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 24 日

目次

第 1 章　自主学習・研究報告　5

1.1　q-類似　5

1.1.1　q-微分について成り立つ諸性質　5

1.1.2　無限和への拡張　7

1.2　型付き λ 計算　7

1.2.1　λ 計算　8

1.2.2　$\lambda\!\rightarrow$ とその拡張　10

1.2.3　λ-cube と Curry-Howard 同型　14

1.3　Coq　15

1.3.1　Coq の使い方　15

1.3.2　mathcomp の型　20

1.4　形式化　21

1.4.1　q-微分の定義　21

1.4.2　$(x-a)^n$ の q-類似　23

1.4.3　関数から多項式へ　30

1.4.4　q-Taylor 展開　33

1.5　今後の展望　37

第 2 章　少人数クラスまとめ　38

2.1　はじめに　38

2.2　型から型を作る　38

2.3　型の同型　39

2.4　Univalence axiom　40

2.5　関数の外延性　40

2.6　可縮, ファイバー　41

2.7　レトラクト　43

2.8　関数の外延性を Univalence axiom から導く　44

参考文献　47

序文

主結果

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. 形式化とは, 証明のために用意された人工言語に数学的な主張とその証明を翻訳し, 証明が論理学の推論規則に沿って正しく書かれていることをコンピュータを用いて機械的に検証することである. また Coq はこの形式化を行うためのソフトウェアの一つであり, このようなソフトウェアを総称して定理証明支援系と呼ぶ.

具体的な形式化の対象は Victor Kac, Pokman Cheung の *Quantum Calculus* [4] 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 である. 本論文での q-類似に関する定義や定理, 証明は [4] 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/tree/thesis` [7] にある. q_analogue.v が [4] 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 (どちらもテキストファイルである). 本論文の主目的である q-Taylor 展開, Gauss's binomial formula はそれぞれ q_analogue.v の 1417 行目, 1512 行目にある.

q-類似

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. 初めは Euler の分割に登場し, その後 Gauss や Heine らによって超幾何関数の研究の中で進展していった. q-類似を系統的に発展させたのは Jackson であり, また q-積分の概念も彼によって導入された.

q-類似とは, 実数パラメータ q, 実数上の関数 f に対して

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

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,

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

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

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

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

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

Coq とは, 前述のとおり定理証明支援系の 1 つであり, 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. λ 計算とは, 大まかにいえば計算の実行をモデル化したもので, Church によって考案された. その後, もともとは Russell のパラドクスを回避するために Russell 自身によって導入された型理論と結びついて型付き λ 計算 (こちらも Church によって考案された) として発展し, Martin-Löf によって直観主義型理論が開発され, 構成主義的な数学の基礎付けがなされた. 更により表現力の高い λ 計算の研究が続けられ, Coquand と Huet の開発した CoC(Calculus of Constructions) と呼ばれる 2 階述語論理よりも表現力の高い型付き λ 計算を用いて Coq が開発された. この CoC を用いていることから, Coq には補題とその証明を同じ言語で記述できるという特徴がある. また, 現在の Coq には Paulin-Mohring によって帰納的型を導入した λ 計算である CIC (Calculus of Inductive Constructions) が用いられている.

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. 更に, 型と形式化に関する数学の理論として, Homotopy Type Theory がある. これについては修士 2 年次に少人数クラスで学んだため, q-類似の形式化とは章を分け, 少人数クラスのまとめとして概要を述べる. 今回の証明に関しては, Coq の標準ライブラリ [2] 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. 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) などがあり, それらも mathcomp に基づいている.

実際に Coq で定義, 補題, 証明を表すコードは例えば以下のようなものである ([7] q_analogue.v 16-25 行目, ただし説明のためコードに変更を加えている).

