Preuves assistées par ordinateur – Projet

Cohérence de l'arithmétique de Heyting

Présentation

Le but de ce projet est de prouver en Coq la **cohérence de l'arithmétique de Heyting du premier ordre** (notée ici HA_1). On rappelle que l'arithmétique de Heyting est la théorie logique du premier ordre intuitionniste constituée par le langage et les axiomes de Peano.

Modalités

Ce projet doit être effectué *individuellement*.

Vous rendrez par e-mail un fichier `FirstOrderHeyting.v` contenant tout le développement. Il devra être basé sur le fichier modèle `FirstOrderHeytingTemplate.v` fourni en annexe.

Il n'est pas explicitement demandé (mais pas interdit non plus) de fournir un rapport séparé des sources. En revanche, vous ferez attention à rendre un code **lisible** (dont les lignes ne dépassent pas 80 caractères). Un nommage explicite des noms utilisés explicitement dans les scripts de preuve sera également apprécié.

Le développement ne devra utiliser que les bibliothèques fournies en standard avec Coq, et aucun axiome (y compris ceux fournis en standard avec Coq... donc pas de logique classique!). Vous pouvez utiliser n'importe quelle version publiée de Coq ; les instructions données dans ce sujet et le fichier modèle fonctionnent avec les versions 8.3 et 8.4. Merci de préciser au début de votre projet la version utilisée.

Il est recommandé de lire tout le sujet avant de commencer. À titre d'information, tout le sujet peut se faire en moins de 1000 lignes (le fichier modèle en fait déjà environ 500).

1 Conseils et notions utiles

Afin d'utiliser au mieux les différentes fonctionnalités du système (commandes, tactiques, notations, bibliothèque standard, etc.), il est fortement recommandé de consulter attentivement la documentation que l'on peut trouver en ligne à l'URL :

`http://coq.inria.fr/`

1.1 Arithmétique

Décidabilité

On rappelle qu'une proposition P est *décidable* si $P \vee \neg P$ est prouvable. En Coq, cela est équivalent à l'existence d'un algorithme renvoyant un booléen indiquant si la proposition est vraie. Bien entendu, $P \vee \neg P$ peut s'exprimer par `P \/ ~P` (type `or`, lui-même de type `Prop`) en Coq, mais il existe également la notation `{P} + {~P}` (type `sumbool`, lui-même de type `Type`).

Tous les opérateurs arithmétiques élémentaires de comparaison sont décidables. On va ici utiliser en particulier les éléments suivants de la bibliothèque standard de Coq :

```
le_lt_dec : forall n m : nat, {n <= m} + {m < n}
nat_compare : nat -> nat -> comparison
nat_compare_spec : forall x y, CompSpec eq lt x y (nat_compare x y)
```

Le type de `le_lt_dec` permet de l'utiliser directement pour construire un terme informatif à l'aide de la construction `if ... then ... else`. Dans un but faisant intervenir un sous-terme de la forme `if le_lt_dec x y then ... else ...`, une analyse de cas peut être effectuée en utilisant `destruct (le_lt_dec x y)`. Cela génère deux sous-buts : le premier correspond au cas où `x <= y`, et le deuxième au cas où `y < x`.

La fonction `nat_compare` est un peu plus générique, et renvoie un objet de type `comparison` différent dans les trois cas possibles. Dans un but faisant intervenir un sous-terme de la forme `match nat_compare x y with ...`, une analyse de cas peut être effectuée en utilisant la tactique `destruct (nat_compare_spec x y)`. Cela génère trois sous-buts, correspondant aux trois cas possibles : `x = y`, `x < y` et `x > y`. Il est également possible d'utiliser `case_eq (nat_compare x y)`, puis d'utiliser les lemmes spécialisés à chaque cas (qui peuvent être trouvés avec la commande `SearchAbout`). Un `case nat_compare` peut aussi s'avérer commode.

Automatisation

Certains fragments de l'arithmétique sont décidables, et des tactiques permettent de les résoudre. Les preuves de ce projet font appel à l'un d'entre eux : l'arithmétique de Presburger.

L'*arithmétique de Presburger* correspond grossièrement à l'arithmétique de Peano du premier ordre ne faisant intervenir que l'addition. En Coq, la tactique `omega` permet d'en résoudre une bonne partie. En particulier, tous les buts ne faisant intervenir que des additions et des comparaisons peuvent être prouvés directement. Cette tactique a par exemple conscience de l'associativité et de la commutativité de l'addition, de la transitivité des inégalités et des compatibilités existant entre addition et inégalités.

1.2 (Autres) tactiques pratiques

Pour plus de détails sur une tactique, consultez le manuel de référence de Coq.

