

Formalisation et étude de l'arithmétique de Peano en Coq

Samuel Mimram

25 janvier 2005

1 Formalisation de l'arithmétique de Peano en Coq

1.1 Objets de preuve

Nous avons choisi de formaliser l'arithmétique de Peano de la façon proposée par le sujet. Les objets des formules de Peano sont définis inductivement de façon naturelle :

```
Inductive Pobj : Set :=
  | zero                               (* 0 *)
  | succ : Pobj -> Pobj                (* S n *)
  | add  : Pobj -> Pobj -> Pobj        (* n + m *)
  | mult : Pobj -> Pobj -> Pobj        (* n * m *)
  | var  : nat -> Pobj.                (* variable *)
```

Les variables sont représentées en indices de De Bruijn par un entier qui signifie : « je suis lié au n -ième lieu », les lieux étant les quantificateurs existentiels ou universels et les `intc` des contextes (voire ci-dessous). C'est la gestion de ces variables qui a bien sûr posé le plus grand nombre de difficultés dans les preuves¹.

Pour la forme nous avons aussi défini un type pour les propositions de Peano (ça n'était pas vraiment utile étant donné qu'il n'y a qu'un seul constructeur, celui de l'égalité) :

```
Inductive Pprop : Set :=
  | eqp : Pobj -> Pobj -> Pprop. (* p = q *)
```

Nous avons enfin défini le type inductif qui correspond aux formules du premier ordre dans l'arithmétique de Peano :

```
Inductive Pformula : Set :=
  | pf : Pprop -> Pformula          (* p = q *)
  | botf : Pformula                 (* false *)
```

¹Vive FreshCoq!

```

| andf : Pformula -> Pformula -> Pformula (* P /\ Q *)
| orf  : Pformula -> Pformula -> Pformula (* P \/ Q *)
| impf : Pformula -> Pformula -> Pformula (* P -> Q *)
| existsf : Pformula -> Pformula (* exists x, P(x) *)
| forallf : Pformula -> Pformula. (* forall x, P(x) *)

```

Nous avons suivi la définition qui était suggérée dans l'énoncé pour les contextes

```

Inductive Pctx : Set :=
| nilc
| intc : Pctx -> Pctx
| assume : Pformula -> Pctx -> Pctx.

```

avec la notation suivante pour les `assume`

```

Infix "::::" := assume (at level 61, right associativity).

```

Cette structure des contexte est assez rigide et de nombreux lemmes sont dûs à la présence des `intc` dans les contextes. Les preuves auraient sans doute été simplifiées en utilisant des listes de formules et en supposant que les variables non liées dans une proposition sont virtuellement liées à l'extérieur du contexte. La règle d'introduction du \forall serait alors devenue

$$\frac{\text{lift}(\Gamma) \vdash P}{\Gamma \vdash \forall(P)} (\forall\text{-I})$$

où `lift(Γ)` est le contexte obtenu en appliquant `lift` à toutes les formules de Γ . L'utilisation de telles liste aurait sans doute permis d'éviter certains lemmes techniques et peu intéressants (les manipulations de `intc` s'avèrent relativement délicates pour être correctes). Elle aurait rendu possible le parcours de Γ dans les deux sens (ça n'est possible que « de la gauche vers la droite » avec la définition inductive), simplifiant ainsi quelques définitions. Un tel traitement des contexte aurait cependant sans doute été moins satisfaisant d'un point de vue théorique car les règles n'auraient alors plus été « locales » (on est obligé d'appliquer `lift` à tout le contexte, modifiant ainsi tout le contexte).

1.2 Opérations syntaxiques sur les termes

1.2.1 Lift

Le `lift` est une opération syntaxique qui permet de lier les variables libres d'une formule n variables plus loin. Sa définition se fait inductivement en passant par des fonctions auxiliaires, d'abord sur les objets

```

Fixpoint lift_obj_r (o:Pobj) (n:nat) (k:nat) {struct o} : Pobj :=
match o with
| zero => zero
| succ o => succ (lift_obj_r o n k)

```

```

    | add o1 o2 => add (lift_obj_r o1 n k) (lift_obj_r o2 n k)
    | mult o1 o2 => mult (lift_obj_r o1 n k) (lift_obj_r o2 n k)
    | var i => if lt_ge_dec i k then var i else var (i+n)
end.

