Une Introduction à la Vérification de Programmes avec Coq

Histoire et épistémologie du calcul et de l'informatique, Master Informatique, Universite de Lille 1

Reynald Affeldt

National Institute of Advanced Industrial Science and Technology

23 février 2016

Objectif

Introduction à la vérification de programmes avec l'assistant de preuve Coq

1. On présente le langage de programmation de Coq. Les types permettent d'imposer des contraintes sur les entrées et les sorties sous la forme de prédicats logiques. Cela permet des spécifications précises et *in fine* de générer des programmes OCaml vérifiés.
2. Quand la spécification est complexe, on peut programmer de manière indirecte en utilisant les tactiques. Cela revient à prouver des formules logiques à propos de programmes fonctionnels.
3. On peut aussi représenter en Coq la syntaxe abstraite d'un langage arbitraire comme une structure de données et sa sémantique comme une prédicat logique. On peut donc utiliser les tactiques pour raisonner en particuler sur du code impératif. On illustre cet aspect avec une logique de Hoare minimale.

Plan

La fonction prédécesseur

- Les entiers naturels en Coq sont définis comme un type inductif (on en verra d'autres exemples) :

```
Inductive nat : Set :=
  O : nat
| S : nat -> nat
```

- La fonction prédecesseur en Coq :

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

- La fonction extraite en OCaml :

```
let prec = function
| O -> O
| S m -> m
```

```
type nat =
| O
| S of nat
```

- C'est une fonction « totale », en particulier `prec O` vaut 0...

Vers une fonction prédécesseur partielle

On restreint l'usage de la fonction au cas où son entrée est strictement positive en ajoutant un nouvel argument qui en est la preuve.

- `pprec` retourne une fonction qui retourne un `nat` si on lui passe une preuve que `0 < n`, où `n` est l'entrée (on verra la définition de `<` au slide 10) :

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

- Que retourner dans le cas où `n` est `0` ?

Comment utiliser une preuve de faux ?

- Le faux est défini comme un type dont on ne peut pas construire les objets (i.e., pas de constructeurs) (on revient sur les prédicats inductifs au slide 8) :

```
Inductive False : Prop := .
```

- À partir d'une preuve de faux, on peut (en particulier) construire n'importe quel entier naturel avec la fonction suivante :

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

- Or, on trouve le théorème suivant dans la librairie de Coq (on ignore la preuve pour l'instant) :

```
Nat.lt_irrefl : forall x : nat, x < x -> False
```

En fait, à partir d'une preuve de `0 < 0`, on peut construire n'importe quoi (*ex falso quodlibet*).

Une fonction prédécesseur partielle

En utilisant `false_nat` [1] vue au slide 6 :

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

(Les arguments inférables automatiquement peuvent être remplacés par _.)
La fonction OCaml extraite :

```
let pprec = function
| O -> assert false (* absurd case *)
| S m -> m
```

1. version générique : `False_rec`

Exemple de prédicat inductif

- Dans la librairie standard de Coq, $\leq$ est défini quand un prédicat inductif :

$$\frac{}{n \leq n}\ \text{le\_n} \qquad \frac{n \leq m}{n \leq m+1}\ \text{le\_S}$$

- Qu'on écrit comme suit en Coq :

```
Inductive le (n : nat) : nat -> Prop :=
  le_n : n <= n
| le_S : forall m : nat, n <= m -> n <= S m
```

  - `n <= m` est une notation pour `le n m`
  - Parmi les arguments du type, on distingue les *paramètres* des *indexes*

- Un preuve est une structure de données (presque) comme les autres

Prouver une inégalité

- Exemples de preuves d'inégalités :
  - `le_n 1` est une preuve de $1 \leq 1$,
  - `le_S _ _ (le_n 1)`, de $1 \leq 2$,
  - `le_S _ _ (le_S _ _ (le_n 1))`, de $1 \leq 3$, etc.
- En fait, la fonction [2] suivante construit des preuves de `1 <= S n`, comme son type l'indique :

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

2. On utilise `Fixpoint` au lieu de `Definition` pour les fonctions récursives.

Utiliser la fonction prédecesseur partielle

- `a < b` est défini comme `a + 1 <= b`
- On peut donc utiliser `spos` (slide 8) pour construire des preuves de `0 < S n` et évaluer la fonction prédécesseur partielle comme suit :

```
> Compute pprec 5 (spos _).
  = 4
  : nat
```

La définition de l'égalité

- Dans Coq, l'égalité n'est pas dans le langage mais est définie :