- Dans le cadre de ce projet, l'usage des tactiques automatiques telles que `auto` ou `intuition` est autorisé.
- La tactique `f_equal` peut aider à prouver une égalité de la forme `f x ... = f x' ...` en vous proposant de prouver les égalités entre arguments : `x=x'`, etc.
- La tactique `simpl` se contente de déplier les définitions et de calculer. Il peut arriver que l'on veuille transformer un sous-terme en un autre convertible (i.e. définitionnellement égal). C'est possible avec la tactique `change`.
- La tactique `replace` permet de remplacer un sous-terme par un autre, s'ils sont prouvablement égaux.
- Les tactiques `set` et `pose` permettent de nommer des sous-termes (i.e. d'en faire des définitions locales à une preuve). Elles peuvent servir à clarifier l'affichage d'un but, ou de simplifier la saisie des arguments de certaines tactiques.
- Les lemmes parlant d'équivalence logique sont directement utilisables de façon confortable avec la tactique `apply`. Par exemple, si vous disposez d'un lemme `truc:A<->B`, alors `apply <- truc` ou `apply -> truc` applique (au sens de `apply`) un sens de l'équivalence. Il est également possible d'utiliser directement la tactique `rewrite` (la même que pour l'égalité) après avoir chargé la bibliothèque `Setoid`.
- Sachez enfin qu'il existe une tactique `admit` permettant d'admettre un sous-but sans le prouver. Évidemment, l'usage de cette tactique n'est pas recommandé, mais peut être commode pour se focaliser d'abord sur certaines portions d'une preuve. Il va de soit que la notation finale dépendra du nombre et de l'importance des `Axiom` (et `Parameter`), `admit` et `Admitted` restant dans le rendu.
- La commande `Print Assumptions` permet d'identifier tous les axiomes utilisés dans la preuve d'une proposition.

1.3 Méthodologie

Un fichier modèle à compléter est fourni avec ce sujet. Des balises `(* TODO *)` y marquent les emplacement à compléter. La suite du sujet suit la structure du fichier modèle.

Toutes les définitions et lemmes attendus dans le rendu sont explicitement demandés et nommés dans ce sujet dans les questions numérotées. Il n'est pas nécessaire de tout prouver dans l'ordre : un lemme peut être prouvé afin d'en prouver un autre (ultérieur). Cependant, l'ordre des énoncés doit être respecté.

Il est bien entendu possible d'introduire des lemmes intermédiaires autres que ceux énoncés dans ce sujet (mais au moins ceux du sujet doivent être prouvés).

2 Syntaxe

Dans cette section, on définit le *langage objet* de notre étude, c'est-à-dire les notions de *terme*, de *formule* et de *dérivation* (et donc de théorème) de HA_1. Ces définitions sont exprimées en Coq, qui sert de *langage méta* et dans lequel sont exprimées (et prouvées) les propriétés du système logique étudié.

2.1 Termes et formules

Par souci de simplificité, on se restreint ici au langage de Peano, bien qu'une bonne partie de tout ce qui est fait ici pourrait être plus général, comme vu en cours.

Les termes du premier ordre font intervenir des variables libres, et des variables liées par des quantificateurs. On a vu en cours que la définition de la substitution est délicate à cause du phénomène de capture de variable, et la notion d'alpha-équivalence a été introduite pour contourner le problème. Cependant, la définition de la substitution en représentant les variables par des noms et en manipulant explicitement l'alpha-équivalence, telle que vue en cours, est difficile à écrire en Coq, et peu pratique à utiliser. C'est pourquoi on va adopter ici une autre représentation des termes.

Les variables liées sont représentées avec des *indices de De Bruijn* : une variable est codée par un nombre représentant le nombre de quantificateurs qu'il faut traverser en remontant dans l'arbre de syntaxe abstrait de la formule. Les variables libres sont les variables « qui remontent trop haut ». Le type des termes est :

```
Inductive term :=
  | Tvar : nat -> term
  | Tzero : term
  | Tsucc : term -> term
  | Tplus : term -> term -> term
  | Tmult : term -> term -> term.
```

et celui des formules est :

```
Inductive formula :=
  | Fequal : term -> term -> formula
  | Ffalse : formula
  | Fand : formula -> formula -> formula
  | For : formula -> formula -> formula
  | Fimplies : formula -> formula -> formula
  | Fexists : formula -> formula
  | Fforall : formula -> formula.
```

Ainsi, avec cet encodage, la formule :

$$\forall n\ \exists p\ n = 1 * p$$

est représentée par :

```
Fforall (Fexists (Fequal (Tvar 1) (Tmult (Tsucc Tzero) (Tvar 0))))
```

Cette représentation a l'avantage que deux formules alpha-équivalentes sont syntaxiquement égales. Cependant, le phénomène de capture de variable peut toujours se produire, et des précautions sont à prendre. Heureusement, ces précautions sont assez systématiques et sont prises en utilisant les fonctions `tlift` et `flift`, ainsi qu'une série de lemmes associés, donnés (et prouvés) dans le fichier modèle. Regardez attentivement ces résultats ; vous devrez les utiliser par la suite.

2.2 Expressions closes

En représentation avec indices de De Bruijn, la notion d'ensemble de variables libres vue en cours est remplacée par une borne sur toutes les variables libres apparaissant dans l'expression. Le prédicat `cterm n t` traduit le fait que toutes les variables libres de `t` sont strictement inférieures à `n`. Attention, les quantificateurs sont pris en compte : lors du passage sous un quantificateur, la borne est augmentée de un. Les prédicats `cterm` et `cformula` sont définis inductivement dans le fichier modèle.

