

という定数関数の q -微分は 0 であるという補題である. 次に, $n < 0$ のときは以下のようになる.

Theorem `Dq_qbinom_neg a n x : q ≠ 0 → x ≠ 0 →`

`(x - q ^ (Negz n) * a) ≠ 0 →`

`qbinom_pos (q ^ (Negz n + 1) * a) n x ≠ 0 →`

`Dq (qbinom_neg a n) x = qnat (Negz n + 1) * qbinom_neg a (n.+1) x.`

Proof.

`move=> Hq0 Hx Hqn Hqbinom.`

`destruct n.`

`- by rewrite /Dq /dq /qbinom_neg /= addrK' qnat0 !mul0r.`

```

- rewrite Dq_quot //.
  rewrite Dq_const mulr0 mulr sub0r.
  rewrite Dq_qbinom_pos // qbinom_qx // -mulNr.
  rewrite [qbinom_pos (q ^ (Negz n.+1 + 1) * a) n.+1 x *
    (q ^ n.+1 * qbinom_pos (q ^ (Negz n.+1 + 1 - 1) *
      a) n.+1 x)] mulrC.
  rewrite -mulf_div.
  have → : qbinom_pos (q ^ (Negz n.+1 + 1) * a) n x /
    qbinom_pos (q ^ (Negz n.+1 + 1) * a) n.+1 x =
    1 / (x - q ^ (-1) * a).
  rewrite -(mulr1 (qbinom_pos (q ^ (Negz n.+1 + 1) * a) n x)) /=.
  rewrite red_frac_l.
    rewrite NegzE mulrA -expfzDr // addrA -addn2.
    rewrite ( _ : Posz (n + 2)%N = Posz n + 2) //.
    by rewrite -{1}(add0r (Posz n)) addrKA.
    by rewrite /=; apply mulnon0 in Hqbinom.
  rewrite mulf_div.
  rewrite -[q ^ n.+1 *
    qbinom_pos (q ^ (Negz n.+1 + 1 - 1) * a) n.+1 x *
    (x - q ^ (-1) * a)]mulrA.
  have → : qbinom_pos (q ^ (Negz n.+1 + 1 - 1) * a) n.+1 x *
    (x - q ^ (-1) * a) =
    qbinom_pos (q ^ (Negz (n.+1)) * a) n.+2 x ⇒ /=.
  have → : Negz n.+1 + 1 - 1 = Negz n.+1.
    by rewrite addrK.
  have → : q ^ n.+1 * (q ^ Negz n.+1 * a) = q ^ (-1) * a ⇒ //.
  rewrite mulrA -expfzDr // NegzE.
  have → : Posz n.+1 - Posz n.+2 = - 1 ⇒ //.
  rewrite -addn1 -[(n + 1).+1]addn1.
  rewrite ( _ : Posz (n + 1)%N = Posz n + 1) //.
  rewrite ( _ : Posz (n + 1 + 1)%N = Posz n + 1 + 1) //.
  rewrite -(add0r (Posz n + 1)).
  by rewrite addrKA.
  rewrite /qbinom_neg /=.
  rewrite ( _ : Negz n.+2 + 1 = Negz n.+1) // -mulf_div.
  congr ( _ * _).
  rewrite NegzE mulrC /qnat -mulNr mulrA.
  congr ( _ / _).
  rewrite oprB mulrBr mulr1 mulrC divff; last by rewrite expnon0.
  rewrite invr_expz ( _ : - Posz n.+2 + 1 = - Posz n.+1) //.
  rewrite -addn1 ( _ : Posz (n.+1 + 1)%N = Posz n.+1 + 1) //.
  by rewrite addrC [Posz n.+1 + 1]addrC -{1}(add0r 1) addrKA sub0r.
  rewrite qbinom_qx // mulf_neq0 //.
  by rewrite expnon0.
  rewrite qbinom_pos_head mulf_neq0 //.
  rewrite ( _ : Negz n.+1 + 1 - 1 = Negz n.+1) //.
  by rewrite addrK.
  move: Hqbinom ⇒ /=.
  move/mulnon0.
  by rewrite addrK mulrA -{2}(expriz q) -expfzDr.

```

Qed.

非常に長くなっているが積の交換則や結合則などが多く, `Dq_quot` が商の q -微分公式の形式化であるため, [1] の証明をそのまま形式化したものになっている. また, いくつかの項が 0 でないという条件がついているが, これらの項は Definition 1.4.2.4 において分母に現れるため, `Dq_of_pow` のときと同様妥当であると考えられる. これらをまとめて以下のように形式化できる.

Theorem `Dq_qbinom a n x : q ≠ 0 → x ≠ 0 →`
`x - q ^ (n - 1) * a ≠ 0 →`
`qbinom (q ^ n * a) (- n) x ≠ 0 →`
`Dq (qbinom a n) x = qnat n * qbinom a (n - 1) x.`

Proof.

```
move⇒ Hq0 Hx Hxqa Hqbinom.
case: n Hxqa Hqbinom ⇒ [|/=] n Hxqa Hqbinom.
- destruct n.
+ by rewrite Dq_qbinomn0.
+ rewrite Dq_qbinom_pos //.
  rewrite (_ : Posz n.+1 - 1 = n) // -addn1.
  by rewrite (_ : Posz (n + 1)%N = Posz n + 1) ?addrK.
- rewrite Dq_qbinom_int_to_neg Dq_qbinom_neg //.
  rewrite Negz_addK.
  rewrite (_ : (n + 1).+1 = (n + 0).+2) //.
  by rewrite addn0 addn1.
  rewrite (_ : Negz (n + 1) = Negz n - 1) //.
  by apply itransposition; rewrite Negz_addK.
  by rewrite Negz_addK addn1.
```

Qed.

`case: n` で n が 0 以上か負かで場合分けを, `destruct n` で 0 か 1 以上かの場合分けをしており, それぞれの場合で `Dq_qbinom_0`, `Dq_qbinom_pos`, `Dq_qbinom_neg` を使っていることが見て取れる.

1.4.3 関数から多項式へ

ここまでは [1] の定義に沿って関数に対して q -微分を定義し, また関数として $(x - a)_q^n$ を定義してきたが, q -Taylor 展開や Gauss's binomial formula を形式化するに当たって多項式に対する q -微分と多項式としての $(x - a)_q^n$ を改めて定義する.

ここで, Coq における多項式の扱いについて説明する.

