Rapport de stage

Extensions d'un cours de logique du premier ordre certifié en Coq

Stage effectué du 1 juin au 17 juillet 2020

Samuel Ben Hamou
École Normale Supérieure Paris-Saclay
1ère année, Département Informatique

Sous la supervision de Pierre Letouzey
IRIF, INRIA et CNRS

Résumé

Ce stage de recherche de fin de licence avait pour but de proposer des extensions à un encodage en Coq déjà existant de la déduction naturelle.

Dans un premier temps, il a fallu se familiariser avec les outils du stage : à la fois la manipulation de git mais aussi l'encodage parfois étonnant des notions classiques de la logique (formules closes, variables liées, modèles, etc.). Un fois ce travail de défrichage effectué, nous avons pu enrichir le codage existant en suivant trois axes principaux, qui correspondent aux trois sections principales de ce rapport :

- l'arithmétique de Peano, qui avait déjà été encodée mais où tout restait à faire du point de vue des exemples et de la maniabilité des preuves,
- la théorie des ensembles ZF, où il s'est agit non seulement d'encoder les axiomes et de faire quelques preuves exemples, mais aussi de proposer une construction de $\mathbb{N}$,
- le λ-calcul enfin, où nous avons pu établir une correspondance de Curry-Howard pour des logiques propositionnelles de plus en plus expressives.

Table des matières

Introduction

Contexte pratique du stage

Ce stage a été effectué sous la direction de Pierre Letouzey, maître de conférence à l'Université Paris-Diderot, et membre du groupe πr^2 de l'IRIF. Compte tenu du contexte sanitaire, la majeur partie des échange a eu lieu par visio-conférence et par mail, à un rythme assez soutenu. J'ai cependant pu me rendre physiquement à l'IRIF afin de discuter dans la vraie vie avec avec mon maître de stage.

Malgré les échanges de mails réguliers, le travail à distance a probablement été le plus gros inconvénient de ce stage : dans la mesure où je travaillais sur un codage créé par Pierre Letouzey, qui n'a pas écrit de manuel d'explication pour justifier ou expliquer ses choix d'implémentation (ce qui se conçoit parfaitement étant donné l'usage qu'il avait prévu de faire de son travail, qui n'était censé être utilisé que pour illustrer le cours de logique qu'il donne en M1), j'ai donc eu grandement besoin d'explications pour me lancer, et il eût probablement été plus aisé d'avoir ce genre de discussions informelles si nous avions pu travailler dans deux bureaux adjacents.

Il n'en demeure pas moins que le stage a pu se dérouler sans trop d'embûches, et sans problèmes matériels majeurs. Par ailleurs, le travail à distance présente l'avantage non négligeable de permettre à chacun de travailler aux horaires qui lui conviennent le mieux, fût-ce tard le soir [1].

Le dépôt GitLab du projet est consultable à l'adresse suivante : `https://gitlab.math.univ-paris-diderot.fr/letouzey/natded`.

Motivations

Le but du stage a été d'enrichir l'encodage `NatDed` – un encodage profond de la déduction naturelle – déjà existant, tout en gardant en tête la vocation pédagogique du projet, puisqu'il s'agit d'illustrer un cours de logique de M1 [5].

`NatDed` étant à l'origine assez rustique, il était intéressant de le rendre plus *user-friendly*, mais aussi d'ajouter un certain nombre d'exemples et d'applications, qui manquaient au projet initial.

Par la suite, il a été à propos d'ajouter d'autres aspects de la logique à cet encodage, tels que le λ-calcul et la théorie des ensembles, qui sont tous deux des point abordés dans le cours donné par Pierre Letouzey. En effet, le λ-calcul est un élément essentiel de la construction des assistants de preuve tels que Coq.

Encodage existant et état de l'art

L'encodage `NatDed` existait déjà avant le stage. À ce moment là, il contenait déjà un système de déduction naturelle avec des séquents, ainsi qu'un grand nombre de lemmes et résultats utiles, y compris sur la méta-théorie. Pour l'encodage des formules de la logique du premier ordre, Pierre Letouzey avait choisi un système *locally nameless*, i.e. où les variables liées sont représentées par des indices de De Bruijn et les variables libres par un constructeur `FVar : string -> term` (par exemple x s'écrira `FVar "x"`) [2]. En parallèle de cette convention, une équivalence avec un système reposant totalement sur des variables nommées était proposée, mais quoiqu'elle eût peut se révéler fort utile, cette équivalence n'a guère été utilisée pendant le stage.

Concernant les preuves, deux méthodes sont possibles pour les établir. `Pr` est une description inductive et relativement légère de la prouvabilité d'un séquent, permettant de construire des preuves pas à pas (sa définition complète est donnée en annexe C). Par contre, Coq ne permet pas ensuite de revenir facilement analyser les étapes internes de cette preuve, c'est pourquoi un autre prédicat, `Valid`, qui explicite l'ensemble de la dérivation de preuve et permet des études méta-théoriques plus poussées est disponible, en contrepartie d'un style de preuve nettement plus lourd. L'équivalence entre `Pr` et `Valid` avait déjà été montrée. Le choix qui a été fait pour stage est d'utiliser Pr systématiquement.

A priori, il s'agit du premier codage Coq de la déduction naturelle qui allie les trois aspects suivants :

— *Calcul* : l'ensemble du système n'est pas que de la syntaxe inerte, puisqu'il tire également partie de la capacité de Coq à effectuer un certain nombre de calculs à la place de l'utilisateur. Par exemple, les substitutions sont entièrement gérées par une fonction `bsubst` ; et l'on dispose d'une d'une fonction booléenne permettant de vérifier la validité d'une dérivation.

— *Méta-théorie* assez poussée, dans la mesure où un théorème de complétude a pu être démontré.

— *Pédagogie* : avec un peu de patience – et, je l'espère, en s'aidant du travail effectué pendant le stage – on peut utiliser ce codage pour illustrer un cours sur la théorie de la démonstration, ou bien justifier que l'on peut admettre un certain nombre de résultats longs à démontrer en exhibant une preuve (plus ou moins lisible à l'œil nu) dans ce système.

1. Ou bien tôt le matin selon la façon dont on voit les choses.
2. Pour la définition complète des formules, on pourra se reporter à l'annexe B.

Mise en place de l'environnement de travail

Avant de se lancer dans le stage à proprement parler, il a fallu consacrer une petite semaine à l'acquisition de réflexes de travail, à la fois sur le support (GitLab) et sur l'encodage lui-même.

Dépôt GitLab

La première journée du stage a été passée à appréhender un outil nouveau pour manipuler de gros projets de code : GitLab. À ce titre, et pour me fixer les idées, j'ai rédigé la cheatsheet présentée en A.

L'usage de git a été d'un grand recours, notamment lorsque nous voulions discuter d'un morceau de preuve en visio-conférence, car chacun pouvait travailler et mettre à jour le dépôt de son côté, la fusion se faisant ensuite aisément grâce à la commande `git pull`.

Premières preuves

Afin de me faire la main sur un encodage déjà très évolué, j'ai tenté de démontrer les petits résultats classiques suivants :

- $\vdash (\phi_1 \wedge \phi_2) \Rightarrow (\phi_1 \vee \phi_2)$,
- $\vdash (\phi_1 \Rightarrow \phi_2 \Rightarrow \phi_3) \Leftrightarrow (\phi_1 \wedge \phi_2 \Rightarrow \phi_3)$,
- $\vdash \phi \vee \neg\phi$.