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

Lemma dq_prod f g x :
  dq (f ** g) x = (f (q * x)) * dq g x + (g x) * dq f x.
Proof.
  rewrite /dq.
  rewrite !mulrBr
  rewrite [g x * f (q * x)]mulrC
  rewrite [g x * f x]mulrC
  by rewrite subrKA.
Qed.
```

基本的には `Lemma` コマンドで証明したい補題の主張 (を Coq にあわせて翻訳したもの) を書き, `Proof` 以下にタクティックとよばれる命令を書くことで証明を進め, 証明が完了したところで `Qed` コマンドを書くというのが一連の流れである. `Qed` コマンドは Coq にその補題を登録する機能があるため, 他の補題の証明に利用することができる. また, `Definition` コマンドで自分で新しく関数を定義することができる.

上記の例について見てみると, まず `Definition` で `dq` という名前の関数を定義している. := の左の `f : R → R` と `x` は関数 `dq` の引数を表しており, := の右側は関数の定義である. これは, 通常の数学での

実数上の関数 f と実数 x に対して,

$$d_q f(x) \coloneqq f(qx) - f(x)$$

と定める.

という定義を形式化したものである.

次に `Lemma` で補題の主張を書いている. `dq_prod` は補題の名前であり, 他の証明に用いる際に使えるように名前を付けている. `:` の左側の `f, g, x` がこの補題の引数であり, `:` の右側が補題の主張である. この `Lemma` は,

関数 f, g と $x \in \mathbb{R}$ について,

$$d_q(f(x)g(x)) = f(qx)d_q g(x) + g(x)d_q f(x)$$

が成り立つ.

という補題の形式化である. 更に, `Proof` 以下について, `rewrite` とは等式変形を行う命令であり, `rewrite /dq` であれば `dq` を定義に基づいて計算していて, `rewrite !mulrBr` であれば積と差についての分配法則をできる限り繰り返している. `mulrC` は積の交換律のことで, `[g x * f (q * x)]mulrC` であれば `g x * f (q * x)` を `f (q * x) * g x` に, `[g x * f x]mulrC` であれば `g x * f x` を `[f x * g x]` に書き換えている. `by rewrite subrKA` は同じものの引き算は 0 になるという書き換えをし, 更にその書き換えで証明が完了することを表している. 最後に `Qed` と書くことで, `dq_prod` という補題を他の証明で使えるようになる.

構成

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* [1] 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 1.3.1 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. [4] 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 [4] の通りであるが, 1.4.3 節では一部 [4] 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との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 章で修士 2 年次に少人数クラスで学習した内容についてまとめている.

謝辞

形式化の方針や Coq の使い方等, 本論文の作成において終始熱心に指導してくださった, アドバイザーである Jacques Garrigue 教授に心からの感謝を申し上げます.

今回の形式化の題材に選んだ q-類似について指導してくださった学部 4 年次の指導教員である古庄英和教授に感謝の意を表します.

多元数理科学研究科研究員の才川隆文氏から数多くの的確な助言をいただきました. 感謝申し上げます.

最後に, 研究室の皆様には常に刺激的な議論とアドバイスを頂き, 精神的にも絶えず支えられていました. 本当にありがとうございます.

第 1 章

自主学習・研究報告

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 [4] で取り上げられている内容について述べる. 1.1.2 節は極限や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

q-微分は [4] において, まず q-差分を, $q, x \in \mathbb{R}$, 実数上の関数 f について

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

を定義し, この d_q を用いて

$$D_q f(x) = \frac{d_q f(x)}{d_q x}$$

と定めている.

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のであった. これらの性質は q-微分では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, D_q を直接計算すると以下のようになる.

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

よって, 通常の微分と変わらない形で成り立つ ([4] 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 について計算してみると

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

となるため, D_q は

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

となり, 積の q-微分法は

$$D_q(f(x)g(x)) = f(qx)D_qg(x) + g(x)D_qf(x) \quad (\text{ [4] p3 (1.11)})$$

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

$$D_q(f(x)g(x)) = f(x)D_qg(x) + g(qx)D_qf(x) \quad (\text{ [4] p3 (1.12)})$$

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

$$D_q\left(\frac{f(x)}{g(x)}\right) = \frac{g(x)D_qf(x) - f(x)D_qg(x)}{g(x)g(qx)} \quad \text{[4] p3 (1.13)}$$

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

$$D_q\left(\frac{f(x)}{g(x)}\right) = \frac{g(qx)D_qf(x) - f(qx)D_qg(x)}{g(x)g(qx)} \quad \text{[4] p3 (1.14)}$$

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, $u(x) = \alpha x^\beta$ のときについてのみ

$$D_qf(u(x)) = (D_q^\beta f)(u(x))\cdot D_qu(x) \quad (\text{ [4] p4 (1.15)})$$

が成り立つことが知られている (証明は [4] 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, 通常の二項係数において 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

$$\begin{bmatrix} n \\ j \end{bmatrix} = \begin{bmatrix} n-1 \\ j-1 \end{bmatrix} + q^j\begin{bmatrix} n-1 \\ j \end{bmatrix} \quad (\text{ [4] p17 (6.2)})$$

が成り立つことが証明されている ([4] p18 参照). この補題も形式化できており, [7] の q_analogue.v の `q_pascal` という補題である.

1.1.2 無限和への拡張

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, 以下の Jacobi の三重積 ([4] p35 Theorem 11.1) はその一例である.

$z, q \in \mathbb{R}, |q| < 1$ 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.

これは楕円関数論の文脈で登場する恒等式であるが ([8] p144 (3.47) 等を参照), q-類似で得られる式

$$(1 + x)_q^{\infty} = \sum_{j=0}^{\infty} q^{j(j-1)/2} \frac{x^j}{(1 - q)(1 - q^2) \cdots (1 - q^j)} \quad \text{([4] p30 (9.3) 式)}$$

$$\frac{1}{(1 - x)_q^{\infty}} = \sum_{j=0}^{\infty} \frac{x^j}{(1 - q)(1 - q^2) \cdots (1 - q^j)} \quad \text{([4] p30 (9.4) 式)}$$

を用いることで簡単に証明できる. これらの等式は Euler によって発見されたことから, [4] ではそれぞれ Euler's first and second identities の意で E1, E2 と呼んでいる.

E1 については, Gauss's binomial formula を無限に拡張することで得られる. まず, 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,

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

となるので,

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

が得られる. また, E2 については, 任意の形式的冪級数に対して, 無限和に拡張した $x = 0$ まわりでの q-Taylor 展開が成り立つこと, $|q| < 1$ のとき $\lim_{n\to\infty}[n] = \frac{1}{1-q}$ となることから証明できる.

更に, E1, E2 をもとにして指数関数 e^x の q-類似 e_q^x, E_q^x を考えることができ ([4] p30 (9.7), p31 (9.10)), また e_q^x, E_q^x を用いて q-三角関数を定義したり ([4] p33) と, E1, E2 は [4] の中でも重要な式である.

有限の範囲で扱うことができ, E1, E2 の基礎となる等式であることが今回の形式化の目標として Gauss's binomial formula と q-Taylor 展開を選んだ理由である. 序文で書いたように, 無限和を含む形式化は今後の課題である.

1.2 型付き λ 計算

この節では, Coq の基礎となっている理論である λ 計算について [1] を基に説明する. 初めは型無しの λ 計算について述べ, その後に型付き λ 計算の説明をしていく.

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] Definition 2.1.1) $V = \{v, v', v'', \ldots\}$ 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.

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

Remark 1.2.2 抽象構文とは, ::= の左側の集合を, 右側の | で区切られた各構成要素で定義する書き方であり, 例えば $V ::= v \mid V'$ であれば, 「V という集合は, v もしくは V の要素に $'$ をつけたものからなる」という意味で, $\Lambda ::= V \mid (\Lambda\Lambda) \mid (\lambda V\Lambda)$ であれば, 「Λ という集合は, V の要素か, Λ の要素を 2 つ並べたものか, λ という記号・V の要素・Λ の要素をこの順で並べたものからなる」という意味である.

Example 1.2.3 v, (vv'), $\lambda v(vv')$, $((\lambda v(vv''))v')$ などが λ 項の例である.

今後, 表記上のルールや省略を以下のように約束する.

1. $x, y, z, \ldots$ で任意の変数を, $M, N, L, \ldots$ 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.

次に, 束縛変数と自由変数, λ 項の間の $\equiv$, 代入を定義する.

Definition 1.2.4 ([1] 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.

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

Example 1.2.5 λ に束縛されている変数が束縛変数, それ以外が自由変数である. 例えば

$$\lambda x.\ x + y$$

であれば束縛変数は x, 自由変数は y である. また,

$$\lambda x\ y.\ x(y + z) - w$$

であれば束縛変数は x, y で, 自由変数は z, w である.

Remark 1.2.6 ([1] Notation 2.1.4) Definition 1.2.42. について, 例えば,

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

となる.

Remark 1.2.7 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.

本節の最後に, 適用である β-簡約を定義する.

Definition 1.2.8 ([1] Definition 2.3.2) Λ 上の二項関係 $\to_\beta$(1 ステップ β-簡約), $\twoheadrightarrow_\beta$(β-簡約), $=_\beta$ を, 以下のように帰納的に定義する.

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

Example 1.2.9 1. $(\lambda x.xyz)y \to_\beta yyz$ である.
2. $(\lambda x.xy)(\lambda z.z) \to_\beta (\lambda z.z)y \to_\beta y$ より, $(\lambda x.xy)(\lambda z.z) \twoheadrightarrow_\beta y$ である.

1.2.2 $\lambda\to$ とその拡張

この節では, 単純型付きラムダ計算である $\lambda\to$ と, その拡張である $\lambda 2$ 及び λP を導入する.

$\lambda\to$

大まかに言えば, 型とはある要素が属する集合のようなものである. $\lambda\to$ は最も単純な型付き λ 計算であり, 関数の型を表現できる. 基本的なルールは,

- σ から τ への関数に σ の値を適用すると, その結果の型は τ である
- 変数 x の型が σ で, λ 項 M の型が τ のとき, $\lambda x.\,M$ の型は $\sigma \to \tau$ である.

以下, [1] での定義を見ていく.

Definition 1.2.10 ([1] Definition 3.1.1) $\lambda\to$ の型の集合 Type($\lambda\to$) は, 以下のように帰納的に定義される. $\mathbb{T}$ = Type($\lambda\to$) と書くことにする.

$$\alpha, \alpha', \ldots \in \mathbb{T} \qquad \text{(型変数)}$$
$$\sigma, \tau \in \mathbb{T} \Rightarrow (\sigma \to \tau) \in \mathbb{T} \qquad \text{(関数型)}$$

抽象構文を用いれば,

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

と書ける. ここで, $\mathbb{V}$ は,

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

である.

Remark 1.2.11 ([1] Notation 3.1.2)

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

$$\sigma_1 \to \sigma_2 \to \cdots \to \sigma_n$$

と書いた場合,

$$(\sigma_1 \to (\sigma_2 \to \cdots \to (\sigma_{n-1} \to \sigma_n) \cdots))$$

を表すものとする (右結合).

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

Definition 1.2.12 ([1] Definition 3.1.3)($\lambda\to$-Curry)

1. $M \in \Lambda, \sigma \in \mathbb{T}$ のとき, $M\colon \sigma$ 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 1.2.13 ([1] Definition 3.1.4) 文 $M\colon \sigma$ が基底 Γ から導出される ($\Gamma \vdash_{\lambda\to\text{-Curry}} M\colon \sigma$ or $\Gamma \vdash_{\lambda\to} M\colon \sigma$ or $\Gamma \vdash M\colon \sigma$ と書く) のは, $\Gamma \vdash M\colon \sigma$ が以下のルールから導かれるときである.

導出規則 0

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

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