```

(par commodité, nous avons introduit le prédicat `lt_ge_dec` de type

```
forall (n m:nat), {n < m} + {n >= m}
```

qui est défini à partir de `lt_eq_lt_dec`) puis est étendu en un morphisme sur les formules

```

Fixpoint lift_r (F:Pformula) (n:nat) (k:nat) {struct F} : Pformula :=
match F with
| pf (eqp o1 o2) => pf (eqp (lift_obj_r o1 n k) (lift_obj_r o2 n k))
| botf => botf
| andf f1 f2 => andf (lift_r f1 n k) (lift_r f2 n k)
| orf f1 f2 => orf (lift_r f1 n k) (lift_r f2 n k)
| impf f1 f2 => impf (lift_r f1 n k) (lift_r f2 n k)
| existsf f => existsf (lift_r f n (S k))
| forallf f => forallf (lift_r f n (S k))
end.

```

Intuitivement la signification de `lift_r P n k` est : lie les variables d'indice supérieur à `k n` lieurs plus loin. L'introduction du paramètre supplémentaire `k` est nécessaire pour ne pas toucher aux variables qui sont liées à une quantification existentielle ou universelle de la formule. Cependant, à part dans les preuves, la fonction `lift` sera utilisée avec `k = 0` :

Definition `lift F n := lift_r F n 0`.

Nous présentons déjà un lemme qui nous sera utile dans les preuves par la suite

Lemma `lift_obj_0_id` : `forall o k, lift_obj_r o 0 k = o`.

Il montre que décaler les variables de 0 lieurs revient à ne rien faire.

1.2.2 Substitution

La substitution est définie sur le même modèle. Elle remplace la première variable libre d'une formule par un terme donné. Là encore il faut passer par des définitions auxiliaires plus générales qui remplacent la n -ième variable libre d'une formule, permettant ainsi de passer sous les quantifications dans les formules.

On les définit d'abord sur les objets

```

Fixpoint subst_obj (o:Pobj) (t:Pobj) (n:nat) {struct o} : Pobj :=
match o with
| zero => zero
| succ o => succ (subst_obj o t n)

```

```

| add o1 o2 => add (subst_obj o1 t n) (subst_obj o2 t n)
| mult o1 o2 => mult (subst_obj o1 t n) (subst_obj o2 t n)
| var k =>
  match lt_eq_lt_dec k n with
  | inleft so =>
    match so with
    | left _ (* k < n *) => var k
    | right _ (* k = n *) => lift_obj t n
    end
  | inright _ (* k > n *) => var k
  end
end.

```

puis sur les formules

```

Fixpoint subst_n (F:Pformula) (t:Pobj) (n:nat) {struct F} : Pformula :=
match F with
| pf (eqp o1 o2) => pf (eqp (subst_obj o1 t n) (subst_obj o2 t n))
| botf => botf
| andf f1 f2 => andf (subst_n f1 t n) (subst_n f2 t n)
| orf f1 f2 => orf (subst_n f1 t n) (subst_n f2 t n)
| impf f1 f2 => impf (subst_n f1 t n) (subst_n f2 t n)
| existsf f => existsf (subst_n f t (S n))
| forallf f => forallf (subst_n f t (S n))
end.

```

et enfin la définition de la substitution

```

Definition subst (F:Pformula) (t:Pobj) := subst_n F t 0.

```

1.2.3 Application

Nous avons défini une opération d'application qui est proche de l'opération de substitution : elle remplace la première variable libre par un terme t et décale d'un rang vers le bas toutes les autres variables libres. Elle ne diffère de la substitution que par sa définition sur les objets

```

Fixpoint app_obj (o:Pobj) (t:Pobj) (n:nat) {struct o} : Pobj :=
match o with
| zero => zero
| succ o => succ (app_obj o t n)
| add o1 o2 => add (app_obj o1 t n) (app_obj o2 t n)
| mult o1 o2 => mult (app_obj o1 t n) (app_obj o2 t n)
| var k =>
  match lt_eq_lt_dec k n with
  | inleft so =>
    match so with
    | left _ (* k < n *) => var k

```

```

      | right _ (* k = n *) => lift_obj t n
    end
  | inright _ (* k > n *) => var (pred k)
end
end.

```

(notez la différence pour les variables dans le cas où $k > n$).

