Jacques Garrigue, 2011 年 12 月 7 日

Coq で関数型プログラミング

1 Coq とは

今まで，Objective Caml を使うと，型システムが多くのバグを見つけてくれることを経験した．しかし，Objective Caml 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，OCaml 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 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．

2 Coq を使う前に

OCaml と同様に，Emacs の中で使うことになる．まず，設定をしなければならない．

講義のホームページから以下のファイルをダウンロードして下さい．

```
http://www.math.nagoya-u.ac.jp/~garrigue/lecture/2011_AW/coq.emacs
```

ダウンロードしたら，今までの`.emacs` を上書きする．Terminal を開いて，以下のコマンドを入力する．

```
$ cp Downloads/coq.emacs ~/.emacs
```

Emacs を起動してから名前が”`.v`” で終わるファイルを開くと ProofGeneral というインターフェースが起動する．

```
<C-x><C-f>test.v<ret>
```

ProofGeneral で以下のコマンドが使える．

`<C-c><C-n>`	一つのコマンドを処理する
`<C-c><C-u>`	最後のコマンドを撤回する
`<C-c><C-return>`	現在の入力位置までのコマンドを処理または撤回する

また，上のアイコンを使ってもいい．右三角 (▷) はコマンドの処理，左三角 (◁) は撤回，蝶ネックタイ (⋈) は現在の入力位置に移動．

3 プログラミング言語としての Coq

Coq では，OCaml と同じ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 := 1.                                          (* 定義 *)
one is defined

Definition double := fun x => x + x.                          (* 関数 *)
double is defined

Print double.                                                 (* 定義の確認 *)
double = fun x : nat => x + x
     : nat -> nat                                        (* nat は自然数の型 *)

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

Definition quad x := let y := double x in 2 * y.              (* let 文 *)
quad is defined

Definition twice (A : Set) (f : A -> A) (x : A) := f (f x).   (* 多相関数 *)
twice is defined

Print twice.                              (* 型は関数の引数になる．Set は型の型 *)
twice = fun (A : Set) (f : A -> A) (x : A) => f (f x)
     : forall A : Set, (A -> A) -> A -> A

Eval compute in twice nat quad 1.                        (* nat を明示的に渡す *)
     = 16
     : nat
```

再帰データ型

```
Inductive nat : Set :=
  | O : 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                                     (* 減らないとエラーになる *)
  | O => n
  | 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
  | O => 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
                                     | O => n
                                     | S m' => S (m' + n)
                                     end
     : nat -> nat -> nat

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

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

Parameter mult : nat -> nat -> nat.                         (* 仮の定数の宣言 *)
mult is assumed

Eval compute in mult (S O) (S O).
     = mult (S O) (S O)                          (* 仮の定数は評価できない *)
     : nat

Reset nat.                                  (* nat 以降を忘れる（元の定義に戻す） *)
```

練習問題 3.1 実際の mult を Fixpoint で定義せよ.

多相的な再帰データ型

```
Inductive list (A:Set) : Set :=                    (* list の型は Set -> Set *)
  | nil : list A
  | cons : A -> list A -> list A.
list is defined
list_rect is defined
list_ind is defined
list_rec is defined

Definition hd (A:Set) (l:list A) :=
  match l with                 (* パターンマッチングは網羅的でなければならない *)
  | cons a _ => a
  end.
Error: Non exhaustive pattern-matching: no clause found for pattern nil

Definition hd (A:Set) (d:A) (l:list A) :=          (* デフォルト値を用意する *)
  match l with
  | cons a _ => a
  | nil => d
  end.
hd is defined

Fixpoint append (A:Set) (l1 l2:list A) {struct l1} : list A :=
  match l1 with
  | nil => l2
  | cons a l' => cons A a (append A l' l2)
  end.
```

```
append is recursively defined (decreasing on 2nd argument)

Eval compute                                          (* 型を全て渡すのが面倒 *)
  in append nat (cons nat 1 (cons nat 2 (nil nat))) (cons nat 3 (nil nat)).
     = cons nat 1 (cons nat 2 (cons nat 3 (nil nat)))
     : list nat

Implicit Arguments nil [A].                           (* A を省略可能にする *)
Implicit Arguments cons [A].
Implicit Arguments append [A].

Eval compute in append (cons 1 (cons 2 nil)) (cons 3 nil).
     = cons 1 (cons 2 (cons 3 nil))                   (* 表示するときも省略 *)
     : list nat

Parameter length : forall (A:Set), list A -> nat.

Eval compute in length _ (cons 1 (cons 2 nil)).
```

練習問題 3.2 実際の `length` を定義せよ.

OCaml と Coq の対応表

Objective Caml	Coq
`let f ... = ... ;;`	`Definition f ... := ... .`
`let rec f ... = ... ;;`	`Fixpoint f ... {struct x} := ... .`
`let x = ... in ...`	`let x := ... in ...`
`fun x -> ...`	`fun x => ...`

`type 'a t =`
`  | Nil`
`  | Cons of 'a * 'a list ;;`
	`Inductive t (A:Set) : Set :=`
	`  | nil : t A`
	`  | cons : A -> t A -> t A .`

4 Coq での型付け

型付け規則 Coq の式は以下の型付け規則によって型付けされる.

変数 $\Gamma \vdash x : \tau \quad (x : \tau \text{は} \Gamma \text{に含まれる})$

抽象 $\dfrac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash \mathtt{fun}\ x{:}\theta \Rightarrow M : \theta \to \tau}$

適用 $\dfrac{\Gamma \vdash M : \theta \to \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash M\ N : \tau}$

定義 $\dfrac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \tau}{\Gamma \vdash \mathtt{let}\ x\ \mathtt{:=}\ M\ \mathtt{in}\ N : \tau}$

不動点 $\dfrac{\Gamma, f : \theta \to \tau, x : \theta \vdash M : \tau}{\Gamma \vdash \mathtt{fix}\ f\ (x{:}\theta)\ \mathtt{:=}\ M : \theta \to \tau}$

直積 $\dfrac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash (M, N) : \tau * \theta}$

型抽象 $\dfrac{\Gamma, A : Set \vdash M : \tau}{\Gamma \vdash \mathtt{fun}\ A{:}\mathrm{Set} \Rightarrow M : \forall A{:}\mathrm{Set}, \tau}$

型適用 $\dfrac{\Gamma \vdash M : \forall A{:}\mathrm{Set}, \tau \quad \Gamma \vdash T : \mathrm{Set}}{\Gamma \vdash M\ T : [T/A]\tau}$

型付けの例

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \mathsf{nat} \vdash \mathsf{S} : \mathsf{nat} \to \mathsf{nat} \quad \Gamma, x : \mathsf{nat} \vdash x : \mathsf{nat}}{\Gamma, x : \mathsf{nat} \vdash \mathsf{S}\ x : \mathsf{nat}}\ \text{適用}}{\Gamma \vdash \mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x : \mathsf{nat} \to \mathsf{nat}}\ \text{抽象} \qquad \Gamma \vdash \mathsf{O} : \mathsf{nat}}{\Gamma \vdash (\mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x)\ \mathsf{O} : \mathsf{nat}}\ \text{適用}$$