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

導出規則 1

$$
\begin{aligned}
&\text{(axiom)} && \Gamma \vdash x\colon \sigma && (x\colon \sigma) \in \Gamma\text{のとき} \\
&(\to\text{-elimination}) && \frac{\Gamma \vdash M\colon (\sigma \to \tau) \quad \Gamma \vdash N\colon \sigma}{\Gamma \vdash (MN)\colon \tau} \\
&(\to\text{-introduction}) && \frac{\Gamma,\ x\colon \sigma \vdash M\colon \tau}{\Gamma \vdash (\lambda x.M)\colon (\sigma \to \tau)}
\end{aligned}
$$

以下, 本論文においては, 導出規則 1 を用いる.

Example 1.2.14 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. いくつか例を見てみる.

1. 任意の $\sigma, \tau \in \mathbb{T}$ について, $\vdash (\lambda xy.x)\colon (\sigma \to \tau \to \sigma)$ 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 $\to$-introduction なので,

$$\frac{x\colon \sigma \vdash (\lambda y.x)\colon (\tau \to \sigma)}{\vdash (\lambda xy.x)\colon (\sigma \to \tau \to \sigma)}$$

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,

$$\frac{\dfrac{x\colon \sigma,\ y\colon \tau \vdash x\colon \sigma}{x\colon \sigma \vdash (\lambda y.x)\colon (\tau \to \sigma)}}{\vdash (\lambda xy.x)\colon (\sigma \to \tau \to \sigma)}$$

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, $\vdash (\lambda xy.x)\colon (\sigma \to \tau \to \sigma)$ 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, $\lambda xy.xy\colon (\sigma \to \tau) \to (\sigma \to \tau)$ であることは,

$$\dfrac{\dfrac{\dfrac{x\colon (\sigma \to \tau), y\colon \sigma \vdash x\colon (\sigma \to \tau) \quad x\colon (\sigma \to \tau), y\colon \sigma \vdash y\colon \sigma}{x\colon (\sigma \to \tau), y\colon \sigma \vdash xy\colon \tau}}{x\colon (\sigma \to \tau) \vdash \lambda y.xy\colon (\sigma \to \tau)}}{\vdash \lambda xy.xy\colon (\sigma \to \tau) \to (\sigma \to \tau)}$$

   という導出木により確認できる.
3. $\lambda\!\to$ においては, $\lambda x.xx$ には型が付かない. 実際,

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

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

$$\dfrac{\Gamma, x\colon \sigma_1 \vdash xx\colon \sigma_2}{\Gamma \vdash \lambda x.xx\colon \sigma_1 \to \sigma_2}$$

   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,

$$\dfrac{\dfrac{\Gamma, x\colon \sigma_1 \vdash x\colon \tau \to \sigma_2 \quad \Gamma, x\colon \sigma_1 \vdash x\colon \tau}{\Gamma, x\colon \sigma_1 \vdash xx\colon \sigma_2}}{\Gamma \vdash \lambda x.xx\colon \sigma_1 \to \sigma_2}$$

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

$\lambda 2$

$\lambda\!\to$ の拡張の 1 つとして, $\lambda 2$ を考える. $\lambda 2$ の特徴は型を $\forall$ で抽象化できることである. 論理学における 2 階命題論理に対応し, $\lambda 2$ の「2」はここから来ている. 同じく論理学との関係を見ると, $\lambda\!\to$ は 1 階命題論理に対応する.

Definition 1.2.15 ([1] Definition 4.1.5) $\lambda 2$ の型の集合 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.

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

Remark 1.2.16 ([1] 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 $\to$ よりも結合が強い, すなわち, $\forall \alpha \sigma \to \tau$ は $(\forall \alpha \sigma) \to \tau$ のことであり, $\forall \alpha.\sigma \to \tau$ は $\forall \alpha(\sigma \to \tau)$ のことである.

Definition 1.2.17 ([1] Definition 4.1.7) $\lambda 2$ の型導出規則を, 以下のように定める.

(start rule) $\dfrac{(x\colon \sigma \in \Gamma)}{\Gamma \vdash x\colon \sigma}$

(→-elimination) $\dfrac{\Gamma \vdash M\colon (\sigma \to \tau) \quad \Gamma \vdash N\colon \sigma}{\Gamma \vdash (MN)\colon \tau}$

(→-introduction) $\dfrac{\Gamma,\ x\colon \sigma \vdash M\colon \tau}{\Gamma \vdash (\lambda x.M)\colon (\sigma \to \tau)}$

(∀-elimination) $\dfrac{\Gamma, \vdash M\colon (\forall \alpha.\sigma)}{\Gamma \vdash M\colon (\sigma[\alpha := \tau])}$

(∀-introduction) $\dfrac{\Gamma \vdash M\colon \sigma}{\Gamma \vdash M\colon (\forall \alpha.\sigma)} \quad (\alpha \notin FV(\Gamma))$

Example 1.2.18 ([1] 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.

$$\vdash (\lambda x.x) \colon (\forall \alpha.\alpha \to \alpha)$$
$$\vdash (\lambda xy.y) \colon (\forall \alpha\beta.\alpha \to \beta \to \beta)$$
$$\vdash (\lambda x.xx) \colon (\forall \alpha\alpha) \to (\forall \alpha\alpha)$$

ここで, 3 つ目の例については,

$$\dfrac{\dfrac{\dfrac{\dfrac{x\colon \forall\alpha\alpha \vdash x\colon \forall\alpha\alpha}{x\colon \forall\alpha\alpha \vdash x\colon \beta \to \alpha} \quad \dfrac{x\colon \forall\alpha\alpha \vdash x\colon \forall\alpha\alpha}{x\colon \forall\alpha\alpha \vdash x\colon \beta}}{x\colon \forall\alpha\alpha \vdash xx\colon \alpha}}{x\colon \forall\alpha\alpha \vdash xx\colon \forall\alpha\alpha}}{\vdash (\lambda x.xx)\colon (\forall\alpha\alpha) \to (\forall\alpha\alpha)}$$

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, $\lambda\!\to$ 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.

λP

$\lambda 2$ とは別の拡張として, λP がある. P は predicate の P であり, 述語論理に対応する. λP の特徴は依存関数型を扱えることと, 命題の証明のシミュレートを行えることである. これからは項と型を区別せず, まとめて擬式と呼ぶことにして, 以下のように定義する.

Definition 1.2.19 ([1] Definition 5.1.8)

1. λP の擬式全体 $\mathcal{T}$ を以下のように定める.

$$\mathcal{T} = V \mid C \mid \mathcal{T}\mathcal{T} \mid \lambda V : \mathcal{T}.\, \mathcal{T} \mid \prod V : \mathcal{T}.\, \mathcal{T}$$

   ここで, V は変数全体, C は定数全体である.

2. C の中から 2 つ要素を選んで, $*, \Box$ と名付ける. $*$ は $\alpha : *$ と書くことで α が型であることを表す記号である. また, この $*$ に対し, $\mathbb{K}$ を

$$\mathbb{K} = * \mid \mathbb{K} \to \mathbb{K}$$

   と定めたとき, $k : \Box$ で k が $\mathbb{K}$ に属することを表す記号である.

型導出規則は以下の通り. ただし, s は $*$ か $\Box$ のいずれかである.

Definition 1.2.20 ([1] Definition 5.1.9) λP の型導出規則を, 以下のように定める.

(axiom) $\quad \vdash *\colon \Box$

(start-rule) $\quad \dfrac{\Gamma \vdash A : s}{\Gamma,\ x : A \vdash x : A} \quad (x \notin \Gamma)$

(weakening rule) $\quad \dfrac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma,\ x : C \vdash A : B} \quad (x \notin \Gamma)$

(type/kind formation) $\quad \dfrac{\Gamma \vdash A : * \quad \Gamma,\ x : A \vdash B : s}{\Gamma \vdash (\Pi x : A.\ B) : s}$

(application rule) $\quad \dfrac{\Gamma \vdash F\colon (\Pi x : A.\ B) \quad \Gamma \vdash a\colon A}{\Gamma \vdash Fa\colon B[x := a]}$

(abstraction rule) $\quad \dfrac{\Gamma,\ x\colon A \vdash b\colon B \quad \Gamma \vdash (\Pi x : A.\ B) : s}{\Gamma \vdash (\lambda x : A.\ b)\colon (\Pi x : A.\ B)}$

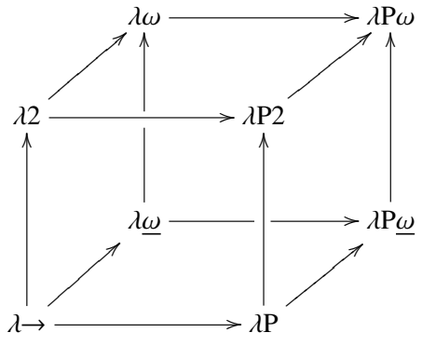
(conversion rule) $\quad \dfrac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_\beta B'}{\Gamma \vdash A : B'}$

Example 1.2.21 ([1] p86) λP での型付けには以下のようなものがある.

$$A : * \vdash (\lambda x.x) : (\Pi x : A.\ *) : \Box$$
$$A : *,\ P : A \to *,\ a : A \vdash Pa : *$$
$$A : *,\ P : A \to * \vdash (\lambda a : A\ \lambda x : Pa.\ x) : (\Pi a : A.\ (Pa \to Pa))$$

1.2.3 λ-cube と Curry-Howard 同型

前節で $\lambda\to$ の拡張として $\lambda 2$ と λP を導入したが, これらを組み合わせたものとして λP2 という λ 計算が存在し, 2 階述語論理に相当する表現力を持つ. また, 本論文では触れないが, $\lambda\to$ のまた別の拡張として $\lambda\underline{\omega}$ というものも存在する ([1] Definition 5.1.4, Definition 5.1.5). $\lambda\to$ を出発点とし, $\lambda 2$, λP, $\lambda\underline{\omega}$ の性質を持つかどうかで 8 つの λ 計算 $\lambda\to$, $\lambda 2$, $\lambda\underline{\omega}$, $\lambda\omega$, λP, λP2, λP$\underline{\omega}$, λPω を考えることができ, この 8 つは以下のように cube を形づくる.



ここで, 各辺となっている $\longrightarrow$ は $\lambda 2$, $\lambda\underline{\omega}$, λP のいずれかの性質の導入を表し, 下から上への矢印は $\lambda 2$ の導入を, 左下から右上への矢印は $\lambda\underline{\omega}$ の導入を, 左から右への矢印は λP の導入を表す.

型付き λ 計算と証明の形式化をつなぐのが, Curry-Howard 同型と呼ばれる,

$$\text{命題} \leftrightarrow \text{型}$$
$$\text{証明} \leftrightarrow \text{型の要素}$$

という対応関係である. λP の説明の際にも触れたが, この対応関係を基に命題の証明をシミュレートすることが形式化であり, Coq に型付き λ 計算が関係する理由である.

Example 1.2.22 命題 P, Q について, $P \Longrightarrow Q$ かつ P であれば, Q が成り立つというモーダスポーネンスの証明をこの同型をもとに考えてみることにする. 論理演算子と型の間にも対応があり, $\Longrightarrow$ に対応するのは $\lambda\to$ で見た関数型 $\to$ であるため, モーダスポーネンスを証明するには,

「P という型の要素と $P \to Q$ という型の要素から Q という型の要素を構成すればよい」

と考えることができる. ここで P の要素に a, 関数型 $P \to Q$ の要素 (つまり関数) に f と名前をつけたとすると, f に a を適用した fa にはどんな型がつくであろうか. $\lambda\to$ の導出規則を思い出してみると, ($\to$ 除去) が使えて fa には Q という型がつく. よって型として Q を持つような要素として具体的に fa という項が構成できたため, Q が成り立つことを証明できたことになる.

論理演算子と型の対応をまとめると以下のようになる.

$$\begin{aligned} P \text{ ならば } Q &\quad \text{関数型 } P \to Q \\ P \text{ かつ } Q &\quad \text{直積型 } P \times Q \\ P \text{ または } Q &\quad \text{直和型 } P + Q \\ \text{ある } x \in A \text{ について } P(x) &\quad \text{依存和型 } \sum_{x:A} P(x) \\ \text{任意の } x \in A \text{ について } P(x) &\quad \text{依存関数型 } \prod_{x:A} P(x) \end{aligned}$$

1.3 Coq

前節で述べた λ-cube のうち, 最も強い $\lambda P\omega$ に帰納的型を導入した λ 計算が Coq で採用されている. 本節では定理証明支援系 Coq の使い方を具体例を用いて説明し, また, この後の形式化でよく使う型についてその構造の説明も行う.

1.3.1 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. ただし, タクティックは証明の中でコンテクスト (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する. 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] ⋯ [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 → R) x := f (q * x) - f x.
```