- `{poly T}` で T 係数多項式を表す型となる. T は `ringType` でなくてはならないが, `rcfType` は `ringType` の構造を引き継いでいるため, 今回用いている R に対して `{poly R}` が定義できる.
- `{poly R}` は `ringType` と `lmodType` の構造を持っている. よって, 前者の性質から今まで使ってきた `ringType` に対する補題がそのまま使え, 後者からスカラー倍 `a *: p` (ここで `a: R`, `p: {poly R}` である) も定義される.
- `\poly (i < n) E(i)` で, 次数が $n - 1$ 次以下, i 次の係数が $E(i)$ である多項式を表す.
- `c%:P` で定数 c のみからなる単項式を表す.
- `'X` で変数 x のみからなる単項式を表す.
- `p'.i` で多項式 p の i 次の係数を表す.
- `size p` で多項式 p の次数 $+1$ を表す. `size` が 0 である多項式はゼロ多項式のみであり, `size` が 1 である多項式は定数項のみからなる式である.

- $p.[x]$ で多項式 p の x での値を表す.

より詳細な内容については `mathcomp` の `poly.v` を参照のこと.

この `{poly R}` を用いて `Dq` や `qbinom` を定義し直していく. まず, q -微分について, 多項式に対する d_q を以下のように定義し直す.

Definition `scale_var (p : {poly R}) := \poly_(i < size p) (q ^ i * p'_i).`

Definition `dqp p := scale_var p - p.`

`scale_var` は多項式 p を受け取り, i 次の係数を q^i 倍した多項式を返す操作である. また, `dqp` は多項式に対しての d_q と同じ結果になることが確認できる (正確には, `dqp` を適用した多項式での x での値と $x \mapsto p.[x]$ という関数に d_q を適用した関数の x での値が等しいということである).

Definition `ap_op_poly (D : (R → R) → (R → R)) (p : {poly R}) := D (fun (x : R) ⇒ p.[x]).`

Notation `"D # p" := (ap_op_poly D p) (at level 49).`

Lemma `dqp_dqE p x : (dqp p).[x] = (dq # p) x.`

この `dqp` を用いて, 多項式に対する D_q を定義する.

Definition `Dqp p := dqp p %/ dqp 'X.`

$p \% p'$ は多項式 p を多項式 p' で割った商を表している. この定義だけでは `dqp` を `dq 'X` で割った余りが 0 でない可能性があるため, q -微分の正しい形式化である保証がない. しかし実際に多項式に対して `Dqp` を計算すると, `dqp` の定義から, `dqp p` は定数項が打ち消しあい, また `dqp 'X` は $(q - 1) * 'X$ となるので割り切れるはずである. よってこのことを証明しておく.

Lemma `Dqp_ok p : dqp 'X %| dqp p.`

ここで, $p' \% p$ で p が p' で割り切れることを表す. 今後は扱いやすさのため, $'X$ で約分した形

Definition `Dqp' (p : {poly R}) := \poly_(i < size p) (qnat (i.+1) * p'_{i.+1}).`

を用いる. このとき, `Dqp` と `Dqp'` が等しいことも示せる.

Lemma `Dqp_Dqp'E p : Dqp p = Dqp' p.`

また, `dqp` のときと同様, 多項式に対しての D_q と同じであることを確認しておく.

Lemma `Dqp'_DqE p x : x ≠ 0 → (Dqp' p).[x] = (Dq # p) x.`

Remark 1.4.3.1 `Dqp_Dqp'E` には特に条件がなく, `Dqp'_DqE` には $x \neq 0$ という条件がついている. この違いは, 前者は $'X / 'X = 1\%P$ という多項式での約分を, 後者は $x / x = 1$ という実数での約分を行っているということから生じている. 後で詳しく述べるが, 約分の際に条件が必要なくなることが多項式で考える利点の一つである.

次に, $(x - a)_q^n$ を多項式として以下のように定義しなおす.

Fixpoint `qbinom_pos_poly a n := match n with`
`| 0 ⇒ 1`
`| n.+1 ⇒ (qbinom_pos_poly a n) * ('X - (q ^ n * a)%:P)`
`end.`

この多項式の x での値は元の定義の `qbinom_pos` と等しくなる.

Lemma `qbinom_posE a n x :`

`qbinom_pos a n x = (qbinom_pos_poly a n).[x].`

更に, このように定義した `Dqp` と `qbinom_pos_poly` に対しても Proposition 1.4.2.2 と同じことが成り立つ.

Lemma `Dqp'_qbinom_poly a n :`

`Dqp' (qbinom_pos_poly a n.+1) = (qnat n.+1) *: (qbinom_pos_poly a n).`

Remark 1.4.3.2 証明の方針はこれまでの関数としての場合と同じだが, `Dq_prod'` (q -微分の積の法則) に対応する補題の証明のため, `scale_var` が積について分解できること, つまり

Lemma `scale_var_prod (p p' : {poly R}) : scale_var (p * p') = scale_var p * scale_var p'.`

を示している. ここで証明の冒頭を抜き出すと以下のようにになっている.

Proof.

```
pose n := size p.
have : (size p ≤ n)%N by [].
clearbody n.
have Hp0 : ∀ (p : {poly R}), size p = 0%N →
  scale_var (p * p') = scale_var p * scale_var p'.
  move⇒ p0 /eqP.
  rewrite size_poly_eq0.
  move/eqP →.
  by rewrite mul0r scale_varC mul0r.
elim: n p ⇒ [|n IH] p Hsize.
...
```

Qed.

`pose n := size p.` で多項式 p の `size` を n と置いており, `have: (size p ≤ n)%N by [].` で `size p` が n 以下という自明な主張をあえて置いているが, これは多項式の `size` に関する帰納法を用いるためである. このように, 当たり前の内容を明示的に書かなければならないことに加え, 形式化するための証明の構造を工夫しなければならない場合もある.

多項式で考える理由は, q -Taylor 展開が多項式に対する定理であることに加え, 以下の 2 つの目的がある.

1. $x/x = 1$ の計算に $x \neq 0$ という条件が必要ない

先に見たように, Coq で約分の計算, つまり $x/x = 1$ を行う際には $x \neq 0$ という条件が必要である. よって, 実数 x に対して $x/x = 1$ を計算する場合, 後から $x = 0$ を代入することはできない. しかし, 多項式で考える場合, ' X ' は単項式であるためゼロ多項式とは異なるので, ' $X \neq 0$ ' という条件は自動的にみたされることになり, ' $X / 'X = 1\%P$ ' の計算には特に条件が必要ない. よって, ' X ' で約分した後でも 0 での値を計算できる. 例えば $D_q(x + a)_q^n = [n](x + a)_q^{n-1}$ という計算には x での約分が必要であるが, 多項式として考える場合には上の計算をした後でも 0 での値を求めることができる. この値は本論文の目的である Gauss's binomial formula の証明に必要である.

