Travail de Recherche Encadré :
Preuve du théorème de complétude de Gödel

Benjamin CATINAUD

Année 2020-2021

Introduction

Cet article a pour but d'écrire une preuve du théorème de complétude de Gödel qui établit un rapport entre les formules valides vis-à-vis des modèles de Tarski et les formules prouvables. Plusieurs preuves existent déjà telle la preuve de Henkin.

Nous allons montrer ce théorème en se basant sur la constatation que la traduction par forcing de l'énoncé de la complétude vis-à-vis des modèles de Tarski est l'énoncé de la complétude vis-à-vis des modèles de Kripke. Cette preuve sera écrite avec le plugin *coq-forcing* de l'assistant aux preuves Coq qui, précisément, permet de raisonner en « style indirect » dans la traduction de forcing tout en donnant l'apparence de raisonner en « style direct ». Selon ce point de vue inspiré de l'approche monadique ou co-monadique des effets de bord en programmation, la sémantique de Tarski se distingue de la sémantique de Kripke au sens où cette dernière possède une « mémoire » et où la traduction par forcing permet de laisser cette mémoire implicite.

Ce travail a été fait avec l'aide de Hugo Herbelin, chercheur attaché au laboratoire de l'IRIF.

I. Cadre de travail

Tout d'abord, on se place dans un langage des formules de la logique. On considère d'abord les atomes, que l'on va numéroter par un entier naturel, ainsi que les constructeurs « et » ($\wedge$) et « implique » ($\Rightarrow$) :

```
Inductive form :=
    | Atom : nat -> form
    | And : form -> form -> form
    | Implies : form -> form -> form.
```

Sur cet ensemble de formules, on peut définir la notion de prouvabilité afin de déterminer la valeur de vérité d'une formule. On définit d'abord la relation $\vdash$ sur les listes de formules notée

$$\Gamma \vdash \Delta.$$

La liste Γ est appelée la liste *subséquente* et la liste Δ la liste *conséquente*. On a alors que si les prédicats de Γ sont vrais alors l'une des conséquences de Δ est vraie.

Lorsque la liste Γ est vide, on note plus simplement $\vdash \Delta$.

En pratique dans ce qui suit, on va considérer que Δ est réduite à une seule formule. De plus, on définit Γ en Coq comme étant une liste de formule appelée **context** :

```
Definition context := list form.
```

Définition 1 (Prouvabilité d'une formule)
On définit la prouvabilité d'une formule comme étant un séquent dérivable à partir d'un certain nombre de règles d'inférence et d'axiomes. Dans notre cadre, les règles sont les règles de la logique intuitionniste définies comme suit :

- $\dfrac{A \in \Gamma}{\Gamma \vdash A}$ $(Axiome)$
- $\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$ $(\wedge - intro)$
- $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$ $(\wedge - élim_1)$ ou $\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$ $(\wedge - élim_2)$
- $\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$ $(\Rightarrow - intro)$

- $\dfrac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow -\textit{élim})$.

On définit alors cette notion en Coq comme suit :

```
Inductive provable : context -> form -> Prop :=
    | Ax { ctx A } : In A ctx -> provable ctx A
    | AndI { ctx A B } : provable ctx A -> provable ctx B
                                                -> provable ctx (And A B)
    | AndE1 { ctx A B } : provable ctx (And A B) -> provable ctx A
    | AndE2 { ctx A B } : provable ctx (And A B) -> provable ctx B
    | ImpliesI { ctx A B } : provable (cons A ctx) B -> provable ctx (Implies A B)
    | ImpliesE { ctx A B } : provable ctx (Implies A B) -> provable ctx A
                                                -> provable ctx B.
```

Pour pouvoir interpréter ces formules, on a alors besoin d'une sémantique. Dans ce travail, on va en définir 2 différentes, l'une est la sémantique de Kripke et l'autre la sémantique de Tarski. À chaque fois, la sémantique se base sur un modèle que l'on va d'abord définir.