であれば, `dq` が定義の名前, `f`, `x` が引数, `R -> 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` 再帰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. タクティックは Proof...Qed の間に使われる. 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クストに移動する. また move: H で補題 H もしくはコンテクストに存在する H をゴールの前提に移す.

apply　補題 *lem* が P1 → 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. コンテクストの仮定 H が P1 → *P*2 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 → (A = B) という形であるとき, 新たなゴールとして H が追加される). 更に, rewrite *lem* in H で, コンテクストの 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 [条件] *lem*　条件に一致する場所を *lem* で書き換える.

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.

このような, タクティックの前後に書いてその機能を拡張するものをタクティカルと言う. move=>の=> や move: の : もタクティカルである.

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 1.3.1 モーダスポーネンス

Curry-Howard 同型の際にも見たモーダスポーネンスを 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, ∧ は「かつ」を表している.

このとき, Coq のゴールエリア (コンテク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
```

---の上がコンテクスト, 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 $P_1 \wedge P_2 \to P_3$ と $P_1 \to P_2 \to P_3$ 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 → Q) があるため, move=> pq で pq という名前をつけてコンテクストに移動する.

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

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

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

ゴールが Q であり, コンテクストに P → 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, コンテクストに 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 行づつ書いたが, 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 (コンテクストに存在する, `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.
```

Example 1.3.2 代入計算

自然数 m, n について,

$$m = 0 \Longrightarrow n + m = n$$

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クストに移動する.

```
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げられることを用いれば以下のように書くこともでできる.

