

# 少人数クラス

中村 薫

2022 年 10 月 10 日

## 目次

<b>1</b>	<b>Coq による <math>q</math>-類似の形式化</b>	<b>1</b>
1.1	Coq	1
1.2	$q$ -類似	2
1.3	形式化	2
<b>2</b>	<b>HoTT</b>	<b>9</b>
<b>3</b>	<b>coqgen プロジェクト</b>	<b>9</b>
3.1	whileloop のコンパイル	9
3.2	insertion sort の証明	10

## 1 Coq による $q$ -類似の形式化

### 1.1 Coq

Coq とは, 定理証明支援系の 1 つであり, 数学的な証明が正しい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. 例えば, 命題  $P, Q$  について,  $P \Rightarrow Q$  かつ  $P$  であれば,  $Q$  が成り立つということは, Coq では

```
From mathcomp Require Import ssreflect.
```

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

```
Proof.
```

```
  move=> [] pq p.
```

```
  by apply pq.
```

```
Qed.
```

と表現できる.

Coq による証明は, Curry-Howard 同型と呼ばれる,

命題  $\leftrightarrow$  型

証明  $\leftrightarrow$  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, 各論理演算子について, 以下のように対応している

$P$  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。

## 1.2 $q$ -類似

$q$ -類似とは、 $q \rightarrow 1$  とすると通常の数学に一致するような拡張のことである。例えば、自然数  $n$  の  $q$ -類似  $[n]$  は

$$[n] = 1 + q + q^2 + \cdots + q^{n-1}$$

であり、 $(x-a)^n$  の  $q$ -類似  $(x-a)_q^n$  は

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

である。本章では、 $D_q(x-a)_q^n = [n](x-a)_q^{n-1}$  (ここで、 $D_q$  は微分の  $q$ -類似) が、 $n$  を整数に拡張しても成り立つことの形式化を目標とする。

## 1.3 形式化

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 1.3.1** ([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  $q$  差分を  $D_q f(x)$  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。

この定義を形式化すると、

```
From mathcomp Require Import all_ssreflect all_algebra.
Import GRing.
```

```
Section q_analogue.
```

```
Local Open Scope ring_scope.
```

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

```
Hypothesis Hq : q - 1 ≠ 0.
```

```
Notation "f // g" := (fun x => f x / g x) (at level 49).
```

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

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

となる。このコードの意味は大まかに以下のとおりである。

- 最初の 2 行で必要なライブラリの指定をしている.
- **Variable** でそのセクション内で共通して使う変数を宣言している.  $\mathbf{R}$  が  $\mathbf{Coq}$  における実数 (正確には  $\mathbf{mathcomp}$  の  $\mathbf{algebra}$  の実数) の役割を果たす. ここではまだ出てきていないが,  $\mathbf{nat}$  が 0 を含む自然数を,  $\mathbf{int}$  が整数に対応する.
- **Hypothesis** で,  $q$  が 1 でないという仮定をしている. 使いやすさのため,  $q \neq 1$  ではなく  $q - 1 \neq 0$  という形にしている.
- **Notation** で関数同士の割り算の記法を定義している.
- 2 つの **Definition** で  $q$ -差分と  $q$ -微分をそれぞれ定義している.  $:=$  以前に定義の名前と引数, 以後に具体的な定義が書いてある. 例えば  $q$ -差分についてであれば,  $\mathbf{d\_q}$  が名前,  $\mathbf{f}$  と  $\mathbf{x}$  が引数,  $\mathbf{f}(q * \mathbf{x}) - \mathbf{f} \mathbf{x}$  が定義である. ( $\mathbf{f}$  の後ろの:  $\mathbf{R} \rightarrow \mathbf{R}$  は  $\mathbf{f}$  の型である. 一方, もう一つの引数である  $\mathbf{x}$  には型を書いていない. これは,  $\mathbf{Coq}$  には強力な型推論があるため, 推論できるものであれば型を書く必要がないためである.)  $D_q$  の定義の中の  $\mathbf{id}$  は恒等関数のことである.

**Remark 1.3.2**  $f$  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, 本稿においては極限操作に関しての形式化は扱わない.

次に,  $x^n (n \in \mathbb{N})$  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 1.3.3** ([1] p2 (1.9))  $n \in \mathbb{N}$  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 1.3.4** ([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**  $\mathbf{qnat} \ n : \mathbf{R} := (q^n - 1) / (q - 1)$ .