Les deux sont nécessaires. La définition du schéma de récurrence est par exemple

```

forall P,
  Heyting_ax (impf (app P zero)
    (impf (forallf (impf P (subst P (succ (var 0)))) (forallf P))))

```

Elle utilise les deux et permet de bien comprendre la différence. Dans le cas de base (`zero`), on utilise une application car on enlève le `forallf` qui liait cette variable. Dans le cas inductif, on ne fait que remplacer la première variable libre par son successeur mais on ne retire pas de lieu qui est encore nécessaire car le terme substitué utilise la variable qu'il définit. On aurait peut-être pu se passer de `app` en remplaçant le cas de base `app P zero` par

```
forallf (subst P zero)
```

Mais cela aurait été relativement peu naturel de garder une quantification qui ne sert à rien à part à ne pas modifier l'ordre des indices.

À la réflexion, la substitution aurait sans doute pu être définie à partir de l'application par

```
subst P t := lift (app P t) 1
```

ce qui nous aurait épargné quelques développements mais nous ne nous en sommes aperçu qu'une fois ces développements faits.

1.3 Formules dérivables

1.3.1 Système de preuve

Nous avons défini inductivement les règles permettant de dériver des formules valides. C'est formules sont (dans une section) paramétrées par un « ensemble » d'axiomes `axiom : Pformula -> Prop` (il est codé par un prédicat qui dit si une formule est un axiome). Voici la définition des formules dérivables

```

Inductive Valid : Ptxt -> Pformula -> Prop :=
  | IRaxiom : forall G A, axiom A -> Valid G A
  | IRassume : forall G (A:Pformula), Valid (A::G) A
  | IRweak : forall G A B, Valid G A -> Valid (B::G) A
  | IRimpI : forall G A B, Valid (A::G) B -> Valid G (impf A B)
  | IRimpE : forall G A B, Valid G (impf A B) -> Valid G A -> Valid G B
  | IRandI : forall G A B, Valid G A -> Valid G B -> Valid G (andf A B)

```

```

| IRandEL : forall G A B, Valid G (andf A B) -> Valid G A
| IRandER : forall G A B, Valid G (andf A B) -> Valid G B
| IRorIL : forall G A B, Valid G A -> Valid G (orf A B)
| IRorIR : forall G A B, Valid G B -> Valid G (orf A B)
| IRorE : forall G A B C, Valid G (orf A B) -> Valid (A::G) C -> Valid (B::G) C ->
| IRbotE : forall G A, Valid G botf -> Valid G A
| IRforallI : forall G A, Valid (intc G) A -> Valid G (forallf A)
| IRforallE : forall G A t, Valid G (forallf A) -> Valid G (app A t)
| IReexistsI : forall G A t, Valid G (app A t) -> Valid G (existsf A)
| IReexistsE : forall G A B, Valid G (existsf A) -> Valid (intc G) (impf A (lift B 1))

```

Il n'y a là que les règles strictement proposées par le sujet. On pourra constater que la règle d'élimination des coupures est légèrement différente de ce qui était suggéré dans l'énoncé :

```
forall G A B, Valid G (existsf A) -> Valid (A::(intc G)) (lift B 1) -> Valid G B.
```

Cette forme permettra de résoudre un problème pour montrer la dérivabilité intuitionniste de la non-non-traduction d'une formule classique (sinon elle ne passe pas à cause des $\neg^A\neg^A$ qui sont rajouté aux formules à prouver mais pas aux formules du contexte).

Nous avons aussi jugé bon d'autoriser l'introduction d'axiomes dans tous les contextes (et pas seulement le contexte vide). En effet, sinon il faudrait rajouter une règle de la forme

```
IRweak' : forall A, Valid nilc A -> Valid (intc nilc) A
```

pour pouvoir « vider » entièrement les contextes (*i.e.* y compris les `intc`). Une autre règle est nécessaire pour pouvoir vider les contextes :

```
IRstruct : forall G A B, Valid (intc (A::G)) B -> Valid ((lift A 1)::(intc G)) B
```

Elle permet de faire remonter les `intc` vers le début des contextes, permettant ainsi de les vider d'abord par une règle `IRweak`. Cette règle semble de plus indispensable pour montrer des buts de la forme

$$\text{intc}(P) :: \text{nilc} \vdash P$$

Il est enfin une troisième règle qu'il nous semblerait utile d'ajouter

```
IRforallE' : forall G A, Valid G (forallf A) -> Valid (intc G) A
```

Elle permet de montrer des buts de la forme

$$\text{intc}(\text{nilc}) \vdash \text{var}_0 = \text{var}_0$$

Il nous semble en effet qu'il n'y a pas d'autre moyen que d'utiliser une telle règle pour prouver le jugement ci-dessous qui clairement doit se dériver en utilisant l'axiome de réflexivité de l'égalité : $\forall x, x = x$.

