N° Ordre……/ Faculté / UMBB / 2014

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université M'hamed Bouguerra – Boumerdès

Faculté des Sciences

Mémoire de magister

Présentée et soutenue publiquement par :

BRAHIMI Farida

Filière : Système informatique et ingénierie des logiciels.

Option : Spécification de logiciels et traitement de l'information

Formalisation du système Elambda

JURY:

M^{r} : AHMED NACER Mohamed	(Pr , USTHB)	President
M^{r} : ZEGOUR Djamel Eddine	(Pr, ESI)	Examinateur
M^{r} : AIT BOUZIAD Ahmed	(MCB, UMBB)	Examinateur
M^{r} : MEZGHICHE Mohamed	(Pr, UMBB)	Rapporteur

Année universitaire : 2013/2014

À mes chers parents qui ont été à mes côtés dans tous les moments difficiles par leurs amours et leurs encouragements.

Que dieu vous protèges.

Remerciements

Il me tient à cœur de remercier toutes les personnes qui m'ont aidé dans mon travail de magister.

La première de ces personnes est bien sûr, mon promoteur Pr Mohamed Mezghiche qui m'a proposé ce sujet et qui a accepté de diriger mon mémoire. Ce travail a vu le jour grâce à ses précieux conseils et ses lectures attentives de mes rapports.

J'adresse un grand merci à Mr Mohamed Ahmed-Nacer d'avoir accepter de présider mon jury.

Je remercier Mr Ahmed Ait bouziad et Mr Djamel Eddine Zegour pour m'avoir fait l'honneur d'examiner mon travail et d'avoir eu la gentillesse de faire partie de mon jury.

Je souhaite aussi remercier Mr Mohamed Chaabani pour son aide et ses précieux conseils.

Enfin, je souhaite remercier chaleureusement mes parents, mon mari Sofiane, mes enfants Aymen et Ritadj et tout mes frères et sœurs.

Résumé

Dans ce mémoire, on a définit un nouveau système Elambda qu'est une extension de lambda calcul classique par l'ajout de deux constantes P et Π qui représentent respectivement l'implication et la quantification universelle. Le système obtenu est assez riche, dans le sens où les deux constantes introduites sont suffisantes pour exprimer et définir le reste des connecteurs et quantificateurs logiques.

La consistance du système Elambda est garantie grâce à l'affection d'un nouvel attribut appelé « niveau du terme » ; qui nous a permet d'avoir une nouvelle définition de la substitution, où le terme (M [N/x]) est définit uniquement quand le niveau du terme « N » est inférieur ou égal à celui de « x ».

Cette restriction nécessite une définition propre du mécanisme de réduction, appelé Ebeta_reduc, qui vérifié la propriété de Church-Rosser (la preuve du théorème de Church-Rosser est donnée en utilisant l'assistant de preuve Coq) et offre un moyen pour éviter le paradoxe de Curry.

Mot clés : λ-calcul, la logique d'ordre supérieur, la logique combinatoire illative, le paradoxe de Curry, la substitution, la confluence, la beta-réduction.

Abstract

In this paper we define an extension of the classical λ-calculus by adding two constants P and Π which represent respectively the implication and the universal quantifier. The obtained system Elambda is consistent, which the two constants introduced are sufficient to define other logics connectors.

The idea behind the consistency proof is to introduce a notion of level of terms that allows an appropriate definition of substitution. The term M [N/x] is defined only when level (N) ≤ level(x).

This restriction on substitution induces a new reduction (Ebeta_reduc) in Eλ which is proved to have Church-Rosser property by using the proof system Coq.

Keywords: λ-calculus, higher order logic, Illative combinatory logic, Curry's paradox, substitution, confluence, beta_reduction.

الملخص

قمنا في هذه الورقة بالتعريف عن نظام جديد Elambda الذي هو عبارة عن امتداد لنظام الحساب الكلاسيكي lambda ,و ذلك بإضافة ثابتين ***P*** و Π الذين يمثلان الاستلزام المنطقي و المكمم الكلي. النظام Elambda غني بحيث الثابتان الجديدان لهما القدرة بالتعريف و التعبير عن بقية الوصول المنطقية.

قمنا بإدخال مفهوم جديد و هو مستوى الكلمة, بحيث M[N/x] معرف فقط إذا كان مستوى "N"=> مستوى "x".

هذا الشرط أدى إلى التعريف بعلاقة جديدة تسمى Ebeta-reduc التي برهنا أنها تحقق علاقة Church-Rosser باستعمال نظام البرهنة Coq.

كلمة المفتاح: نظام الحساب الكلاسيكي lambda, المنطق الممتاز, مفارقة Curry, علاقة التحام, علاقة beta-reduction.

Table des matières

Table des Figures

Introduction

Contexte et objectifs de ce mémoire

Le Lambda-calcul est une théorie de la fonctionnalité, introduite à la fin des années 30 par le logicien Alonzo Church, qui procède d'un point de vue intentionnel. Les fonctions du λ-calcul sont des fonctions anonymes, construites par abstraction (fonctionnelle) d'une variable (λ x.) dans un terme. Ce sont donc des fonctions d'une (seule) variable. Elles peuvent être appliquées à n'importe quel terme du λ-calcul, y compris à elles-mêmes.

Le λ-calcul pur, est une théorie intéressante de plusieurs points de vue. Tout d'abord, du point de vue de la calculabilité, les termes du λ-calcul représentent exactement toutes les fonctions récursives partielles (résultat de Kleene). Du point de vue de la programmation, le λ-calcul est un outil essentiel puisqu'il s'agit d'un système formel qui décrit la construction et l'évaluation des fonctions. Il apparaît ainsi comme le prototype des langages de programmation dits fonctionnels (ou de la partie fonctionnelle des autres langages). Dans ces langages, un programme est d'abord une fonction. Il est donc très naturellement modélisé comme une expression du λ-calcul. L'expression (λx.a) représente la fonction de paramètre formel x, de corps a, le terme (λx.a) b correspond à l'appel de cette fonction sur le paramètre effectif b. Le résultat de cet appel est une valeur, celle du corps de la fonction a, dans lequel on a remplacé toutes les occurrences du paramètre formel x par la valeur effectivement fournie b. Cette évaluation est exprimée dans le λ-calcul par la règle de β-réduction :

$$(\lambda x.a)\ b \longrightarrow_{\beta} a\{b/x\}$$

a {b/x} est le résultat de la substitution de x par b dans a.

Le résultat fondamental dans ce domaine est le **théorème de confluence**, dit encore propriété de Church-Rosser, pour la β-réduction. Il signifie que l'ordre dans lequel les

différentes étapes de β-réduction sont menées, donc l'ordre dans lequel les évaluations sont effectuées, n'a pas d'importance : on parvient toujours à la même valeur.

La logique combinatoire et le lambda-calcul sont des théories analysent successivement la notion de calcul effective. Cependant les fondateurs originaux de ces deux théories Curry et Church ont toujours comme but de fournir une base pour la logique et les mathématiques.
Malheureusement, Kleen et Rosser(1935) ont montré que le lambda calcul et la logique combinatoire étendus sans restriction par les règles d'introduction et d'élimination d'implication sont inconsistants. Ce qui a conduit Curry à chercher de définir un nouveau système vérifiant la propriété de consistance. Ce dernier est appelé « Logique Combinatoire Illative (ICL). ».
La logique combinatoire illative c'est la logique combinatoire (ou lambda-calcul pur) étendue par des constantes supplémentaires et un ensemble d'axiomes et règles de dérivation.

Le principal objet de ce travail est « la formalisation du système Elambda ». Il s'agit d'une extension du lambda-calcul classique permettant d'interpréter la logique d'ordre supérieur. Le système Elambda est obtenu en étendant le lambda calcul classique par deux constantes P et Π qui représentent respectivement l'implication et la quantification universelle. Le système obtenu est assez riche dans le sens où les deux constantes introduites sont suffisantes pour définir le reste des connecteurs et quantificateurs logiques.
La consistance du système E-lambda est garantie grâce à l'affection d'un nouvel attribut, appelé « niveau » d'un terme, utilisé pour introduire une nouvelle définition du processus de substitution ; où le terme (M [N/x]) n'est définit que si le niveau du terme « N » est inférieur ou égal à celui de « x ».
Cette restriction nécessite une définition propre du mécanisme de réduction, appelé Ebeta_reduc, qu'on a prouvé qu'elle vérifié la propriété de Church-Rosser par la méthode de réduction parallèle, en utilisant l'assistant de preuve Coq.

Coq est un système permettant de faire des preuves dans une logique très expressive, dite d'ordre supérieur. Ces preuves sont construites de façon interactive, assisté par des outils de recherche automatique de preuve.

Pour arriver à notre objectif qu'est de démontrer que la relation Ebeta_reduc est confluente, on a définit les termes et les prédicats sous forme inductive, telle qu'elle est requise par le

système Coq. Puis on a prouvé tout les lemmes intermédiaires nécessaires pour la preuve du théorème de Church-Rosser pour éviter de démontrer la même propriété plusieurs fois, d'écourter la preuve et de la rendre lisible et simple à maintenir dès le changement de la spécification associée. Finalement, on a démontré la confluence de Ebeta_reduc par la méthode de William Tait et Per Martin-Löf (réduction parallèle) tel que :

[1]. On a crée une nouvelle relation nommée MCD tel que beta_red ⊂ MCD ⊂ Ebeta_reduc
[2]. On a prouvé que MCD est fortement confluente.
[3]. On a crée la fermeture réflexive-transitive de la relation MCD (mcd_star).
[4]. On a prouvé que la relation MCD est confluente
[5]. On a démontré l'égalité entre Ebeta_reduc et mcd_star. D'où on a conclut qu'Ebeta_reduc est confluente.

Pour atteindre mon objectif j'ai adapté le plan suivant :

Plan :

Chapitre 1 : Le sujet de chapitre 1 est d'introduire le lambda calcul comme étant un modèle formel de calcul, ainsi de donner un aperçu sur la logique combinatoire et la logique combinatoire illative.

Chapitre2 : Le sujet de chapitre 2 est de donner un aperçu sur la logique constructive et les concepts de base utilisés dans la mécanisation d'un processus de déduction. Nous nous intéressons particulièrement à la déduction naturelle et le système de preuve Coq. Ainsi on va donner une idée sur l'interprétation des λ-termes comme les preuves et les type assignés à ces termes comme les formules de la théorie. Cette idée est connue sous le nom « Curry-Howard isomorphism ».

Chapitre 3 : Présenter le système Elambda, en définissant les diverses notions de termes, niveau du terme, la relation de substitution et la relation de réduction Ebeta_reduc, ainsi les règles d'inférence.

Ainsi on a démontré que la relation Ebeta_reduc est confluente en utilisant l'assistant de preuve Coq.

CHAPITRE I

THEORIE DU LAMBDA-CALCUL

Introduction

En tant que théorie générale des fonctions mathématiques, le lambda-calcul fut introduit par Church (1933-1934) dans le cadre de recherches des fondements des mathématiques [22]. Bien que Kleene et Rosser (1936) ont démontré l'inconsistance du système original de Church, le lambda-calcul est une approche couronnée de succès en ce qui concerne un effort de formalisation légèrement moins ambitieux et qui avait lieu au même moment : la formalisation du calcul. Church (1936) avait déjà proposé la notion de lambda-définissable, basée sur le lambda-calcul, comme un candidat à la définition formelle de la notion d'*effectivement calculable*. Un peu plus tard quand Turing (1936, 1937) proposa sa propre analyse des machines de calcul par les notions universellement acceptées de *Machines de Turing* et de *Turing Calculable*, il prouva lui-même que cette dernière notion était équivalente à la notion de lambda-définissable.

Aujourd'hui le lambda-calcul en tant que modèle de calcul est un outil performant pour l'informaticien théorique. En effet et contrairement à l'approche *bas niveau* de Turing, le lambda-calcul se base sur des notions *haut niveau* qui sont des pierres d'angles de la plupart des langages de programmation : les variables représentent les données et les abstractions (fonctions) représentent des procédures.

Par conséquent, il peut se révéler utile de construire la traduction d'un langage de programmation vers le lambda-calcul afin de donner une sémantique formelle au langage. C'est ce que proposait déjà Landin (1965) pour le langage ALGOL en 1965.

Le succès du lambda-calcul par cette approche mena alors à la définition de langages

de programmation fonctionnels comme OCaml (Leroy *et al.*, 2009) ou encore Haskell (Peyton-Jones, 2003) qui ne sont plus reliés à la couche matérielle qu'au travers de la compilation. Ces langages sont directement inspirés par la syntaxe et la sémantique du lambda-calcul et par conséquent bénéficient de son grand pouvoir expressif. Le pont construit par Turing entre le lambda-calcul et le calcul bas niveau n'a cependant pas comblé la brèche : c'est en effet toujours aujourd'hui un domaine de recherche actif qui a produit des techniques très efficaces comme les indices de De Bruijn (1972) ou les substitutions explicites (Abadi *et al.*, 1991; Bloo et Rose, 1995; Benaissa *et al.*, 1996).

À l'instar des informaticiens, les logiciens n'ont pas non plus abandonné le lambda-calcul. En effet De Bruijn (1968) proposa un système informatique appelé Automath dédié à la construction de preuves mathématiques formelles. Le métalangage qu'il utilisa pour formaliser les preuves est très proche du lambda-calcul.

Beaucoup plus tôt, Curry (1934) avait observé que les combinateurs de sa Logique Combinatoire (un système équivalent au lambda-calcul) correspondaient en fait aux schémas d'axiomes de la logique intuitionniste implicationnelle, puis (1958) qu'un fragment de la Logique Combinatoire coïncidaient avec un fragment des systèmes déductifs à la Hilbert.

Enfin Howard (1980) observa que la déduction naturelle intuitionniste de Gentzen peut être interprétée comme une variante typée du lambda-calcul : à travers d'un système de typage, les preuves en déduction naturelle sont mises en correspondance avec une certaine classe de lambda-termes (les termes simplement typés). Ce qu'on appelle la correspondance de Curry-Howard.

Dans ce chapitre, nous allons présenter les concepts de base utiles pour la compréhension de la suite de notre mémoire. Nous rappelons très brièvement la théorie du lambda-calcul pur, la théorie de lambda-calcul typé et la logique combinatoire.

1.1. Lambda-calcul pur :

1.1.1. La syntaxe des λ-expressions :

Le lambda calcul est caractérisé par :

1. L'ensemble des λ-expressions ou λ-termes.
2. Les règles de réécriture permettant de modifier ces λ-expressions.

Définition 1.1 :

Le lambda calcul définit des entités syntaxiques que l'on appelle des lambda-termes et qui se rangent en trois catégories :

[1]. Les variables : x, y... sont des lambda-termes ;

[2]. les applications : u v est un lambda-terme si u et v sont des lambda-termes ;

[3]. les abstractions : λ x.v est un lambda-terme si x est une variable et v est un lambda-terme.

L'application peut être vue ainsi : si u est une fonction et si v est son argument, alors (u v) est le résultat de l'application de la fonction u à v.

L'abstraction λx.v peut être interprétée comme la formalisation de la fonction qui, à x, associe v, où v contient en général des occurrences de x.

Ainsi, la fonction f qui prend en paramètre le lambda-terme x et lui ajoute 2 (c'est-à-dire en notation mathématique courante la fonction f: x → x+2) sera dénotée en lambda -calcul par l'expression *λ x.x+2.*

L'application de cette fonction au nombre 3 s'écrit (λ x.x+2)3 et s'évalue (ou se normalise) en l'expression 3+2.

Avant de donner les règles de réécriture nous avons besoin d'introduire quelques notations et définitions.

Notations :

En général, on n'écrit pas les parenthèses les plus externes et on utilise la notation suivante :

1. On regroupe les abstractions successives sous un seul « λ » :
   λx1x2…xn.M ≡ λx1λx2…λxn.M =(λx1(λx2…(λxn.M)…)).
2. Les applications sont regroupées implicitement à gauche. Donc M N1 N2… Nn≡ (…((M N1)N2)… Nn) .
3. Dans la suite $=_{def}$ et ≡ désignent respectivement l'égalité définitionnelle et l'identité syntaxique.

Définition 1.2 (longueur d'un terme)

La longueur d'un terme notée Lg(t) est définie par induction comme suit :

[1]. *Pour toute variable x, Lg(x) =1.*

[2]. *Lg(M N) = Lg (M) +Lg (N).*

[3]. *Lg(λx.M) =1+Lg(M).*

Remarque 1.1

Raisonner par induction sur la structure d'un λ-terme est équivalent à raisonner par induction sur Lg (t).

Définition 1.3 (l'ensemble des sous termes)

L'ensemble des sous-termes d'un terme t est l'ensemble des termes obtenus au cours de la construction de t, on le définit formellement par induction sur la structure de t.

[1]. $S(x) =_{def} \{x\}$.

[2]. $S((u)v) =_{def} \{(u)v\} \cup S(u) \cup S(v)$.

[3]. $S(\lambda x.u) =_{def} \{\lambda x.u\} \cup S(u)$.

Dans les expressions mathématiques en général et dans le lambda calcul en particulier, il y a deux catégories de variables : **les variables libres** et **les variables liées** (ou *muettes*).

Définition 1.4 (les variables libres) On définit l'ensemble FV(t) des variables libres d'un terme t par induction sur la structure de t :

[1]. *Si t est une variable alors FV(t) = {t}*

[2]. *Si* $t \equiv (u\ v)$, *alors* $FV(t) = FV(u) \cup FV(v)$

[3]. *Si* $t \equiv \lambda x.u$, *alors* $FV(\lambda x.u) = FV(u) \setminus \{x\}$

Définition 1.5 (les variables liées)

Une occurrence liée de *t* est une variable apparaissant dans *t* derrière le symbole λ. L'ensemble des variables liées est donné par induction sur la structure du terme comme suit :

[1]. $BV(x) = \Phi$.

[2]. $BV(M\ N) = BV(M) + BV(N)$.

[3]. $BV(\lambda x.M) = BV(M) + \{x\}$.

Définition 1.6 (terme clos)

Un terme clos (appelé aussi combinateur) est un terme qui n'a pas de variable libre.

Exemple 1.1:

Dans λx.xy, la variable *x* est liée et la variable *y* est libre. On peut réécrire ce terme en λt.ty.

λxyzt.z(xt)ab(zsy) est équivalent à λwjit. i(wt)ab(isj).

1.1.2. Curryfication :

L'abstraction en lambda calcul se fait selon une seule variable. Il n'y a pas à proprement parler de « fonctions à plusieurs variables". En mathématiques, la situation est identique, et on utilise à la place d'une fonction selon les variables a et b une fonction sur le couple (a , b) (on a alors un seul paramètre, le couple). Les couples ne sont pas accessibles dans le lambda calcul de base, donc la solution choisie est différente : on représente une fonction à deux arguments comme une fonction selon le premier argument, qui renvoie une fonction selon le deuxième argument, qui renvoie le résultat final. On appelle cette transformation la **curryfication** (en mathématiques, pour A, B et C des ensembles, elle correspond à l'isomorphisme entre $(A \times B) \rightarrow C$ et $A \rightarrow (B \rightarrow C)$).

1.1.3. Les règles de réécriture :

1.1.3.1. La substitution :

La substitution à la variable x du *λ-t*erme t dans le *λ*-terme M, notée M [t/x] (Penser « M avec t pour x »), correspond à un remplacement des occurrences de x par t dans M.
Il faut cependant prendre garde au phénomène de **capture** qui rend la définition formelle de la substitution un peu plus délicate : on veut que seules les variables libres de M soient remplacées, et que des variables libres de t ne deviennent pas liées dans M. Par exemple, si on remplace naïvement x par y dans « *λ y. x* » (intuitivement, la fonction qui à y associe x), on obtient « *λy. y* », qui est la fonction identité: l'occurrence libre de *y* est devenue liée (on dit qu'elle a été capturée), ce qui change la signification du *λ*-terme.

1.1.3.1.1. Substitution avec renommage :

On définit la substitution par induction sur la structure du terme M :

I. *Si* $M \equiv x$ *alors* $M[t / x] =_{def} t$

II. *Si M est une variable et* $M \neq x$ *alors* $M[t / x] =_{def} M$

III. *Si* $M \equiv (AB)$ *alors* $(A\ B)[t / x] =_{def} A[t / x]\ B[t / x]$

IV. *Si* $M \equiv (\lambda y.E)$ *et* $x \notin FV(M)$ *alors* $(\lambda y.E)[t / x] =_{def} \lambda y.E$

V. *Si* $M \equiv (\lambda y\ .E)$ *et* $y \notin FV(t)$ *alors* $(\lambda y\ .E)[t / x] =_{def} \lambda y.\ (E\ [t / x])$

VI. *Si* $M \equiv (\lambda y\ .E)$ *et* $y \in FV(t)$ *alors* $(\lambda y\ .E)[t / x] =def \lambda z.\ (E[z / y][t / x])$

où $z \notin FV(M)$ *et* $z \notin FV(t)$.