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  eq_refl : x = x
```

  - `x = y` est une notation pour `eq x y`
  - L'argument `A : Type` est *implicite* : il est inféré automatiquement sans même avoir besoin de l'indiquer par un `_`

- Exemples de preuves :
  - `eq_refl 0` est une preuve de $0 = 0$,
  - `eq_refl true`, de $true = true$, etc.
- On peut écrire `0 = 1` mais pas le prouver
- `eq_refl 4` est une preuve de $4 = 4$ mais aussi de $2 + 2 = 4$, on peut donc effectuer des calculs (β-réduction) dans les types
  - les calculs ne donnent pas lieu à des termes de preuve (principe de Poincaré)
  - c'est la « réflexion »

Une fonction prédécesseur partielle avec inégalité booléenne

- On trouve dans la librairie de Coq une fonction qui décide l'inégalité en renvoyant `true` ou `false` :

```
Nat.ltb : nat -> nat -> bool
```

- On réécrit la fonction prédécesseur avec les équivalents booléens :

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

(`Nat.ltb_lt` est une preuve qui fait la conversion entre `ltb` et `lt`)

- La preuve d'inégalité devient un terme de preuve minimal :

```
Compute pprecb 5 eq_refl.
```

Plan

La fonction prédécesseur complètement spécifiée

- On peut partir d'un type encore plus précis :

```
Definition pprec_interactif (n : nat) :
  0 < n -> {m | n = S m}.
```

- où le type de retour est une preuve d'existence :

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> {x : A | P x}
```

  - Par exemple, `exist (fun x =>x =0) 0 eq_refl` est une preuve qu'il existe un entier naturel égal à 0

- Après une déclaration de type sans terme, on tombe dans le mode interactif.
- Écrire directement ce genre de fonction requiert des astuces (typiquement, transport des preuves d'égalités). L'extension `Program` [CDT16, Chapitre 24] permet de fournir des squelettes de fonctions et de fournir les preuves après coup. On peut aussi programmer la fonction indirectement en utilisant les *tactiques* de Coq.

Construction interactive

tactique `destruct n as [|m]`

équivalent à insérer un terme de la forme
`match n with 0 =>... |S m =>... end`

Entrée (mode interactif)

```
Definition pprec_interactif
  (n : nat) : 0 < n -> {m | n = S m}.

destruct n as [|m.]
```

Fonction construite (cachée)

```
fun n : nat => ?

fun n : nat =>
 match n as n0 return
 (0 < n0 -> {m : nat | n0 = S m})
 with
 | 0 => ?0
 | S m => ?1
 end
```

(On a créé deux sous-buts ?0 et ?1 qu'on va traiter séparément.)

Construction interactive

tactique `intros x`

équivaut à `fun x => ...`

tactique `generalize t`

introduit un terme-argument `t`

Entrée (mode interactif)	Fonction construite (cachée)
(Rappel du but courant : `0 < 0 -> {m : nat | 0 = S m}`)	
`- intros abs.`	`fun n : nat => match n as ... with | 0 => fun abs : 0 < 0 => ?0 | S m => ?1 end`
`generalize (Nat.lt_irrefl _ abs).`	`fun n : nat => match n as ... with | 0 => fun abs : 0 < 0 => ?0 (Nat.lt_irrefl 0 abs) | S m => ?1 end`

Construction interactive

La tactique `destruct 1` s'applique à l'hypothèse « top »
La tactique `intros _` crée fonction qui ignore son argument

Entrée (mode interactif) Fonction construite (cachée)

(Rappel du but : `False ->{m : nat |0 = S m}`)

```
destruct 1.
```

```
fun n : nat => match n as
  n0 return (0 < n0 -> {m : nat | n0 = S m}) with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False =>
     match H return m : nat | 0 = S m with end)
     (Nat.lt_irrefl 0 abs)
 | S m => ?1
 end
```

(But suivant : `0 < S m ->{m0 : nat |S m =S m0}`)

```
- intros _.
```

```
fun n : nat => match n as ... with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False => match H return ... with end)
     (Nat.lt_irrefl 0 abs)
 | S m => fun _ : 0 < S m => ?1
 end
```

Construction interactive

tactique `apply f`

application de la fonction `f`

Entrée (mode interactif) — Fonction construite (cachée)

(Rappel du but : `{m0 : nat |S m =S m0}`)

```
apply (exist _ m).
```

```
fun n : nat => match n as ... with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False => match H return ... with end)
     (Nat.lt_irrefl 0 abs)
 | S m => fun _ : 0 < S m =>
   exist (fun m0 : nat => S m = S m0) m ?1
 end
```

```
apply eq_refl.
```

```
fun n : nat =>
 match n as ... with
 | 0 => fun abs : 0 < 0 =>
     (fun H : False => match H return ... with end)
     (Nat.lt_irrefl 0 abs)
 | S m => fun _ : 0 < S m =>
   exist (fun m0 : nat => S m = S m0) m eq_refl
 end
```

