Coopération d'outils de preuve interactifs et automatiques

Nicolas Ayache

Stage de Master M2

Laboratoire : LRI – CNRS 8623, Université Paris Sud, 91405 ORSAY Cedex
Directeur de laboratoire : M. Beaudouin-Lafon (mbl@lri.fr)
Équipe : Démonstration et Programmation (http ://www.lri.fr/demons/)
Directeur de stage : J.-C. Filliâtre (filliatr@lri.fr)

30 mars 2006

Table des matières

Introduction

Depuis la fin du siècle dernier, l'essor de l'informatique a été tel qu'il existe à présent des logiciels embarqués et critiques dont peut dépendre la vie d'hommes et de femmes. Ainsi, il est apparu un réel besoin de prouver que les programmes écrits sont corrects.

Il n'est donc pas étonnant que des assistants à la démonstration soient utilisés afin de se servir de la puissance calculatoire des ordinateurs pour nous aider à vérifier des programmes.

Typiquement, il existe deux grandes familles d'assistants de preuves sur ordinateur : les prouveurs interactifs et les prouveurs automatiques.

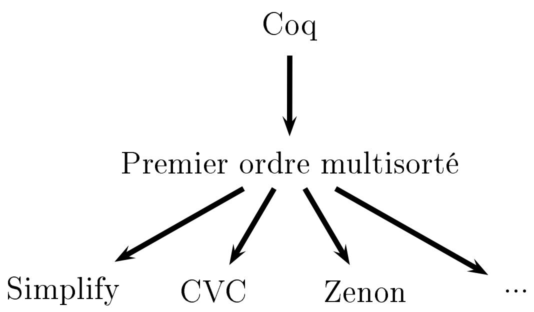
Les logiciels qui font partis de la première catégorie ont pour avantage de s'appuyer sur un langage mathématique très expressif, mais ne laissant place qu'à une très faible dose d'automatisme. L'utilisateur de tels logiciels devra en effet expliciter toutes les étapes d'une démonstration.

Les logiciels de la seconde catégorie, en revanche, sont entièrement automatiques. L'utilisateur donne une requête — une proposition à prouver — et le programme s'efforce alors d'en trouver une preuve ou une réfutation. Cependant, le langage mathématique sur lequel s'appuie ces logiciels est très peu expressif. On ne peut y exprimer qu'une partie trop restreinte des énoncés mathématiques envisageables pour pouvoir se passer des prouveurs interactifs.

Lorsque l'on veut s'atteler à la tâche ardue qu'est la certification d'un programme, on se heurte rapidement aux limites des systèmes automatiques et on est donc amené à utiliser les deux familles parallèlement : les logiciels ne collaborent pas entre eux. On peut, par exemple, laisser à un prouveur automatique le travail à effectuer et, s'il échoue, prendre la preuve en main à l'aide d'un prouveur interactif. Le fait est que lors d'une preuve interactive, on se retrouve bien souvent confronté à des sous-problèmes assez triviaux que les prouveurs automatiques peuvent résoudre.

Le but de ce projet est de montrer qu'il existe une traduction de la logique implantée par un prouveur interactif particulier, le système Coq [7], vers celle implantée par les prouveurs automatiques, qui soit correcte du point de vue de la prouvabilité, et alors de la mettre en pratique : on pourra ainsi appeler au sein même de Coq, de façon interactive, un prouveur automatique qui essaiera de résoudre le problème courant. Notre projet s'appuie sur des travaux similaires déjà effectués. Ainsi, [15] présente l'intégration de CVC [2] dans Isabelle [4] et [20] celle de JProver [5] dans NuPRL [6] en particulier, même si l'intégration est plus générale. En revanche, d'un point de vue théorique notre travail se rapproche de [16] qui traite le cas d'une logique de premier ordre clausale et de [11] qui est l'article le plus proche de notre travail.

Comme nous l'avons dit, les problèmes de premier ordre ne sont pas décidables. Les prouveurs automatiques existants aujourd'hui, comme Simplify [9], CVC Lite [3], Zenon [13] ou encore haRVey [19], s'en sortent différemment selon les cas. Un prouveur peut échouer sur un problème alors qu'un autre l'a résolu aisément, et la situation peut s'inverser sur un autre problème. Ceci est dû au fait qu'ils n'implantent pas les mêmes méthodes et les mêmes heuristiques. Aussi, pour profiter au maximum des capacités des procédures de décision, nous devions pouvoir « brancher » simplement de nouveaux prouveurs automatiques à Coq. Les syntaxes des prouveurs automatiques sont différentes les unes des autres. Donc, plutôt que de traduire directement un séquent Coq vers Simplify ou Zenon, nous avons choisi de « factoriser » la traduction vers une logique de premier ordre multisortée et seulement alors de traduire le séquent de cette logique vers celles des prouveurs. Puisque les prouveurs automatiques implantent déjà une logique du premier ordre, la dernière étape est constituée essentiellement de traduction mot-à-mot. Ainsi le branchement d'un nouveau prouveur se fait très simplement dès lors que l'on en connaît la syntaxe. Le schéma suivant résume les étapes de traduction d'un séquent Coq vers différents prouveurs automatiques :



Si nous nous écartons donc en pratique du travail réalisé dans [17] ou [18], une partie de la théorie reste la même. En effet, le but recherché dans ces articles est d'écrire une procédure de décision au sein d'assistants à la démonstration d'ordre supérieur. Ceci n'est pas notre but mais la traduction reste au cœur du sujet.

Le projet présente tout d'abord les logiques mises en jeu : celle des prouveurs automatiques — la logique du premier ordre multisortée — puis celle de Coq — le Calcul des Constructions Inductives. Ensuite, nous donnons la traduction attendue d'une logique vers l'autre, et nous en montrons la correction.

Finalement, nous finissons par présenter l'implantation de la traduction, avec ses spécificités, ses résultats expérimentaux et le travail qui reste à effectuer pour rendre le projet encore plus complet.

Je tiens à remercier un grand nombre de personnes pour avoir participé ou pour m'avoir aidé à réaliser ce projet. En premier plan se trouve Jean-Christophe Filliâtre, mon directeur de stage, ainsi que l'équipe Démons du LRI au sein de laquelle j'ai évolué ces six derniers mois. J'y ai beaucoup appris autant sur le plan théorique et pratique de ma spécialité que sur le plan humain. Je remercie encore une fois Jean-Christophe Filliâtre mais aussi Alexandre Miquel et Roberto Di Cosmo qui ont été les professeurs dont j'ai le plus appris durant mes cinq années d'études supérieures, et qui ont su me communiquer leur passion au point qu'elle devienne la mienne. Enfin je remercie des personnes qui me sont particulièrement chères et sans qui je ne serais pas la personne que je suis aujourd'hui : mes parents, Vincent Dupont, Nicolas Bouquet, Catherine Cucciardi et leur fille — ma filleule — Julie Bouquet, Pierre et Antonin Vacheret-Gallois.

Chapitre 1

Supports Logiques

1.1 Logique du premier ordre multisortée

La présentation de cette logique reprend celle donnée dans le Chapitre 1 de [14].

1.1.1 Les expressions

Un langage logique permet de définir des objets (un entier, une personne, une liste de choses à faire, ...) et d'exprimer des phrases quant à ces objets (il y a une infinité d'entiers, tout le monde — sous-entendu toutes les personnes — est soit une femme soit un homme, j'ai plus d'une chose à faire, ...). Une expression qui désigne un objet est appelée un *terme*, une expression qui permet d'énoncer une phrase est une *proposition*.

Un langage logique est constitué de variables (objets de base), de symboles de fonction (qui permettent de construire de nouveaux objets), et de symboles de prédicat (qui permettent de former les phrases de base). Les variables doivent être en nombre infini ; on appellera Var l'ensemble des variables.

De plus, à chaque symbole de fonction et de prédicat, on associe son nombre d'arguments ou *arité* : c'est le nombre d'objets dont ce symbole traite.

Exemple : Le langage de l'arithmétique est constitué
- de variables x, x_0, y, ... ;
- des symboles de fonctions 0 (d'arité 0), S (d'arité 1) et $+$ et $*$ (d'arité 2) ;
- du symbole de prédicat $=$ d'arité 2.

Les termes sont formés par les deux règles suivantes :
- les variables sont des termes ;
- si f est un symbole de fonction d'arité n et t_1, ..., t_n sont des termes, alors l'expression $f(t_1, ..., t_n)$ est un terme.

Lorsque l'on forme un ensemble comme ce dernier par récurrence, nous utiliserons en plus de la définition « en français » une écriture plus concise. Ici, l'ensemble t des termes sera décrit comme suit :

$$\begin{array}{rl} t ::= & x \text{ où } x \in Var \\ | & f(t, ..., t) \text{ pour tout symbole de fonction } f. \end{array}$$

Attention : Les t de la règle $f(t, ..., t)$ ne sont pas obligatoirement les mêmes. Ce sont juste des éléments de l'ensemble des termes qui peuvent être différents.

Exemple : en arithmétique, les expressions suivantes sont des termes :
0, $S(0)$, $S(S(0))$, $+(0,0)$, x, $+(*(x, S(S(0))), S(y))$, ...

Le moyen le plus simple de former une proposition (une phrase) est d'appliquer un symbole de prédicat à un nombre de termes égal à son arité ; on peut, par exemple, former ainsi la proposition $=(+(S(0), S(0)), S(S(0)))$.

Par soucis de lisibilité, les symboles *binaires*, c'est-à-dire d'arité 2 sont représentés de façon infixe, et non plus préfixe. Par la suite, on écrira donc l'expression $=(+(S(0), S(0)), S(S(0)))$ plutôt sous la forme $S(0) + S(0) = S(S(0))$.

Mais on a également besoin de former des propositions plus complexes. Pour cela, on utilise les connecteurs $\wedge$ (et), $\vee$ (ou), $\rightarrow$ (implique), $\neg$ (non), $\top$ (vrai) et $\bot$ (faux) ainsi que les quantificateurs $\forall$ (pour tout) et $\exists$ (il existe).

Les quantificateurs servent à exprimer que tous les objet du discours vérifient une propriété ($\forall$), ou qu'au moins un objet vérifie une propriété ($\exists$), mais sans spécifier lequel.

Par exemple, la proposition $\forall x : Int, (x > 0) \rightarrow x \neq 0$ signifie "Pour tout entier x, si x est strictement plus grand que 0, alors x est différent de 0".

Nous avons vu un exemple de proposition, voyons à présent quelles sont les règles qui régissent leurs formations :

- si P est un symbole de prédicat d'arité n, et t_1, ..., t_n sont des termes, alors l'expression $P(t_1, ..., t_n)$ est une proposition ;
- si t_1 et t_2 sont des termes, alors $t_1 = t_2$ est une proposition ;
- $\top$ et $\bot$ sont des propositions et si A et B sont des propositions alors $\neg A$, $A \wedge B$, $A \vee B$ et $A \rightarrow B$ sont des propositions ;
- si A est une proposition, x une variable et S une sorte, alors $\forall x : S, A$ et $\exists x : S, A$ sont des propositions.

$$
\begin{array}{rll}
P ::= & Pred(t, ..., t) & \text{pour tout symbole de prédicat } Pred \\
| & t = t & \\
| & \top & \\
| & \bot & \\
| & \neg P & \\
| & P \wedge P & \\
| & P \vee P & \\
| & P \rightarrow P & \\
| & \forall x : S, P & \text{où } x \in Var \text{ et } S \in Sort \\
| & \exists x : S, P & \text{où } x \in Var \text{ et } S \in Sort.
\end{array}
$$

On voit apparaître dans la grammaire des quantificateurs la notion de *sorte*.

En effet, dans notre théorie, nous seront amenés à distinguer plusieurs sortes d'objets. Par exemple, si on a des symboles de fonction d'arité nulle *italien*, *allemand*, *Italie* et *Allemagne* et un prédicat binaire *langue*, on peut former les termes $langue(Italie, italien)$, $langue(Allemagne, allemand)$, mais aussi le terme $langue(Allemagne, Italie)$ qui semble dénué de sens.

Pour contrôler ce genre d'absurdité dans notre langage, on se donne un ensemble potentiellement infini $Sort = \{S_1, S_2, ...\}$ de sortes. Alors, on associe aux symboles de fonctions une liste ordonnée de sortes dont la longueur est égale à son arité et dont les éléments sont respectivement la sorte de ses arguments. Enfin, on lui associe également une sorte qui représente la sorte du résultat de la fonction. On procède de même avec les symboles de prédicat, à ceci près que l'on ne leur associe que la sorte de leurs arguments. Ainsi, lorsqu'un symbole ne se voit pas associer de sorte de résultat, on en déduit que c'est un symbole de prédicat et qu'il permet d'exprimer une proposition, une phrase.

Formellement, on définit les listes de sortes, et plus généralement les listes d'élément d'un ensemble E ainsi :

- la liste vide [] est une liste d'éléments de E ;
- si $e \in E$ et si l est une liste d'éléments de E, alors $e :: l$ est une liste d'éléments de E, dont le premier élément est e et le reste l.

$$\begin{array}{rl} l ::= & [] \\ | & e :: l \text{ où } e \in E. \end{array}$$

Exemple : La liste des chiffres entiers pairs dans l'ordre croissant est $0 :: 2 :: 4 :: 6 :: 8 :: []$.

Toujours par soucis de lisibilité, nous préférerons écrire les éléments d'une liste entre crochets et les séparer par des point-virgules. $0 :: 2 :: 4 :: 6 :: 8 :: []$ est équivalent à $[0 ; 2 ; 4 ; 6 ; 8]$.

Si on reprend l'exemple de l'arithmétique, on peut le compléter ainsi pour avoir un langage sorté :

- à 0 on associe la liste vide comme liste des sortes de ses arguments (c'est la seule liste de longueur 0) et *Int* (de *integer* pour les entiers en anglais) comme sorte du résultat. On associe à $+$ et $*$ $[Int ; Int]$ et Int ($+$ et $*$ sont des symboles de fonction qui prennent deux entiers en argument et dont le résultat est un entier).
- $=$ aura pour sorte $[Int ; Int]$.

Notation : Lorsque qu'un symbole de fonction f aura pour sorte de ses arguments une liste $[S_1 ; ... ; S_n]$ et pour sorte de résultat S, nous noterons ce fait $f : S_1, ..., S_n \rightarrow S$.

De plus, si un symbole de fonction a de sorte de résultat S et est d'arité nulle — a est appelée constante — , alors nous noterons ce fait $a : S$.

1.1.2 Typage, bonne formation

Pour vérifier qu'une expression est bien sortée, il existe des algorithmes *décidables* : ils finissent toujours et répondent soit "oui, l'expression est bien sortée", soit "non, elle ne l'est pas". Une méthode est d'utiliser un algorithme sur des *règles d'inférence*.

Une règle d'inférence se présente sous la forme

$$\text{conditions-auxiliaires}\ \frac{C_1 \qquad ... \qquad C_n}{C}\ \text{rule-name}$$

et se lit : « si les conditions C_1, ..., C_n sont respectées, alors on peut déduire C ».

Les conditions C_1, ..., C_n qui se trouvent au-dessus de la barre horizontale s'appellent les *prémisses* ; la déduction C que l'on peut en faire s'appelle la *conclusion.*

Intéressons-nous donc à la sorte des termes dans un environnement donné pour notre langage du premier ordre multisorté. Nous allons utiliser des règles d'inférence portant sur des *jugements de typage*. Un jugement de la forme $\Gamma \vdash_{FOL} t : S$ signifie que le terme t est bien formé et a pour sorte S dans l'environnement Γ.

Un environnement permet de déclarer la sorte des objets et est un ensemble de faits de la forme $x : S$ dans le cas des variables, $f : S_1, ..., S_n \to S$ dans le cas d'un symbole de fonction f et $P : [S_1, ..., S_n]$ dans le cas d'un symbole de prédicat P. Ainsi, si Γ est un environnement et si $f : S_1, ..., S_n \to S \in \Gamma$, cela signifie que f a n arguments de sortes respectives S_1, ..., S_n et pour sorte de résultat S dans l'environnement Γ, conformément à la notation précédente. La définition similaire attendue vaut pour les variables et les symboles de prédicat.

Par abus de langage, nous disons $x \in \Gamma$ s'il existe $S \in \mathit{Sort}$ tel que $x : S \in \Gamma$ si x est une variable. Nous dirons $f \in \Gamma$ s'il existe $S_1, ..., S_n$ et $S \in \Gamma$ tels que $f : S_1, ..., S_n \to S \in$ *mathitSort* lorsque f est un symbole de fonction et $P \in \Gamma$ s'il existe S_1, ..., $S_n \in \mathit{Sort}$ tels que $P : [S_1; ...; S_n] \in \Gamma$, P étant un symbole de prédicat.

Voici donc la règle d'inférence quant au type des termes :

$$\text{si } f : S_1, ..., S_n \to S \in \Gamma \ \frac{\Gamma \vdash_{FOL} t_1 : S_1 \qquad ... \qquad \Gamma \vdash_{FOL} t_n : S_n}{\Gamma \vdash_{FOL} f(t_1, ..., t_n) : S} \text{ fun-type}$$

En particulier, si $n = 0$, la règle revient à

$$\text{si } a : S \in \Gamma \ \frac{}{\Gamma \vdash_{FOL} a : S} \text{ axiome}$$

qui est l'axiome de typage de notre théorie. Notons que cette règle s'applique aussi bien aux fonctions qu'aux variables.

Les propositions n'ont pas de type. Ce que l'on doit vérifier alors, c'est leur bonne formation. On introduit naturellement des *jugements de bonne formation* de la forme $\Gamma \vdash_{FOL} P\ bf$ qui signifie que P est bien formée dans l'environnement Γ. Les règles de bonne formation des propositions sont les suivantes :

$$\frac{}{\Gamma \vdash_{FOL} \top\ bf} \text{ top-type} \qquad \frac{}{\Gamma \vdash_{FOL} \bot\ bf} \text{ bot-type}$$

$$\frac{\Gamma \vdash_{FOL} t_1 : S \qquad \Gamma \vdash_{FOL} t_2 : S}{\Gamma \vdash_{FOL} t_1 = t_2\ bf} \text{ eq-type}$$

$$\text{si } P : [S_1, ..., S_n] \in \Gamma \ \frac{\Gamma \vdash_{FOL} t_1 : S_1 \qquad ... \qquad \Gamma \vdash_{FOL} t_n : S_n}{\Gamma \vdash_{FOL} P(t_1, ..., t_n)\ bf} \text{ pred-type}$$

$$\frac{\Gamma \vdash_{FOL} P_1\ bf \qquad \Gamma \vdash_{FOL} P_2\ bf}{\Gamma \vdash_{FOL} P_1 \wedge P_2\ bf} \text{ and-type} \qquad \frac{\Gamma \vdash_{FOL} P_1\ bf \qquad \Gamma \vdash_{FOL} P_2\ bf}{\Gamma \vdash_{FOL} P_1 \vee P_2\ bf} \text{ or-type}$$

$$\frac{\Gamma \vdash_{FOL} P_1 \ bf \qquad \Gamma \vdash_{FOL} P_2 \ bf}{\Gamma \vdash_{FOL} P_1 \rightarrow P_2 \ bf}\text{imp-type} \qquad \frac{\Gamma \vdash_{FOL} P \ bf}{\Gamma \vdash_{FOL} \neg P \ bf}\text{not-type}$$

$$x \notin \Gamma \ \frac{\{x : S\} \cup \Gamma \vdash_{FOL} P \ bf}{\Gamma \vdash_{FOL} \forall x : S, P \ bf} \qquad x \notin \Gamma \ \frac{\{x : S\} \cup \Gamma \vdash_{FOL} P \ bf}{\Gamma \vdash_{FOL} \exists x : S, P \ bf}\text{exists-type}$$

Pour vérifier qu'un terme est bien sorté, il suffit alors de remonter l'arbre de typage du terme et de s'assurer que toutes les feuilles sont des règles *axiome*. Pour vérifier qu'une proposition est bien formée, il suffit de remonter l'arbre de bonne formation de la proposition et de s'assurer que les termes qui y interviennent sont bien typés ; les feuilles doivent donc être des règles *axiome*, *top-type* ou *bot-type*. Cet algorithme est trivial puisqu'au plus une seule règle ne peut s'appliquer à un terme ou une proposition donnée. Si aucune règle ne peut s'y appliquer, c'est que le terme n'est pas bien sorté ou que la proposition est mal formée.

A propos de la règle *eq-type* : cette règle est particulière en ce sens que les sortes des termes t_1 et t_2 n'interviennent pas dans le reste de la règle. On pourrait donc croire qu'il faut « deviner » l'instanciation qui rend bien formée la proposition $t_1 = t_2$. Notre algorithme ne serait alors pas décidable. En réalité, du fait que l'algorithme du typage des termes est lui décidable, cela assure qu'une unique sorte S peut être déterminée afin que les termes t_1 et t_2 soient bien typés. Il suffit alors de choisir cette sorte pour la règle *eq-type* afin de compléter l'arbre de bonne formation de la proposition.

Exemple : vérifions que la proposition $\forall x : Int, x + 2 = x$ est bien formée dans l'environnement $\Gamma = \{+ : Int, Int \rightarrow Int \,;\, 2 : Int\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash_{FOL} x : Int}\text{axiome} \quad \dfrac{}{\Gamma' \vdash_{FOL} 2 : Int}\text{axiome}}{\Gamma' \vdash_{FOL} x + 2 : Int}\text{fun-type} \qquad \dfrac{}{\Gamma' \vdash_{FOL} x : Int}\text{axiome}}{\Gamma' = \{x : Int\} \cup \Gamma \vdash_{FOL} x + 2 = x \ bf}\text{eq-type}}{\Gamma \vdash_{FOL} \forall x : Int, x + 2 = x \ bf}\text{forall-type}$$

1.1.3 Les démonstrations

Maintenant que l'on a décrit notre langage et que l'on sait comment former des propositions et vérifier qu'elles ont un sens, nous voudrions nous donner les outils qui permettent de les démontrer. Nous avons tous eu, étant plus jeunes, à « Montrer que le triangle ABC est rectangle ». Pour cela nous devions présenter une *preuve*, écrite en mélangeant notre langue naturelle et le langage mathématique mais qui obéit à certaines lois, afin qu'à partir de données on puisse parvenir à une conclusion. Puisque des règles régissent les démonstrations, elles sont sujettes à une formalisation.

Tout d'abord, nous présentons quelques notions fondamentales en théorie de la démonstration.

Une démonstration se construit à partir d'*axiomes* qui sont des propositions dont la vérité est admise sans argumentation. Un ensemble d'axiomes s'appelle une *théorie*.

Exemple : La théorie de l'égalité est formée des axiomes suivants, étant donné une sorte S :

- Principe d'identité : $\forall x : S, x = x$
- Schéma d'axiome de Leibniz pour toute proposition P qui contient z de sorte S : $\forall x : S, \forall y : S, x = y \rightarrow P[x/z] \rightarrow P[y/z]$ où $P[u/v]$ est la proposition P dans laquelle l'on a remplacé toutes les occurrences de v par u

Notons que la notion de remplacement est fondamentale dans un langage. La substitution d'une variable x par un terme t dans un terme est défini par récurrence sur ce terme :
- $x[t/x] = t$,
- si $y \neq x$ est une variable, alors $y[t/x] = y$,
- $f(t_1, ..., t_n)[t/x] = f(t_1[t/x], ..., t_n[t/x])$ pour tout symbole de fonction f.

Et la substitution d'une variable x par un terme t dans une proposition est définie par récurrence sur cette proposition :
- $\top[t/x] = \top$,
- $\bot[t/x] = \bot$,
- $P(t_1, ..., t_n)[t/x] = P(t_1[t/x], ..., t_n[t/x])$ pour tout symbole de prédicat P,
- $(t_1 = t_2)[t/x] = (t_1[t/x] = t_2[t/x]$,
- $(\neg P)[t/x] = \neg P[t/x]$,
- $(P \wedge Q)[t/x] = P[t/x] \wedge Q[t/x]$,
- $(P \vee Q)[t/x] = P[t/x] \vee Q[t/x]$,
- $(P \rightarrow Q)[t/x] = P[t/x] \rightarrow Q[t/x]$,
- $(\forall x : S, P)[t/x] = \forall x : S, P$,
- $(\exists x : S, P)[t/x] = \exists x : S, P$,
- si $y \neq x$ est une variable, alors $(\forall y : S, P)[t/x] = \forall y : S, P[t/x]$.
  La sorte de y peut-être différente de celle de x et t, mais y ne doit pas apparaître dans t. Si c'est le cas, il est nécessaire de renommer la variable y dans $\forall y : S, P$ avec une nouvelle variable.
- si $y \neq x$ est une variable, alors $(\exists y : S, P)[t/x] = \exists y : S, P[t/x]$.
  La sorte de y peut-être différente de celle de x et t, mais y ne doit pas apparaître dans t. Si c'est le cas, il est nécessaire de renommer la variable y dans $\exists y : S, P$ avec une nouvelle variable.

Dès lors que la définition d'une notion est posée, il paraît évident que nous aurons besoin de certains lemmes la concernant pour montrer ensuite des théorèmes dont des hypothèses la font intervenir. Voici donc quelques lemmes au sujet de la substitution de variable dans un terme ou une proposition et ce que l'on peut en conclure quant à leur type ou leur bonne formation.