Exemple 1.2 :

1. $(\lambda x.xy)[a / y] =_{def} \lambda x.xa$
2. $(\lambda x.xy)[x / y] =_{def} \lambda z.zx$
3. $(\lambda x.xx)(\ \lambda x.xx)\ [\ t / x\ ] =_{def} (\lambda x.xx)(\ \lambda x.xx)$.

1.1.3.2. L'α-équivalence :

Le lambda-calcul a originellement été proposé comme une formalisation de la théorie des fonctions mathématiques. En l'occurrence, les lambda-termes symbolisent des fonctions. Par exemple le lambda-terme $\lambda x.x$ représente la fonction identité qui associe à n'importe quel objet M le même objet M. Dans l'expression $\lambda x.x$, le symbole x est une variable représentant « n'importe quel objet M » qui pourrait être l'argument de la fonction identité. Nous aurions pu d'ailleurs choisir un autre symbole de variable : par exemple si nous avions choisi y, nous aurions obtenu $\lambda y.y$. C'est d'ailleurs une autre représentation syntaxique de la fonction identité.

Les lambda-termes qui diffèrent seulement par le changement d'une variable liée sont dits **α-congruents** (α-convertibles). L'α-congruence (élémentaire) est définie par:

$$\lambda x.P =_{\alpha} \lambda y.\ P\ [y / x] \qquad \text{où } y \notin FV(P).$$

Nous dirons aussi que le λ-terme X est congruent au λ-terme Y (X est α–convertible à Y), noté

$$X =_{\alpha} Y$$

Exemple 1.3

1. $\lambda x.\ \lambda y.y =_\alpha \lambda x.\lambda x.x.$
2. $(\lambda y.\ \lambda x.xy)xz =_\alpha (\lambda y.\lambda x'.x'y)xz.$
3. $(\lambda x.\lambda x.xx)x =_\alpha (\lambda x.\ \lambda y.yy)x =\alpha (\lambda z.\ \lambda x.xx)x.$
4. $\lambda x.y \neq_\alpha \lambda x.x$ si x et y sont des variables distinctes.

Remarque 1.2

=α est une relation d'équivalence c'est-à-dire que:

- =α est réflexive : $u =_\alpha u$,
- =α est symétrique : $u =_\alpha v \Rightarrow v =_\alpha u$,
- =α est transitive : $u =_\alpha v,\ v =_\alpha w \Rightarrow u =_\alpha w$.

Il est immédiat que, si $u =_\alpha u'$ alors :

1. *u'* est de même nature que u (variable, abstraction ou application).
2. *u* et *u'* ont la même longueur, les mêmes variables libres et les mêmes occurrences de variables libres.
3. Si *u* ne contient pas de λ alors $u \equiv u'$.

1.1.3.3. Substitution et convention de Barendregt :

1.1.3.3.1. La convention de Barendregt :

C'est une **convention sur les variables libres** d'un terme dans un énoncé mathématique [20].

Il n'existe aucun sous-terme dans lequel une variable apparaît à la fois libre et liée.

- On suppose que dans tout théorème que l'on énonce, on suit la convention de Barendregt.
- Si l'on a un terme qui ne satisfait pas la convention de Barendregt, on s'y ramène par l'α-conversion.

Avec la convention de Barendregt, la définition des substitutions devient beaucoup plus simple.

I. Si $M \equiv x$ *alors* $M[t / x] =_{def} t$

II. *Si M est une variable et* $M \neq x$ *alors* $M[t / x] =_{def} M$

III. *Si* $M \equiv (AB)$ *alors* $(A\ B)[t / x] =_{def} A[t / x]\ B[t / x]$

IV. *Si* $M \equiv (\lambda y.E)$ *et* $x \notin FV(M)$ *alors* $(\lambda y.E)[t / x] =_{def} \lambda y.E$

V. *Si* $M \equiv (\lambda y\ .E)$ *et* $y \notin FV(t)$ *alors* $(\lambda y\ .E)[t / x] =_{def} \lambda y.\ (E\ [t / x])$

1.1.3.4. La β-réduction

Le mécanisme le plus important du lambda-calcul est appelé la bêta-réduction.

C'est le mécanisme qui permet d'évaluer une fonction $\lambda x.M$ appliquée à un certain argument N.

Un λ -terme de la forme "$(\lambda\ x.M)\ N$" est appelé **β_redex**[1], il lui correspond le β_contractum $M\ [N / x]$.

Une étape de calcul à l'intérieur d'un terme *t,* consiste à choisir un redex arbitraire et à le remplacer par son réduit.

On écrit $\boldsymbol{t \rightarrow_{\beta 0} u}$ si *u* est obtenu à partir de *t* en effectuant une étape de réduction.

Cette relation est définie comme suit :

[1]. $((\lambda x\ .\ t)\ u) \rightarrow_{\beta 0} t\ [u / x]$.

[2]. *Si* $t \rightarrow_{\beta 0} u$ *alors* $(t\ v) \rightarrow_{\beta 0} (u\ v)$.

[3]. *Si* $t \rightarrow_{\beta 0} u$ *alors* $(v\ t) \rightarrow_{\beta 0} (v\ u)$.

[4]. *Si* $t \rightarrow_{\beta 0} u$ *alors* $\lambda x.t \rightarrow_{\beta 0} \lambda\ x\ .u$.

La relation $\boldsymbol{t \rightarrow_{\beta *} u}$ **(*t* se β-réduit sur *u*)** est définie comme la fermeture réflexive-transitive de la relation $\rightarrow_{\beta 0}$, c'est-à-dire comme la plus petite relation telle que :

[1]. *Si* $t \rightarrow_{\beta 0} u$ *alors* $t \rightarrow_{\beta *} u$,

[2]. $t \rightarrow_{\beta *} t$,

[3]. *Si* $t \rightarrow_{\beta *} u$ *et* $u \rightarrow_{\beta *} v$ *alors* $t \rightarrow_{\beta *} v$.

La relation $\boldsymbol{t =_{\beta} u}$ **(t est β-équivalent à u)** est définie comme la fermeture réflexive-symétrique –transitive de la relation $\rightarrow_{\beta 0}$, c'est-à-dire comme la plus petite relation telle que:

[1] Le mot **redex** veut dire en anglais **red**ex **ex**pression

[1]. *si* $t \rightarrow_{\beta 0} u$ *alors* $t =_\beta u$,

[2]. $t =_\beta t$,

[3]. *si* $t =_\beta u$ alors $u =_\beta t$,

[4]. *si* $t =_\beta u$ et $u =_\beta v$ alors $t =_\beta v$.

Exemple 1.4:

- $(\lambda x.\ x(x\ y))N \rightarrow_{\beta 0} (x(xy))\ [N / x] \equiv N(Ny)$.
- $(\lambda x.y)\ N \rightarrow_{\beta 0}\ y[N / x] \equiv y$.
- $(\lambda x.xx)(\lambda y.y)N \rightarrow_{\beta 0}\ (xx)\ N\ [\lambda y.\ y / x]$

$\equiv (\lambda y.y)\ (\lambda y.y)N$

$\rightarrow_{\beta 0}\ y\ N\ [\lambda y.y /y]$

$\equiv (\lambda y.y)N$

$\rightarrow_{\beta 0}\ y[N/y]$

$\equiv\ N$

1.1.3.5. βη-réduction :

On appelle **η-redex** un λ -terme de la forme "$\lambda\ x.Mx$" où $x \notin FV(M)$. Son contractum est M. La βη -réduction est l'extension de la λβ -réduction obtenue en introduisant l'axiome (η):

$$(\eta)\ \lambda x.\ M\,x \rightarrow M \qquad\qquad x \notin FV(M).$$

1.1.4. Normalisation

Un λ-terme X qui ne contient aucun β-redex est dit en **β-forme normale (β-fn)**.

Si M se β-réduit à N et N est en β-forme normale alors N est appelé la β-forme normale de M.

La question de savoir quels sous-termes réduire en premier n'est pas forcément évidente : quand on veut réduire « mécaniquement » les sous-termes au lieu de le faire à la main (par exemple si l'on code un interprète ou compilateur pour un λ-calcul ou langage dérivé), il faut choisir une procédure décidant du sous-terme à réduire le premier. On appelle ces procédures **des stratégies de réduction**. Le choix de la stratégie de réduction peut être très important dans un langage de programmation.[2]

[2] Pour plus de détail consulter la section 1.5 « stratégies de réduction » du chapitre 1.

Définition 1.7 :

- Un terme u est **fortement normalisable** si et seulement si toutes les réductions partant de u sont finies.
- Un terme u est **faiblement normalisable** si et seulement si au moins une réduction partant de u est finie.

Exemple 1.5 :

- Posons $\Delta \equiv \lambda x.xx$, $\Omega = (\lambda x.xx)(\lambda x.xx) = \Delta\Delta$.

  Le lambda terme Ω n'est non seulement pas fortement normalisable, il n'est même pas faiblement normalisable, car sa réduction boucle indéfiniment sur elle-même.

  $(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx)$.

- $(\lambda x.x)((\lambda y.y)z)$ est un lambda-terme fortement normalisable et sa forme normale est z.
- $(\lambda x.y)(\Delta\Delta)$ est normalisable et sa forme normale est *y*, mais il n'est pas fortement normalisable car la réduction du terme $(\lambda x.y)(\Delta\Delta)$ peut aboutir au terme *y* mais aussi au terme $(\lambda x.y)(\Delta\Delta)$.
- $(\lambda x.xxx)(\lambda x.xxx) \rightarrow (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow$ ... crée des termes de plus en plus grand.

Tous les termes fortement normalisables sont faiblement normalisables, mais le contraire n'est pas vrai.

1.1.5. Confluence

Church et Rosser(1936), ont énoncé la confluence duλ -calcul muni de la β-réduction, c'est-à-dire que pour deux réductions issues d'unλ -terme t quelconque atteignant des termes u et v, on peut toujours trouver un λ -terme w où elles se rejoignent. L'importance de ce résultat de confluence vient de deux conséquences fondamentales de ce théorème. En effet, la confluence assurant que si un terme possède une forme normale, celle-ci est unique, il entraîne que :

- la théorie de la β-équivalence est consistante ;
- la notion de forme normale peut être prise comme notion de valeur dans les versions opérationnelle du λ -calcul.

1.1.5.1. Diverses notions de confluence

Définition 1.8 (Confluence, confluence locale (ou faible), confluence forte) :

Soit *R* une relation binaire sur un ensemble A. On définit les notions suivantes :

- R est confluente si pour tout a, b, c ∈ A; a *R** b & a *R** c
  => Il existe d ∈ A; b *R** d & c *R** d ;
- *R* est localement confluente si pour tout a, b, c ∈ A; a R b & a R c
  => Il existe d ∈ A; b *R** d & c *R** d ;
- *R* est fortement confluente si pour tout a, b, c ∈ A; a *R* b & a *R* c
  => il existe d ∈ A; b *R* d & c *R* d.

On désignera par *R** la fermeture réflexive-transitive de la relation *R*.

1.1.5.2. La relation entre les différentes formes de confluence

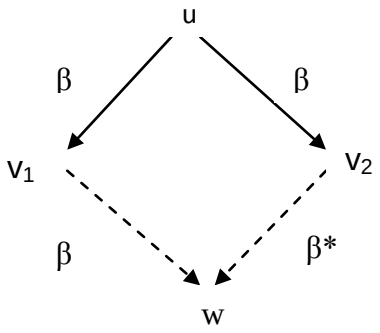
On désignera respectivement par CR, WCR et SCR les propriétés de confluence, confluence locale et la confluence forte.

On a la chaîne d'implications suivante : SCR ➔ CR ➔ WCR

1.1.5.3. Le théorème de Church-Rosser et ses conséquences

La β-réduction est confluente c'est-à-dire pour tout λ-termes u v_1 v_2, ***si u*** $\rightarrow_{\beta^*}$ ***v_1 et u*** $\rightarrow_{\beta^*}$ ***v_2*** alors il existe un λ-terme w tel que ***v_1*** $\rightarrow_{\beta^*}$ ***w et v_2*** $\rightarrow_{\beta^*}$ ***w.***

On utilisera pour écrire ce théorème la notation diagrammatique plus parlante:



Où les lignes pleines indiquent des réductions universellement quantifiées, et les lignes pointillées des réductions dont l'existence est affirmée.

Corollaire 1: Si M a une forme normale, celle-ci est unique.

Corollaire 2 : Si M = $_\beta$ N, alors il existe P tel que M $\rightarrow_{\beta^*}$ P et N $\rightarrow_{\beta^*}$ P.

1.2. Lambda-calcul simplement typé :

Remarquons qu'avec la définition de la β-réduction, définie sur les expressions du lambda-calcul pur, il est possible d'appliquer un terme à un terme quelconque, ce qui peut aboutir à des résultats inconsistants. Pour éviter quelques anomalies de cette application, le lambda-calcul typé peut intervenir [13]. Typer un lambda-terme revient à déterminer son type dans le but de vérifier s'il est correct et dans quel contexte l'utiliser, et aussi pour assurer la terminaison du calcul. Ce phénomène permet de restreindre les expressions du langage à une classe particulière de fonctions.

Par exemple l'expression du lambda-calcul simplement typé suivante :

$\lambda (f : i \rightarrow i)\ \lambda(x : i).(fx)$

où i est un type arbitraire. Cette expression dénote la fonction qui prend en argument « f » (une valeur fonctionnelle) de type « $i \rightarrow i$ » et x de type « i », et retourne le résultat de l'application « fx ».

Le λ-calcul simplement typé est un formalisme fortement lié à la déduction naturelle.[3] Avec une telle interprétation les termes du λ-calcul représentent les dérivations de la déduction naturelle ; c'est-à-dire à chaque nœud, de l'arbre de dérivation, correspond un λ-terme et les types des λ-termes représentent les formules de la déduction naturelle.

1.2.1. Syntaxe :

Considérons un langage dont les termes contiennent des informations de type. Plus précisément, le type d'un paramètre sera défini lors de la construction d'une abstraction. Il s'agit d'une représentation du lambda-calcul simplement typé dite *représentation à la Church.* Il est possible de garder les mêmes termes du lambda-calcul pur (aucune information de type dans les termes), cette représentation est dite *à la Curry*.

Les types considérés sont un type de base *Var* et les types fonctionnels : $\tau \rightarrow \tau'$ (types des fonctions dont le paramètre est de type τ et le résultat de type τ').

$$T ::= i \mid T \rightarrow T$$

La flèche est associative à droite : ainsi l'expression de type $T_1 \rightarrow T_2 \rightarrow T_3$ se lit $T_1 \rightarrow (T_2 \rightarrow T_3)$.

[3] Pour plus de détail consulter la section déduction naturelle de chapitre 2.

La syntaxe des lambda-termes du lambda-calcul simplement typé est définie par la grammaire suivante :

$$\begin{aligned} <Term> \;\longrightarrow\; & x : \alpha \\ & \mid \lambda x : \alpha . <Term> \\ & \mid <Term>\ <Term> \end{aligned}$$

1.2.2. Système de type:

Un système de type est composé d'un langage de type et d'un ensemble de règles de typage. Le langage de type a déjà été donné et les informations de type relatives aux variables non liés des expressions sont maintenues dans un environnement de typage.

Un environnement de typage peut être vu comme une fonction partielle des variables vers des types qui associe à une variable son type.

Les règles de typage décrivent pour chaque construction du langage, les conditions qui permettront de confirmer que cette construction est bien typée : par exemple la construction de type « MN » est bien typée si l'expression « M » est bien typée et a le type fonctionnel de la forme « $\tau \rightarrow \tau'$ » et si « N » est aussi bien typé et de type « τ ». Le prédicat « Est Bien Typé » est défini inductivement en fonction de l'environnement de typage, le lambda-terme et le type de ce dernier. Les séquents de typage sont indiqués dans la figure suivante :

$$\text{(VAR)} \qquad \Gamma \vdash x : \alpha$$

$$\text{(ABS)} \qquad \frac{\Gamma \cup \{x : \alpha\} \vdash M : \beta}{\Gamma \vdash \lambda x.M : \alpha \rightarrow \beta}$$

$$\text{(APP)} \qquad \frac{\Gamma \vdash M : \alpha \rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

Figure 1-1 : les règles de typage

$\Gamma \vdash x : \alpha$ désigne le type associé à la variable x dans l'environnement Γ.

$\Gamma \cup \{x : \alpha\}$ pour étendre un environnement avec une nouvelle hypothèse.

L'écriture $\Gamma \vdash M : \alpha$ indique que le lambda terme « M » a le type α relativement à l'environnement Γ.

Un terme M a le type α dans l'environnement Γ, si le séquent $\Gamma \vdash M : \alpha$ est dérivable dans le système formel formé par les trois règles de typage.

Exemple 1.6 :

Le terme $\lambda x :\alpha.x$ est bien typé, il a le type « $\alpha \to \alpha$ » dans l'environnement vide. En effet on peut construire l'arbre de dérivation suivant :

$$\frac{(x : \alpha) \vdash x : \alpha}{\vdash (\lambda x :\alpha.x) : \alpha \to \alpha}$$

De même le terme « $\lambda x :i.\lambda f :i \to i \to i.fxx$ » est un terme bien typé de type :

« $i \to (i \to i \to i) \to i$ »

$$\frac{\dfrac{\dfrac{(x :i)\ (f : i \to i \to i) \vdash f : i \to i \to i \ ,\ (x :i)\ (f : i \to i \to i) \vdash x: i}{(x :i)\ (f : i \to i \to i) \vdash (fx): i \to i} \ , (x :i)\ (f : i \to i \to i) \vdash x: i}{(x :i)\ (f : i \to i \to i) \vdash (fxx):i}}{(x :i) \vdash (\lambda f : i \to i \to i.fxx): (i \to i \to i) \to i}$$

$$\lambda x :i.\lambda f : i \to i \to i.fxx):i \to (i \to i \to i) \to i$$

En revanche, il est impossible de construire une dérivation permettant de montrer que le terme « $\lambda x :\alpha.xx$ » est bien typé (quelque soit le type α).

1.2.3. Normalisation forte :

La normalisation forte d'un terme est la propriété qui affirme que le processus de réduction se termine. Cette propriété est satisfaite dans les calculs de substitutions explicites pour les termes typables.

Pour réduire les termes nous conservons les mêmes règles de Bêta-réductions. Le calcul reste confluent. Si on restreint aux termes bien typés, la terminaison du calcul est assurée.

Si un terme est bien typé, alors il admet une forme normale et une seule [13].

1.3. Extension de lambda-calcul :

L'utilisation du paradigme type as formula, pour interpréter la logique est limité à la logique propositionnel. Dans le but d'interpréter les logiques d'ordre supérieur des extensions pour le lambda-calcul typé sont définies. Parmi ces extensions nous présenterons dans ce qui suit le système polymorphique du second ordre $\lambda 2$ et le calcul des constructions inductives.

1.3.1. Le système polymorphique du second ordre $\lambda 2$:

Il s'agit d'une extension du système de Curry introduite par Girard et Reynolds [4]. Il est défini comme suit :

1. L'alphabet de ce système est défini par deux ensembles :
   - propvars = ensemble des variables propositionnelles
   - termvars = ensemble des variables de termes.
2. La syntaxe formelle des expressions de $\lambda 2$ est définit par la grammaire suivante :

<proposition> ::= <propvars> |
(<proposition> → <proposition>) |
(∀ <propvars>) <proposition>

<term> ::= <termvars> |
(<term> <term>) |
<term>{<proosition>} |
(λ<termvars> : <proposition> . <term>) |
(Λ<propvars>. <term>)

<formule> ::= <term> : <proposition>

Figure 1-2 : Syntaxe formelle du système λ2.

[4] Pour plus de détaille consulter le chapitre 16 du [15].

