Une Introduction à la Vérification de Programmes avec Coq

Master Informatique, Histoire et épistémologie du calcul et de l'informatique,

Complément de cours

Reynald Affeldt
National Institute of Advanced Industrial Science and Technology

23 février 2016

Les définitions Coq qui correspondent au cours.

Table des matières

1 Vérification de programmes Coq

1.1 La fonction prédécesseur

```
Print nat.
Definition prec (n : nat) : nat :=
  match n with
  | O => O
  | S m => m
  end.

Compute prec 5.
```

```
Compute prec 0.

Recursive Extraction prec.
```

À utiliser avec précaution puisqu'elle retourne 0 pour 0.

1.2 La fonction prédécesseur partielle

Prend en argument une preuve que l'entrée est strictement positive.

```
Print False.
Definition false_nat (abs : False) : nat :=
  match abs with end.

Require Import Arith.
Check lt_irrefl.

Axiom faux : O < O.
Check (Nat.lt_irrefl _ faux).
Check (false_nat (Nat.lt_irrefl _ faux)).

Definition pprec (n : nat) : 0 < n -> nat :=
  match n with
  | O => fun H => false_nat (Nat.lt_irrefl _ H)
  | S m => fun _ => m
  end.

Recursive Extraction pprec.
```

1.3 Construire des preuves d'inégalités

```
Print le.
Check le_S _ _ (le_n 1).

Fixpoint spos (n : nat) : 1 <= S n :=
  match n with
  | O => le_n 1
  | S m => le_S _ _ (spos m)
  end.

Compute pprec 5 (spos _).
```

1.4 Construire des preuves d'égalités

```
Print eq.
Check (eq 0 1).
Check (@eq _ 0 1).
Check eq_refl 0.
Check (@eq_refl _ 0).
Check eq_refl (2 + 2) : 4 = 2 + 2.

About Nat.leb.
Print Nat.ltb.
```

```
About Nat.ltb_lt.

Definition pprecb (n : nat) : Nat.ltb 0 n = true → nat :=
  match n with
  | O ⇒ fun H ⇒ false_nat
    (Nat.lt_irrefl _ (proj1 (Nat.ltb_lt _ _) H))
  | S m ⇒ fun _ ⇒ m
  end.

Compute pprecb 5 eq_refl.
Fail Compute pprecb 0 eq_refl.

Recursive Extraction pprecb.
```

1.5 Fonction prédécesseur partielle complètement spécifiée

Tout est dans le type.

1.5.1 Progammation interactive

```
Print sig.
Print proj1_sig.
Check (exist (fun x ⇒ x = O) 0 eq_refl) : {x : nat | x = 0}.

Definition pprec_interactif (n : nat) :
  0 < n → {m | n = S m}.
destruct n as [|m].
- intros abs.
  generalize (Nat.lt_irrefl _ abs).
  destruct 1.
- intros _.
  apply (exist _ m).
  apply eq_refl.
Defined.

Print pprec_interactif.

Recursive Extraction pprec_interactif.
```

1.5.2 Progammation avec l'extension Program

```
Program Definition pre_auto (n : nat) : 0 < n → {m | n = S m} :=
  match n with
  | O ⇒ fun H ⇒ False_rect _ (Nat.lt_irrefl _ H)
  | S m ⇒ fun _ ⇒ exist (fun x ⇒ n = S x) m _
  end.

Obligation Tactic := idtac.
Program Definition pre_manual (n : nat) : 0 < n → {m | n = S m} :=
  match n with
  | O ⇒ fun H ⇒ False_rect _ (Nat.lt_irrefl _ H)
```

```
  | S m ⇒ fun _ ⇒ exist (fun x ⇒ n = S x) m _
  end.
Next Obligation.
intros n m mn _.
simpl.
rewrite mn.
apply eq_refl.
Qed.
Next Obligation.
intros n m mn Om.
simpl.
apply eq_refl.
Qed.

Print pre_auto.
Print pre_manual.
```

1.5.3 Progammation directe

```
About eq_ind.

Definition pre (n : nat) : 0 < n → {m | n = S m} :=
  (match n as n' return n = n' → _ with
  | O ⇒ fun _ H ⇒ False_rect _ (Nat.lt_irrefl _ H)
  | S m ⇒ fun Heq _ ⇒ exist (fun x ⇒ n = S x) m Heq
  end) eq_refl.

Print pre.

Compute proj1_sig (pre 5 (spos _)).
```

2 Vérification de programmes avec la logique de Hoare

2.1 Syntaxe des expressions arithmétiques et booléennes

```
Definition var := nat.

Inductive exp :=
| exp_var : var → exp
| cst : nat → exp
| mul : exp → exp → exp
| sub : exp → exp → exp.

Inductive bexp :=
| equa : exp → exp → bexp
| neg : bexp → bexp.
```

2.2 Un langage impératif minimal

```
Inductive cmd : Type :=
```

```
| assign : var → exp → cmd
| seq : cmd → cmd → cmd
| while : bexp → cmd → cmd.
```

2.3 Sémantique des expressions

État d'un programme :

```
Definition state := var → nat.

Definition sample_state : state :=
  fun x ⇒
    match x with
    | O ⇒ 4
    | 1 ⇒ 5
    | _ ⇒ O
    end.

Require Import Arith.

Definition upd (v : var) (a : nat) (s : state) : state :=
  fun x ⇒ match Nat.eq_dec x v with
           | left _ ⇒ a
           | right _ ⇒ s x
           end.
```

Évaluation des expressions :

```
Fixpoint eval e s :=
  match e with
  | exp_var v ⇒ s v
  | cst n ⇒ n
  | mul v1 v2 ⇒ eval v1 s × eval v2 s
  | sub v1 v2 ⇒ eval v1 s - eval v2 s
  end.

Fixpoint beval b s :=
  match b with
    | equa e1 e2 ⇒ eval e1 s = eval e2 s
    | neg b ⇒ ¬ beval b s
  end.
```

Exemple d'expression :

```
Definition ret : var := O.
Definition x : var := 1.

Compute eval (mul (exp_var ret) (exp_var x)) sample_state.
```

2.4 Répresentation de la logique de Hoare

Définition des pré/post-conditions :

```
Definition assert := state → Prop.
```

```
Definition imp (P Q : assert) := ∀ s, P s → Q s.
```

Les règles d'inférence :

```
Inductive hoare : assert → cmd → assert → Prop :=
| hoare_assign : ∀ (Q : assert) v e,
   hoare (fun s ⇒ Q (upd v (eval e s) s)) (assign v e) Q
| hoare_seq : ∀ Q P R c d,
   hoare P c Q → hoare Q d R →
   hoare P (seq c d) R
| hoare_conseq : ∀ (P' Q' P Q : assert) c,
   imp P P' → imp Q' Q → hoare P' c Q' →
   hoare P c Q
| hoare_while : ∀ P b c,
   hoare (fun s ⇒ P s ∧ beval b s) c P →
   hoare P (while b c) (fun s ⇒ P s ∧ ¬ (beval b s)).
```