Pour ce dernier point, il a été utile d'établir l'admissibilité des règles suivantes :

$$\frac{\Gamma \vdash \neg\neg\phi}{\Gamma \vdash \phi}\ \text{RAA} \qquad \frac{\Gamma \vdash \neg(\neg\phi_1 \wedge \phi_2)}{\Gamma \vdash \phi_1 \vee \neg\phi_2}\ \text{Morgan}$$

puisque la règle de l'absurde présente dans l'encodage initial était

$$\frac{\neg\phi, \Gamma \vdash \bot}{\Gamma \vdash \phi}\ \text{absu}$$

Évidemment, les règles RAA et absu sont équivalentes, mais la forme RAA était plus commode pour montrer le tiers-exclu.

À titre d'illustration, voici l'énoncé de ces résultats tels que démontrés dans le projet. On donne également la démonstration du premier, afin d'avoir une idée de ce à quoi ressemble une preuve en déduction naturelle dans cet encodage.

```
Lemma ex1 f1 f2 : Pr J ([] ⊢ (f1 /\ f2) -> (f1 \/ f2)).
Proof.
  apply R_Imp_i.
  apply R_Or_i1.
  apply R_And_e1 with (B := f2).
  apply R_Ax.
  apply in_eq.
Qed.

Lemma ex2 f1 f2 f3 : Pr J ([] ⊢ (f1 -> f2 -> f3) <-> (f1 /\ f2 -> f3)).

Lemma RAA f1 Γ : Pr K (Γ ⊢ ~~f1) -> Pr K (Γ ⊢ f1).

Lemma DeMorgan f1 f2 Γ : Pr K (Γ ⊢ ~(~f1 /\ f2)) -> Pr K (Γ ⊢ ~~(f1 \/ ~f2)).

Lemma ExcludedMiddle f1 : Pr K ([] ⊢ f1 \/ ~f1).
```

Au niveau de Coq, `f1`, `f2` et `f3` sont des variables de type `forumla`. Quant aux lettres `K` et `J`, elles font respectivement référence aux logiques classique et intuitionniste.

Les symboles `->`, `/\`, `\/` et `~` utilisés à l'intérieur des séquents n'ont *rien à voir* avec les opérateurs Coq du même nom : il s'agit d'une surcharge de notation pour rendre les preuves plus lisibles. C'est d'ailleurs cet usage du *deep-embedding* qui permet de démontrer des résultats méta-théoriques.

Pour revenir sur la remarque précédente, on voit dans l'annexe B que, par exemple, `A /\ B` n'est qu'un sucre syntaxique pour `Op And A B`.

Par ailleurs, on verra dans la suite que les quantificateurs ne sont pas associés à des noms de variables. Cela vient du fait que la représentation des formules est *locally-nameless*, et donc les variables liées sont représentées par des indices de De Bruijn grâce à un constructeur `BVar : nat -> term` (qui est ensuite abrégé par `#`, cf. annexe B).

1 Arithmétique de Peano

Dans cette partie du stage, nous avons illustré au premier ordre et simplifié l'encodage de PA préexistant.

1.1 Constats de départ

L'encodage de PA contenait déjà les axiomes ainsi que quelques éléments méta-théoriques. En outre, le fichier `Theories.v` propose un prédicat `IsTheorem` défini comme suit :

```
Definition IsTheorem th T :=
  WC th T /\
  exists axs,
    Forall th.(IsAxiom) axs /\
    Pr logic (axs ⊢ T).
```

Ce prédicat permet d'énoncer des théorèmes en prenant en compte le fait que les séquents à dériver respectent bien la signature de la théorie, et que les axiomes choisis sont licites.

Mon premier travail a été de mettre cet encodage à l'épreuve, afin de pouvoir ensuite déterminer les points à améliorer, et comparer la longueur des preuves avec et sans tactiques adjuvantes. Le but, bien entendu, était d'écrire les tactiques les plus générales possible, et ce pari a été plutôt réussi puisque certaines d'entre elles ont pu être réutilisées dans `ZF.v` par la suite. À cet effet, les résultats suivants ont été démontrés (on rappelle que # sert à désigner les indices de de Bruijn) :

```
Lemma ZeroRight : IsTheorem J PeanoTheory (∀ (#0 = #0 + Zero)).

Lemma SuccRight : IsTheorem J PeanoTheory (∀∀ (Succ(#1 + #0) = #1 + Succ(#0))).

Lemma Comm :
  IsTheorem J PeanoTheory
            ((∀ #0 = #0 + Zero) -> (∀∀ Succ(#1 + #0) = #1 + Succ(#0)) ->
               (∀∀ #0 + #1 = #1 + #0)).

Lemma Commutativity : IsTheorem J PeanoTheory (∀∀ #0 + #1 = #1 + #0).
```

Le but initial a été de démontrer la commutativité de la loi $+$ dans le cadre de PA, cependant, quelques lemmes intermédiaires furent nécessaires. L'existence d'un lemme `Comm` et d'un autre lemme `Commutativity` illustre la première pierre d'achoppement de l'encodage existant des théories : la difficulté à utiliser des lemmes auxiliaires dans le corps d'une preuve, puisque cela demanderait de savoir quels axiomes sont utilisés dans lesdits lemmes. La solution proposée dans ce problème est détaillée en section 1.2.3.

Par ailleurs, les débuts de preuve avaient tendance à être particulièrement inintéressants et répétitifs : passer la liste des axiomes de Peano en argument à la quantification existentielle [3], justifier que la formule que l'on souhaite montrer respecte bien la signature de PA, initier un raisonnement par récurrence avec la bonne instance du schéma d'axiomes de récurrence si nécessaire. Tous ces éléments rendaient les preuves lourdes et paraissaient pourtant hautement automatisables.

Quelques propriétés utiles étaient par ailleurs sans cesse utilisées. Si la transitivité et la symétrie de l'égalité font partie de toute théorie égalitaire qui se respecte, le besoin de règles de dérivation telles que

$$\frac{\Gamma \vdash u = v}{\Gamma \vdash v = u}\ \text{sym}$$

se fait cruellement ressentir. En l'état, l'utilisation de l'axiome de symétrie de l'égalité nécessitait d'avoir recours à la tactique `assert` pour rajouter un jugement précisant que le séquent $\mathsf{PA} \vdash \forall x, \forall y \cdot x = y \Rightarrow y = x$ est dérivable, puis d'utiliser les règles d'élimination adéquates : il s'agit de *forward-reasoning*. Il en va de même pour les axiomes du type $\forall x, \forall y \cdot x = y \Rightarrow \mathrm{Succ}(x) = \mathrm{Succ}(y)$. Cette difficulté à manipuler l'égalité provient du choix qui a été fait par Pierre Letouzey de ne pas coder en dur les axiomes de l'égalité dans la méta-théorie (ce qui reviendrait à poser le principe de Leibniz comme axiome), mais plutôt d'ajouter à toute théorie T que l'on est amené à définir les axiomes de symétrie, réflexivité et transitivité de l'égalité, ainsi qu'un certain nombre d'axiomes de compatibilités avec les autres symboles de la signature de T. Ce choix permet de travailler à un plus grand niveau d'abstraction, et laisse tout de même la possibilité de démontrer – non sans peine – un principe de Leibniz dans chacune des théories ainsi créées.