3. Axiome : aucun.
4. Les règles d'assignation de type dans le système λ2 est données par le tableau suivant :

	Introduction	**Elimination**
→	$\dfrac{\Gamma\ (x : A) \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \rightarrow B}$	$\dfrac{\Gamma \vdash M : A, \Gamma \vdash N : A \rightarrow B}{\Gamma \vdash MN : B}$
∀	$\dfrac{M : A}{\vdash \Lambda p{:}M : (\forall p)\ A}$	$\dfrac{\Gamma \vdash M : \forall p\ A(p)}{\Gamma \vdash M\ \{B\}\ : A[B/p]}$

Condition : x n'est pas libre dans aucune hypothèse non déchargée

Figure 1-3 : Règles d'inférence du système λ2

Soit par exemple le terme $D_{A,B} \equiv \lambda x : A\ \lambda y : B\ \lambda z : A \rightarrow (B \rightarrow p).\ z\ x\ y$ défini en fonction de deux propositions A, B. Le terme $D \equiv \Lambda p\ \Lambda q.D_{A,B}$ est la fonction appliquée aux propositions A et B donne la valeur $D_{A,B}$. Mais si nous voulons définir une fonction f telle que $f\ \{A\}\{B\} =_{\beta} A \wedge B$, nous nous heurterons aux limites de la syntaxe du langage du système λ2. Le terme $A \wedge B \equiv\ \forall\ p((A \rightarrow (B \rightarrow p)) \rightarrow p)$ appartient au système λ2 mais le terme $\Lambda u.\ \Lambda v.\forall p\ ((u \rightarrow (v \rightarrow p)) \rightarrow p)$ ne peut être engendré par la grammaire du langage.
Cette difficulté est contournée en introduisant la possibilité de quantifier sur le domaine des propositions. Ceci est possible dans le calcul des constructions.

1.3.2. Le calcul des constructions :

1. La syntaxe formelle des expressions de ce système est définit par la grammaire suivante :

```
<term>     :: =     <var> |
                    Prop |
                    (<term> <term>) |
                    λ <var> : <term>.<term> |
                    (∀ <var> : <term>) <term>

<formule> ::= <term> : <term> | <term> : Type
```

Figure 1-4 : Syntaxe formelle du calcul des constructions

2. Axiome : Prop : Type.
3. Les règles de construction des types pour cette extension sont comme suit :

$$\frac{\Gamma, x :\alpha \vdash \beta : Prop \quad \Gamma \vdash \alpha : Prop}{(\forall\, x :\alpha)\beta : Prop}\ (P\,\forall_1) \qquad \frac{\Gamma, x :\alpha \vdash \beta : Prop \quad \Gamma \vdash \alpha : Type}{(\forall\, x :\alpha)\beta : Prop}\ (P\,\forall_1)$$

$$\frac{\Gamma, x :\alpha \vdash \beta : Type \quad \Gamma \vdash \alpha : Prop}{(\forall\, x :\alpha)\beta : Type}\ (T\,\forall_1) \qquad \frac{\Gamma, x :\alpha \vdash \beta : Type \quad \Gamma \vdash \alpha : Type}{(\forall\, x :\alpha)\beta : Type}\ (T\,\forall_1)$$

Condition : x n'est pas libre dans α et dans aucune hypothèse non déchargée.

4. Les règles d'assignation de type pour cette extension sont comme suit :

	Introduction	**Elimination**
∀	$\dfrac{\Gamma, x :\alpha \vdash M : \beta \quad \Gamma \vdash \alpha : Prop}{\Gamma \vdash \lambda\, x{:}\alpha.M : (\forall\, x{:}\alpha)\beta}\ (\forall\, P)$ $\dfrac{\Gamma, x :\alpha \vdash M : \beta \quad \Gamma \vdash \alpha : Type}{\Gamma \vdash \lambda\, x{:}\alpha.M : (\forall\, x{:}\alpha)\beta}\ (\forall\, T)$	$\dfrac{\Gamma \vdash N :\alpha \quad \Gamma \vdash M : (\forall\, x :\alpha)\beta}{\Gamma \vdash M\, N : \beta[N/x]}$

Condition : x n'est pas libre dans α et dans aucune hypothèse non déchargée.

Figure 1-5 : Règles d'assignation des types dans le calcul des constructions

1.4. Programmation en λ-calcul

1.4.1. Codage de données simples

Tel quel, le λ- calcul a l'air d'être un langage théorique d'étude des applications de fonctions. Il semble difficile d'y programmer (ou plutôt d'y décrire des programmes) concrètement, du fait du manque d'opérations primitives. Il n'y a même pas de symboles de constantes !

Quand on étudie un langage particulier, il arrive d'étendre le λ-calcul de base avec de nouvelles constructions, ou des opérations et des constantes supplémentaires.

Cependant, le langage donné suffit en fait à construire la plupart des objets « classiques » des langages de programmation : il s'agit de représenter des concepts (ici, des structures de données : booléens, entiers naturels, etc.) sous forme de λ -termes. On parle de codage ; les codages présentés portent le nom général de **church encoding**.

1.4.2. Booléens

Pour commencer à représenter des programmes en lambda-calcul, on aimerait disposer d'une instruction conditionnelle. On voudrait encoder une structure de la forme (IF p THEN a ELSE b), qui vaut a si le booléen p est vérifié, et b sinon. Il s'agit de choisir une représentation des booléens en λ -calcul qui rende cette forme naturelle (et qui existe !).

L'idée la plus naturelle est d'utiliser une application de fonctions. On voudrait pouvoir écrire ce test tout simplement p a b, ce qui revient à représenter les deux valeurs booléennes, « vrai » et « faux », par des fonctions à deux arguments.

$$vrai := \lambda\ a\ b.\ a$$
$$faux := \lambda\ a\ b.\ b$$

Cet encodage permet effectivement de représenter les booléens. On peut alors écrire avec les fonctions booléennes usuelles. La forme (si p a b), équivalente à
(IF p THEN a ELSE b), est par construction particulièrement simple.

$$si := \lambda\ p\ a\ b.\ p\ a\ b$$

On peut aussi représenter la disjonction, la conjonction et la négation.

$$et := \lambda\ p.\ \lambda\ q.\ p\ q\ faux$$
$$ou := \lambda\ p.\ \lambda\ q.\ p\ vrai\ q$$
$$non := \lambda\ p.\ \lambda\ a.\ \lambda\ b.\ p\ b\ a$$

Il faut noter que cet encodage n'est pas unique : on pourrait par exemple inverser les définitions de vrai et faux, et on pourrait toujours définir ces opérations (il suffirait d'inverser « et » et « ou », ainsi que si et non).

1.4.3. Entiers

On peut aussi représenter les entiers naturels. L'idée sous-jacente est de représenter l'entier n par l'opération d'itération n fois d'une fonction : $f \rightarrow (x \rightarrow f^n(x))$

$$0 := \lambda fx.\ x$$
$$1 := \lambda fx.\ fx$$
$$2 := \lambda fx.\ f\ (fx)$$

Plus généralement, on peut définir une fonction succ qui prend en paramètre le code d'un entier, et renvoie le code de son successeur.

$$succ := \lambda n.\ \lambda fx.\ f(nfx)$$

Enfin, on peut coder un prédicat iszero qui renvoie vrai si l'entier fourni est nul, et faux sinon. Il suffit d'itérer n fois la fonction qui renvoie toujours faux, en lui donnant vrai comme valeur initiale : si n > 0, la fonction sera appliquée et renverra faux, sinon f^n est l'identité et renvoie vrai.

$$iszero := \lambda n.\ n\ (\lambda x.\ faux)\ vrai$$

Enfin, on peut coder facilement diverses fonctions numériques usuelles.

$$plus := \lambda n\ m\ \lambda f\ x.\ n\ f\ (m\ f\ x)$$
$$mult := \lambda n\ m.\ \lambda f.\ n\ (m\ f)$$

Pour des raisons de lisibilité, on notera (a+b) et (a × b) au lieu de (plus a b) et (mult a b) respectivement, mais cela reste des termes du λ-calcul pur.

1.4.4. Paires

Enfin, on voudrait pouvoir représenter des couples de λ-termes. On sait déjà représenter des entiers, donc on pourrait imaginer d'implémenter l'usuelle bijection de N^2 vers N, pour représenter un couple d'entier par un simple entier. Pour généraliser à tout λ-terme, il suffit de choisir un Godel-codage sur les λ –termes pour les représenter par des entiers. Cependant, il faudrait alors pouvoir reconstruire le lambda-terme correspondant à son Godel-codage, soit

en substance implémenter un méta-évaluateur (de λ-calcul en λ-calcul). Pour commencer, on ne sait même pas coder la fonction pred (qui à 0 associe 0, et à n associe n - 1).

Il existe une solution beaucoup plus simple et naturelle. Il suffit de représenter les couples comme une application partielle attendant les deux membres du couple comme arguments :

$$pair := \lambda\, x\, y.\ \lambda\, f.\ f\, x\, y$$

(pair a b) construira donc un λ-terme qui prend en paramètre une fonction f et l'applique aux deux éléments du couple. En particulier, si l'on veut obtenir l'un des deux éléments, il suffit de lui passer une fonction de sélecteur qui renvoie soit le premier, soit le second de ses arguments. Par chance, nous avons déjà rencontré ces fonctions, ce sont les booléens.

$$fst := \lambda\, p.\ p\ vrai$$
$$snd := \lambda\, p.\ p\ faux$$

Pour des raisons de lisibilité, on notera (a, b) au lieu de (pair a b).

Il devient alors possible de coder plutôt simplement la fonction pred. L'idée est d'associer par itérations un couple valant (pred n , n) à l'entier n : pour 0 il suffit de donner (0, 0), et ensuite d'itérer n fois l'opération (a, b) → (b, b + 1).

$$succpair := \lambda\, p.\ ((\lambda\, n\ (n, succ\ n))(snd\ p))$$
$$pred := \lambda\, n.\ fst\ (n\ succpair\ (0, 0))$$

On peut aussi généraliser le codage des couples à des dimensions supérieures.

Pour n entier, on peut représenter le constructeur de n-uplet (couple à n membres) et les projections.

$$prod_n := \lambda\, f.\ \lambda\, x_1 \ldots x_n.\ f\, x_1 \ldots x_n$$
$$proj_i^n := \lambda\, p.\ p(\lambda\, x1 \ldots x_n .\ x_i)$$

1.4.5. Récursion :

1.4.5.1. Fonctions récursives primitives

Grâce au codage de Church des entiers naturels, on peut définir une notion de calculabilité des fonctions de N dans N selon le λ-calcul : une fonction f' : N→ N est λ-calculable s'il

existe un λ-terme f tel que pour tout entier naturel n' de code n, le résultat de l'évaluation (par β réductions successives) du terme (f n) est le codage de l'entier f' (n').

Les fonctions récursives primitives sont λ-calculables : on sait représenter les fonctions constantes, n-uplets, et projections. Il est clair que l'on peut écrire les projections de fonctions (pour les fonctions d'arité 1, f o g = λ x. f (g x)). Il reste à prouver que l'on peut représenter en λ-calcul la fonction h, définie par récursion primitive à partir de deux fonctions f et g :

$$h(0, x_1, \ldots, x_n) = f(x_1, \ldots, x_n)$$
$$h(n + 1, x_1, \ldots, x_n) = g(h(n, x_1, \ldots, x_n), x_1, \ldots, x_n)$$

Nous traiterons le cas des fonctions d'arité 1, la généralisation ne posant pas de difficulté. Si le terme c représente une constante entière (ou fonction d'arité 0), et g une fonction $N \times N \rightarrow N$ λ-calculable, on voudrait un λ-terme h vérifiant
h(0) = c et h (n + 1) = f(h(n), n).

Si on avait h (n + 1) = f(h(n)), il s'agirait en fait de l'itérée de la fonction h, que l'on peut représenter facilement en λ -calcul. Il manque cependant le paramètre n supplémentaire. On va utiliser la même méthode que pour pred :
On transmet le couple (h(n), n) à la fonction itérée.

$$succf := \lambda\, p.\ p(\lambda\, x\ n.\ (f\ x\ n.\ succ\ n))$$
$$h := \lambda\, n.\ fst\ (n\ (c, 1)\ succf)$$

1.4.5.2. Combinateur de point fixe

On voudrait maintenant pouvoir définir des fonctions récursives. L'exemple le plus courant est la fonction factorielle :

$$fac\ ?= \lambda\, n.\ (iszero\ n\ 1\ (n \times fac(pred\ n))))$$

Une telle définition ne serait pas correcte puisqu'elle utilise le terme fac qui n'est pas défini. On peut la modifier légèrement pour remplacer l'appel récursif de fac par l'appel d'une fonction f passée en paramètre.

$$fac' := \lambda\, f\ n.\ (iszero\ n\ 1\ (n \times f(pred\ n)))$$

Cette fonction fac' n'est pas la factorielle : pour obtenir une factorielle, il faudrait l'appeler en lui donnant « elle-même » en argument :

$$fac = fac'(fac) = fac'(fac'(fac)) = \ldots$$

Autrement dit, on cherche le point fixe de l'opération f fac' f. Plus généralement, on voudrait (afin de pouvoir écrire toute fonction récursive, et pas seulement fac) disposer d'un combinateur de point fixe, c'est-à-dire un λ-terme vérifiant l'équation suivante :

$$Y\ (f) = f\ (Y\ (f))$$

Il se trouve qu'il est possible de définir ce terme en λ-calcul.

$$auto := \lambda\ x.\ x\ x$$
$$Y := \lambda\ f.\ auto\ (\lambda\ x.\ f(auto\ x))$$

Le terme vérifie bien l'équation demandée :

$$Y\ (f) = (\lambda\ f.\ auto\ (\lambda\ x.\ f(auto\ x)))f$$
$$=_{\beta}\ auto\ (\lambda\ x.\ f(auto\ x))$$
$$=_{\beta}\ f\ (auto\ (\lambda\ x.\ f(auto\ x)))$$
$$=\ f\ (Y\ (f))$$

On peut alors définir fac.

$$fac := Y\ (fac')$$

Il est en fait possible de définir fac plus simplement, comme une fonction récursive primitive.

Conclusion Nous constatons que le λ-calcul possède un pouvoir expressif qui lui permet de représenter tout les objets nécessaire pour un langage de programmation fonctionnel.

1.5. Stratégies de réduction

On a vu qu'un λ-terme pouvait contenir plusieurs redex. Si l'on vu calculer la forme normale d'un terme u (si elle existe), on va pouvoir l'obtenir en choisissant un redex dans u, en le contractant, et en répétant le processus jusqu'à ce qu'il n'y ait plus de redex. Il y a donc deux problèmes à résoudre [18].

- Si un terme possède une forme normale, existe-t-il une stratégie de dérivation qui y amène à coup sûr?

- Parmi toutes les dérivations menant à la forme normale, y en a-t-il de meilleures? (Que veut d'ailleurs dire "meilleure"?).

Exemple 1.7

Soit $\Delta = \lambda x.\ x\ x$ *et* $I = \lambda y.y$

Examinons les deux dérivations suivantes:

$\Delta\ (I\ x) \ \rightarrow_\beta\ \Delta\ x\ \rightarrow_\beta\ xx$

$\Delta\ (Ix)\ \rightarrow_\beta\ (I\ x)\ (I\ x)\ \rightarrow_\beta\ x\ (I\ x) \rightarrow_\beta\ x\ x$

La première dérivation qui réduit d'abord les radicaux les plus internes (les plus profonds dans l'arbre) est donc plus courte que la dérivation réduisant d'abord le plus externe. Considérons ensuite les dérivations suivantes:

$(\lambda\ x.x\ I)\ \lambda\ y.\ (\Delta\ (yz)) \rightarrow_{20} (\lambda\ x.xI)\ (\lambda\ y.(yz)(yz)) \rightarrow_\beta (\lambda\ y.(yz)(yz))\ I \rightarrow_\beta (Iz)(Iz) \rightarrow z\ z.$

$(\lambda\ x.x\ I)\ \lambda\ y.(\ \Delta\ (yz)) \rightarrow_\beta \lambda\ y.(\Delta(yz))I \rightarrow_\beta \Delta\ (Iz) \rightarrow_2 \Delta\ z \rightarrow z\ z.$

La dérivation réduisant toujours le radical le plus interne est cette fois plus longue que la seconde dérivation. Trouver une stratégie optimale n'a donc rien d'évident.

1.5.1. Stratégies internes

Prenons le cas d'un langage de programmation usuel, comme OCaml ou Pascal ou C. Dans ces langages, un terme de la forme « u v » (noté u(v) en Pascal ou en C) s'évalue typiquement en calculant d'abord la valeur f de u, qui doit être une fonction, puis en calculant la valeur a de v, puis en appliquant f à a. (En général, u est un identificateur qui dénote déjà une fonction, sans qu'on ait à calculer quelle fonction c'est exactement.)

Remis dans un cadre λ-calculatoire, on peut dire qu'on a d'abord réduit les redex de u, puis quand il n'y en a plus eu, on a réduit les redex de v, enfin on regarde ce qu'est devenu u, et s'il est devenu un terme de la forme « $\lambda\ x\ .\ t$ », on applique cette dernière fonction à v pour donner $t\ [v\ /\ x]$, et on continue.

Ce genre de stratégie est appelée une stratégie interne (innermost en anglais), parce qu'elle réduit d'abord les redex les plus à l'intérieur des termes d'abord. La stratégie particulière présentée préfère de plus les redex de gauche à ceux de droite: elle ne réduit les redex de droite que quand il n'y en a plus à gauche. Il s'agit d'une stratégie interne gauche. Une

stratégie interne droite est évidemment envisageable, et se rapprocherait de la sémantique d'OCaml. Un petit test qui le montre est de taper:

```
let a = ref 0;;
let f x y = ();;
f (a:=1) (a:=2);;
a;;
```

qui crée une référence (une cellule de valeur modifiable par effet de bord), puis une fonction bidon f qui ignore ses deux arguments. Ensuite on calcule f appliquée à deux arguments qui affectent respectivement 1 et 2 à a.

Le langage étant séquentiel, la valeur finale de a permettra de savoir lequel des arguments a été évalué en dernier. En OCaml, la réponse est 1, montrant que c'est l'argument de gauche qui a été évalué en dernier.

Evidemment, en λ-calcul on n'a pas d'affectations, et le fait qu'une stratégie interne soit gauche ou droite ne change rien au résultat final.

1.5.2. Stratégies externes

On peut aussi décider d'utiliser une stratégie externe, qui réduit les redex les plus externes d'abord. Autrement dit, une stratégie interne réduit (λ x . u) v en normalisant λ x .u vers λ x. u0, en normalisant v vers v0, puis en réduisant u0[v0 / x], par contre une stratégie externe commence par contracter (λ x. u)v en u[v / x], qu'il réduit à son tour.

En fait, le λ-calcul pur est une sorte de langage-machine. Il est utilisable notamment pour modéliser des phénomènes de compilation. Mais il permet de simuler à bas niveau les types avec les opérations usuelles. Pour raisonner à plus haut niveau, on cherche à typer les λ -termes, ce qui permet d'ailleurs d'éliminer des termes comme Δ dont la sémantique n'a rien d'intuitif.

1.6. Logique combinatoire :

La logique combinatoire est une notation introduite par Moses Schönfinkel et Haskell Curry pour supprimer le besoin de variables en mathématiques, pour formaliser rigoureusement la notion de fonction et pour minimiser le nombre d'opérateurs nécessaires pour définir le calcul des prédicats à la suite de Henry M. Sheffer. Plus récemment elle a été utilisée en informatique comme modèle théorique de calcul et comme base pour la conception de langages de programmation fonctionnels.

Le concept de base de la logique combinatoire est celui de **combinateur** qui est une fonction d'ordre supérieur ; elle utilise uniquement l'application de fonctions et éventuellement d'autres combinateurs pour définir de nouvelles fonctions d'ordre supérieur. Elle a des liens très forts avec le lambda calcul et avec la logique intuitionniste grâce à la correspondance de Curry-Howard [17].

1.6.1. Réduction faible :

L'alphabet de la logique combinatoire (CL) se compose:

- Un ensemble de variables x, y, z, ...
- Des constantes **S**, **K**, **I** appelées combinateurs de base.

A partir des variables et des combinateurs de base **S**, **K** et **I** les termes de CL (CLtermes) sont définis inductivement par:

- **K**, **S**, **I** et chaque variable est un CL-terme
- si X et Y sont des CL-termes alors (XY) est un CL-terme.

Chaque occurrence d'un CL-terme de la forme **K**UV, **S**UVW et **I**U est appelée redex faible (w-redex). Contracter un CL-terme est l'action de remplacer dans ce CL-terme un redex **K**UV, **S**UVW ou **I**U respectivement par son contractum U, UV(VW) ou U. Une suite de contractions de ce type est appelée **réduction faible**. Si cette suite est finie et qu'elle permet à partir d'un CL-terme M d'obtenir un CL-terme N, nous dirons que M se réduit faiblement à N, et nous le notons

$$M \longrightarrow_{cw} N$$

Définition 1.8 : La réduction faible $\longrightarrow_{cw}$ est définie par les axiomes et règles suivants :

(S) $SXYZ \longrightarrow_{cw} XZ(YZ)$

(K) $KXY \longrightarrow_{cw} X$

(I) $IX \longrightarrow_{cw} X$

(ρ) $X \longrightarrow_{cw} X$

(μ) $X \longrightarrow_{cw} Y \Longrightarrow ZX \longrightarrow_{cw} ZY$

(ν) $X \longrightarrow_{cw} Y \Longrightarrow XZ \longrightarrow_{cw} YZ$

(τ) $X \longrightarrow_{cw} Y$ et $Y \longrightarrow_{cw} Z \Longrightarrow X \longrightarrow_{cw} Z$

On appellera longueur d'une réduction le nombre d'applications d'axiomes et de règles utilisés dans cette réduction.