```
Lemma substitution m n : m = 0 → n + m = n.
Proof. by move→; rewrite addn0. Qed.
```

Remark 1.3.3 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.

1.3.2 mathcomp の型

本節では, 今回の形式化においてよく用いる型, rcfType, nat, int について説明する.

rcfType

本論文では 実数の形式化として mathcomp の ssrnum にある rcfType を用いることにする. rcf とは real closed field, つまり実閉体の意味であり, rcfType とは実閉体全体を表す型のことである. よって, この型そのものではなくこの型の要素を実数として形式化に用いる. 実際の形式化では

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

としており, R が rcfType の要素であること, q が R の要素であることを宣言している. このとき 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変数を 1 つ固定したものとして扱っている.

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 のヒエラルキーは

```
eqType → choiceType
       → zmodType → ringType → comRingType → comUnitRingType → idomainType → fieldType
       → numFieldType → realFieldType → rcfType
```

となっており, 多くの性質を引き継いでいる. 特に Coq での環全体を表す型 ringType の性質を持っていることが重要で, 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.

nat

自然数を表す型は nat であり, 以下のように帰納的に定義されている.

```
Inductive nat : Set :=  0 : nat | S : nat → nat.
```

このコードの意味を大まかに説明すると, nat という型は 0 という要素と, nat の要素から新しく nat の要素を作る操作 S で定義されており, またこれ以外には nat の要素がないということである. よって, 0, S0, S(S0) などが nat の要素である. 0 は 0 に対応し (したがって Coq での自然数は 0 を含んでいる), S は次の数を作る操作, つまり +1 に対応するため, 0, S 0, S (S 0) はそれぞれ 0, 1, 2 のことである. また, 今後形式化で登場する .+1 という記号は S の別表記であり, n: nat に対して n.+1 は S n と同じ意味である.

帰納的に定義された型の特徴は場合分けが行えることであり, nat であれば n = 0 のときと n = S n0 の場合, つまり 0 かある数の +1 であるかの場合分けができる. また, nat に対しては数学的帰納法を用いることもできる.

Remark 1.3.4 証明中で場合分けをしたいときに対応する Coq のタクティックは case であり, 定義などに使いたい場合は match with 等の構文が用意されている. 帰納法に対応するタクティックは elim である.

int

整数の形式化には int という型を利用している. こちらは以下のように定義されている.

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

nat を用いており, Posz は自然数から 0 以上の整数に送る関数, Negz は自然数から負の整数に送る関数であり, Negz n = - (Posz n.+1) という関係がある.

int に対しても場合分けができ, n : int とすれば, n = Posz n0 のときと n = Negz n0 のとき, すなわち 0 以上か負かでの場合分けが行える.

1.4 形式化

本節から実際に q-類似の形式化を行っていく.

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

$$d_q f(x) \coloneqq f(qx) - f(x)$$

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

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

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.
```

2 行目の Import GRing は ringType に対する補題を使いやすくするためのコマンドで, 7 行目の Notation は記法を定めるためのコマンドである. 今回であれば関数同士の割り算の記法を定義している. また, Dq の定義の中の id は恒等関数のことである.

Remark 1.4.2 f が微分可能であるとき,

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

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 1.4.3 ([4] 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 1.4.4 ([4] 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通りである.

```
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}(mul1r x) -mulrBl expfzMl -add_div; last by apply mulf_neq0.
  rewrite [in x ^ n](_ : n = (n -1) +1) //; last by rewrite subrK.
  rewrite expfzDr ?expr1z ?mulrA -?mulNr ?red_frac_r ?add_div //.
  rewrite -{2}[x ^ (n - 1)]mul1r -mulrBl mulrC mulrA.
  by rewrite [in (q - 1)^-1 * (q ^ n - 1)] mulrC.
Qed.
```

この形式化についていくつか説明を加える. まず用いているタクティックを見てみると, `x ≠ 0` という前提があるため, `move=> Hx` で名前をつけてコンテクストに読み込んでいる. その後は基本的に `rewrite` しか用いていないため, 式変形を行っているだけである. また, 証明中 3, 4 行目の `last` は証明するゴールの順番を入れ替えるタクティックであり, 5 行目最後の`//`は `rewrite` の後や `move`, `apply` の後で`=> //`という形で用いるタクティカルで, すべてのゴールに対して `done` を試みる機能を持つ (`done` が適用できないゴールがあってもエラーにならない). 同じような形で用いるタクティカルに`/=`があり, こちらは定義を開く程度の簡単な計算を試みるものである. この二つを合わせて`//=`とすることもできる.

証明で用いている補題について, 例えば `mul1r` 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 `mul1r` の `r` は `ringType` に対する補題であることを表しており, `mulrBl`, `subrK`, `mulrC` なども同じである.

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 ≠ 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 1.4.5 `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の形式化になっている.

文献 [4] 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 1.4.6 ([4] 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 1.4.7 $n \in \mathbb{Z}_{>0}$ 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 ≠ 0 →
  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 !mul1r mulr1 expr1z.
```

```
    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}(mul1r (qbinom_pos a n.+1 x)).
    by rewrite -mulrDl -qnat_cat1.
Qed.
```

elim: n で n に対する数学的帰納法をしており, 後はすべて rewrite による式変形である. elim の直後の-はゴールを変形するタクティックではなく, ゴールが増えた際にコードを読みやすくするために入れている記号である.

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, 以下のようになる.

Proposition 1.4.8 ([4] 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.

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

より成立する. □

この形式化は次の通りである.

```
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 !mul1r.
    + rewrite addnS [LHS]/= IH /= !mulrA.
      by rewrite -[q ^ n.+1 * q ^ m] expfz_n0addr // addnC.
Qed.
```

[4] 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, [4] の定義は

Definition 1.4.9 ([4] 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. `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 1.4.10 ([4] p10 Proposition 3.2) $m, n \in \mathbb{Z}$ について, Proposition 1.4.8 は成り立つ, つまり

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

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,

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

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

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

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 ≠ 0 →
  qbinom_denom a m x ≠ 0 →
  qbinom_denom (q ^ m * a) n x ≠ 0 →
  qbinom a (m + n) x = qbinom a m x * qbinom (q ^ m * a) n x.
Proof.
  move⇒ Hq0.
  case: m ⇒ m Hm.
  - case: n ⇒ n Hn.
    + by apply qbinom_pos_explaw.
    + rewrite qbinom_exp_pos_neg //.
      by rewrite addrC expfzDr // -mulrA.
  - case: n ⇒ n 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 mul1r.
      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,

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