Enfin, certaines tactiques et règles définies dans l'encodage de `NatDed` étaient passablement pénibles à utiliser. Entre autres, on peut citer `R_All_i`, la règle d'introduction du quantificateur universel, qui engendrait un grand nombre de

3. Avec toutefois un petit problème lorsqu'il s'agissait de raisonner par récurrence, puisque le schéma d'axiomes de récurrence engendre une infinité d'axiomes, ce que Coq a beaucoup de mal à gérer.

buts ennuyants à démontrer. On peut également mentionner tous les résultats faisant intervenir des ensembles finis, par exemple le lemme d'affaiblissement, car la représentation de ces ensembles en Coq est surprenante et se prête fort mal à des tactiques de simplification usuelles telles que `cbn` lorsqu'il reste des éléments abstraits.

1.2 Définition de nouvelles tactiques

Afin de régler les susmentionnés problèmes, nous avons mis au point un certain nombre de tactiques permettant de rendre les preuves plus courtes et plus lisibles, en somme plus proches des preuves que l'ont ferait sur papier. L'objectif ultime étant que l'on puisse facilement voir émerger l'arbre de dérivation du séquent que l'on est en train de démontrer. Afin de constater l'apport de ces nouvelles tactiques, l'annexe D propose une mise en regard des preuves du lemme `ZeroRight` avant simplifications (73 lignes) et après (13 lignes).

1.2.1 Pour les débuts de preuve

Dans l'objectif d'alléger les débuts de preuve, nous avons mis en place deux tactiques distinctes.

La première, `thm`, permet de traiter à peu de frais le but `WC th T` affirmant que la formule à démontrer respecte la signature de la théorie T. Sa mise en place a été relativement aisée car la preuve du but qu'elle entend éliminer est toujours la même, indépendamment du théorème à démontrer ou de la théorie considérée. Il a donc suffit de copier-coller la preuve donnée dans `ZeroRight` (par exemple) pour obtenir cette tactique qui a fonctionné quasiment du premier coup. Évidemment, le stage avançant, la tactique a été enrichie par d'autres tactiques permettant notamment de calculer plus efficacement sur les listes, etc.

La seconde tactique élaborée, `rec`, est dédiée aux preuves pas récurrence. Elle permet, à partir du but fourni par la tactique `thm` de trouver la bonne instance A du schéma d'axiomes de récurrence et d'instancier ensuite la quantification existentielle par `(induction_schema A) :: axioms_list`.

Concrètement, cette tactique s'appuie sur une tactique auxiliaire `parse` qui prend en argument une formule ϕ contenant au moins une quantification universelle sous la forme générale

$$\phi_1 \Rightarrow \cdots \Rightarrow \forall x \cdot \phi_n$$

avec éventuellement $n = 1$, et qui retourne ϕ_n. Une fois la formule ϕ_n identifiée, `rec` l'utilise pour instancier le schéma d'axiomes de récurrence et initie la preuve, de sorte que l'utilise se retrouve avec deux buts, qui correspondent à l'initialisation et à l'hérédité de la preuve par récurrence. Pour ce faire, nous avons eu besoin de prouver deux lemmes auxiliaires qui permettent essentiellement de montrer que l'instance du schéma d'axiomes de récurrence choisie est bien licite, et qui proposent une forme plus facilement manipulable du principe de récurrence.

Il est à noter que, en allant chercher le premier quantificateur universel qu'elle trouve dans la formule, `parse` et `rec`, bien qu'elles n'illustrent pas toutes les instances du schéma d'axiomes de récurrence, permettent tout de même d'effectuer toutes les récurrence que l'on souhaite, pour peu que l'on prenne le temps de permuter convenablement les quantificateurs universels au préalable. En effet, si l'on veut faire une récurrence sur la variable qui n'est la première variable quantifiée, on peut toujours éliminer les quantificateurs qui la précèdent (avec la règle `R_All_i`) afin d'appliquer la tactique `rec`, puis les réintroduire avec `R_All_e`. En Coq, cela revient essentiellement à utiliser les tactiques `intro` puis `revert` pour permuter les quantificateurs universels, puis de lancer `induction` sur la nouvelle première variable. Au passage, on remarque que le schéma d'axiomes de récurrence usuellement énoncé est en fait redondant : certaines instances du schéma d'axiomes peuvent se retrouver à partir d'autres.

1.2.2 Pour raccourcir certains raisonnements

La création de tactiques permettant d'utiliser rapidement des propriétés telles que la symétrie de l'égalité ou sa compatibilité avec le prédicat Succ s'est fait en deux étapes.

Dans un premier temps, nous avons démontré les lemmes suivants :

```
Lemma Symmetry :
  forall logic A B Γ, BClosed A -> In ax2 Γ -> Pr logic (Γ ⊢ A = B) ->
                      Pr logic (Γ ⊢ B = A).

Lemma Transitivity :
  forall logic A B C Γ, BClosed A -> BClosed B -> In ax3 Γ ->
                        Pr logic (Γ ⊢ A = B) -> Pr logic (Γ ⊢ B = C) -> Pr logic (Γ ⊢ A = C).

Lemma Hereditarity :
  forall logic A B Γ, BClosed A -> In ax4 Γ -> Pr logic (Γ ⊢ A = B) ->
                      Pr logic (Γ ⊢ Succ A = Succ B).
```

```
Lemma AntiHereditarity :
  forall logic A B Γ, BClosed A -> In ax13 Γ -> Pr logic (Γ ⊢ Succ A = Succ B) ->
                      Pr logic (Γ ⊢ A = B).
```

L'intervention du prédicat `BClosed` provient de ce que, pour des raisons techniques liées à l'instantiation des axiomes par la fonction `bsubst`, des résultats tels que `Symmetry` ne passent bien que dans le cas de formules utilisant correctement les indices de DE BRUIJN.

Une fois ces résultats établis, nous avons facilement pu créer les tactiques qui nous intéressaient, où il s'agissait essentiellement d'appliquer le théorème idoine et de simplifier à l'aide de tactiques de calcul spécifiques, telles que `calc` ou `cbm`, qui ont été écrites par Pierre Letouzey pour faciliter la manipulation d'ensembles finis et de listes.

```
Ltac sym := apply Symmetry; calc.

Ltac ahered := apply Hereditarity; calc.

Ltac hered := apply AntiHereditarity; calc.

Ltac trans b := apply Transitivity with (B := b); calc.
```

En plus de cela, nous avons aussi défini des tactiques qui permettent de gérer plus facilement la règle `R_All_i` ou d'ajouter la dérivabilité d'axiomes dans les hypothèses, pour pouvoir raisonner à rebours (i.e. de haut de bas de l'arbre de dérivation). Ces tactiques très simples ne sont finalement que des macros qui évitent d'avoir à réécrire plusieurs fois les mêmes petits bouts de preuve et de se concentrer sur l'aspect déduction naturelle plutôt que sur l'aspect Coq.

Bien entendu, toutes les tactiques sus-mentionnées sont perfectibles, notamment dans la cherche des instances. L'algorithme d'unification nous assure, en effet, que le problème de savoir par quels termes instancier une règle de déduction naturelle est décidable. Cependant, il faut garder en tête la vocation pédagogique de ce projet, qui nous a conduits à souhaiter laisser l'utilisateur choisir les instances lui-même.

