Les formules logiques en Coq

Yves Bertot

Introduction

- Pour faire des preuves en Coq, il faut énoncer des faits
- Les énoncés sont appelés des propositions
- Les propositions peuvent être construites en connectant des propositions plus petites

Structure des formules logiques

- Formules logiques de base
  - exemples : `True`, `False`, `1 <= 3`, `f 1 = 3`
- Des connecteurs logiques: conjonction, disjonction, implication, négation
  - `1 <= 3 /\ 5 <= 2`, `1 <= 3 \/ 5 <= 2`, `5 < 2 -> 1 < 3`, `~5 < 2`, `1 <> 2`
- Des quantificateurs universels ou existentiels
  - `forall x, 5 <= x -> 12 <= 3 * x`
  - `exists x, 5 * x <= 4 * x`

Détails sur les quantifications

- Les quantifications introduisent une variable liée
- Le nom de la variable peut changer avec toutes les instances de la variable
- La variable appartient toujours à un type que Coq devine
- Si Coq ne peut pas deviner, la formule est mal formée
- `Check forall x, x = x + 1.`
- Le système répond
- `forall x : nat, x = x + 1 : Prop`
- En revanche
- `Check forall x, x = x`
- Le système répond
- `Error: Cannot infer the type of x`

Définir de nouvelles propositions

- Propositions constantes
  - ```
    Definition square_triangle_nat :=
        exists x, exists y, exists z,
        x * x + y * y = z * z.
    ```
- Propositions fonctionnelles : prédicats
  - `Definition even (x : nat) := exists y, x = 2 * y.`
- Le type d'une proposition est `Prop`
- Les propositions fonctionnelles peuvent prendre des propositions en argument
- `Definition not (P : Prop) := P -> False.`
    ```

Implication et conjonction

- De nombreux théorèmes ont plusieurs prémisses
- Si `A` et `B` sont satisfaites alors `C` est satisfaite
- La conjonction des prémisses implique la conclusion
- Usage: ne pas utiliser de connecteur `/\` pour les prémisses mais seulement des implications
- On écrit `A -> B -> C`, la même chose que `A -> (B -> C)`
- Logiquement équivalent à `(A /\ B) -> B` mais plus pratique

Faire des preuves en Coq

Yves Bertot

Introduction

Pour faire une preuve

- On énonce une formule logique
- On décompose la formule en suivant un raisonnement logique
- Certaines formules issues de la décomposition peuvent être éliminées
- Lorsque toutes sont éliminées on déclare la preuve terminée
- Le lemme pourra être réutilisé

Enoncer un nouveau théorème

- La commande `Lemma` *`nom`* `:` *`formule`*`.`
- Le nom doit être nouveau,
- La formule doit être bien formée
- D'autres mots-clefs peuvent être utilisés
  - `Theorem`, `Fact`, `Example`.

Décomposer une formule logique

- Exemple: `A /\ B`
- On veut prouver `A` et `B` en une seule formule
- Mais logiquement, il suffit de prouver `A` puis `B`
- Passer de `A /\ B` à `A` et `B` séparés est fait dans Coq par une commande spécifique
- Le comportement est différent pour les formules en *hypothèse* et les formules dans la *conclusion*

Hypothèses et conclusion

- En cours de preuve, le système Coq affiche des *buts*
- Chaque but contient un contexte et une conclusion
- Le contexte contient des déclaration de variables et des hypothèses
  - Chaque hypothèse a un nom et un énoncé, séparés par le caractère :
- La conclusion du but est séparée des hypothèses par une "barre horizontale"
- Example

```
H1 : x <= y
H2 : y <= z
====================
 x <= z
```

Tactique pour la quantification universelle (conclusion)

- Quantification universelle dans la conclusion: `intros x`
  - Signification intuitive :
    fixons `x` *arbitraire, prouvons la propriété en* `x`
- passage du but

```
======================
 forall x : A, B
```

au but

```
x : A
======================
 B
```

Tactique pour la quantification universelle (en hypothèse)

- Quantification universelle dans une hypothèse `H`: `apply H`
- La conclusion doit être une instance de la formule finale de `H`
- Passe du but