2. $q = 0$ のとき高階 D_q が定義できる

$q = 0$ のときに 2 階 D_q を計算してみると

$$\begin{aligned}
(D_q^2 f)(x) &= (D_0^2 f)(x) = (D_0(D_0 f))(x) \\
&= D_0 \left(\lambda x. \frac{f(0x) - f(x)}{(0-1)x} \right)(x) \\
&= D_0 \left(\lambda x. \frac{f(x) - f(0)}{x} \right)(x) \\
&= (D_0 F)(x) \quad (\text{ここで } F := \lambda x. \frac{f(x) - f(0)}{x} \text{ とおいた}) \\
&= \lambda x. \frac{F(x) - F(0)}{x}(x) \\
&= \frac{F(x) - F(0)}{x}
\end{aligned}$$

となるが,

$$F(0) = \frac{f(0) - f(0)}{0} = \frac{0}{0}$$

となってしまう (Coq では $0/0$ は 0 と定義されているが, これでも正しい計算結果とはならない). この問題が起きるのはもともとの \mathbf{dq} を関数の引数に対して各点ごとに定義しているからであり, 多項式の係数を変化させることで定義している \mathbf{dqp} では $q = 0$ でも問題が起きない. よってこの \mathbf{dqp} を用いている \mathbf{Dqp} および \mathbf{Dqp}' については $q = 0$ かどうかにかかわらず高階 q -微分を定義できる.

1.4.4 q -Taylor 展開

この節では, 有限次 Taylor 展開の q -類似が成り立つこと, そしてその系として Gauss's binomial formula が成り立つことを示し, 形式化する. まず, 一般に以下のことが成り立つことを確認しておく.

Theorem 1.4.4.1 ([1] p5 Theorem 2.1) $\mathbb{K} := \mathbb{R}$ または \mathbb{C} , $V := \mathbb{K}[x]$ とし, D を V 上の線型作用素とする. また, $\{P_n(x)\}_{n=0} \subset V$ ($n = 0, 1, 2, \dots$) は次の三条件をみたすとする.

- (i) $P_0(a) = 1, P_n(a) = 0 \quad (\forall n \geq 1)$
- (ii) $\deg P_n = n \quad (\forall n \geq 0)$
- (iii) $DP_n(x) = P_{n-1}(x) \quad (\forall n \geq 1), \quad D(1) = 0$

ただし, $a \in \mathbb{K}$ である. このとき, 任意の多項式 $f(x) \in V$ に対し, $\deg f(x) = N$ とすると,

$$f(x) = \sum_{n=0}^N (D^n f)(a) P_n(x)$$

が成り立つ.

この定理を形式化すると以下ようになる.

Theorem `general_Taylor D n P (f : {poly R}) a :`

```

islinear D → isfderiv D P →
(P 0%N).[a] = 1 →
(∀ n, (P n.+1).[a] = 0) →
(∀ m, size (P m) = m.+1) →
size f = n.+1 →
f = \sum_ (0 ≤ i < n.+1)
  ((D ^ i) f).[a] *: P i.

```

記号の意味などは以下の通りである.

- `islinear, isfderiv` はそれぞれ

Definition `islinear` ($D : \{\text{poly } R\} \rightarrow \{\text{poly } R\}$) :=
 $\forall a b f g, D ((a * f) + (b * g)) = a * D f + b * D g.$

Definition `isfderiv` $D (P : \text{nat} \rightarrow \{\text{poly } R\}) := \forall n,$
`match n with`
`| 0 => (D (P n)) = 0`
`| n.+1 => (D (P n.+1)) = P n`
`end.`

という定義であり, 前者が線形作用素であること, 後者は条件 (iii) を形式化したものである.

- [1] での証明には, $\{P_0(x), P_1(x), \dots, P_n(x)\}$ が V の基底となることを用いている. これを以下のように形式化した.

Lemma `poly_basis` $n (P : \text{nat} \rightarrow \{\text{poly } R\}) (f : \{\text{poly } R\}) :$
 $(\forall m, \text{size } (P m) = m.+1) \rightarrow$
 $(\text{size } f \leq n.+1) \rightarrow$
 $\exists (c : \text{nat} \rightarrow R), f = \sum_{(0 \leq i < n.+1)} c i * P i.$

この主張には係数列 c の一意性は含まれていないため, 実際には生成系であることを示している.

この定理において,

$$D \equiv D_q, \quad P_n \equiv \frac{(x-a)_q^n}{[n]!}$$

(ただし, $n \in \mathbb{Z}_{\geq 0}$ に対し, $[n]!$ を

$$[n]! := \begin{cases} 1 & (n = 0) \\ [n] \times [n-1] \times \dots \times [1] & (n \geq 1) \end{cases}$$

と定める) とすることで, 有限次 Taylor 展開の q -類似が得られる.

Theorem 1.4.4.2 ([1] p12 Theorem 4.1) $f(x)$ を, N 次の実数係数多項式とする. 任意の $c \in \mathbb{R}$ に対し,

$$f(x) = \sum_{j=0}^N (D_q^j f)(c) \frac{(x-c)_q^j}{[j]!}$$

が成り立つ.

Proof. $\frac{(x-a)_q^n}{[n]!}$ が, a, D_q に対して Theorem 1.4.4.1 の三条件をみたすことを確かめればよい. (i), (ii) は $(x-a)_q^n$ の定義から, (iii) は Proposition 1.4.2.2 から分かる. \square

前節で準備した `Dqp, qbinom_pos_poly` を用いて Theorem 1.4.4.2 を形式化する.

Fixpoint `qfact` $n := \text{match } n \text{ with}$
`| 0 => 1`
`| n.+1 => qfact n * qnat n.+1`
`end.`

Theorem `q_Taylorp` $n (f : \{\text{poly } R\}) c :$
 $(\forall n, \text{qfact } n \neq 0) \rightarrow$
 $\text{size } f = n.+1 \rightarrow$
 $f = \sum_{(0 \leq i < n.+1)} ((Dqp' \wedge^i) f).[c] * (qbinom_pos_poly \ c \ i \ / \ (qfact \ i) \% P).$

`Dqp, qbinom_pos_poly` をもとの定義に戻したものについては,

Theorem `q_Taylor n (f : {poly R}) x c :`

```

q ≠ 0 →
c ≠ 0 →
(∀ n, qfact n ≠ 0) →
size f = n.+1 →
f.[x] = \sum_(0 ≤ i < n.+1)
      ((Dq \^ i) # f) c * qbinom_pos c i x / qfact i.