1.2.3 Pour utiliser des lemmes auxiliaires

Le principal problème posé par l'usage de lemmes auxiliaires est le fait qu'ils sont toujours de la forme `IsTheorem`, alors que le corps d'une preuve fait plutôt intervenir le prédicat `Pr`. La solution proposée pour palier à cela a été d'établir une sorte de *modus ponens* méta-théorique sur `IsTheorem`. Plus précisément, on a démontré

```
Lemma ModusPonens th :
  forall A B , IsTheorem th (A -> B) -> IsTheorem th A ->
               IsTheorem th B.
```

Grâce à ce résultat, la preuve d'un théorème ϕ utilisant des lemmes auxiliaires se fait en deux temps : dans un premier temps, on montre que ϕ est effectivement impliqué par tous les lemmes auxiliaires $\psi_1, \ldots, \psi_n$ que l'on souhaite utiliser (c'est là que repose toute la preuve), et dans un second temps, on utilise `ModusPonens` pour montrer que $\psi_1, \ldots, \psi_n$ sont bien des théorèmes eux aussi.

Le théorème `ModusPonens` nous a semblé être la solution la plus susceptible de passer à l'échelle, car il permet de fusionner les listes d'axiomes nécessaires à l'établissement d'un résultat de la forme `IsTheorem` au niveau méta-théorique. L'utilisateur n'a alors plus besoin de s'occuper des axiomes nécessaires à la démonstration des lemmes auxiliaires et peut se concentrer, sans peur et sans reproche, sur la preuve du théorème principal.

2 Théorie des ensembles de Zermelo-Fraenkel

Cette section est relative aux choix d'encodage des axiomes de ZF, aux preuves de quelques résultats dans cette théorie ainsi qu'à l'usage que l'on a pu faire des tactiques créées pour l'arithmétique de Peano.

Pour justifier l'utilité d'une théorie non-triviale des ensembles, nous avons commencé par montrer l'incohérence de la théorie naïve des ensembles qui ne contient qu'un schéma d'axiomes de compréhension non-restreint :

$$\forall x_1, \ldots, \forall x_n, \exists a, \forall y \cdot y \in a \Leftrightarrow A$$

pour toute formule A dont toutes les variables libres appartiennent à $\{x_1, \ldots, x_n, y\}$. Pour ce faire, nous avons établi le lemme suivant, qui n'est rien d'autre que le paradoxe du Russell.

```
Lemma Russell : Pr J ([ ∃∀ (#0 ∈ #1 <-> ~(#0 ∈ #0)) ] ⊢ False).
```

On remarque au passage que $\exists a, \forall y \cdot y \in a \Leftrightarrow \neg y \in y$ est bien une instance du schéma d'axiomes de compréhension non-restreint, puisqu'il suffit de poser $A := \neg y \in y$.

2.1 Encodage des axiomes et méta-théorèmes

Comme pour PA, nous avons fait le choix d'inclure les axiomes de l'égalité dans ceux de ZF. La liste exacte des axiomes que l'on a considérés se trouve en page 30 dans [5].

La différence majeure avec PA est que les axiomes de ZF contiennent un grand nombre de quantificateurs. Cette différence était particulièrement frappante dans les schémas d'axiomes de séparation et de remplacement, pour lesquels nous avons été obligés de créer un nouvel opérateur `lift_above` (renommé `lift` par la suite) afin d'ajuster convenablement les indices de de Bruijn d'une formule A quelconque.

Le principal écueil rencontré dans l'encodage des axiomes résidait donc dans l'usage des indices de de Bruijn. En effet, outre les problèmes d'ajustement susmentionnés, l'usage d'indices à la place de variables nommées entraîne inévitablement un grand nombre de typos qu'il est extrêmement ardu de repérer sans vraiment éprouver la théorie. À ce stade, l'équivalence déjà établie par Pierre Letouzey entre les formulations locally-nameless et nommées aurait pu se révéler salutaire, car les axiomes auraient été éminemment plus facile à encoder dans cette seconde formulation. Il a par ailleurs été intéressant, afin de vérifier que les axiomes sont correctement encodés, de faire le lien avec des modèles de ZF, comme proposé dans [6].

L'autre difficulté rencontrée lors de cette phase d'axiomatisation est liée à l'analyse syntaxique de Coq, mais un usage intensif – voire excessif – des parenthèses permet de régler ce problème sans trop d'efforts.

À titre d'exemple, voici une typo détectée (due à un problème d'analyse syntaxique) puis corrigée dans l'axiome d'extensionnalité

$$\forall a, \forall b \cdot (\forall x \cdot (x \in a \Leftrightarrow x \in b) \Rightarrow a = b).$$

La forme initialement écrite était

```
Definition ext := ∀∀ (∀ (#0 ∈ #2 <-> #0 ∈ #1) -> #2 = #1).
```

ce qui s'écrirait en formulation nommée $\forall a, \forall b, \forall x \cdot (x \in a \Leftrightarrow x \in b) \Rightarrow a = b$. En d'autres termes, cela voudrait dire que si deux ensembles a et b ont un élément en commun alors $a = b$, ce qui rendrait la théorie incohérente (en plus de n'être pas du tout l'axiome d'exensionnalité au sens où on l'entend). Cette erreur a été aisément corrigée par

```
Definition ext := ∀∀ ((∀ #0 ∈ #2 <-> #0 ∈ #1) -> #1 = #0).
```

Une erreur semblable avait aussi été commise dans l'axiome de l'infini.

Une fois les axiomes de ZF encodés, il a fallu établir un certain nombre de méta-théorèmes nécessaire à la bonne définition de la théorie. En particulier, il a fallu établir le lemme suivant, qui permet de définir la signature de ZF :

```
Lemma WCAx A : IsAx A -> WC ZFSign A.
```

En premier lieu, ce résultat a été fort problématique, en raison des structures finies qu'il fait intervenir, et de l'incapacité des tactiques `intuition` ou `cbn` à gérer ces structures. Cependant, les lemmes établis par Pierre Letouzey plus tôt dans le développement ainsi que les théorèmes de la bibliothèque standard et du module Nat ont été d'une efficacité redoutable et ont permis d'obtenir une preuve d'une vingtaine de ligne, bien que cela ait plus fait appel à l'expérience et à la grande connaissance des librairies de Pierre Letouzey qu'à ma simple intuition mathématique.

2.2 Preuves exemples

Pour mettre à l'épreuve la théorie ZF fraîchement encodée, nous avons démontré un certain nombre de résultats classiques : l'existence de l'ensemble vide, des singletons et la construction de l'opérateur $\cup$.

```
Lemma emptyset : IsTheorem J ZF (∃∀ ~(#0 ∈ #1)).

Lemma singleton : IsTheorem J ZF (∀∃∀ (#0 ∈ #1 <-> #0 = #2)).

Lemma unionset : IsTheorem J ZF (∀∀∃∀ (#0 ∈ #1 <-> #0 ∈ #3 \/ #0 ∈ #2)).
```

Grâce au travail fait pour PA, ces preuves ont essentiellement ressemblé à des dérivations en déduction naturelle, et n'ont donc posé aucun problème majeur. En fait, Coq s'est révélé un atout précieux, notamment car il permet l'usage de règles gauches (dont l'admissibilité a bien sûr été préalablement établie) et le regroupement de morceaux de dérivation semblables grâce à la tactique `assert`, là où une dérivation papier nous aurait contraint à un copier-coller manuel.