Lemme 1 : Soient x une variable, u et t des termes, S et T des sortes et Γ un environnement. Si $\{x : S\} \cup \Gamma \vdash_{FOL} u : T$ et si $\Gamma \vdash_{FOL} t : S$ alors $\Gamma \vdash_{FOL} u[t/x] : T$.

Preuve : par récurrence sur $\{x : S\} \cup \Gamma \vdash_{FOL} u : T$.
- Si la dernière règle est *axiome* alors
  - Si $u = x$ est une variable, alors $T = S$ et $u[t/x] = x[t/x] = t$. Or $\Gamma \vdash_{FOL} t : S$. Donc $\Gamma \vdash_{FOL} u[t/x] : S$ où $S = T$. Donc $\Gamma \vdash_{FOL} u[t/x] : T$.

– Si u est une variable différente de x alors $u[t/x] = u$. On sait que $u : T \in \{x : S\} \cup \Gamma$ où u est une variable différente de x. Puisque $u \neq x$, $u : T \notin \{x : S\}$ et on sait alors que $u : T \in \Gamma$. Par *axiome* on obtient donc que $\Gamma \vdash_{FOL} u : T$ avec $u = u[t/x]$, donc que $\Gamma \vdash_{FOL} u[t/x] : T$.
– Si u est une constante alors $u[t/x] = u$. On sait que $u : T \in \{x : S\} \cup \Gamma$ où u est une constante et donc différente de x. Puisque $u \neq x$, $u : T \notin \{x : S\}$ et on sait alors que $u : T \in \Gamma$. Par *axiome* on obtient donc que $\Gamma \vdash_{FOL} u : T$ avec $u = u[t/x]$, donc que $\Gamma \vdash_{FOL} u[t/x] : T$.

– Si la dernière règle est *fun-type*, alors u est de la forme $f(t_1, ..., t_n)$, et $f : T_1, ..., T_n \rightarrow T$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$ pour $0 \leq i \leq n$. Par hypothèse de récurrence on a alors $\Gamma \vdash_{FOL} t_i[t/x] : T_i$. Donc par *fun-type* on obtient que $\Gamma \vdash_{FOL} f(t_1[t/x], ..., t_n[t/x]) : T$. D'où $\Gamma \vdash_{FOL} u[t/x] : T$ (règle de substitution concernant les symboles de fonction).

Lemme 2 : Soient x une variable, S et T deux sortes, t un terme et Γ un environnement. Si $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$ alors $\{x : T\} \cup \Gamma \vdash_{FOL} t : S$.

Preuve : par récurrence sur la taille de t.

– Si t est une variable ou une constante alors on ne peut obtenir $\Gamma \vdash_{FOL} t : S$ qu'en appliquant la règle *axiome*. Dans ce cas, on a donc $t : S \in \Gamma$. Donc on a également $t : S \in \{x : T\} \cup \Gamma$. Et par la règle *axiome*, on a donc $\{x : T\} \cup \Gamma \vdash_{FOL} t : S$.
– Si t est de la forme $f(t_1, ..., t_n)$ $(n > 0)$, on ne peut obtenir $\Gamma \vdash_{FOL} t : S$ qu'en appliquant la règle *fun-type*. On sait donc que $\Gamma \vdash_{FOL} t_i : S_i$ $(1 \leq 0 \leq n$, et que $f : S_1, ..., S_n \rightarrow S \in \Gamma$. De plus, on sait que les t_i sont de taille plus petite que t. Donc avec le fait que $x \notin \Gamma$ et par hypothèse de récurrence on en déduit que $\{x : T\} \cup \Gamma \vdash_{FOL} t_i : S_i$. $f : S_1, ..., S_n \rightarrow S \in \Gamma$ donc $f : S_1, ..., S_n \rightarrow S \in \{x : T\} \cup \Gamma$. Donc par *fun-type* on en déduit que $\{x : T\} \cup \Gamma \vdash_{FOL} f(t_1, ..., t_n) : S$.

Lemme 3 : Soient x une variable, t un terme, S une sorte, P une proposition et Γ un environnement. Si $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$ et si $\Gamma \vdash_{FOL} t : S$ alors $\Gamma \vdash_{FOL} P[t/x]\ bf$.

Preuve : par récurrence sur $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.

– Si la dernière règle est *top-type* c'est que $P = \top$, et dans ce cas $P[t/x] = \top[t/x] = \top$. Par *top-type*, on a bien $\Gamma \vdash_{FOL} P[t/x]\ bf$.
– Si la dernière règle est *bot-type* c'est que $P = \bot$, et dans ce cas $P[t/x] = \bot[t/x] = \bot$. Par *bot-type*, on a bien $\Gamma \vdash_{FOL} P[t/x]\ bf$.
– Si la dernière règle est *eq-type* c'est que P est de la forme $t_1 = t_2$ et que $\{x : S\} \cup \Gamma \vdash_{FOL} t_1 : T$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_2 : T$ pour une certaine sorte T. Par le **Lemme 1** on obtient que $\Gamma \vdash_{FOL} t_1[t/x] : T$ et $\Gamma \vdash_{FOL} t_2[t/x] : T$. On peut alors appliquer la règle *eq-type* pour obtenir que $\Gamma \vdash_{FOL} t_1[t/x] = t_2[t/x]\ bf$, ou encore que $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le =).
– Si la dernière règle est *pred-type* P est donc de la forme $R(t_1, ..., t_n)$ pour un certain symbole de prédicat R. On sait que $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$ pour $0 \leq i \leq n$. Puisque $\Gamma \vdash_{FOL} t : S$, par le **Lemme 1**, on obtient : $\Gamma \vdash_{FOL} t_i[t/x] : T_i$. Ainsi, par la règle *pred-type*, on conclut que $\Gamma \vdash_{FOL} R(t_1[t/x], ..., t_n[t/x])\ bf$ et donc que $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant les symboles de prédicat).

- Si la dernière règle est *and-type* P est donc de la forme $A \wedge B$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Donc par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} A[t/x]\ bf$ et $\Gamma \vdash_{FOL} B[t/x]\ bf$. D'où, par *and-type*, $\Gamma \vdash_{FOL} A[t/x] \wedge B[t/x]\ bf$. On a bien $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le $\wedge$).
- Si la dernière règle est *or-type* P est donc de la forme $A \vee B$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Donc par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} A[t/x]\ bf$ et $\Gamma \vdash_{FOL} B[t/x]\ bf$. D'où, par *or-type*, $\Gamma \vdash_{FOL} A[t/x] \vee B[t/x]\ bf$. On a bien $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le $\vee$).
- Si la dernière règle est *imp-type* P est donc de la forme $A \to B$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Donc par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} A[t/x]\ bf$ et $\Gamma \vdash_{FOL} B[t/x]\ bf$. D'où, par *imp-type*, $\Gamma \vdash_{FOL} A[t/x] \to B[t/x]\ bf$. On a bien $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le $\to$).
- Si la dernière règle est *not-type* P est donc de la forme $\neg Q$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$. Donc par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} Q[t/x]\ bf$. D'où par *not-type*, $\Gamma \vdash_{FOL} \neg Q[t/x]\ bf$ et donc $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le $\neg$).
- Si la dernière règle est *forall-type* P est donc de la forme $\forall y : T, Q$ et l'on a $\{y : T; x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$ avec $y \notin \{x : S\} \cup \Gamma$ (en particulier $y \neq x$). Puisque $y \notin \{x : S\} \cup \Gamma$, alors $y \notin \Gamma$ et comme on a $\Gamma \vdash_{FOL} t : S$, alors par le **Lemme 2** on déduit $\{y : T\} \cup \Gamma \vdash_{FOL} t : S$. On peut donc utiliser l'hypothèse de récurrence pour obtenir $\{y : T\} \cup \Gamma \vdash_{FOL} Q[t/x]\ bf$. Enfin par *forall-type* on démontre $\Gamma \vdash_{FOL} \forall y : T, Q[t/x]\ bf$, c'est-à-dire $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le $\forall$ avec $y \neq x$).
- Si la dernière règle est *exists-type* P est donc de la forme $\exists y : T, Q$ et l'on a $\{y : T; x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$ avec $y \notin \{x : S\} \cup \Gamma$ (en particulier $y \neq x$). Puisque $y \notin \{x : S\} \cup \Gamma$, alors $y \notin \Gamma$ et comme on a $\Gamma \vdash_{FOL} t : S$, alors par le **Lemme 2** on déduit $\{y : T\} \cup \Gamma \vdash_{FOL} t : S$. On peut donc utiliser l'hypothèse de récurrence pour obtenir $\{y : T\} \cup \Gamma \vdash_{FOL} Q[t/x]\ bf$. Enfin par *exists-type* on démontre $\Gamma \vdash_{FOL} \exists y : T, Q[t/x]\ bf$, c'est-à-dire $\Gamma \vdash_{FOL} P[t/x]\ bf$ (règle de substitution concernant le $\exists$ avec $y \neq x$).

L'introduction d'*hypothèses* paraît également une étape naturelle dans l'expression d'une démonstration. Lorsque l'on a pour donnée $n = 0$, et qu'il faut montrer $n + 1 = 1$, on suppose que $n = 0$ est vraie et l'on veut pouvoir, à partir de cette hypothèse, montrer que $n + 1 = 1$. Une démonstration se fera donc dans un *contexte* qui est un ensemble d'hypothèses.

Pour réaliser une démonstration, on doit pouvoir déduire la véracité de nouvelles propositions à partir d' hypothèses et des axiomes de la théorie dans lesquels on se place. Les règles qui traduisent ces déductions sont appelées *axiomes logiques*. Elles se comprennent assez intuitivement pour la plupart et sont les étapes atomiques d'une démonstration. Puisque ces règles permettent de construire des preuves de nouvelles propositions, c'est qu'elles introduisent et éliminent les connecteurs logiques qui permettent de construire les propositions. Plusieurs systèmes de démonstration existent, nous avons choisi d'utiliser la *déduction naturelle* qui s'appuie sur des règles d'inférence.

Tout d'abord, nous allons définir une notion qui aura son importance dans certaine règle et qui intervient dans la définition de la substitution : l'*apparition* d'une variable x

dans un terme est défini par récurrence sur les termes.
- x apparaît dans y si et seulement si $y = x$,
- x apparaît dans $f(t_1, ..., t_n)$ si et seulement si x apparaît dans un t_i, $1 \leq i \leq n$.

La relation similaire est définie sur les propositions par récurrence sur leur structure :
- x n'apparaît ni dans $\top$ ni dans $\bot$,
- x apparaît dans $t_1 = t_2$ si et seulement si x apparaît dans t_1 ou dans t_2.
- x apparaît dans $P(t_1, ..., t_n)$ si et seulement si x apparaît dans un t_i, $1 \leq i \leq n$,
- x apparaît dans $A \wedge B$, $A \vee B$ ou dans $A \rightarrow B$ si et seulement si x apparaît dans A ou dans B,
- x apparaît dans $\neg P$ si et seulement si x apparaît dans P,
- x apparaît dans $\forall y : S, P$ ou $\exists y : S, P$ si et seulement si $y \neq x$ et x apparaît dans P.

Dans la littérature, on trouve souvent l'expression *liée* qui se dit d'une variable lorsqu'elle apparaît dans un terme ou une proposition, et *libre* sinon.

Comme pour la substitution, il paraît judicieux de poser dès à présent des lemmes concernant l'apparition de variable dans un terme ou une proposition et d'en tirer des conclusions quant à leur type ou leur bonne formation.

Lemme 4 : Soient x une variable, t un terme, S et T deux sortes et Γ un environnement. Si x n'apparaît pas dans t et que $\Gamma \vdash_{FOL} t : T$, alors $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$.

Preuve : par récurrence sur la taille de t.
- Si t est une variable, puisque x n'apparaît pas dans t, c'est que $t \neq x$. On ne peut avoir $\Gamma \vdash_{FOL} t : T$ que par *axiome* et donc seulement si $t : T \in \Gamma$. Donc $t : T \in \{x : S\} \cup \Gamma$ et par *axiome* $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$.
- Si t est une constante, x ne peut pas apparaître dans t. On ne peut avoir $\Gamma \vdash_{FOL} t : T$ que par *axiome* et donc seulement si $t : T \in \Gamma$. Donc $t : T \in \{x : S\} \cup \Gamma$ et par *axiome* $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$.
- Si t est de la forme $f(t_1, ..., t_n)$ $(n > 0)$, puisque x n'apparaît pas dans t, c'est que x n'apparaît dans aucun t_i, $0 < i \leq n$. On ne peut avoir $\Gamma \vdash_{FOL} f(t_1, ..., t_n) : T$ que par *fun-type* et donc seulement si $f : T_1, ..., T_n \rightarrow T \in \Gamma$ et $\Gamma \vdash_{FOL} t_i : T_i$. De plus, les t_i sont tous plus petits que t. Donc par hypothèse de récurrence, on a $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$. Puisque $f : T_1, ..., T_n \rightarrow T \in \Gamma$ on a $f : T_1, ..., T_n \rightarrow T \in \{x : S\} \cup \Gamma$ et donc par *fun-type* on a bien $\{x : S\} \cup \Gamma \vdash_{FOL} f(t_1, ..., t_n) : T$.

Lemme 5 : Soient x une variable, P une proposition, S une sorte et Γ un environnement. Si x n'apparaît pas dans P et que $\Gamma \vdash_{FOL} P$ *bf*, alors $\{x : S\} \cup \Gamma \vdash_{FOL} P$ *bf*.

Preuve : par récurrence sur $\Gamma \vdash_{FOL} P$ *bf*.
- Si la dernière règle est *top-type*, il faut montrer $\{x : S\} \cup \Gamma \vdash_{FOL} \top$ qui est vrai par *top-type*.
- Si la dernière règle est *bot-type*, il faut montrer $\{x : S\} \cup \Gamma \vdash_{FOL} \bot$ qui est vrai par *bot-type*.
- Si la dernière règle est *eq-type*, P est de la forme $t_1 = t_2$. On a $\Gamma \vdash_{FOL} t_1 : T$ et $\Gamma \vdash_{FOL} t_1 : T$. De plus, puisque x n'apparaît pas dans P, c'est que x n'apparaît ni

dans t_1 ni dans t_2. Donc par le **Lemme 4**, $\{x : S\} \cup \Gamma \vdash_{FOL} t_1 : T$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_2 : T$. Donc par *eq-type*, $\{x : S\} \cup \Gamma \vdash_{FOL} t_1 = t_2\ bf$ et donc $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.

- Si la dernière règle est *pred-type*, P est donc de la forme $R(t_1, ..., t_n)$. On a $R : [T_1, ..., T_n] \in \Gamma$ et $\Gamma \vdash_{FOL} t_i : T_i$ pour $0 \leq i \leq n$. De plus, puisque x n'apparaît pas dans P, c'est que x n'apparaît dans aucun des t_i. Donc par le **Lemme 4**, $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$. Et donc par *pred-type*, $\{x : S\} \cup \Gamma \vdash_{FOL} R(t_1, ..., t_n)\ bf$, d'où $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *and-type*, P est donc de la forme $A \wedge B$. On a $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$. De plus, puisque x n'apparaît pas dans P, c'est que x n'apparaît ni dans A ni dans B. Donc par hypothèse de récurrence, $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Donc par *and-type*, $\{x : S\} \cup \Gamma \vdash_{FOL} A \wedge B\ bf$, d'où $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *or-type*, P est donc de la forme $A \vee B$. On a $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$. De plus, puisque x n'apparaît pas dans P, c'est que x n'apparaît ni dans A ni dans B. Donc par hypothèse de récurrence, $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Donc par *or-type*, $\{x : S\} \cup \Gamma \vdash_{FOL} A \vee B\ bf$, d'où $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *imp-type*, P est donc de la forme $A \rightarrow B$. On a $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$. De plus, puisque x n'apparaît pas dans P, c'est que x n'apparaît ni dans A ni dans B. Donc par hypothèse de récurrence, $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Donc par *imp-type*, $\{x : S\} \cup \Gamma \vdash_{FOL} A \rightarrow B\ bf$, d'où $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *not-type*, P est donc de la forme $\neg Q$. On a $\Gamma \vdash_{FOL} Q\ bf$. De plus, puisque x n'apparaît pas dans P, c'est que x n'apparaît pas dans Q. Donc par hypothèse de récurrence, $\{x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$. Donc par *not-type*, $\{x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$, d'où $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *forall-type*, P est donc de la forme $\forall y : T, Q$. On a $\{y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$ et $y \notin \Gamma$. De plus, puisque x n'apparaît pas dans P, c'est que $x = y$ ou x n'apparaît pas dans Q. Si $x = y$, on renomme toutes les occurrences de y dans P par une nouvelle variable qui n'y apparaît pas et différente de x. Sinon, puisque $\{y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$ et x n'apparaît pas dans Q alors, par hypothèse de récurrence, $\{x : S; y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$. De plus, $y \notin \Gamma$ et $y \neq x$, donc $y \notin \{x : S\} \cup \Gamma$. Donc par *forall-type* on a $\{x : S\} \cup \Gamma \vdash_{FOL} \forall y : T, Q\ bf$, ou encore $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *exists-type*, P est donc de la forme $\exists y : T, Q$. On a $\{y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$ et $y \notin \Gamma$. De plus, puisque x n'apparaît pas dans P, c'est que $x = y$ ou x n'apparaît pas dans Q. Si $x = y$, on renomme toutes les occurrences de y dans P par une nouvelle variable qui n'y apparaît pas et différente de x. Sinon, puisque $\{y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$ et x n'apparaît pas dans Q alors, par hypothèse de récurrence, $\{x : S; y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$. De plus, $y \notin \Gamma$ et $y \neq x$, donc $y \notin \{x : S\} \cup \Gamma$. Donc par *exists-type* on a $\{x : S\} \cup \Gamma \vdash_{FOL} \exists y : T, Q\ bf$, ou encore $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.

Lemme 6 : Soient x une variable, S une sorte, Γ un environnement et Δ un contexte. Si pour tout $H \in \Delta$, x n'apparaît pas dans H — x n'apparaît pas dans Δ par abus de langage —, et si pour tout $H \in \Delta$, $\Gamma \vdash_{FOL} H\ bf$ — ce que l'on notera $\Gamma \vdash_{FOL} \Delta\ bf$ à présent —, alors $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta \; bf$.

Preuve : par récurrence sur la taille de Δ.

- Si $\Delta = \emptyset$, alors la propriété est satisfaite par vacuité.
- Si $\Delta = \Delta' \cup \{P\}$, x n'apparaît pas dans Δ donc x n'apparaît ni dans Δ' ni dans P. Puisque $\Gamma \vdash_{FOL} \Delta \; bf$, c'est que $\Gamma \vdash_{FOL} \Delta' \; bf$ et $\Gamma \vdash_{FOL} P \; bf$. Puisque $\Gamma \vdash_{FOL} \Delta' \; bf$ et que x n'apparaît pas dans Δ', alors par hypothèse de récurrence $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta' \; bf$. Enfin x n'apparaît pas dans P et $\Gamma \vdash_{FOL} P \; bf$, donc par le **Lemme 5**, $\{x : S\} \cup \Gamma \vdash_{FOL} P \; bf$. $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta' \; bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} P \; bf$, donc $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta' \cup \{P\} \; bf$, c'est-à-dire $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta \; bf$.

Lemme 7 : Soient x une variable, t et u deux termes, S et T deux sortes et Γ un environnement. Si $x \notin \Gamma$, x n'apparaît pas dans t, $\Gamma \vdash_{FOL} u[t/x] : T$ et $\Gamma \vdash_{FOL} t : S$, alors $\{x : S\} \cup \Gamma \vdash_{FOL} u : T$.

Preuve : par récurrence sur la taille de u.

- Si u est une variable et que $u = x$, alors $u[t/x] = t$. On a donc $S = T$ et puisque $\Gamma \vdash_{FOL} t : T$ et x n'apparaît pas dans t alors d'après le **Lemme 4** $\{x : S\} \cup \Gamma \vdash_{FOL} u : T$.
- Si u est une variable et que $u \neq x$, alors $u[t/x] = u$. On a donc $\Gamma \vdash_{FOL} u : T$. Puisque u est une variable, ceci ne peut être vrai que par la règle *axiome* si bien que $u : T \in \Gamma$ et donc $u : T \in \{x : S\} \cup \Gamma$ et enfin par la règle *axiome*, $\{x : S\} \cup \Gamma \vdash_{FOL} u : T$.
- Si u est une constante, $u \neq x$. Alors $u[t/x] = u$. On a donc $\Gamma \vdash_{FOL} u : T$. Puisque u est une constante, ceci ne peut être vrai que par la règle *axiome* si bien que $u : T \in \Gamma$ et donc $u : T \in \{x : S\} \cup \Gamma$ et enfin par la règle *axiome*, $\{x : S\} \cup \Gamma \vdash_{FOL} u : T$.
- Si u est de la forme $f(t_1, ..., t_n)$, alors $u[t/x] = f(t_1[t/x], ..., t_n[t/x])$. On a donc $\Gamma \vdash_{FOL} f(t_1[t/x], ..., t_n[t/x]) : T$. Ceci ne peut être vrai que par la règle *fun-type* si bien que l'on a également $\Gamma \vdash_{FOL} t_i[t/x] : T_i$, $0 < i \leq n$, et $f : T1, ..., T_n \to T \in \Gamma$. Par hypothèse de récurrence, on a bien $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$. Donc par la règle *fun-type*, on conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} f(t_1, ..., t_n) : T$.

Lemme 8 : Soient x une variable, t est terme, S une sorte, P une proposition et Γ un environnement. Si $x \notin \Gamma$, x n'apparaît pas dans P, $\Gamma \vdash_{FOL} P[t/x] \; bf$ et $\Gamma \vdash_{FOL} t : S$ alors $\{x : S\} \cup \Gamma \vdash_{FOL} P \; bf$.

Preuve : par récurrence sur la structure de P.

- Si $P = \top$ alors $\{x : S\} \cup \Gamma \vdash_{FOL} P \; bf$ par *top-type*.
- Si $P = \bot$ alors $\{x : S\} \cup \Gamma \vdash_{FOL} P \; bf$ par *bot-type*.
- Si P est de la forme $t_1 = t_2$, alors $P[t/x]$ est de la forme $t_1[t/x] = t_2[t/x]$ d'après la définition de la substitution. Puisque x n'apparaît pas dans P, alors x n'apparaît ni dans t_1 ni dans t_2. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x] \; bf$ qu'en appliquant la règle *eq-type* si bien que l'on a $\Gamma \vdash_{FOL} t_1[t/x] : T$ et $\Gamma \vdash_{FOL} t_2[t/x] : T$. De plus $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$. Donc par le **Lemme 7**, on déduit $\{x : S\} \cup \Gamma \vdash_{FOL} t_1 : T$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_2 : T$. Enfin, par *eq-type*, on en conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} t_1 = t_2 \; bf$, c'est-à-dire que $\{x : S\} \cup \Gamma \vdash_{FOL} P \; bf$.
- Si P est de la forme $R(t_1, ..., t_n)$, alors $P[t/x]$ est de la forme $R(t_1[t/x], ..., t_n[t/x])$ d'après la définition de la substitution. Puisque x n'apparaît pas dans P, alors

x n'apparaît dans aucun t_i, $0 \leq i \leq n$. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *pred-type* si bien que l'on a $\Gamma \vdash_{FOL} t_i[t/x] : T_i$ et $P : [T_1; ...; T_n] \in \Gamma$. De plus, $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$. Donc par le **Lemme 7**, on déduit $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$. Enfin, par *pred-type*, on en conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} R(t_1, ..., t_n)\ bf$, c'est-à-dire que $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.