```

Remark 1.4.4.3 約分のための $c \neq 0$ という条件に加え, 高階 D_q を扱うため前述の通り $q \neq 0$ も必要となる. 具体的には高階 $D_{qp'}$ と D_q を一致させる補題

Lemma `hoDqp'_DqE p x n : q ≠ 0 → x ≠ 0 →`
`((Dqp' \^ n) p).[x] = ((Dq \^ n) # p) x.`

Proof.

```

move⇒ Hq0 Hx.
rewrite /(_ # _).
elim: n x Hx ⇒ [|n IH] x Hx //=.
rewrite Dqp'_DqE // {2}/Dq /dq -!IH //.
by apply mulf_neq0 ⇒ //.

```

Qed.

の証明において, IH(Inductive Hypothesis, 帰納の仮定) を使う際に $q * x \neq 0$ という条件が必要となる.

本論文の最後に, x^n と $(x-a)_q^n$ にこの Taylor 展開の q -類似を適用する.

Lemma 1.4.4.4 ([1] p12 Example (4.4)) $n \in \mathbb{Z}_{>0}$ について,

$$x^n = \sum_{j=0}^n \left[\begin{matrix} n \\ j \end{matrix} \right] (x-1)_q^j \quad \left(\text{ここで, } \left[\begin{matrix} n \\ j \end{matrix} \right] := \frac{[n]!}{[j]![n-j]!} \right)$$

が成り立つ.

Proof. Theorem 1.4.4.2 において, $f(x) = x^n, c = 1$ とする. 任意の正整数 $j \leq n$ に対して, $D_q x^n = [n]x^{n-1}$ より,

$$(D_q^j f)(x) = [n][n-1] \cdots [n-j+1] x^{n-j}$$

となるので,

$$(D_q^j f)(1) = [n][n-1] \cdots [n-j+1]$$

が得られる. □

Lemma 1.4.4.5 ([1] p15 Example (5.5)) $n \in \mathbb{Z}_{>0}$ について,

$$(x+a)_q^n = \sum_{j=0}^n \left[\begin{matrix} n \\ j \end{matrix} \right] q^{j(j-1)/2} a^j x^{n-j}$$

が成り立つ. この式は Gauss's binomial formula と呼ばれる.

Proof. $f = (x+a)_q^n$ とすると, 任意の正整数 $j \leq n$ に対して,

$$(D_q^j f)(x) = [n][n-1] \cdots [n-j+1] (x+a)_q^{n-j}$$

であり, また

$$(x+a)_q^m = (x+a)(x+qa) \cdots (x+q^{m-1}a)$$

から, $(0 + a)_q^m = a \cdot qa \cdots q^{m-1}a = q^{m(m-1)/2}a^m$ となるので,

$$(D_q^j f)(0) = [n][n-1] \cdots [n-j+1] q^{(n-j)(n-j-1)/2} a^{n-j}$$

が成り立つ. よって, Theorem 1.4.4.2 において, $f = (x + a)_q^n$, $c = 0$ として,

$$(x + a)_q^n = \sum_{j=0}^n \begin{bmatrix} n \\ j \end{bmatrix} q^{(n-j)(n-j-1)/2} a^{n-j} x^j$$

が得られる. この式の右辺において j を $n-j$ に置き換えることで,

$$\begin{bmatrix} n \\ n-j \end{bmatrix} = \frac{[n]!}{[n-j]![n-(n-j)]!} = \frac{[n]!}{[j]![n-j]!} = \begin{bmatrix} n \\ j \end{bmatrix}$$

に注意すれば,

$$(x - a)_q^n = \sum_{j=0}^n \begin{bmatrix} n \\ j \end{bmatrix} q^{j(j-1)/2} a^j x^{n-j}$$

が成り立つ. □

この二つの等式の形式化はそれぞれ次の通り.

Lemma `q_Taylorp_pow` $n : (\forall n, \text{qfact } n \neq 0) \rightarrow$
 $'X^n = \sum_{(0 \leq i < n.+1)} (\text{qbicoef } n \ i * : \text{qbinom_pos_poly } 1 \ i).$

Definition `qbicoef` $n \ j := \text{qfact } n / (\text{qfact } j * \text{qfact } (n - j)).$

Theorem `Gauss_binomial` $a \ n : (\forall n, \text{qfact } n \neq 0) \rightarrow$
`qbinom_pos_poly` $(-a) \ n =$
 $\sum_{(0 \leq i < n.+1)} (\text{qbicoef } n \ i * q^{+(i * (i - 1))./2 * a^{+ i}} * : 'X^{(n - i)}).$

Remark 1.4.4.6 `Gauss_binomial` は `q_Taylorp` において $c = 0$ として証明している. `q_Taylorp` では約分の計算をしているが, 多項式を用いて定義しているため 0 での値を計算できる.

第 2 章

少人数クラスまとめ

2.1 はじめに

本章では, [9] を教科書にして修士 2 年次に少人数クラスで学習した Homotopy Type Theory (HoTT) についてまとめる. HoTT とは,

$$\begin{aligned} a \text{ が型 } A \text{ の要素である} &\leftrightarrow a \text{ が空間 } A \text{ の点である} \\ a = b \text{ である} &\leftrightarrow \text{点 } a \text{ と点 } b \text{ の間にパスが存在する} \end{aligned}$$

というように, 型理論に対してホモトピー的解釈を与えたものである. 2.4 節で HoTT の大きな特徴の一つである, univalence axiom について説明する. 大雑把に言えば, univalence axiom は「型 A と型 B が同型ならば, A と B は等しい」という公理である. この意味を正確にとらえるため, 型同士の等しさや同型を定義していく. また, いくつかの定義や補題を準備した後, univalence axiom から関数の外延性がしたがうことを 2.8 節で確認する.