Les CL-termes de la forme S, K, I, SUV, KU, SU sont appelés des termes fonctionnels (fnl).

Un CL-terme ne contenant aucun w-redex est dit sous forme normale faible (w-fn).

De même que pour le lambda-calcul, le théorème suivant entraine l'unicité de la forme normale faible, quand elle existe.

Théorème 1. Les propriétés de Church Rosser sont satisfaites pour CLw.

1.6.2. Abstraction :

On peut introduire dans CL l'analogue de l'abstraction de lambda-calcul. En effet, pour chaque CL-terme X on peut trouver un CL-terme X^*qui ne contient pas la variable x et tel que:

$$X^*x \rightarrow_{cw} X$$

Ces CL-termes X^* sont construits à partir des combinateurs S, K, I et du CL-terme X à l'aide d'algorithmes dits d'abstraction. On donne un exemple de deux algorithmes d'abstraction :

Algorithme d'abstraction ($abc_\beta f$)

a) $[x]_\beta M \equiv I$ si $M \equiv x$

b) $[x]_\beta M \equiv KM$ si x n'appartient pas à M

c) $[x]_\beta M\, x \equiv M$ si M fonctionnel et x n'appartient pas à M

f) $[x]_\beta M\, N \equiv S([x]_\beta M)\,([x]_\beta N)$ si non.

Algorithme d'abstraction ($abc_\eta f$)

a) $[x]_\eta M \equiv I$ si $M \equiv x$

b) $[x]_\eta M \equiv KM$ si x n'appartient pas à M

c) $[x]_\eta M\, x \equiv M$ si M fonctionnel et x n'appartient pas à M

f) $[x]_\eta M\, N \equiv S([x]_\beta M)\,([x]_\beta N)$ si non.

1.6.3. Cβη-réduction et extensionalité.

La notion de réduction forte a été introduite dans la logique combinatoire comme l'analogue de la λβη-réduction dans le lambda-calcul. L'équivalence entre les deux théories CL et λ-calcul a suggéré la définition de la Cβη-réduction. Cette dernière est obtenue en ajoutant dans la définition de la réduction faible la règle (ξη):

$$X \longrightarrow_{cw} Y \Longrightarrow [x]\,\eta X \longrightarrow_{cw} [x]\eta Y$$

Théorème 2 : Un CL-terme X est dit en cβη-forme normale ssi :

- X ≡ a où a est une variable quelconque ou
- X ≡ aX1...Xn où a est une variable et Xi est un CL-terme en cβη-forme normale (1<=i<= n) ou
- X ≡ [x]ηY où Y est un CL-terme en cβη-forme normale.

Théorème 3 : Les propriétés de Church Rosser sont satisfaites pour $\longrightarrow_{cw}$ et $\longrightarrow_{c\beta\eta}$.

1.7. La logique combinatoire illative :

La logique combinatoire et le lambda-calcul sont des théories analysent successivement la notion de calcul effective. Cependant les fondateurs originaux de ces deux théories Curry et Church ont toujours comme but de fournir une base pour la logique et les mathématiques. Malheureusement Kleene et Rosser (1935) ont montré que le lambda calcul et la logique combinatoire étendus sans restriction par les règles d'introduction et d'élimination d'implication sont inconsistants. Ce qui a conduit Curry à chercher de définir un nouveau système vérifiant la propriété de consistance appelé « la logique combinatoire illative (ICL). » La logique combinatoire illative c'est la logique combinatoire (ou lambda-calcul) étendue par des constantes supplémentaires et un ensemble d'axiomes et règles de dérivation [10].

1.7.1. Le système ICL *I* de Curry :

Nous présentons le système de la logique combinatoire illative *I*, c'est le premier et le plus simple système ICL.

I. L'ensemble des termes de système I est défini comme suit :

$$T = V \mid \Xi \mid T\ T \mid \lambda\ V.\ T$$

Avec V est l'ensemble des variables et Ξ est une constante. Alors l'ensemble des termes de système I c'est l'ensemble des termes de lambda-calcul pur étendu par la constante Ξ.

II. La βη-réduction est donnée par les règles de contraction suivantes :

$$(\lambda x.M)N \rightarrow M\ [N/x]$$
$$\lambda x.M\ x \rightarrow M \text{ si } x \notin FV(M)$$

III. Un jugement de système I est un élément de T et un contexte est un ensemble de termes.

IV. Soit Γ un contexte et X un jugement, on dit que X est dérivable à partir de Γ, noté $\Gamma \vdash X$, si $\Gamma \vdash X$ peut être produit par le système de déduction naturelle suivant :

(a) $X \in \Gamma \implies \Gamma \vdash X,$

(b) $\Gamma \vdash X, X = Y \implies \Gamma \vdash Y,$

(c) $\Gamma \vdash \Xi\ X\ Y, \Gamma \vdash X\ Z \implies \Gamma \vdash Y\ Z,$

(d) $\Gamma, Xx \vdash Yx, x \notin FV(\Gamma, X, Y) \implies \Gamma \vdash \Xi\ X\ Y.$

Remarque 1.9 :

a) La βη-réduction et la βη-convertibilité sont notées respectivement par ↠ et =.

b) Le terme XZ est interprété par 'Z ∈ X' ou 'Z satisfait le prédicat X'.

c) Le terme Ξ X Y est interprété par 'X ⊆ Y' ou ' (∀ x ∈ X) Y x'.

Définition 1.9 : Soient X, Y ∈ T, on peut écrire :

i) $X \supset Y \equiv \Xi\ (\boldsymbol{K}\ X)(\boldsymbol{K}\ Y)$,

ii) $\forall x \in X,\ Y \equiv \Xi\ X\ (\lambda\ u.\ Y).$

Les dérivations suivantes sont des conséquences directes de la définition du système *I*.

(a) $\Gamma \vdash X \supset Y, \Gamma \vdash X \implies \Gamma \vdash Y.$

(b) $\Gamma, X \vdash Y \Longrightarrow \Gamma \vdash X \supset Y$.

(c) $\Gamma \vdash \forall u \in X, Y, \Gamma \vdash X\ t \Longrightarrow \Gamma \vdash Y\ [t/u]$.

(d) $\Gamma, Xu \vdash Y, u \notin FV(\Gamma, X) \Longrightarrow \Gamma \vdash \forall u \in X, Y$.

Maintenant, il est possible d'interpréter les fragment { ⊃, ∀} de la logique des prédicats de premier ordre dans système *I*. Par exemple la formule « (∀ x ∈ A, Rx ⊃Rx) « est traduit dans le système *I* par « Ξ A (λ x. Ξ(**K** (R x))(**K** (R x))) » (avec K ≡ λ p q. p) et elle est prouvable dans le système *I*. Malheureusement cette interprétation n'est pas complète car le système *I* est inconsistant (c.-à-d. toute formule est dérivable dans le système *I* à partir d'un contexte vide).

Soient X et Y deux termes de système *I* tel que Y ≡ (λ y. (y y ⊃ X)) (λ y.(y y ⊃ X)), alors Y $=_\beta$ Y ⊃ X. Les règles de dérivation de système *I* montrent que ⊢X de la manière suivante :

(1) $Y \vdash Y$,

(2) $Y \vdash Y \supset X$, car $Y = Y \supset X$,

(3) $Y \vdash X$, d'après la règle (a),

(4) $\vdash Y \supset X$, d'après la règle (b),

(5) $\vdash Y$, car $Y \supset X = Y$,

(6) $\vdash X$, d'après (4) + (5) + (a).

1.7.2. Des systèmes ICL consistants :

Maintenant, nous présentons les quatre systèmes ICL ***I*P *I*F *I*G *I*Ξ** tel que '***I*P**' et '***I*F**' traitent la logique propositionnelle PROP et '***I*G**' et '***I*Ξ**' traitent la logique des prédicats PRED.

Soit T = Λ (**Ξ**, **L**) est un ensemble des λ-termes étendu par deux constantes **Ξ** et **L**

(a) . Les constantes **P, F, G, H** sont définies de la manière suivante :

- $\boldsymbol{P} \equiv \lambda x\, y\, .\, \boldsymbol{\Xi}(\boldsymbol{K}\ x)\ (\boldsymbol{K}\ y)$,
- $\boldsymbol{F} \equiv \lambda x\, y\, z.\, \boldsymbol{\Xi}\ x\ (y \circ z)$,
- $\boldsymbol{G} \equiv \lambda x\ y\, z\, .\, \boldsymbol{\Xi}\ x\ (\boldsymbol{S}\ y\ z)$,

- $\boldsymbol{H} \equiv \boldsymbol{L} \circ \boldsymbol{K}$

Avec **K** ≡ λ p q.p, M ∘ N ≡ λ x. M(N x) et **S** ≡ λ p q r. p r (q r).

Le terme **P** X Y est interprété comme' X ⊃ Y', **F** X Y est interprété comme 'Y^X, et **G** X Y est interprété comme' ∀ x Є X, Y x'.

Tous les quatre systèmes possèdent les deux règles de dérivation suivantes :

[1]. $X \in \Gamma \Longrightarrow \Gamma \vdash X$;
[2]. $\Gamma \vdash X, X =_{\beta\eta} Y \Longrightarrow \Gamma \vdash Y$.

Les règles de dérivation spécifiques au système *I* **P** sont :

(Pe) $\Gamma \vdash X \supset Y, \Gamma \vdash X \Longrightarrow \Gamma \vdash Y$;

(Pi) $\Gamma, X \vdash Y, \Gamma \vdash H X \Longrightarrow \Gamma \vdash X \supset Y$;

(PH) $\Gamma, X \vdash H Y, \Gamma \vdash H X \Longrightarrow \Gamma \vdash H(X \supset Y)$.

Les règles de dérivation spécifiques au système *I* **Ξ** sont :

(Ξe) $\Gamma \vdash \Xi X Y, \Gamma \vdash X V \Longrightarrow \Gamma \vdash Y V$;

(Ξi) $\Gamma, X x \vdash Y x, \Gamma \vdash L X, x \notin FV(\Gamma, X, Y) \Longrightarrow \Gamma \vdash \Xi X Y$;

(ΞH) $\Gamma, X x \vdash H(Y x), \Gamma \vdash L X, x \notin FV(\Gamma, X, Y) \Longrightarrow \Gamma \vdash H(\Xi X Y)$.

Les règles de dérivation spécifiques au système *I* **F** sont :

(Fe) $\Gamma \vdash \boldsymbol{F} X Y Z, \Gamma \vdash X V \Longrightarrow \Gamma \vdash Y (Z V)$;

(Fi) $\Gamma, X x \vdash Y(Z x), \Gamma \vdash \boldsymbol{L} X, x \notin FV(\Gamma, X, Y, Z) \Longrightarrow \Gamma \vdash \boldsymbol{F} X Y Z$;

(FL) $\Gamma, X x \vdash \boldsymbol{L} Y, \Gamma \vdash \boldsymbol{L} X, x \notin FV(\Gamma, X, Y) \Longrightarrow \Gamma \vdash \boldsymbol{L} (\boldsymbol{F} X Y)$.

Les règles de dérivation spécifique au système *I* **G** sont :

(Ge) $\Gamma \vdash \boldsymbol{G} X Y Z, \Gamma \vdash X V \Longrightarrow \Gamma \vdash Y V (Z V)$;

(G_i) $\Gamma, X\,x \vdash Y\,x\,(Z\,x),\ \Gamma \vdash \mathbf{L}\,X, x \notin FV(\Gamma, X, Y, Z) \implies \Gamma \vdash \mathbf{G}\,X\,Y\,Z$;

(G_L) $\Gamma, X\,x \vdash \mathbf{L}\,(Y\,x),\ \Gamma \vdash \mathbf{L}\,X, x \notin FV(\Gamma, X, Y) \implies \Gamma \vdash \mathbf{L}\,(\mathbf{G}\,X\,Y)$.

CHAPITRE II

L'ASSISTANT DE PREUVE COQ

Introduction

Le développement de programmes sûrs exempts d'erreurs offre aujourd'hui diverses possibilités d'applications dans le monde industriel. Les méthodes formelles de validation de programmes ont constituées des approches appropriées pour ce genre de problème. Une des approches qui se révèle aussi fructueuse est celle de considérer l'activité de programmer comme celle de prouver des propriétés. La garantie de sûreté serait donc assurée par la preuve mathématique de la spécification. La théorie des types, rend possible l'objectif de développer des programmes à partir de leur preuves [21].

Pour cela, il serait intéressant de présenter un aperçu sur la logique constructive et les concepts de base utilisés dans la mécanisation d'un processus de déduction. Nous nous intéressons particulièrement à la déduction naturelle et le système de preuve Coq. Ainsi on va donner une idée sur l'interprétation des λ-termes comme les preuves et les type assignés à ces termes comme les formules de la théorie. Cette idée est connue sous le nom « Curry-Howard isomorphism ».

2.1. Logique constructive et Déduction naturelle :

L'objectif de la logique constructive (appelée souvent logique **intuitionniste**) n'est pas celui de savoir si une telle proposition est vraie ou non, comme dans la logique classique, mais son intérêt est de savoir si une telle proposition possède ou non une preuve.

La valeur de vérité du calcul propositionnel dans la logique classique peut être définie à l'aide des tables de vérité. Dans la logique constructive nous avons besoin d'introduire une sorte de calcul pour les preuves.

Dans la logique intuitionniste la loi du tiers exclu n'est pas valable, ainsi certaines démonstrations qui sont valables en logique classique ne le sont pas en logique intuitionniste. D'une manière générale :

- Toute proposition vraie en logique intuitionniste est aussi vraie dans la logique classique.
- Toute proposition vraie en logique classique peut être vraie en logique intuitionniste en ajoutant l'axiome du tiers exclu.

2.1.1. La sémantique de Heyting et Kolmogorov

Pour donner un sens à cette approche dite constructive, une nouvelle réinterprétation des connecteurs logiques doivent être nécessairement introduite.

Pour définir le connecteur de négation on introduit dans le langage l'objet ⊥ qui signifie l'absurde (par exemple 0 = 1). On pose par définition : ¬ A ≡ A → ⊥.

- Une preuve de $A \wedge B$ consiste en preuve de *A* et une preuve de *B*. C'est donc une paire formée de la preuve de *A* et de la preuve de *B*.
- Une preuve de $A \vee B$ consiste en une preuve de *A* ou en une preuve de *B*.
- Une preuve de $A \rightarrow B$ est une construction c qui transforme toute preuve de *A* en une preuve de *B*. Si d prouve *A* alors (c d) prouve *B*.
- Une démonstration de $\forall x \in A : P(x)$ est une fonction qui prend en argument un élément quelconque de A et rend une preuve de $P(x)$. On voit ici le paradigme "les preuves sont des objets" (proof as object).
- Une preuve de $\exists x \in A : P(x)$ est une paire $< a, h >$, où a est un élément particulier (Le témoin) de A et h une preuve de P (a). Comme pour la disjonction, c'est la notion intuitionniste de l'existentielle qui est exprimée ici : pour prouver une formule existentielle, il faut exhiber un témoin.
- Une preuve de : $\neg A$ est donc une construction qui transforme une preuve *p* de *A* en une preuve de $\perp$.

2.1.2. Déduction naturelle :

Les systèmes de déduction naturelle interviennent pour remédier aux défauts des systèmes de Hilbert[5] et offrent une stratégie, à travers les règles de déduction, permettant la manipulation de l'ensemble d'hypothèses. Le mécanisme de base de la déduction naturelle peut être présenté sous des formats différents mais tous sont équivalents.

2.1.2.1. Contextes et jugements :

Un contexte est une liste de formules.

Un jugement est une expression de la forme :

$$\Gamma \vdash A$$

- Γ est le contexte d'hypothèses.
- $\vdash$ est le symbole de déduction.
- A est une formule qui correspond à la conclusion du jugement.

2.1.2.2. Règles de la déduction naturelle :

Le système utilisé pour démontrer une formule à partir d'autres formules est appelé système de dérivation. Un système de dérivation est un système contenant des règles d'inférence sur des termes d'un langage, représentées par des déductions d'un ou plusieurs jugements vers un autre. Les règles, pour la logique du premier ordre, sont classées selon la symétrie introduction/élimination (figure 2.1).

L'application de ces règles en série nous permet de dériver des formules et donc faire des preuves.

Par exemple, si (π) est une dérivation qui prouve la proposition A, et (π') est une dérivation qui prouve la proposition B, alors la dérivation suivante prouve la conjonction $A \wedge B$:

[5] Sont les systèmes permettant la recherche de théorèmes. Une preuve en format de Hilbert est une suite de propositions, qui sont des axiomes ou des conséquences de propositions précédentes, déduites par des règles de preuve bien définies.

$$\frac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}\ (\wedge I)$$

La règle d'introduction du connecteur de conjonction ($\wedge I$) permet de construire, à partir de deux preuves des formules A et B, la preuve de la conjonction de ces derniers. Dans l'autre sens les règles ($\wedge E_1$) et ($\wedge E_2$) permettent de prouver une des sous formules d'une conjonction déjà prouvée.

Introduction	**Elimination**
$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B}\ (\wedge I)$	$\dfrac{\vdots}{A \wedge B}\ (\wedge E_1) \qquad \dfrac{\vdots}{A \wedge B}\ (\wedge E_2)$ $A \qquad\qquad B$
$\dfrac{\begin{array}{c}[A]_i \\ \vdots \\ B\end{array}}{A \rightarrow B}\ (\rightarrow I)$	$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & A \rightarrow B\end{array}}{B}\ (\rightarrow E)$
$\dfrac{\vdots}{A}\ (\vee I_1) \qquad \dfrac{\vdots}{B}\ (\vee I_2)$ $A \vee B \qquad\qquad A \vee B$	$\dfrac{\begin{array}{ccc} & [A]i & [B]i \\ \vdots & \vdots & \vdots \\ A \vee B & C & C\end{array}}{C}\ (\vee E)$
$\dfrac{\begin{array}{c}[A]_i \\ \vdots \\ B \wedge \neg B\end{array}}{\neg A}\ (\neg I)$	$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & \neg A\end{array}}{B}\ (\neg E)$

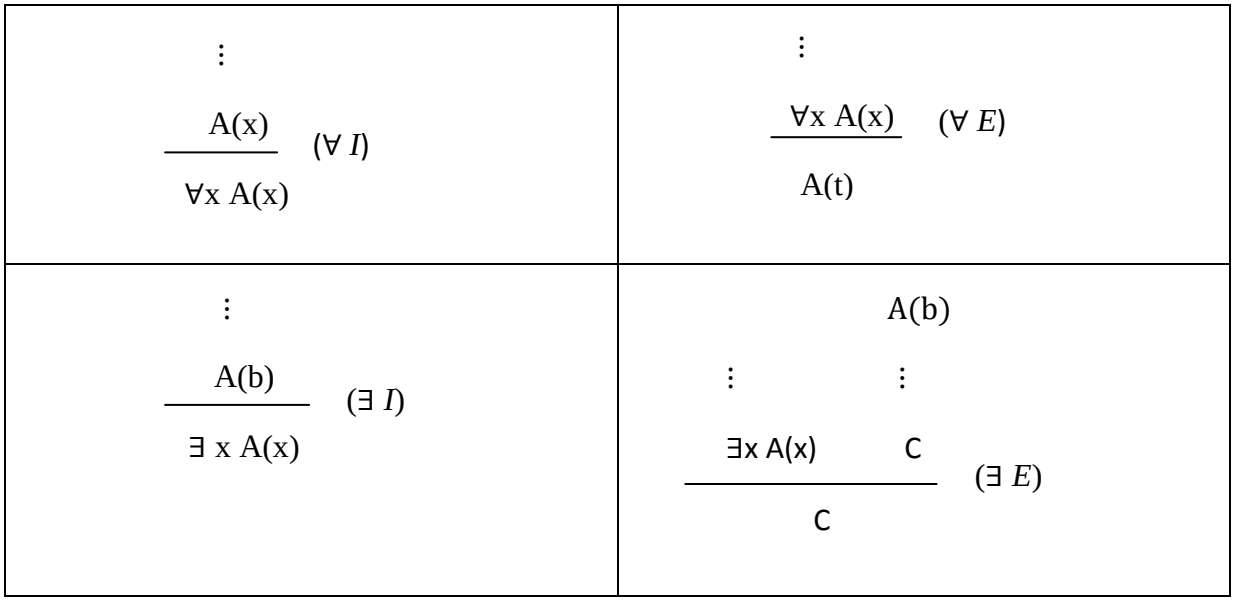


Figure 2-1 : Règles de la déduction naturelle

La règle d'élimination de la négation a comme conclusion une formule arbitraire : à partir d'une contradiction, où on a les preuves d'une formule et de sa négation, nous pouvons déduire n'importe quelle proposition. Les autres règles s'expliquent d'elle-même, sauf celle qui appellent la notation $[\ ]_i$ ($\rightarrow I$), ($\neg I$) et ($\vee E$). Les crochets représentent une hypothèse qui a été déchargée. Par exemple la règle d'introduction de l'implication ($\rightarrow I$) qui peut être interprété par :

Si (π) est une preuve de B utilisant A comme hypothèse, soit si :

A

⋮

(π)

⋮

B

Alors nous pouvons déduire à partir de cette preuve une preuve de $A \rightarrow B$ sans utiliser A comme hypothèse auxiliaire. La notation utilisée pour décharger A est $[A]_i$, ou i est une étiquette unique (disons un entier) dont nous nous servons pour associer l'inférence de $A \rightarrow B$ avec les occurrences de A que nous déchargeons. Déchargeons A ci-dessus :

$$\dfrac{\begin{array}{c}[A]_1\\ \vdots\\ (\pi)\\ \vdots\\ B\end{array}}{A \to B}\ 1(\to I)$$

2.1.2.3. Formalisation du processus de dérivation :

Une présentation plus formelle de la déduction naturelle, qui montre à la fois les formules et l'ensemble des hypothèses auxiliaires, est la déduction naturelle en format des séquents. Avec cette représentation, au lieu de dériver juste des formules, nous dérivons des séquents de la forme suivante :

$$\Gamma \vdash A$$

Où Γ est un ensemble de formules appelé le *contexte du séquent* (l'ensemble des hypothèses auxiliaires courantes) et le but est de dériver la formule A.