- Si P est de la forme $A \wedge B$, alors $P[t/x]$ est de la forme $A[t/x] \wedge B[t/x]$ d'après la définition de la substitution. Puisque x n'apparaît pas dans P, alors x n'apparaît ni dans A ni dans B. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *and-type* si bien que l'on a $\Gamma \vdash_{FOL} A[t/x]\ bf$ et $\Gamma \vdash_{FOL} B[t/x]\ bf$. De plus, $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$. Donc par hypothèse de récurrence, on déduit $\{x : S\} \cup \Gamma A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Enfin, par *and-type*, on en conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} A \wedge B\ bf$, c'est-à-dire que $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si P est de la forme $A \vee B$, alors $P[t/x]$ est de la forme $A[t/x] \vee B[t/x]$ d'après la définition de la substitution. Puisque x n'apparaît pas dans P, alors x n'apparaît ni dans A ni dans B. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *or-type* si bien que l'on a $\Gamma \vdash_{FOL} A[t/x]\ bf$ et $\Gamma \vdash_{FOL} B[t/x]\ bf$. De plus, $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$. Donc par hypothèse de récurrence, on déduit $\{x : S\} \cup \Gamma A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Enfin, par *and-type*, on en conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} A \vee B\ bf$, c'est-à-dire que $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si P est de la forme $A \rightarrow B$, alors $P[t/x]$ est de la forme $A[t/x] \rightarrow B[t/x]$ d'après la définition de la substitution. Puisque x n'apparaît pas dans P, alors x n'apparaît ni dans A ni dans B. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *imp-type* si bien que l'on a $\Gamma \vdash_{FOL} A[t/x]\ bf$ et $\Gamma \vdash_{FOL} B[t/x]\ bf$. De plus, $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$. Donc par hypothèse de récurrence, on déduit $\{x : S\} \cup \Gamma A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. Enfin, par *imp-type*, on en conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} A \rightarrow B\ bf$, c'est-à-dire que $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si P est de la forme $\neg Q$, alors $P[t/x]$ est de la forme $\neg Q[t/x]$ d'après la définition de la substitution. Puisque x n'apparaît pas dans P, alors x n'apparaît pas dans Q. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *not-type* si bien que l'on a $\Gamma \vdash_{FOL} Q[t/x]\ bf$. De plus, $x \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$. Donc par hypothèse de récurrence, on déduit $\{x : S\} \cup \Gamma Q\ bf$. Enfin, par *imp-type*, on en conclut que $\{x : S\} \cup \Gamma \vdash_{FOL} \neg Q\ bf$, c'est-à-dire que $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si P est de la forme $\forall y : T, Q$, $y = x$ ou x n'apparaît pas dans Q puisque x n'apparaît pas dans P. Si $y = x$, alors on renomme toutes les occurrences de y dans P par une nouvelle variable qui n'y apparaît pas et différente de x. Sinon $P[t/x]$ est donc de la forme $\forall y : T, Q[t/x]$ d'après la définition de la substitution. On ne peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *forall-type* si bien que l'on a $y \notin \Gamma$ et $\{y : T\} \cup \Gamma \vdash_{FOL} Q[t/x]\ bf$. Puisque $x \notin \Gamma$ et $x \neq y$ alors $x \notin \{y : T\} \cup \Gamma$. $y \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$ donc d'après le **Lemme 2** $\{y : T\} \cup \Gamma \vdash_{FOL} t : S$. Enfin, x n'apparaît pas dans Q et $\{y : T\} \cup \Gamma \vdash_{FOL} Q[t/x]\ bf$ donc par hypothèse de récurrence $\{x : S; y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$. $y \neq x$ et $y \notin \Gamma$ donc $y \notin \{x : S\} \cup \Gamma$. D'où, par *forall-type*, $\{x : S\} \cup \Gamma \forall y : T, Q\ bf$, c'est-à-dire $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.
- Si P est de la forme $\forall y : T, Q$, $y = x$ ou x n'apparaît pas dans Q puisque x n'apparaît pas dans P. Si $y = x$, alors on renomme toutes les occurrences de y dans P par une nouvelle variable qui n'y apparaît pas et différente de x. Sinon $P[t/x]$ est donc de la forme $\exists y : T, Q[t/x]$ d'après la définition de la substitution. On ne

peut avoir $\Gamma \vdash_{FOL} P[t/x]\ bf$ qu'en appliquant la règle *exists-type* si bien que l'on a $y \notin \Gamma$ et $\{y : T\} \cup \Gamma \vdash_{FOL} Q[t/x]\ bf$. Puisque $x \notin \Gamma$ et $x \neq y$ alors $x \notin \{y : T\} \cup \Gamma$. $y \notin \Gamma$ et $\Gamma \vdash_{FOL} t : S$ donc d'après le **Lemme 2** $\{y : T\} \cup \Gamma \vdash_{FOL} t : S$. Enfin, x n'apparaît pas dans Q et $\{y : T\} \cup \Gamma \vdash_{FOL} Q[t/x]\ bf$ donc par hypothèse de récurrence $\{x : S; y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$. $y \neq x$ et $y \notin \Gamma$ donc $y \notin \{x : S\} \cup \Gamma$. D'où, par *exists-type*, $\{x : S\} \cup \Gamma \exists y : T, Q\ bf$, c'est-à-dire $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.

Maintenant que l'on dispose de tous les outils nécessaires, nous pouvons enfin présenter les règles d'inférence de la déduction naturelle. Celles-ci vont nous permettre de vérifier qu'une proposition est valide dans un contexte donné. Nous avons donc besoin de *jugements de preuve* ; ils auront la forme $\Gamma; \Delta \models P$ qui signifie que dans l'environnement Γ, sous les hypothèses Δ, il existe une démonstration de P. La théorie dans laquelle l'on se place ne sera pas précisée pendant une preuve, elle sera simplement énoncée avant.

Voici les règles de la déduction naturelle :

$$\text{si } P \in \Delta \ \frac{}{\Gamma; \Delta \models P}\ \text{assumption}$$

$$\frac{\Gamma; \Delta \models P \qquad \Gamma; \Delta \models Q}{\Gamma; \Delta \models P \wedge Q}\ \text{and-intro}$$

$$\frac{\Gamma; \Delta \models P \wedge Q}{\Gamma; \Delta \models P}\ \text{and-élim1} \qquad \frac{\Gamma; \Delta \models P \wedge Q}{\Gamma; \Delta \models Q}\ \text{and-élim2}$$

$$\frac{\Gamma; \Delta \models P \qquad \Gamma \vdash_{FOL} Q\ bf}{\Gamma; \Delta \models P \vee Q}\ \text{or-intro1} \qquad \frac{\Gamma; \Delta \models Q \qquad \Gamma \vdash_{FOL} P\ bf}{\Gamma; \Delta \models P \vee Q}\ \text{or-intro2}$$

$$\frac{\Gamma; \Delta \models P \vee Q \qquad \Gamma; \Delta\cup \models P \to R \qquad \Gamma; \Delta\cup \models Q \to R}{\Gamma; \Delta \models R}\ \text{or-élim}$$

$$\frac{\Gamma; \Delta \cup \{P\} \models Q \qquad \Gamma \vdash_{FOL} P\ bf}{\Gamma; \Delta \models P \to Q}\ \text{imp-intro}$$

$$\frac{\Gamma; \Delta \models P \to Q \qquad \Gamma; \Delta \models P}{\Gamma; \Delta \models Q}\ \text{imp-élim}$$

$$\frac{\Gamma; \Delta \cup \{P\} \models \bot \qquad \Gamma \vdash_{FOL} P\ bf}{\Gamma; \Delta \models \neg P}\ \text{not-intro}$$

$$\frac{\Gamma; \Delta \models P \qquad \Gamma; \Delta \models \neg P}{\Gamma; \Delta \models \bot}\ \text{not-élim}$$

$$\frac{}{\Gamma; \Delta \models \top}\ \text{top-intro} \qquad \frac{\Gamma; \Delta \models \bot \qquad \Gamma \vdash_{FOL} P\ bf}{\Gamma; \Delta \models P}\ \text{bot-élim}$$

$$\text{si } x \notin \Gamma \text{ et } x \text{ n'apparaît pas dans } \Delta \quad \frac{\{x : S\} \cup \Gamma; \Delta \models P}{\Gamma; \Delta \models \forall x : S, P}\ \text{forall-intro}$$

$$\frac{\Gamma; \Delta \models \forall x : S, P \qquad \Gamma \vdash_{FOL} t : S}{\Gamma; \Delta \models P[t/x]}\ \text{forall-élim}$$

$$\text{si } x \notin \Gamma \text{ et } x \text{ n'apparaît pas dans } P \quad \frac{\Gamma; \Delta \models P[t/x] \qquad \Gamma \vdash_{FOL} t : S}{\Gamma; \Delta \models \exists x : S, P}\ \text{exists-intro}$$

$$\text{si } x \text{ n'apparaît pas dans } \Delta \cup \{Q\} \quad \frac{\Gamma; \Delta \models \exists x : S, P \qquad \{x : S\} \cup \Gamma; \Delta \cup \{P\} \models Q}{\Gamma; \Delta \models Q}\ \text{exists-élim}$$

Une démonstration sous un ensemble d'axiomes est alors un arbre dont chaque nœud est étiqueté par une proposition, tel que la proposition étiquetant un noeud soit produite par une des règles de déduction ci-dessus. Les feuilles doivent être des règles *assumption* et la racine est étiquetée par la conclusion : c'est la proposition que l'on cherche à démontrer.

Exemple : Dans une théorie égalitaire, montrer la symétrie de l'égalité sur les entiers. On dispose pour cela de deux axiomes : $\forall y_1 : Int, y_1 = y_1$ qui sera appelé *identité* et $\forall z_1 : Int, \forall x_1 : Int, \forall y_1 : Int, x_1 = y_1 \to y_1 = z_1 \to z_1 = x_1$ qui sera appelée *transitivité*.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\pi_{aux} \qquad \dfrac{}{\Gamma; \Delta \models x = y}\ \text{assumption}}{\Gamma; \Delta \models (y = y) \to (y = x)}\ \text{imp-élim} \qquad \dfrac{\dfrac{}{\Gamma; \Delta \models \forall y_1 : Int, y_1 = y_1}\ \text{identité} \qquad \dfrac{}{\Gamma \vdash_{FOL} y : Int}\ \text{axiome}}{\Gamma; \Delta \models y = y}\ \text{forall-élim}}{\Gamma; \Delta = \{x = y\} \models y = x}\ \text{imp-élim}}{\Gamma = [x : Int; y : Int]; \emptyset \models x = y \to y = x}\ \text{imp-intro}}{[x : Int]; \emptyset \models \forall y : Int, x = y \to y = x}\ \text{forall-intro}}{[]; \emptyset \models \forall x : Int, \forall y : Int, x = y \to y = x}\ \text{forall-intro}$$

$\pi_{aux} =$

$$\dfrac{\dfrac{\pi_{end} \qquad \dfrac{}{\Gamma \vdash_{FOL} x : Int}\ \text{axiome}}{\Gamma; \Delta \models \forall y_1 : Int, (x = y_1) \to (y_1 = y) \to (y = x)}\ \text{forall-élim} \qquad \dfrac{}{\Gamma \vdash_{FOL} y : Int}\ \text{axiome}}{\Gamma; \Delta \models (x = y) \to (y = y) \to (y = x)}\ \text{forall-élim}$$

$\pi_{end} =$

$$\dfrac{\dfrac{}{\Gamma; \Delta \models \forall z_1 : Int, \forall x_1 : Int, \forall y_1 : Int, x_1 = y_1 \to y_1 = z_1 \to z_1 = x_1}\ \text{Leibniz} \qquad \dfrac{}{\Gamma \vdash_{FOL} y : Int}\ \text{axiome}}{\Gamma; \Delta \models \forall x_1 : Int, \forall y_1 : Int, (x_1 = y_1) \to (y_1 = y_1) \to (y_1 = x_1)}\ \text{forall-élim}$$

Les propositions mal formées ne doivent pas pouvoir être démontrées. Ceci est assuré par le théorème suivant qui signifie que s'il existe une preuve d'une proposition, c'est que celle-ci est bien formée.

Nous aurons besoin de deux lemmes supplémentaires pour cela ; le théorème suit.

Lemme 9 : Soient x une variable, t un terme, S et T deux sortes et Γ un environnement. Si $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$ et que x n'apparaît pas dans t, alors $\Gamma \vdash_{FOL} t : T$.

Preuve : par récurrence sur la taille de t.

- Si t est une variable, $t \neq x$ puisque x n'apparaît pas dans t. On ne peut avoir $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$ que par la règle *axiome* et donc seulement si $t : T \in \{x : S\} \cup \Gamma$. Dans ce cas $t : T \in \Gamma$ puisque $t \neq x$. Et donc par *axiome*, on a bien $\Gamma \vdash_{FOL} t : T$.
- Si t est une constante, $t \neq x$. On ne peut avoir $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$ que par la règle *axiome* et donc seulement si $t : T \in \{x : S\} \cup \Gamma$. Dans ce cas $t : T \in \Gamma$ puisque $t \neq x$. Et donc par *axiome*, on a bien $\Gamma \vdash_{FOL} t : T$.
- Si t est de la forme $f(t_1, ..., t_n)$, on sait que x n'apparaît dans aucun t_i $(0 \leq i \leq n)$ puisque x n'apparaît pas dans t. De plus on ne peut avoir $\{x : S\} \cup \Gamma \vdash_{FOL} t : T$ que par la règle *fun-type* si bien que l'on a également $f : T_1, ..., T_n \to T \in \{x : S\} \cup \Gamma$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$. Or, x n'apparaît dans aucun t_i et les t_i sont tous plus petits que t. Donc par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} t_i : T_i$ et par *fun-type* on a bien $\Gamma \vdash_{FOL} f(t_1, ..., t_n) : T$ ou encore $\Gamma \vdash_{FOL} t : T$.

Lemme 10 : Soient x une variable, S une sorte, P une proposition et Γ un environnement. Si $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$ et que x n'apparaît pas dans P, alors $\Gamma \vdash_{FOL} P\ bf$.

Preuve : par récurrence sur $\{x : S\} \cup \Gamma \vdash_{FOL} P\ bf$.

- Si la dernière règle est *top-type*, $P = \top$ et $\Gamma \vdash_{FOL} \top\ bf$ par *top-type*.
- Si la dernière règle est *bot-type*, $P = \bot$ et $\Gamma \vdash_{FOL} \bot\ bf$ par *bot-type*.
- Si la dernière règle est *eq-type*, P est donc de la forme $t_1 = t_2$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} t_1 : T$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_2 : T$. Puisque x n'apparaît pas dans P, c'est que x n'apparaît ni dans t_1 ni dans t_2. Donc par le **Lemme 9** $\Gamma \vdash_{FOL} t_1 : T$ et $\Gamma \vdash_{FOL} t_2 : T$. Et donc par *eq-type*, $\Gamma \vdash_{FOL} t_1 = t_2\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$
- Si la dernière règle est *pred-type*, P est donc de la forme $R(t_1, ..., t_n)$ et l'on a $R : [T_1; ...; T_n] \in \{x : S\} \cup \Gamma$ et $\{x : S\} \cup \Gamma \vdash_{FOL} t_i : T_i$, $0 \leq i \leq n$. Puisque x n'apparaît pas dans P, alors x n'apparaît dans aucun des t_i. Donc par le **Lemme 9** $\Gamma \vdash_{FOL} t_i : T_i$ et par *pred-type* on a bien $\Gamma \vdash_{FOL} R(t_1, ..., t_n)\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *and-type*, P est donc de la forme $A \wedge B$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. De plus, x n'apparaît pas dans P donc x n'apparaît ni dans A ni dans B. Donc par hypothèse de récurrence on obtient que $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$. D'où, par *and-type*, $\Gamma \vdash_{FOL} A \wedge B\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *or-type*, P est donc de la forme $A \vee B$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. De plus, x n'apparaît pas dans P donc x n'apparaît ni dans A ni dans B. Donc par hypothèse de récurrence on obtient que $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$. D'où, par *or-type*, $\Gamma \vdash_{FOL} A \vee B\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.

- Si la dernière règle est *imp-type*, P est donc de la forme $A \to B$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} A\ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} B\ bf$. De plus, x n'apparaît pas dans P donc x n'apparaît ni dans A ni dans B. Donc par hypothèse de récurrence on obtient que $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$. D'où, par *imp-type*, $\Gamma \vdash_{FOL} A \to B\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *imp-type*, P est donc de la forme $\neg Q$ et l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$. De plus, x n'apparaît pas dans P donc x n'apparaît pas dans Q. Donc par hypothèse de récurrence on obtient que $\Gamma \vdash_{FOL} Q\ bf$. D'où, par *not-type*, $\Gamma \vdash_{FOL} \neg Q\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *forall-type*, P est donc de la forme $\forall y : T, Q$ avec $y \neq x$ et x n'apparaît pas dans Q puisque x n'apparaît pas dans P. De plus on a $y \notin \{x : S\} \cup \Gamma$ et $\{y : T; x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$. Donc par hypothèse de récurrence puisque $x \notin \{y : T\} \cup \Gamma$ (car $x \neq y$ et $x \notin \Gamma$), on obtient $\{y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$. Et donc par *forall-type* puisque $y \notin \Gamma$ (car $y \notin \{x : S\} \cup \Gamma$), on a bien $\Gamma \vdash_{FOL} \forall y : T, Q\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *exists-type*, P est donc de la forme $\exists y : T, Q$ avec $y \neq x$ et x n'apparaît pas dans Q puisque x n'apparaît pas dans P. De plus on a $y \notin \{x : S\} \cup \Gamma$ et $\{y : T; x : S\} \cup \Gamma \vdash_{FOL} Q\ bf$. Donc par hypothèse de récurrence puisque $x \notin \{y : T\} \cup \Gamma$ (car $x \neq y$ et $x \notin \Gamma$), on obtient $\{y : T\} \cup \Gamma \vdash_{FOL} Q\ bf$. Et donc par *exists-type* puisque $y \notin \Gamma$ (car $y \notin \{x : S\} \cup \Gamma$), on a bien $\Gamma \vdash_{FOL} \exists y : T, Q\ bf$ ou encore $\Gamma \vdash_{FOL} P\ bf$.

Théorème 1 : Soient Γ un environnement, Δ un contexte et P une proposition. Si $\Gamma \vdash_{FOL} \Delta\ bf$, et $\Gamma; \Delta \models P$, alors $\Gamma \vdash_{FOL} P\ bf$.

Preuve : procédons par récurrence sur la preuve $\Gamma; \Delta \models P$:
- Si la dernière règle de cette preuve est *assumption*, c'est que $P \in \Delta$. Or on sait que pour tout $H \in \Delta$, $\Gamma \vdash_{FOL} H\ bf$. Donc en particulier puisque $P \in \Delta$, $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *and-intro*, P est donc de la forme $A \wedge B$ et on a $\Gamma; \Delta \models A$ et $\Gamma; \Delta \models B$. Donc par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} A\ bf$ et $\Gamma \vdash_{FOL} B\ bf$ (Δ reste le même). La règle *and-type* nous dit alors que $\Gamma \vdash_{FOL} A \wedge B\ bf$, c'est-à-dire que $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *and-élim1*, c'est que l'on a $\Gamma; \Delta \models P \wedge Q$. Par hypothèse de récurrence, on sait donc que $\Gamma \vdash_{FOL} P \wedge Q\ bf$. Or ceci ne peut être satisfait qu'en appliquant à ce jugement la règle *and-type*, si bien que l'on peut en conclure $\Gamma \vdash_{FOL} P\ bf$ et $\Gamma \vdash_{FOL} Q\ bf$. On a bien $\Gamma \vdash_{FOL} P\ bf$.
- De même, si la dernière règle est *and-élim2*, on a $\Gamma; \Delta \models Q \wedge P$ et par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} Q \wedge P\ bf$. Seule la règle *and-type* peut-être utilisée pour satisfaire ce jugement et l'on a donc $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *or-intro1*, P est donc de la forme $A \vee B$ et l'on a $\Gamma; \Delta \models A$ et $\Gamma \vdash_{FOL} B\ bf$. Par hypothèse de récurrence, on en déduit $\Gamma \vdash_{FOL} A\ bf$ qui, associé à $\Gamma \vdash_{FOL} B\ bf$ et par la règle *or-type*, permet d'obtenir $\Gamma \vdash_{FOL} A \vee B\ bf$, soit $\Gamma \vdash_{FOL} P\ bf$.
- De même si la dernière règle est *or-intro2*, P est de la forme $A \vee B$ et par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} B\ bf$. Avec $\Gamma \vdash_{FOL} A\ bf$ et par la règle *or-type* on obtient $\Gamma \vdash_{FOL} A \vee B\ bf$, soit $\Gamma \vdash_{FOL} P\ bf$.

- Si la dernière règle est *or-élim*, c'est qu'il existe un certain A et un certain B tels que $\Gamma; \Delta \models A \vee B$, $\Gamma; \Delta \models A \rightarrow P$ et $\Gamma; \Delta \models B \rightarrow P$. Par hypothèse de récurrence sur $\Gamma; \Delta \models A \rightarrow P$, on déduit $\Gamma \vdash_{FOL} A \rightarrow P \ bf$, qui ne peut être vrai que par la règle *imp-type*, si bien que $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *imp-intro*, P est de la forme $A \rightarrow B$ et on a $\Gamma; \Delta \cup \{A\} \models B$ et $\Gamma \vdash_{FOL} A \ bf$. De ce dernier fait et de $\Gamma \vdash_{FOL} \Delta \ bf$, on sait que $\Gamma \vdash_{FOL} \Delta \cup \{A\} \ bf$. D'où $\Gamma \vdash_{FOL} B \ bf$ par hypothèse de récurrence. Avec $\Gamma \vdash_{FOL} A \ bf$ et par la règle *imp-type* on obtient donc $\Gamma \vdash_{FOL} A \rightarrow B \ bf$, ou encore $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *imp-élim*, c'est qu'il existe un certain Q tel que $\Gamma; \Delta \models Q \rightarrow P$ et $\Gamma \models Q$. Par hypothèse de récurrence, on a bien $\Gamma \vdash_{FOL} Q \rightarrow P \ bf$ qui ne peut être obtenu qu'en appliquant la règle *imp-type*. Ce qui veut dire que l'on a $\Gamma \vdash_{FOL} Q \ bf$ et surtout $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *not-intro*, P est donc de la forme $\neg Q$ pour un certain Q et l'on a $\Gamma \vdash_{FOL} Q \ bf$. Donc par *not-type*, on a bien $\Gamma \vdash_{FOL} \neg Q \ bf$ et donc $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *not-élim* il faut montrer $\Gamma \vdash_{FOL} \perp \ bf$, ce qui est immédiatement vrai d'après la règle *bot-type*.
- Si la dernière règle est *top-intro*, il faut montrer $\Gamma \vdash_{FOL} \top \ bf$, ce qui est immédiatement vrai d'après *top-type*.
- Si la dernière règle est *bot-élim* il faut montrer $\Gamma \vdash_{FOL} P \ bf$ qui est une prémisse.
- Si la dernière règle est *forall-intro*, P est de la forme $\forall x : S, Q$. Tout d'abord, on sait que x n'apparaît pas dans Δ et $\Gamma \vdash_{FOL} \Delta \ bf$, donc par le **Lemme 6**, on obtient $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta \ bf$. Avec ceci et la prémisse de la règle, $\{x : S\} \cup \Gamma; \Delta \models Q$, et par hypothèse de récurrence, on déduit $\{x : S\} \cup \Gamma \vdash_{FOL} Q \ bf$. Il suffit alors d'appliquer à cela la règle *forall-type* ($x \notin \Gamma$ d'après le cas *forall-intro*) pour en déduire $\Gamma \vdash_{FOL} \forall x : S, Q \ bf$ et donc $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *forall-élim*, P est de la forme $Q[t/x]$ et l'on a $\Gamma; \Delta \models \forall x : S, Q$ et $\Gamma \vdash_{FOL} t : S$. Par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} \forall x : S, Q \ bf$. Ceci ne peut être obtenu qu'en appliquant la règle *forall-type* si bien que l'on a également $\{x : S\} \cup \Gamma \vdash_{FOL} Q \ bf$ et $x \notin \Gamma$. Puisque $\{x : S\} \cup \Gamma \vdash_{FOL} Q \ bf$ et que $\Gamma \vdash_{FOL} t : S$, alors par le **Lemme 3**, $\Gamma \vdash_{FOL} Q[t/x] \ bf$ et donc $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *exists-intro*, P est de la forme $\exists x : S, Q$ et l'on a $\Gamma; \Delta \models Q[t/x]$ et $\Gamma \vdash_{FOL} t : S$. Par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} Q[t/x] \ bf$. De ceci et puisque $x \notin \Gamma$, x n'apparaît pas dans Q et $\Gamma \vdash_{FOL} t : S$ alors, par le **Lemme 8**, $\{x : S\} \cup \Gamma \vdash_{FOL} Q \ bf$. Enfin, puisque $x \notin \Gamma$, on peut alors appliquer la règle *exists-type* pour obtenir $\Gamma \vdash_{FOL} \exists x : S, Q \ bf$ ou encore $\Gamma \vdash_{FOL} P \ bf$.
- Si la dernière règle est *exists-élim*, on a $\Gamma; \Delta \models \exists x : S, Q$, $\{x : S\} \cup \Gamma; \Delta \cup \{Q\} \models P$ et x n'apparaît pas dans $\Delta \cup \{P\}$. Tout d'abord, par hypothèse de récurrence, on a $\Gamma \vdash_{FOL} \exists x : S, Q \ bf$. Et ceci ne peut être vrai qu'en appliquant la règle *exists-type* si bien que l'on a $\{x : S\} \cup \Gamma \vdash_{FOL} Q \ bf$ et $x \notin \Gamma$. De plus, on sait que x n'apparaît pas dans $\Delta \cup \{P\}$, donc x n'apparaît pas ni dans Δ ni dans P. Avec $\Gamma \vdash_{FOL} \Delta \ bf$ et par le **Lemme 6**, on obtient que $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta \ bf$. Puisque $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta \ bf$ et $\{x : S\} \cup \Gamma \vdash_{FOL} Q \ bf$, alors $\{x : S\} \cup \Gamma \vdash_{FOL} \Delta \cup \{Q\} \ bf$. On peut alors utiliser encore une fois l'hypothèse de récurrence, ici sur $\{x : S\} \cup \Gamma; \Delta \cup \{Q\} \models P$, pour déduire que $\{x : S\} \cup \Gamma \vdash_{FOL} P \ bf$. De plus, on sait que x n'apparaît pas dans P donc par le **Lemme 10**, $\Gamma \vdash_{FOL} P \ bf$.