2.2 型から型を作る

A と B の 2 つの型が与えられたとき, そこから関数型 $A \rightarrow B$ が構成できる. このとき,

$$\begin{aligned} f : A \rightarrow B, a : A &\Longrightarrow f(a) : B \\ a : A, b(x) : B &\Longrightarrow \lambda a. b : A \rightarrow B \end{aligned}$$

である. より一般に, 型 A と A 上の型族 $B \rightarrow \mathcal{U}$ が与えられれば (\mathcal{U} はユニバース), 依存関数型 $\prod_{a:A} B(a)$ が構成でき,

$$\begin{aligned} f : \prod_{a:A} B(a), a : A &\Longrightarrow f(a) : B(a) \\ a : A, b(x) : B(x) &\Longrightarrow \lambda a. b : \prod_{a:A} B(a) \end{aligned}$$

である. さらに, 既存の型から新たな型を作るやり方として, 構成規則, 導入規則, 除去規則, 計算規則の 4 つを与える帰納的な方法がある. 例えば, 依存和型 $\sum_{x:A} B(x)$ は,

- 構成規則 : $A : \mathcal{U}, B : A \rightarrow \mathcal{U} \Longrightarrow \sum_{x:A} B(x)$
- 導入規則 : $a : A, b : B(a) \Longrightarrow (a, b) : \sum_{x:A} B(x)$
- 除去規則 : $\text{ind}_{\sum_{x:A} B(x)} : \prod_{C : (\sum_{x:A} B(x)) \rightarrow \mathcal{U}} \left(\prod_{a:A} \prod_{b:B(a)} C((a, b)) \right) \rightarrow \prod_{w : \sum_{x:A} B(x)} C(w)$
- 計算規則 : $\text{ind}_{\sum_{x:A} B(x)}(C, g, (a, b)) \equiv g(a)(b)$

で定義できる. 除去規則は, 「任意の $w : \sum_{x:A} B(x)$ について $C(w)$ を示したければ, 任意の $a : A, b : B(a)$ について $C((a, b))$ を示せばよい」と読むことができる. ここで, Curry-Howard 同型に基づいて考えると, 「ある要素 a とある要素 b が等しい」という命題は, なにかしらの型と対応するはずである. よってその型 identity type を,

- 構成規則 : $A : \mathcal{U} \implies _ =_A _ : \mathcal{U}$
- 導入規則 : $\text{refl}_a : \prod_{a:A} (a =_A a)$
- 除去規則 : $\text{ind}_{=_A} : \prod_{(C : \prod_{(x,y:A)} (x =_A y) \rightarrow \mathcal{U})} \left(\prod_{(x:A)} C(x, x, \text{refl}_x) \right) \rightarrow \prod_{(x,y:A)} \prod_{(p:x =_A y)} C(x, y, p)$
- 計算規則 : $\text{ind}_{=_A}(C, c, x, x, \text{refl}_x) \equiv c(x)$

と定義する. 除去規則は, 依存和型のと看同様に考えると, 「任意の $x, y : A$, $x = y$ について $C(x, y, p)$ を示したければ, 任意の $x : A$ について $C(x, x, \text{refl}_x)$ を示せばよい」となる.

2.3 型の同型

ここで, 型と型の間同型を定義したい. まず, 関数の間のホモトピーを定義する.

Definition 2.3.0.1 ([9] Definition 2.4.1) $A : \mathcal{U}$, $P : A \rightarrow \mathcal{U}$ とする. $f, g : \prod_{x:A} P(x)$ に対して,

$$(f \sim g) := \prod_{x:A} (f(x) = g(x))$$

と定める.

次に, 「逆写像」を定義する.

Definition 2.3.0.2 ([9] Definition 2.4.6) $A, B : \mathcal{U}$, $f : A \rightarrow B$ とする. このとき, f の quasi-inverse $\text{qinv}(f)$ を,

$$\text{qinv}(f) := \sum_{g:B \rightarrow A} ((f \circ g \sim \text{id}_B) \times (g \circ f \sim \text{id}_A))$$

例えば, id_A の quasi-inverse は id_A 自身である. さらに, この qinv を用いて, isequiv を,

- $\text{qinv}(f) \rightarrow \text{isequiv}(f)$
- $\text{isequiv}(f) \rightarrow \text{qinv}(f)$
- $e_1, e_2 : \text{isequiv}(f)$ ならば $e_1 = e_2$

をみたすものとして定義したい. ここでは,

$$\text{isequiv}(f) := \left(\sum_{g:B \rightarrow A} (f \circ g \sim \text{id}_B) \right) \times \left(\sum_{h:A \rightarrow B} (h \circ f \sim \text{id}_A) \right) \quad ([9] \text{ p73 (2.4.10)})$$

と定めることにする. isequiv を使って型同士の同型を定義する.

Definition 2.3.0.3 ([9] p73 (2.4.11)) $A, B : \mathcal{U}$ について,

$$A \simeq B := \sum_{f:A \rightarrow B} \text{isequiv}(f)$$

と定める.

型の同型については, 例えば以下のような例がある.

Example 2.3.0.4 ([9] Lemma 2.4.12) $A, B, C : \mathcal{U}$ について,

- $A \simeq A$
- $A \simeq B \rightarrow B \simeq A$
- $A \simeq B \rightarrow B \simeq C \rightarrow A \simeq C$

が成り立つ.

Example 2.3.0.5 ([9] Exercise 2.10) Σ 型は「結合的」である. つまり任意の型 A , 型族 $B : A \rightarrow \mathcal{U}$, 型族上の型

族 $C : (\sum_{x:A} B(x)) \rightarrow \mathcal{U}$ に対して,

$$\left(\sum_{x:A} \sum_{y:B(x)} C((x, y)) \right) \simeq \left(\sum_{p:\sum_{x:A} B(x)} C(p) \right)$$

が成り立つ.

2.4 Univalence axiom

これまでに定義した $=$ と \simeq を用いて, univalence axiom の主張を正しく述べる. まず,

$$\text{idtoeqv} : \prod_{A, B: \mathcal{U}} (A =_{\mathcal{U}} B) \rightarrow (A \simeq B) \quad ([9] \text{ p89 (2.10.2)})$$

を定める. この関数が存在することは, path induction よりわかる ([9] Lemma 2.10.1). この idtoeqv に対して,

Axiom 2.4.0.1 ([9] Axiom 2.10.3)

$$\text{ua} : \prod_{A, B: \mathcal{U}} \text{isequiv}(\text{idtoeqv}(A, B))$$

が univalence axiom である. とくに, この公理を仮定すれば,

$$(A =_{\mathcal{U}} B) \simeq (A \simeq B)$$

が成り立つ. 2.2 章のときのように, ua や idtoequiv を $A =_{\mathcal{U}} B$ という型を構成する規則だと考えれば,

- 導入規則 : $A : \mathcal{U}, B : \mathcal{U} \implies \text{ua} : (A \simeq B) \rightarrow (A =_{\mathcal{U}} B)$
- 除去規則 : $\text{idtoequiv} \equiv \text{transport}^{X \mapsto X} : (A =_{\mathcal{U}} B) \rightarrow (A \simeq B)$
- (propositional) 計算規則 : $\text{idtoequiv}(\text{ua}(f), x) = f(x)$
- (propositional) 一意性 : 任意の $p : A = B$ について, $p = \text{ua}(\text{idtoequiv}(p))$

となる (transport の定義は [9] p72 Lemma 2.3.1). univalence axiom をみたすようなユニバース \mathcal{U} を univalent であるという.