Questions

1. Prouvez les propriétés suivantes :

```
Lemma cterm_1 : forall n t, cterm n t ->
  forall n', n <= n' -> cterm n' t.

Lemma cterm_2 : forall n t, cterm n t ->
  forall k, tlift k t n = t.

Lemma cterm_3 : forall n t, cterm n t ->
  forall t' j, n <= j -> tsubst j t' t = t.

Lemma cterm_4 : forall n t, cterm (S n) t ->
  forall t', cterm 0 t' -> cterm n (tsubst n t' t).
```

2. Prouvez les propriétés analogues pour les formules.

2.3 Déduction naturelle

Une dérivation en déduction naturelle du jugement `e` $\vdash$ `A` est représentée en Coq par `ND e A`. Le prédicat `ND` (fourni dans le fichier modèle) est défini inductivement avec les règles de déduction naturelle vues en cours, adaptées à l'utilisation d'indices de De Bruijn.

Questions

1. On a omis du langage et des règles de dérivation les connecteurs $\top$ (`Ftrue`), la négation (`Fnot`) et l'équivalence (`Fequiv`). Définissez-les, et prouvez que les règles d'introduction et d'élimination associées sont admissibles.
2. Définissez un opérateur `nFforall`, itération du connecteur `Fforall`. À titre d'exemple, la formule `nFforall 2 A` devra être convertible à `Fforall (Fforall A)`. Prouvez la propriété suivante :

```
Lemma nFforall_1 : forall n x t A,
  fsubst x t (nFforall n A) = nFforall n (fsubst (n + x) t A).
```

2.4 Notations

Afin de rendre les formules plus lisibles, des notations sont introduites dans le fichier modèle. Ces notations, qui redéfinissent beaucoup de notations existant en standard dans Coq, sont mises dans un *scope* différent. Ainsi, les notations déjà existantes ont les interprétations habituelles (au niveau du langage méta), mais les interprétations introduites ici (au niveau du langage objet) peuvent être utilisées avec la notation `(...)%pa` : par exemple, `A \/ B` représente la conjonction méta, et `(A \/ B)%pa` représente la conjonction objet (i.e. la formule `Fand A B`).

2.5 Théorie

Les axiomes de Peano sont représentés dans le fichier modèle par le prédicat `Ax` : la proposition `Ax P` traduit le fait que `P` est un axiome de Peano. Regardez attentivement comment `Ax` est définie. Cette définition permet de définir en Coq la notion de *théorème* :

```
Definition Th T := exists axioms,
  (forall A, In A axioms -> Ax A) /\ ND axioms T.
```

Questions

1. Prouvez dans HA_1 la formule suivante : $\forall n\ n \neq s(n)$. On demande ici une preuve au niveau objet : il faut exprimer cette formule comme un terme Coq `A` de type `formula`, puis prouver que `Th A`.

3 Sémantique

La preuve de cohérence de HA_1 proposée ici consiste en la construction d'un modèle, ayant le type `nat` de Coq comme support.

3.1 Interprétation

Une valuation des variables est représentée par une liste `b` de valeurs (qui sont ici des `nat`) : la variable numéro `i` est ainsi interprétée par l'élément numéro `i` de la liste (noté `nth i b 0`). Le fichier modèle définit les fonctions suivantes :

```
tinterp : list nat -> term -> nat
finterp : list nat -> formula -> Prop
```

qui interprètent les termes et les formules du langage objet et entiers et en propositions de Coq. Une formule objet (de type `formula`) sera dite *valide* si son interprétation est prouvable en Coq.

Questions

1. Prouvez les propriétés suivantes :

```
Lemma tinterp_1 : forall t' t b1 b2,
  tinterp (b1 ++ b2) (tsubst (length b1) t' t) =
  tinterp (b1 ++ (tinterp b2 t') :: b2) t.

Lemma tinterp_2 : forall t j, cterm j t ->
  forall b1 b2, j <= length b1 -> j <= length b2 ->
  (forall i, i < j -> nth i b1 0 = nth i b2 0) ->
  tinterp b1 t = tinterp b2 t.

Lemma tinterp_3 : forall t b0 b1 b2,
  tinterp (b0++b2) t =
  tinterp (b0++b1++b2) (tlift (length b1) t (length b0)).
```

2. Prouvez les propriétés analogues pour les formules.

3.2 Correction du modèle

Questions

1. Prouvez que les règles de déduction naturelles sont correctes vis-à-vis de l'interprétation des formules :

   ```
   Lemma ND_soundness : forall e A, ND e A ->
     forall b, (forall B, In B e -> finterp b B) -> finterp b A.
   ```

2. Prouvez que tous les axiomes de Peano sont valides :

   ```
   Lemma Ax_soundness : forall A, Ax A -> forall b, finterp b A.
   ```

3. En déduire que `Ffalse` n'est pas un théorème :

   ```
   Theorem coherence : ~Th Ffalse.
   ```