1.2 Calcul des Constructions Inductives

1.2.1 Présentation générale

Nous avons décrit dans la section précédente un langage permettant d'exprimer différents énoncés et des outils permettant de les démontrer. Mais ce langage est assez peu expressif, du fait que les quantifications et les arguments des fonctions et prédicats ne peuvent être que des objets de type simple. Par exemple, il est impossible d'exprimer le fait que toute fonction de N dans N admet un minimum.

Depuis longtemps maintenant, certains mathématiciens se sont efforcés de formaliser les mathématiques afin de pouvoir en étudier les propriétés.

L'origine du Calcul des Constructions Inductives commence en 1932 lorsque Church introduit le λ-calcul pur. Ce formalisme qui a pour avantage d'être extrêmement simple (3 règles de formation des termes et une seule règle de calcul) est un paradigme de la programmation fonctionnelle et il a le même pouvoir d'expression que les machines de Turing.

Sur la base de ce λ-calcul, différents systèmes typés ont alors vu le jour comme le *λ-calcul simplement typé*, le *système F*, le *système F_ω*, le *Calcul des Constructions* puis le *Calcul des Constructions Inductives* — que l'on notera CCI à présent. Ces systèmes se devaient avant tout d'être cohérents, ensuite de tenir compte d'une expressivité aussi riche que possible.

Nous ne présentons pas tous ces calculs mais seulement le CCI. Le lecteur intéressé peut se référer à [14] pour une présentation complète de ces calculs ainsi que de leur relation entre eux. En revanche nous présentons des constructions particulières du CCI dont certaines se trouvent déjà dans d'autres des systèmes cités.

Effectivement, il faut voir ces systèmes comme une hiérarchie où chacun est plus riche que son prédécesseur par l'introduction de nouvelles constructions. Nous donnons des exemples de ces constructions pour aider le lecteur à comprendre au final la richesse du CCI.

Pour ce dernier, nous présentons la grammaire des termes ainsi que les règles de typage, de bonne formation et de déduction (qui sont intrinsèquement liées comme nous le verrons par la suite). Une présentation complète peut être trouvée dans [1] et dans [12].

Tout d'abord, le CCI introduit la notion de *types simples*. Cette notion est semblable à celle de la logique du premier ordre multisortée définie dans la section précédente.

Ensuite, on y trouve le *polymorphisme* qui va permettre d'abstraire un terme par rapport à un type. On peut alors exprimer une fonction identité sur tous les types. Une telle fonction prend en argument un type, un objet de ce type et retourne cet objet.

L'*ordre supérieur* vient s'ajouter à ces constructions pour exprimer des propositions abstraites par rapport à d'autres propositions. Dans ce langage, tout objet a donc un type, y compris les types eux-mêmes. Les fonctions peuvent alors prendre en argument des propositions pour en retourner de nouvelles.

Les *types dépendants* est une notion introduite pour exprimer des propriétés sur les termes. Pour exemple, on aimerait pouvoir parler des entiers pairs en tant qu'ensemble. On veut donc pouvoir faire dépendre les types par rapport à des termes.

Enfin, les *types inductifs* viennent compléter toutes ces constructions pour que, à partir du λ-calcul pur, on obtienne le CCI. Un type inductif est la représentation d'une algèbre libre engendrée par un nombre fini de constructeurs. Des objets simples comme les entiers, mais aussi plus complexes comme la relation entre une liste d'objets et la taille de cette liste, peuvent alors être définis. De plus, une construction particulière aux types inductifs leur est associée : le schéma d'*élimination forte*. Ce principe permet notamment de filtrer un objet d'un type inductif à travers les constructeurs de ce type. Enfin, de par le principe d'élimination forte, on peut également définir des fonctions *récursives*.

Nous n'expliquons pas en détail les principes d'élimination forte ni en quoi le CCI reste cohérent malgré des constructions comme les types inductifs ou les fonctions récursives. Cependant, une preuve complète de cohérence est donnée dans [21].

1.2.2 Les termes

Nous allons maintenant présenter la grammaire des termes du CCI ; quelques explications et exemples suivront. Comme en logique du premier ordre, on dispose d'un ensemble infini de variables.

- *Prop* est un terme. Il représente l'ensemble particulier des propositions ;
- *Set* est un terme. Typiquement, un objet de type *Set* est un ensemble ;
- *TypeSet* est un terme. Il représente le type de tous les types.
- si x est une variable, x est un terme.
- si T et U sont des termes, alors $(T\ U)$ est un terme. C'est l'*application* de U à T ;
- si x est une variable, T et U deux termes, alors $\lambda x : T, U$ est un terme. C'est l'*abstraction* qui permet de former des fonctions ;
- si x est une variable, T et U deux termes, alors $\forall x : T, U$ est un terme. C'est le *produit dépendant*. Il permet de former des types qui dépendent de termes. À noter que si x n'apparaît pas dans U, le produit est alors *non dépendant* et est noté $T \rightarrow U$;
- si x_1, ... , x_n, I_1, ..., I_m, c_1, ..., c_k sont des variables, T_1, ..., T_n, C_1, ..., C_k sont des termes et Ar_1, ..., Ar_m sont des *arités* (*Prop*, *Set* ou un produit d'arité), alors $Ind[x_1 : T_1; ...; x_n : T_n](I_1 : Ar_1; I_m : Ar_m := c_1 : C_1 \mid ... \mid c_k : C_k)$ est un terme. Cet objet représente un (des) type(s mutuellement) inductif(s) ;
- si I est un type inductif qui a au moins i constructeur, alors $Constr(i, I)$ est un terme. Il représente le $i$ème constructeurs de I ;
- si x_{11}, ..., x_{ni_n} sont des variables, c_1, ..., c_n des constructeurs et m, T, f_1, ..., f_n des termes, alors $case(m, T, (c_1\ x_{11}\ ...\ x_{1i_1} \Rightarrow f_1 \mid ... \mid c_n\ x_{n1}\ ...\ x_{ni_n} \Rightarrow f_n)$ est un terme. Il correspond au filtrage du terme m et aux termes retournés alors ;
- si f_1, ..., f_n sont des variables et A_1, ..., A_n, F_1, ..., F_n sont des termes, alors *Fix* f_i $[f_1 : A_1 := F_1 ; ... ; f_n : A_n := F_n]$ est un terme $(1 \leq i \leq n)$. C'est le terme correspondant à la $i$ème fonction du bloc de déclaration de fonctions mutuellement récursives.

$$
\begin{array}{rl}
T ::= & Prop \\
| & Set \\
| & x \\
| & (T\ T) \\
| & \lambda x : T, T \\
| & \forall x : T, T \\
| & Ind[x : T\,;\, ...\,;\, x : T](x : Ar\,;\, ...\,;\, x : Ar := x : T \mid ... \mid x : T) \\
| & Constr(i, x) \text{ si } i \text{ est un entier} \\
| & case(T, T, Constr(i, x)\ x\ ...\ x \Rightarrow T \mid ... \mid Constr(i, x)\ x\ ...\ x \Rightarrow T) \\
| & Fix\ x\ [x : T := T\,;\, ...\,;\, x : T := T]
\end{array}
$$

et

$$Ar ::= Prop \mid Set \mid \forall x : T, Ar$$

Exemples : Les expressions suivantes sont des termes du CCI :
- $\lambda A : Set, \lambda x : A, x$ est la fonction identité polymorphe ;
- $Ind[](nat : Set := 0 : nat \mid S : nat \rightarrow nat)$ décrit l'ensemble des entiers naturels ;
- $(((\lambda A : Set, \lambda x : A, x)\ nat)\ (S\ 0))$ est l'application de la fonction identité polymorphe à l'ensemble nat et de l'élément $S\ 0$. Nous verrons par la suite comment *calculer* le résultat d'une telle application ;
- $\forall f : nat \rightarrow nat, \forall x : nat, f\ x = x$ signifiera, lorsque le symbole $=$ sera défini, que toute fonction des entiers vers les entiers est l'identité ; si la véracité de cette proposition peut être — heureusement — mise en doute, on peut cependant l'énoncer ;
- $Fix\ plus\ [plus : nat \rightarrow nat \rightarrow nat := \lambda n : nat, \lambda m : nat, case(n, nat, 0 \Rightarrow m \mid S\ p \Rightarrow S(plus\ p\ m))]$ est la fonction d'addition de deux entiers. Dans le corps du *case*, le m représente le retour de la fonction *plus* lorsque le terme filtré, ici n, est sous la forme du premier constructeur de *nat*, donc 0. Ensuite se trouve le cas du second constructeur et correspond à la *branche* $S\ p \Rightarrow S(plus\ p\ m)$.

La nécessité de définir les inductifs d'une part et les fonctions récursives d'autre part dans des blocs est directement en rapport avec la cohérence du langage. Encore une fois, nous n'entrons pas dans les détails ici puisque ce n'est pas l'aspect cohérent du CCI qui nous intéresse, mais la possibilité d'une traduction vers une logique plus faible.

1.2.3 Typage, bonne formation

Comme en logique du premier ordre multisorté, on dispose d'algorithmes permettant de tester si une expression est bien typée ou non. Une fois de plus, nous allons utiliser des règles d'inférence et des jugements de typage. Ici, nous allons en réalité définir deux notions qui feront appel l'une à l'autre. On définit la substitution et l'apparition de façon similaire au premier ordre ; les lieurs sont ici le λ et le $\forall$, et la substitution d'une variable x par un terme t dans un terme u sera noté $u\{t/x\}$.

Pour vérifier le type d'un terme, nous avons besoin d'un objet comme l'environnement en logique du premier ordre multisortée.

On dispose tout d'abord d'un *contexte local.* C'est une liste ordonnée de déclarations ou de définitions de variables :

$$\begin{array}{rcl} G ::= & & [] \\ & | & (x : T) :: G \\ & | & (x := T : T) :: G \end{array}$$

On dispose également d'un *environnement global* qui est une liste ordonnée de déclarations ou de définitions de constantes. On peut y insérer des objets plus complexes que dans le cas des contextes locaux :

$$\begin{array}{rcl} E ::= & & [] \\ & | & (c : T) :: E \\ & | & (c := T : T) :: E \\ & | & Ind[x : T\, ;\, ...\, ;\, x : T](x : Ar\, ;\, ...\, ;\, x : Ar := x : T \mid ... \mid x : T) :: E \end{array}$$

Ceci étant défini, nous aurons besoin de deux fonctions, l'une permettant de tester si un élément se trouve dans une liste, noté $e \in l$, et la concaténation de deux listes, noté l_1 @ l_2 qui retourne une liste dont les premiers éléments sont ceux de l_1 et dont les éléments suivants sont ceux de l_2, les ordres des éléments respectifs des deux listes étant conservés.

Appartenance à une liste :
- Aucun élément n'appartient à la liste vide []
- $e \in x :: l$ si et seulement si $e = x$ ou $e \in l$.

Concaténation de deux listes :
- $[]$ @ $l = l$
- $(e :: l_1)$ @ $l_2 = e :: (l_1$ @ $l_2)$.

Nous ajoutons à cela la notation $c :! \ T \in E$ qui signifie que :
- $c : T \in E$ ou
- il existe un certain terme t tel que $c := t : T \in E$.

Le typage d'un terme va donc se faire dans un environnement global et dans un contexte local. Le point important étant que les éléments du contexte local peuvent dépendre des éléments de l'environnement global, le contraire n'étant pas permis. De plus, l'ordre imposé au sein de ces objets est le suivant : un élément d'un environnement global ou du contexte local ne peut dépendre que des éléments qui se trouvent à sa droite. Il va donc falloir vérifier la bonne formation des couples environnement global / contexte local. Ce fait sera noté $WF(E; G)$ qui signifie que le couple $E; G$ est bien formé. On définit alors le jugement $E; G \vdash_{CCI} t : T$ qui signifie que dans $E; G$, le terme t a pour type T. On note PST l'ensemble $\{Prop, Set, TypeSet\}$. Voici alors les règles de typage du CCI :

$$\frac{}{WF([]; [])}\ \text{W-E}$$

$$\text{si } s \in PST, c \notin E \text{ @ } G\ \frac{E; G \vdash_{CCI} T : s}{WF((c : T) :: E; G)}\ \text{W-DeclE}$$

$$\text{si } s \in PST, x \notin E \mathbin{@} G \ \frac{E; G \vdash_{CCI} T : s}{WF(E; (x : T) :: G)} \text{ W-DeclG}$$

$$\text{si } c \notin E \mathbin{@} G \ \frac{E; G \vdash_{CCI} t : T}{WF((c := t : T) :: E; G)} \text{ W-DefE}$$

$$\text{si } x \notin E \mathbin{@} G \ \frac{E; G \vdash_{CCI} t : T}{WF((x := t : T) :: E; G)} \text{ W-DefG}$$

$$\frac{WF(E; G)}{E; G \vdash_{CCI} Prop : TypeSet} \text{ Ax-Prop}$$

$$\frac{WF(E; G)}{E; G \vdash_{CCI} Set : TypeSet} \text{ Ax-Set}$$

$$\text{si } c :! T \in E \mathbin{@} G \ \frac{WF(E; G)}{E; G \vdash_{CCI} c : T} \text{ Const}$$

$$\text{si } s \in PST \ \frac{E; G \vdash_{CCI} T : s \qquad E; (x : T) :: G \vdash_{CCI} U : Prop}{E; G \vdash_{CCI} \forall x : T, U : Prop} \text{ Prod-Prop}$$

$$\text{si } s \in \{Prop; Set\} \ \frac{E; G \vdash_{CCI} T : s \qquad E; (x : T) :: G \vdash_{CCI} U : Set}{E; G \vdash_{CCI} \forall x : T, U : Set} \text{ Prod-Set}$$

$$\frac{E; G \vdash_{CCI} \forall x : T, U : s \qquad E; (x : T) :: G \vdash_{CCI} t : U}{E; G \vdash_{CCI} \lambda x : T, t : \forall x : T, U} \text{ Lam}$$

$$\frac{E; G \vdash_{CCI} u : \forall x : T, U \qquad E; G \vdash_{CCI} t : T}{E; G \vdash_{CCI} (u\ t) : U\{t/x\}} \text{ App}$$

Si $G_P = p_1 : P_1 ; ... ; p_r : P_r$, $G_I = I_1 : A_1 ; ... ; I_k : A_k$ et $G_C = c_1 : C_1 \mid ... \mid c_n : C_n$,

$$1 \leq j \leq k \ \frac{Ind[G_P](G_I := G_C) \in E}{I_j : \forall p_1 : P_1, ..., \forall p_r : P_r, A_j \in E} \text{ Ind-type}$$

$$1 \leq i \leq n \ \frac{Ind[G_P](G_I := G_C) \in E}{c_i : \forall p_1 : P_1, ..., \forall p_r : P_r, C_i\{I_1\ p_1\ ...\ p_r/I_1\}...\{I_k\ p_1\ ...\ p_r/I_k\} \in E} \text{ Ind-constr}$$

$$1 \leq i \leq n, 1 \leq j \leq k \ \frac{E; G_I \mathbin{@} G_P \mathbin{@} G \vdash_{CCI} C_i : s_{p_i} \qquad E; G_P \mathbin{@} G \vdash_{CCI} A_j : s'_j}{WF(Ind[G_P](G_I := G_C) :: E; G)} \text{ W-Ind}$$

avec :
- $k > 0$, les I_j et c_i ont des noms deux à deux distincts ;
- les A_j sont de la forme $\forall x_{j1} : T_{j1}, ...\forall x_{jn} : T_{jn}, Prop$ ou $\forall x_{j1} : T_{j1}, ...\forall x_{jn} : T_{jn}, Set$ quels que soient T_{j1}, ..., T_{jn}, et $I_j \notin E \ @ \ G$;
- les C_i sont des constructeurs de type I_{p_i} et $c_i \notin E \ @ \ G$.

La règle de typage du *case* est assez complexe. Nous posons tout d'abord la notation suivante plus concise pour représenter le type des branches :
- $\{c : (I_i \ p_1 \ ... \ p_r \ t_1 \ ... \ t_p)\}^P$ est équivalent à $P \ t_1 \ ... \ t_p \ c$;
- $\{c : \forall x : T, C\}^P$ est équivalent à $\forall x : T, \{(c \ x) : C\}^P$.

$$1 \leq i \leq l \ \frac{E; G \vdash_{CCI} c : I \ q_1 \ ... \ q_r \ t_1 \ ... \ t_s \qquad E; G \vdash_{CCI} f_i : \{(c_{p_i} \ q_1 \ ... \ q_r)\}^P}{E; G \vdash_{CCI} case(c, P, f_1 \mid ... \mid f_l) : P \ t_1 \ ... \ t_s \ c} \ \text{Case-type}$$

avec :
- $Ind[G_P](G_I := G_C) \in E$;
- $I \in G_i$
- c_{p_1}, ..., c_{p_l} sont les seuls constructeurs de I.

$$1 \leq i \leq n \ \frac{E; [f_n : A_n; ...; f_1 : A_1] \ @ \ G \vdash_{CCI} t_i : A_i \qquad E; G \vdash_{CCI} A_i : s_i}{E; G \vdash_{CCI} Fix \ f_i \ [f_1 : A_1 := t_1 \ ; \ ... \ ; \ f_n : A_n := t_n] : A_i} \ \text{Fix-type}$$

À ces règles vont venir s'ajouter des règles de calculs. Ces quatre règles permettent de *réduire* les termes vers des termes plus simples. Elles jouent un rôle direct sur les démonstrations et sur la cohérence du système ; nous en reparlerons en temps voulu.

$$\frac{}{E; G \vdash (\lambda x : T, t)u \rightarrow_\beta t\{u/x\}} \ \beta\text{-r}$$

$$\text{si } t := u : T \in E \ @ \ G \ \frac{}{E; G \vdash t \rightarrow_\delta u} \ \delta\text{-r}$$

$$\frac{}{E; G \vdash case(c_{p_i} \ q_1 \ ... \ q_r \ a_1 \ ... \ a_m, P, f_1 \ ... \ f_n) \rightarrow_\iota f_i \ a_1 \ ... \ a_m} \ \iota\text{-rc}$$

$$1 \leq k \leq n \ \frac{}{E; G \vdash Fix \ f_i \ F \ a_1 \ ... \ a_{ki} \rightarrow_\iota t_i\{f_k / Fix \ f_k \ F\}} \ \iota\text{-rF}$$

avec F de la forme $[f_1 : A_1 : t_1; ...; f_n : A_n := t_n]$.

Soient $\rightarrow_{conv}$ la clôture contextuelle de l'union des trois relations précédentes, $\rightarrow^*_{conv}$ sa clôture contextuelle, réflexive et transitive et enfin $=_{conv}$ sa clôture contextuelle, réflexive, symétrique et transitive.

On observe qu'il existe des termes dont différents sous-termes peuvent être réduits par $\rightarrow_{conv}$.

Exemple : dans $(\lambda x : T, x) \ ((\lambda y : T, y) \ z)$, on peut appliquer la β-réduction au terme tout entier ou bien au sous-terme $(\lambda y : T, y) \ z$.

Théorème de normalisation forte : Soient t un terme du CCI bien typé dans un environnement global E et un contexte local G. Il existe un unique u tel que $E;G \vdash t \rightarrow^*_{conv} u$ et u est irréductible pour les des règles de réduction. De plus, quelle que soit la stratégie d'application des règles, la réduction termine.

Théorème de Church-Rosser : Soient t_1, t_2 et u des termes, E un environnement global et G un contexte local. Si $E;G \vdash t_1 =_{conv} t_2$, et que u est a forme normale de t_1, alors u est la forme normale de t_2.

Des preuves de ces deux théorèmes se trouvent dans [21].

Lorsque $E;G \vdash t \rightarrow^*_{conv} u$ et u ne peut pas être réduit, on dit que u est la forme normale de t, et lorsque $E;G \vdash t =_{conv} u$, on dit que t et u sont convertibles.

Le calcul de la forme normale est décidable. Nous noterons $E;G \vdash t \rightarrow_{NF} u$ le fait que u est la forme normale de t dans $E;G$.

La relation de conversion permet alors d'introduire une nouvelle règle de typage qui signifie que deux types convertibles ont les mêmes éléments.

Pour cela, nous avons tout d'abord besoin d'introduire une relation d'ordre de conversion $\leq_{conv}$ définie comme suit :

- Si $E;G \vdash t =_{conv} u$ alors $E;G \vdash t \leq_{conv} u$.
- $E;G \vdash Prop \leq_{conv} TypeSet$.
- $E;G \vdash Set \leq_{conv} TypeSet$.
- Si $E;G \vdash T =_{conv} U$ et $E;G \vdash T' \leq_{conv} U'$, alors $E;G \vdash \forall x : T, T' \leq_{conv} \forall x : U, U'$.

La règle de conversion est alors :

$$\frac{E;G \vdash_{CCI} U : s \qquad E;G \vdash_{CCI} t : T \qquad E;G \vdash T \leq_{conv} U}{E;G \vdash_{CCI} t : U}\ \text{Conv}$$

1.2.4 Les démonstrations

Nous avons ainsi à notre disposition les règles de typage et de calcul du CCI, nous aimerions maintenant, comme cela était le cas pour la logique du premier ordre multisortée, disposer de règles de déduction pour réaliser des démonstrations sur les propositions de ce langage. Parler de ces règles va être simple : il n'y en a pas, ou plutôt, nous les avons déjà énoncées. En effet, dans ce langage les règles de déduction sont confondues avec les règles de typage. Ceci est dû à l'*isomorphisme de Curry-Howard* que nous allons expliquer maintenant.

Commençons par considérer la règle de typage *Lam* dans le cas d'un produit non dépendant et où $s = Prop$. La règle devient alors :

$$\frac{E;G \vdash_{CCI} T \rightarrow U : Prop \qquad E;(x:T)::G \vdash_{CCI} t : U}{E;G \vdash_{CCI} \lambda x : T, t : T \rightarrow U}\ \text{Lam}$$

On remarque tout d'abord par la prémisse de gauche que les propositions ont un type qui s'appelle *Prop*. De plus, par *Prod-Prop*, on sait que $E; G \vdash_{CCI} U : Prop$. Mais alors que peut signifier cet objet t qui est de type U d'après la prémisse de droite ? Si on suit la logique qui dit qu'un type n'est ni plus ni moins qu'un ensemble d'objets, alors les propositions deviendraient des ensembles. Maintenant, réécrivons cette même règle en ne considérant que le type des objets :

$$\frac{E; G \vdash_{CCI} Prop \qquad E; T :: G \vdash_{CCI} U}{E; G \vdash_{CCI} T \rightarrow U}$$

On voit naturellement apparaître le *imp-intro* de la logique du premier ordre. En français, cette règle pourrait se traduire par « s'il existe une preuve de U en supposant T, alors il existe une preuve de $T \rightarrow U$ ». Il suffit donc de définir les propositions comme des ensembles dont les objets en sont les preuves, et dans ce cas, dire que t a le type U signifie simplement qu'il existe une preuve de U et que le terme de preuve correspondant est t. Les règles de typage du CCI sont également des règles de constructions de termes de preuve.

C'est ce paradigme des propositions comme types que l'on appelle l'isomorphisme de Curry-Howard. Cet isomorphisme nous dit que déterminer une preuve d'une certaine proposition est équivalent à déterminer un terme dont le type est la proposition en question.

Si les connecteurs $\forall$ et $\rightarrow$ se trouvent initialement dans la grammaire du langage, qu'en est-il des autres connecteurs logiques ? En fait, il n'est pas besoin de les poser de façon primitive : ils peuvent être définis au sein même du langage comme suit.

Définition de *True*, également noté $\top$:
$Ind[](True : Prop := I : \top)$

$\top$ n'a donc qu'une seule preuve : I. Pourtant on pourrait trouver un terme différent de I qui soit une preuve de $\top$. Il suffit pour cela, par exemple, de prendre le terme constitué de la fonction identité sur les propositions, à laquelle l'on applique I : $(\lambda x : Prop, x)\ I$. Effectivement. Sauf que pour typer ce terme en $\top$, il faut utiliser la règle de conversion. Le fait est que lorsque l'on parle d'un terme, on en parle modulo convertibilité. Ainsi les termes convertibles sont confondus (ils se calculent vers le même objet : leur forme normale commune) et I est bien l'unique preuve de $\top$.

Définition de *False*, également noté $\bot$:
$Ind[](False : Prop :=)$

Il semble évident que $\bot$ ne doit pas avoir de preuve pour satisfaire la cohérence du système.

Définition de *and*, noté $\wedge$ en infixe :
$Ind[A : Prop\,;\, B : Prop](and : Prop := conj : A \rightarrow B \rightarrow A \wedge B)$

Définition de *or*, noté $\vee$ en infixe :
$Ind[A : Prop\,;\, B : Prop](or : Prop := or\_introl : A \rightarrow A \vee B \mid or\_intror : A \rightarrow A \vee B)$

Définition de *ex*, également noté $\exists$:
$Ind[A : TypeSet\,;\, P : A \rightarrow Prop](ex : Prop := ex\_intro : \forall x : A, P\ x \rightarrow ex\ P)$

Dans cette définition, P a pour type $A \rightarrow Prop$. Dans le cas où P est de la forme $\lambda x : A, Q$ où Q a pour type $Prop$, alors $ex\ P$, qui est équivalent à $ex\ (\lambda x : A, Q)$, pourra être noté $\exists x : A, Q$.