Définition 2.1 :

Une dérivation en déduction naturelle est un arbre inversé dans les nœuds sont les séquents connectés par des règles de déduction de la figure 2-2. Les feuilles sont des instances de l'axiome (Ax).

- Une *preuve* P d'un jugement : $\Gamma \vdash A$ est une dérivation en déduction naturelle dont la racine est décorée par $\Gamma \vdash A$.
- Nous disons aussi que P est une preuve de A *à partir* de Γ.
- S'il existe une preuve de $\Gamma \vdash A$, nous disons que $\Gamma \vdash A$ est *prouvable*, ou bien que la proposition A est prouvable à partir de l'environnement Γ.
- Un *théorème* est une formule prouvable à partir d'un ensemble vide d'hypothèses.

$\dfrac{}{\Gamma, A \vdash A}$ (Ax)	
Introduction	**Elimination**
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$ $(\wedge I)$	$\dfrac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_i \;\; i = 1,2}$ $(\wedge E)$
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$ $(\rightarrow I)$	$\dfrac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$ $(\rightarrow E)$
$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$ $(\vee I)$ $\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$ $(\vee I)$	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$ $(\vee E)$
$\dfrac{\Gamma, A \vdash B \quad \Gamma, A \vdash \neg B}{\Gamma \vdash \neg A}$ $(\neg I)$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash B}$ $(\neg E)$
$\dfrac{\Gamma \vdash A(x)}{\Gamma \vdash \forall x\, A(x)}$ $(\forall I)$	$\dfrac{\Gamma \vdash \forall x\, A(x)}{\Gamma \vdash A(t)}$ $(\forall E)$
$\dfrac{\Gamma \vdash A(t)}{\Gamma \vdash \exists x\, A(x)}$ $(\exists I)$	$\dfrac{\Gamma \vdash \exists x\, A(x) \quad \Gamma, A(t) \vdash C}{\Gamma \vdash C}$ $(\exists E)$

Figure 2-2 : Déduction naturelle en format des séquents

2.2. Théorie des types et l'isomorphisme de Curry-Howard :

Aujourd'hui, on désigne généralement par « théorie des types » un formalisme logique dont les objets sont des λ-termes typés. Il existe plusieurs formalismes rentrant dans cette catégorie, et ils se distinguent essentiellement par le système de types plus ou moins riche des objets, ainsi que par la logique pour parler de ces objets. On peut citer en particulier la logique d'ordre supérieur de Church, la Théorie des Types de Martin-Löf, la logique du système PVS et le Calcul des Constructions Inductives (CCI)

Les objets de la logique d'ordre supérieur de Church sont les λ-termes simplement typés.
Les trois autres formalismes utilisent essentiellement des variantes (ou plutôt des fragments) du langage de programmation ML. Ils se distinguent par leur logique:

- PVS utilise un calcul des prédicats classique, sous forme de calcul des séquents; ses objets sont un fragment fortement terminant de ML, enrichi par une notion de sous-type. Les preuves en revanche ne sont pas un objet du formalisme.

- La théorie des types de Martin-Löf est une logique prédicative. Les preuves sont elles-mêmes des λ-termes et l'accent est mis sur la constructivité.

- Le Calcul des Constructions Inductives est lui une extension de la logique d'ordre supérieur et autorise la quantification imprédicative sur toutes les propositions. Il est à la base du système d'assistance à la preuve Coq développé par l'INRIA [11]. Ce système offre entre autre la possibilité d'extraire d'une preuve constructive en Coq la fonction correspondante en OCaml. Il extrait conjointement les types et les fonctions auxiliaires dont a besoin la fonction. dont les variantes implantées par Coq seront l'objet.

Un point commun essentiel de CCI et de la théorie de Martin-Löf est bien sûr qu'ils sont construits sur l'isomorphisme de Curry-Howard.

Ainsi l'isomorphisme de Curry-Howard[6] montre que les démonstrations et les algorithmes sont deux facettes d'un même concept et qu'ils peuvent se représenter de manière uniforme par

[6] L'isomorphisme de Curry-Howard connu souvent sous le paradigme « type as formula ».

des λ-termes. Pour illustrer cela, nous allons prendre trois règles de déduction logique issues du système de déduction naturelle et les trois règles de typage de lambda calcul simplement typé correspondantes. Nous constaterons alors la correspondance existant entre les deux formalismes.

Déduction naturelle	lambda calcul simplement typé
$\dfrac{}{\Gamma, A \vdash A}$ (Ax)	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$ *(Var)*
$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$ $(\to E)$	$\dfrac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B}$ *(App)*
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$ $(\to I)$	$\dfrac{\Gamma \cup \{x : A\} \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$ *(ABS)*

Figure 2-3: L'isomorphisme de Curry Howard

Il est facile de constater la correspondance syntaxique entre les règles de la première et de la deuxième colonne. Les propriétés dans la première colonne correspondent exactement aux types dans la deuxième colonne. Cette correspondance va même plus loin. La β-réduction dans un λ-terme correspond à l'opération d'élimination des coupures dans une déduction.

2.3. Les systèmes de preuve

Ces dernières années, le domaine de recherche consacré à l'automatisation des preuves sur ordinateur a connu un développement important. Ceci a donné comme résultat l'apparition de plusieurs systèmes de preuves qui peuvent être classés en trois catégories :

1. Les vérificateurs de preuve (proof-checkers) : dans ces systèmes la preuve est construite par un utilisateur et sa validité est vérifiée par la machine. Automath [20]et Mizar [21] sont les systèmes les plus répandus dans cette classe de vérificateurs.

2. Les démonstrateurs Automatiques : la preuve est effectuée par la machine sans intervention du programmeur. Leurs domaines d'application sont réduits.

3. Les assistants de preuve sont des systèmes interactifs dans lesquels l'utilisateur guide la démonstration à l'aide des fonctions appelées les tactiques dont les parties faciles sont démontrées automatiquement. Parmi les premiers systèmes d'aide à la preuve le système LCF (Logic for Computable Function) [22] de Robin Milner qui introduisit les notions des tactiques. Pour écrire ces tactiques, Milner introduisit un meta-langage qui évolua par la suite pour donner naissance aux langages de la famille ML. Les autres systèmes comme HOL [5], PVS [7], Isabelle [29], Lego [26], Alf [25] et Coq sont basés sur les mêmes concepts que ceux de LCF.

2.4. L'assistant de preuve Coq

Coq est un assistant de preuves permettant l'expression de spécifications et le développement de programmes cohérents avec leur spécification. La première implémentation date de 1984 et fut réalisée en CAML par Gérard Huet et Thierry Coquand.
Cet outil s'applique alors parfaitement au développement de programmes en lesquels une confiance absolue est requise : télécommunications, sécurité des transports, énergie, cryptologie, etc. Dans ces domaines, le besoin de programmes rigoureusement conformes à leur spécification justifie l'effort demandé par leur validation [26].

L'intérêt de Coq ne se limite pas au développement de programmes sûrs.

Coq est avant tout un système permettant de faire des preuves dans une logique très expressive, dite d'ordre supérieur. Ces preuves sont construites de façon interactive, assistée par des outils de recherche automatique de preuves dans des domaines ou cela est possible.

Les domaines d'utilisation de Coq sont très varies : logique, automates, syntaxe et sémantique des langues naturelles, algorithmique, etc.

On peut aussi le considérer comme un cadre logique permettant l'axiomatisation de logiques et le développement interactif de preuves dans ces logiques. Citons par exemple l'implémentation de systèmes de raisonnement dans des logiques modales, temporelles, logiques de ressources et les systèmes de raisonnement sur les programmes impératifs.

Coq fait partie d'une longue tradition de systèmes informatiques d'aide à la démonstration de théorèmes. Citons par exemple les systèmes Automath , Nqthm , Isabelle, Lego, HOL, PVS …etc

Un des points les plus remarquables de Coq est la possibilité de synthétiser des programmes certifiés à partir de preuves, et, depuis peu, des modules certifiés.

2.4.1. Le langage de spécification Gallina

Gallina est le langage dans lequel sont écrit les termes repose sur le CIC [16]. Dans ce calcul tous les objets ont un type. Il y a des types pour les fonctions (ou les programmes), les données, les preuves et les types eux-mêmes possèdent un type. Par exemple, on ne permet pas la déclaration "pour tout x, P", on doit dire au lieu de cela : "Pour tout *x* appartenant à *T, P*". L'expression " *x* appartenant à *T*" est écrite *"x:T".* On dit aussi : "x de type *T*".

Pour manipuler l'environnement Coq, on dispose d'un ensemble de commandes qui permettent la spécification des différents objets. En général, le langage de commande de Gallina est défini par la syntaxe décrite par la figure 2-4.

Sentence ::= *declaration*
| *definition*
| *statement*
| *inductive*
| *fixpoint*
| *statement proof*

Params ::= *typed_idents* ; ... ; *typed_idents*

declaration ::= Axiom *ident* : *term* .
| *declaration_keyword params* .

declaration_keyword ::= Parameter | Parameters
| Variable | Variables
| Hypothesis | Hypotheses

Definition ::= Definition *ident [*: *term]* := *term* .
| Local *ident [*: *term]* := *term* .

Inductive ::= *[*Mutual*]* Inductive *ind_body* with ... with *ind_body* .
| *[*Mutual*]* CoInductive *ind_body* with ... with *ind_body* .

ind_body ::= *ident [*[*params*]*]* : *term* := *[constructor* | ... | *constructor]*

constructor ::= *ident* : *term*

fixpoint ::= Fixpoint *fix_body* with ... with *fix_body* .
| CoFixpoint *cofix_body* with ... with *cofix_body* .

statement ::= Theorem *ident* : *term* .
| Lemma *ident* : *term* .
| Definition *ident* : *term* .

proof ::= Proof . ... Qed .
| Proof . ... Defined .

Figure 2-4 Langage de commande du Gallina

2.4.1.1. Les sortes

Les types sont vus comme des termes du langage et alors devraient appartenir à un autre type. Le type d'un type est toujours une constante du langage appelé une sorte. Les deux sortes de base dans le langage du CIC sont *Prop* et *Set*. *Prop* est le type des propositions logiques et *Set* est le type des spécifications qui inclut les programmes et les ensembles usuels comme les booléens, les naturels, et les listes etc.

Ces sortes eux-mêmes peuvent être manipulées comme des termes ordinaires. Par conséquent on devrait donner aussi un type aux sortes car la supposition que *Set* de type *Set* mène à une théorie inconsistante. Il y'a en plus de *Set* et *Prop* une hiérarchie des *Type(i)* pour n'importe quel nombre entier *i*. Nous appelons S l'ensemble des sortes défini par *:*

$S=\{Prop, Set, Type(i) / i \in \mathbb{N}\}$.

Les sortes ont les propriétés suivantes: Prop:Type(0) et Type(*i*):Type(*i*+1).

Exemple 2.1 :

1. Prop a un type Type :

   Coq < Check Prop.

   Prop: Type

2. Set est de type Type *:*

   Coq < Check Set.

   Set : Type

3. Type est de type Type :

   Coq < Check Type.

   Type : Type

2.4.1.2. Les constantes

En plus des sortes, les autres constantes du langage sont :

- Les constantes définies dans l'environnement.
- Les constantes correspondant aux définitions inductives (constructeur, destructeur).

2.4.1.3. Les termes

La notion de terme constitue une catégorie syntaxique très générale dans le langage de spécification Gallina[16]. Elle correspond à la notion intuitive d'expression bien formée prenant en compte les règles de construction du langage.

Un terme est un type ou une variable ou une constante de l'environnement. Comme d'habitude dans le λ-calcul, nous combinons des objets en utilisant l'abstraction et

l'application. Plus précisément le langage du Calcul des Constructions Inductives est construit à partir des règles suivantes :

1. Les sorts Set, Prop et Type sont des termes.
2. les constantes de l'environnement sont des termes.
3. Les variables sont des termes.
4. Si *x* est une variable et *T, U* des termes alors *(x:T) U* est un terme. Si *x* se produit dans *U*, *(x:T) U* est lu "pour tout *x* de type *T, U*". Si *U* dépend de *x*, on dit que *(x:T) U* est un produit dépendant. Si *x* n'appartient pas à *U* alors *(x:T) U* est lu "si *T* alors *U*".Un produit non dépendant peut être écrit: $T \rightarrow U$.
5. Si *x* est une variable et *T, U* sont des termes alors *U [T/x]* est un terme. C'est une notation pour β -abstraction.
6. Si *T* et *U* sont des termes alors *(T U)* est un terme. Le terme *(T U)* se lit "*T* appliqué à *U*".

La table suivante représente la syntaxe des termes dans le système Coq :

term ::= *ident*
| *sort*
| *term* -> *term*
| (*typed_idents* ; ... ; *typed_idents*) *term*
| [*idents* ; ... ; *idents*] *term*
| (*term* ... *term*)
| *[annotation]* Cases *term* of *[equation* | ... | *equation]* end
| Fix *ident* { *fix_body* with ... with *fix_body* }
| CoFix *ident* { *cofix_body* with ... with *cofix_body* }
sort ::= *Prop*
| *Set*
| *Type*
annotation ::= < *term*>
typed_idents ::= *ident* , ... , *ident* : *term*
idents ::= *ident* , ... , *ident [*: *term]*
fix_body ::= *ident* [*typed_idents* ; ... ; *typed_idents*]: *term* := *term*
cofix_body ::= *ident* : *term* := *term*
simple_pattern ::= *ident*
| (*ident* ... *ident*)
equation ::= *simple_pattern* => *term*

Figure2-5 Syntaxe des termes dans le système Coq

2.4.1.4. Le contexte

Un terme bien typé dépend d'un ensemble de déclarations et de variables appelé contexte.

Un contexte Γ écrit $[x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n]$ où les x_i sont des variables distinctes et les T_i sont des termes.

2.4.1.5. L'environnement

Parce que nous manipulons les constantes, nous devons aussi considérer un environnement E, qui est un ensemble de définitions des constantes.

2.4.1.6. Notations

Méta-variables : nous emploierons les symboles E et Γ pour désigner respectivement un environnement et un contexte arbitraire.

Le contexte vide : nous utilisons la notation [ ] pour dénoter un contexte ne déclarant aucune variable locale. C'est en particulier le contexte courant associé à une partie de développement à l'extérieur de toute section.

Déclarations : la déclaration spécifiant que l'identificateur v a pour type A se note $(v : A)$.

Suite de déclarations : un contexte peut se présenter sous la forme d'une suite de déclarations, présentée comme $[v_1 : A_1 , v_2 : A_2 , \ldots , v_n : A_n]$.

L'adjonction d'une déclaration $(v : A)$ à un contexte se note $:: (v : A)$.

Présence d'une déclaration : pour exprimer que la variable v est spécifiée de type A dans le contexte Γ, on utilise la notation $(v : A) \in \Gamma$. Des variantes sont également utilisées : $v \in \Gamma$ (v est déclarée dans Γ sans plus de précision), $(v : A) \in \Gamma \cup E$ (déclaration soit locale, soit globale) etc.

Jugement de typage : la notation $E, \Gamma \vdash t : A$, où E, Γ, t et A dénotent respectivement un environnement, un contexte, un terme et un type et se lit « Dans l'environnement E et le contexte Γ le terme t a pour type A ».

Définition 2.2 (Types habités)

On dira qu'un type A est habité dans un environnement E et un contexte Γ s'il existe un terme t pour lequel on a le jugement $E, \Gamma \vdash t : A$.

2.4.2. Les types atomiques et les types composés

Les types peuvent être classés en deux catégories : les types atomiques et les types composés.

a) types atomiques : réduits à un seul identificateur comme nat, Z et bool.

b) types composés (flèches) : des types de la forme (A→B) où A et B sont deux types. Ces types forment un cas particulier d'une construction appelée produit dépendant.

2.4.3. Mécanisme des sections de Coq

Les sections définissent un mécanisme de blocs similaire à celui de nombreux langages de programmation (C, Java, Pascal, etc.) permettant la déclaration et la définition de variables locales et contrôlant leur portée.

En Coq, les sections sont nommées, et les commandes de début et de fin de section sont respectivement " Section id " et " End id ", ou id est le nom choisi pour la section. Naturellement, les sections peuvent être imbriquées, et les ouvertures/fermetures de sections doivent respecter une discipline de système de parenthèses.

Exemple 2.2 :

Section binomial_def.
Variables a b:Z.
*Definition binomial z:Z := a*z + b.*
Section trinomial_def.
Variable c : Z.
*Definition trinomial z:Z := (binomial z)*z + c.*
End trinomial_def.
End binomial_def.

2.4.4. Les règles de typage

Les règles pour présenter des nouveaux objets dans l'environnement :

2.4.4.1. Expressions réduites à un identificateur

La forme syntaxique la plus simple d'expression est une simple variable ou une constante *x*. Un tel terme ne peut être accepté que si *x* est déclarée dans le contexte ou l'environnement courant.

Soit *A* le type de *x* dans cette déclaration, alors le terme réduit à *x* a pour type *A*.

Il est d'usage de présenter une règle de typage sous forme d'une règle d'inférence; les prémisses sont placées au dessus d'une barre horizontale, et la conclusion en dessous de cette barre.

$$\text{Var}\ \frac{(x,A) \in E \cup \Gamma}{E\,[\Gamma] \vdash x : A}$$

Cette règle se lit : « si l'identificateur x est spécifié de type A dans l'environnement E ou le contexte Γ, alors le terme x a pour type A dans cet environnement et ce contexte ».

Exemple 2.3 :
- Les constantes *O : nat* et *true : bool*

Check O.
O: nat

Coq < Check true .
true
: bool

2.4.4.2. Application

L'application d'une fonction à un argument est la principale structure de contrôle de notre langage. Nous en présentons d'abord la syntaxe, accompagnée de règles assurant la cohérence des expressions ainsi formées.

Soient un environnement E et un contexte Γ ; soient deux expressions e_1 et e_2 de types respectifs $A \rightarrow B$ et A dans $E \cup \Gamma$; alors l'application de e_1 à e_2 est le terme s'écrivant « e_1 e_2 » ; ce terme est de type B dans le contexte et l'environnement considérés.