**Lemma**  $\mathbf{qderiv\_of\_pow} \ n \ x :$

$x \neq 0 \rightarrow D_q (\mathbf{fun} \ x \Rightarrow x^n) \ x = \mathbf{qnat} \ n * x^{n-1}$ .

**Proof.**

**move**  $\Rightarrow$   $\mathbf{Hx}$ .

**rewrite**  $/Dq /dq /qnat$ .

**rewrite**  $\{-4\}(\mathbf{mul1r} \ x) \ \mathbf{-mulrBl} \ \mathbf{expfzM1}$ .

```

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

```

ここでも、コードについて少し説明を加える。

- **Definition** と同様, **Lemma** について,  $:=$  の前に補題の名前と引数が, 後に補題の主張が書いてある. 今回であれば, `qderiv_of_pow` が補題の名前で, `n` と `x` が引数である.
- **Proof.** 以下が補題の証明である.
- `def` が定義のとき, `rewrite /def` で定義を展開している.
- `lem` が  $A = B$  という形の補題のとき, `rewrite lem` で結論に出現する  $A$  を  $B$  に書き換えている.
- `red_frac_r` は,

$$\text{red\_frac\_r} : \forall x y z : \mathbb{R}, z \neq 0 \rightarrow x * z / (y * z) = x / y$$

という補題である. この補題を使うため, もともとはなかった  $x \neq 0$  という前提を加えている. 実際,  $D_q$  の定義において分母に  $x$  が出現するので,  $x$  が  $0$  でないという前提は妥当である.

**Remark 1.3.5** `qnat` という名前であるが, 実際には `n` の型は `nat` ではなく `R` にしている. また, `qderiv_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.

[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.

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

**Definition 1.3.6** ([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 1.3.7**  $n \in \mathbb{Z}_{>0}$  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と,

```

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

```

となる。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。補題の証明については

```

Theorem qderiv_qpoly_nonneg a n x :
  x ≠ 0 → Dq (qpoly_nonneg a n.+1) x = qnat n.+1 * qpoly_nonneg a n x.

```

Proof.

```

move=> Hx.
elim: n => [n IH].
- rewrite /Dq /dq /qpoly_nonneg /qnat.
  rewrite !mulr mulr1 expr1z.
  rewrite opprB subrKA !divff //.
  by rewrite denom_is_nonzero.
- rewrite ( _ : Dq (qpoly_nonneg a n.+2) x =
    Dq ((qpoly_nonneg a n.+1) **
      (fun x => (x - q ^ (n.+1) * a))) x) //.
  rewrite qderiv_prod' //.
  rewrite [Dq (+%R^~ (- (q ^ n.+1 * a))) x] /Dq /dq.
  rewrite opprB subrKA divff //.
  rewrite mulr1 exprSz.
  rewrite -[q * q ^ n * a] mulrA -(mulrBr q) IH.
  rewrite -[q * (x - q ^ n * a) * (qnat n.+1 * qpoly_nonneg a n x)] mulrA.
  rewrite [(x - q ^ n * a) * (qnat n.+1 * qpoly_nonneg a n x)] mulrC.
  rewrite -[qnat n.+1 * qpoly_nonneg a n x * (x - q ^ n * a)] mulrA.
  rewrite ( _ : qpoly_nonneg a n x * (x - q ^ n * a) = qpoly_nonneg a n.+1 x) //.
  rewrite mulrA.
  rewrite -{1}(mulr (qpoly_nonneg a n.+1 x)).
  rewrite -mulrDl addrC.
  rewrite -(@divff _ (q - 1)) //.
  rewrite [qnat n.+1] /qnat.
  rewrite [q * ((q ^ n.+1 - 1) / (q - 1))] mulrA.
  rewrite (add_div _ _ (q - 1)) //.
  by rewrite mulrBr -exprSz mulr1 subrKA.
by apply denom_is_nonzero.

```

Qed.

となる。ここで  $\text{elim:n}$  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、以下のようになる。

**Proposition 1.3.8** ([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.

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

より成立する。

□

この形式化は次のとおりである.

**Lemma** `qpoly_nonneg_explaw` `x a m n` :  
`qpoly_nonneg a (m + n) x =`  
`qpoly_nonneg a m x * qpoly_nonneg (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, `qpoly_nonneg` が再帰的に定義されているため, 形式化の証明では `m, n` に関する帰納法を用いている.