2.3 Construction des entiers

Le but initialement fixé était de montrer un résultat du type ZF $\vdash$ PA. Cependant, par manque de temps et peut-être de courage face à l'ampleur de la tâche, qui aurait demandé d'établir un théorème de Skolem [4], nous nous somme rabattus sur un objectif plus modeste, à savoir mettre en œuvre la construction de von Neumann des entiers :

$$0 := \varnothing$$
$$\mathrm{Succ}(n) := \{n\} \cup n.$$

Comme nous avions déjà établi l'existence de l'ensemble vide, il ne nous restait plus qu'à montrer que, pour tout n, l'ensemble $n \cup \{n\}$ existait bien. Pour ce faire, nous avons défini un prédicat `succ`

```
Definition succ x y := ∀ (#0 ∈ lift 0 y <-> #0 = lift 0 x \/ #0 ∈ lift 0 x).
```

et démontré, grâce à `ModusPonens`, le lemme suivant :

```
Lemma Successor : IsTheorem J ZF (∀∃ succ (#1) (#0)).
```

Là encore, le travail effectué sur l'arithmétique de Peano a permis de faire en sorte que la preuve ressemble à une dérivation en déduction naturelle, avec en plus les avantages mentionnés en 2.2. De plus, puisque nous avions déjà démontré l'existence, pour tous a et b, du singleton $\{a\}$ ainsi que celle de l'union $a \cup b$, la preuve principale (regroupée dans un théorème auxiliaire où l'énoncé des lemmes utilisés est posé en impliquant du théorème à démontrer, comme vu en 1.2.3) tient en une trentaine de ligne.

Une fois démontrés ces résultats, la construction des entiers successifs est immédiate en utilisant le théorème de Skolem.

4. Dont nous ne disposions pas à l'époque, mais que Pierre Letouzey a démontré ensuite.

3 Lien avec le λ-calcul

Pour finir le stage, nous avons encodé un λ-calcul enrichi et démontré une correspondance de Curry-Howard, qui est une alternative possible aux preuves dans NJ_0.

Cette section reprend les concepts et les constructions du cours de λ-calcul de Jean Goubault-Larrecq [4]. Par ailleurs, on se placera toujours dans le cadre de la *logique propositionnelle intuitionniste*.

3.1 Choix d'encodage

Le choix général d'encodage a été d'implémenter un λ-calcul à la Curry, c'est-à-dire où l'on traite les termes usuels du λ-calcul, sans annotation de type. Les différences entre les syntaxes de Curry et de Church étant plutôt anecdotiques pour l'usage que nous souhaitions faire du λ-calcul, nous avons opté pour celle présentée dans [4].

L'ensemble des constructions utilisées est détaillé dans les sous-sections qui suivent, et repris en annexe E.

3.1.1 Logique minimale intuitionniste

Le λ-calcul simplement typé permet de simuler la logique minimale intuitionniste, i.e. la logique propositionnelle intuitionniste pourvue de l'implication.

Nous avons donc repris la syntaxe du λ-calcul pur et y avons ajouté celle des types simples : les variables de type et les types flèche.

```
Inductive term :=
  | Var : nat -> term
  | App : term -> term -> term
  | Abs : term -> term.

Inductive type :=
  | Arr : type -> type -> type
  | Atom : name -> type.
```

Les règles de dérivation sont au nombre de trois : une pour chaque construction syntaxique.

```
Inductive typed : context -> term -> type -> Prop :=
  | Type_Var : forall Γ τ n, nth_error Γ n = Some τ -> typed Γ (Var n) τ
  | Type_App : forall Γ u v σ τ, typed Γ u (Arr σ τ) -> typed Γ v σ -> typed Γ (App u v) τ
  | Type_Abs : forall Γ u σ τ,  typed (σ :: Γ) u τ -> typed Γ (Abs u) (Arr σ τ).
```

Concernant les contextes de typage, comme il est difficile de manipuler des listes d'association en Coq, nous avons décidé de donner au constructeur `Var` le type `nat -> term`, ce qui permet d'associer à chaque variable un indice. Cet indice est ensuite utilisé pour retrouver la variable dans le contexte de typage, qui ne sera donc qu'une liste de type, le type en première position étant associé à la première variable, etc.

3.1.2 Faux et constructeur ∇

Nous avons ensuite voulu enrichir notre λ-calcul pour y ajouter un type correspondant au faux. Pour ce faire, nous avons rajouté un constructeur de termes `Nabla`, correspondant essentiellement à une exception dans un programme, et un type `Bot`.

```
Inductive term :=
  ...
  | Nabla : term -> term.

Inductive type :=
  ...
  | Bot : type
```

Nous avons ensuite ajouté une règle de typage correspondant à l'élimination du $\bot$.

```
Inductive typed : context -> term -> type -> Prop :=
  ...
  | Type_Nabla : forall Γ u τ, typed Γ u Bot -> typed Γ (Nabla u) τ.
```

Il est à noter que le faux permet facilement de construire la négation, puisqu'il suffit de poser $\neg A := A \Rightarrow \bot$.

3.1.3 Conjonction : couples et projections

Nous avons ensuite voulu ajouter un opérateur $\wedge$. Pour cela, nous avons rajouté trois constructeurs syntaxiques aux termes : `Couple`, qui permet d'implémenter des couples de λ-termes, ainsi que des projections `Pi1` et `Pi2` (censées permettre d'accéder respectivement au premier et au second élément d'un couple). Quant aux types, nous les avons augmentés d'un constructeur `And`, dont le rôle est assez bien résumé par son nom.

```
Inductive term :=
  ...
  | Couple : term -> term -> term
  | Pi1 : term -> term
  | Pi2 : term -> term.

Inductive type :=
  ...
  | And : type -> type -> type.
```

Aux règles de dérivations, nous avons ajouté trois nouvelles règles : une d'introduction de la conjonction et deux d'élimination, une à gauche et une à droite.

```
Inductive typed : context -> term -> type -> Prop :=
  ...
  | Type_Couple : forall Γ u v σ τ, typed Γ u σ -> typed Γ v τ ->
                                    typed Γ (Couple u v) (And σ τ)
  | Type_Pi1 : forall Γ u σ τ, typed Γ u (And σ τ) -> typed Γ (Pi1 u) σ
  | Type_Pi2 : forall Γ u σ τ, typed Γ u (And σ τ) -> typed Γ (Pi2 u) τ
```

3.1.4 Disjonction : filtrage par motif et injections

Enfin, pour obtenir une logique propositionnelle intuitionniste complète, il ne nous restait qu'à avoir un opérateur $\vee$ [5].

Nous avons donc rajouté trois constructeurs aux termes : `Case`, qui simule un filtrage par motif à deux motifs, et deux injections `I1` et `I2` qui permettent d'accéder respectivement au premier et au second motif. Nous avons aussi ajouté un constructeur de type `Or`.

```
Inductive term :=
  ...
  | Case : term -> term -> term -> term
  | I1 : term -> term
  | I2 : term -> term.

Inductive type :=
  ...
  | Or : type -> type -> type.
```

Nous avons, pour finir, ajouté trois nouvelles règles de dérivation de typage : deux introductions du $\vee$ et une élimination.

```
Inductive typed : context -> term -> type -> Prop :=
  ...
  | Type_Case : forall Γ u v1 v2 τ1 τ2 σ, typed Γ u (Or τ1 τ2) -> typed (τ1 :: Γ) v1 σ ->
                                          typed (τ2 :: Γ) v2 σ -> typed Γ (Case u v1 v2) σ
  | Type_I1 : forall Γ u σ τ, typed Γ u σ -> typed Γ (I1 u) (Or σ τ)
  | Type_I2 : forall Γ u σ τ, typed Γ u τ -> typed Γ (I2 u) (Or σ τ).
```

Le λ-calcul ainsi créé possède autant de règle de typage que de constructeurs syntaxiques de termes et il est encore dirigé par la syntaxe, ce qui permet d'affirmer que les problèmes d'inférence et de vérification de type sont décidables. C'est dans cette propriété que réside tout l'intérêt de la correspondance de Curry-Howard.