1.3.2 Axiomes

Les systèmes de preuve sont paramétrés par des « ensembles » (des types inductifs en réalité) d'axiomes. Voici les axiomes de l'arithmétique de Heyting

```
Inductive Heyting_ax : Pformula -> Prop :=
| AXeqrefl : Heyting_ax (forallf (pf (eqp (var 0) (var 0))))
| AXeqE : forall P, Heyting_ax (forallf (forallf (impf (pf (eqp (var 0) (var 1))) (impf P (subst P (succ (var 0))))))
| AXdiscr : Heyting_ax (forallf (notf (pf (eqp (succ (var 0)) zero))))
| AXnotOS : Heyting_ax (forallf (existsf (impf (notf (pf (eqp (var 1) zero))) (pf (eqp (succ (var 1)) zero))))
| AXinjS : Heyting_ax (forallf (forallf (impf (pf (eqp (succ (var 1)) (succ (var 0)))) (pf (eqp (succ (var 1)) (succ (var 0))))))
| AXadd0 : Heyting_ax (forallf (pf (eqp (add (var 0) zero) (var 0))))
| AXaddS : Heyting_ax (forallf (forallf (pf (eqp (add (succ (var 1)) (var 0)) (succ (add (var 1) (var 0))))))
| AXmult0 : Heyting_ax (forallf (pf (eqp (mult zero (var 0)) zero)))
| AXmultS : Heyting_ax (forallf (forallf (pf (eqp (mult (succ (var 1)) (var 0)) (add (mult (var 1) (var 0)) (succ (mult (var 1) (var 0))))))
| AXrec : forall P, Heyting_ax (impf (app P zero) (impf (forallf (impf P (subst P (succ (var 0))))))
```

et ceux de Peano

```
Inductive Peano_ax : Pformula -> Prop :=
| AXhey : forall F, Heyting_ax F -> Peano_ax F
| AXexcl : forall P, Peano_ax (impf (notf notf P) P).
```

1.4 Réflexion

Nous avons réalisé entièrement les preuves de la première partie du DM sur la réflexion de l'arithmétique de Heyting dans Coq.

1.4.1 Définition de la réflexion

Le premier point délicat lors de la définition de la réflexion a été le choix de la représentation des environnements dans Coq (nous y reviendrons). Nous avons choisi de les représenter par des fonctions de type `nat -> nat` qui à n associent la valeur de la n -ième variable.

Nous avons alors défini l'insertion qui consiste à insérer la valeur d'une variable dans un environnement à la position n (les valeurs des variables à partir de n sont décalées d'un rang vers le haut)

```
Definition insert (vars:nat->nat) (t:nat) (n:nat) (v:nat) : nat :=
match lt_eq_lt_dec v n with
| inleft so =>
  match so with
  | left _ (* v < n *) => vars v
  | right _ (* v = n *) => t
  end
| inright _ (* v > n *) => vars (pred v)
end.
```

Cette opération est la contrepartie sémantique de l'application comme nous le verrons.

Nous avons aussi défini le remplacement qui change la valeur de la n -ième variable dans un environnement

```

Definition replace (vars:nat->nat) (t:nat) (n:nat) (v:nat) : nat :=
match lt_eq_lt_dec v n with
| inleft so =>
  match so with
  | left _ (* v < n *) => vars v
  | right _ (* v = n *) => t
  end
| inright _ (* v > n *) => vars v
end.

```

Cette opération est la contrepartie sémantique de la substitution.

La traduction des propositions syntaxiques dans les types de Coq se fait alors naturellement par induction

```

Fixpoint tr_obj (o:Pobj) (vars:nat -> nat) {struct o} : nat :=
match o with
| zero => 0
| succ o => S (tr_obj o vars)
| add o1 o2 => (tr_obj o1 vars) + (tr_obj o2 vars)
| mult o1 o2 => (tr_obj o1 vars) * (tr_obj o2 vars)
| var n => (vars n)
end.

```

```

Definition tr_prop (p:Pprop) (vars:nat -> nat) : Prop :=
match p with
| eqp o1 o2 => (tr_obj o1 vars) = (tr_obj o2 vars)
end.

