

再帰的な定義と帰納法

1 プログラミング言語としての Coq

Coq では、関数型言語のようにプログラムが書ける。

注意：Coq では各命令が”.” で終わる。

定義と関数

```
Definition one : nat := 1. (* 定義 *)
one is defined
```

```
Definition one := 1. (* 再定義はできない *)
Error: one already exists.
```

```
Definition one' := 1. (* 型を書かなくてもいい *)
Print one'. (* 定義の確認 *)
one' = 1
: nat (* nat は自然数の型 *)
```

```
Definition double x := x + x. (* 関数の定義 *)
Print double.
double = fun x : nat => x + x (* 関数も値 *)
: nat -> nat (* 関数の型 *)
```

```
Eval compute in double 2. (* 式を計算する *)
= 4
: nat
```

```
Definition double' := fun x => x + x. (* 関数式で定義 *)
Print double'.
double' = fun x : nat => x + x
: nat -> nat
```

```
Definition quad x := let y := double x in 2 * y. (* 局所的な定義 *)
Eval compute in quad 2.
= 8
: nat
```

```
Definition quad' x := double (double x). (* 関数適用の入れ子 *)
Eval compute in quad' 2.
= 8
: nat
```

```
Definition triple x :=
  let double x := x + x in (* 局所的な関数定義。上書きもできる *)
  double x + x.
Eval compute in triple 3.
```

```
= 9
: nat
```

計算の仕組み

Coq の計算がいくつかの簡約規則に基づいています。

δ 簡約 証明の中で Definition を展開するために `rewrite /def` を使うが、`compute` はそれを勝手にやっている。

β 簡約 $(\text{fun } x \Rightarrow \text{Body}) \text{ Arg}$ のような関数適用は代入で評価される。

$$(\text{fun } x \Rightarrow B) A \longrightarrow [A/x]B$$

x が束縛変数なので、述語論理と同様に名前の衝突が回避される。

ζ 簡約 `let x := Def in Body` も同様に代入になる。

$$\text{let } x := D \text{ in } B \longrightarrow [D/x]B$$

ι 簡約 $(\text{fix } f \text{ } x \Rightarrow \text{Body}) \text{ Arg}$ の場合、 Arg を代入するだけでなく、 $(\text{fix } f \dots)$ 自身を f に代入する。

$$(\text{fix } f \text{ } x \Rightarrow B) A \longrightarrow [(\text{fix } f \text{ } x \Rightarrow B)/f, A/x]B$$

データ型の定義

```
Inductive janken : Set :=
  | gu
  | choki
  | pa.
(* じゃんけんの手 *)
```

```
Definition weakness t :=
  match t with
  | gu    => pa
  | choki => gu
  | pa    => choki
end.
(* 弱点を返す *)
(* 簡単な場合分け *)
```

```
Eval compute in weakness pa.
= choki
: janken
```

```
Print bool.
```

```
Inductive bool : Set := true : bool / false : bool
```

```
Print janken.
```

```
Inductive janken : Set := gu : janken / choki : janken / pa : janken
```

```
Definition wins t1 t2 :=
  match t1, t2 with
  | gu, choki => true
  | choki, pa => true
  | pa, gu => true
  | _, _ => false
end.
(* 「t1はt2に勝つ」という関係 *)
(* 二つの値で場合分け *)
(* 残りは全部勝たない *)
```

```

Check wins.
wins : janken -> janken -> bool          (* 関係は bool への多引数関数 *)
Eval compute in wins gu pa.
      = false
      : bool

```

場合分けによる証明

```

Lemma weakness_wins t1 t2 :
  wins t1 t2 = true <-> weakness t2 = t1.
Proof.
  split.
  - by case: t1; case: t2.                (* 全ての場合を考える *)
  - move=> <-; by case: t2.              (* t2 の場合分けで十分 *)
Restart.
  case: t1; case: t2; by split.          (* 最初から全ての場合でも OK *)
Qed.

```

再帰データ型と再帰関数

```

Module MyNat.                             (* nat を新しく定義する *)
Inductive nat : Set :=
  | 0 : nat
  | S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined

```

```

Fixpoint plus (m n : nat) {struct m} : nat := (* 帰納法の対象を明示する *)
  match m with
  | 0 => n                                     (* 減らないとエラーになる *)
  | S m' => S (plus m n)
  end.