Définition de not, également noté $\neg$:
$\neg := \lambda A : Prop, A \rightarrow \bot$
Le $\neg$ n'est donc en réalité qu'un alias.

On peut également définir l'égalité eq, noté $=$ en infixe, par le biais des types inductifs :
$Ind[A : TypSet\,;\, x : A](eq : A \rightarrow Prop := refl\_equal : x = x)$

On retrouve ainsi la sémantique de l'introduction des connecteurs logiques. Leur élimination est une conséquence directe de leur définition et de l'élimination forte. Nous allons donc donner les termes et les types de l'élimination des connecteurs :

$True\_ind := \lambda P : Prop, \lambda f : P, \lambda t : True, case(t, P, I \Rightarrow f)$
Donc dans un environnement où $True$ est déclaré comme précédemment, $True\_ind$ aura pour type : $\forall P : Prop, P \rightarrow True \rightarrow P$.

$False\_ind := \lambda P : Prop, \lambda f : False, case(f, P, [])$
Puisque $False$ n'a aucun constructeur, alors un filtrage sur une preuve de $False$ a la liste vide pour troisième argument. Dans un environnement où $False$ est déclaré comme précédemment, $False\_ind$ aura donc pour type : $\forall P : Prop, False \rightarrow P$.

$and\_ind := \lambda A : Prop, \lambda B : Prop, \lambda P : Prop, \lambda f : A \rightarrow B \rightarrow P, \lambda h : A \wedge B, case(h, P, conj\ A\ B\ a\ b \Rightarrow f\ a\ b)$
Donc dans un environnement où and est déclaré comme précédemment, and_ind aura pour type : $\forall A : Prop, \forall B : Prop, \forall P : Prop, A \rightarrow B \rightarrow P \rightarrow A \wedge B \rightarrow P$.

$or\_ind := \lambda A : Prop, \lambda B : Prop, \lambda P : Prop, \lambda h_1 : A \rightarrow P, \lambda h_2 : B \rightarrow P, \lambda h : A \vee B, case(h, P, or\_introl\ A\ B\ a \Rightarrow h_1\ a \mid or\_intror\ A\ B\ b \Rightarrow h_2\ b)$
Donc dans un environnement où or est déclaré comme précédemment, or_ind aura pour type : $\forall A : Prop, \forall B : Prop, \forall P : Prop, (A \rightarrow P) \rightarrow (B \rightarrow P) \rightarrow A \vee B \rightarrow P$.

$ex\_ind := \lambda A : TypeSet, \lambda P : A \rightarrow Prop, \lambda P_0 : Prop, \lambda f : \forall x : A, P\ x \rightarrow P_0, \lambda e : ex\ P, case(e, P_0, ex\_intro\ A\ P\ x\ P_x \Rightarrow f\ x\ P_x)$
Donc dans un environnement où ex est déclaré comme précédemment, ex_ind aura pour type : $\forall A : TypeSet, \forall P : A \rightarrow Prop, \forall P_0 : Prop, (\forall x : A, P\ x \rightarrow P_0) \rightarrow ex\ P \rightarrow P_0$.

Enfin, il existe également un destructeur de l'égalité mais dont le terme est assez complexe. Nous ne donnons ici que son type : $eq\_ind : \forall A : TypSet, \forall x : A, \forall P : A \rightarrow Prop, P\ x \rightarrow \forall y : A, x = y \rightarrow P\ y$

Nous disposons à présent des schémas d'introduction et d'élimination des connecteurs logiques. À partir de maintenant, nous supposerons que tout environnement global contient ces définitions. Donnons alors un exemple de preuve dans le CCI.

Exemple : montrer que $\forall A : Prop, \forall B : Prop, A \wedge B \rightarrow A \vee B$.

Typiquement, montrer une telle proposition en écrivant un arbre de preuve complet est fastidieux et les parties intéressantes de la preuve se perdent dans les parties triviales. Le but du jeu est plutôt de comprendre l'isomorphisme de Curry-Howard afin de construire le terme de preuve « à la main ».

Par exemple, on peut commencer par observer que la proposition commence par trois $\forall$ successifs (rappelons-nous que le $\rightarrow$ est un cas particulier de $\forall$). Ainsi le terme de preuve de cette proposition commencera par $\lambda A : Prop, \lambda B : Prop, \lambda h : A \wedge B$. Pour démontrer $A \vee B$, le plus simple est encore d'essayer d'utiliser la définition du $\vee$ et donc l'un des deux constructeurs or_introl ou or_intror. Selon que l'on choisisse l'un ou l'autre, nous serons ensuite amenés à trouver un terme de preuve pour A ou pour B, ce qui intuitivement semble possible dans les deux cas vu que l'on dispose d'une preuve de $A \wedge B$. Choisissons de montrer A. Alors notre terme de preuve se complète en $\lambda A : Prop, \lambda B : Prop, \lambda h : A \wedge B, or\_introl\ A\ B\ t$, où t est une preuve de A. Reste donc à déterminer t. Pour avoir une preuve de A à partir de $A \wedge B$, on aimerait pouvoir « détruire » ou plutôt « éliminer » le $\wedge$. Pour cela, on dispose déjà de la fonction and_ind (on pourrait également passer par un filtrage, mais celui-ci est sous-entendu dans and_ind). Si $u := \lambda a : A, \lambda b : B, a$, on a bien $u : A \rightarrow B \rightarrow A$. Alors, le terme and_ind $A\ B\ A\ u\ h$ a bien le type A. Finalement, il suffit de remplacer t par ce terme pour obtenir une preuve de notre proposition initiale : $\lambda A : Prop, \lambda B : Prop, \lambda h : A \wedge B, or\_introl\ A$ $B\ (and\_ind\ A\ B\ A\ u\ h)$ est de type $\forall A : Prop, \forall B : Prop, A \wedge B \rightarrow A \vee B$.

Chapitre 2

Du CCI au premier ordre multisorté

2.1 Traduction

Maintenant que l'on a décrit les deux logiques mises en jeu, nous allons tenter, autant que possible, de traduire les expressions de la logique la plus riche vers la logique la moins expressive.

Cependant, il ne faut pas perdre de vue notre objectif : pouvoir utiliser des algorithmes de décision sur une formule se trouvant *a priori* dans un contexte d'ordre supérieur. Ainsi, notre traduction devra impérativement conserver la *typabilité* — le fait d'être bien typé ou non — ; de plus, la *prouvabilité* — le fait d'être prouvable ou non — doit rester correcte. Ce sont les théorèmes qui seront démontrés dans la deuxième section de ce chapitre.

Si la traduction des expressions semble assez évidente (le $\wedge$ du CCI se traduira naturellement en le $\wedge$ du premier ordre), il faut s'assurer que certaines conditions sont effectivement respectées. Et c'est précisément déterminer les conditions à satisfaire qui constitue toute la difficulté du projet, puisque ce sont de celles-ci que découlent les théorèmes de typabilité et de correction.

Il est à noter le rôle primordial des connecteurs logiques : ceux-ci existent en logique du premier ordre multisorté mais ne peuvent pas être défini dans ce langage. Ils sont primitifs. Ainsi, même si ce n'est pas le cas de leur contrepartie d'ordre supérieur, il faudra les traiter particulièrement. On suppose donc que les environnements globaux contiennent tous la déclaration de ces objets.

Ce que l'on veut pouvoir faire à travers la traduction, c'est d'obtenir un jugement de preuve de premier ordre à partir d'un jugement de preuve du CCI. C'est pourquoi nous allons avoir besoin de traduire différents objets.

Quels que soient ces objets, nous avons choisi de représenter les traductions par des règles d'inférence. Des *jugements de traduction* seront définis dans chaque cas.

Enfin, avant de commencer à énoncer les *règles de traduction*, nous introduisons les notations suivantes pour gagner en clarté :

- $\forall_{k=i}^{n} x_k : s_k, P$ pour $\forall x_i : s_i, ..., \forall x_n : s_n, P$;
- $\lambda_{k=i}^{n} x_k : T_k, u$ pour $\lambda x_i : T_i, ..., \lambda x_n : T_n, u$

Tout d'abord, les sortes en premier ordre sont des objets différents des types que l'on trouve dans le CCI. On doit pouvoir tester si un type est une sorte. Typiquement, c'est le cas des types inductifs sans paramètres, des déclarations de types et de leurs alias.

Le jugement utilisé pour la traduction d'un type du CCI en sorte du premier ordre lorsque ceci est possible aura la forme : $E; G \vdash T \Rightarrow_{Sort} s$ qui signifie que dans l'environnement global E et dans le contexte local G, le type T du CCI se traduit en la sorte s du premier ordre. Les règles sont les suivantes :

$$\text{si } c : Set \in E \; \frac{}{E; G \vdash c \Rightarrow_{Sort} c} \text{ sa-trad}$$

$$\frac{E; G \vdash c \rightarrow_{NF} T \qquad E; G \vdash T \Rightarrow_{Sort} s}{E; G \vdash c \Rightarrow_{Sort} s} \text{ sd-trad}$$

Une stratégie d'application de ces règles qui rend décidable la traduction d'un type en sorte serait de commencer par essayer la règle *sa-trad* et, si elle échoue, d'utiliser alors la règle *sd-trad*. Si cette dernière échoue également, c'est que le type n'est pas une sorte.

Un jugement de traduction des termes à la forme $E; G \vdash t \Rightarrow_{Term} t'$ et signifie naturellement que le terme t du CCI se traduit en le terme t' du premier ordre dans l'environnement global E et le contexte local G.

Pour la traduction des termes, une règle est suffisante. Elle regroupe à la fois la traduction des variables, des constantes, mais également des applications totales de fonction. De plus, pour garantir que le terme tout entier est de premier ordre, il faut s'assurer que le type de retour est une sorte. En revanche, il n'est pas utile de vérifier que le type des arguments est une sorte ; le fait que ces arguments se traduisent en terme le garantit, comme le prouve le **Lemme 11** de la section suivante.

$$\text{si } f :! T_1 \rightarrow ... \rightarrow T_n \rightarrow T \in E \mathbin{@} G,\ 0 \leq i \leq n \; \frac{E; G \vdash t_i \Rightarrow_{Term} t'_i \qquad E; G \vdash T \Rightarrow_{Sort} s}{E; G \vdash f\ t_1\ ...\ t_n \Rightarrow_{Term} f(t'_1, ..., t'_n)} \text{ f-trad}$$

Le prochaine étape est la traduction des propositions. On dispose donc de jugements de la forme $E; G \vdash P \Rightarrow_{Prop} P'$ qui signifie que la proposition P du CCI se traduit en la proposition P' du premier ordre dans l'environnement global E et dans le contexte local G. Comme pour les termes, on ne vérifie pas avant traduction que P est bien de type *Prop*. C'est la structure des objets que l'on traduit via ces règle qui assure que ce sont bien des propositions. L'énoncé et la preuve correspondent au **Lemme 15** de la section suivante.

$$\text{si } P :! T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop \in E,\ 0 \leq i \leq n \; \frac{E; G \vdash t_i \Rightarrow_{Term} t'_i}{E; G \vdash P\ t_1\ ...\ t_n \Rightarrow_{Prop} P(t'_1, ..., t'_n)} \text{ p-trad}$$

$$\frac{E; G \vdash T \Rightarrow_{Sort} s \qquad E; G \vdash t_1 \Rightarrow_{Term} t'_1 \qquad E; G \vdash t_2 \Rightarrow_{Term} t'_2}{E; G \vdash eq\ T\ t_1\ t_2 \Rightarrow_{Prop} t'_1 = t'_2} \text{ eq-trad}$$

$$\frac{}{E; G \vdash \top \Rightarrow_{Prop} \top} \text{ top-trad}$$

$$\frac{}{E;G \vdash \bot \Rightarrow_{Prop} \bot}\text{ bot-trad}$$

$$\frac{E;G \vdash P \Rightarrow_{Prop} P' \qquad E;G \vdash Q \Rightarrow_{Prop} Q'}{E;G \vdash P \wedge Q \Rightarrow_{Prop} P' \wedge Q'}\text{ and-trad}$$

$$\frac{E;G \vdash P \Rightarrow_{Prop} P' \qquad E;G \vdash Q \Rightarrow_{Prop} Q'}{E;G \vdash P \vee Q \Rightarrow_{Prop} P' \vee Q'}\text{ or-trad}$$

$$\frac{E;G \vdash P \Rightarrow_{Prop} P' \qquad E;G \vdash Q \Rightarrow_{Prop} Q'}{E;G \vdash P \rightarrow Q \Rightarrow_{Prop} P' \rightarrow Q'}\text{ imp-trad}$$

$$\frac{E;G \vdash P \Rightarrow_{Prop} P'}{E;G \vdash \neg P \Rightarrow_{Prop} \neg P'}\text{ not-trad}$$

$$\frac{E;G \vdash t_1 \Rightarrow_{Term} t_1' \qquad E;G \vdash t_2 \Rightarrow_{Term} t_2'}{E;G \vdash t_1 = t_2 \Rightarrow_{Prop} t_1' = t_2'}\text{ eq-trad}$$

$$\frac{E;G \vdash T \Rightarrow_{Sort} s \qquad E;(x:T) :: G \vdash P \Rightarrow_{Prop} P'}{E;G \vdash \forall x:T, P \Rightarrow_{Prop} \forall x:s, P'}\text{ forall-trad}$$

$$\frac{E;G \vdash T \Rightarrow_{Sort} s \qquad E;(x:T) :: G \vdash P \Rightarrow_{Prop} P'}{E;G \vdash \exists x:T, P \Rightarrow_{Prop} \exists x:s, P'}\text{ exists-trad}$$

On peut maintenant poser les règles de traduction des environnements. Ils ont la forme $E;G \Rightarrow_{Env} S;\Gamma;\Delta$ qui signifie que l'environnement global E et le contexte local G se traduisent en l'ensemble de sortes S, l'environnement Γ et le contexte Δ.

$$\frac{E;G \Rightarrow_{Env} S,\Gamma,\Delta}{(c:Set) :: E;G \Rightarrow_{Env} \{c\} \cup S,\Gamma,\Delta}\text{ ta-trad}$$

$$\frac{E;G \vdash t \Rightarrow_{Sort} s \qquad E;G \Rightarrow_{Env} S,\Gamma,\Delta}{(c := t : Set) :: E;G \Rightarrow_{Env} \{s\} \cup S,\Gamma,\Delta}\text{ tda-trad}$$

Les règles ci-dessus sont telles que l'on traduit un type par la traduction de sa forme normale. Si un type et son alias sont traduits, en réalité on ne trouvera qu'un seul objet pour les deux dans l'ensemble de sortes.

$$0 \leq i \leq n+1\ \frac{E;G \vdash T_i \Rightarrow_{Sort} s_i \qquad E;G \Rightarrow_{Env} S,\Gamma,\Delta}{(c:T_1 \rightarrow \ldots \rightarrow T_{n+1}) :: E;G \Rightarrow_{Env} S,\{c:s_1,\ldots,s_n \rightarrow s_{n+1}\} \cup \Gamma,\Delta}\text{ fa-trad}$$

$$\frac{E;G \vdash T_j \Rightarrow_{Sort}\ s_j \qquad E;G \Rightarrow_{Env}\ S;\Gamma;\Delta \qquad E;[x_1 : T_1;...;x_n : T_n]\ @\ G \vdash u\ x_1\ ...\ x_n \Rightarrow_{Term}\ t}{((c := \lambda_{k=1}^{i} x_k : T_k, u\ x_1\ ...\ x_i : T_1 \rightarrow ... \rightarrow T_{n+1}) :: E;G \Rightarrow_{Env}\ S;\{c : s_1,...,s_n \rightarrow s_{n+1}\} \cup \Gamma;\{\forall_{k=1}^{n} x_k : s_k, c(x_1,...,x_n) = t\} \cup \Delta}\ \text{fda-trad} \quad 0 \leq i \leq n,\ 0 \leq j \leq n+1$$

La règle *fda-trad* prend en compte les définitions de fonction par application partielle. Pour rendre une telle définition premier ordre, il est donc nécessaire d'η-expanser (rajouter des arguments à la main) la définition afin d'obtenir une application totale.

$$\frac{E;G \vdash T_i \Rightarrow_{Sort}\ s_i \qquad E;G \Rightarrow_{Env}\ S,\Gamma,\Delta}{(c : T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop) :: E;G \Rightarrow_{Env}\ S,\{c : [s_1;...;s_n]\} \cup \Gamma,\Delta}\ \text{pa-trad} \quad 0 \leq i \leq n$$

$$\frac{E;G \vdash T_i \Rightarrow_{Sort}\ s_i \qquad E;G \Rightarrow_{Env}\ S;\Gamma;\Delta \qquad E;[x_1 : T_1;...;x_n : T_n]\ @\ G \vdash u\ x_1\ ...\ x_n \Rightarrow_{Prop}\ P}{((c := \lambda_{k=1}^{i} x_k : T_k, u\ x_1\ ...\ x_i : T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop) :: E;G \Rightarrow_{Env}\ S;\{c : [s_1;...;s_n]\} \cup \Gamma;\{\forall_{k=1}^{n} x_k : s_k, c(x_1,...,x_n) \Leftrightarrow P\} \cup \Delta}\ \text{pda-trad} \quad 0 \leq i \leq n$$

$$\frac{E;G \vdash_{CCI} P : Prop \qquad E;G \vdash P \Rightarrow_{Prop}\ P' \qquad E;G \Rightarrow_{Env}\ S,\Gamma,\Delta}{(c : P) :: E;G \Rightarrow_{Env}\ S,\Gamma,\{P'\} \cup \Delta}\ \text{aa-trad}$$

$$\frac{E;G \Rightarrow_{Env}\ S;\Gamma;\Delta}{d :: E;G \Rightarrow_{Env}\ S;\Gamma;\Delta}\ \text{ska-trad}$$

La règle *ska-trad* peut donc s'appliquer dans n'importe quel cas pour peu que E ne soit pas vide. Pour décider de la traduction d'un environnement global et d'un contexte local, on peut tout d'abord essayer les règles excepté *ska-trad* (les domaines de traduction des autres règles sont deux-à-deux disjoints) et, si aucune règle n'est applicable, d'utiliser alors *ska-trad* qui peut toujours être utilisée quel que soit un environnement global non vide.

$$\frac{E;G \vdash T \Rightarrow_{Sort}\ s \qquad E;G \Rightarrow_{Env}\ S,\Gamma,\Delta}{E;(c : T) :: G \Rightarrow_{Env}\ S,\{c : s\} \cup \Gamma,\Delta}\ \text{vc-trad}$$

Le fait est que par la traduction des environnement globaux et des contextes locaux, seules des variables de type une sorte sont insérées dans les contextes locaux, c'est pourquoi on ne dispose que d'une règle pour la traduction de ces déclarations.

$$\frac{G_P\ @\ C\ @\ I\ @\ E;G \Rightarrow_{Env}\ S;\Gamma;\Delta}{Ind[G_P](I := C) :: E;G \Rightarrow_{Env}\ S;\Gamma;(Injection(G_P\ @\ I\ @\ E,C)) \cup \Delta}\ \text{ind-trad}$$

La traduction d'un bloc de types inductifs est en réalité assez simple : on ne considère les constructeurs que comme des objets du type qui les suit. Le fait que plusieurs inductifs soient déclarés dans un même bloc pour satisfaire des propriétés de terminaison et de cohérence n'intervient pas dans notre traduction. Nous ne traduirons de toute façon que des jugements où les environnements globaux et les contexte locaux sont bien formés, et les termes bien typés.

En revanche, certaines propriétés peuvent être déduites de l'élimination forte lors de la déclaration d'un type inductif. L'une de ces propriétés est l'injection sur chaque constructeur. Ces axiomes d'injection sont ajoutés au contexte premier ordre par la fonction *Injection* définie ainsi :

- $\mathit{Injection}(E, []) = \emptyset$
- $\mathit{Injection}(E, ((c : C) :: l)) =$
  - si $C = T_1 \rightarrow ... \rightarrow T_{n+1}$ et que $E; [] \vdash T_i \Rightarrow_{\mathit{Sort}} s_i$ $(0 \leq i \leq n+1)$, alors $\{\forall_{i=1}^{n} x_i : s_i, \forall_{i=1}^{n} y_i : s_i, c(x_1, ..., x_n) = c(y_1, ..., y_n) \rightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n\} \cup (\mathit{Injection}(E, l))$
  - sinon $\mathit{Injection}(E, l)$

$$\frac{\begin{array}{c} E; G \Rightarrow_{\mathit{Env}} S; \Gamma; \Delta \qquad E; G \vdash T_v \Rightarrow_{\mathit{Sort}} s_v \\ (f : T_1 \rightarrow ... \rightarrow T_{n+1}) :: E; G \vdash \\ \forall_{k=1}^{w-1} x_k : T_k, \forall_{k=w+1}^{n}, x_k : T_k, \forall_{k=1}^{z_w} y_k : C_k, f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n = f_w \Rightarrow_{\mathit{Prop}} P_w \end{array}}{\begin{array}{c} (f := \mathit{Fix}\ f\ (f : T_1 \rightarrow ... \rightarrow T_{n+1} := \lambda_{k=1}^{i} x_k : T_k, \mathit{case}(x_j, T_{n+1}, F))) :: E; G \Rightarrow_{\mathit{Env}} \\ S; \{f : s_1, ..., s_n \rightarrow s_{n+1}\} \cup \Gamma; \{P_1 ; ... ; P_l\} \cup \Delta \end{array}} \text{ fix-trad}$$

avec :

- $1 \leq v \leq n+1$, $0 \leq w \leq l$, $1 \leq i \leq n$, $1 \leq j \leq n$;
- $F = c_1\ y_1\ ...\ y_{z_1} \Rightarrow f_1 \mid ... \mid c_l\ y_1\ ...\ y_{z_l} \Rightarrow f_l$;
- $T_j \in G_I$, $G_C = c_1 : \forall_{k=1}^{z_1 - 1} x_k : C_k, C_{z_1} \mid ... \mid c_C : \forall_{k=1}^{z_C - 1} x_k : C_k, C_{z_C}$ et $\mathit{Ind}[G_P](G_I := G_C) \in E$.

Cette règle peut facilement s'étendre à la définition d'une fonction d'un bloc de fonctions mutuellement récursives en ajoutant toutes les déclarations des fonctions à Γ dans la traduction et à E dans la prémisse, ainsi que les prémisses correspondantes et les ajouts à Δ.

$$\frac{}{[]; [] \Rightarrow_{\mathit{Env}} \emptyset; \emptyset; \emptyset} \text{ empty-trad}$$

La règle *empty-trad* représente l'arrêt de la traduction, c'est-à-dire lorsqu'il n'y a plus rien à traduire.

Pour finir, une stratégie décidable d'application des règles de traduction des environnements globaux et des contextes locaux serait d'utiliser autant que possible les règles concernant les environnements globaux pour finir par celle concernant les contextes locaux.

Exemple : traduisons la proposition $P = \forall x : Int, \forall y : Int, x = y \to y = x$ dans $E; G = [0 : Int; S : Int \to Int; Int : Set]; []$.

On commence par traduire $E; G$.

$$\dfrac{\dfrac{}{[S : Int \to Int; Int : Set]; [] \vdash Int \Rightarrow_{Sort} Int}\,\text{sa-trad} \qquad \pi_{aux}}{E; G \Rightarrow_{Env} \{Int\}; \{0 : Int; S : Int \to Int\}; \emptyset}\,\text{fa-trad}$$

$\pi_{aux} =$

$$\dfrac{\pi_{Int} \qquad \pi_{Int} \qquad \dfrac{\dfrac{}{[]; [] \Rightarrow_{Env} \emptyset; \emptyset; \emptyset}\,\text{empty-trad}}{[Int : Set]; [] \Rightarrow_{Env} \{Int\}; \emptyset; \emptyset}\,\text{ta-trad}}{[S : Int \to Int; Int : Set]; [] \Rightarrow_{Env} \{Int\}; \{S : Int \to Int\}; \emptyset}\,\text{fa-trad}$$

$\pi_{Int} =$

$$\dfrac{}{[Int : Set]; [] \vdash Int \Rightarrow_{Sort} Int}\,\text{sa-trad}$$

Alors, on traduit P.

$$\dfrac{\pi_{Int} \qquad \dfrac{\dfrac{}{E; [x : Int] \vdash Int \Rightarrow_{Sort} Int}\,\text{sa-trad} \qquad \pi_1}{E; [x : Int \vdash \forall y : Int, x = y \to y = x \Rightarrow_{Prop} \forall y : Int, x = y \to y = x}\,\text{forall-trad}}{E; G \vdash P \Rightarrow_{Prop} \forall x : Int, \forall y : Int, x = y \to y = x}\,\text{forall-trad}$$

$\pi_1 =$

$$\dfrac{\dfrac{\pi'_{Int} \qquad \pi_x \qquad \pi_y}{E; G' \vdash x = y \Rightarrow_{Prop} x = y}\,\text{eq-trad} \qquad \dfrac{\pi'_{Int} \qquad \pi_y \qquad \pi_x}{E; G' \vdash y = x \Rightarrow_{Prop} y = x}\,\text{eq-trad}}{E; G' = [y : Int; x : Int] \vdash x = y \to y = x \Rightarrow_{Prop} x = y \to y = x}\,\text{imp-trad}$$

$\pi_x =$

$$x : Int \in E \mathbin{@} G' \;\dfrac{\pi'_{Int}}{E; G' \vdash x \Rightarrow_{Term} x}\,\text{fa-trad}$$

$\pi_y =$

$$y : Int \in E \mathbin{@} G' \;\dfrac{\pi'_{Int}}{E; G' \vdash y \Rightarrow_{Term} y}\,\text{fa-trad}$$

$\pi'_{Int} =$

$$\dfrac{}{E; G' \vdash Int \Rightarrow_{Sort} Int}\,\text{sa-trad}$$

Ceci étant fait, nous voudrions trouver une preuve de $\{Int\}; \{0 : Int; S : Int \to Int\}; \emptyset \models \forall x : Int, \forall y : Int, x = y \to y = x$ et en déduire qu'il existe h tel que $E; G \vdash_{CCI} h : P$. C'est ce que nous allons développer dans la section suivante.

