

Coq で関数型プログラミング

1 Coq とは

通常のプログラミング言語でプログラムを書くと、しばしばバグに悩まされる。Coq の開発に
使われる OCaml という型付き関数型プログラミング言語では、型システムが多くのバグを見つ
けてくれることが知られている。しかし、OCaml の型推論で全てのバグを取るのが不可能である。
プログラムの正しさは、満たすべき性質を定義し、それを証明しなければならない。しかし、通
常の型システムではそういう性質を書くことができない。

Coq は型理論に基づいた定理証明支援系である。具体的には、論理式の形で定理を述べるこ
とができ、コンピュータとの対話で証明を作ることもできる。しかも、Coq の中で OCaml とよく
似た形でプログラムが書けるので、そのプログラムを対象とした性質や証明もできる。最終的に
作ったプログラムを OCaml などに自動的に翻訳する機能も付いているので、証明付きのプログ
ラムが作れる訳である。ちなみに、Coq 自身も OCaml で書かれている。そもそも、1980 年代に
Caml を開発したのは Coq を作るためであった。

残念ながら、日本語の資料が非常に少ないので、以下の URL は英語で書かれている。

<http://coq.inria.fr/>

Coq の開発元。英語での資料と処理系が置いてある。

<http://www.proofcafe.org/>

Proof Café. 名古屋における Coq のコミュニティで、様々な日本語資料もある。

2 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.
```

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

```
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

整数とモジュール

Eval compute in 1 - 2. (* 自然数の引き算は整数と違う *)
= 0
: nat

Require Import ZArith. (* 整数を使ってみよう *)

Module Z. (* 定義の範囲を区切るために Module を使う *)
 Open Scope Z_scope. (* 数値や演算子を整数として解釈する *)

Eval compute in 1 - 2.
= -1
: Z (* Z は整数の型 *)

Eval compute in (2 + 3) / 2. (* 割り算も使える *)
= 2
: Z

Definition p (x y : Z) := 2 * x - y * y. (* 多引数の関数 *)
Print p.
p = fun x y : Z => 2 * x - y * y
: Z -> Z -> Z (* 多引数の関数の型 *)

Eval compute in p 3 4.
= -10
: Z

Definition p' := fun x => fun y => 2 * x - y * y. (* 関数式で *)
Print p'.
p' = fun x y : Z => 2 * x - y * y
: Z -> Z -> Z

```

Definition q := p 3.                                     (* 部分適用 *)

Eval compute [p q] in q.                                (* p と q の定義だけを展開する *)
  = fun y : Z => 2 * 3 - y * y
  : Z -> Z

Eval compute in q 4.
  = -10
  : Z

Eval compute in let x := 0 in q x.                      (* q 中の x の値は変わらない *)
  = 6
  : Z

End Z.
Module Z is defined
Print Z.p.                                              (* Module の中味へのアクセス *)
Z.q = Z.p 3
  : Z -> Z

Eval compute in 1 - 2.                                  (* Scope は元に戻る *)
  = 0
  : nat

```

練習問題 2.1 Z の中で二つの整数値の平均を計算する関数 `heikin : Z -> Z -> Z` を定義せよ。

データ型の定義

```

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

Definition weakness t :=                                (* 弱点を返す *)
  match t with
  | gu    => pa
  | choki => gu
  | pa    => choki
  end.                                                (* 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 :=                               (* 「t1 は t2 に勝つ」という関係 *)
  match t1, t2 with
  | gu, choki => true
  | choki, pa => true
  | pa, gu => true
  | _, _ => false
  end.                                                (* 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
```

```
Module Play2.
```

(* 二人でゲームしよう *)

```
Inductive winner : Set :=
| first
| second
| aiko.
```

```
Definition play t1 t2 :=
  if wins t1 t2 then first else
  if wins t2 t1 then second else
  aiko.
```

```
Eval compute in play gu pa.
= second
: winner
```

```
Eval compute in play choki choki.
= aiko
: winner
```

```
End Play2.
```

```
Print andb.
```

```
Print orb.
```

```
Module Play3.
```

```
Inductive winner : Set :=
| first
| second
| third
| aiko.
```

```
Definition play (t1 t2 t3 : janken) : winner := aiko.
End Play3.
```

練習問題 2.2 Play3.play を正しく定義せよ .

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 :=
  match m with
  | 0 => n
  | S m' => S (plus m n)
  end.
```

(* 帰納法の対象を明示する *)

(* 減らないとエラーになる *)

```
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)
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).
plus (S (S 0)) (S 0)
    : nat

Eval compute in plus (S (S 0)) (S 0).
    = S (S (S 0))
    : nat

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

Eval compute in mult (S (S 0)) (S 0).
    = S (S 0)
    : nat
End MyNat.

```

(* 正しい定義 *)

(* 式の型を調べる *)

(* 式を評価する *)

(* 期待している値 *)

練習問題 2.3 mult を正しく定義せよ .

文字列の扱い

```

Require Import Ascii String.
Open Scope string_scope.

Definition s := "hello".
Print s.
s = "hello"
    : string
Print string.
Inductive string : Set :=
    EmptyString : string | String : ascii -> string -> string
Print ascii.
Inductive ascii : Set :=
    Ascii : bool ->
        bool -> bool -> bool -> bool -> bool -> bool -> bool -> ascii

Definition s2 := s ++ " " ++ "everybody".
Eval compute in s2.
    = "hello everybody"
    : string

Eval compute in (" ")%char.
    = " "%char
    : ascii

Definition remove_head_space s :=
    match s with

```

(* 必要なライブラリーを読み込む *)
(* 文字列リテラルを使えるようにする *)

(* 文字列の結合 *)

(* 文字リテラル *)

(* 先頭の空白を一個取る *)

```

| String " " s' => s'
| _ => s
end.

```

```

Eval compute in remove_head_space " hello".
= "hello"
: string

```

```

Fixpoint remove_head_spaces (s : string) : string := "".

```

(* 先頭の空白を全て取る *)

練習問題 2.4 remove_head_spaces を正しく定義せよ .

Coq の基本的な構文

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

Coq のコマンド

Print f.	定義を出力する
Check	型を出力する
Eval compute in	評価結果を出力する
Module m.	モジュールを開始する
End m.	モジュールを閉じる