3.2 Correspondance de Curry-Howard

Afin d'établir un lien avec la déduction naturelle, nous avons établi, sans grande difficulté, le résultat suivant :

5. Cette étape était nécessaire car, en logique intuitionniste, la règle de de Morgan $A \vee B \Leftrightarrow \neg(\neg A \wedge \neg B)$ devient fausse.

```
Theorem CurryHoward Γ u τ :
  typed Γ u τ -> Pr J (to_ctxt Γ ⊢ to_form τ).
```

L'opérateur `to_form` fait le lien – plutôt évident – entre les types et les formules, la seule subtilité étant que l'on transforme les variables de type en symboles de prédicat d'arité nulle. Quant à `to_ctxt`, il s'agit de l'application de `to_form` à tous les éléments d'une liste grâce à `List.map`.

Le théorème de Curry-Howard a ensuite été utilisé pour redémontrer le premier résultat du stage, à la savoir que le séquent $\vdash (\phi_1 \wedge \phi_2) \Rightarrow (\phi_1 \vee \phi_2)$ était dérivable.

```
Lemma ex : Pr J ([] ⊢ (Pred "A" [] /\ Pred "B" []) -> (Pred "A" [] \/ Pred "B" [])).
```

Pour ce faire, il a suffit d'exhiber le λ-terme témoin $\lambda x \cdot \iota_1 \pi_1 x$, qui est beaucoup plus aisé à manipuler qu'un arbre de dérivation complet. Notons au passage que le témoin contient en fait toute l'information de la preuve car le problème de la vérification de type est décidable dans le λ-calcul que nous avons construit.

Conclusion

Apports mutuels du stage

Ce stage m'a permis d'avoir un premier aperçu, quelque peu biaisé du fait des conditions sanitaires, de ce à quoi peut ressembler le quotidien d'un chercheur. J'ai notamment pu découvrir la suprême frustration que constitue le fait d'achopper pendant plusieurs jours sur un problème, qui se trouve ensuite balayé d'un revers de main par quelqu'un d'autre – en l'occurrence Pierre Letouzey – soit car cette personne est plus coutumière des outils utilisés, soit tout simplement parce qu'elle trouve la bonne idée qui nous échappait depuis le début. J'ai aussi eu la joie de découvrir la satisfaction que procure un travail qui avance à bon rythme et qui prend une tournure plus intéressante que prévu, satisfaction qui efface sans nul doute l'amertume des échecs passés.

Par ailleurs, j'ai pu me rendre compte à quel point l'activité de recherche scientifique est, pour reprendre les mots de Pierre Letouzey, un « artisanat », et qu'il me reste encore un long chemin à parcourir avant de pouvoir devenir moi-même un chercheur autonome, travail pour lequel ma motivation s'est encore développée pendant ce stage.

Enfin, j'ai eu la chance de travailler sur un sujet qui m'intéresse énormément, la logique et les assistants de preuve, auprès d'un connaisseur du domaine, que je ne remercierai jamais assez pour l'infinie patience dont il a fait preuve avec moi. Bien qu'assez proche du cours de logique que j'ai suivi au second semestre, ce stage m'a permis d'approfondir ma compréhension des concepts élémentaires de la logique, ainsi que d'améliorer ma maîtrise de Coq, un outil que je prenais pour ésotérique et peu utilisable en pratique, mais qui s'est révélé un allié puissant et parfois salutaire.

De mon côté, j'espère avoir contribué à rendre le projet `NatDed` plus utilisable dans le cadre d'un cours d'introduction à la logique, d'une part en le rendant plus ergonomique (et donc plus aisé à utiliser en temps réel), et d'autre part en lui ajoutant de nouvelles perspectives d'illustration, notamment en ce qui concerne ZF et le λ-calcul.

J'espère aussi que mon travail aura permis à Pierre Letouzey de mieux cerner les limites du projet, et que cela l'aidera s'il souhaite poursuivre son travail sur `NatDed`.

Prolongements envisageables

Le sujet du stage était volontairement long, et plusieurs points n'ont pas pu être abordés. De plus, au cours du stage, plusieurs perspectives non explorées ont été évoquées, voici donc une liste non-exhaustive de prolongements envisageables de ce stage.

- ★ Comme mentionné plusieurs fois dans le rapport, l'encodage *locally-nameless* des formules logique présente un certain nombre de défauts, dont la lisibilité. Il serait dès lors intéressant, dans l'optique d'un cours magistral, d'utiliser un encodage nommé pour toutes les variables, qu'elles soient libres ou liées. Ce travail a déjà été amorcé puisque Pierre Letouzey a démontré un théorème d'équivalence entre les deux formulations, il ne resterait donc plus qu'à traduire tous les énoncés des résultats que l'on veut présenter.
- ★ Les tactiques définies au cours du travail sur l'arithmétique de Peano gagneraient peut-être à être totalement automatisées grâce à l'algorithme d'unification, même si exhiber des témoins de $\forall_{\mathrm{E}}$ et $\exists_{\mathrm{I}}$ a également son intérêt pédagogique.
- ★ Ce stage s'est principalement concentré sur des aspects syntaxiques. Il pourrait être intéressant de creuser d'avantage les aspects méta-théoriques et sémantiques de la logique, par exemple en implémentant la sémantique de Kripke pour la logique intuitionniste, ou bien en creusant les modèles de ZF grâce à [6].
- ★ Nous avons montré un sens de la correspondance de Curry-Howard – celui qui permet de passer d'une dérivation de typage à une preuve – mais la réciproque mérite aussi d'être considérée, par exemple pour établir des résultats de non-prouvabilité.
- ★ Enfin, si le théorème de complétude a déjà été établi par Pierre Letouzey, nous n'avons en revanche pas eu le temps de nous pencher sur les théorèmes d'incomplétude de Gödel, dont la démonstration permettrait de proposer un cadre unifié à tout l'encodage `NatDed`.

Références

[1] Henk Barendregt. *The Lambda Calculus, its Syntax and Semantics*. College Publications, 1980.

[2] René Cori and Daniel Lascar. *Logique mathématique, tome 2*. Dunod, 2019.

[3] René David, Karim Nour, and Christophe Raffalli. *Introduction à la logique, théorie de la démonstration*. Dunod, 2019.

[4] Jean Goubault-Larrecq. Lambda-calcul et logique informatique, aspects logiques, 1999. `http://www.lsv.fr/~goubault/Lambda/types.pdf`.

[5] Alexandre Miquel. Eléments de logique du premier ordre, 2008. `https://www.irif.fr/~letouzey//preuves/cours.pdf`.

[6] Benjamin Werner. Sets in types, types in sets, 1997. `http://www.lix.polytechnique.fr/Labo/Benjamin.Werner/publis/tacs97.pdf`.