Définition 2 (Modèle de Kripke)
On définit un modèle de Kripke comme un triplet $\mathcal{K} = (\mathcal{W}, \leqslant, h)$ où

- *$\mathcal{W}$ est l'ensemble des mondes.*
- *$\leqslant$ est une relation d'ordre sur les mondes de $\mathcal{W}$.*
- *$h_{\mathcal{K}}$ est une relation qui indique pour chaque atome X l'ensemble des mondes $\mathcal{W}$ où X est vraie. En théorie des types, elle se traduit comme étant une fonction vers **Prop**.*

$$h_{\mathcal{K}} : \mathcal{W} \times \mathbb{N} \to \textbf{\textit{Prop}}.$$

On définit alors le modèle de Kripke standard, le modèle qui fait coïncider sémantique et prouvabilité sur les atomes :

$$\mathcal{K}_0 = \Big(\mathbf{context}, \mathbf{extend}, h_{\mathcal{K}_0}(\Gamma, X) \triangleq (\Gamma \vdash X)\Big)$$

Définition 3 (Sémantique de Kripke)
On définit la sémantique de Kripke à partir du modèle de Kripke $\mathcal{K}$ comme suit :

- $\omega \Vdash_{\mathcal{K}} X \triangleq h_{\mathcal{K}}(\omega, X)$.
- $\omega \Vdash_{\mathcal{K}} \big(P \dot{\wedge} Q\big) \triangleq \big(\omega \Vdash_{\mathcal{K}} P\big) \wedge \big(\omega \Vdash_{\mathcal{K}} Q\big)$.
- $\omega \Vdash_{\mathcal{K}} \big(P \dot{\Rightarrow} Q\big) \triangleq \forall\, \omega' \geqslant \omega, (\omega' \Vdash_{\mathcal{K}} P) \Rightarrow (\omega' \Vdash_{\mathcal{K}} Q)$.

On peut déjà définir cette sémantique en Coq comme suivant :

```
Fixpoint semK (K : context -> nat -> Prop) (ctx : context) (A : form) :=
    match A with
        | Atom n   => K ctx n
        | And A1 A2 => ((semK K ctx A1) * (semK K ctx A2))%type
        | Implies A1 A2 => forall ctx', extend ctx' ctx -> (semK K ctx' A1)
                                                        -> (semK K ctx' A2)
    end.
```

Définition 4 (Modèle de Tarski)
On caractérise un modèle de Tarski à partir d'une interprétation des atomes, que l'on rappelle être représentés par des entiers :

$$\mathcal{M} : \mathbb{N} \to \textbf{\textit{Prop}}.$$

On remarque alors qu'un modèle de Tarski n'a pas de « mémoire » contrairement à un modèle de Kripke. C'est ce qui fait la difficulté de la preuve du théorème de complétude et, pour contourner cette difficulté, on utilisera alors le plugin *coq-forcing*.

Définition 5 (Sémantique de Tarski)
On définit la sémantique de Tarski à partir du modèle $\mathcal{M}$ comme suit :

- $[\![X]\!]_{\mathcal{M}} \triangleq \mathcal{M}(X)$
- $[\![P \dot{\wedge} Q]\!]_{\mathcal{M}} \triangleq [\![P]\!]_{\mathcal{M}} \wedge [\![Q]\!]_{\mathcal{M}}$.
- $[\![P \dot{\Rightarrow} Q]\!]_{\mathcal{M}} \triangleq [\![P]\!]_{\mathcal{M}} \Rightarrow [\![Q]\!]_{\mathcal{M}}$.

On définit alors la notion de formule valide au sens d'une sémantique comme suit :

Définition 6 (Formule valide)
On dit que la formule φ est valide, au sens de la sémantique ***sem*** *lorsque :*

Pour tout modèle $\mathcal{M}$ de la sémantique ***sem****, on a :* $\boldsymbol{sem}\,\mathcal{M}\,\varphi$.