2.5 関数の外延性

関数 f, g について,

$$f = g \leftrightarrow \text{任意の } f, g \text{ の定義域の要素 } x \text{ に対して, } f(x) = g(x)$$

という関数の外延性の公理について述べる. 正確には, まず path induction から, 依存関数 $f, g : \prod_{x:A} B(x)$ に対して

$$\text{happly} : (f = g) \rightarrow (f \sim g) \left(\prod_{x:A} (f(x) = g(x)) \right) \quad ([9] \text{ p86 (2.9.2)})$$

という関数を定義できる. このとき,

Axiom 2.5.0.1 ([9] Axiom 2.9.3) 任意の A, B, f, g について, happly は equivalence である

を関数の外延性と呼ぶ. この公理から happly の quasi-inverse

$$\text{funext} : \left(\prod_{x:A} (f(x) = g(x)) \right) \rightarrow (f = g)$$

の存在が従う. この関数のことを関数の外延性と呼ぶこともある. univalence axiom のときと同じように考えると,

- 導入規則 : funext
- 除去規則 : happly
- (propositional) 計算規則 : $\text{happly}(\text{funext}(h), x) = h(x)$ ($h : \prod_{x:A} (f(x) = g(x))$)
- (propositional) 一意性 : $p = \text{funext}(x \mapsto \text{happly}(p, x))$ ($p : f = g$)

となる.

2.6 可縮, ファイバー

この章と次の章では, univalence axiom から関数の外延性が従うことを示すために必要な定義や補題の準備をする. まずは, 型が可縮であるということを定義する.

Definition 2.6.0.1 ([9] Definition 3.11.1) A を型とする. 中心と呼ばれる $a : A$ が存在して, 任意の $x : A$ に対して $a = x$ をみたすとき, A は可縮 (contractible) であるという. このことを表す型 $\text{isContr}(A)$ を,

$$\text{isContr}(A) := \sum_{a:A} \prod_{x:A} (a = x)$$

と定める.

次に, ホモトピー論ではホモトピーファイバーに対応する概念を定める.

Definition 2.6.0.2 ([9] Definition 4.2.4) 関数 $f : A \rightarrow B$ の点 $y : B$ の上でのファイバーを

$$\text{fib}_f(y) := \sum_{x:A} (f(x) = y)$$

と定める.

このファイバーを用いて, 関数の可縮性を定義する.

Definition 2.6.0.3 ([9] Definition 4.4.1) 関数 $f : A \rightarrow B$ が可縮であるとは, 任意の $y : B$ に対して $\text{fib}_f(y)$ が可縮であることである. このことを表す型 $\text{isContr}(f)$ を,

$$\text{isContr}(f) := \prod_{y:B} \text{isContr}(\text{fib}_f(y))$$

と定める.

このように定義した $\text{isContr}(f)$ について,

$$\text{isContr}(f) \simeq \text{isequiv}(f)$$

が成り立つ ([9] p138 4.5 節参照). よって, ある関数が equivalence であることを示すには, 可縮であることを示せば十分である. 可縮性に関するものでよく使う補題として, 以下のようなものがある.

Lemma 2.6.0.4 ([9] Lemma 3.11.8) 任意の $A, a : A$ について, $\sum_{x:A} (a = x)$ は可縮である

Lemma 2.6.0.5 ([9] Lemma 3.11.9) $P : A \rightarrow \mathcal{U}$ を型族とする. このとき, 以下の2つが成り立つ.

1. 各 $P(x)$ が可縮である, $\sum_{x:A} P(x) \simeq A$
2. A が a を中心として可縮であるとき, $\sum_{x:A} P(x) \simeq P(a)$

この2つの補題と univalence axiom から示せることとして,

Lemma 2.6.0.6 ([9] Lemma 4.8.1) 任意の型族 $B : A \rightarrow \mathcal{U}$ について, $\text{pr}_1 : \sum_{x:A} B(x) \rightarrow A$ の $a : A$ 上でのファイバーは $B(a)$ と同型である, つまり,

$$\text{fib}_{\text{pr}_1}(a) \simeq B(a)$$

が成り立つ.

Proof.

$$\begin{aligned} \text{fib}_{\text{pr}_1}(a) &\equiv \sum_{u: \sum_{x:A} B(x)} \text{pr}_1(u) a \\ &\simeq \sum_{x:A} \sum_{b:B(x)} (x = a) \quad (\text{by Example 2.3.0.5}) \\ &\simeq \sum_{x:A} \sum_{b:B(x)} \sum_{p:x=a} \mathbf{1} \\ &\simeq \sum_{x:A} \sum_{p:x=a} \sum_{b:B(x)} \mathbf{1} \\ &\simeq \sum_{x:A} \sum_{p:x=a} B(x) \\ &\simeq B(a) \quad (\text{by Example 2.3.0.5, Lemma 2.6.0.4, Lemma 2.6.0.5}) \end{aligned}$$

□

がある. この補題は, 写像はファイブレーションと同型であるというホモトピー論での基本的な結果に対応している. 最後に, ファイバーごとの equivalence は全空間 (total space) の equivalence という言葉で特徴付けられることを示す.

Definition 2.6.0.7 ([9] Definition 4.7.5) 型族 $P, Q : A \rightarrow \mathcal{U}$ と依存関数 $f : \prod_{x:A} P(x) \rightarrow Q(x)$ (このような関数を fiberwise map または fiberwise transformation と呼ぶ) が与えられたとき,

$$\text{total}(f) := \lambda w. (\text{pr}_1 w, f(\text{pr}_1 w, \text{pr}_2 w)) : \sum_{x:A} P(x) \rightarrow \sum_{x:A} Q(x)$$

と定める.

Theorem 2.6.0.8 ([9] Theorem 4.7.6) f が A 上の型族 P と Q の間の fiberwise transformation (つまり $f : \prod_{x:A} P(x) \rightarrow Q(x)$) であり, $x : A$ と $v : Q(x)$ が与えられたとする. このとき,

$$\text{fib}_{\text{total}(f)}((x, v)) \simeq \text{fib}_{f(x)}(v)$$

である.