A Cheatsheet pour git

Utilisation de base de git

Samuel Ben Hamou

10 juillet 2020

Résumé

Un petit guide pour le total néophyte que je suis avec git. J'y recence des GUI d'intérêt général ainsi que les commandes que l'on m'a présentées comme étant les plus utiles. Enfin, quelques liens utiles vers des cheat sheets provenant directement de GitHub.

1 GUI utiles

- `git gui` permet d'éditer les nouveaux commits et, tant que la branche courante n'a pas été mise à jour sur le serveur, il permet aussi de corriger les anciens commits.
- `gitk` permet de visualiser les branches d'édition, et éventuellement de gérer des commits simples.

2 Lignes de commande

2.1 Commandes d'utilité générale

- `git branch [branch name]` crée une nouvelle branche. Pour accéder à cette nouvelle branche (ce n'est pas le cas par défaut), utiliser `git checkout [branch name]`.
- `git branch` affiche les branches du projet courant.
- `git status` affiche l'historique des modifications du projet et indique s'il est à jour par rapport au serveur.
- `git gc` permet d'effacer définitivement les fichiers et branches inutiles.

2.2 Interaction avec le serveur

- `git clone [deposite name] [destination]` permet d'importer un projet.
- `git push` enregistre les modifications locales sur le serveur *pour la branche courante seulement*.
- `git pull` permet de mettre à jour le projet local depuis le serveur.

2.3 Gestion des commit

- `git commit -a -m "text"` commite la dernière action effectuée avec `text`.
- `git revert [pointer]` pose un anti-commit (c'est à dire qu'il rajoute un commit sur une action déjà commitée pour annuler les changements du commit pointé). Le pointeur `[HEAD^]` correspond au dernier commit.
- `git reset [pointer]` fait la même chose que `git reset` mais en modifiant le commit pointé.
- `git checkout -f [file]` permet d'annuler les actions effectuées sur le fichier `[file]` depuis le dernier commit.

3 Liens utiles

Cheat sheet 1. Très générale, sur les branches et les commit : `https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf`

Cheat sheet 2. Un peu plus complète : `https://education.github.com/git-cheat-sheet-education.pdf`

Cheat sheet 3. Une troisième, qui ressemble à la deuxième : `https://github.github.com/training-kit/downloads/fr/github-git-cheat-sheet.pdf`

B Définition des formules

```
Inductive term :=
  | FVar : variable -> term (** Free variable (global name) *)
  | BVar : nat -> term (** Bounded variable (de Bruijn indices) *)
  | Fun : function_symbol -> list term -> term.

Inductive formula :=
  | True
  | False
  | Pred : predicate_symbol -> list term -> formula
  | Not : formula -> formula
  | Op : op -> formula -> formula -> formula
  | Quant : quant -> formula -> formula.

Definition Iff a b := Op And (Op Impl a b) (Op Impl b a).

Notation "~ f" := (Not f) : formula_scope.
Infix "/\" := (Op And) : formula_scope.
Infix "\/" := (Op Or) : formula_scope.
Infix "->" := (Op Impl) : formula_scope.
Infix "<->" := Iff : formula_scope.

Notation "# n" := (BVar n) (at level 20, format "# n") : formula_scope.

Notation "∀ A" := (Quant All A)
 (at level 200, right associativity) : formula_scope.
Notation "∃ A" := (Quant Ex A)
 (at level 200, right associativity) : formula_scope.

Definition context := list formula.

Inductive sequent :=
| Seq : context -> formula -> sequent.
```

C Définition du prédicat de prouvabilité `Pr`

```
Inductive Pr (l:logic) : sequent -> Prop :=
 | R_Ax Γ A : In A Γ -> Pr l (Γ ⊢ A)
 | R_Tr_i Γ : Pr l (Γ ⊢ True)
 | R_Fa_e Γ A : Pr l (Γ ⊢ False) ->
                Pr l (Γ ⊢ A)
 | R_Not_i Γ A : Pr l (A::Γ ⊢ False) ->
                 Pr l (Γ ⊢ ~A)
 | R_Not_e Γ A : Pr l (Γ ⊢ A) -> Pr l (Γ ⊢ ~A) ->
                 Pr l (Γ ⊢ False)
 | R_And_i Γ A B : Pr l (Γ ⊢ A) -> Pr l (Γ ⊢ B) ->
                   Pr l (Γ ⊢ A/\B)
 | R_And_e1 Γ A B : Pr l (Γ ⊢ A/\B) ->
                    Pr l (Γ ⊢ A)
 | R_And_e2 Γ A B : Pr l (Γ ⊢ A/\B) ->
                    Pr l (Γ ⊢ B)
 | R_Or_i1 Γ A B : Pr l (Γ ⊢ A) ->
                   Pr l (Γ ⊢ A\/B)
 | R_Or_i2 Γ A B : Pr l (Γ ⊢ B) ->
                   Pr l (Γ ⊢ A\/B)
 | R_Or_e Γ A B C :
     Pr l (Γ ⊢ A\/B) -> Pr l (A::Γ ⊢ C) -> Pr l (B::Γ ⊢ C) ->
     Pr l (Γ ⊢ C)
 | R_Imp_i Γ A B : Pr l (A::Γ ⊢ B) ->
```

```
                    Pr l (Γ ⊢ A->B)
| R_Imp_e Γ A B : Pr l (Γ ⊢ A->B) -> Pr l (Γ ⊢ A) ->
                   Pr l (Γ ⊢ B)
| R_All_i x Γ A : ~Names.In x (fvars (Γ ⊢ A)) ->
                   Pr l (Γ ⊢ bsubst 0 (FVar x) A) ->
                   Pr l (Γ ⊢ ∀A)
| R_All_e t Γ A : Pr l (Γ ⊢ ∀A) -> Pr l (Γ ⊢ bsubst 0 t A)
| R_Ex_i t Γ A : Pr l (Γ ⊢ bsubst 0 t A) -> Pr l (Γ ⊢ ∃A)
| R_Ex_e x Γ A B : ~Names.In x (fvars (A::Γ⊢B)) ->
    Pr l (Γ ⊢ ∃A) -> Pr l ((bsubst 0 (FVar x) A)::Γ ⊢ B) ->
    Pr l (Γ ⊢ B)
| R_Absu Γ A : l=Classic -> Pr l (Not A :: Γ ⊢ False) ->
                 Pr l (Γ ⊢ A).
```