  ```
  case Hmn : (Posz m + Negz n) ⇒ [l|l]  /=.
  ```

  として, `m - n'` の値を `l` とおき, `l` が `0` 以上かどうかで場合分けをしている.
- `have` は仮定を追加するためのタクティックで,

  `have H: P`

  と書くことで `P` がゴールに追加され, その証明が終わればコンテクストに `P` が追加される. また,

  `have →:  A = B`

  と書くと `rewrite (_ :  A = B)` とほとんど同じ働きをする.
- 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 1.4.11 ([4] 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 1.4.7 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 1.4.9 と, 商の微分公式の q-類似版である

$$D_q\left(\frac{f(x)}{g(x)}\right) = \frac{g(x)D_qf(x) - f(x)D_qg(x)}{g(x)g(qx)} \quad \text{([4] p3 (1.13))}$$

及び Proposition 1.4.7 を用いて示される. □

[4] 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 mul1r 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 opprB 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}(expr1z 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, [4] の証明をそのまま形式化したものになっている. また, 約分のためいくつかの項が 0 でないという条件がついている.

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

本節では, 今まで関数として形式化していた q-微分や $(x-a)_q^n$ を多項式として定義しなおしていく. 多項式として捉えなおす理由は, q-Taylor 展開が多項式に関する定理であることに加え, 次の 2 つのメリットがあるからである.

$x/x = 1$ 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 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, $x/x = 1$ 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.

$q = 0$ 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

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

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 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すれば $q = 0$ でも問題が起きない. よって多項式に対して定義しなおした q-微分は $q = 0$ かどうかにかかわらず高階 q-微分を定義できる.

通常の数学では関数と多項式の違いをあまり意識しないことも多く, $(x-a)_q^n$ は, [4] の定義から多項式になるこ

とは当たり前に思えるかもしれないが, Coq では関数と多項式は明確に区別されている. 前節での $(x-a)_q^n$ の形式化 qbinom_pos は, R の要素を受け取って R の要素を返す関数として定義されており, Coq では一度関数として定義したものを多項式として扱うことはできない. そのため $(x-a)_q^n$ の形式化を多項式として改めて定義し, 前節での定義と一致することを確認するという方法をとることにする.

また, これまでの q-微分の形式化 Dq は関数の各点での値として定義していたため, 多項式として定義しなおした qbinom_pos_poly に適用することはできない. そのため q-微分の形式化も多項式に対する定義に変えなくてはならない.

Remark 1.4.12 実際には, 多項式 p に対して $x \mapsto$ (p の x での値) という関数を考えればこれまでの Dq に適用できる. しかし, 各点での定義のままでは前述のとおり約分を行うために 0 でないという条件が必要となり不都合が生じるので, 多項式を受け取り多項式を返す操作として定義し直すことにする.

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 係数多項式全体を表す型となる. 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. また, もともと通常の数学において環を係数にもつ多項式全体の集合は環を成し, また加群の構造を持っているが, Coq において多項式全体を表す型 {poly T} も 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: T, p: {poly T} である) に関する補題が利用できる. {poly} に関する記号や操作については以下の通り.

\poly_(i < n) E(i)　次数が $n-1$ 次以下, i 次の係数が $E(i)$ である多項式
c%:P　定数 c のみからなる単項式
'X　変数 x のみからなる単項式
p`_i　多項式 p の i 次の係数
size p　多項式 p の次数 +1 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]　多項式 p の x での値

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, すなわち x を qx に変えている. また, dqp は多項式に対しての dq 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 ↦ p.[x] という関数に dq 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.

Remark 1.4.13 mathcomp の fraction.v では整域から構成する商体を形式化している. 通常の数学と同様 {poly R} は整域になるため, fraction.v の内容を使って Dqp の定義が妥当であることを証明できる.

```
Import FracField.

Local Notation tofrac := (@tofrac [idomainType of {poly R}]).
Local Notation "x %:F" := (tofrac x).

Theorem Dqp_ok_frac p : (dqp p)%:F / (dqp 'X)%:F = (Dqp p)%:F.
```

ここで x %:F は整域の要素 x を商体の要素 x/1 に送る操作である.

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 1.4.14 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. 前述の通り, 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 1.4.7 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 1.4.15 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.

1.4.4 q-Taylor 展開

この節では, 本論文の主目的である有限次 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 1.4.16 ([4] 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, $\{P_n(x)\}_{n=0} \subset V$ $(n = 0, 1, 2, \cdots)$ は次の三条件をみたすとする.

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

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のようになる.

```
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.
Proof.
  move⇒ Hl Hd HP0 HP HdP Hdf.
  have Hdf' : (size f ≤ n.+1)%N.
    by rewrite Hdf leqnn.
  move: (poly_basis n P f HdP Hdf') ⇒ [c] Hf.
  have Hc0 : c 0%N = ((D \^ 0) f).[a] ⇒ /=.
```

```
    rewrite Hf.
    destruct n.
      by rewrite big_nat1 hornerZ HP0 mulr1.
    rewrite hornersumD.
    rewrite (@big_cat_nat _ _ _ 1) //= big_nat1.
    rewrite hornerZ HP0 mulr1.
    have → : (1 = 0 + 1)%N by [].
    rewrite big_addn subn1 /=.
    under eq_big_nat ⇒ i /andP [_ _].
      rewrite hornerZ addn1 HP mulr0.
    over.
    by rewrite big1 // addr0.
  have ithD : ∀ j, (j.+1 ≤ n)%N →
    (D \^ j.+1) f = \sum_(j.+1 ≤ i < n.+1) c i *: P (i - j.+1)%N.
    move⇒ j Hj.
    rewrite Hf linear_distr; last by apply nth_islinear.
    rewrite {1}(lock j.+1).
    rewrite (@big_cat_nat _ _ _ j.+1) //=; last by apply leqW.
    rewrite -lock.
    under eq_big_nat ⇒ i /andP [_ Hi].
      rewrite nthisfderiv_0 // scaler0.
    over.
    rewrite big1 // add0r.
    by under eq_big_nat ⇒ i /andP [Hi _] do rewrite nthisfderiv_pos //.
  have coef : ∀ j, (j ≤ n)%N → c j = ((D \^ j) f).[a].
    move⇒ j Hj.
    destruct j ⇒ //.
    rewrite ithD //.
    rewrite (@big_cat_nat _ _ _ j.+2) //= big_nat1 hornerD.
    rewrite subnn hornerZ HP0 mulr1 hornersumD.
    under eq_big_nat ⇒ i /andP [Hi Hi'].
      rewrite hornerZ.
      move: (Hi).
      rewrite -addn1 -leq_subRL //; last by apply ltnW.
      case: (i - j.+1)%N ⇒ // k Hk.
      rewrite HP mulr0.
    over.
    by rewrite big1 // addr0.
  rewrite {1}Hf big_nat_cond [RHS]big_nat_cond.
  apply eq_bigr ⇒ i /andP [/andP [Hi Hi'] _].
  by rewrite coef.
Qed.
```

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 : {poly R} → {poly R}) :=
    ∀ a b f g, D ((a *: f) + (b *: g)) = a *: D f + b *: D g.

  Definition isfderiv D (P : nat → {poly R}) := ∀ 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.

- `under / over` は総和 $\sum$, 総積 $\prod$ など同じ演算を繰り返す記号 (これを big operator と呼ぶ) に対して用いるタクティックで, big operator で束縛されている各項を書き換えたい際に使う. `under` で各項に注目し, 書き換えが終わったら `over` でもとの big operator に戻る. big operator について詳しくは mathcomp の bigop.v を参照のこと.
- [4] での証明には, $\{P_0(x), P_1(x), \ldots, P_n(x)\}$ 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 : nat → {poly R}) (f : {poly R}) :
  (∀ m, size (P m) = m.+1) →
  (size f ≤ n.+1)%N →
  ∃ (c : nat → R), f = \sum_(0 ≤ 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が, それでも問題なく証明は完成する.

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

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

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 1.4.17 ([4] 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 1.4.16 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 Proposition1.4.7 から分かる. □

前節で準備した `Dqp`, `qbinom_pos_poly` を用いて Theorem 1.4.17 を形式化する.

```
Fixpoint qfact n :=
  match n with
  | 0 ⇒ 1
  | n.+1 ⇒ qfact n * qnat n.+1
  end.