On écrit alors en Coq, en utilisant la fonction **sem** associée à la sémantique de Tarski que l'on définira dans la 3e partie, le prédicat **valid** comme suit :

```
Definition valid A := forall (M : nat -> Prop), sem M A.
```

Théorème 1 (Théorème de Complétude de Gödel)
Toute formule φ valide, au sens de Tarski, est prouvable à partir du contexte vide : $\vdash \varphi$.

II. Présentation du plugin *Forcing*

Le principe du *forcing* est d'étendre un univers pour le rendre plus expressif. Dans notre cas, le *forcing* de la sémantique de Tarski va permettre de le rendre équivalent à la sémantique de Kripke, pour laquelle on sait déjà faire la démonstration du théorème de complétude de Gödel, via les fonctions mutuellement récursives **reify** et **reflect**.

Le plugin *coq-forcing* est composé de 2 fonctions principales : **Forcing Translate** et **Forcing Definition**.

La fonction **Forcing Translate** s'appuie sur la définition suivante :

Définition 7 (Forcing Translate)
La traduction par forcing est définie par induction structurelle sur les termes comme suit, cf la référence [3] :

$$
\begin{array}{rcc}
[*]_\sigma & := & \lambda(q\,f : \sigma).\Pi(r\,g : \sigma \cdot (q,f))\,.\,* \\
[\square_i]_\sigma & := & \lambda(q\,f : \sigma).\Pi(r\,g : \sigma \cdot (q,f))\,.\,\square_i \\
[x]_\sigma & := & x\,\sigma_e\,\sigma(x) \\
[\lambda x : A.M]_\sigma & := & \lambda x : [\![A]\!]^!_\sigma.[M]_{\sigma\cdot x} \\
[M\,N]_\sigma & := & [M]_\sigma [N]^!_\sigma \\
[\Pi x : A.B]_\sigma & := & \lambda(q\,f : \sigma).\Pi x : [\![A]\!]^!_{\sigma\cdot(q,f)}.[\![B]\!]_{\sigma\cdot(q,f)\cdot x} \\
[\![A]\!]_\sigma & := & [A]_\sigma\,\sigma_e\,\boldsymbol{id}_{\sigma_e} \\
[M]^!_\sigma & := & \lambda(q\,f : \sigma).[M]_{\sigma\cdot(q,f)} \\
[\![A]\!]^!_\sigma & := & \Pi(q\,f : \sigma).[\![A]\!]_{\sigma\cdot(q,f)}
\end{array}
$$

La syntaxe pour **Forcing Translate** dans le plugin est la suivante :

```
Forcing Translate foo {as id} using Obj Hom.
```

Le terme entre accolades est pour signifier que c'est un paramètre optionnel de la commande. Ici, **as id** permet de définir un alias au nom du terme forcé **foo**. Par défaut, le nom est alors $\mathbf{foo}^f$.

Lorsqu'aucune règle de traduction n'est applicable, on peut tout de même utiliser la fonction **Forcing Definition** pour effectuer la preuve de la traduction « à la main ».

La syntaxe pour **Forcing Definition** dans le plugin est la suivante :

```
Forcing Definition foo : type {as id} using Hom Obj.
```

Lors de ce travail, deux modifications du plugin ont été faites.

La première concerne la spécification de **Obj** utilisé lors du *forcing*. En effet, dans la définition de la traduction par forcing (définition 7), le terme $\lambda(q\, f : \sigma)$ permet de généraliser à un **Obj** (dénoté par q) quelconque. Ainsi, la modification apportée est de spécifier ce terme q.