2.2 Typabilité, Correction

Pour finir notre travail de traduction, il faut pouvoir déduire d'une preuve de traduction qu'il existe une preuve du traduit. Deux théorèmes seront donc fondamentaux : celui de typabilité qui assure que la traduction d'une expression bien typée reste bien typée, et celui de prouvabilité qui associe aux preuves de premier ordre des termes du CCI.

Remarque : Soit T un type. Il est facile de se convaincre que si T est une sorte, alors T se traduit de façon unique. Pour s'en convaincre il suffit de remarquer que pour traduire un type, on traduit en réalité sa forme normale, qui se traduit en elle-même si sa traduction existe.

Lemme 11 : Soient t un terme du CCI, t' un terme du premier ordre, T une sorte, E un environnement global et G un contexte local. Si $E; G \vdash_{CCI} t : T$ et que $E; G \vdash t \Rightarrow_{Term} t'$ alors il existe une sorte s telle que $E; G \vdash T \Rightarrow_{Sort} s$.

Preuve : On ne peut avoir $E; G \vdash t \Rightarrow_{Term} t'$ qu'en utilisant la règle *f-trad* si bien que t a la forme $f\ t_1\ ...\ t_n$, que $f :!\ T_1 \rightarrow ... \rightarrow T_{n+1} \in E\ @\ G$ et que $E; G \vdash T_{n+1} \Rightarrow_{Sort} s$. On en déduit que $E; G \vdash_{CCI} f : T_1 \rightarrow ... \rightarrow T_n \rightarrow T_{n+1}$ et donc que $E; G \vdash_{CCI} f\ t_1\ ...\ t_n : T_{n+1}$. Mais on sait par hypothèse que $E; G \vdash_{CCI} t : T$, c'est-à-dire que $E; G \vdash_{CCI} f\ t_1\ ...\ t_n : T$. C'est donc que $T_{n+1} =_{conv} T$. Puisque T et T_{n+1} sont convertibles, ils ont donc la même forme normale, et c'est de cette forme normale que l'on peut déterminer $E; G \vdash T_{n+1} \Rightarrow_{Sort} s$ par les règles *sa-trad* et *sd-trad*. Par les mêmes règles, on en déduit que $E; G \vdash T \Rightarrow_{Sort} s$.

Lemme 12 : Soient t un terme du CCI, t' un terme du premier ordre, T un type du CCI, s une sorte, E un environnement global, G un contexte local, S un ensemble de sortes, Γ un environnement et Δ un contexte. Si $E; G \vdash_{CCI} t : T$, $E; G \vdash T \Rightarrow_{Sort} s$, $E; G \vdash t \Rightarrow_{Term} t'$ et $E; G \Rightarrow_{Env} S; \Gamma; \Delta$, alors $\Gamma \vdash_{FOL} t' : s$.

Preuve : par récurrence sur la taille de t

- Si t est une variable ou une constante, on ne peut avoir $E; G \vdash t \Rightarrow_{Term} t'$ que par la règle *f-trad* qui implique dans ce cas-là que $E; G \vdash t \Rightarrow_{Term} t$, $t :!\ T' \in E\ @\ G$ et $E; G \vdash T' \Rightarrow_{Sort} s$. Ainsi, T et T' sont convertibles et $E; G \vdash T \Rightarrow_{Sort} s$. Donc dans la traduction de $E; G$, on pourra utiliser une des règles *fa-trad*, *fc-trad*, *fda-trad*, *fdc-trad* ou *fix-trad* (selon que l'on ait déclaré ou définit t dans l'environnement global ou dans le contexte local) pour obtenir le même effet : $t : s \in \Gamma$ et donc $\Gamma \vdash_{FOL} t : s$ (*axiome*).
- Si t est de la forme $f\ t_1\ ...\ t_n$, $n > 0$, on ne peut avoir $E; G \vdash t \Rightarrow_{Term} t'$ que par la règle *f-trad* si bien que l'on a $f :!\ T_1 \rightarrow ... \rightarrow T_{n+1} \in E\ @\ G$, $E; G \vdash t_i \Rightarrow_{Term} t_i'$ $(1 \leq i \leq n)$, $E; G \vdash T_{n+1} \Rightarrow_{Sort} s_{n++1}$ et t' est de la forme $f(t_1, ..., t_n)$. Tout d'abord, puisque $E; G \vdash_{CCI} t : T$ et $E; G \vdash_{CCI} t : T_{n+1}$, on sait que T et T_{n+1} sont convertibles. Donc ils se traduisent vers la même sorte (car ils ont alors la même forme normale) et donc $s = s_{n+1}$. D'autre part, $f :!\ T_1 \rightarrow ... \rightarrow T_{n+1} \in E\ @\ G$, donc $E; G \vdash_{CCI} f : T_1 \rightarrow ... \rightarrow T_{n+1}$. Donc, puisque $E; G \vdash_{CCI} f\ t_1\ ...\ t_n : T_{n+1}$ (car T et T_{n+1} sont convertibles), alors $E; G \vdash t_i : T_i$. D'où, puisque $E; G \vdash t_i \Rightarrow_{Term} t_i'$, $E; G \vdash T_i \Rightarrow_{Sort} s_i$ d'après le **Lemme 11**. Donc par hypothèse de récurrence $\Gamma \vdash_{FOL} t_i' : s_i$. Enfin, $f :!\ T_1 \rightarrow ... \rightarrow T_{n+1} \in E\ @\ G$ et $E; G \vdash$

$T_{n+1} \Rightarrow_{Sort} s$, donc par une des règles *fa-trad*, *fda-trad*, *fc-trad*, *fdc-trad* ou *fix-trad*, on obtiendra que $f : s_1, ..., s_n \rightarrow s \in \Gamma$. Donc par *fun-type*, on peut conclure que $\Gamma \vdash_{FOL} f(t'_1, ..., t'_n) : s$.

Théorème 2 : Soient P une proposition du CCI, P' une proposition du premier ordre multisorté, E en environnement global, G un contexte local, S un ensemble de sorte, Γ un environnement et Δ un contexte. Si $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et $E; G \vdash P \Rightarrow_{Prop} P'$ alors $\Gamma \vdash_{FOL} P'\ bf$.

Preuve : par récurrence sur $E; G \vdash P \Rightarrow_{Prop} P'$.

- Si la dernière règle est *top-trad*, $P = \top$ et $P' = \top$. Par *top-type* on a $\Gamma \vdash_{FOL} \top\ bf$ et donc $\Gamma \vdash_{FOL} P'\ bf$.
- Si la dernière règle est *bot-trad*, $P = \bot$ et $P' = \bot$. Par *bot-type* on a $\Gamma \vdash_{FOL} \bot\ bf$ et donc $\Gamma \vdash_{FOL} P'\ bf$.
- Si la dernière règle est *eq-trad*, P est de la forme $t_1 = t_2$, $E; G \vdash t_1 \Rightarrow_{Term} t'_1$, $E; G \vdash t_2 \Rightarrow_{Term} t'_2$, et P' est de la forme $t'_1 = t'_2$. Puisque $E; G \vdash_{CCI} t_1 = t_2 : Prop$, c'est qu'il existe un type T tel que $E; G \vdash_{CCI} t_1 : T$, $E; G \vdash_{CCI} t_2 : T$ par le *type du = du CCI*. Puisque $E; G \vdash_{CCI} t_1 : T$ et que $E; G \vdash t_1 \Rightarrow_{Term} t'_1$ alors par le **Lemme 11** il existe une unique sorte s telle que $E; G \vdash T \Rightarrow_{Sort} s$. Alors, par le **Lemme 12**, $\Gamma \vdash t'_1 : s$ et $\Gamma \vdash t'_2 : s$. Donc, par la règle *eq-type*, on a bien $\Gamma \vdash_{FOL} t'_1 = t'_2\ bf$ ou encore $\Gamma \vdash_{FOL} P'\ bf$.
- Si la dernière règle est *pred-trad*, P est de le forme $R(t_1, ..., t_n)$, $E; G \vdash t_i \Rightarrow_{Term} t'_i$ $(0 \leq i \leq n)$, $R :! T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop \in E$, $E; G \vdash_{CCI} R\ t_1\ ...\ t_n : Prop$ et P' est de la forme $R(t'_1, ..., t'_n)$. Puisque $R :! T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop \in E$, alors $E; G \vdash_{CCI} R : T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop$. Puisque $E; G \vdash_{CCI} R : T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop$ et que $E; G \vdash_{CCI} R\ t_1\ ...\ t_n : Prop$, alors $E; G \vdash_{CCI} t_i : T_i$. Et puisque que $E; G \vdash t_i \Rightarrow_{Term} t'_i$ alors, par le **Lemme 11**, $E; G \vdash T_i \Rightarrow_{Sort} s_i$. Les quatre conditions nécessaires sont réunies pour déduire par le **Lemme 12** que $\Gamma \vdash_{FOL} t'_i : s_i$. De plus, $R :! T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop \in E\ @\ G$ et $E; G \vdash T_i \Rightarrow_{Sort} s_i$, donc lors de la traduction de $E; G$, on pourra utiliser une des règles *pa-trad*, *pc-trad*, *pda-trad* ou *pdc-trad* (selon que l'on ait déclaré ou défini R dans l'environnement global ou dans le contexte) qui auront toutes le même effet : $R : [s_1; ...; s_n] \in \Gamma$. Donc par utilisation de la règle *pred-type*, on obtient que $\Gamma \vdash_{FOL} R(t'_1, ..., t'_n)\ bf$ et alors $\Gamma \vdash_{FOL} P\ bf$.
- Si la dernière règle est *and-trad*, P est de la forme $A \wedge B$, $E; G \vdash A \Rightarrow_{Prop} A'$, $E; G \vdash B \Rightarrow_{Prop} B'$, P est de la forme $A' \wedge B'$ et $E; G \vdash A \wedge B \Rightarrow_{Prop} A' \wedge B'$. Donc par hypothèse de récurrence, on sait que $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. D'où, par la règle *and-type*, $\Gamma \vdash A' \wedge B'\ bf$ et donc $\Gamma \vdash P'\ bf$.
- Si la dernière règle est *or-trad*, P est de la forme $A \vee B$, $E; G \vdash A \Rightarrow_{Prop} A'$, $E; G \vdash B \Rightarrow_{Prop} B'$, P est de la forme $A' \vee B'$ et $E; G \vdash A \vee B \Rightarrow_{Prop} A' \vee B'$. Donc par hypothèse de récurrence, on sait que $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. D'où, par la règle *or-type*, $\Gamma \vdash A' \vee B'\ bf$ et donc $\Gamma \vdash P'\ bf$.
- Si la dernière règle est *imp-trad*, P est de la forme $A \rightarrow B$, $E; G \vdash A \Rightarrow_{Prop} A'$, $E; G \vdash B \Rightarrow_{Prop} B'$, P' est de la forme $A' \rightarrow B'$ et $E; G \vdash A \rightarrow B \Rightarrow_{Prop} A' \rightarrow B'$. Donc par hypothèse de récurrence, on sait que $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. D'où, par la règle *imp-type*, $\Gamma \vdash A' \rightarrow B'\ bf$ et donc $\Gamma \vdash P'\ bf$.

– Si la dernière règle est *not-trad*, P est de la forme $\neg Q$, $E; G \vdash Q \Rightarrow_{Prop} Q'$, P' est de la forme $\neg Q'$ et $E; G \vdash \neg Q \Rightarrow_{Prop} \neg Q'$. Donc par hypothèse de récurrence, on sait que $\Gamma \vdash_{FOL} Q'\ bf$. D'où, par la règle *not-type*, $\Gamma \vdash_{FOL} \neg Q'\ bf$ et donc $\Gamma \vdash_{FOL} P'\ bf$.
– Si la dernière règle est *forall-trad*, P est de la forme $\forall x : T, Q$, $E; G \vdash T \Rightarrow_{Prop} s$, $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$, P' est de la forme $\forall x : s, Q'$ et $E; G \vdash \forall x : T, Q \Rightarrow_{Prop} \forall x : s, Q'$. Puisque $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et que $E; G \vdash T \Rightarrow_{Prop} s$ alors par *vtc-trad* $E; (x : T) :: G \Rightarrow_{Env} S; \{x : s\} \cup \Gamma; \Delta$. De plus, $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$. Donc par hypothèse de récurrence, $\{x : s\} \cup \Gamma \vdash Q'\ bf$. D'où, par la règle *forall-type*, $\Gamma \vdash_{FOL} \forall x : s, Q'\ bf$ et donc $\Gamma \vdash_{FOL} P'\ bf$.
– Si la dernière règle est *exists-trad*, P est de la forme $\exists x : T, Q$, $E; G \vdash T \Rightarrow_{Prop} s$, $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$, P' est de la forme $\exists x : s, Q'$ et $E; G \vdash \exists x : T, Q \Rightarrow_{Prop} \exists x : s, Q'$. Puisque $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et que $E; G \vdash T \Rightarrow_{Prop} s$ alors par *vtc-trad* $E; (x : T) :: G \Rightarrow_{Env} S; \{x : s\} \cup \Gamma; \Delta$. De plus, $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$. Donc par hypothèse de récurrence, $\{x : s\} \cup \Gamma \vdash Q'\ bf$. D'où, par la règle *exists-type*, $\Gamma \vdash_{FOL} \exists x : s, Q'\ bf$ et donc $\Gamma \vdash_{FOL} P'\ bf$.

Lemme 13 : Soient s une sorte, T un type, E un environnement global, E un contexte local, S un ensemble de sortes, Γ un environnement et Δ un contexte. Si $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et que $s \in S$, alors il existe T tel que $E; G \vdash T \Rightarrow_{Sort} s$.

Preuve :
Puisque $E; G \Rightarrow_{Env} S; \Gamma; \Delta$, il n'y a que deux règles qui puissent permettre que $s \in S$: *ta-trad* et *tda-trad*. Dans les deux cas, c'est qu'il existe T tel que $T :! Set \in E$ et $E; G \vdash T \Rightarrow_{Sort} s$.

Lemme 14 : Soient t' un terme du premier ordre, s une sorte, T un type, E un environnement global, G un contexte local, S un ensemble de sortes, Γ un environnement et Δ un contexte. Si $E; G \Rightarrow_{Env} S; \Gamma; \Delta$, $E; G \vdash T \Rightarrow_{Sort} s$, $\Gamma \vdash_{FOL} t' : s$, alors il existe t tel que $E; G \vdash t \Rightarrow_{Term} t'$ avec $E; G \vdash_{CCI} t : T$.

Preuve : par récurrence sur la taille de t.
– Si t' est une variable, alors $T :! Set \in E$ car seules les règles *ta-trad* et *tda-trad* permettent d'ajouter une sorte à l'ensemble de sortes. De plus, on ne peut avoir $\Gamma \vdash_{FOL} t' : s$ que par utilisation de la règle *axiome* si bien que $t' : s \in \Gamma$. Hors, pour qu'une variable soit insérée dans Γ, il faut avoir utilisé *fa-trad* ou *vc-trad* si bien que $t' : T \in E$ ou $t' : T \in G$. Dans les deux cas on a bien $E; G \vdash_{CCI} t' : T$, et $E; G \vdash t' \Rightarrow_{Term} t'$.
– Si t' est une constante, alors $T :! Set \in E$ car seules les règles *ta-trad* et *tda-trad* permettent d'ajouter une sorte à l'ensemble de sortes. De plus, on ne peut avoir $\Gamma \vdash_{FOL} t' : s$ que par utilisation de la règle *axiome* si bien que $t' : s \in \Gamma$. Hors, pour qu'une constante soit insérée dans Γ, il faut avoir utilisé *fa-trad* ou *fda-trad* si bien que $t' : T \in E$. Alors, on a bien $E; G \vdash_{CCI} t' : T$, et $E; G \vdash t' \Rightarrow_{Term} t'$.
– Si t' est de la forme $f(t'_1, ..., t'_n)$, alors $T :! Set \in E$ car seules les règles *ta-trad* et *tda-trad* permettent d'ajouter une sorte à l'ensemble de sortes. De plus, on ne peut avoir $\Gamma \vdash_{FOL} f(t'_1, ..., t'_n) : s$ que par utilisation de la règle *fun-type* si bien que $f : s_1, ..., s_n \rightarrow s \in \Gamma$ et $\Gamma \vdash_{FOL} t'_i : s_i$ pour $0 \leq i \leq n$. Hors, pour qu'un

symbole de fonction soit inséré dans Γ, il faut avoir utilisé l'une des règles *fa-trad*, *fda-trad* ou *fix-trad* si bien que qu'il existe T_i tels que $f : T_1 \rightarrow ... \rightarrow T \in E$ et $E; G \vdash T_i \Rightarrow_{Sort} s_i$. Par hypothèse de récurrence, il existe t_i tels que $E; G \vdash_{CCI} t_i : T_i$ et $E; G \vdash t_i \Rightarrow_{Term} t'_i$. Donc par les règles App et *f-trad*, on a bien $E; G \vdash_{CCI} f\ t_1\ ...\ t_n : T$ et $E; G \vdash f\ t_1\ ...\ t_n \Rightarrow_{Term} f(t'_1, ..., t'_n)$.

Lemme 15 : Soient P' une proposition du premier ordre, E un environnement global, G un contexte local, S un ensemble de sortes, Γ un environnement et Δ un contexte. Si $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et $\Gamma \vdash_{FOL} P'\ bf$ alors il existe P tel que $E; G \vdash P \Rightarrow_{Prop} P'$ et $E; G \vdash_{CCI} P : Prop$.

Preuve : par récurrence sur $\Gamma \vdash_{FOL} P'\ bf$.

- Si la dernière règle est *top-type*, P' a la forme $\top$; alors avec P de la forme $\top$, on obtient bien que $E; G \vdash_{CCI} P : Prop$ mais aussi que $E; G \vdash P \Rightarrow_{Prop} P'$.
- Si la dernière règle est *bot-type*, P' a la forme $\bot$, alors avec P de la forme $\bot$, on obtient bien que $E; G \vdash_{CCI} P : Prop$ mais aussi que $E; G \vdash P \Rightarrow_{Prop} P'$.
- Si la dernière règle est *pred-type*, P' a la forme $R'(t'_1, ..., t'_n)$, $R' : [s_1; ...; s_n] \in \Gamma$ et $\Gamma \vdash_{FOL} t'_i : s_i$ pour $0 \leq i \leq n$. $s_i \in S$, c'est donc qu'il existe T_i tels que $E; G \vdash T_i \Rightarrow_{Sort} s_i$. Puisque $R' : [s_1; ...; s_n] \in \Gamma$ et que $E; G \Rightarrow_{Env} S; \Gamma; \Delta$, c'est qu'il existe R tel que $R :! T_1 \rightarrow ... \rightarrow T_n \rightarrow Prop \in E$ car les deux seules façons d'ajouter un symbole de prédicat à l'environnement par traduction est par utilisation de *pa-trad* ou de *pda-trad*. De plus, par le **Lemme 14**, puisque $E; G \vdash T_i \Rightarrow_{Sort} s_i$ et $\Gamma \vdash_{FOL} t'_i : s_i$, alors il existe t_i tel que $E; G \vdash t_i \Rightarrow_{Term} t'_i$ et $E; G \vdash_{CCI} t_i : T_i$. Si P a la forme $R\ t_1\ ...\ t_n$, on a bien $E; G \vdash P \Rightarrow_{Prop} P'$ par *p-trad*, mais aussi $E; G \vdash_{CCI} P : Prop$ par App.
- Le cas de l'égalité peut être inclus dans celui des prédicats (cas ci-dessus), avec P' de la forme $t'_1 = t'_2$, $\Gamma \vdash_{FOL} t'_i : s$ $(i \in \{1; 2\})$, $E; G \vdash T \Rightarrow_{Sort} s$, $E; G \vdash t_i \Rightarrow_{Term} t'_i$ et P est de la forme $eq\ T\ t_1\ t_2$.
- Si la dernière règle est *and-type*, P' a la forme $A' \wedge B'$, $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. Par hypothèse de récurrence, il existe A et B tels que $E; G \vdash A \Rightarrow_{Prop} A'$, $E; G \vdash B \Rightarrow_{Prop} B'$, $E; G \vdash_{CCI} A : Prop$ et $E; G \vdash_{CCI} B : Prop$. Donc si P est de la forme $A \wedge B$, $E; G \vdash P \Rightarrow_{Prop} P'$ par *and-trad* et $E; G \vdash_{CCI} P : Prop$ d'après le type du $\wedge$ du CCI.
- Si la dernière règle est *or-type*, P' a la forme $A' \vee B'$, $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. Par hypothèse de récurrence, il existe A et B tels que $E; G \vdash A \Rightarrow_{Prop} A'$, $E; G \vdash B \Rightarrow_{Prop} B'$, $E; G \vdash_{CCI} A : Prop$ et $E; G \vdash_{CCI} B : Prop$. Donc si P est de la forme $A \vee B$, $E; G \vdash P \Rightarrow_{Prop} P'$ par *or-trad* et $E; G \vdash_{CCI} P : Prop$ d'après le type du $\vee$ du CCI.
- Si la dernière règle est *imp-type*, P' a la forme $A' \rightarrow B'$, $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. Par hypothèse de récurrence, il existe A et B tels que $E; G \vdash A \Rightarrow_{Prop} A'$, $E; G \vdash B \Rightarrow_{Prop} B'$, $E; G \vdash_{CCI} A : Prop$ et $E; G \vdash_{CCI} B : Prop$. Donc si P est de la forme $A \rightarrow B$, $E; G \vdash P \Rightarrow_{Prop} P'$ par *imp-trad* et $E; G \vdash_{CCI} P : Prop$ d'après le type du $\rightarrow$ du CCI.
- Si la dernière règle est *not-type*, P' a la forme $\neg Q'$ et $\Gamma \vdash_{FOL} Q'\ bf$. Par hypothèse de récurrence, il existe Q tel que $E; G \vdash Q \Rightarrow_{Prop} Q'$ et $E; G \vdash_{CCI} Q : Prop$. Donc si P est de la forme $\neg Q$, $E; G \vdash P \Rightarrow_{Prop} P'$ par *not-trad* et $E; G \vdash_{CCI} P : Prop$ d'après le type du $\neg$ du CCI.

– Si la dernière règle est *forall-type*, P' a la forme $\forall x : s, Q'$ et $\{x : s\} \cup \Gamma \vdash_{FOL} Q'$ bf. On sait d'après le **Lemme 13** qu'il existe T tel que $E; G \vdash T \Rightarrow_{Sort} s$. Donc $E; (x : T) :: G \Rightarrow_{Env} S; \{x : s\} \cup \Gamma; \Delta$ par *vc-trad*. Alors par hypothèse de récurrence, il existe Q tel que $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$ et $E; (x : T) :: G \vdash_{CCI} Q' : Prop$. Si P a la forme $\forall x : T, Q$, non seulement d'après *Prod*, $E; G \vdash_{CCI} P : Prop$ mais en plus $E; G \vdash P \Rightarrow_{Prop} P'$.
– Si la dernière règle est *exists-type*, P' a la forme $\exists x : s, Q'$ et $\{x : s\} \cup \Gamma \vdash_{FOL} Q'$ bf. On sait d'après le **Lemme 13** qu'il existe T tel que $E; G \vdash T \Rightarrow_{Sort} s$. Donc $E; (x : T) :: G \Rightarrow_{Env} S; \{x : s\} \cup \Gamma; \Delta$ par *vc-trad*. Alors par hypothèse de récurrence, il existe Q tel que $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$ et $E; (x : T) :: G \vdash_{CCI} Q' : Prop$. Si P a la forme $\exists x : T, Q$, non seulement d'après *Prod*, $E; G \vdash_{CCI} P : Prop$ mais en plus $E; G \vdash P \Rightarrow_{Prop} P'$.

Théorème 3 : Soient P une proposition du CCI, P' une proposition du premier ordre, E un environnement global, G un contexte local, S un ensemble de sortes, Γ un environnement et Δ un contexte. Si $E; G \Rightarrow_{Env} S; \Gamma; \Delta$, $E; G \vdash P \Rightarrow_{Prop} P'$ et $\Gamma; \Delta \models P'$ alors il existe un terme h du CCI tel que $E; G \vdash h : P$.