```

```

Fixpoint tr_r (F:Pformula) (vars:nat -> nat) {struct F} : Prop :=
match F with
| pf p => tr_prop p vars
| botf => False
| andf f1 f2 => (tr_r f1 vars) /\ (tr_r f2 vars)
| orf f1 f2 => (tr_r f1 vars) \/ (tr_r f2 vars)
| impf f1 f2 => (tr_r f1 vars) -> (tr_r f2 vars)
| existsf f => exists x, tr_r f (insert vars x 0)
| forallf f => forall x, tr_r f (insert vars x 0)
end.

```

On passe à chaque fois un environnement (`vars`) qui est la valeur des variables (c'est l'interprétation courante des variables). Cet environnement n'est modifié que dans le cas où l'on traduit des formules commençant par `existsf` ou `forallf` qui sont les seuls lieux. Ils insèrent une variable fraîche au début de l'environnement.

On vérifie que cette traduction donne bien le résultat escompté sur l'exemple proposé :

```
Definition reflexion_ex :=
  forallf (existsf (orf (pf (eqp (var 1) (succ (var 0)))) (pf (eqp (var 1) zero))))).
```

```
Eval compute in (tr reflexion_ex).
```

donne le résultat

```
= forall x : nat, exists x0 : nat, x = S x0 \ / x = 0 : Prop
```

1.5 Problèmes rencontrés avec l'encodage des environnements

Nous avons choisi de coder les environnements sous formes de fonctions de type `nat -> nat` plutôt que sous formes de listes de valeurs des variables car cette représentation nous semble plus proche de la notion traditionnelle d'interprétation des variables. En effet, avec des listes nous aurions été obligés d'assigner une valeur arbitraire aux accès hors de la liste ce qui serait revenu à donner une valeur unique à toutes les variables hors du support de la liste. De plus, l'utilisation des listes nous aurait sans doute amené à introduire des prédicats pour savoir si pour une formule donnée est close dans une liste (*i.e.* si la liste est assez longue pour effectivement contenir des valeurs pour toutes les variables) ce qui aurait certainement compliqué les preuves.

Le codage des environnements sous forme de fonctions nous a cependant posé des problèmes. Le premier problème rencontré est que deux environnements « moralement » égaux ne sont en réalité qu'extensionnellement égaux en tant que fonctions. Ainsi, si nous les avons stockés sous formes de listes de valeurs des variables, nous aurions eu l'égalité entre les listes `x::vars0 ++ vars'` et `(x::vars0)++vars'` (ce lemme n'est pas un exemple fictif, nous en avons réellement eu besoin); avec des fonctions pour les environnements on n'a plus qu'une égalité extensionnelle entre les environnements et le lemme correspondant est

```
Lemma insert_struct :
forall vars o x n,
  insert (insert vars x 0) (tr_obj o (shift (insert vars x 0) (S n))) (S n) ==
  insert (insert vars (tr_obj o (shift vars n)) n) x 0.
```

où `==` est l'égalité extensionnelle définie par

```
Definition ext_eq A B (f g:A->B) := forall x:A, f x = g x.
```

```
Implicit Arguments ext_eq [A B].
```

```
Notation "f == g" := (ext_eq f g) (at level 20) : type_scope.
```

De plus, si la preuve de $x::\text{vars0} ++ \text{vars}' = (x::\text{vars0})++\text{vars}'$ se fait à l'aide d'un simple `auto`, la preuve du lemme `insert_struct` requiert elle de distinguer de nombreux cas suivant la valeur du numéro de la variable. Une fois tous les `lt_eq_lt_dec` cassés, les buts se résolvent généralement simplement avec `auto with *` ou `omega`. L'écriture d'une tactique pourrait être envisagée pour résoudre automatiquement ce genre de lemmes.

La première réaction constatant cela est de penser rajouter un axiome d'extensionnalité de l'égalité (`forall f g, f == g -> f = g`) mais en réalité cela n'est pas nécessaire. En effet, on peut montrer que les traductions d'un même formule dans deux environnements extensionnellement égaux sont équivalentes :

```
Lemma tr_ext : forall A vars vars', vars' == vars -> (tr_r A vars <-> tr_r A vars').
```

Remarque : les deux traductions ne sont pas égales, à moins d'ajouter les deux axiomes suivants

```
Axiom exists_ext :
```

```
  forall (P Q:nat -> Prop), (forall x, P x = Q x) -> ((exists x, P x) = (exists x, Q x))
```

```
Axiom forall_ext :
```

```
  forall (P Q:nat -> Prop), (forall x, P x = Q x) -> ((forall x, P x) = (forall x, Q x))
```

qui sont des formes affaiblies d'extensionnalité. Ceci est légèrement ennuyeux car on ne peut alors pas utiliser la tactique `rewrite` et l'on a été obligé de faire de nombreux lemmes montrant que l'équivalence passe au contexte (à peu de choses près) :

```
Lemma iff_and_compat :
```

```
  forall (A B C D:Prop), (A <-> C) -> (B <-> D) -> (A /\ B <-> C /\ D).
```

etc. Et d'utiliser ces lemmes explicitement pour « simuler » des réécritures.