La nouvelle syntaxe pour **Forcing Definition** est alors la suivante :

```
Forcing Definition foo : type {as id} using Hom Obj {from obj}.
```

La deuxième concerne la simplication de la traduction par forcing quand l'ordre est une proposition logique. Les modifications apportées sont les suivantes :

$$\begin{aligned} [*]_\sigma &:= * \\ [\square_i]_\sigma &:= \square_i \\ [\Pi x : A.B]_\sigma &:= \Pi x : [\![A]\!]^!_\sigma.[\![B]\!]_{\sigma.x} \\ [\![A]\!]_\sigma &:= [A]_\sigma \end{aligned}$$

Cette modification permet alors par exemple de traduire **Prop** en **Prop** au lieu de

$$\lambda(q\, f : \sigma), \Pi(r\, g : \sigma \cdot (q, f)).\mathbf{Prop}.$$

Cependant, suivant les types **Obj** et **Hom** utilisés, cette modification est moins expressive et ne fonctionne alors plus. Dans notre cas, cette modification peut s'appliquer et simplifie alors l'expression résultante du *forcing*.

III. Preuve du théorème de complétude

Pour prouver le théorème, on va avoir besoin d'effectuer plusieurs étapes. Tout d'abord, on traduit via le *forcing* **valid** et **provable**. Ensuite, en spécialisant le modèle au modèle de Kripke standard, on se retrouve à montrer que $\mathbf{sem}^f$ implique $\mathbf{provable}^f$. Comme le *forcing* permet d'avoir une équivalence entre $\mathbf{sem}^f$ et **semK** ainsi qu'entre $\mathbf{provable}^f$ et **provable**, on se retrouve alors à montrer que :

$$\forall\, \varphi, \Vdash_{\mathcal{K}_0} \varphi \Rightarrow \vdash \varphi. \tag{1}$$

L'énoncé du théorème de complétude s'écrit comme suit :

```
Forcing Definition completeness : forall A, valid A -> provable nil A
        using context extend from nil.
```

1) Forcing du monde de la sémantique de Tarski

Dans le module de *forcing*, on a besoin d'un objet **Obj** et d'un morphisme **Hom** qui « classe » les objets. Pour l'objet on utilise alors le **context** qu'on avait défini en première partie comme suit :

```
Definition context := list form.
```

Ensuite, pour le morphisme **Hom**, on utilise la relation d'ordre d'*être contenu dans* définie comme suit :

```
Inductive extend : context -> context -> Prop :=
    | extend_nil : extend nil nil
    | extend_cons_no : forall A ctx ctx', extend ctx ctx'
                                        -> extend (cons A ctx) ctx'
    | extend_cons_yes : forall A ctx ctx', extend ctx ctx'
                                        -> extend (cons A ctx) (cons A ctx').
```

On peut alors appliquer le *forcing* pour les définitions qui peuvent être directement traduites :

```
Forcing Translate nat using context extend.
Forcing Translate form using context extend.

Forcing Translate list using context extend.
Forcing Translate context using context extend.

Forcing Translate In using context extend.
Forcing Translate provable using context extend.
```

Il reste alors à définir la fonction **sem**. Pour cela, comme vu dans la définition 5, cette fonction est en fait une fonction récursive. Or, comme les fonctions récursives ne sont pas traitées par le plugin de forcing, on va alors devoir donner la preuve « à la main ». En fait, la preuve peut être donnée sous forme de fonction récursive comme définie dans la définition de la sémantique de Tarski.

```
Forcing Definition sem : forall (M : nat -> Prop) (A : form), Prop
        using context extend.
Proof.
exact (fun p M A =>
    (fix sem p1 Hincl1 A := (* Hincl1 : preuve de p1 INCL p *)
    match A with
    | Atom^f _ n   => M p1 Hincl1 n
    | And^f _ A1 A2 => (sem p1 Hincl1 (A1 p1 (refl_incl p1)))
                           /\ (sem p1 Hincl1 (A2 p1 (refl_incl p1)))
    | Implies^f  _ A1 A2 => (sem p1 Hincl1 (A1 p1 (refl_incl p1)))
                           -> (sem p1 Hincl1 (A2 p1 (refl_incl p1)))
    end) p (refl_incl p) (A p (refl_incl p))
).
Defined.
```

Il reste plus qu'à forcer la définition de **valid** :

```
Forcing Translate valid using context extend.
```

2) Équivalence entre le *forcing* du monde de la sémantique de Tarski et le monde de la sémantique de Kripke

Pour la suite de la preuve, on doit montrer qu'en réalité, forcer la sémantique de Tarski revient à retrouver la sémantique de Kripke. Il s'agit alors de jongler ici entre la traduction donnée par le plugin et les définitions que l'on a posées de la sémantique de Kripke.