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 1.3.9** ([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** `qpoly_neg` `a n x` := `1 / qpoly_nonneg (q ^ ((Negz n) + 1) * a) n x.`

となる. ここで, `Negz n` とは `Negz n = - n.+1` をみたすものであって, `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,

**Definition** `qpoly` `a n x` :=

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

と定義できる.

整数に拡張した  $(x - a)_q^n$  も,  $q$ -微分にたいしてうまく振る舞う.

**Proposition 1.3.10** ([1] p10 Proposition3.3)  $n \in \mathbb{Z}$  について,

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

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 1.3.7 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 1.3.9 と, 商の微分公式の  $q$ -類似版である

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

及び Proposition 1.3.7 を用いて示される. □

この補題の証明の形式化が本章の目標である.[1]と同じ方針で証明する. まず,  $n = 0$  のとき,

**Lemma** `qderiv_qpoly_0`  $a\ x$  :

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

**Proof.** `by rewrite Dq_const qnat_0 mul0r. Qed.`

である. ここで, `Dq_const` は

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

**Proof.** `by rewrite /Dq /dq addrK' mul0r. Qed.`

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

**Lemma** `qderiv_qpoly_neg`  $a\ n\ x$  : `q ≠ 0 → x ≠ 0 →`

`(x - q ^ (Negz n) * a) ≠ 0 →`

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

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

**Proof.**

`move=> Hq0 Hx Hqn Hqpoly.`

`destruct n.`

`- by rewrite /Dq /dq /qpoly_neg /= addrK' qnat_0 !mul0r.`

`- rewrite qderiv_quot //.`

`rewrite Dq_const mulr0 mul1r sub0r.`

`rewrite qderiv_qpoly_nonneg // qpoly_qx // -mulNr.`

`rewrite [qpoly_nonneg (q ^ (Negz n.+1 + 1) * a) n.+1 x *  
(q ^ n.+1 * qpoly_nonneg (q ^ (Negz n.+1 + 1 - 1) *  
a) n.+1 x)] mulrC.`

`rewrite -mulf_div.`

`have → : qpoly_nonneg (q ^ (Negz n.+1 + 1) * a) n x /  
qpoly_nonneg (q ^ (Negz n.+1 + 1) * a) n.+1 x =  
1 / (x - q ^ (- 1) * a).`

`rewrite -(mulr1`

`(qpoly_nonneg (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) //.`

`rewrite -{1}(add0r (Posz n)).`

`by rewrite addrKA.`

`by rewrite /=; apply mulnon0 in Hqpoly.`

`rewrite mulf_div.`

`rewrite -[q ^ n.+1 *`

`qpoly_nonneg (q ^ (Negz n.+1 + 1 - 1) * a) n.+1 x *  
(x - q ^ (-1) * a)]mulrA.`

`have → : qpoly_nonneg (q ^ (Negz n.+1 + 1 - 1) * a) n.+1 x *  
(x - q ^ (-1) * a) =`

`qpoly_nonneg (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 /qpoly_neg /=.`

`rewrite ( _ : Negz n.+2 + 1 = Negz n.+1) //.`

`rewrite -mulf_div.`

`congr ( _ * _).`

`rewrite NegzE mulrC.`

```

rewrite /qnat.
rewrite -mulNr mulrA.
congr (_ / _).
rewrite opprB mulrBr mulr1 mulrC divff.
  rewrite invr_expz.
  rewrite (_ : - Posz n.+2 + 1 = - Posz n.+1) //.
  rewrite -addn1.
  rewrite (_ : Posz (n.+1 + 1)%N = Posz n.+1 + 1) //.
  rewrite addrC.
  rewrite [Posz n.+1 + 1]addrC.
  by rewrite -{1}(add0r 1) addrKA sub0r.
  by rewrite expnon0 //.
rewrite qpoly_qx // mulf_neq0 //.
  by rewrite expnon0.
rewrite qpoly_nonneg_head mulf_neq0 //.
rewrite (_ : Negz n.+1 + 1 - 1 = Negz n.+1) //.
  by rewrite addrK.
move: Hqpoly ⇒ /=.
move/mulnon0.
by rewrite addrK mulrA -{2}(expriz q) -expfzDr.