1.6 Quelques lemmes de correction

Nous commençons par montrer quelques lemmes de correction. Ils permettent de « transformer » des opérations syntaxiques (sur les formules de Heyting) en opérations sémantiques (sur les types Coq ou sur les environnements). Ils nous seront utiles par la suite pour montrer la prouvabilité en Coq de la traduction d'une formule de l'arithmétique de Heyting.

Nous commençons par définir la fonction `shift_from` qui décale dans un environnement les variables à partir de k de n positions

```
Definition shift_from (vars:nat->nat) n k v :=
```

```
  if lt_ge_dec v k then vars v else vars (v+n).
```

puis nous montrons que cette opération sur les environnements correspond à faire un `lift` sur une formule, après traduction

```
Lemma lift_sound :
```

```
  forall vars P n k, tr_r (lift_r P n k) vars <-> tr_r P (shift_from vars n k).
```

Nous avons de même que la substitution revenait à remplacer une valeur dans un environnement :

```
Lemma subst_sound :
  forall P t n vars, tr_r (subst_n P t n) vars <->
    tr_r P (replace vars (tr_obj t (shift vars n)) n).
```

et que l'application revenait à insérer une valeur dans un environnement :

```
Lemma app_sound :
  forall P t n vars, tr_r (app_n P t n) vars <->
    tr_r P (insert vars (tr_obj t (shift vars n)) n).
```

Les preuves de ces lemmes se font naturellement par induction sur la structure de la formule P . Elles ont nécessité quelques lemmes auxiliaires techniques montrant certaines propriétés de pseudo-commutativité sur les fonctions `insert` et `replace` comme par exemple

```
Lemma struct_thing :
  forall t vars x n,
    ((replace (insert vars x 0) (tr_obj t (shift (insert vars x 0) (S n))) (S n)) ==
     (insert (replace vars (tr_obj t (shift vars n)) n) x 0)).
```

Du lemme `subst_sound`, nous pouvons tirer un lemme de *subject reduction* qui s'énonce ainsi

```
Lemma tr_subj_red :
  forall P t vars, tr_r (subst P t) vars <-> tr_r P (replace vars (tr_obj t vars) 0).
```

Il énonce que substituer dans le concret (les formules de Heyting) puis traduire revient à substituer dans l'abstrait (les types de Coq) sur la traduction.

1.7 Réflexion

Nous pouvons maintenant montrer que les traductions des axiomes de Heyting sont des formules valides de Coq.

```
Lemma Heyting_reflexion : forall P vars, Heyting_ax P -> tr_r P vars.
```

La preuve se fait par induction sur la preuve de `Heyting_ax P`.

L'extension de la traduction aux contextes est légèrement plus compliquée que de « mapper » la fonction de traduction (`tr_r`) à tous les éléments du contexte. Il faut en effet prendre en compte les liaisons induites par les `intc`.

```
Fixpoint tr_ctxt (glob:nat->nat) (G:Ptxt) (A:(nat->nat)->Prop) { struct G } : Prop :=
match G with
| nilc => A glob
| intc G => tr_ctxt glob G (fun vars => forall x, A (insert vars x 0))
| assume P G => tr_ctxt glob G (fun vars => tr_r P vars -> A vars)
end.
```

Cette fonction traduit un contexte dans l'environnement `glob` puis fournit à une formule `A` l'environnement correspondant à ce contexte (ce qui explique le type `(nat->nat)->Prop` de `A`). La traduction d'un jugement est enfin

Definition `tr_judg glob G A := tr_ctxt glob G (fun vars => tr_r A vars)`.

Le théorème de réflexion auquel nous avons abouti est le suivant

Theorem `reflexion : forall G P glob, Valid Heyting_ax G P -> tr_judg glob G P`.

Il se fait par induction sur la preuve de `Valid Heyting_ax G P` en transformant les étapes de preuves qui utilisent les règles d'inférence de l'arithmétique de Heyting en preuves dans Coq.