```
H: forall x : A, P x -> Q x
======================
 Q e
```
au but
```
H: forall x : A, P x -> Q x
======================
 P e
```
- Assure que toutes les conditions sont réunies pour appliquer `H`
- Traite toutes les variables quantifiées d'un coup
- Echoue si certaines variables ne peuvent pas être devinées

spécialiser la tactique apply

- Cas d'échec de la tactique `apply`
- ```
  H : forall x y, P x y -> Q x
  ======================
   Q e
  ```
- `apply` ne sait pas déterminer la valeur de `x`
- Aide de l'utilisateur: mot-clef `with`
- `apply H with (y := f).`
- passe au but
  ```
  H : forall x y, P x y -> Q x
  ======================
   P e f
  ```
  ```

Tactiques pour l'implication

Implication dans la conclusion : `intros H`

- Signification intuitive : *supposons la formule de gauche, prouvons la formule de droite*
- passe du but

  ```
  ======================
   P -> Q
  ```

  au but

  ```
  H : P
  ======================
   Q
  ```

- donne le nom `H` à la nouvelle hypothèse

Implication dans une hypothèse : `apply H`

- Cas particulier de la quantification universelle

Preuves avec les connecteurs logiques

Yves Bertot

Tactiques pour la conjonction

- conjonction dans un but : `split`
- Passe du but

  ```
  ======================
   A/\B
  ```

  à deux buts, un pour `A`, l'autre pour `B`
- Conjonction dans une hypothèse : `destruct H as [H1 H2]`

  Passe du but

  ```
  H : A /\ B
  ======================
   C
  ```

  au but

  ```
  H1 : A
  H2 : B
  ======================
   C
  ```
- Fait passer de "et" formel à "et" pratique

Tactiques pour la disjonction

- disjonction dans un but : `left` ou `right`
- Il s'agit de choisir celle des deux formules que l'on veut prouver
- Passe du but

  ```
  ======================
   A \/B
  ```

  au but `A` (pour `left`) ou `B` (pour `right`)
- disjonction dans une hypothèse : `destruct H as [H1 | H2]`
  passe du but

  ```
  H : A \/ B
  ======================
   C
  ```

  à deux nouveaux buts, l'un avec `A` en hypothèse, l'autre avec `B`
- Intuitivement il faut couvrir tous les cas

Tactiques pour la négation

- Négation dans la conclusion : `intros H`
- Passe du but

```
======================
 ~A
```

  au but

```
H : A
======================
 False
```

- Intuitivement, il faut montrer qu'il y aurait contradiction si `A` était satisfait

Tactiques pour la négation (en hypothèse)

- Négation dans une hypothèse : `case H`
- Passe du but

```
H : ~A
============================
  C
```

  au but

```
H : ~A
============================
  A
```

- A n'utiliser que lorsque `~A` est en contradiction avec d'autres hypothèses

Quantification existentielle

- Quantification existentielle dans la conclusion : `exists e`
- Passe du but

```
==========================
 exists x:A, P x
```
au but
```
==========================
 P e
```
- Pour montrer qu'il existe quelqu'un, il faut le désigner et montrer qu'il a la propriété demandée

Quantification existentielle (en hypothèse)

- quantification existentielle dans une hypothèse :
  `destruct H as [x H1]`
- Passe du but

  ```
  H : exists x : A, P x
  =========================
  C
  ```

  au but

  ```
  x : A
  H1 : P x
  =========================
  C
  ```
- Fait disparaitre la quantification existentielle
- Remplace par un contexte qui exprime pratiquement *il existe*

Remarques générales sur les connecteurs logiques

- Chaque tactique fait disparaître la construction logique de la conclusion ou de l'hypothèse
- Remplace par un ou plusieurs buts qui ont un sens équivalent mais plus simple

Finir une preuve

- Un but est entièrement résolu lorsque l'un des deux cas suivants est vérifié:
- la conclusion est parmi les hypothèses
  - La tactique à appliquer est `exact H` ou bien `assumption`
- c'est une instance d'un théorème qui n'a pas de prémisses
  - La tactique à appliquer est `apply t`