Proof.

$$\begin{aligned}
\text{fib}_{\text{total}(f)}((x, v)) &\equiv \sum_{w: \sum_{x:A} P(x)} (\text{pr}_1 w, f(\text{pr}_1 w, \text{pr}_2 w)) = (x, v) \\
&\simeq \sum_{a:A} \sum_{u:P(a)} (a, f(a, u)) = (x, v) \quad (\text{by Example 2.3.0.5}) \\
&\simeq \sum_{a:A} \sum_{u:P(a)} \sum_{p:a=x} p_*(f(a, u)) = v \quad (\text{by [9] Theorem 2.7.2}) \\
&\simeq \sum_{a:A} \sum_{p:a=x} \sum_{u:P(a)} p_*(f(a, u)) = v \\
&\simeq \sum_{g: \sum_{a:A} a=x} \left(\sum_{u:P(\text{pr}_1(g))} (\text{pr}_2(g))_*(f(\text{pr}_1(g), u)) = v \right) \quad (\text{by Example 2.3.0.5}) \\
&\simeq \sum_{u:P(x)} (\text{refl}_x)_*(f(x, u) = v) \quad (\text{by Lemma 2.6.0.4 and Lemma 2.6.0.5 (ii) as center} \equiv (x, \text{refl}_x)) \\
&\simeq \sum_{u:P(x)} f(x, u) = v
\end{aligned}$$

□

fiberwise transformation $f : \prod_{x:A} P(x) \rightarrow Q(x)$ が fiberwise equivalence であるということを, 任意の $x : A$ について $f(x) : P(x) \rightarrow Q(x)$ が equivalence であることと定めると, 以下が成り立つ.

Theorem 2.6.0.9 ([9] Theorem 4.7.7) f を A 上の型族 P と Q の間の fiberwise transformation とする. このとき, f が fiberwise equivalence であることと $\text{total}(f)$ が equivalence であることは同値である

Proof.

$$\begin{aligned}
f \text{ が fiberwise equivalence} &\equiv \prod_{x:A} \text{isequiv}(f(x)) \\
&\leftrightarrow \prod_{x:A} \text{isContr}(f(x)) \\
&\equiv \prod_{x:A} \prod_{v:Q(x)} \text{isContr}(\text{fib}_{f(x)}(v)) \\
&\simeq \prod_{x:A} \prod_{v:Q(x)} \text{isContr}(\text{fib}_{\text{total}(f)}((x, v))) \quad (\text{by Theorem 2.6.0.8}) \\
&\leftrightarrow \prod_{w: \sum_{x:A} Q(x)} \text{isContr}(\text{fib}_{\text{total}(f)}(w)) \\
&\equiv \text{isContr}(\text{total}(f)) \\
&\leftrightarrow \text{isequiv}(\text{total}(f))
\end{aligned}$$

□

2.7 レトラクト

この章では, 型のレトラクトと関数のレトラクトをそれぞれ定義し, 後の章で使う補題を紹介する. まず, 型のレトラクトと定義は以下のとおりである.

Definition 2.7.0.1 ([9] p125) 型 A, B について, 2 つの関数レトラクション $r : A \rightarrow B$ とセクション $s : B \rightarrow A$ が存在し, 更にホモトピー

$$\epsilon : \prod_{y:B} (r(s(y)) = y)$$

が存在するとき, B は A のレトラクトであるという.

このレトラクトに対して, 次の補題が成り立つ.

Lemma 2.7.0.2 ([9] Lemma 3.11.7) B が A のレトラクトであり, かつ A が可縮であるとき, B も可縮である.

次に, 関数のレトラクトを定義する.

Definition 2.7.0.3 ([9] Definition 4.7.2) 関数 $g : A \rightarrow B$ が関数 $f : X \rightarrow Y$ のレトラクトであるとは, 4 つのホモトピー

$$\begin{aligned} R &: r \circ s \sim \text{id}_A \\ R' &: r' \circ s' \sim \text{id}_B \\ L &: f \circ s \sim s' \circ g \\ K &: g \circ r \sim r' \circ f \end{aligned}$$

が存在するような $r : X \rightarrow A, s : A \rightarrow X, r' : Y \rightarrow B, s' : B \rightarrow Y$ が存在し, 更に任意の $a : A$ に対して, パス

$$H(a) : K(s(a)) \cdot r'(L(a)) = g(R(a)) \cdot R'(g(a))^{-1}$$

が存在することをいう.

この定義において, ホモトピー R, R' はそれぞれ A が X のレトラクト, B が Y のレトラクトであるという条件である. また, 型についてのレトラクションは, 上の定義において $B \equiv Y \equiv \mathbf{1}$ としたときの特別な場合である. 関数のレトラクトについて, まず次のことが成り立つ.

Lemma 2.7.0.4 ([9] Lemma 4.7.3) 関数 $g : A \rightarrow B$ が関数 $f : X \rightarrow Y$ のレトラクトであるならば, $\text{fib}_g(b)$ は $\text{fib}_f(s'(b))$ のレトラクトである. ただし, s' は Definition 2.7.0.3 のものとする.

Proof. 証明の概略を述べる. Definition 2.7.0.3 のとおりに記号を定める. 任意の $b : B$ に対して,

$$\begin{aligned} \varphi_b &: \text{fib}_g(b) \rightarrow \text{fib}_f(s'(b)), & \varphi(a, p) &:= (s(a), L(a) \cdot s'(p)) \\ \psi_b &: \text{fib}_f(s'(b)) \rightarrow \text{fib}_g(b), & \psi(x, q) &:= (r(x), K(x) \cdot r'(q) \cdot R'(b)) \end{aligned}$$

と定義したとき, (Σ の induction から) 任意の $(a, p) : \text{fib}_g(b)$ に対して $\psi_b \varphi(a, p) = (a, p)$ が成り立つ, つまり

$$\prod_{b:B} \prod_{a:A} \prod_{p:g(a)=b} \psi_b \varphi(a, p) = (a, p)$$

を示せば良い. 詳細は [9] p140 参照のこと. □

この補題から, equivalence とレトラクトに関する補題が従う.

Theorem 2.7.0.5 ([9] Theorem 4.7.4) g が equivalence な f のレトラクトであれば, g も equivalence である.

Proof. Lemma 2.7.0.4 より, g の任意のファイバーは f のファイバーのレトラクトであり, $\text{isequiv}(f) \leftrightarrow \text{isContr}(f)$ と Lemma 2.7.0.2 から $\text{isequiv}(g)$ が分かる. □

2.8 Univalence implies function extensionality

最後に, univalence axiom から関数の外延性が導けることを示す. 証明は, まず univalence axiom から弱い関数の外延性が導けることを示し, 次に弱い関数の外延性から通常関数の外延性が従うことを示すという 2 段階で行われる. \mathcal{U} をユニバースとし, どこで univalent を仮定しているかを明記することにする.

Definition 2.8.0.1 ([9] Lemma 4.9.1) 弱い関数の外延性の公理を, 任意の A 上の型族 $B : A \rightarrow \mathcal{U}$ に対して

$$\left(\prod_{x:A} \text{isContr}(B(x)) \right) \rightarrow \text{isContr} \left(\prod_{x:A} B(x) \right)$$

が成り立つことと定義する.