Pour le montrer, quelques lemmes supplémentaires ont été nécessaire comme

Lemma `tr_indep : forall glob G A, (forall vars, tr_r A vars) -> tr_judg glob G A`.

qui montre que si la traduction d'une proposition alors sa traduction dans un contexte donné `G` est aussi valide. Nous avons aussi utilisé le lemme

Lemma `tr_cut :`
`forall glob G A B,`
`(forall gvars, tr_r A gvars -> tr_r B gvars) ->`
`tr_judg glob G A -> tr_judg glob G B.`

qui montre que si une proposition `A` implique une proposition `B` dans tout contexte alors si la traduction de `A` est valide dans un contexte `G` donné alors la traduction de `B` est aussi valide dans `G`.

D'autres lemmes ont été utilisé pour montrer que au niveau de Coq, tout se passe comme dans la syntaxe. Par exemple

Lemma `tr_andfI :`
`forall glob G A B, tr_judg glob G A ->`
`tr_judg glob G B ->`
`tr_judg glob G (andf A B).`

qui montre que si l'on a une preuve de la traduction d'une formule `A` dans un contexte `G` et une preuve de la traduction d'une formule `B` dans `G` alors on a une preuve de la traduction de la conjonction des deux formules dans le contexte `G`. C'est donc le pendant en Coq de la règle d'inférence

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

dans l'arithmétique de Heyting.

L'idée commune de tous ces lemmes annexes est de permettre de faire sous la traduction d'un environnement les manipulations qui sont valides dans tout environnement.

2 Traduction de Friedman

2.1 non-non-traduction

La non-non-traduction se définit inductivement en transcrivant la définition proposée

```
Fixpoint notnottrad (A P:Pformula) {struct P} : Pformula :=
match P with
| pf p => orf (pf p) A
| botf => orf botf A
| andf f1 f2 => andf (impf (impf (notnottrad A f1) A) A) (impf (impf (notnottrad A f2) A) A)
| orf f1 f2 => orf (impf (impf (notnottrad A f1) A) A) (impf (impf (notnottrad A f2) A) A)
| impf f1 f2 => impf (impf (impf (notnottrad A f1) A) A) (impf (impf (notnottrad A f2) A) A)
| existsf f => existsf (notnottrad (lift A 1) f)
| forallf f => forallf (impf (impf (notnottrad (lift A 1) f) (lift A 1)) (lift A 1))
end.
```

Il est à noter qu'il ne faut pas oublier de lifter les A lorsque l'on passe sous une quantification (qui est un lieu) afin de garder une liaison des variables cohérente pour A .

La non-non-traduction s'étend aux contextes de façon simple. Là encore, le seul point délicat est de correctement lifter les A afin de maintenir la cohérence des liaisons des variables lorsque l'on passe sous un `intc`. Ceci nous a amené à compter le nombre de `intc` dans le contexte avant la proposition à non-non-traduire et à lifter la proposition A d'autant.

```
Fixpoint nbintc (G:Pctxt) {struct G} : nat :=
match G with
| nilc => 0
| intc G => 1 + (nbintc G)
| assume _ G => nbintc G
end.
```

```
Fixpoint notnottrad_ctxt (A:Pformula) (G:Pctxt) {struct G} : Pctxt :=
match G with
| nilc => nilc
| intc G => intc (notnottrad_ctxt A G)
| assume P G => assume (notnottrad (lift A (nbintc G)) P) (notnottrad_ctxt A G)
end.
```

Il faut ensuite montrer que la non-non-traduction des axiomes de Peano est valide dans l'arithmétique de Heyting

```
Theorem Heyting_axiom_trad_valid :
forall A G P, Peano_ax P -> Valid Heyting_ax G (notnottrad A P).
```

Nous avons admis ce théorème qui est technique mais ne devrait pas poser de problèmes majeurs : les mêmes techniques que dans `trad_valid` (ci-dessous) doivent pouvoir être employées.