Pour cela, on doit montrer les propriétés suivantes :

```
Lemma psi_nat : forall { p }, (forallI p0 INCL p, nat^f) -> nat.
Lemma phi_nat : forall (p : context), nat -> nat^f p

Lemma psi_form : forall { p }, (forallI p0 INCL p, form^f) -> form.
Lemma phi_form : forall { p }, form -> form^f p.

Lemma phi_model : forall (M : context -> nat -> Prop),
    forall { p : context }, forall p0 : context, p0 INCL p ->
        (forallI p1 INCL p0, nat^f) -> Prop.

Lemma psi_sem : forall { p M A }, sem^f p (phi_model M) A
                                    -> semK M nil (psi_form A).
Lemma phi_context : forall { p }, context -> context^f p.
Lemma phi_provable : forall { ctx A }, provable ctx (psi_form A)
            -> provable^f nil (fun p0 Hincl => (phi_context ctx) ) A.
```

Pour prouver ces lemmes, on peut par exemple exhiber une fonction récursive de traduction. Par exemple pour **phi_form**, on peut écrire :

```
Lemma phi_form : forall { p }, form -> form^f p.
Proof.
exact (fun p A =>
    (fix phi_form_rec p A :=
    match A with
    | Atom n => Atom^f p (fun p0 _ => phi_nat p0 n)
    | And A1 A2 => And^f p (fun p0 _ => phi_form_rec p0 A1)
                           (fun p0 _ => phi_form_rec p0 A2)
    | Implies A1 A2 => Implies^f p (fun p0 _ => phi_form_rec p0 A1)
                                   (fun p0 _ => phi_form_rec p0 A2)
    end) p A
).
Defined.
```

Et de même pour les autres lemmes.

Cependant, les lemmes **psi_sem** et **phi_provable** sont plus complexes et ont alors été admis.

3) Fonctions reify et reflect et preuve du théorème de complétude

Grâce aux deux parties précédentes, le théorème de complétude de Gödel se ramène alors à montrer que :

```
forall (p : context) (A : form), semK K0 p A -> provable p A.
```

où $\mathcal{K}_0$ est le modèle de Kripke standard définie dans la première partie.

Pour démontrer ce théorème, on va alors définir deux fonctions mutuellement récursives : **reify** et **reflect** telles que :

- **reify** permet de passer de la sémantique à la syntaxe (avec **provable**)

- **reflect** permet de faire l'inverse : passer de la syntaxe à la sémantique.

La définition est alors la suivante :

```
Fixpoint reify p A : semK K0 p A -> provable p A :=
    match A with
        | Atom n   => (fun v => v)
        | And A1 A2 => (fun '(v1, v2) => AndI (reify p A1 v1) (reify p A2 v2))
        | Implies A1 A2 => (fun v => ImpliesI (reify (A1 :: p) A2
                    (v (A1 :: p) (extend_cons_no A1 p p (id_extend p))
                        (reflect (A1 :: p) A1 (Ax (In_cons_yes A1 p))))))
    end
with reflect p A : provable p A -> semK K0 p A :=
    match A with
        | Atom n   => (fun t => t)
        | And A1 A2 => (fun t => (reflect p A1 (AndE1 t), reflect p A2 (AndE2 t)))
        | Implies A1 A2 => (fun t => (fun p0 Hinclp0 a => reflect p0 A2
                    (ImpliesE (compatibility_extend_provable Hinclp0 t)
                        (reify p0 A1 a))))
    end.
```

Pour la partie « implique » de la fonction **reflect**, comme dans la définition de la sémantique de Kripke il y a un « pour tout », on doit alors écrire un lemme qui permet d'exprimer la compatibilité entre **extend** et **provable** :

```
Lemma compatibility_extend_provable :
    forall { A ctx ctx' }, extend ctx' ctx -> provable ctx A
                                    -> provable ctx' A.
```

Pour la preuve de ce lemme, *cf* la preuve en annexe.