Preuve : par récurrence sur $\Gamma; \Delta \models P'$.
– Si la dernière règle est *assumption*, c'est que $P' \in \Delta$. Nous devons alors trouver un terme de preuve de chacune des propositions que l'on inclut dans Δ à travers la traduction $E; G \Rightarrow_{Env} S; \Gamma; \Delta$.
  – Dans le cas de *fda-trad*, P est alors de la forme $\forall_{k=1}^{n} x_k : T_k, c\ x_1\ ...\ x_n = u\ x_1\ ...\ x_n$ et on alors donc $E; G \vdash P \Rightarrow_{Prop} P'$ d'après les prémisses de la règle. Soit $h = \lambda_{k=1}^{n} x_k : T_k, \mathit{refl\_equal}\ T_{n+1}\ (u\ x_1\ ...\ x_n)$. Alors, $E; G \vdash_{CCI} h : \forall_{k=1}^{n} x_k : T_k, u\ x_1\ ...\ x_n = u\ x_1\ ...\ x_n$. De plus, on a $c := \lambda_{k=1}^{i} x_k : T_k, u\ x_1\ ...\ x_i$, donc par *Conv*, on obtient $E; G \vdash_{CCI} h : \forall_{k=1}^{n} x_k : T_k, c\ x_1\ ...\ x_n = u\ x_1\ ...\ x_n$, ou encore $E; G \vdash_{CCI} h : P$.
  – Dans le cas de *pda-trad*, P est alors de la forme $\forall_{k=1}^{n} x_k : T_k, c\ x_1\ ...\ x_n \Leftrightarrow u\ x_1\ ...\ x_n$ et on a donc bien $E; G \vdash P \Rightarrow_{Prop} P'$ d'après les prémisses de la règle. Soient $A = u\ x_1\ ...\ x_n$, $Id_A = \lambda x : A, x$ et $h' = \mathit{conj}\ A\ A\ Id_A\ Id_A$. On remarque tout d'abord que $E; [x_1 : T_1; ...; x_n : T_n]\ @\ G \vdash_{CCI} Id_A : A \rightarrow A$ et donc $E; [x_1 : T_1; ...; x_n : T_n]\ @\ G \vdash_{CCI} h' : A \Leftrightarrow A$. Soit $h = \lambda_{k=1}^{n} x_k : T_k, h'$. Puisque l'on a $E; G \vdash_{CCI} \lambda_{k=1}^{n} x_k : T_k, h' : \forall_{k=1}^{n} A \Leftrightarrow A$, par *Conv*, on obtient bien $E; G \vdash_{CCI} h : P$ car $c := \lambda_{k=1}^{i} x_k : T_k, u\ x_1\ ...\ x_i$.
  – Dans le cas de *aa-trad*, on a directement que $c : P \in E$. Donc il suffit de prendre $h = c$ et on a bien $E; G \vdash_{CCI} h : P$.
  – Dans le cas de *ind-trad*, il faut justifier les propositions ajoutées via la fonction *Injection*. Il faut donc montrer qu'il existe un terme de type P lorsque P est de la forme $\forall_{i=1}^{n} x_i : T_i, \forall_{i=1}^{n} y_i : T_i, c\ x_1\ ...\ x_n = c\ y_1\ ...\ y_n \rightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n$. On sait que T_{n+1} est un inductif dans *Set* qui a pour constructeur, entre autres, c. On définit une famille F_i $(1 \leq i \leq n)$ de fonctions ainsi : $F_i := \lambda e : T_{n+1}, \mathit{case}(e, T_i, c\ e_1\ ...\ e_n \Rightarrow e_i \mid \_ \Rightarrow x_i)$ où $\_ \Rightarrow x_i$ signifie que la fonction F_i retourne x_i dans le cas de tous les autres constructeurs qui n'apparaissent pas dans le corps du *case*. On a donc $F_i\ (c\ x_1\ ...\ x_n) \rightarrow_{conv} x_i$ (cas de ι-réduction) et de même $F_i\ (c\ y_1\ ...\ y_n) \rightarrow_{conv} y_i$. Donc par *Conv*, il existe n termes H_i tels que $H_i : F_i\ (c\ x_1\ ...\ x_n) = x_i$ et n termes H'_i tels que $H'_i : F_i\ (c\ y_1\ ...\ y_n) = y_i$. De plus, en posant

$eq\_sym := \lambda A : TypeSet, \lambda x : A, \lambda y : A, \lambda h : x = y, eq\_ind\ A\ x\ (\lambda z : A, z = x)$ $(refl\_equal\ x)\ y$, on obtient $eq\_sym : \forall A : TypeSet, \forall x : A, \forall y : A, x = y \to y = x$. Ainsi, on peut déduire une famille de termes h_i définis par $h_i := eq\_sym\ T_i$ $(F_i\ (c\ x_1\ ...\ x_n))\ x_i\ H_i$ tels que $h_i : x_i = F_i\ (c\ x_1\ ...\ x_n)$ et une famille de termes h'_i définis par $h'_i := eq\_sym\ T_i\ (F_i\ (c\ y_1\ ...\ y_n))\ y_i\ H'_i$ tels que $h'_i : y_i = F_i\ (c\ y_1\ ...\ y_n)$. Alors si on prend $P_i := eq\_ind\ T_{n+1}\ (c\ x_1\ ...\ x_n)\ (\lambda z : T_{n+1}, x_i = F_i\ z)\ h_i\ (c\ y_1\ ...\ y_n)\ h_0$ où $h_0 : c\ x_1\ ...\ x_n = c\ y_1\ ...\ y_n$, $P_i : x_i = F_i\ (c\ y_1\ ...\ y_n)$. Puisque $F_i\ (c\ y_1\ ...\ y_n) =_{conv} y_i$, on a bien par *Conv* $P_i : x_i = y_i$. Finalement, le terme $h := \lambda_{i=1}^{n} x_i : T_i, \lambda_{i=1}^{n} y_i = T_i, \lambda h_0 : c\ x_1\ ...\ x_n = c\ y_1\ ...\ y_n, conj\ (x_1 = y_1)$ $(x_2 = y_2 \wedge ... \wedge x_n = y_n)\ P_1\ (conj\ ...\ (conj\ (x_{n-1} = y_{n-1})\ (x_n = y_n)\ P_{n-1}\ P_n)\ ...)$ a pour type $\forall_{i=1}^{n} x_i : T_i, \forall_{i=1}^{n} y_i : T_i, c\ x_1\ ...\ x_n = c\ y_1\ ...\ y_n \to x_1 = y_1 \wedge ... \wedge x_n = y_n$.

 – Le dernier cas est celui de *fix-trad* pour lequel il faut trouver un terme de preuve de $\forall_{k=1}^{w-1} x_k : T_k, \forall_{k=w+1}^{n}, x_k : T_k, \forall_{k=1}^{z_w} y_k : C_k, f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n = f_w$ pour $0 \leq w \leq l$. Par *ι-rF*, on sait que $f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n \to_{conv} case(c_w\ y_1\ ...\ y_{z_w}, T_{n+1}, F)$ avec F de la forme $c_1\ y_1\ ...\ y_{z_1} \Rightarrow f_1 \mid ... \mid c_l\ y_1\ ...\ y_{z_l} \Rightarrow f_l$. Donc par *ι-rc*, on obtient directement que $f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n \to_{conv} f_w$. Soit h un terme de la forme $\lambda_{k=1}^{w-1} x_k : T_k, \lambda_{k=w+1}^{n}, x_k : T_k, refl\_equal\ T_{n+1}\ f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n$. h est donc de type $\forall_{k=1}^{w-1} x_k : T_k, \forall_{k=w+1}^{n}, x_k : T_k, \forall_{k=1}^{z_w} y_k : C_k, f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n = f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n$. Par *Conv*, on en déduit donc que $h : \forall_{k=1}^{w-1} x_k : T_k, \forall_{k=w+1}^{n}, x_k : T_k, \forall_{k=1}^{z_w} y_k : C_k, f\ x_1\ ...\ x_{w-1}\ (c_w\ y_1\ ...\ y_{z_w})\ x_{w+1}\ ...\ x_n = f_w$.

– Si la dernière règle est *and-intro*, P' est de la forme $A' \wedge B'$, $\Gamma; \Delta \models A'$ et $\Gamma; \Delta \models B'$. De plus P est de la forme $A \wedge B$ avec $E; G \vdash A \Rightarrow_{Prop} A'$ et $E; G \vdash B \Rightarrow_{Prop} B'$ (c'est la seule façon d'obtenir une traduction en $A' \wedge B'$). Donc par hypothèse de récurrence, il existe deux termes h_1 et h_2 tels que $E; G \vdash_{CCI} h_1 : A$ et $E; G \vdash_{CCI} h_2 : B$. Soit $h = and\_intro\ A\ B\ h_1\ h_2$. On a alors $E; G \vdash_{CCI} and\_intro\ A\ B\ h_1\ h_2 : A \wedge B$ et donc $E; G \vdash_{CCI} h : P$.
– Si la dernière règle est *and-élim1*, il existe Q' tel que $\Gamma; \Delta \models P' \wedge Q'$. On sait donc d'après le **Théorème 1** que $\Gamma \vdash_{FOL} P' \wedge Q'\ bf$. C'est donc que $\Gamma \vdash_{FOL} Q'\ bf$ (par la seule règle possible : *and-type*). Alors, par le **Lemme 15**, il existe Q tel que $E; G \vdash Q \Rightarrow_{Prop} Q'$. Par hypothèse de récurrence, on peut en déduire qu'il existe un terme h_2 tel que $E; G \vdash_{CCI} h_2 : P \wedge Q$. De plus, soit $h_1 = \lambda p : P, \lambda q : Q, p$. On a $E; G \vdash_{CCI} h_1 : P \to Q \to P$. Enfin, soit $h = and\_ind\ P\ Q\ P\ h_1\ h_2$. On a alors $E; G \vdash_{CCI} h : P$.
– Si la dernière règle est *and-élim2*, il existe Q' tel que $\Gamma; \Delta \models Q' \wedge P'$. On sait donc d'après le **Théorème 1** que $\Gamma \vdash_{FOL} Q' \wedge P'\ bf$. C'est donc que $\Gamma \vdash_{FOL} Q'\ bf$ (par la seule règle possible : *and-type*). Alors, par le **Lemme 15**, il existe Q tel que $E; G \vdash Q \Rightarrow_{Prop} Q'$. Par hypothèse de récurrence, on peut en déduire qu'il existe un terme h_2 tel que $E; G \vdash_{CCI} h_2 : Q \wedge P$. De plus, soit $h_1 = \lambda q : Q, \lambda p : P, P$. On a $E; G \vdash_{CCI} h_1 : Q \to P \to P$. Enfin, soit $h = and\_ind\ Q\ P\ P\ h_1\ h_2$. On a alors $E; G \vdash_{CCI} h : P$.
– Si la dernière règle est *or-intro1*, P' est de la forme $A' \vee B'$, $\Gamma; \Delta \models A'$ et $\Gamma \vdash_{FOL} B'\ bf$. De ce dernier fait et par le **Lemme 15**, on sait qu'il existe B tel que $E; G \vdash B \Rightarrow_{Prop} B'$. P est alors de la forme $A \wedge B$. De plus, par hypothèse de récurrence, il existe a tel que $E; G \vdash_{CCI} a : A$. Soit $h = or\_introl\ A\ B\ a$. On a

alors $E; G \vdash_{CCI} or\_introl\ A\ B\ a : A \vee B$ et donc $E; G \vdash_{CCI} h : P$.

- Si la dernière règle est *or-intro2*, P' est de la forme $A' \vee B'$, $\Gamma; \Delta \models B'$ et $\Gamma \vdash_{FOL} A'$ bf. De ce dernier fait et par le **Lemme 15**, on sait qu'il existe A tel que $E; G \vdash A \Rightarrow_{Prop} A'$. P est alors de la forme $A \wedge B$. De plus, par hypothèse de récurrence, il existe b tel que $E; G \vdash_{CCI} b : B$. Soit $h = or\_intror\ A\ B\ b$. On a alors $E; G \vdash_{CCI} or\_introl\ A\ B\ b : A \vee B$ et donc $E; G \vdash_{CCI} h : P$.
- Si la dernière règle est *or-élim*, il existe A' et B' tels que $\Gamma; \Delta \models A' \vee B'$, $\Gamma; \Delta \models A' \rightarrow P'$ et $\Gamma; \Delta \models B' \rightarrow P'$. Par le **Théorème 1**, on sait tout d'abord que $\Gamma \vdash_{FOL} A' \vee B'$ bf. Par la seule règle applicable, le *or-type*, on en déduit que $\Gamma \vdash_{FOL} A'\ bf$ et $\Gamma \vdash_{FOL} B'\ bf$. Par le **Lemme 15**, il existe donc A et B tels que $E; G \vdash A \Rightarrow_{Prop} A'$ et $E; G \vdash B \Rightarrow_{Prop} B'$, et donc $E; G \vdash A \vee B \Rightarrow_{Prop} A' \vee B'$ par *or-trad*. Par hypothèse de récurrence, on obtient qu'il existe h_3 tel que $E; G \vdash_{CCI} h_3 : A \vee B$, mais aussi qu'il existe h_1 et h_2 tels que $E; G \vdash_{CCI} h_1 : A \rightarrow P$ et $E; G \vdash_{CCI} h_2 : B \rightarrow P$. Soit $h = or\_ind\ A\ B\ P\ h_1\ h_2\ h_3$. On a alors $E; G \vdash_{CCI} or\_ind\ A\ B\ P\ h_1\ h_2\ h_3 : P$ et donc $E; G \vdash_{CCI} h : P$.
- Si la dernière règle est *imp-intro*, P' est de la forme $A' \rightarrow B'$. On a $\Gamma; \Delta \cup \{A'\} \models B'$ et $\Gamma \vdash_{FOL} A'\ bf$. La seule façon de traduire une implication est d'utiliser la règle *imp-trad* si bien qu'il existe A et B tels que P est de la forme $A \rightarrow B$, $E; G \vdash A \rightarrow B \Rightarrow_{Prop} A' \rightarrow B'$ et $E; G \vdash A \Rightarrow_{Prop} A'$ et $E; G \vdash B \Rightarrow B'$. Remarquons que si $E; G \vdash Q \Rightarrow_{Prop} Q'$, alors $E; G \vdash_{CCI} Q : Prop$. En effet, les règles terminales traitent d'objets de type *Prop* et les connecteurs retournent tous des objets de type *Prop*. Donc $E; G \vdash_{CCI} A \rightarrow B : Prop$. Puisque $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et $E; G \vdash A \Rightarrow_{Prop} A'$, alors par *aa-trad*, $(a : A) :: E; G \Rightarrow_{Env} S; \Gamma; \Delta \cup \{A'\}$. Et comme on a $\Gamma; \Delta \cup \{A'\} \models B'$ alors, par hypothèse de récurrence, il existe b tel que $(a : A) :: E; G \vdash_{CCi} b : B$. Soit $h = \lambda a : A, b$. Par la règle *Lam* on a $E; G \vdash_{CCI} \lambda a : A, b : A \rightarrow B$ (cas du produit non-dépendant). Donc on a bien $E; G \vdash h : P$.
- Si la dernière règle est *imp-élim*, il existe Q' tel que $\Gamma; \Delta \models Q' \rightarrow P'$. Donc par le **Théorème 1**, $\Gamma \vdash_{FOL} Q' \rightarrow P'\ bf$. Ceci ne peut être possible que par la règle *imp-type* si bien que $\Gamma \vdash_{FOL} Q'\ bf$ et $\Gamma \vdash_{FOL} P'\ bf$. Donc par le **Lemme 15**, il existe Q et P tels que $E; G \vdash Q \Rightarrow_{Prop} Q'$ et $E; G \vdash P \Rightarrow_{Prop} P'$ et donc $E; G \vdash Q \rightarrow P \Rightarrow_{Prop} Q' \rightarrow P'$. Donc par hypothèse de récurrence, il existe f et q tels que $E; G \vdash_{CCI} f : Q \rightarrow P$ et $E; G \vdash_{CCI} q : Q$. Soit $h = f\ q$, on a alors $E; G \vdash_{CCI} f\ q : P$ par *App* et donc $E; G \vdash_{CCI} h : P$.
- Puisque $\neg P'$ n'est qu'un alias de $P' \rightarrow \bot$, la preuve quant à ce connecteur se déduit des règles du $\rightarrow$ et du $\bot$.
- Si la dernière règle est *top-intro*, P' est donc de la forme $\top$ et P est également de la forme $\top$ car c'est là la seule façon d'obtenir $E; G \vdash P \Rightarrow_{Prop} P'$ (c'est la règle *top-trad*). Alors soit $h = I$, on a bien $E; G \vdash_{CCI} I : \top$ et donc $E; G \vdash_{CCI} h : P$.
- Si la dernière règle est *bot-élim*, on a $\Gamma; \Delta \models \bot$ et $\Gamma \vdash_{FOL} P'\ bf$. Par le **Lemme 15**, il existe donc P tel que $E; G \vdash P \Rightarrow_{Prop} P'$ et $E; G \vdash_{CCI} P : Prop$. De plus, par hypothèse de récurrence, il existe ff tel que $E; G \vdash_{CCI} ff : \bot$. Soit $h = bot\_ind\ P\ ff$. On a $E; G \vdash_{CCI} bot\_ind\ P\ ff : P$ et donc $E; G \vdash_{CCI} h : P$.
- Si la dernière règle est *forall-intro*, P' est de la forme $\forall x : s, Q'$, $x \notin \Gamma$, x n'apparaît pas dans Δ et $\{x : s\} \cup \Gamma; \Delta \models Q'$. Puisque $E; G \vdash P \Rightarrow_{Prop} P'$, c'est que P est de la forme $\forall x : T, Q$ avec $E; G \vdash T \Rightarrow_{Sort} s$ et $E; G \vdash Q \Rightarrow_{Prop} Q'$. Puisque

$E; G \Rightarrow_{Env} S; \Gamma; \Delta$, par *vc-trad*, $E; (x : T) :: G \Rightarrow_{Env} S; \{x : s\} \cup \Gamma; \Delta$. De plus, puisque $\{x : s\} \cup \Gamma; \Delta \models Q'$, alors par le **Théorème 1** $\{x : s\} \cup \Gamma \vdash_{FOL} Q'\ bf$. Donc par hypothèse de récurrence, il existe t tel que $E; (x : T) :: G \vdash_{CCI} t : Q$. Comme expliqué précédemment dans la règle *imp-intro*, $E; G \vdash_{CCI} P : Prop$. Soit $h = \lambda x : T, t$. Par *Lam*, $E; G \vdash_{CCI} \lambda x : T, t : \forall x : T, Q$ et donc $E; G \vdash_{CCI} h : P$.

– Si la dernière règle est *forall-élim*, P' est de la forme $Q'[t'/x]$, $\Gamma; \Delta \models \forall x : s, Q'$ et $\Gamma \vdash_{FOL} t' : s$. Puisque s est une sorte, $s \in S$. Donc d'après le **Lemme 13**, il existe T tel que $E; G \vdash T \Rightarrow_{Sort} s$. Par le **Lemme 14**, il existe t tel que $E; G \vdash t \Rightarrow_{Term} t'$ et $E; G \vdash_{CCI} t : T$. De plus, $\Gamma; \Delta \models \forall x : s, Q'$, donc par le **Théorème 1**, $\Gamma; \Delta \vdash_{FOL} \forall x : s, Q'\ bf$ ce qui n'est possible que par *forall-type* et donc $\{x : s\} \cup \Gamma \vdash_{FOL} Q'\ bf$. Donc par le **Lemme 15**, il existe Q tel que $E; (x : T) :: G \vdash Q \Rightarrow_{Prop} Q'$. Donc si P est de la forme $Q\{t/x\}$, on a bien $E; G \vdash P \Rightarrow_{Prop} P'$. Puisqu'on a $E; G \vdash \forall x : T, Q \Rightarrow_{Prop} \forall x : s, Q'$ par *forall-trad*, alors par hypothèse de récurrence il existe f tel que $E; G \vdash_{CCI} f : \forall x : T, Q$. Soit $h = f\ t$. Par *App*, $E; G \vdash_{CCI} f\ t : Q\{t/x\}$ ou encore $E; G \vdash_{CCI} h : P$.
– Si la dernière règle est *exists-intro*, P' est donc de la forme $\exists x : s, Q'$, $x \notin \Gamma$, x n'apparaît pas dans Δ, $\Gamma; \Delta \models Q'[t'/x]$ et $\Gamma \vdash_{FOL} t' : s$. Puisque $E; G \vdash P \Rightarrow_{Prop} P'$, c'est que P est de la forme $\exists x : T, Q$ avec $E; G \vdash T \Rightarrow_{Sort} s$ et $E; G \vdash Q \Rightarrow_{Prop} Q'$ par la seule règle applicable : *exists-trad*. Notons que $\exists x : T, Q$ est une façon équivalente d'écrire $ex\ (\lambda x : T, Q)$. Tout d'abord, puisque $\Gamma \vdash_{FOL} t' : s$, alors par le **Lemme 14**, il existe t tel que $E; G \vdash t \Rightarrow_{Term} t'$ et $E; G \vdash_{CCI} t : T$. Alors, $E; G \vdash Q\{t/x\} \Rightarrow_{Prop} Q'[t'/x]$. Par hypothèse de récurrence, on obtient donc qu'il existe w tel que $E; G \vdash_{CCI} w : Q\{t/x\}$, c'est-à-dire $E; G \vdash_{CCI} w : (\lambda x : T, Q)\ t$ (règle de conversion concernant la β-réduction). Soit $h = ex\_intro\ T\ P\ t\ w$. On a $E; G \vdash_{CCI} ex\_intro\ T\ P\ t\ w : ex\ (\lambda x : T, Q)$ et donc $E; G \vdash_{CCI} h : P$.
– Si la dernière règle est *exists-élim*, il existe Q' tel que $\Gamma; \Delta \models \exists x : s, Q'$, $\{x : s\} \cup \Gamma; \Delta \cup \{Q'\} \models P'$ et x n'apparaît pas dans $\Delta \cup \{P'\}$. Puisque $\Gamma; \Delta \models \exists x : s, Q'$, alors par le **Théorème 1**, $\Gamma \vdash \exists x : s, Q'\ bf$. Par le **Lemme 15**, il existe donc A tel que $E; G \vdash A \Rightarrow_{Prop} \exists x : s, Q'$. Ceci ne peut être vrai que par *exists-trad* si bien que A a la forme $\exists x : T, Q$ avec $E; G \Rightarrow_{Sort} s$ et $E; G \vdash Q \Rightarrow_{Prop} Q'$. Par hypothèse de récurrence, il existe h_2 tel que $E; G \vdash_{CCI} h_2 : \exists x : T, Q$, ou encore $E; G \vdash_{CCI} h_2 : ex\ (\lambda x : T, Q)$. De plus, puisque $E; G \Rightarrow_{Env} S; \Gamma; \Delta$ et que $E; G \vdash T \Rightarrow_{Sort} s$, alors par *fa-trad* et *aa-trad*, $(q : Q) :: (x : T) :: E; G \Rightarrow_{Env} \{x : s\} \cup \Gamma; \Delta \cup \{Q'\}$. Par hypothèse de récurrence, on obtient donc qu'il existe h'_1 tel que $(q : Q) :: (x : T) :: E; G \vdash_{CCI} h'_1 : P$. Par deux utilisations successives de la règle *Lam*, on en déduit alors $E; G \vdash_{CCI} \lambda x : T, \lambda q : Q, h'_1 : \forall x : T, Q \rightarrow P$ (P et Q sont non-dépendants du fait que P' et Q' ne peuvent pas l'être). Avec $h_1 = \lambda x : T, \lambda q : Q, h'_1$, on a $E; G \vdash_{CCI} h_1 : \forall x : T, Q \rightarrow P$. Soit $h = ex\_ind\ T\ (\lambda x : T, Q)\ P\ h_1\ h_2$. On a $E; G \vdash_{CCI} ex\_ind\ T\ (\lambda x : T, Q)\ P\ h_1\ h_2 : P$ et donc $E; G \vdash_{CCI} h : P$.

Chapitre 3

Implantation

La traduction présentée dans le chapitre précédent a été codée en Ocaml [8] dans la version 8 de Coq. Nous ne détaillons pas le code de cette implantation. Ce chapitre est un guide d'utilisateur afin de présenter les résultats pratiques de notre projet.

3.1 Le système Coq

Le prouveur interactif que nous utilisons et qui s'appuie sur le CCI s'appelle Coq. Il exploite l'isomorphisme de Curry-Howard pour aider l'utilisateur à construire des preuves. Les règles d'inférence sont les règles de typage du CCI et peuvent être invoquées au travers de *tactiques* : ce sont des commandes qui représentent l'utilisation des règles. Par exemple, la tactique *intro* représente les règles *Prod-Prop* et *Prod-Set*. Le système peut alors déterminer ce qu'il reste à prouver, avec une certaine dose d'automatisme (par exemple, dans ce cas, le fait que ce soit *Prod-Prop* ou *Prod-Set* qu'il faille appliquer).

Pour commencer une preuve en Coq, une solution possible est d'utiliser la commande *Goal*. On doit donner explicitement le terme dont il faudra trouver une preuve.