Écrire une fonction = prouver un lemme

C'est une illustration de l'isomorphisme de Curry-Howard.

```
Definition pprec_interactif (n : nat) :
  0 < n -> {m | n = S m}.
...
Defined.
```

→ La fonction construite reste visible et donc exécutable.

```
Lemma pprec_interactif (n : nat) :
  0 < n -> {m | n = S m}.
Proof.
...
Qed.
```

→ La fonction construite est cachée, ce qui rend possible l'identification de deux preuves du même lemme.
→ on lit le symbole -> comme une implication logique

On peut toujours extraire un programme OCaml à partir d'une preuve *constructive* (qui n'utilise pas le principe du tiers-exclus)

Les tactiques de Coq

- Il y en a beaucoup (avec de nombreuses variations) [CDT16, Chapitre 8] mais peu sont essentielles
  - Index en ligne : `https://coq.inria.fr/refman/tactic-index.html`
- Autres tactiques importantes :
  - `induction` : destruction d'un type inductif récursif (correspond au raisonnement par induction)
  - `rewrite ->`/`rewrite <-` : réécriture (en fait une application de la tactique `apply`)
  - `simpl` : β-réduction
  - `unfold` : développement des `Definition`
- Quelques tactiques automatiques :
  - `auto` : quand c'est vraiment évident
  - `tauto` : procédure de décision pour la logique propositionnelle intuitioniste
  - `omega` : procédure de décision pour l'arithmétique de Presburger

Plan

Rappel sur la logique de Hoare

- On écrit $\{P\}c\{Q\}$ pour signifier que l'exécution du programme c depuis un état satisfaisant P mène si elle termine à un état satisfaisant Q.
  - P et Q peuvent être vues comme des fonctions booléennes sur l'état du programme
- La logique de Hoare prend la forme d'un jeu de règles d'inférence entre triplets de Hoare
  - avec environ une règle par construction syntaxique, voir slide 23
- On peut voir la logique de Hoare comme une sémantique
  - souvent on cherche à prouver une équivalence avec une sémantique opérationnelle (*soundness* et *relative completeness*)

Les règles de la logique de Hoare

- Un jeu de règles minimal :

$$\frac{}{\{Q\{e/v\}\}\, v \leftarrow e\, \{Q\}}\ \text{assign}$$

$$\frac{\{P\}\, c\, \{Q\} \quad \{Q\}\, d\, \{R\}}{\{P\}\, c; d\, \{R\}}\ \text{seq}$$

$$\frac{P \to P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \to Q}{\{P\}\, c\, \{Q\}}\ \text{conseq}$$

$$\frac{\{P \wedge t = \mathit{true}\}\, c\, \{P\}}{\{P\}\, \texttt{while}(t)\{c\}\, \{P \wedge t = \mathit{false}\}}\ \text{while}$$

  Regarder un exemple pour se convaincre de la règle assign ; la condition de la règle while s'appelle *invariant*

- À partir de pré/post-conditions et d'invariants de boucle, il est théoriquement possible d'automatiser la vérification
- En pratique, on a souvent recours à la preuve interactive, en particulier avec un assistant de preuve (exemple d'application réaliste : [WKS+09])

Un exemple de preuve en logique de Hoare

- On montre que le programme $\mathit{while}(x \neq 0)\{\mathit{ret} = \mathit{ret} * x; x = x - 1\}$ calcule $x!$ de la manière suivante :

$$\dfrac{\dfrac{\dfrac{}{\{Q\{\mathit{ret} * x/\mathit{ret}\}\}\mathit{ret} = \mathit{ret} * x\{Q\}}\,\text{assign}}{\{\mathit{ret} * x! = X! \wedge x \neq 0\}\ \mathit{ret} = \mathit{ret} * x\ \{\underbrace{\mathit{ret} * (x-1)! = X! \wedge 0 \leq x - 1}_{Q}\}}\,\text{stren} \quad \dfrac{\dfrac{}{\{Q\{x-1/x\}\}x = x - 1\{Q\}}\,\text{assign}}{\{\mathit{ret} * (x-1)! = X! \wedge 0 \leq x - 1\}\ x = x - 1\ \{\underbrace{\mathit{ret} * x! = X!}_{Q}\}}\,\text{stren}}{\dfrac{\{\mathit{ret} * x! = X! \wedge x \neq 0\}\mathit{ret} = \mathit{ret} * x; x = x - 1\{\mathit{ret} * x! = X!\}}{\dfrac{\{\mathit{ret} * x! = X!\}\mathit{while}(x \neq 0)\{\mathit{ret} = \mathit{ret} * x; x = x - 1\}\{\mathit{ret} * x! = X! \wedge x = 0\}}{\{x = X \wedge \mathit{ret} = 1\}\mathit{while}(x \neq 0)\{\mathit{ret} = \mathit{ret} * x; x = x - 1\}\{\mathit{ret} = X!\}}\,\text{conseq}}\,\text{while}}\,\text{seq}$$