D Preuve de `ZeroRight` : avant-après

La preuve originelle était :

```
Lemma ZeroRight :
  IsTheorem J PeanoTheory
            (∀ (#0 = #0 + Zero)).
Proof.
  unfold IsTheorem.
  split.
  + unfold Wf. split; [ auto | split; auto ].
  + exists ((PeanoAx.induction_schema (#0 = #0 + Zero))::axioms_list).
    split.
    - apply Forall_forall.
      intros. destruct H.
      * simpl. unfold IsAx.
        right. exists (#0 = #0 + Zero).
        split; [ auto | split ; [ auto | auto ] ].
      * simpl. unfold IsAx. left. exact H.
    - apply R_Imp_e with
      (A := (nForall (Nat.pred (level (# 0 = # 0 + Zero)))
       ((∀ bsubst 0 Zero (# 0 = # 0 + Zero)) /\
        (∀ # 0 = # 0 + Zero -> bsubst 0 (Succ (# 0)) (# 0 = # 0 + Zero))))).
      * apply R_Ax. unfold induction_schema. apply in_eq.
      * simpl. apply R_And_i. cbn.
        change (Fun "O" []) with Zero.
        apply R_All_i with (x := "x").
        ++ compute. inversion 1.
        ++ cbn. change (Fun "O" []) with Zero.
           eapply R_Imp_e. set (hyp := (_ -> _)%form).
           assert ( sym : Pr J (hyp::axioms_list ⊢ ∀∀ (#1 = #0 -> #0 = #1))).
           { apply R_Ax. compute; intuition. }
           apply R_All_e with (t := Zero + Zero) in sym. cbn in sym.
           apply R_All_e with (t := Zero) in sym. cbn in sym. exact sym.
           -- reflexivity.
           -- reflexivity.
           -- set (hyp := (_ -> _)%form). change (Fun "O" []) with Zero.
              change (Zero + Zero = Zero) with (bsubst 0 Zero (Zero + #0 = #0)).
              apply R_All_e. reflexivity.
              apply R_Ax. compute; intuition.
       ++ cbn. change (Fun "O" []) with Zero.
          apply R_All_i with (x := "y").
          -- compute. inversion 1.
          -- cbn. change (Fun "O" []) with Zero.
             apply R_Imp_i. set (H1 := FVar _ = _). set (H2 := _ -> _).
             assert (hyp : Pr J
                           (H1 :: H2 :: axioms_list ⊢ Fun "S" [FVar "y"] =
```

```
                              Fun "S" [FVar "y" + Zero] /\
                              Fun "S" [FVar "y" + Zero] = Fun "S" [FVar "y"] + Zero)).
                { apply R_And_i.
                  - assert (AX4 : Pr J (H1 :: H2 :: axioms_list ⊢ ax4)).
                    { apply R_Ax. compute; intuition. }
                    apply R_Imp_e with (A := (FVar "y" = FVar "y" + Zero)%form);
                      [ | apply R_Ax; compute; intuition ].
                    unfold ax4 in AX4. apply R_All_e with (t := FVar "y") in AX4; [ | auto ].
                    apply R_All_e with (t := FVar "y" + Zero) in AX4; [ | auto ].
                    cbn in AX4. exact AX4.
                  - apply R_Imp_e with (A := Fun "S" [FVar "y"] + Zero = Fun "S" [FVar "y" + Zero]).
                    + assert (AX2 : Pr J (H1 :: H2 :: axioms_list ⊢ ax2)).
                      { apply R_Ax. compute; intuition. }
                      unfold ax2 in AX2.
                      apply R_All_e with (t := Fun "S" [FVar "y"] + Zero) in AX2; [ | auto ].
                      apply R_All_e with (t := Fun "S" [FVar "y" + Zero]) in AX2; [ | auto ].
                      cbn in AX2. exact AX2.
                    + assert (AX10 : Pr J (H1 :: H2 :: axioms_list ⊢ ax10)).
                      { apply R_Ax. compute; intuition. }
                      unfold ax10 in AX10.
                      apply R_All_e with (t:= FVar "y") in AX10; [ | auto ].
                      apply R_All_e with (t := Zero) in AX10; [ | auto ].
                      cbn in AX10. exact AX10. }
                apply R_Imp_e with
                (A :=  Fun "S" [FVar "y"] = Fun "S" [FVar "y" + Zero] /\
                       Fun "S" [FVar "y" + Zero] = Fun "S" [FVar "y"] + Zero).
                ** assert (AX3 : Pr J (H1 :: H2 :: axioms_list ⊢ ax3)).
                   { apply R_Ax. compute; intuition. }
                   unfold ax3 in AX3.
                   apply R_All_e with (t:= Fun "S" [FVar "y"]) in AX3; [ | auto ].
                   apply R_All_e with (t := Fun "S" [FVar "y" + Zero]) in AX3; [ | auto ].
                   apply R_All_e with (t := Fun "S" [FVar "y"] + Zero) in AX3; [ | auto ].
                   cbn in AX3. exact AX3.
                ** exact hyp.
 Qed.
```

et en fin de stage, elle était devenue

```
Lemma ZeroRight :
  IsTheorem J PeanoTheory
            (∀ (#0 = #0 + Zero)).
Proof.
 thm.
 rec.
 + sym.
   inst_axiom ax9 [Zero].
 + app_R_All_i "y" y.
   apply R_Imp_i. set (H1 := _ = _).
   sym.
   trans (Succ (y + Zero)).
   - inst_axiom ax10 [y; Zero].
   - ahered.
     sym.
     apply R_Ax.
     apply in_eq.
Qed.
```

E Syntaxe et règles de typage du λ-calcul utilisé

La syntaxe des types est la suivante :

$\sigma, \tau, \ldots ::= a, b, \ldots$	variables de type
$\mid \sigma \Rightarrow \tau$	type flèche
$\mid \bot$	faux
$\mid \sigma \wedge \tau$	type conjonction
$\mid \sigma \vee \tau$	type disjonction

et celle des λ-termes est

$u, v, \ldots ::= x, y, \ldots$	variables
$\mid uv$	application
$\mid \lambda x \cdot u$	abstraction
$\mid \nabla u$	nabla
$\mid \langle u, v \rangle$	couple
$\mid \pi_1 u$	première projection
$\mid \pi_2 u$	seconde projection
$\mid \texttt{case}\, u\, \texttt{of}\, \iota_1 x_1 \mapsto v_1 \mid \iota_2 x_2 \mapsto v_2$	filtrage par motif
$\mid \iota_1 u$	première injection
$\mid \iota_2 u$	seconde injection.

Le constructeur ∇ correspond au `False_rec` de Coq.

Les règles de typage sont :

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{var} \qquad \frac{\Gamma \vdash u : \sigma \Rightarrow \tau \qquad \Gamma \vdash v : \sigma}{\Gamma \vdash uv : \tau}\ \text{app} \qquad \frac{\Gamma, x : \sigma \vdash u : \tau}{\Gamma \vdash \lambda x \cdot u : \sigma \Rightarrow \tau}\ \text{abs}$$

$$\frac{\Gamma \vdash u : \bot}{\Gamma \vdash \nabla u : \tau}\ \nabla$$

$$\frac{\Gamma \vdash u : \sigma \qquad \Gamma \vdash v : \tau}{\Gamma \vdash \langle u, v \rangle : \sigma \wedge \tau}\ \text{paire} \qquad \frac{\Gamma \vdash u : \sigma \wedge \tau}{\Gamma \vdash \pi_1 u : \sigma}\ \pi_1 \qquad \frac{\Gamma \vdash u : \sigma \wedge \tau}{\Gamma \vdash \pi_2 u : \tau}\ \pi_2$$

$$\frac{\Gamma \vdash u : \sigma \vee \tau \qquad \Gamma, x_1 : \sigma \vdash v_1 : \mu \qquad \Gamma, x_2 : \tau \vdash v_2 : \mu}{\Gamma \vdash \texttt{case}\, u\, \texttt{of}\, \iota_1 x_1 \mapsto v_1 \mid \iota_2 x_2 \mapsto v_2 : \mu}\ \text{pattern}$$

$$\frac{\Gamma \vdash u : \sigma}{\Gamma \vdash \iota_1 u : \sigma \vee \tau}\ \iota_1 \qquad \frac{\Gamma \vdash u : \tau}{\Gamma \vdash \iota_2 u : \sigma \vee \tau}\ \iota_2$$