次の補題は関数の外延性を仮定すればすぐに証明できるが, 関数の外延性を仮定しなくても univalence axiom から証明できるのがポイントである.

Lemma 2.8.0.2 ([9] Lemma 4.9.2) \mathcal{U} が univalent であると仮定する. 任意の $A, B, X : \mathcal{U}$ と任意の $e : A \equiv B$ に対して,

$$(X \rightarrow A) \simeq (X \rightarrow B)$$

の同型射は, $(e(\text{の同型射}) \circ -)$ で与えられる.

Proof. ある $p : A = B$ について $e = \text{idtoeqv}(p)$ と仮定してよい. よって, path induction より $B \equiv A$, $p \equiv \text{refl}_A$ とすれば, $e = \text{idtoeqv}(\text{refl}_A) = \text{id}_A$ となる. このとき, $(e \circ -)$ は $(\text{id}_A \circ -)$ となり, これは $X \rightarrow A$ 上の id であるので, equivalence である. \square

Corollary 2.8.0.3 ([9] Corollary 4.9.3) $P : A \rightarrow \mathcal{U}$ を可縮な型の族, つまり $\prod_{x:A} \text{isContr}(P(x))$ とする. このとき, 射影 $\text{pr}_1 : (\sum_{x:A} P(x)) \rightarrow A$ は equivalence である. 更に, \mathcal{U} が univalent であれば,

$$(\text{pr}_1 \circ -) : \left(A \rightarrow \sum_{x:A} P(x) \right) \simeq (A \rightarrow A)$$

である.

Proof. 前半について, Lemma 2.6.0.6 から, $x : A$ について, $\text{fib}_{\text{pr}_1}(x) \equiv P(x)$ である. P が可縮な型の族なので, $\text{isContr}(\text{pr}_1)$ が成り立つ. 後半については, Lemma 2.8.0.2 よりわかる. \square

上記の $\alpha \equiv (\text{pr}_1 \circ -)$ のホモトピーファイバーは可縮であるため, 特に id_A 上でも可縮, つまり $\text{isContr}(\text{fib}_\alpha(\text{id}_A))$ である. よって, $\prod_{x:A} P(x)$ が $\text{fib}_\alpha(\text{id}_A)$ のレトラクトであることを示せば, 弱い関数の外延性が univalence axiom から従うことになる.

Theorem 2.8.0.4 ([9] Theorem 4.9.4) \mathcal{U} が univalent であるとし, $P : A \rightarrow \mathcal{U}$ を可縮な型の族とする. $\alpha : (A \rightarrow \sum_{x:A} P(x)) \equiv (A \rightarrow A)$ とすると, $\prod_{x:A} P(x)$ は $\text{fib}_\alpha(\text{id}_A)$ のレトラクトである. つまり, $\prod_{x:A} P(x)$ は可縮となるので, univalence axiom から弱い関数の外延性の公理が従う.

Proof. 関数 $\varphi : (\prod_{x:A} P(x)) \rightarrow \text{fib}_\alpha(\text{id}_A)$, $\psi : \text{fib}_\alpha(\text{id}_A) \rightarrow \prod_{x:A} P(x)$ をそれぞれ,

$$\begin{aligned} \varphi(f) &:= (\lambda x. (x, f(x)), \text{refl}_{\text{id}_A}) \\ \psi(g, p) &:= \lambda x. \text{happly}(p, x) * (\text{pr}_2(g(x))) \end{aligned}$$

と定める. このとき,

$$\begin{aligned} \psi(\varphi(f)) &\equiv \lambda x. \text{happly}(\text{refl}_{\text{id}_A}, y) * (\text{pr}_2(x, f(x))) \\ &\equiv \lambda x. \text{refl}_*(f(x)) \\ &\equiv \lambda x. f(x) \\ &= f \end{aligned}$$

より成り立つ. \square

Theorem 2.8.0.5 ([9] Theorem 4.9.5) 弱い関数の外延性から通常の関数の外延性が従う.

Proof. 示したいことは

$$\prod_{A:\mathcal{U}} \prod_{P:A \rightarrow P} \prod_{f,g:\prod_{x:A} P(x)} \text{isequiv}(\text{happly}(f,g))$$

である. このとき,

$$\prod_{g:\prod_{x:A} P(x)} \text{isequiv}(\text{happly}(f,g))$$

は, $\lambda g. \text{happly}(f,g)$ が fiberwise equivalence であるということなので, Theorem 2.6.0.9 から,

$$\text{total}(\lambda g. \text{happly}(f,g)) : \sum_{g:\prod_{x:A} P(x)} (f = g) \rightarrow \sum_{g:\prod_{x:A} P(x)} (f \sim g)$$

が equivalence であればよい. ここで, Lemma 2.6.0.4 より送り元の型は可縮なので, 送り先の型

$$\sum_{g:\prod_{x:A} P(x)} \prod_{x:A} (f(x) = g(x))$$

が可縮であれば十分である. [9] Theorem 2.15.7 の証明のうち, 関数の外延性を仮定しなければ,

$$\sum_{g:\prod_{x:A} P(x)} \prod_{x:A} (f(x) = g(x))$$

は

$$\prod_{x:A} \sum_{u:P(x)} f(x) = u$$

のレトラクトであることが示せる (逆向きの合成が id とホモトピックであることにのみ関数の外延性を使っている). さらに, $\sum_{u:P(x)} f(x) = u$ は Lemma 2.6.0.4 から可縮であるため, 弱い関数の外延性から $\prod_{x:A} \sum_{u:P(x)} f(x) = u$ も可縮になる. したがって,

$$\sum_{g:\prod_{x:A} P(x)} \prod_{x:A} (f(x) = g(x))$$

も可縮である. □

Remark 2.8.0.6 Theorem 2.8.0.5 の証明には univalent axiom は用いていない.

参考文献

- [1] Victor Kac, Pokman Cheung, *Quantum Calculus*, Springer, 2001.
- [2] <https://github.com/nakamurakaoru/q-analogue>
- [3] 萩原 学/アフェルト・レナルド, *Coq/SSReflect/Mathcomp*, 森北出版, 2018
- [4] <https://coq.inria.fr/distrib/current/stdlib/>
- [5] <https://github.com/math-comp/math-comp>
- [6] <https://github.com/math-comp/analysis>
- [7] 梅村 浩, 『楕円関数論 楕円曲線の解析学』, 東京大学出版会, 2000.
- [8] H.P.Barendregt, *Lambda Calculi with Types*
- [9] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*