```

Error: Recursive definition of plus is ill-formed.

In environment ...

Recursive call to plus has principal argument equal to m instead of m'.

```

Fixpoint plus (m n : nat) {struct m} : nat := (* 同じ型の引数をまとめる *)
  match m with
  | 0 => n
  | S m' => S (plus m' n)                    (* 正しい定義 *)
  end.

```

plus is recursively defined (decreasing on 1st argument)

Print plus.

```

plus = fix plus (m n : nat) : nat := match m with
  | 0 => n
  | S m' => S (m' + n)
  end
      : nat -> nat -> nat

```

```

Check plus (S (S 0)) (S 0).                (* 式の型を調べる *)
plus (S (S 0)) (S 0)
      : nat

```

定義	Definition f ... :=
再帰的な定義	Fixpoint f ... {struct x} :=
データ型の定義	Inductive t : Set := a b : t -> t c .
局所的な定義	let x := ... in ...
局所関数	fun x => ...
局所再帰関数	fix f ... {struct x} := ...
if 文	if ... then ... else ...
場合分け	match ... with pat ₁ => pat _n => ... end

表 1: Coq の基本的な構文

```
Eval compute in plus (S (S 0)) (S 0). (* 式を評価する *)
= S (S (S 0))
: nat
```

```
Fixpoint mult (m n : nat) struct m : nat := 0.
```

```
Eval compute in mult (S (S 0)) (S 0). (* 期待している値 *)
= S (S 0)
: nat
End MyNat.
```

練習問題 1.1 mult を正しく定義せよ.

2 帰納法による証明

Coq でデータ型を定義すると、自動的に帰納法の原理が生成される.

```
Check nat_ind.
nat_ind
: ∀ P : nat -> Prop,
  P 0 ->
  (∀ n : nat, P n -> P (S n)) ->
  ∀ n : nat, P n
```

もっと分かりやすく書くと、nat_ind の型は

$$\forall P, P 0 \rightarrow (\forall n, P n \rightarrow P (S n)) \rightarrow (\forall n, P n)$$

である. 即ち P は 0 でなりたち, 任意の n について P が n でなりたてば, $n+1$ でもなりたつことが証明できれば, 任意の n について P がなりたつ.

算数の様々な性質を帰納法で証明できる. elim は焦点の型によって帰納法の原理を選び, それを結論に適用する.

```
Lemma plusS m n : m + S n = S (m + n). (* m, n は仮定 *)
Proof.
  elim: m => /=. (* nat_ind を使う *)
```

```

- done. (* 0 の場合 *)
- move => m IH. (* S の場合 *)
  by rewrite IH. (* 帰納法の仮定で書き換える *)
Restart.
  elim: m => /= [|m ->] //. (* 一行にまとめた *)
Qed.
Check plusnS. (*  $\forall m n : \text{nat}, m + S n = S (m + n)$  *)

Lemma plusSn m n : S m + n = S (m + n).
Proof. rewrite /=. done. Show Proof. Qed. (* 簡約できるので帰納法は不要 *)

Lemma plusn0 n : n + 0 = n.
Admitted. (* 定理を認めて証明を終わらせる *)
Lemma plusC m n : m + n = n + m.
Admitted.
Lemma plusA m n p : m + (n + p) = (m + n) + p.
Admitted.

Lemma multnS m n : m * S n = m + m * n.
Proof.
  elim: m => /= [|m ->] //.
  by rewrite !plusA [n + m]plusC.
Qed.

Lemma multn0 n : n * 0 = 0.
Admitted.
Lemma multC m n : m * n = n * m.
Admitted.
Lemma multnDr m n p : (m + n) * p = m * p + n * p.
Admitted.
Lemma multA m n p : m * (n * p) = (m * n) * p.
Admitted.

Fixpoint sum n :=
  if n is S m then n + sum m else 0.
Print sum. (* if .. is は match .. with に展開される *)

Lemma double_sum n : 2 * sum n = n * (n + 1).
Admitted.
Lemma square_eq a b : (a + b) * (a + b) = a * a + 2 * a * b + b * b.
Admitted. (* 帰納法なしで証明できる *)

```

練習問題 2.1 前回のタクティックを参考にし、上の Admitted を全て証明せよ。