On peut maintenant écrire la preuve du théorème de complétude. Pour cela, on applique d'abord le lemme **psi_sem** pour passer de la forme forcée de **sem** à **semK**. Ensuite, on applique **reify** pour passer de **semK** à **provable**. Enfin, on applique **phi_provable** pour passer de **provable** à sa forme forcée.

```
Forcing Definition completeness : forall A, valid A -> provable nil A
        using context extend from nil.
Proof.
    intros.

    specialize H with (p := nil) (alpha := refl_incl nil).
    unfold valid^f in H.
    specialize H with (M := phi_model K0).

    apply psi_sem, reify, phi_provable in H . assumption.
Qed.
```

Conclusion

Lors de ce travail, nous avons été freinés par le plugin *coq-forcing* pour plusieurs raisons. Ce plugin ne prend pas en charge les fonctions de type **Fixpoint**. Les traductions, notamment celle des types inductifs, rajoutent beaucoup de « bruit » : $\mathbf{nat}^f$ devrait être équivalent à **nat**. Nous avons cependant soumis 2 pull requests pour modifier ce plugin. Moralement les lemmes **psi_sem** et **phi_provable** devraient fonctionner. Une solution serait peut être de contourner ce problème en utilisant directement le type $\mathbf{provable}^f$ dans les définitions de **reify** et de **reflect** ainsi que celle de $\mathcal{K}_0$.

Bibliography

[1] Olivier Danvy, Morten Rhiger, Kristoffer H. Rose. *Normalization by Evaluation with Typed Abstract Syntax*, BRICS Report Series, RS-01-16, May 2001.

[2] Catarina Coquand. *A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions*, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 412 96 Göteborg, Sweden.

[3] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau. *The Definitional Side of the Forcing*, Logics in Computer Science, May 2016, New York, United States.

[4] Hugo Herbelin. *A proof with side effects of Gödel's completeness theorem*, Exposé à l'Institut Hausdorff de Mathématiques (HIM), 2018.

[5] CoqHott. *Plugin Coq pour le Forcing*, https://github.com/CoqHott/coq-forcing.