Il faut enfin montrer que ce résultat est encore valide pour n'importe quelle formule de Peano

```
Theorem trad_valid :
  forall A (G:Pctxt) (P:Pformula),
  Valid Peano_ax G P ->
  Valid Heyting_ax (notnottrad_ctxt A G)
  (impf (impf (notnottrad (lift A (nbintc G)) P) (lift A (nbintc G)))
  (lift A (nbintc G))).
```

Ce théorème se montre naturellement par induction sur la dérivation de `Valid Peano_ax G P`. Il énonce que si $\Gamma \vdash P$ alors $\Gamma^A \vdash P^A$ est prouvable dans l'arithmétique intuitionniste et ce, pour tout A . La plupart des cas reviennent juste à faire de petites dérivations dans l'arithmétique de Heyting. Ainsi voici par exemple la preuve du cas correspondant à l'implication :

$$\begin{array}{c}
 \text{hypothèse d'induction} \\
 \hline
 B \vdash A \\
 \hline
 \neg^A C, B \vdash A \\
 \hline
 \neg^A C \vdash \neg^A B \\
 \hline
 \neg^A \neg^A B, \neg^A C \vdash A \\
 \hline
 \neg^A \neg^A B \vdash \neg^A \neg^A C \\
 \hline
 \vdash \neg^A \neg^A B \Rightarrow \neg^A \neg^A C \\
 \hline
 \neg^A (\neg^A \neg^A B \Rightarrow \neg^A \neg^A C) \vdash A \\
 \hline
 \vdash \neg^A \neg^A (\neg^A \neg^A B \Rightarrow \neg^A \neg^A C)
 \end{array}$$

Nous avons réussi à montrer tous les cas sauf `IRExistsE` qui s'avère être assez technique notamment à cause de la gestions de la cohérence des liaisons (utilisations `lift`, etc.).

2.2 $P^A \Leftrightarrow P \vee A$

Nous avons défini un prédicat qui est prouvable si et seulement si une formule est sans quantificateurs

```
Fixpoint wq (F : Pformula) {struct F} : Prop :=
match F with
| pf p => True
| botf => True
| andf f1 f2 => (wq f1) /\ (wq f2)
| orf f1 f2 => (wq f1) /\ (wq f1)
| impf f1 f2 => (wq f1) /\ (wq f1)
| existsf f => False
| forallf f => False
end.
```

Nous avons eu des problèmes pour prouver l'équivalence $P^A \Leftrightarrow P \vee A$ si P est une formule sans quantificateur. En effet, si cette formule était vraie dans l'arithmétique de Heyting, alors l'implication $P^A \Rightarrow P \vee A$ serait aussi vraie et donc par réflexion sa traduction serait vraie en Coq. Or, lorsque nous essayons de prouver le lemme

Lemma `wq_or_A_R_tr` :

`forall A P glob, wq P -> (tr_r (notnottrad A P) glob -> tr_r (orf P A) glob).`

nous sommes amenés à prouver une formule du type $((A \rightarrow C) \rightarrow C) \rightarrow (A \rightarrow B \vee C) \rightarrow B \vee C$. On peut prouver que cette formule est vraie en logique classique par la dérivation suivante (nous avons omis quelques étapes)

$$\frac{\frac{\frac{\overline{B \vdash B, C} \quad \overline{C \vdash B, C}}{B \vee C \vdash B, C}}{A, A \rightarrow B \vee C \vdash B, C}}{A \rightarrow B \vee C \vdash B, A \rightarrow C}}{(A \rightarrow C) \rightarrow C, A \rightarrow B \vee C \vdash B, C}}{(A \rightarrow C) \rightarrow C, A \rightarrow B \vee C \vdash B \vee C}$$

mais elle ne semble pas être dérivable en logique intuitionniste! Elle ne serait donc pas valide dans Coq ce qui nous incline à penser que le théorème n'est pas vrai. L'équivalence à démontrer était peut-être $\neg^A \neg^A P^A \Leftrightarrow P \vee A$ mais nous n'avons pas eu le temps de voir si cette équivalence était valide. Cette modification serait compatible avec la définition corrigée de la non-non-traduction.

En admettant la véracité du dernier théorème (ci-dessus), la dernière question se résolvait simplement en instanciant A par $\exists x.P(x)$.

3 Conclusion

Ce DM nous a permis de nous familiariser avec Coq. La première partie concernant la réflexion a été intégralement menée à bien et nous avons réussi à prouver la réflexion de l'arithmétique de Heyting dans le système de types de Coq sans avoir besoin d'axiome (ce qui aurait grandement diminué son intérêt puisque le calcul aurait été bloqué par ces axiomes). Nous avons défini la non-non-traduction et donné la structure de la preuve montrant que la non-non-traduction d'une formule Peano-valide est Heyting-valide.

Il a été très intéressant de voir à quel points des changements qui peuvent sembler minimes sur la formalisation du problème peuvent influencer le déroulement des preuves qui s'en suivent.