Theorem q_Taylorp n (f : {poly R}) c :
  (∀ n, qfact n ≠ 0) →
  size f = n.+1 →
  f = \sum_(0 ≤ i < n.+1) ((Dqp' \^ 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以下の通り.

```
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 1.4.18 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 Dqp' と Dq 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 1.4.19 ([4] p12 Example (4.4)) $n \in \mathbb{Z}_{>0}$ について,

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

が成り立つ.

Proof. Theorem 1.4.17 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 1.4.20 ([4] p15 Example (5.5)) $n \in \mathbb{Z}_{>0}$ について,

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

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,

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

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 1.4.17 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 : (∀ n, qfact n ≠ 0) →
  'X^n = \sum_(0 ≤ i < n.+1) (qbicoef n i *: qbinom_pos_poly 1 i).

Definition qbicoef n j := qfact n / (qfact j * qfact (n - j)).
Theorem Gauss_binomial a n : (∀ n, qfact n ≠ 0) →
  qbinom_pos_poly (-a) n =
  \sum_(0 ≤ i < n.+1) (qbicoef n i * q ^+ (i * (i - 1))./2 * a ^+ i) *: 'X^(n - i).
```

Remark 1.4.21 `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.

1.5 今後の展望

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 $q \to 1$ 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.

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,

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

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. 大雑把にいえば, 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 $A \to B$ が構成できる. このとき,

$$f : A \to B,\ a : A \Longrightarrow f(a) : B$$
$$a : A,\ b(x) : B \Longrightarrow \lambda a.b : A \to B$$

である. より一般に, 型 A と A 上の型族 $B \to \mathcal{U}$ 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,

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

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

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

と定義する. 除去規則は, 依存和型のときと同様に考えると, 「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 $C(x, x, \mathrm{refl}x)$ を示せばよい」となる. 以下, この identity type の除去規則を用いることを path induction と呼ぶ.

2.3 型の同型

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

Definition 2.3.1 ([9] Definition 2.4.1) $A : \mathcal{U}, P : A \to \mathcal{U}$ とする. $f, g : \prod_{x:A} P(x)$ に対して,

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

と定める.

次に, 「逆写像」を定義する.

Definition 2.3.2 ([9] Definition 2.4.6) $A, B : \mathcal{U}, f : A \to B$ とする. このとき, f の quasi-inverse $\mathrm{qinv}(f)$ を,

$$\mathrm{qinv}(f) :\equiv \sum_{g:B\to A} ((f \circ g \sim \mathrm{id}_B) \times (g \circ f \sim \mathrm{id}_A))$$

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,

- $\mathrm{qinv}(f) \to \mathrm{isequiv}(f)$
- $\mathrm{isequiv}(f) \to \mathrm{qinv}(f)$
- e_1, e_2 :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,

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

と定めることにする. 関数 f について $\mathrm{isequiv}(f)$ が成り立つとき, f は equivalence であるという.

isequiv を使って型同士の同型を定義する.

Definition 2.3.3 ([9] p73 (2.4.11)) $A, B : \mathcal{U}$ について,

$$A \simeq B :\equiv \sum_{f:A\to B} \mathrm{isequiv}(f)$$

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 2.3.4 ([9] Lemma 2.4.12) $A, B, C : \mathcal{U}$ について,

- $A \simeq A$
- $A \simeq B \to B \simeq A$
- $A \simeq B \to B \simeq C \to A \simeq C$

が成り立つ.

Example 2.3.5 ([9] Exercise 2.10) $\sum$ 型は「結合的」である. つまり任意の型 A, 型族 $B : A \to \mathcal{U}$, 型族上の型族 $C : (\sum_{x:A} B(x)) \to \mathcal{U}$ 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,

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

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 2.4.1 ([9] Axiom 2.10.3)

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

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

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

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

という関数を定義できる. このとき,

Axiom 2.5.1 ([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

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

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：happly(funext(h), x) = $h(x)$　($h : \prod_{x:A}(f(x) = g(x))$)
- (propositional) 一意性：p = funext($x \mapsto$ 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 2.6.1 ([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 isContr(A) を,

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

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 2.6.2 ([9] Definition 4.2.4) 関数 $f : A \to B$ の点 $y : B$ の上でのファイバーを

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

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 2.6.3 ([9] Definition4.4.1) 関数 $f : A \to B$ が可縮であるとは, 任意の $y : B$ に対して $\mathrm{fib}_f y$ が可縮であることである. このことを表す型 isContr(f) を,

$$\mathrm{isContr}(f) :\equiv \prod_{y:B} \mathrm{isContr}(\mathrm{fib}_f(y))$$

と定める.

このように定義した isContr(f) について,

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

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 2.6.4 ([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 2.6.5 ([9] Lemma 3.11.9) $P : A \to \mathcal{U}$ 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 2.6.6 ([9] Lemma 4.8.1) 任意の型族 $B : A \to \mathcal{U}$ について, $\mathrm{pr}_1 : \sum_{x:A} B(x) \to A$ の $a : A$ 上でのファイバーは $B(a)$ と同型である, つまり,

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

が成り立つ.

Proof.

$$\begin{aligned}
\mathrm{fib}_{\mathrm{pr}_1}(a) &\equiv \sum_{u:\sum_{x:A} B(x)} \mathrm{pr}_1(u) a \\
&\simeq \sum_{x:A} \sum_{b:B(x)} (x = a) \quad \text{(by Example 2.3.5)} \\
&\simeq \sum_{x:A} \sum_{b:B(x)} \sum_{p:x=a} \mathbf{1} \\
&\simeq \sum_{x:A} \sum_{p:x=a} \sum_{b:B(x)} \mathbf{1} \\
&\simeq \sum_{x:A} \sum_{p:x=a} B(x) \\
&\simeq B(a) \quad \text{(by Example 2.3.5, Lemma 2.6.4, Lemma 2.6.5)}
\end{aligned}$$

□

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 2.6.7 ([9] Definition 4.7.5) 型族 $P, Q : A \to \mathcal{U}$ と依存関数 $f : \prod_{x:A} P(x) \to Q(x)$ (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,

$$\mathrm{total}(f) :\equiv \lambda w.\, (\mathrm{pr}_1 w, f(\mathrm{pr}_1 w, \mathrm{pr}_2 w)) : \sum_{x:A} P(x) \to \sum_{x:A} Q(x)$$

と定める.

Theorem 2.6.8 ([9] Theorem 4.7.6) f が A 上の型族 P と Q の間の fiberwise transformation (つまり $f : \prod_{x:A} P(x) \to Q(x)$) であり, $x : A$ と $v : Q(x)$ が与えられたとする. このとき,

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

である.