Dans l'expression « e_1 e_2 », e_1 est en position fonctionnelle et e_2 est l'argument de l'application. La présentation sous forme de règle d'inférence est la suivante :

$$App\ \frac{E, \Gamma \vdash e_1 : A \rightarrow B \quad E, \vdash e_2 : B}{E, \vdash (e_1\, e_2) : B}$$

Exemple 2.4 :

Check negb.
negb : bool→ bool

Check (negb true).
negb true : bool

Check (negb (negb true)).
negb (negb true) : bool

2.4.4.3. Abstraction

La λ-abstraction (ou plus simplement l'abstraction) est un moyen simple de construire des fonctions en associant un paramètre formel et une expression. La fonction qui a tout v de type A associe l'expression e se note « fun (v:A) ⇒ e ». Par exemple, la fonction qui a n de type nat associe son cube n^3 se note :

fun n:nat ⇒ (n*n*n)%nat

La règle de typage de l'abstraction est donnée ci-dessous :

$$\text{Abs}\quad \frac{E,\ (v : A) \vdash e : B}{E, \Gamma \vdash \text{fun } v{:}A \Rightarrow e : A \rightarrow B}$$

Example 1.5:

*Check fun n:nat => (n*n*n)%nat*

*fun n : nat => n * n * n*
: nat -> nat

2.4.5. Produit dépendant

Le produit dépendant nous permet de travailler avec des fonctions dont le type du résultat dépend de celui de leur argument. Il est un type de la forme « ∀ *v:T, U* » où *T* et *U* sont des types et *v* est une variable liée dont la portée est le type *U*. La variable *v* peut avoir des occurrences libres dans le type *U*. Le produit dépendant se lit pour tout *v* de type *T, U.*
On pense bien sûr au polymorphisme de certains langages de programmation, et surtout au polymorphisme paramétrique des langages de la famille ML.

Le produit dépendant permet aussi d'exprimer la quantification universelle, tant sur des expressions que sur des types ; citons par exemple le théorème affirmant que la relation ≤ est réflexive sur N, ainsi que la commutativité de la disjonction :

∀ n : nat. n <= n

∀P,Q : Prop. P ∨ Q→ Q ∨ P

2.4.6. Règles de conversion

Les conversions utilisées dans Coq sont de quatre sortes :

2.4.6.1. La δ-réduction (prononcez delta-réduction)

Permet de remplacer un identificateur par sa définition : soit t un terme, et v un identificateur défini par t' dans l'environnement ou le contexte courant ; alors la δ-conversion transforme le terme t en t {t' / v}.

Dans les exemples suivants, nous utilisons la δ-conversion sur les constantes Zsqr et my_fun. On notera que les arguments de la commande Eval précisent qu'on utilise une stratégie d'appel par valeur (cbv). Le mot clef delta peut-être suivi d'une liste d'identificateurs, auquel cas les δ- réductions se limitent aux identificateurs présents dans cette liste.

*Coq< Definition Zsqr (z:Z) : Z := z*z.*
Zsqr is defined.
Coq< Definition my_fun (f:Z→ Z)(z:Z) : Z := f (f z).
My_fun is defined.
Coq<Eval cbv delta [my_fun Zsqr] in (my_fun Zsqr).
= (fun (f:Z→Z)(z:Z) => f (f z))(fun z:Z => z* z)
: Z→Z
Coq< Eval cbv delta [my_fun] in (my_fun Zsqr).
= (fun (f:Z→Z)(z:Z) ⇒ f (f z)) Zsqr
: Z→Z

2.4.6.2. La β-réduction (prononcez beta-réduction) permet de transformer un

β-radical[7] c'est à dire un terme de la forme « (fun v:A=> t) u » en le terme t {u/v}.

coq< Eval cbv beta delta [my_fun Zsqr] in (my_fun Zsqr).
*= fun z:Z => z*z*(z*z) : Z→ Z*

2.4.6.3. La ξ-réduction (prononcez zeta-réduction) consiste en l'élimination des liaisons locales « let-in » ; plus précisément, elle remplace une expression de la forme " let v:=u in t " par t{u/v}.

[7] β-redex en anglais.

L'expérimentation ci-dessous montre le résultat de l'évaluation avec ou sans ξ -conversion :

Coq< Variables a b:Z.
a is assumed
b is assumed
coq< Let s:Z := a+b.
s is defined
coq< Let d:Z := a-b.
d is defined
*coq< Definition h : Z := s*s + d*d.*
h is defined
coq< Eval cbv beta delta [h] in (h 56 78).
*= let s := 56+78 in let d := 56-78 in s*s + d*d*
: Z
Coq< Eval cbv beta zeta delta [h] in (h 56 78).
= (56+78)(56+78)+(56-78)*(56-78)*
:Z

2.4.6.4. La ι-réduction (prononcez iota-réduction)

La ι-réduction est responsable de certaines simplifications liées aux schémas de programmes récursifs, par exemple les réductions de « plus O n » en n, de « mult O p » en O et de « mult (S n) p » en « plus p (mult n p) ».

2.4.7. Les définitions et les déclarations :

Comme dans la plupart des langages de programmation, il est d'usage de distinguer les notions de définition et déclaration, ainsi que leur portée locale ou globale.

Une déclaration permet d'attacher un type à un identificateur sans lui donner de valeur. Comme dans les fichiers d'interface de C, Java ou ML, on peut déclarer par exemple la variable *max_nat* est de type *nat* , sans lui attacher de valeur précise.

Coq< Variable max_nat : nat.
max_nat is assumed

En revanche, une définition donne une valeur à un identificateur sous la forme d'un terme associé. Dans la mesure où l'on peut déterminer le type de ce terme, une définition précise à la fois le type et la valeur d'un identificateur.

```
Coq< Definition Double (n: nat) := (plus n n).
Double is defined
 Coq< Check Double.
Double
: nat->nat
```

Par conséquent, toute définition peut aussi jouer le rôle d'une déclaration, et un énoncé concernant une déclaration quelconque s'applique donc à une définition en oubliant la valeur qu'elle précise.

La portée d'une déclaration ou d'une définition peut être soit globale, c'est à dire tout le reste du développement, soit locale, c'est à dire restreinte à une sous-expression ou à une section (nom donné en Coq à un mécanisme similaire aux blocs des langages de programmation). À tout point d'un développement en Coq sont associés à la fois un environnement et un contexte dits courants.

2.4.8. Structures de données inductives

La définition de types inductifs en Gallina étend les différentes notions de définitions de types fournies dans les langages de programmation. On peut les comparer aux définitions de types récursifs dans les langages fonctionnels ML, Ocaml, Haskell. Mais la possibilité de mélanger types récursifs et produits dépendants rend les types inductifs de Gallina beaucoup plus précis et expressifs.

À chaque type de données inductif correspond une structure de calcul, basée sur le filtrage et la récursion. Ces structures de calcul sont le noyau de la programmation récursive en Gallina.

2.4.8.1. Types sans récursion

Les types inductifs les plus simples sont les types énumérés, utilisés pour décrire des ensembles finis. L'exemple le plus fréquemment utilisé d'un tel ensemble fini est celui des valeurs booléennes, qui contient seulement deux éléments. On peut former la définition de ce type comme suit :

Coq< Inductive bool : Set := true : bool
| false : bool .

2.4.8.2. Types avec récursion

Les types inductifs sans récursion permettent de décrire toute sorte de données, mais toujours des données dont la taille est connue à l'avance. Il est nécessaire de pouvoir raisonner sur des structures de données dont la taille peut varier, par exemple des tableaux de données de taille non définie à l'avance.

La récursion fournit une solution extrêmement simple. On exprime que certaines données comportent des fragments qui sont de même nature que ces données elles-mêmes. Par exemple, si l'on considère les arbres binaires, ils peuvent être des feuilles ou des arbres composés. Si ce sont des arbres composés, alors ils contiennent deux fragments qui sont eux aussi des arbres binaires, donc des objets de même nature que l'arbre lui-même.

Ces types de données représentent des ensembles infinis, où la construction de chaque élément est faite en un nombre fini d'étapes. Cette caractéristique nous permettra de disposer d'un moyen de raisonnement et de calcul systématique pour chacun de ces types. La preuve par récurrence (*proof by induction*) sera le moyen de raisonnement et la construction de fonctions récursives sera le moyen de calcul.

2.4.8.2.1. Le type des entiers naturels

La présentation formelle la plus naturelle des nombres naturels est inspirée des travaux de Peano. Tout nombre naturel peut être obtenu, soit en prenant le nombre 0, soit en appliquant la fonction successeur à un nombre déjà construit. Dans le système Coq, ceci va s'exprimer par la définition suivante:

Coq<Inductive nat : Set := O : nat
| S: nat -> nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined

En effet, il n'existe que deux méthodes pour construire un nombre naturel. Soit on prend le nombre 0 (Zéro), soit on prend un nombre *x* déjà construit et l'on en construit un nouveau grâce à la fonction *S*.

Pour une propriété *P* donnée, si l'on arrive à démontrer que cette propriété est bien satisfaite par 0 et que si l'on prend un nombre *x* qui la satisfait alors le nombre construit *(S x)* la satisfait également alors, tout nombre naturel satisfait P.

Ceci s'exprime à l'aide d'un principe de récurrence, dont l'énoncé mathématique est le suivant:

$$\forall P.(P(0) \wedge (\forall x.P(x) \rightarrow P(S(x)))) \rightarrow \forall x.P(x)$$

Dans le système Coq, cet énoncé s'écrit de la manière suivante:

nat_ind :
(P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n))) ->(n:nat)(P n)
nat_rec :
(P:(nat->Set))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
nat_rect :
(P:(nat->Type))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

En pratique le système Coq engendre automatiquement le principe de récurrence *nat_ind, nat_rec et nat_rect* à partir de la définition inductive.

2.4.9. Types co-inductifs

Comme pour les types inductifs, un type co-inductif est spécifié par la signature (le type) de ses constructeurs. Les types co-inductifs se distinguent des types inductifs par le fait que leurs habitants contiennent une infinité de constructeurs, il faut signaler qu'à la suite de la définition d'un co-inductif, le système ne génère pas le principe d'induction et donc pour raisonner sur les habitants d'un tel type on disposera seulement l'analyse par cas.

Coq< Variable A : Set.
Coq<CoInductive Set Stream := Cons : A->Stream->Stream.

2.4.10. Les fonctions récursives

Une fonction [16] se définit généralement en indiquant la valeur qu'elle retourne pour un paramètre donné. On dit la fonction qui associe à *x* l'expression *e*. La variable *x* est autorisée à

apparaître dans l'expression *e*, ce qui permet d'assurer que la fonction n'est pas constante. Pour une fonction récursive, la fonction *f* qui associe à *x* l'expression *e*, avec la possibilité que *f* apparaisse aussi dans l'expression *e*. En d'autres termes, on suppose que la fonction *f* est déjà partiellement définie lorsque l'on essaie de déterminer la valeur qu'elle retourne pour une nouvelle valeur du paramètre.

En Coq, l'utilisateur doit déterminer les valeurs prises par la fonction dans un ordre précis, qui suit l'ordre dans lequel on a construit les termes des types inductifs. Pour les nombres naturels, on est obligé de construire *0* avant *(S 0)*, avant *(S (S 0))* et ainsi de suite. C'est cet ordre de construction des nombres qui est suivi pour la définition d'une fonction récursive *f*, de sorte que l'on s'autorise à utiliser la valeur *(f 0)* pour définir la valeur *(f (S 0)),* la valeur *(f (S 0))* pour définir la valeur *(f (S (S 0)))* et ainsi de suite. De manière générale, on s'autorise à utiliser la valeur *(f n)* pour définir la valeur *(f (S n)).*

En pratique, ce procédé de construction est fourni dans le système Coq par la commande *Fixpoint* qui prend la forme suivante dans le cas des nombres naturels :

Coq< Fixpoint f (n:nat) : T := expr.

Dans cette définition, *f* est le nom de la fonction que l'on est entrain de définir, *n* est le nom de l'argument sur lequel la récursion s'organise, *T* est le type retourné et *expr* indique comment la valeur de *(f n)* est déterminée.

Voici une fonction récursive simple sur les entiers naturels, qui multiplie son argument par 2.

```
Fixpoint mult2 [n:nat] : nat :=
Cases n of
O => O
| (S p) => (S (S (mult2 p)))
end.
```

Cette fonction fait bien apparaître un appel récursif dans la sous-expression *(mult2 p)*. Par l'intermédiaire du traitement par cas sur *n*, cet appel récursif est utilisé lorsque l'on veut déterminer la valeur de *mult2* sur l'argument *(S p)*.

Les fonctions récursives définies avec *Fixpoint* contiennent toujours un filtrage sur l'argument de la fonction, de sorte que l'on est amené à donner d'abord la valeur de la fonction lorsque cet

argument est *0*, puis la valeur de la fonction lorsque l'argument est de la forme *(S p)* avec la possibilité d'utiliser la valeur de la même fonction en *p*.

2.4.11. Manipulation des preuves

Dans cette partie, nous abordons les techniques de raisonnement en Coq, en commençant par la syntaxe de présentation des théorèmes :

Theorem < nom_du_théorème > : < énoncé >.
Lemma < nom_du_lemme> : < énoncé >.
Remark < nom_de_la_remarque > : < énoncé >.

Cet énoncé est appelé le but initial, d'une manière générale un but est la donnée d'une formule à prouver sous certaines hypothèses, celles-ci forme le contexte local d'hypothèses du but. Dans le cas d'un but initial, le contexte initial est vide.

Le développement d'une preuve se fait d'une manière interactive par l'usage de commandes appelées *tactiques*.

2.4.11.1. La tactique

Coq est basé sur la déduction naturelle, mais heureusement, l'utilisateur n'est pas toujours obligé de préciser, une à une, les règles de déduction utilisées pour construire une preuve. Il peut aussi s'aider des tactiques. Celles-ci permettent de regrouper un enchaînement d'applications de règles de déduction, ou même deviner quelles sont les règles que l'on peut utiliser dans certains cas.

Quand une tactique est appliquée à un but peut réussir à le résoudre c'est-à-dire à terminer la preuve, ou le transformer à un autre (ou plusieurs sous buts) avec un contexte différent ou bien échoué en laissant le but inchangé.

Les tactiques peuvent être combinées en utilisant des opérateurs appelés *tacticielles* (*tacticals en anglais) ;* On présente les plus importants :

- tac1 ; tac2 applique tac1 puis tac2 à tous les sous-buts issus de tac1.
- tac1 || tac2 tac1 est essayée et si tac1 échoue alors tac2
- tac1 ;[tac2|tac3] applique tac1 puis tac2 au premier sous-but généré puis tac3 au deuxième ...
- idtac ne fait rien
- repeat tac applique tac jusqu'à échec (fail)

- do num tac applique num fois tac
- try tac rattrape l'exception levée par tac
- first [tac1|tac2] applique la première tactique qui marche

2.4.11.1.1. Les tactiques de base

Nous présentons trois tactiques simples associées aux règles de typage *Var, App et Lam*.

1. La tactique *assumption* associe à la règle de typage *Var. Elle* est utilisée pour résoudre un sous-but T lorsque le contexte H contient une déclaration *(v : T)* (en terme de raisonnement, on dirait que la propriété qu'on veut prouver fait déjà l'objet d'une hypothèse.)
2. La tactique *intro* associe à la règle de typage *Abs* qui permet d'introduire les hypothèses et des variables quantifiées dans le contexte du but courant
   - Si le but courant est un produit dépendant (∀x :A)B, le nouveau but sera b et la variable x de type A sera poussée dans le contexte local.
   - Si le but est un produit non dépendant A → B une nouvelle hypothèse *Hi : A* est introduite dans le contexte et le but courant devient B.
3. La tactique *apply* associe à la règle de typage *App,* elle réduit le but à prouver en d'autre plus élémentaire par l'application d'un axiome ou un théorème déjà prouvé.

Construire une preuve revient à montrer un λ-terme du type attendu. Voici une illustration de leur principe, sur un exemple, on montre comment Coq aide à la construction du terme typé.

Les commandes fournies sont placées à gauche, et les réponses de Coq à droite. Un court texte explicatif est associé à chaque ordre donné au système.

Exemple 2.6 :

Parameters P Q R T : Prop. On définit P, Q, R et T comme symboles de propositions. Ils sont ajoutés à l'environnement.	P is assumed Q is assumed R is assumed T is assumed
Theorem diamond *:(P→Q)→(P→R)→(Q→R→T)→P→T.* On déclare qu'on veut définir et démontrer un	1 subgoal ========== (P→Q)→(P→R)→(Q→R→T)→P→T

théorème qu'on nomme diamond. Coq passe en mode de démonstration interactive. Il y a un seul but à résoudre qui est affiché en dessous de la ligne. Il n'y a pas d'hypothèse ; l'environnement est vide.	
intros. Cette tactique correspond à plusieurs applications de la règle ($\rightarrow I$). Attention, il faut se souvenir qu'on évolue depuis le but, et donc on « remonte » la déduction. Remarque: Coq choisit lui-même le nom des hypothèses qu'il introduit. En fait, il est possible de fournir les noms des hypothèses à la tactique : intros H H0 H1 H2.	1 subgoal H : P →Q H0 : P →R H1 : Q →R →T H2 : P ========== T
apply H1. Correspond à des applications successives de ($\rightarrow E$). On peut aussi le voir comme l'application d'une règle ($\rightarrow E$) généralisée $\Gamma \vdash A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$ $$\frac{\Gamma \vdash A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_{n-1}}{\Gamma \vdash A_n}$$ avec n = 3, A_3 = T, A_1 = Q et A_2 = R. Deux buts sont générés, seul le premier est affiché avec son environnement associé.	2 subgoals H : P →Q H0 : P →R H1 : Q →R →T H2 : P ========== Q subgoal 2 is: R
apply H. Correspond à une application de la règle ($\rightarrow E$). On avance dans la résolution du premier but, le second n'est pas modifié, ni son environnement associé.	2 subgoals H : P → Q H0 : P → R H1 : Q → R → T H2 : P ========== P subgoal 2 is: R
assumption. Correspond à l'application de la règle (Var). Le premier but est résolu, il ne reste plus que le second à résoudre.	1 subgoal H : P → Q H0 : P → R H1 : Q →R → T H2 : P ========== R

apply H0; assumption. C'est une tactique composée : applique *apply H0* au but courant, puis *assumption* à tous les sous-buts générés. Ici il n'y a qu'un seul but généré par *apply H0*.	Proof completed.
Qed. On explicite la fin de la démonstration, comme en mathématiques. Coq va alors vérifier que le λ-terme créé est bien typé du type cherché. En plus il affiche l'historique de la preuve.	intros. apply H1. apply H. assumption. apply H0; assumption. diamond is defined
Print diamond. On peut afficher le terme construit. Avec les notations usuelles du λ-calcul typé, le terme Coq correspond au terme : $\lambda\, h^{P\to Q}\, h_0{}^{P\to R}\, h_1{}^{Q\to R\to T}\, h_2{}^{P}.h_1\,(h\,h_2)\,(h_0\,h_2)$	diamond = fun (H:P→Q) (H0:P→R) (H1:Q→R→T) (H2:P) ⇒ H1 (H H2) (H0 H2) : (P→Q)→(P→R)→(Q→R→T)→P→T

Finalement, Coq nous a assisté à la construction du - terme ; ce terme a été créé en appliquant des tactiques dont le comportement est identique aux règles d'inférences de la déduction naturelle. Ceci souligne l'opérationnalité de l'isomorphisme de Curry-Howard, et donc de Coq.

2.4.11.1.2. Quelques autres tactiques

- **assert :** La tactique *assert A* introduit une coupure dans la démonstration du but courant B en remplaçant celui-ci par deux sous-buts:
  1. la proposition A dans le contexte courant.
  2. la proposition B dans le contexte courant étendu avec l'hypothèse A.

  ✓ **Variante:**

  La tactique *assert H* : A donne explictement le nom de l'hypothèse ajoutée au contexte courant dans le deuxième sous-but engendré.

- **assumption** : Cette tactique implémente la règle d'axiome de la déduction naturelle. Elle résout le but courant lorsque celui-ci figure parmi les hypothèses.

- **change :** La tactique *change A* remplace le but courant par la proposition A, sous réserve que celle-ci est convertible (au sens des règles de calcul du système) avec le but courant.
  - ✓ **Variante :**

    change A in H fait la même chose, mais dans l'hypothèse désignée par *H* plutôt que dans le but courant. La proposition *A* doit être convertible avec la proposition désignée par l'hypothèse *H*.