Exemple : prouver $\forall A : \mathit{Prop}, A \rightarrow A$ (c'est-à-dire déterminer un terme dont le type est $\forall A : \mathit{Prop}, A \rightarrow A$).

```
Coq < Goal forall A : Prop, A -> A.
1 subgoal

  ============================
   forall A : Prop, A -> A

Unnamed_thm < intros.
1 subgoal

  A : Prop
  H : A
  ============================
   A
```

```
Unnamed_thm < exact H.
Proof completed.
```

La commande *exact* permet de donner explicitement le terme en question, celui dont le type est A.

Nous conseillons le lecteur intéressé par la syntaxe et l'utilisation du système Coq de consulter la documentation en ligne disponible sur [7] et de lire [12].

3.2 Retour des prouveurs

L'appel à une procédure de décision depuis l'environnement Coq se fait par une tactique qui reprend le nom de la procédure en question. Une fois l'appel effectué, la traduction commence, le prouveur automatique tente de résoudre le but, puis renvoie sa réponse. Différents cas se présentent :

– La traduction échoue car le but n'est pas une proposition :
  ```
  Coq < Goal nat -> nat.
  1 subgoal

    ============================
     nat -> nat

  Unnamed_thm < simplify.
  Toplevel input, characters 73-81
  > simplify.
  > ^^^^^^^^
  User error: Conclusion is not a Prop
  ```
– La traduction échoue car le but n'est pas de premier ordre :
  ```
  Coq < Goal exists f : nat -> nat, forall x : nat, f x = x.
  1 subgoal

    ============================
     exists f : nat -> nat, (forall x : nat, f x = x)

  Unnamed_thm < simplify.
  Toplevel input, characters 201-209
  > simplify.
  > ^^^^^^^^
  User error: Not a first order goal
  ```
– La traduction réussie mais le prouveur échoue dans le temps qui lui était imparti (10 secondes par défaut) :
  ```
  Coq < Goal forall n : nat, add n 0 = n.
  1 subgoal

    ============================
     forall n : nat, add n 0 = n
  ```

```
Unnamed_thm < zenon.
timeout 10 Simplify /tmp/coq_dp830544.sx > out 2>&1 && grep -q -w Valid out
Toplevel input, characters 30-38
> zenon.
> ^^^^^
User error: Timeout
```

– La traduction réussie mais le prouveur a déterminé que le but était faux :

```
Coq < Goal 0 = 1.
1 subgoal

  ============================
   0 = 1

Unnamed_thm < cvcl.
timeout 10 cvcl < /tmp/coq_dp0675c8.cvc > out 2>&1 && grep -q -w Valid out
Toplevel input, characters 59-63
> cvcl.
> ^^^^
User error: Invalid
```

À noter que chez la plupart des prouveurs, la réponse *Invalid* ne signifie pas nécessairement que le but est faux. En réalité, on ne peut rien déduire quant à un séquent qui est dit *Invalid* par beaucoup de procédures de décision.

– La traduction réussie et le prouveur est parvenu à prouver le séquent :

```
Coq < Goal forall z : Z, z = z.
1 subgoal

  ============================
   forall z : Z, z = z

Unnamed_thm < simplify.
timeout 10 Simplify /tmp/coq_dp4daf1f.sx > out 2>&1 && grep -q -w Valid out
Proof completed.
```

3.3 Spécificités

À présent, nous allons montrer quelques spécificités de nos appels aux procédures de décision.

Tout d'abord, la traduction est présentée dans la syntaxe du prouveur appelé, juste après l'appel.

```
Coq < Goal forall f : nat -> nat, forall n : nat, f n = n.
1 subgoal

  ============================
```

```
   forall (f : nat -> nat) (n : nat), f n = n

Unnamed_thm < cvcl.
nat: TYPE;
O: nat;
S: [nat -> nat];
ASSERT % injection_S
 (FORALL (x1:nat): (FORALL (y1:nat): ((S(x1) = S(y1))
                                      => ((x1 = y1) AND TRUE))));
f: [nat -> nat];
n: nat;
QUERY (f(n) = n);
```

Ceci permet de vérifier exactement ce qui a été envoyé à la procédure. De plus, les prouveurs ont tous des fonctions prédéfinies. Que ce soit l'égalité pour tous, ou les entiers relatifs et les fonctions arithmétiques pour la plupart, ces objets sont primitifs dans leur système. Nous ne devons pas traduire les définition Coq de ces objets. En effet, les procédures ont des heuristiques particulières pour ceux-ci. Ainsi le + Coq sur les relatifs doit être traduit vers le + de CVC Lite par exemple.

Remarque : on peut être surpris que ce but ait été traduit. Effectivement, on trouve dans la proposition à prouver une quantification sur une fonction. Le fait est qu'en logique du premier ordre une telle quantification est impossible ; en revanche, les déclarations de fonctions le sont (lorsque celles-ci sont de premier ordre). L'astuce consiste à effectuer autant d'introduction de $\forall$ que possible avant traduction afin de faire passer les quantifications sur des objets d'ordre supérieur en déclaration. Bien évidemment cette astuce ne marche que si les objets ainsi passer en déclaration sont de premier ordre.

Il est important de comprendre la démarche de notre travail. Avant le branchement des procédures de décision à Coq, nous devions traduire les buts Coq à la main et tester leur validité par un ou des prouveurs. Si un prouveur déclare valide le but courant, on admet le résultat et on passe au reste de la preuve. En ceci, notre démarche reste la même. C'est-à-dire qu'après appel à un prouveur, si la réponse de ce dernier est positive, il faut faire comprendre à l'environnement Coq qu'on admet le résultat, alors qu'on ne dispose en réalité d'aucune preuve formelle. C'est pourquoi une tactique a été écrite afin de poser pour axiome le but courant lorsque qu'un prouveur automatique l'a validé. Cette tactique s'appelle *admit* et peut-être utilisée en dehors du simple cadre de notre projet.

```
Coq < Lemma admit_ex :
Coq <   forall P : Prop, P -> False.
1 subgoal

  ============================
   forall P : Prop, P -> False

admit_ex < admit.
Proof completed.
```

```
Coq < Check admit_ex_admitted.
admit_ex_admitted
     : forall P : Prop, P -> False
```

Ici, l'axiome créé spécialement pour la preuve s'appelle *admit_ex_admitted*. On comprend avec cet exemple les dangers de la tactique.

En théorie, il est possible de traduire la totalité de l'environnement global. En pratique, ceci n'est pas envisageable car celui-ci est trop chargé si bien que, après traduction, les prouveurs risquent de se perdre dans le dédale de possibilités qui leur sont proposées pour prouver un certain séquent. Donc ce qui est traduit est en priorité ce dont on a besoin pour que la traduction soit bien typée. Ensuite viennent s'ajouter quelques axiomes comme lors de la définition de fonction ou comme l'axiome d'injection pour les constructeurs d'un inductif dans *Set*.

```
Coq < Definition f (z : Z) := 2 * z.
f is defined

Coq < Definition g (z : Z) := z.
g is defined

Coq < Goal exists z, f z = 0.
1 subgoal

  ============================
   exists z : Z, f z = 0

Unnamed_thm < simplify.
;; f : INT -> INT
(BG_PUSH ;; f
 (FORALL (z0) (EQ (f z0) (* (* 2 1) z0))))
(EXISTS (z) (EQ (f z) 0))

Proof completed.
```

Cependant il arrive que des lemmes ou des théorèmes soient définis, et que ceux-ci ne sont pas indispensables à la traduction d'un séquent s'il ne sont pas explicitement utilisés dans le but. Mais la preuve du séquent, elle, peut devoir les utiliser. Ainsi, nous avons introduit une nouvelle commande, *Dp_int*, qui prend en argument des noms de théorèmes et qui les ajoute à la traduction. Les expressions qui ne sont pas des propositions et les propositions non premier ordre sont ignorées.

Exemple où la commande *Dp_hint* n'est pas utilisée :

```
Coq < Parameter add : nat -> nat -> nat.

Coq < Axiom add_0 : forall n : nat, add 0 n = n.
```

```
Coq < Axiom add_S : forall (n1 n2 : nat), add (S n1) n2 = S (add n1 n2).

Coq < Goal forall n : nat, add n 0 = n.
1 subgoal

  ============================
   forall n : nat, add n 0 = n

Unnamed_thm < induction n ; zenon.
;; nat: TYPE
;; %sort_nat: nat -> BOOLEAN
;; O: nat
"sort_ax_1" (%sort_nat O)
;; S: nat -> nat
"sort_ax_2" (A. ((x1 "nat") (=> (%sort_nat x1) (%sort_nat (S x1)))))
"injection_S" (A. ((x1 "nat")
              (=> (%sort_nat x1)
              (A. ((y1 "nat")
              (=> (%sort_nat y1) (=> (= (S x1) (S y1)) (/\ (= x1 y1) True))))))))
;; add: nat -> nat -> nat
"sort_ax_3" (A. ((x1 "nat")
            (A. ((x2 "nat")
            (=> (%sort_nat x1) (=> (%sort_nat x2) (%sort_nat (add x1 x2))))))))
$goal (= (add O O) O)

timeout 10 zenon /tmp/coq_dp4debb7.znn > out 2>&1 && grep -q PROOF-FOUND out
Toplevel input, characters 193-198
> induction n ; zenon.
>               ^^^^^
User error: Timeout
```

Ce qui est important de voir ici, c'est que les axiomes *add_0* et *add_S* n'ont pas été envoyés au prouveur. Nous reviendrons plus tard sur ce qui est exactement envoyé selon les prouveurs.

Et voici le même exemple lorsque la commande *Dp_hint* a été utilisée pour envoyer à la traduction les axiomes important pour prouver le but :

```
Coq < Parameter add : nat -> nat -> nat.

Coq < Axiom add_0 : forall n : nat, add 0 n = n.

Coq < Axiom add_S : forall (n1 n2 : nat), add (S n1) n2 = S (add n1 n2).

Coq < Dp_hint add_0 add_S.
```

```
Coq < Goal forall n : nat, add n 0 = n.
1 subgoal

  ============================
   forall n : nat, add n 0 = n

Unnamed_thm < induction n ; zenon.
...
"add_0" (A. ((n "nat") (=> (%sort_nat n) (= (add O n) n))))
"add_S" (A. ((n1 "nat")
        (=> (%sort_nat n1)
        (A. ((n2 "nat")
        (=> (%sort_nat n2) (= (add (S n1) n2) (S (add n1 n2)))))))))
$goal (= (add O O) O)

"IHn" (= (add n 0) n)
$goal (= (add (S n) 0) (S n))

Proof completed.
```

La traduction reste la même en tout point à ceci près que les axiomes portant les noms *add_0* et *add_S* ont été ajoutés. La commande *Dp_hint* est donc semblable à la commande *hint* déjà existante pour la tactique *auto*.

Un autre effort a été fourni concernant les définitions de fonctions obtenues par application partielle. Il faut donc η-expanser jusqu'à obtenir une application totale afin de pouvoir poser un axiome premier ordre correspondant, comme le montre l'exemple ci-dessous :

```
Coq < Definition snd_of_3 (x y z : Z) := y.

Coq < Definition f : Z -> Z -> Z := snd_of_3 0.

Coq < Goal forall (x y z z1 : Z), snd_of_3 x y z = f y z1.
1 subgoal

  ============================
   forall x y z z1 : Z, snd_of_3 x y z = f y z1

Unnamed_thm < simplify.
;; f : INT, INT -> INT
;; snd_of_3 : INT, INT, INT -> INT
(BG_PUSH ;; snd_of_3
 (FORALL (x0) (FORALL (y1) (FORALL (z2) (EQ (|snd_of_3| x0 y1 z2) y1)))))
(BG_PUSH ;; f
 (FORALL (y0) (FORALL (z0) (EQ (f y0 z0) (|snd_of_3| 0 y0 z0)))))
;; x : INT
```

```
;; y : INT
;; z : INT
;; z1 : INT
(EQ (|snd_of_3| x y z) (f y z1))

Proof completed.
```

Dans le cas des types inductifs dans *Set*, les constructeurs sont vus comme des déclarations de fonctions. Dans ce cas, comme nous l'avons vu dans la traduction, un axiome d'injection est ajouté pour chaque constructeur :

```
Coq < Inductive T : Set := C : Z -> Z -> T.

Coq < Goal forall x y : Z, C x y = C 1 2 -> x = 1.
1 subgoal

  ============================
   forall x y : Z, C x y = C 1 2 -> x = 1

Unnamed_thm < zenon.
;; T: TYPE
;; %sort_T: T -> BOOLEAN
;; C: INT -> INT -> T
"sort_ax_1" (A. ((x1 "INT") (A. ((x2 "INT") (%sort_T (C x1 x2))))))
"injection_C" (A. ((x1 "INT")
              (A. ((x2 "INT")
              (A. ((y1 "INT")
              (A. ((y2 "INT")
              (=> (= (C x1 x2) (C y1 y2)) (/\ (= x1 y1) (/\ (= x2 y2)
                                                              True))))))))))
;; x: INT
;; y: INT
"H" (= (C x y) (C 1 (* 2 1)))
$goal (= x 1)

Proof completed.
```

Pour les inductifs dans *Prop*, les constructeurs sont vus comme des preuves de leur type :

```
Coq < Inductive pair : nat -> Prop :=
| pair_0 : pair 0
| pair_Sn : forall n : nat, impair n -> pair (S n)
with
impair : nat -> Prop :=
| impair_Sn : forall n : nat, pair n -> impair (S n).
```

```
Coq < Goal impair (S 0).

1 subgoal

  ============================
   impair (S 0)

Unnamed_thm < cvcl.
nat: TYPE;
O: nat;
S: [nat -> nat];
ASSERT % injection_S
 (FORALL (x1:nat): (FORALL (y1:nat): ((S(x1) = S(y1))
                                     => ((x1 = y1) AND TRUE))));
impair: [nat -> BOOLEAN];
pair: [nat -> BOOLEAN];
ASSERT % pair_0
 pair(O);
ASSERT % pair_Sn
 (FORALL (n0:nat): (impair(n0) => pair(S(n0))));
ASSERT % impair_Sn
 (FORALL (n:nat): (pair(n) => impair(S(n))));
QUERY impair(S(O));

Proof completed
```

Pour le cas des fonctions par filtrage qui définissent des termes ι-réductible, nous avons donc construit une égalité par branche du filtrage. Par exemple, la fonction *plus* sur les entiers naturels peut être définie comme suit :

```
Fixpoint plus (n m : nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end.
```

Alors une preuve de

```
Coq < forall n : nat, plus n O = n.
```

pourra se faire par récurrence sur n et engendrera les deux sous-buts suivants :

```
Unnamed_thm < induction n ; zenon.
2 subgoals


  ============================
```

```
   plus 0 0 = 0

$goal (= (plus 0 0) 0)


  n : nat
  IHn : plus n 0 = n
  ============================
   plus (S n) 0 = S n

"IHn" (= (plus n 0) n)
$goal (= (plus (S n) 0) (S n))
```

Dans les deux cas, l'environnement complet envoyé sera :

```
;; nat: TYPE
;; %sort_nat: nat -> BOOLEAN
;; 0: nat
"sort_ax_1" (%sort_nat 0)
;; S: nat -> nat
"sort_ax_2" (A. ((x1 "nat") (=> (%sort_nat x1) (%sort_nat (S x1)))))
"injection_S" (A. ((x1 "nat")
              (=> (%sort_nat x1)
              (A. ((y1 "nat")
              (=> (%sort_nat y1) (=> (= (S x1) (S y1)) (/\ (= x1 y1) True))))))))
;; plus: nat -> nat -> nat
"sort_ax_3" (A. ((x1 "nat")
            (A. ((x2 "nat")
            (=> (%sort_nat x1) (=> (%sort_nat x2) (%sort_nat (plus x1 x2))))))))
"plus_0" (A. ((m "nat") (=> (%sort_nat m) (= (plus 0 m) m))))
"plus_S" (A. ((m "nat")
         (=> (%sort_nat m)
         (A. ((p "nat")
         (=> (%sort_nat p) (= (plus (S p) m) (S (plus p m)))))))))
```

pour lequel on voit bien apparaître les deux branches de la fonction *plus* comme axiomes à la fin de l'environnement.

Cet exemple va nous permettre de présenter une autre difficulté de l'implantation : certains prouveurs sont de premier ordre mais n'utilise aucune notion de sorte ou de typage. C'est le cas de Zenon que nous avons utilisé précédemment.

Il faut tout d'abord comprendre que la traduction d'une logique de premier ordre multisortée vers une logique de premier ordre non sortée qui se ferait naïvement en oubliant les informations de type conduirait rapidement à une incohérence. Il suffit pour cela de déclarer un type inductif dans *Set* à un seul constructeur : $Ind[](Unit : Set := U : Unit)$. Par élimination forte, on prouve facilement que $\forall x : Unit, \forall y : Unit, x = y$. La traduction de cette expression vers une logique de premier ordre multisortée est syntaxiquement la même. En revanche, si l'on espère traduire naïvement cette expression vers une logique de premier ordre non sortée on obtiendrait alors $\forall x, \forall y, x = y$ qui, couplée avec $0 \neq 1$, donne une preuve de $\perp$. Cette solution n'est donc pas satisfaisante. En réalité, ce qu'il faut faire pour obtenir une traduction équivalente, c'est de créer pour chaque sorte un nouveau prédicat. Celui-ci va permettre de distinguer le type des objets. Si on prend l'exemple de *Unit*, on crée une sorte *sort_Unit* telle que *sort_Unit n* est vraie si et seulement n était de type *Unit* avant traduction. $\forall x : Unit, \forall y : Unit, x = y$ se traduit donc en $\forall x, sort\_Unit(x) \rightarrow \forall y, sort\_Unit(y) \rightarrow x = y$. La logique ainsi obtenue est strictement équivalente. Il suffit de regarder de près les exemples de ce chapitre pour apercevoir cette construction.

La dernière fonctionnalité que nous présentons est celle d' abstraction. Il s'agit d'abstraire les termes ou les prédicats qui ne sont pas premier ordre pour retomber sur un problème de premier ordre. Ainsi, $\forall x : A, F\ (\lambda y : A, y)\ x = F\ (\lambda y : A, y)\ x$ n'est pas un problème de premier ordre, mais on peut abstraire $F\ (\lambda y : A, y)$ en une fonction $F_0 : A \rightarrow A$ pour réduire le problème à $\forall x : A, F_0\ x = F_0\ x$ qui lui est premier ordre.

```
Coq < Definition App (A : Set) (B : Set) (f : A -> B) (x : A) := f x.

Goal forall (x : nat), App nat nat (fun y : nat => y) x = App nat nat
(fun y : nat => y) x.

1 subgoal

  ============================
   forall x : nat,
   App nat nat (fun y : nat => y) x = App nat nat (fun y : nat => y) x

Unnamed_thm < simplify.
;; abstraction_1 : nat -> nat
(BG_PUSH ;; sort_ax_3
 (FORALL (x1)
 (IMPLIES (EQ (|%sort_nat| x1) |@true|)
 (EQ (|%sort_nat| (|abstraction_1| x1)) |@true|))))
;; x : nat
(BG_PUSH ;; sort_ax_4
 (EQ (|%sort_nat| x) |@true|))
(EQ (|abstraction_1| x) (|abstraction_1| x))

Proof completed.
```

App n'est effectivement pas un symbole de fonction de premier ordre du fait du type de ses trois premiers arguments. L'abstraction créée et qui s'appelle *abstraction_1* correspond à *App* pour lequel on a gardé que le dernier argument qui est de premier ordre. Beaucoup d'informations ont donc été oubliées lors d'une telle abstraction, mais le but est devenu premier ordre lorsqu'on remplace toutes les occurrences de *App nat nat* (*fun y* : *nat* => *y*) par *abstraction_1*.

D'autres spécificités peuvent être ajoutées (nous en parlerons dans la conclusion) mais celles présentées ici sont les principales qui ont déjà été implantées.

Toute personne ayant un accès à la version CVS de Coq peut d'ores et déjà se servir des commandes et tactiques que nous avons écrites. Les autres utilisateurs pourront en jouir lors de la prochaine mise-à-jour de Coq.

Conclusion

Nous avons donc présenté deux logiques bien distinctes. La logique de premier ordre multisortée qu'implantent les prouveurs automatiques et le Calcul des Constructions Inductives de Coq. Une traduction assez complète a été donnée pour passer des termes complexes du CCI vers les termes simples de la logique du premier ordre. De plus, le théorème de correction nous assure que si la traduction d'un séquent du CCI est valide, alors le séquent l'est. Ceci nous permet d'utiliser les prouveurs automatiques au sein de Coq.

L'implantation dans Coq s'est faite en Ocaml et a nécessité la création de tactiques et de commandes permettant une utilisation aussi pratique que complète. Bientôt, les outils développés dans ce stage seront partie intégrante de la version distribuée de Coq.

Si le travail fournit a permis d'observer des résultats convaincants dans le gain de temps lors de la certification d'un programme, l'effort peut être prolongé.

Un point important serait la traduction de *traces de preuve*. Certains prouveurs automatiques, comme Zenon, fournissent des preuves des assertions qu'ils ont validées. En traitant ces traces, on pourrait les réinterpréter en des preuves Coq afin de ne plus être obligé de définir de nouveaux axiomes à chaque validation d'un séquent par une procédure de décision.

De plus, le panel des expressions traduites peut être plus grand encore. Entre autres, le polymorphisme peut être traduit dans une certaine mesure, même si des informations seront évidemment oubliées lors du passage au premier ordre. Ainsi, un type polymorphe comme *list* qui étant donné un ensemble retourne le type des listes des éléments de cet ensemble peut donner lieu à la création d'un nouveau type lors de son application à un ensemble particulier. La difficulté est donc de rendre le problème monomorphe lorsque cela est possible. Le fait est que ce travail a déjà été réalisé lors du développement de l'outil Why [10] ; il suffirait de se servir de Why comme intermédiaire de traduction.

Un autre point intéressant serait de nettoyer l'environnement à chaque traduction. Si la traduction prendrait plus de temps puisque le même travail pourrait alors être à refaire plusieurs fois, des informations potentiellement inutiles seraient effacées afin que les prouveurs automatiques ne s'y perdent pas. Il faut cependant faire attention à ne pas enlever ce qui a été ajouté à l'aide de *Dp_hint*. À propos de ce dernier, il serait également intéressant d'écrire une commande qui permette d'effacer de l'environnement des propositions ajoutées par le biais de cette fonctionnalité. On peut également continuer de brancher d'autres prouveurs automatiques afin de déterminer lesquels ont le plus de réussite selon les types de problèmes. Il y a donc plusieurs moyens de rendre notre outil plus performant, ce à quoi nous nous attachons dernièrement.

Bibliographie

[1] The calculus of inductive constructions (chapter 4 of the coq reference manual). http://coq.inria.fr/doc/Reference-Manual006.html.

[2] Cvc : a cooperating validity checker. http://verify.stanford.edu/CVC.

[3] Cvc lite homepage. http://verify.stanford.edu/CVCL.

[4] Isabelle. http://isabelle.in.tum.de.

[5] Jprover : an integrated first order prover to metaprl. http://www.metaprl.org.

[6] Prl automated reasoning project at cornell. http://www.cs.cornell.edu/Info/Projects/NuPrl.

[7] The Coq Proof Assistant. http://coq.inria.fr/.

[8] The Objective Caml language. http://caml.inria.fr/.

[9] The Simplify decision procedure (part of ESC/Java). http://research.compaq.com/SRC/esc/simplify/.

[10] The Why verification tool. http://why.lri.fr/.

[11] Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. In Bernhard Gramlich, editor, *5th International Workshop on Frontiers of Combining Systems, FroCoS'05, Vienna, Austria, September 19-21, 2005*, 2005.

[12] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. http://www.labri.fr/Perso/~casteran/CoqArt/index.html.

[13] Damien Doligez. Zénon (procédure de décision distribuée avec le système Focal). http://focal.inria.fr/.

[14] Gilles Dowek. Théorie des types (notes de cours de master m2). http://www.lix.polytechnique.fr/~dowek/Cours/theories_des_types.ps.gz.

[15] Tom Harke. Toward a sound integration of Isabelle with a combined decision procedure. Unpublished ; available at www.cs.pdx.edu/~harke/RPE-paper.ps.

[16] Joe Hurd. An lcf-style interface between hol and first-order logic. In *CADE-18 : Proceedings of the 18th International Conference on Automated Deduction*, pages 134–138, London, UK, 2002. Springer-Verlag.

[17] Ramayya Kumar, Thomas Kropf, and Klaus Schneider. Integrating a first-order automatic prover in the hol environment. In *TPHOLs*, pages 170–176, 1991.

[18] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers : A case study combining HOL-Light and CVC Lite. In *Proceedings of the* 3^{rd} *International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, July 2005. Edinburgh, Scotland.

[19] Silvio Ranise and David Déharbe. The haRVey decision procedure. `http://www.loria.fr/~ranise/haRVey/`.

[20] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver : Integrating connection-based theorem proving into interactive proof assistants. *Lecture Notes in Computer Science*, 2083 :421+, 2001.

[21] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives.* PhD thesis, Univ. Paris VII, 1994.