- On va maintenant formaliser en Coq l'infrastructure nécessaire pour faire ce genre de raisonnements

Un langage d'expressions arithmétiques et booléennes

- On représente les variables par des entiers naturels :

```
Definition var := nat.
```

- Une expression arithmétique est une variable, un entier natural, une multiplication, une soustraction, etc. :

```
Inductive exp :=
| exp_var : var -> exp
| cst : nat -> exp
| mul : exp -> exp -> exp
| sub : exp -> exp -> exp.
```

Par exemple, on écrit $ret * x$ comme suit :
`mul (exp_var ret) (exp_var x).`

- Une expression booléenne est une égalité, une négation, etc. :

```
Inductive bexp :=
| equa : exp -> exp -> bexp
| neg : bexp -> bexp.
```

Un langage impératif minimal

- Un programme n'est fait que d'affectation de variables, de sequences, et de boucles while :

```
Inductive cmd : Type :=
| assign : var -> exp -> cmd
| seq : cmd -> cmd -> cmd
| while : bexp -> cmd -> cmd.
```

- Par exemple, le programme

$$while(x \neq 0)\{ret = ret * x; x = x - 1\}$$

s'écrit :

```
while (neg (equa (exp_var x) (cst 0)))
  (seq
     (assign ret (mul (exp_var ret) (exp_var x)))
     (assign x (sub (exp_var x) (cst 1)))).
```

- On s'en tient ici à la syntaxe abstraite mais en pratique on rend cela plus lisible en utilisant `Notation`, `Coercion`, etc.

Sémantique des expressions (1/3)

- On représente un état comme une fonction des variables vers les entiers naturels :

```
Definition state := var -> nat.
```

- L'affectation d'une variable correspond à l'ajout d'un cas à la définition de la fonction état :

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

où `Nat.eq_dec` est une preuve que l'égalité sur les entiers naturels est décidable :

```
Nat.eq_dec
     : forall n m : nat, {n = m} + {n <> m}
```

(lire `{...} + {...}` comme une disjunction)

Sémantique des expressions (2/3)

- Pour évaluer une expression dans un état, il suffit de la parser et d'appeler l'état-function quand on rencontre une variable :

```
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.
```

Sémantique des expressions (3/3)

- Par exemple, soient deux variables *ret* et x :

```
Definition ret : var := 0.
Definition x : var := 1.
```

- supposons un état où *ret* vaut 4 et x vaut 5 :

```
Definition sample_state : state :=
  fun x =>
    match x with
    | 0 => 4
    | 1 => 5
    | _ => 0
    end.
```

- dans cet état l'expression $ret * x$ évalue en 20 :

```
> Compute eval (mul (exp_var ret) (exp_var x)) sample_state.
  = 20 : nat
```

Représentation des pré/post-conditions

- On définit les pré/post-conditions comme des fonctions sur les états (*shallow encoding*) :

```
Definition assert := state -> Prop.
```

Par exemple, $ret = 1$ devient :

```
fun s => eval (exp_var ret) s = 1
```

On réalise qu'il y a beaucoup d'implicite dans la notation papier...

- On doit aussi « lifter » les connectives de logiques pour pouvoir parler, e.g., d'implication entre assertions :

```
Definition imp (P Q : assert) :=
  forall s, P s -> Q s.
```

Représentation de la logique de Hoare (1/2)