Egalités

- Prouver une égalité : `reflexivity`
  - Les deux membres de l'égalité doivent être les mêmes
  - Comparaison modulo les définitions de Coq
  - Fait disparaître le but
- Utiliser une égalité : `rewrite`, `rewrite ->` ou `rewrite <-`
- L'égalité peut être quantifiée universellement
  - Dans ce cas, le système cherche de bonnes valeurs pour les variables quantifiées

Outils avancés pour les preuves

Yves Bertot

Plan de ce cours

- Composer les tactiques
- Faire appel à des tactiques automatiques
- Suivre la structure des formules logiques
- Enoncer des faits intermédiaires

Composer des tactiques

- Tactiques en séquence: notation t_1 ; t_2
  - La tactique t_2 est appliquée à tous les buts créés par t_1
- Tactiques réparties: notation t ;[t_1 | t_2 | $\ldots$]
  - La tactique t_1 est appliquée sur le premier but créé par t
  - t_2 est appliquée sur le deuxième but ...

Essais de tactiques

- Tactiques en alternative: notation `||`
  - La deuxième tactique est appliquée si la première a une erreur
- Alternative avec obligation de conclure:
  notation `solve[ ...| ...]`
  - La deuxième tactique est appliquée si la première n'arrive pas à conclure
  - L'ensemble lève une erreur s'il n'arrive pas à conclure

Exemples de composition de tactiques

```
Lemma or_comm :  forall A B, A \/ B -> B \/ A.
Proof.
intros A B H; destruct H as [a | b];
    solve[left; assumption | right; assumption].
Qed.
```

```
Lemma or_comm :  forall A B, A \/ B -> B \/ A.
Proof.
intros A B H; destruct H as [a | b];
    (left; assumption) || (right; assumption).
Qed.
```

Des tactiques automatiques

- Une tactique `auto` vérifie si la conclusion est une conséquence des implications du contexte
  - Utilise également des bases de données de théorèmes
- Une tactique `intuition` fait la même chose modulo les conjonctions et disjonctions
- Une tactique `firstorder` fait la même chose modulo les quantifications
  - les variables quantifiées ne doivent pas être des fonctions
- D'autres tactiques pour des théories particulières
  - `ring` pour les égalités entre polynômes
  - `omega` pour les inégalités linéaires sur les entiers
  - `fourier` pour les inégalités linéaires sur les nombres réels

Exemples de preuve automatique

```
Lemma ex1 : forall (A : Type) (P Q : A -> Prop),
    (forall x : A, P x /\ Q x) -> forall y, Q y.
Proof. firstorder. Qed.

Require Import ZArith. Open Scope Z_scope.

Lemma ex2 : forall x y : Z,
    0 <= x -> 4 * y <= 3 * x -> 5 * y <= 4 * x.
Proof. intros x y H; omega. Qed.

Lemma ex3 : forall x y : Z,
    (x - y) ^ 2 = x ^ 2 - 2 * x * y + y ^ 2.
Proof. intros x y; ring. Qed.
```

Suivre la structure des formules logiques

- Traiter chaque connecteur logique à l'aide de `destruct` est long
- Il est possible de décomposer directement dans `intros`
  - Traiter les conjonctions et quantifications existentielles comme des paires
  - Traiter les disjonctions comme des alternatives séparées par des barres verticales

Exemple d'introduction avec structure

```
Lemma ex4 : forall P Q R : Z -> Prop,
    (exists x, (P x /\ Q x) \/ R x) ->
    exists x, P x \/ R x.

Proof.
intros P Q R [x [[px qx] | rx]].
```