Code source du projet

```
From Forcing Require Import Forcing.
Require Import List.
Require Import Program.

Import ListNotations.

Inductive form :=
  | Atom : nat -> form
  | And : form -> form -> form
  | Implies : form -> form -> form.

Definition context := list form.

Inductive extend : context -> context -> Prop :=
  | extend_nil : extend nil nil
  | extend_cons_no : forall A ctx ctx', extend ctx ctx' -> extend (cons A ctx) ctx'
  | extend_cons_yes : forall A ctx ctx', extend ctx ctx' -> extend (cons A ctx) (cons A ctx').

Inductive In (A : form) : context -> Prop :=
  | In_cons_yes : forall ctx, In A (cons A ctx)
  | In_cons_no : forall ctx, In A ctx -> forall B, In A (cons B ctx).

Inductive provable : context -> form -> Prop :=
  | Ax { ctx A } : In A ctx -> provable ctx A
  | AndI { ctx A B } : provable ctx A -> provable ctx B -> provable ctx (And A B)
  | AndE1 { ctx A B } : provable ctx (And A B) -> provable ctx A
  | AndE2 { ctx A B } : provable ctx (And A B) -> provable ctx B
  | ImpliesI { ctx A B } : provable (cons A ctx) B -> provable ctx (Implies A B)
  | ImpliesE { ctx A B } : provable ctx (Implies A B) -> provable ctx A -> provable ctx B.

Lemma compatibility_extend_in :
  forall A ctx ctx', extend ctx' ctx -> In A ctx -> In A ctx'.
Proof.
  intros A ctx ctx'. intros Hextend Hin.
  induction Hextend.
  - assumption.
  - apply IHHextend in Hin. apply In_cons_no. assumption.
  - dependent induction Hin.
    * apply In_cons_yes.
    * apply In_cons_no. apply IHHextend. assumption.
Qed.

Lemma compatibility_extend_provable :
  forall { A ctx ctx' }, extend ctx' ctx -> provable ctx A -> provable ctx' A.
Proof.
```

```
  intros A ctx ctx' Hextend Hprovable.
  induction Hprovable in ctx', Hextend.
  - apply Ax. apply compatibility_extend_in with (ctx:=ctx) ; assumption.
  - apply AndI.
    * apply IHHprovable1. assumption.
    * apply IHHprovable2. assumption.
  - apply IHHprovable in Hextend. apply AndE1 in Hextend. assumption.
  - apply IHHprovable in Hextend. apply AndE2 in Hextend. assumption.
  - apply ImpliesI. specialize IHHprovable with (cons A ctx').
    apply IHHprovable. apply extend_cons_yes. assumption.
  - assert (HextendBis : extend ctx' ctx). assumption.
    apply IHHprovable1 in Hextend. apply IHHprovable2 in HextendBis.
    apply @ImpliesE with (A:=A) ; assumption.
Qed.

Fixpoint semK (K : context -> nat -> Prop) (ctx : context) (A : form) :=
  match A with
    | Atom n   => K ctx n
    | And A1 A2 => ((semK K ctx A1) * (semK K ctx A2))%type
    | Implies A1 A2 => forall ctx', extend ctx' ctx -> (semK K ctx' A1) -> (semK K ctx' A2)
  end.

Notation "p 'INCL' q" := (forall R, extend p R -> extend q R)
                                (at level 70, q at next level).
Notation "'forallI' q 'INCL' p , P" := (forall q, q INCL p -> P q)
                                (at level 200, q at level 69, p at level 69).

Lemma refl_incl :
  forall p, p INCL p.
Proof.
  auto.
Defined.

Lemma id_extend :
  forall p, extend p p.
Proof.
  intros. induction p.
  - apply extend_nil.
  - apply extend_cons_yes. assumption.
Qed.

Lemma trans_incl :
  forall {p q r}, p INCL q -> q INCL r -> p INCL r.
Proof.
  auto.
Defined.

Forcing Translate nat using context extend.
Forcing Translate form using context extend.

Forcing Definition sem : forall (M : nat -> Prop) (A : form), Prop using context extend.
Proof.
  exact (fun p M A =>
    (fix sem p1 Hincl1 A := (* Hincl1 : preuve de p1 INCL p *)
      match A with
      | Atom^f _ n   => M p1 Hincl1 n
```

```
      | And^f _ A1 A2 => (sem p1 Hincl1 (A1 p1 (refl_incl p1)))
                          /\ (sem p1 Hincl1 (A2 p1 (refl_incl p1)))
      | Implies^f _ A1 A2 => (sem p1 Hincl1 (A1 p1 (refl_incl p1)))