Les triplets de Hoare sont définis par un prédicat inductif entre les pré/post-conditions et la syntaxe des programmes :

```
Inductive hoare : assert -> cmd -> assert -> Prop :=
```

```
| hoare_assign : forall Q v e,
  hoare
   (fun s => Q (upd v (eval e s) s))
   (assign v e)
   Q
```

$$\frac{}{\{Q\{e/v\}\}v \leftarrow e\{Q\}}$$

```
| hoare_seq : forall Q P R c d,
  hoare P c Q ->
  hoare Q d R ->
  hoare P (seq c d) R
```

$$\frac{\{P\}c\{Q\} \quad \{Q\}d\{R\}}{\{P\}c; d\{R\}}$$

Représentation de la logique de Hoare (2/2)

```
| hoare_conseq : forall P' Q' P Q c,
  imp P P' -> imp Q' Q ->
  hoare P' c Q' ->
  hoare P c Q
```

$$\frac{P \to P' \quad \{P'\}c\{Q'\} \quad Q' \to Q}{\{P\}c\{Q\}}$$

```
| hoare_while : forall 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)).
```

$$\frac{\{\lambda s.P\,s \wedge \mathtt{eval}(b,s)\}\ c\ \{P\}}{\{P\}\ \mathtt{while}(b)\{c\}\ \{\lambda s.P\,s \wedge \neg\mathtt{eval}(b,s)\}}$$

(/\, ~, etc. sont des notations pour la logique propositionnelle dans Coq)

Application : factoriel

- Notation papier :

$$\{x = X \wedge ret = 1\}$$
$$while(x \neq 0)\{ret = ret * x; x = x - 1\}$$
$$\{ret = X!\}$$

- En Coq (on a vu le programme slide 26) :

```
Lemma facto_fact x X ret : x <> ret ->
  hoare
    (fun s => eval (exp_var x) s = X /\
              eval (exp_var ret) s = 1)
    (facto x ret)
    (fun s => eval (exp_var ret) s = fact X).
```

La première étape

- Rappels :

$$\frac{\{\overbrace{ret * x! = X!}^{P'}\}\, while(x \neq 0)\{ret = ret * x; x = x - 1\}\{\overbrace{ret * x! = X! \wedge x = 0}^{Q'}\}}{\{x = X \wedge ret = 1\}\, while(x \neq 0)\{ret = ret * x; x = x - 1\}\{ret = X!\}}$$

$$\frac{P \to P' \quad \{P'\}c\{Q'\} \quad Q' \to Q}{\{P\}c\{Q\}}\ \text{conseq}$$

- En Coq :

```
set (P' := fun s : state => ...).
set (Q' := fun s : state => ...).
apply (hoare_conseq P' Q').
```

ce qui génère trois sous-buts à prouver...

Plan

Utilisation des librairies

- Il y a déjà beaucoup de lemmes dans la librairie standard de Coq : `https://coq.inria.fr/library`
- On peut peupler le contexte avec de nouveaux lemmes et tactiques, par exemple :

```
Require Import Arith.
Require Import Omega.
Require Import Factorial.
```

et chercher parmi les lemmes présents en utilisant des motifs, par exemple :

```
SearchPattern (_ + S _ = _).
SearchRewrite (_ + S _).
```

Autres sujets à propos de la vérification de programmes dans Coq

- Les preuves de terminaison : les fonctions dont la terminaison n'est pas « évidente » (i.e., structurelle) nécessite des techniques avancées (voir [BC04, Chapitre 15])
- La logique de séparation : une extension de la logique de Hoare pour les pointeurs (un exemple d'encodage en Coq : [MAY06])
- On peut étendre la logique de Hoare présentée ici pour représenter des langages réalistes (assembleur : [Aff13], C : [AS14])
- Étendre la logique de Hoare avec des appels de functions (un exemple détaillé en Coq : [Aff15])

Bibliographie


Reynald Affeldt, *On construction of a library of formally verified low-level arithmetic functions*, Innovations in Systems and Software Engineering **9** (2013), no. 2, 59–77.


———, *Proving properties on programs—from the coq tutorial at itp 2015—*, Coq tutorial @ ITP'15 : `https://coq.inria.fr/coq-itp-2015`, Aug. 2015, Available at : `https://coq.inria.fr/files/coq-itp-2015/course-5.pdf`.


Reynald Affeldt and Kazuhiko Sakaguchi, *An intrinsic encoding of a subset of C and its application to TLS network packet processing*, Journal of Formalized Reasoning **7** (2014), no. 1, 63–104.


Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development—Coq'Art : The calculus of inductive constructions*, Springer, 2004.


The Coq Development Team, *The Coq proof assistant reference manual*, INRIA, 2016, Version 8.5.


Nicolas Marti, Reynald Affeldt, and Akinori Yonezawa, *Formal verification of the heap manager of an operating system using separation logic*, Proceedings of the 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1–3, 2006, Lecture Notes in Computer Science, vol. 4260, Springer, 2006, pp. 400–419.


Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish, *Mind the gap*, Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009, Munich, Germany, August 17–20, 2009, Lecture Notes in Computer Science, vol. 5674, Springer, 2009, pp. 500–515.