- **destruct** La tactique *destruct H* (où *H* est le nom d'une hypothèse du contexte) applique parmi les règles ∧-gauche, ⇒-gauche, ⊤-gauche, ⊥-gauche et ∃-gauche du calcul des séquents celle qui correspond au connecteur (ou au quantificateur) principal de l'hypothèse désignée par H.
  1. Dans le cas où l'hypothèse est de la forme *A ∧ B*, la tactique *destruct H* remplace dans le contexte courant cette hypothèse par deux nouvelles hypothèses *A* et *B*.
  2. Dans le cas où l'hypothèse est de la forme *A ∨ B*, la tactique *destruct H* remplace le sous-but courant par deux nouveaux sous-buts, dans lesquels l'hypothèse *A ∨B* est remplacée par l'hypothèse *A* et par l'hypothèse *B* respectivement.
  3. Dans le cas où l'hypothèse est *True*, la tactique *destruct H* fait disparaître cette hypothèse.
  4. Dans le cas où l'hypothèse est *False*, la tactique *destruct H* résout le but courant.
  5. Dans le cas où l'hypothèse est de la forme *exists x:T,A(x)*, la tactique *destruct H* introduit dans le contexte une déclaration de la forme *x : T* et remplace l'hypothèse désignée par *H* par une nouvelle hypothèse de la forme *A(x)*. Au cours de cette opération, il se peut que la variable x soit renommée pour éviter une collision avec un autre objet déclaré avec le même nom.

- **elim** La tactique *elim H* (où *H* est le nom d'une hypothèse du contexte) applique parmi les règles (∧ *E)*, (⇒ *E)*, (⊤ *E)*, (⊥ *E* et (∃ *E)* de la déduction naturelle celle qui correspond au connecteur (ou au quantificateur) principal de l'hypothèse désignée par *H*. En général, on lui préfère la tactique *destruct H*, qui est plus facile d'utilisation.
- **exact** La tactique *exact M* (où *M* est un terme de preuve de CCI) résout le but courant à l'aide du terme de preuve donné en argument. C'est la tactique de plus bas niveau du système, puisqu'elle se contente d'appeler le vérificateur de types/preuves, qui constitue le noyau de Coq.

- **exists** Cette tactique implémente la règle (∃*I)* de la déduction naturelle. Lorsque le but est de la forme *exists x:T, A(x),* la tactique *exists M* (où *M* est un terme de type *T* destiné à jouer le rôle du témoin) remplace le but courant par un but de la forme *A M.*
- **induction :** La tactique *induction id* (où *id* est un identificateur déclaré dans le contexte) applique sur le but courant le principe de récurrence associé au type de *id*. Cette tactique n'est valable que dans le cas où le type de *id* est un type inductif défini auparavant avec la commande *Inductive.*
- **reflexivity** : Cette tactique résout tout but de la forme *M1 = M2*, où *M1* et *M2* sont deux termes convertibles au sens des règles de calcul de Coq, comme par exemple les termes 2 + 2 et 4. En particulier, elle résout tous les buts de la forme *M = M*.
- **rewrite** : La tactique *rewrite H* (où *H* désigne une hypothèse de la forme *M1 = M2*) remplace dans le but courant toutes les occurrences du terme *M1* par le terme *M2*.
  - ✓ **Variantes :**

1. rewrite <- H utilise l'égalité *M1=M2* dans l'autre sens, en remplaçant *M2* par *M1*.
2. rewrite H in H0 fait la même chose que rewrite *H*, mais dans l'hypothèse désignée par *H0* plutôt que dans le but courant.
3. *rewrite <- H in H0* fait la même chose que *rewrite <- H*, mais dans l'hypothèse désignée par *H0* plutôt que dans le but courant.

- **simpl :**La tactique *simpl* réduit (au sens des règles de calcul du système) l'expression qui constitue le sous-but courant. Ce qui ne la simplifie pas toujours.
  - ✓ **Variantes:**

simpl in H fait la même chose, mais dans l'hypothèse désignée par H plutôt que dans le but courant.

- **split :** suivant la forme du but courant, cette tactique applique la règle (∧ *I)* ou la règle (⊤*I).*
  1. Lorsque le but courant est de la forme *A ∧ B,* la tactique split remplace le but courant par deux nouveaux sous-buts *A* et *B* (dans un contexte inchangé).
  2. Lorsque le but courant est *True*, split le résout.
- **symmetry :** la tactique symmetry remplace tout but courant de la forme *M1 = M2* par le but *M2 = M1*.

✓ **Variante**

symmetry in H fait la même chose dans une hypothèse *H* (de la forme *M1 = M2*).

- **transitivity :** la tactique *transitivity M* remplace tout but courant de la forme *M1 = M2* par deux sous-buts de la forme *M1 = M* et *M = M2*. (Les trois termes *M*, *M1* et *M2* doivent avoir le même type.)
- **unfold :** la tactique *unfold id* remplace dans le but courant toutes les occurrences de l'identificateur id par le corps de sa définition dans l'environnement courant.

✓ **Variante:**

unfold id in H fait la même chose dans une hypothèse *H*

2.4.12. L'extraction de programmes

Le système Coq permet de modéliser des programmes en les décrivant comme des fonctions dans un langage fonctionnel pur. Il faut cependant bien remarquer que le système Coq a pour but de synthétiser des programmes corrects ou de vérifier des programmes, mais pas de les exécuter. Cette tache est normalement déléguée aux outils habituels (compilateurs, machines abstraites, etc.)

Pour permettre la génération automatique de code exécutable certifié à partir des modèles formels, le système Coq permet la traduction des programmes fonctionnels du Calcul des Constructions vers des programmes fonctionnels purs d'un langage fonctionnel efficace, principalement le langage OCAML. Elle s'effectue en deux étapes :

- Extraction du λ-terme correspondant à la preuve après la suppression de l'information logique.
- Traduction de ce λ-terme en un programme écrit en langage fonctionnel (Caml, Ocaml ou Haskell).

En effet, la preuve de la consistance de la spécification contient à la fois l'information logique (qui nous assure que la démonstration est correcte) et l'information dite calculatoire (qui permet de construire la fonction ou le programme désiré). C'est cette dernière qui nous intéresse pour la génération du programme.

Après le chargement du module de l'extraction par la commande

Require Extraction.

La génération du code de programme dans le fichier nom_du_fichier par la commande :

Write Caml File " nom_du_fichier " [Liste_des_constantes].

Liste_des_constantes est la liste des théorèmes et/ou des lemmes que l'on voudra extraire.

2.4.13. Le langage de définition de tactiques

Le système Coq fournit également un langage de définition de tactiques appelé *L*tac. Ce langage permet d'écrire des tactiques paramétrées et récursives sans faire appel à la programmation directe en OCAML, ce qui imposerait une connaissance très précise des structures de données internes du système de preuve.

L'un des premiers avantages de l'utilisation du langage *L*tac est la possibilité d'attacher un nom court à des opérations parfois complexes.

Exemple 2.7 :

La tactique suivante applique répétitivement les théorèmes le_n[8] et le_S[9] pour démontrer qu'un nombre naturel arbitraire est inferieur a un autre nombre :

Ltac le_S_star := apply le_n || (apply le_S; le_S_star).

[8] « Le_n » est un théorème de type : $\forall\ n{:}nat,\ n \leq n$

[9] « Le_s » : est un théorème de type : $\forall\ n\ m{:}nat,\ n \leq m \rightarrow n \leq S\ m$

CHAPITRE III

LE SYSTEME ELAMBDA

Introduction

La logique combinatoire et le lambda-calcul sont des théories analysent successivement la notion de calcul effective. Cependant les fondateurs originaux de ces deux théories Curry et Church ont toujours comme but de fournir une base pour la logique et les mathématiques.

Malheureusement, Kleen et Rosser(1935) ont montré que le lambda calcul et la logique combinatoire étendus sans restriction par les règles d'introduction et d'élimination d'implication sont inconsistants. Ce qui a conduit Curry à chercher de définir un nouveau système vérifiant la propriété de consistance. Ce dernier est appelé « Logique Combinatoire Illative (ICL). ».

La logique combinatoire illative c'est la logique combinatoire (ou lambda-calcul pur) étendue par des constantes supplémentaires et un ensemble d'axiomes et règles de dérivation.

Le système E-lambda est une extension du lambda-calcul pur, où deux constantes « P et Π » sont introduites qui représentent respectivement l'implication et la quantification universelle. Le système obtenu est assez riche, dans le sens où les deux constantes introduites sont suffisantes pour exprimer et définir le reste des connecteurs [22].

La consistance du système Elambda est garantie grâce à l'affection d'un nouvel attribut, appelé « niveau » d'un terme, utilisé pour introduire une nouvelle définition du processus de substitution ; où terme M [N/x] n'est défini que si le niveau du terme « N » est inférieur ou égal à celui de « x ». Cette restriction nécessite une définition propre du mécanisme de

réduction, appelé Ebeta_rduc, qui vérifié la propriété de Church-Rosser et offre un moyen pour éviter le paradoxe de Curry. Le système Elambda est aussi un système avec lequel on peut interpréter la logique d'ordre supérieur.

L'objectif principal de ce chapitre est de formaliser la théorie Elambda en utilisant l'assistant de preuve Coq et de démontrer que la réduction Eβ est confluente (Théorème de Church-Rosser).

Dans notre formalisation, la définition des termes et des prédicats est donnée sous forme inductive, telle qu'elle est requise par le système Coq. La majorité des preuves dans coq sont basées sur le principe d'induction.
La preuve du théorème de Church-Rosser nécessite des lemmes intermédiaires. L'objectif est d'éviter de démontrer la même propriété plusieurs fois, d'écourter la preuve et de la rendre lisible et simple à maintenir dès le changement de la spécification associée.

Le contenu de ce chapitre se résume comme suit :

- Modélisation de Elambda-calcul: Spécification des types de données (Termes) et des relations (libre, réduction, égalité syntaxique..).
- Démonstration de quelques lemmes intermédiaires.
- Démonstration du théorème de Church-Rosser.

3.1. Les termes du Système Elambda

Le système Elambda est caractérisé par :

1. L'ensemble des termes.
2. La relation de réduction des termes.
3. Les règles d'inférence.

L'ensemble des termes de système Elambda « C_w » est l'union des ensembles C_0, C_1, C_2, . . Chaque ensemble C_i (i >= 1) est construit par l'ajout à l'ensemble C_{i-1} des termes construits par l'introduction de deux constantes P et Π.
L'ensemble C_0 est l'ensemble des termes de λ –calcul classique tel que les variables sont de la forme x^0, y^0, z^0, . . .
L'ensemble C_0 ne contient pas les constantes P et Π.

Définition3. 1 *Les termes d'un ensemble C_0*

L'ensemble des termes C_0 est construit à partir d'un ensemble de variables V_0 tel que $V_0 = \{x^0, y^0, z^0, \ldots\}$ en utilisant l'application et λ-abstraction de la manière suivante :

- Si $x \in V_0$ alors $x \in C_0$
- Si M, N deux termes de C_0 alors $(M\ N) \in C_0$
- Si $M \in C_0$ et $x \in V_0$ alors $(\lambda x.\ M) \in C_0$

Les étapes de construction de l'ensemble C_i ($i >= 1$) à partir de l'ensemble C_{i-1} sont données par la définition suivante.

Définition 3.2 *Les termes d'un ensemble C_i*

L'ensemble des termes C_i est construit de manière inductive à partir d'un ensemble infini de variables V_i tel que $V_i = V_{i-1} \cup \{x^i, y^i, z^i, \ldots\}$ ($i >= 1$) et les constantes P et Π :

- Si $x \in V_i$, alors $x \in C_i$.
- Si $M \in C_i$, $N \in C_j$ alors $(M\ N) \in C_{max\ (i\ ,\ j)}$.
- Si $x \in V_i$, $M \in C_j$ alors $(\lambda x.M) \in C_{max\ (i\ ,\ j)}$
- Si $X \in C_i$ alors $(\Pi\ X) \in C_{max\ (1\ ,\ i)}$
- Si $X \in C_i$, $Y \in C_j$ alors $(P\ X\ Y) \in C_{max(i\ ,\ j)\ +\ 1}$

Exemple 3.1

- $x^i \in C_j$ avec $j >= i$
- $\lambda x^i . x^j \in C_{max(i,j)}$
- $\Pi x^i \in C_{max(1,i)}$

Remarque 3.1

Il est facile de remarquer que $C_0 \subset C_1 \subset C_2 \subset \ldots \subset C_i \subset \ldots .$

Définition 3.3 *Les termes du système Elambda*

L'ensemble des termes C_w est définie comme suit

$C_w = \cup\ C_i$. (avec $0 \leq i \leq \infty$)

Une variable du système E-lambda est définie en Coq comme suit :

Inductive evar: Set:=

*|var: nat*nat → evar.*

Les termes du système E-lambda sont définis en Coq comme suit :

Inductive c_term: Set:=

|cvar: evar → c_term

|App: c_term → c_term → c_term

|Abs: evar → c_term → c_term

|Imp: c_term → c_term → c_term

|Pl: c_term → c_term.

Dans ce qui suit nous définissons la notion de niveau d'un Eλ-terme qui nous permet de modifier l'opération de substitution dans le but d'éviter le paradoxe de Curry.

Définition 3.4 *Le niveau d'un Eλ-terme*

Le niveau d'un Eλ-terme est calculé par les règles suivantes :

- niveau (x^i) = i.
- niveau (λx.M) = max (niveau (x), niveau (M)).
- niveau (P X Y) = max (niveau (X), niveau (Y)) +1.
- niveau (Π X) = max (1, niveau (X)).
- niveau (X Y) = max (niveau (X), niveau (Y)).

Le niveau d'un terme est défini dans le système Coq comme suit :

Inductive level: c_term → nat → Prop:=

|cvar_level: forall n1 n2: nat, (level (cvar (var(n1, n2))) n2)

|Abs_level: forall n1 n2 m:nat, forall M:c_term, (level M m) → (level(Abs (n1,n2)M) (max m n2))

|App_level: forall m n:nat, forall M N: c_term,(level M m) → (level N n) → (level (App M N) (max m n))

|Imp_level: forall m n:nat, forall M N:c_term, (level M m) → (level N n) → (level (Imp M N) (S(max m n)))

|Pl_level: forall m:nat, forall M:c_term, (level M m) → (level (Pl M) (max 1 m)).

Notation : Dans la suite nous utilisons les notations suivantes

- « X ⊃ Y » au lieu de « P X Y ».
- X^k représente un terme de niveau k.

Exemple3. 2

Si le niveau de X = i et le niveau de Y = 0 alors le niveau de X ⊃ Y = max(i,0) + 1 = i + 1.

Soit Y ≡ (λx.(x ⊃ y)) (λx.(x ⊃ y)) alors le niveau(Y) = max(niveau (λx.(x⊃y)), niveau (λx.(x⊃y))) = niveau (λx.(x⊃y)) = max (niveau (x), niveau (x ⊃y)) = niveau(x ⊃y) = max(niveau (x), niveau(y)) + 1.

Définition 3.5 *Variable libre*

La notion de variable libre et variable liée est définie comme d'habitude dans λ-calcul classique de Church[10], la variable libre est définie en coq de manière inductive par le prédicat is_free comme suit :

```
Inductive is_free : evar -> c_term -> Prop:=
|cvar_free : forall p q, (is_free (var(p, q)) (cvar(var(p,q))))

|Abs_free : forall n1 n2 p q M, (n1 < > p \/ n2 < > q) -> (is_free (var(n1,n2)) M) ->
                                (is_free (var(n1,n2)) (Abs(p,q)M))
|App_free : forall p q M N, ((is_free (var(p,q)) M) \/ (is_free (var(p,q)) N)) ->
                                (is_free (var(p,q))( App M N))


|Imp_free: forall p q M N, ((is_free (var(p,q)) M) \/ (is_free (var(p,q)) N)) ->
                                (is_free (var(p,q))( Imp M N))

|Pl_free : forall p q M, (is_free (var(p,q)) M) -> (is_free (var(p,q))( Pl M )).
```

[10] Pour plus de détaille sur la notion du variable libre voir la définition 1.4 du Chapitre I

3.2. La relation de réduction Ebeta_reduc

Avant de définir la relation de réduction des termes, on doit d'abord définir l'opération de substitution qu'est définie en Coq de manière inductive comme suit:

```
Inductive substitution : evar-> c_term -> c_term -> c_term -> Prop :=
|sub_var_eq  : forall p q X,  ((level X ) <= q) ->
          (substitution (var(p,q))  X (cvar(var(p,q))) X)

|sub_var_neq  : forall p q m n  X, (level X  <= q)  -> ((p,q) <>(m,n)) ->
          (substitution (var(p,q))  X (cvar(var(m,n))) (cvar(var(m,n))))

|sub_abs_nfree0 : forall p q  M  X, (level X  <= q) ->
           (substitution (var(p,q))  X (Abs (var (p,q))M) (Abs (var (p,q))M))

|sub_abs_nfree1 : forall p q m n M  Z X,  ~(is_free (var(m,n)) X) ->
           (level X  <= q) -> ((p,q) <>(m,n)) -> (substitution (var(p,q))  X M  Z) ->
           (substitution (var(p,q))  X (Abs (var(m,n))M) (Abs (var(m,n))Z))

|sub_abs_nfree2 :forall p q m n M   X,  ~(is_free (var(p,q)) M) ->
          ((level X)  <= q) -> ~(is_free (var(m,n)) X) ->((p,q) <>(m,n))->
          (substitution (var(p,q))  X (Abs (var(m,n))M) (Abs (var (m,n))M))

|sub_app :  forall p q M N M1 N1   X , ((level X ) <= q) ->
        (substitution (var(p,q))  X M M1) ->
        (substitution (var(p,q)) X N  N1 ) ->
        (substitution (var(p,q)) X (App M N)  (App M1 N1))

|sub_imp :  forall p q M N M1 N1  X, ( (level X ) <= q) ->
        (substitution (var(p,q))  X M M1) -> (substitution (var(p,q)) X N  N1 ) ->
        (substitution (var(p,q)) X (Imp M N)  (Imp M1 N1))

| sub_pi :  forall p q M   X N,  ((level X ) <= q) ->
        (substitution (var(p,q)) X M N) -> (substitution (var(p,q)) X (PI M) (PI N)).
```

Définition 3. 6 *La congruence*

On dit que P est congruent à Q ($P \equiv_\alpha Q$), si Q se diffère de P uniquement par le nom de quelques variables liées de même niveau. La congruence est définie en Coq comme suit :

```
Inductive alpha_red: c_term -> c_term -> Prop:=
 |alpha_red0: forall p q r:nat, forall t1 t2: c_term, ~(is_free (var(r,q)) t1) ->
 (substitution (var(p,q)) (cvar(var(r,q))) t1 t2) -> (alpha_red (Abs (p,q) t1) (Abs (r,q)  t2)).
```

Exemple 3. 3

Les termes $(\lambda x.(xx) \supset Z)$ et $(\lambda y.(yy) \supset Z)$ sont congruents uniquement si level(x) = level(y).

Définition 3. 7 *La β-réduction*

La relation de beta_réduction est définie dans le système coq comme suit :

```
Inductive beta_red: c_term -> c_term -> Prop:=
 |beta_red0: forall p q:nat, forall N M1 M2:c_term, (substitution (var(p,q)) N M1 M2) ->
    (beta_red ( App (Abs (p,q) M1)N) M2).
```

Définition 3.8 *La Eβ-réduction*

Le processus de reduction dans le Eλ-calcul est défini par les règles suivantes :

(ρ) $M \rightarrow_{E\lambda\beta} M$

(μ) $M \rightarrow_{E\lambda\beta} N \Rightarrow Z\ M \rightarrow_{E\lambda\beta} Z\ N$

(μ') $M \rightarrow_{E\lambda\beta} N \Rightarrow X \supset M \rightarrow_{E\lambda\beta} X \supset N$

(μ'') $M \rightarrow_{E\lambda\beta} N \Rightarrow \Pi\ M \rightarrow_{E\lambda\beta} \Pi\ N$

(ν) $M \rightarrow_{E\lambda\beta} N \Rightarrow M\ Z \rightarrow_{E\lambda\beta} N\ Z$

(ν') $M \rightarrow_{E\lambda\beta} N \Rightarrow M \supset X \rightarrow_{E\lambda\beta} N \supset X$

(τ) $M \rightarrow_{E\lambda\beta} N, N \rightarrow_{E\lambda\beta} T \Rightarrow M \rightarrow_{E\lambda\beta} T$

(Eβ) $(\lambda x.M)N \rightarrow_{E\lambda\beta} M[N/x]$, Si le niveau(N) ≤ niveau(x) ou niveau $(\lambda x.M) = 0$

(Eξ) $M \rightarrow_{E\lambda\beta} N, \lambda\ x.\ M \rightarrow_{E\lambda\beta} \lambda\ x.\ N$

La relation Ebeta_reduc est défini dans le système Coq comme suit :

```
Inductive Ebeta_reduc: c_term -> c_term -> Prop:=
```

```
|refl_reduc: forall M:c_term, (Ebeta_reduc M M)

|Appl_reduc: forall M N Z:c_term, (Ebeta_reduc M N) ->
    (Ebeta_reduc (App Z M) (App Z N))

|Impl_reduc: forall M N X:c_term, (Ebeta_reduc M N) ->
(Ebeta_reduc (Imp X M) (Imp X N))

|Pl_reduc: forall M N:c_term, (Ebeta_reduc M N) -> (Ebeta_reduc (Pl M) (Pl N))

|Appr_reduc: forall M N Z:c_term, (Ebeta_reduc M N) ->
    (Ebeta_reduc (App M Z) (App N Z))

|Impr_reduc: forall M N X:c_term, (Ebeta_reduc M N) ->
    (Ebeta_reduc (Imp M X) (Imp N X))

|beta_reduc: forall M N:c_term, (beta_red M N) -> (Ebeta_reduc M N)

|trans_reduc: forall M N T:c_term, (Ebeta_reduc M N) -> (Ebeta_reduc N T) ->
(Ebeta_reduc M T)

|Abs_reduc: forall M N:c_term, forall p q:nat, (Ebeta_reduc M N) ->
    (Ebeta_reduc (Abs (p, q) M) (Abs (p, q) N)).
```

La relation $t =_{E\lambda\beta} u$ (t est β-équivalent à u) est définie comme la fermeture réflexive-symétrique-transitive de la relation $\rightarrow_{Ebeta\_reduc}$, on la définit dans le système Coq comme suit :

```
Inductive Ebeta_equal: c_term -> c_term -> Prop:=
| Def_Ebeta_equal: forall x y :c_term, (Ebeta_reduc x y) -> (Ebeta_equal x y)
| Ebeta_equal_refl: forall X: c_term, (Ebeta_equal X X)
| Ebeta_equal_trans: forall X Y Z: c_term, (Ebeta_equal X Y) -> (Ebeta_equal Y Z) ->
(Ebeta_equal X Z)
| Ebeta_equal_sym: forall X Y: c_term, (Ebeta_equal X Y) -> (Ebeta_equal Y X).
```

Exemple 3.4:

Maintenant, nous montrerons quelques exemples qui démontrent que la relation de réduction d'un terme dépend de son niveau (level).