A cette étape le premier but contient

`x : Z`, `px : P x` et `qx : Q x`

Le deuxième but contient

`x : Z` et `rx : R x`

Enoncer des faits intermédiaires

- La tactique `apply` impose un style *en arrière*
  - On part de la formule finale de l'énoncé, et l'on cherche des théorèmes qui permettent de l'obtenir
- Le raisonnement mathématique usuel suit un style *en avant*
  - On part des hypothèses et on en tire des conséquences de plus en plus fortes
- la tactique `assert` permet d'énoncer des résultats intermédiaires
  - Dans un premier but on doit montrer que le nouvel énoncé est une conséquence des hypothèses jusque là
  - Dans un deuxième but on peut utiliser le résultat intermédiaire comme une nouvelle hypothèse
  - Parfois les résultats intermédiaires sont très généraux et il vaut mieux définir un nouveau lemme

Structuration des preuves

Yves Bertot

Suivre la structure des formules logiques

- Traiter chaque connecteur logique à l'aide de `destruct` est long
- Il est possible de décomposer directement dans `intros`
  - Traiter les conjonctions et quantifications existentielles comme des paires
  - Traiter les disjonctions comme des alternatives séparées par des barres verticales

Exemple d'introduction avec structure

```
Lemma ex4 : forall P Q R : Z -> Prop,
    (exists x, (P x /\ Q x) \/ R x) ->
    exists x, P x \/ R x.

Proof.
intros P Q R [x [[px qx] | rx]].
```

A cette étape le premier but contient

`x : Z`, `px : P x` et `qx : Q x`

Le deuxième but contient

`x : Z` et `rx : R x`

Enoncer des faits intermédiaires

- La tactique `apply` impose un style *en arrière*
  - On part de la formule finale de l'énoncé, et l'on cherche des théorèmes qui permettent de l'obtenir
- Le raisonnement mathématique usuel suit un style *en avant*
  - On part des hypothèses et on en tire des conséquences de plus en plus fortes
- la tactique `assert` permet d'énoncer des résultats intermédiaires
  - Dans un premier but on doit montrer que le nouvel énoncé est une conséquence des hypothèses jusque là
  - Dans un deuxième but on peut utiliser le résultat intermédiaire comme une nouvelle hypothèse
  - Parfois les résultats intermédiaires sont très généraux et il vaut mieux définir un nouveau lemme

Premières preuves de programmes

Yves Bertot

Introduction

Pour faire une preuve de programme

- Le programme est une fonction
- On énonce une formule logique qui contient la fonction
- On raisonne sur les entrées du programme
  - Raisonnements par cas sur la forme des données
  - Raisonnement par récurrence si les données sont récursives

Exemple de raisonnement sur l'addition

- Définition de l'addition

```
Fixpoint plus (x y:nat) : nat :=
match x with
| O => y
| S p => S (plus p y)
end.
```

- Notation : `x + y` pour `plus x y`
- Exemple d'énoncé sur ce programme

```
forall x y z, x + (y + z) = (x + y) + z
```