Proof.

$$
\begin{aligned}
\mathrm{fib}_{\mathrm{total}(f)}((x,v)) &\equiv \sum_{w:\sum_{x:A}P(x)} (\mathrm{pr}_1 w, f(\mathrm{pr}_1 w, \mathrm{pr}_2 w)) = (x,v) \\
&\simeq \sum_{a:A}\sum_{u:P(a)} (a, f(a,u)) = (x,v) \quad \text{(by Example 2.3.5)} \\
&\simeq \sum_{a:A}\sum_{u:P(a)}\sum_{p:a=x} p_*(f(a,u)) = v \quad \text{(by [9] Theorem 2.7.2)} \\
&\simeq \sum_{a:A}\sum_{p:a=x}\sum_{u:P(a)} p_*(f(a,u)) = v \\
&\simeq \sum_{g:\sum_{a:A}a=x}\left(\sum_{u:P(\mathrm{pr}_1(q))} (\mathrm{pr}_2(q))_*(f(pr_1(q),u) = v\right) \quad \text{(by Example 2.3.5)} \\
&\simeq \sum_{u:P(x)} (\mathrm{refl}_x)_*(f(x,u) = v) \quad \text{(by Lemma 2.6.4 and Lemma 2.6.5 (ii) as center} \equiv (x, \mathrm{refl}_x)) \\
&\simeq \sum_{u:P(x)} f(x,u) = v
\end{aligned}
$$

□

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

Theorem 2.6.9 ([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 $\mathrm{total}(f)$ が equivalence であることは同値である

Proof.

$$
\begin{aligned}
f \text{ が fiberwise equivalence} &\equiv \prod_{x:A} \mathrm{isequiv}(f(x)) \\
&\leftrightarrow \prod_{x:A} \mathrm{isContr}(f(x)) \\
&\equiv \prod_{x:A}\prod_{v:Q(x)} \mathrm{isContr}(\mathrm{fib}_{f(x)}(v)) \\
&\simeq \prod_{x:A}\prod_{v:Q(x)} \mathrm{isContr}(\mathrm{fib}_{\mathrm{total}(f)}((x,v))) \quad \text{(by Theorem 2.6.8)} \\
&\leftrightarrow \prod_{w:\sum_{x:A}Q(x)} \mathrm{isContr}(\mathrm{fib}_{\mathrm{total}(f)}(w)) \\
&\equiv \mathrm{isContr}(\mathrm{total}(f)) \\
&\leftrightarrow \mathrm{isequiv}(\mathrm{total}(f))
\end{aligned}
$$

□

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通りである.

Definition 2.7.1 ([9] p125) 型 A, B について, 2 つの関数レトラクション $r : A \to B$ とセクション $s : B \to A$ 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 2.7.2 ([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 2.7.3 ([9] Definition 4.7.2) 関数 $g : A \to B$ が関数 $f : X \to Y$ のレトラクトであるとは, 4 つのホモトピー

$$R : r \circ s \sim \mathrm{id}_A$$
$$R' : r' \circ s' \sim \mathrm{id}_b$$
$$L : f \circ s \sim s' \circ g$$
$$K : g \circ r \sim r' \circ f$$

が存在するような $r : X \to A$, $s : A \to X$, $r' : Y \to B$, $s' : B \to Y$ 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 2.7.4 ([9] Lemma 4.7.3) 関数 $g : A \to B$ が関数 $f : X \to Y$ のレトラクトであるならば, $\mathrm{fib}_g(b)$ は $\mathrm{fib}_f(s'(b))$ のレトラクトである. ただし, s' は Definition 2.7.3 のものとする.

Proof. 証明の概略を述べる. Definition 2.7.3 の通りに記号を定める. 任意の $b : B$ に対して,

$$\varphi_b : \mathrm{fib}_g(b) \to \mathrm{fib}_f(s'(b)), \quad \varphi(a, p) :\equiv (s(a), L(a) \cdot s'(p))$$
$$\psi_b : \mathrm{fib}_f(s'(b)) \to \mathrm{fib}_g(b), \quad \psi(x, q) :\equiv (r(x), K(x) \cdot r'(q) \cdot R'(b))$$

と定義したとき, ($\sum$ の induction から) 任意の $(a, p) : \mathrm{fib}_g(b)$ 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 2.7.5 ([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 2.7.4 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, $\mathrm{isequiv}(f) \leftrightarrow \mathrm{isContr}(f)$ と Lemma 2.7.2 から $\mathrm{isequiv}(g)$ が分かる. □

2.8 関数の外延性を Univalence axiom から導く

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の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 2.8.1 ([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 $B : A \to \mathcal{U}$ に対して

$$\left(\prod_{x:A} \mathrm{isContr}(B(x))\right) \to \mathrm{isContr}\left(\prod_{x:A} B(x)\right)$$

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 2.8.2 ([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,

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

の同型射は, (e(の同型射) $\circ -$) で与えられる.

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

Corollary 2.8.3 ([9] Corollary 4.9.3) $P : A \to \mathcal{U}$ を可縮な型の族, つまり $\prod_{x:A} \mathrm{isContr}(P(x))$ とする. このとき, 射影 $\mathrm{pr}_1 : (\sum_{x:A} P(x)) \to A$ は equivalence である. 更に, $\mathcal{U}$ が univalent であれば,

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

である.

Proof. 前半について, Lemma 2.6.6 から, $x : A$ について, $\mathrm{fib}_{\mathrm{pr}_1}(x) \equiv P(x)$ である. P が可縮な型の族なので, $\mathrm{isContr}(\mathrm{pr}_1)$ が成り立つ. 後半については, Lemma 2.8.2 よりわかる. □

上記の $\alpha \equiv (\mathrm{pr}_1 \circ -)$ 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 $\mathrm{isContr}(\mathrm{fib}_\alpha(\mathrm{id}_A))$ である. よって, $\prod_{x:A} P(x)$ が $\mathrm{fib}_\alpha(\mathrm{id}_A)$ 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 2.8.4 ([9] Theorem 4.9.4) $\mathcal{U}$ が univalent であるとし, $P : A \to \mathcal{U}$ を可縮な型の族とする. $\alpha : (A \to \sum_{x:A} P(x)) \equiv (A \to A)$ とすると, $\prod_{x:A} P(x)$ は $\mathrm{fib}_\alpha(\mathrm{id}_A)$ 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 $\varphi : (\prod_{x:A} P(x)) \to \mathrm{fib}_\alpha(\mathrm{id}_A)$, $\psi : \mathrm{fib}_\alpha(\mathrm{id}_A) \to \prod_{x:A} P(x)$ をそれぞれ,

$$\begin{aligned} \varphi(f) &:\equiv (\lambda x.\ (x, f(x)), \mathrm{refl}_{id_A}) \\ \psi(g, p) &:\equiv \lambda x.\ \mathrm{happly}(p, x)_*(\mathrm{pr}_2(g(x))) \end{aligned}$$

と定める. このとき,

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

より成り立つ. □

Theorem 2.8.5 ([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

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

である. このとき,

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

は, $\lambda g.\ \mathrm{happly}(f,g)$ が fiberwise equivalence であるということなので, Theorem 2.6.9 から,

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

が equivalence であればよい. ここで, Lemma 2.6.4 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 2.6.4 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 2.8.6 Theorem 2.8.5 の証明には univalent axiom は用いていない.

参考文献

[1] Henk Barendregt, *Lambda Calculi with Types*. In S. Abramsky, Dov M. Gabbay, S. E. Maibaum, *Handbook of logic in computer science (vol. 2): background: computational structures*, Oxford University Press, 1993.

[2] Coq Team, *The Coq Standard Library*, `https://coq.inria.fr/distrib/current/stdlib/`, 2023.

[3] 萩原 学/アフェルト・レナルド, *Coq/SSReflect/Mathcomp*, 森北出版, 2018.

[4] Victor Kac, Pokman Cheung, *Quantum Calculus*, Springer, 2001.

[5] Mathematical Components Team, *Mathematical Components*, `https://github.com/math-comp/math-comp`, 2023.

[6] Mathematical Components Team, *Mathematical Components compliant Analysis Library*, `https://github.com/math-comp/analysis`, 2023.

[7] 中村 薫, *q-analogue*, `https://github.com/nakamurakaoru/q-analogue/tree/thesis`, 2023.

[8] 梅村 浩, 『楕円関数論 楕円曲線の解析学』, 東京大学出版会, 2000.

[9] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, `https://homotopytypetheory.org/book`, 2013.