```

**Qed.**

と、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、`qderiv_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 1.3.9 において分母に現れるため、`qderiv_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、

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

**Proof.**

```

move⇒ Hq0 Hx Hxqa Hqpoly.
case: n Hxqa Hqpoly ⇒ [|/=] n Hxqa Hqpoly.
- destruct n.
  + by rewrite qderiv_qpoly_0.
  + rewrite qderiv_qpoly_nonneg //.
    rewrite (_ : Posz n.+1 - 1 = n) //.
    rewrite -addn1.
    rewrite (_ : Posz (n + 1)%N = Posz n + 1) //.
    by rewrite addrK.
- rewrite Dq_qpoly_int_to_neg qderiv_qpoly_neg //.
  rewrite NegzK.
  rewrite (_ : (n + 1).+1 = (n + 0).+2) //.
  by rewrite addn0 addn1.
  rewrite (_ : Negz (n + 1) = Negz n - 1) //.
  by apply itransposition; rewrite NegzK.
  by rewrite NegzK addn1.

```

**Qed.**

と形式化できる。 `case: n` で  $n$  が 0 以上か負かで場合分けを行い、`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 `qderiv_qpoly_0`, `qderiv_qpoly_nonneg`, `qderiv_qpoly_neg` を使っていることが見て取れる。

ここまでの現在形式化できている主な内容である。 今後は、まずは  $(x - a)_q^{m+n}$  の指数法則が  $m, n$  が整数の場合にも成り立つことの形式化を行い、最終的には Jacobi の三重積公式 ([1] Theorem 11.1) の形式化を目標としたい。



## 2 HoTT

### 3 coqgen プロジェクト

coqgen プロジェクトとは, OCaml という関数型言語の型推論の正しさを Coq で確認することを目標としたプロジェクトであり, 具体的には OCaml から Coq へのコンパイラの作成と, コンパイルされたプログラムに対する証明を行っている. 本章では, RA として本プロジェクトに参加し得られた結果の一部を扱う.

#### 3.1 whileloop のコンパイル

OCaml での whileloop を使う例として,

```
let fact n =  
  let i = ref n in let v = ref 1 in  
  while !i > 0 do v := !v * !i; i := !i - 1 done; !v
```

という階乗を求めるプログラムをコンパイルすることを考える. まず, Coqgen のライブラリに whileloop のための関数を用意しておく.

```
Fixpoint whileloop (h : nat) (f : M bool) (b : M unit) : M unit :=  
  if h is h.+1 then  
    do v <- f; if v then (do _ <- b; whileloop h f b) else Ret tt  
  else FailGas.
```

ここで, Coq の性質上, ガス  $h$  とモナド  $M$  が必要になる.

- Coq では, 停止するかわからない再帰関数を定義することはできない. そこで引数  $h$  を与え, ループを回すごとに 1 ずつ減らすことで停止性を保証している.
- Coq は純粋に数学的な関数しか扱えないため, OCaml の store や例外を表現するためにモナドを用いている. 型  $T$  に対して, store を  $Env$ , 例外を  $Exn$  で表し,

**Definition**  $M\ T := Env \rightarrow Env * (T + Exn).$

と定義している.

この whileloop を使って, 以下のようにコンパイルされる.

```
Definition fact (h : nat) (n : coq_type ml_int) : M (coq_type ml_int) :=  
  do i <- newref ml_int n;  
  do v <- newref ml_int 1%int63;  
  do _ <-  
  whileloop h (do v_1 <- getref ml_int i; ml_gt h ml_int v_1 0%int63)  
    (do _ <-  
      (do v_1 <-  
        (do v_1 <- getref ml_int i;  
          do v_2 <- getref ml_int v; Ret (Int63.mul v_2 v_1));  
          setref ml_int v v_1);  
        do v_1 <- (do v_1 <- getref ml_int i; Ret (Int63.sub v_1 1%int63));  
          setref ml_int i v_1);  
      getref ml_int v.
```

となる. `do` が多くやや複雑であるが, `newref`, `getref`, `setref` がそれぞれ `ref`, `!`, `:=` に対応しているので, 全体としては同じ構造になっていることがわかる.

### 3.2 insertion sort の証明

#### 参考文献

- [1] Victor Kac, Pokman Cheung, *Quantum Calculus*, Springer, 2001.
- [2] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*