Raisonnement sur l'addition

- ```
  Lemma plus_assoc :
      forall x y z, x + (y + z) = (x + y) + z.
  Proof.
  induction x as [ | p IHp].
    simpl; intros y z; reflexivity.
  simpl; intros y z; rewrite IHp; reflexivity.
  Qed.
  ```
- La tactique `induction` permet de faire un traitement par cas avec hypothèses de récurrence
- La tactique `simpl` permet de faire avancer les calculs
- L'hypothèse `IHp`, créée par `induction`, est une hypothèse de récurrence.
  ```

Traitement par cas et récurrence

- Les types inductifs décrivent plusieurs cas: les constructeurs
- soit `T` un type inductif avec les constructeurs
  $\mathtt{c}_1 \ldots \mathtt{c}_n$
- si `x` a le type `T` alors `x` est obtenu par $\mathtt{c}_1$ ... ou $\mathtt{c}_n$
- La tactique `case x` construit n buts correspondants à ces cas
- `x` est remplacé par la valeur correspondante dans chaque but
- des quantifications universelles sont ajoutées pour les arguments des constructeurs

Variantes dans les traitements par cas

- La tactique `case x` n'affecte que les occurrences de `x` dans la conclusion
- La tactique `destruct` fait deux choses en plus
  - introduire les variables quantifiées universellement dans le contexte
  - remplacer toutes les instances de `x` également dans le contexte
- La tactique `elim x` fait comme `case` mais ajoute des hypothèses de récurrence
  - une hypothèse de récurrence pour chaque argument de chaque constructeur qui est dans le type inductif de `x`
- La tactique `induction` fait les mêmes choses en plus

Raisonner sur les fonctions

- Raisonner sur une fonction récursive
  - Preuve par récurrence
- Raisonner sur une expression de traitement par cas
  - Utiliser une preuve par cas
- La preuve est une recherche de *bug* symbolique
  - Tous les cas doivent être couverts
  - La récurrence permet de décomposer un domaine infini en parties finies

Raisonner sur les égalités

- De nombreuses propriétés des programmes s'expriment par des égalités
- égalités entre éléments de types inductifs
- Deux propriétés importantes:
  - des constructeurs différents donnent des valeurs différentes
  - chaque constructeur est injectif

Egalités entre constructeurs différents

- Lorsque le contexte contient une égalité `c`$_i$ `... = c`$_j$ ` ...`
- Cette égalité est contradictoire
- Le but peut être résolu en une seule tactique `discriminate`
- S'applique aussi pour les conclusions de la forme `c`$_i$ `... <> c`$_j$ ` ...`

Injectivité des constructeurs

- Lorsque le contexte contient une égalité
  `ci a1... = ci b1 ...` ($\mathtt{c}_i\ \mathtt{a}_1 \ldots = \mathtt{c}_i\ \mathtt{b}_1\ \ldots$)
- On peut déduire $\mathtt{a}_1 = \mathtt{b}_1$, $\mathtt{a}_2 = \mathtt{b}_2 \ldots$
- La tactique s'appelle `injection H`
- Les nouvelles égalités sont ajoutées dans la conclusion

Exemple d'injectivité

- ```
  H : a::l1 = b::l2
  =======================
  C
  ```
- `injection H.`
  Le système répond
  ```
  H : a::l1 = b::l2
  =======================
  l1 = l2 -> a = b -> C
  ```
  ```

Faire calculer

- Pour raisonner il faut voir les résultats des calculs
- Une tactique puissante : `simpl`
  - Bien adaptée pour les fonctions récursives
  - difficile à freiner
- Une tactique plus fine : `change` $\texttt{e}_1$ `with` $\texttt{e}_2$
- Parfois nécessaire d'utiliser des théorèmes annexes
  - Exprimés par des égalités

Egalités

Yves Bertot

Raisonner sur les égalités

- De nombreuses propriétés des programmes s'expriment par des égalités
- égalités entre éléments de types inductifs
- Deux propriétés importantes:
  - des constructeurs différents donnent des valeurs différentes
  - chaque constructeur est injectif

Egalités entre constructeurs différents

- Lorsque le contexte contient une égalité $\texttt{c}_i \texttt{ ... = c}_j \texttt{ ...}$
- Cette égalité est contradictoire
- Le but peut être résolu en une seule tactique `discriminate`
- S'applique aussi pour les conclusions de la forme $\texttt{c}_i \texttt{ ... <> c}_j \texttt{ ...}$

Injectivité des constructeurs

- Lorsque le contexte contient une égalité
  `c`$_i$ `a`$_1$`...` `=` `c`$_i$ `b`$_1$ `...`
- On peut déduire `a`$_1$ `=` `b`$_1$, `a`$_2$ `=` `b`$_2$ `...`
- La tactique s'appelle `injection H`
- Les nouvelles égalités sont ajoutées dans la conclusion

Exemple d'injectivité

- ```
  H : a::l1 = b::l2
  =======================
  C
  ```
- `injection H.`
  Le système répond
  ```
  H : a::l1 = b::l2
  =======================
  l1 = l2 -> a = b -> C
  ```
  ```

Faire calculer

- Pour raisonner il faut voir les résultats des calculs
- Une tactique puissante : `simpl`
  - Bien adaptée pour les fonctions récursives
  - difficile à freiner
- Une tactique plus fine : `change` $\mathtt{e}_1$ `with` $\mathtt{e}_2$
- Parfois nécessaire d'utiliser des théorèmes annexes
  - Exprimés par des égalités