                          -> (sem p1 Hincl1 (A2 p1 (refl_incl p1)))
    end) p (refl_incl p) (A p (refl_incl p))
  ).
Defined.

Definition valid A := forall (M : nat -> Prop), sem M A.
Forcing Translate valid using context extend.

Forcing Translate list using context extend.
Forcing Translate context using context extend.

Forcing Translate In using context extend.
Forcing Translate provable using context extend.

Lemma psi_nat : forall { p }, (forallI p0 INCL p, nat^f) -> nat.
Proof.
  exact (fun p n =>
    (fix psi_nat_rec n :=
      match n with
      | O^f _    => O
      | S^f _ m  => S (psi_nat_rec (m p (refl_incl p)))
      end
    ) (n p (refl_incl p))
  ).
Qed.

Lemma phi_nat : forall (p : context), nat -> nat^f p.
Proof.
  exact (fun p n =>
    (fix phi_nat_rec p n :=
      match n with
      | O => O^f p
      | S m => S^f p (fun p0 Hp0 => (phi_nat_rec p0 m))
      end
    ) p n
  ).
Qed.

Lemma psi_form : forall { p }, (forallI p0 INCL p, form^f) -> form.
Proof.
  exact (fun p A =>
    (fix psi_rec A :=
      match A with
      | Atom^f _ n    => Atom (psi_nat n)
      | And^f _ A1 A2  => And (psi_rec (A1 p (refl_incl p)))
                              (psi_rec (A2 p (refl_incl p)))
      | Implies^f _ A1 A2 => Implies (psi_rec (A1 p (refl_incl p)))
                              (psi_rec (A2 p (refl_incl p)))
      end) (A p (refl_incl p))
  ).
Defined.

Lemma phi_form : forall { p }, form -> form^f p.
```

```
Proof.
  exact (fun p A =>
    (fix phi_form_rec p A :=
      match A with
      | Atom n => Atom^f p (fun p0 _ => phi_nat p0 n)
      | And A1 A2 => And^f p (fun p0 _ => phi_form_rec p0 A1)
                             (fun p0 _ => phi_form_rec p0 A2)
      | Implies A1 A2 => Implies^f p (fun p0 _ => phi_form_rec p0 A1)
                             (fun p0 _ => phi_form_rec p0 A2)
      end) p A
  ).
Defined.

(* Modèle universel *)
Definition K0 ctx n := provable ctx (Atom n).

Fixpoint reify p A : semK K0 p A -> provable p A :=
  match A with
    | Atom n   => (fun v => v)
    | And A1 A2 => (fun '(v1, v2) => AndI (reify p A1 v1) (reify p A2 v2))
    | Implies A1 A2 => (fun v => ImpliesI (reify (A1 :: p) A2
           (v (A1 :: p) (extend_cons_no A1 p p (id_extend p))
               (reflect (A1 :: p) A1 (Ax (In_cons_yes A1 p))))))
  end
with reflect p A : provable p A -> semK K0 p A :=
  match A with
    | Atom n   => (fun t => t)
    | And A1 A2 => (fun t => (reflect p A1 (AndE1 t), reflect p A2 (AndE2 t)))
    | Implies A1 A2 => (fun t => (fun p0 Hinclp0 a => reflect p0 A2
           (ImpliesE (compatibility_extend_provable Hinclp0 t) (reify p0 A1 a))))
  end.

Lemma phi_model : forall (M : context -> nat -> Prop),
  forall { p : context }, forall p0 : context, p0 INCL p ->
    (forallI p1 INCL p0, nat^f) -> Prop.
Proof.
  exact (fun M p (p0 : context) Hinclp0 (n : forallI p INCL p0, nat^f)
     => M p0 (psi_nat n) ).
Defined.

Lemma psi_sem : forall { p M A }, sem^f p (phi_model M) A -> semK M nil (psi_form A).
Proof.
Admitted.

Lemma phi_context : forall { p }, context -> context^f p.
Proof.
  exact (fun p ctx =>
    (fix phi_ctx_rec p ctx : context^f p :=
      match ctx with
      | [] => nil^f p _
      | A :: ctx' => cons^f p _ (fun p1 _ => phi_form A) (fun p1 _ => phi_ctx_rec p1 ctx')
      end
    ) p ctx
  ).
Defined.
```

```
Lemma phi_provable : forall { ctx A }, provable ctx (psi_form A)
        -> provable^f nil (fun p0 Hincl => (phi_context ctx) ) A.
Proof.
Admitted.

Forcing Definition completeness : forall A, valid A -> provable nil A
        using context extend from nil.
Proof.
  intros.

  specialize H with (p := nil) (alpha := refl_incl nil).
  unfold valid^f in H.
  specialize H with (M := phi_model K0).

  apply psi_sem, reify, phi_provable in H . assumption.
Qed.
```