- $(\lambda x^i . (y^j \supset x^i))Z^0 \rightarrow_{E\lambda\beta} [Z/x^i](y^j \supset x^i)$ d'après la relation beta_reduc
- Le level de terme (λx.xx) est égal au level de la variable x, donc on peut appliquer la règle beta_reduc ; (λx.xx) (λx.xx) $\rightarrow_{E\lambda\beta}$ (λx.xx) (λx.xx) $\rightarrow_{E\lambda\beta}$ . . . $\rightarrow_{E\lambda\beta}$ (λx.xx) (λx.xx) $\rightarrow_{E\lambda\beta}$ . . .
- Le terme $(\lambda x^i.(x^i \supset x^i))$ $(\lambda y^j . y^j \supset y^j)$ est non réductible. La règle Ebeta_reduc ne peut être pas appliquer car le level $(\lambda y^j . y^j \supset y^j) = i+1 >$ level (x^i) .

3.3. Le processus d'inférence

- Une formule du système Elambda est un Eλ-terme
- Un contexte est un ensemble fini de formules et les hypothèses sont les formules d'un ensemble fini.
- Si Γ est un contexte et X est une formule, on dit que X est dérivable à partir de Γ et on écrit $X \vdash \Gamma$, si X est produit par le système de déduction suivant :

1. (ax) $X \in \Gamma \Rightarrow \Gamma \vdash X$.
2. (Eq_Ebeta) $\Gamma \vdash X$, $X =_{E\lambda\beta} Y$, niveau (Y) < = niveau(X) $\Rightarrow \Gamma \vdash Y$.
3. (E_imp) $\Gamma \vdash X \supset Y$, $\Gamma \vdash X \Rightarrow \Gamma \vdash Y$.
4. (I_imp) $\Gamma , X \vdash Y \Rightarrow \Gamma \vdash X \supset Y$.
5. (E_PI) $\Gamma \vdash \Pi (\lambda x^i . X) \Rightarrow \Gamma \vdash (\lambda x^i . X) Y^k$ avec k < = i.
6. (I_PI) $\Gamma \vdash M \Rightarrow \Gamma \vdash \Pi (\lambda x . M)$ avec $x \notin FV (\Gamma)$

Pour formaliser les règles de dérivation par le système Coq, on a défini d'abord le contexte Γ sous forme de liste de termes et on a défini aussi un prédicat qui décrit la notion de variable libre dans un contexte Γ.

- **Formalisation du contexte Γ par Coq**

Definition gamma:= list c_term.

- **Formalisation de la variable libre dans un contexte Γ par Coq**

Inductive var_free_gamma: evar -> gamma -> Prop:=

|free_cons_nil: forall p q:nat, (var_free_gamma (var(p,q)) ((cvar(var(p,q)))::nil))

|free_cons_cons: forall (p q:nat) (X:c_term) (G:gamma), (var_free_gamma (var(p,q)) G) -> (var_free_gamma (var(p,q)) (X:: G)).

- **Formalisation du processus de déduction**

Inductive derivable : c_term -> gamma -> Prop:=

|ax: forall (x:c_term) (G:gamma), In x G -> (derivable x G)

|Eq_Ebeta: forall x y:c_term, forall G:gamma, forall p q:nat, (derivable x G) -> (Ebeta_equal x y) -> (level y q) -> (level x p) -> (q <= p)->(derivable y G)

|E_imp: forall x y: c_term,forall G:gamma, (derivable (Imp x y) G) -> (derivable x G) -> (derivable y G)

|I_imp: forall x y:c_term, forall G:gamma, (derivable y (x:: G)) -> (derivable (Imp x y) G)

|E_PI: forall p q r:nat, forall M N: c_term, forall G:gamma, (derivable (PI(Abs(p,q) M)) G) ->(level N r) -> r <= q -> (derivable (App (Abs(p,q) M)N) G)

|I_PI: forall p q:nat, forall M:c_term, forall G: gamma, (derivable M G) -> ~(var_free_gamma (var(p,q)) G) -> (derivable (PI(Abs(p,q) M)) G).

3.4. Les connecteurs et les quantificateurs logiques

En Elambda-calcul, On est capable de définir les différents connecteurs et quantificateurs logiques.

En suivant le travail de Curry dans la logique combinatoire illative, on définit les connecteurs et les quantificateurs logiques avec l'utilisation de la notion de niveau.

$(\forall x^i)X \equiv \Pi (\lambda x^i. X)$

$\perp^i \equiv (\forall x^i)\, x^i$

$\neg^{i+1} \equiv (\lambda x^i.\ (x^i \supset \bot))$

$\Xi^{i+1} \equiv (\lambda x^i y^i.\ (\forall\ z^i)\ (x^i\ z^i) \supset (y^i\ z^i))$

$F^{i+1} \equiv (\lambda\ x^i\ y^i\ z^i.\ (\forall\ u^i)\ (x^i\ u^i) \supset (y^i\ (z^i\ u^i)))$

$G^{i+1} \equiv (\lambda\ x^i\ y^i\ z^i.\ (\forall\ u^i)\ (x^i\ u^i) \supset (y^i\ u^i\ (z^i\ u^i)))$

$\bigwedge^{i+3} \equiv \lambda\ x^i\ y^i.\ (\forall z^i)\ ((x^i \supset (y^i \supset z^i)) \supset z^i)$

$\bigvee^{i+3} \equiv \lambda\ x^i\ y^i.\ (\forall z^i)\ ((x^i \supset z^i) \supset (y^i \supset z^i)) \supset z^i$

$EQ^{i+1} \equiv \lambda\ x^i\ y^i.\ (\forall z^i)\ ((z^i\ x^i) \supset (z^i\ y^i))$

Les connecteurs et les quantificateurs logiques sont définis dans le système coq comme suit :

Definition eforall(x i:nat)(X: c_term):= PI (Abs(x,i) X).

Definition bottom (i:nat) := (eforall x i (cvar(var(x,i)))).

Definition neg(i :nat) := (Abs(x,i) (Imp (cvar(var(x,i))) (bottom i))).

Definition segma (i:nat):= Abs(x,i)(Abs(y, i)(eforall z i (Imp(App(cvar(var(x,i))) (cvar(var(z, i)))) (App(cvar(var(y, i))) (cvar(var(z, i))))))).

Definition F(i:nat):= Abs(x,i)(Abs(y,i)(Abs(z, i)(eforall u i (Imp(App(cvar(var(x,i))) (cvar(var(u, i)))) (App(cvar(var(y, i))) (App(cvar(var(z, i))) (cvar(var(u, i)))))))).

Definition G(i:nat):= Abs(x, i)(Abs(y, i)(Abs(z, i)(eforall u i (Imp(App (cvar(var(x, i))) (cvar(var(u, i)))) (App(App(cvar(var(y, i))) (cvar(var(u, i)))) (App(cvar(var(z, i))) (cvar(var(u, i)))))))).

Definition And(i:nat):= Abs(x,i)(Abs(y,i)(eforall z i (Imp(Imp(cvar(var(x, i))) (Imp(cvar(var(y,i))) (cvar(var(z,i))))) (cvar(var(z, i))))))).

Definition Or(i :nat):= Abs(x, i)(Abs(y,i)(eforall z i (Imp(Imp(Imp(cvar(var(x,i))) (cvar(var(z,i)))) (Imp (cvar(var(y,i))) (cvar(var(z,i))))) (cvar(var(z,i))))))).

Definition EQ (i:nat):= Abs(x,i)(Abs(y,i)(eforall z i (Imp (App(cvar(var(z,i))) (cvar(var(x,i)))) (App(cvar(var(z,i))) (cvar(var(y,i))))))).

Remarques

Si on interprète le terme « X Y » par « Y Є X « ou « Y satisfait le prédicat X », alors le terme « Ξ X Y » peut être interprété comme « X ⊆ Y » ou « ∀ x (X(x) ⊃ Y(x)) »

La constante F est utilisé pour représenter les types de fonctions, par exemple « F a b X » est interprété par « X est une fonction définie de a vars b ».

La constante G est utilisée pour représenter les types de fonctions dépendants, par exemple « G A (λ x. B) » est interprété par « (∀ x Є A) B ».

On peut remarquer que chaque connecteur et chaque quantificateur est un Eλ-terme avec son propre niveau : pour un connecteur nous avons une infinité de Eλ-termes.

Les deux constantes du système Elambda, correspondent au quantificateur universel et à l'implication, peuvent être vus comme un système complet dans le sens où on peut exprimer les autres connecteurs (F, Ξ, G, ∧, …) en fonction de ces deux constantes.

Exemple 3. 5 :

Pour définir le terme représentant la négation d'un terme X avec niveau (X) = K, on doit appliquer le connecteur $\neg^{i+1}$ au terme X avec $K \leq i$.

$\neg^{i+1}\ X \equiv (\lambda x^i.\ (x^i \supset \bot))\ X \rightarrow_{E\lambda\beta} X \supset \bot$.

3.5. Le système Elambda et le paradoxe de Curry :

Le paradoxe de curry fut présenté par le mathématicien Haskell Curry en 1942 et permet d'arriver à n'importe qu'elle conclusion à partir d'une phrase auto-référentielle et de quelques règles logiques simple. Cela peut s'exprimer de façon tout à fait formelle.

Soit $Y \equiv (\lambda x.(xx)\supset X)\ (\lambda x.(xx)\supset X)$ alors $Y =_\beta (Y \supset X)$.

Ainsi on a :

1. $Y \vdash Y$
2. $Y \vdash Y \supset X$ car $Y =_\beta (Y \supset X)$.
3. $Y \vdash X$ 1 + 2 + élimination de ⊃.
4. $\vdash Y \supset X$ introduction de ⊃.
5. $\vdash Y$ car $Y =_\beta (Y \supset X)$.
6. $\vdash X$ 4 + 5 + élimination de ⊃.

A partir de ces étapes de preuves Curry a déduit que lambda calcul est un système inconsistant.

Mais heureusement dans le système Elambda, le paradoxe de Curry est évité grâce à la notion de niveau du terme (level), car si le niveau(x) = i et le niveau ($\lambda x.(xx) \supset X$) = j alors $i \leq j$, donc la règle Ebeta_reduc ne peut être appliqué et $Y =_{E\lambda\beta} (Y \supset X)$ n'est pas vérifiée.

3.6. Démonstration de lemmes intermédiaires :

Avant de démontrer la confluence d'Elambda-beta-réduction qui est le but de notre travail, on doit démontrer d'abord quelques lemmes intermédiaires. Le code en coq de ces lemmes est donné dans l'annexe B.

Proposition 1 :

Pour tout termes M P et pour toute variable x,si niveau(M) ≤ niveau (x) alors niveau (P[M/x]) ≤ niveau (P).

La spécification formelle de cette proposition dans le système coq est :

Lemma subst_level: forall(N M P:c_term) (n1 n2:nat), ((level M) <= n2)-> (substitution (var(n1,n2)) M P N) -> (level N) <= (level P).

La preuve est par récurrence sur le terme P :

1. Si P est une variable y tel que :
   - $y \equiv x$ alors (x [M/x]) = M et par hypothèse on niveau(M) ≤ niveau(x).
   - $y \neq x$ alors (y [M/x]) = y et niveau(y) ≤ niveau(y).

2. Si $P \equiv$ App N_1 N_2 alors on a niveau(App N_1 N_2) = max(niveau (N_1), niveau(N_2)).
   Par hypothèse de récurrence on a niveau (N_1 [M/x]) ≤ niveau (N_1) et niveau (N_2 [M/x]) ≤ niveau (N_2). D'où niveau (App N_1 N_2 [M/x]) = max ((niveau(N_1 [M/x]), niveau (N_2 [M/x]) <= max(niveau (N_1), niveau(N_2)).
   D'où niveau (App N_1 N_2 [M/x]) ≤ niveau (P).

3. Si $P \equiv \lambda y.N$ alors on a niveau (λ y.N) = (max (niveau (y)), niveau(N)) niveau (λ y . N[M/x]) = (max (niveau (N[M/x]) , niveau (y)) et par hypothèse de récurrence

niveau (N[M/x])≤ niveau(N). D'où niveau (λ y . N[M/x])≤ (max niveau (y), niveau(N)) .

4. Si P ≡ Imp N_1 N_2 alors on a niveau (Imp N_1 N_2) = S(max(niveau (N_1), niveau(N_2))).
   Par hypothèse de récurrence on a niveau (N_1 [M/x]) ≤ niveau (N_1) et niveau (N_2 [M/x]) ≤ niveau (N_2) .
   D'où niveau (Imp N_1 N_2 [M/x]) = S(max((niveau(N_1 [M/x]), niveau (N_2 [M/x]))≤ S(max(niveau (N_1), niveau(N_2))).

5. Si P ≡ PI N, on a niveau (PI N) = max (niveau (N),1) et par hypothèse de récurrence on a niveau(N[M/x]) ≤ niveau(N). D'où niveau(PI N[M/x]) = max (niveau(N[M/x]),1) <= max(niveau (N),1) .

Proposition 2 :

Pour tout termes P Q, si P $\rightarrow_{E\lambda\beta}$ Q alors niveau(Q) ≤ niveau (P).
La spécification formelle de cette proposition dans le système coq est :
Lemma ebeta_red_level_decr: forall M N:c_term, (Ebeta_reduc M N) ->
(level N) <= (level M).

La preuve est par récurrence sur la longueur de réduction P $\rightarrow_{E\lambda\beta}$ Q.

1. Si P $\rightarrow_{E\lambda\beta}$ P, alors par l'application du théorème le_refl on a niveau(P) ≤ niveau(P)

2. Si (App Z P) $\rightarrow_{E\lambda\beta}$ (App Z Q), alors on a niveau (App Z P) = max(niveau(Z),niveau(P)) et niveau(App Z Q) = max(niveau(Z),niveau(Q)) et par hypothèse de récurrence on déduit que niveau(P) ≤ niveau(Q). D'où niveau (App Z P) ≤ niveau (App Z Q).
3. Si (App P Z) $\rightarrow_{E\lambda\beta}$ (App Q Z), alors on a niveau (App P Z) = max (niveau (P), niveau(Z)) et niveau (App Q Z) = max (niveau(Q), niveau(Z)) et par hypothèse de récurrence on déduit que niveau(P) ≤ niveau(Q). D'où niveau (App P Z) ≤ niveau (App Q Z).

4. Si (Imp Z P) $\rightarrow_{E\lambda\beta}$ (Imp Z Q), alors on a niveau(Imp Z P) =S(max(niveau(Z),niveau(P)))et niveau(Imp Z Q) = S(max(niveau(Z),niveau(Q)))et par

hypothèse de récurrence on déduit que niveau(P) ≤ niveau(Q). D'où niveau (Imp Z P) ≤niveau (Imp Z Q).

5. Si (Imp P Z) $\rightarrow_{E\lambda\beta}$ (Imp Q Z), alors on a niveau (Imp P Z) = max (niveau (P),niveau (Z)) et niveau(App Q Z) = max (niveau(Q),niveau(Z)) et par hypothèse de récurrence on déduit que niveau(P) ≤ niveau(Q). D'où niveau (App P Z) ≤ niveau(App Q Z).

6. Si (App ($\lambda x.P_1$) Q_1) $\rightarrow_{E\lambda\beta}$ ($P_1[Q_1/x]$) avec niveau(Q_1) ≤ niveau(x). Par définition de niveau on a niveau(App ($\lambda x.P_1$) Q_1) = max(niveau($\lambda x.P_1$), niveau(Q_1)) = niveau($\lambda x.P_1$) car niveau(Q_1) ≤ niveau(x). D'après la proposition 1 on déduit que niveau(P_1 [Q_1/x]) ≤ niveau(P_1) ≤ niveau($\lambda x.P_1$).

7. $\lambda x.P \rightarrow_{E\lambda\beta} \lambda x.Q$, par définition de niveau on a niveau ($\lambda x.P$) = max(niveau(P),niveau(x)) et niveau($\lambda x.Q$) = max(niveau(Q),niveau(x)). Par hypothèse de récurrence on déduit que niveau(Q) ≤ niveau(P). D'où niveau($\lambda x.Q$) ≤ niveau ($\lambda x.P$).

8. Si (PI P) $\rightarrow_{E\lambda\beta}$ (PI Q), par définition de niveau on a niveau(PI P) = max (1,niveau(P)) et niveau(PI Q) = max(1,niveau(Q)). Par hypothèse de récurrence on déduit que niveau(Q) ≤ niveau(P). D'où niveau(PI P) ≤ niveau(PI Q).

9. Si P$\rightarrow_{E\lambda\beta}$ M et M $\rightarrow_{E\lambda\beta}$ Q, par hypothèse de récurrence on déduit que niveau (M) ≤ niveau (P) et niveau(Q) ≤ niveau(M). D'où par transitivité on déduit que niveau(Q) ≤ niveau (P)

Proposition 3:

Pour tout termes M P Q et pour toute variable x, si P $\rightarrow_{E\lambda\beta}$ Q alors (M [P/x]) $\rightarrow_{E\lambda\beta}$ (M [Q/x]).

La spécification formelle de cette proposition est :

Lemma Ebeta_subst_Ebeta: forall P Q M P1 Q1:c_term, forall n1 n2:nat, (Ebeta_reduc P Q) -> (substitution (var(n1,n2)) P M P1) -> (substitution (var(n1,n2)) Q M Q1) ->(Ebeta_reduc P1 Q1).

La preuve est par récurrence sur le terme M.

1. Si M est une variable tel que :
   - M = x alors (x [P/x]) = P et (x [Q/x]) = Q, d'où par hypothèse du contexte on déduit P $\rightarrow_{E\lambda\beta}$ Q.
   - M ≠ x alors (M [P/x]) = M et (M [Q/x]) = M. D'où on déduit que M$\rightarrow_{E\lambda\beta}$ M car $\rightarrow_{E\lambda\beta}$ est une relation réflexive.

2. Si M ≡ (App M_1 M_2) alors ((App M_1 M_2) [P / x]) = App (M_1[P/x]) (M_2[P/x]) et ((App M_1 M_2) [Q / x]) = App (M_1[Q/x]) (M_2[Q/x]). Par hypothèse de récurrence on déduit que (M_1[P/x]) $\rightarrow_{E\lambda\beta}$($M_1$[Q/x]) et ($M_2$[P/x]) $\rightarrow_{E\lambda\beta}$ (M_2[Q/x]). D'où par l'application de la propriété de transitivité de la relation $\rightarrow_{E\lambda\beta}$ on déduit que ((App M_1 M_2) [P / x]) $\rightarrow_{E\lambda\beta}$ ((App M_1 M_2) [Q / x]).

3. Si M ≡ λy.M_1,alors on a (λy .M_1[P/x]) = λy.(M_1[P/x]) et (λy .M_1[Q/x]) = λy.(M_1[Q/x]) . Par hypothèse de récurrence on déduit que (M_1[P/x]) $\rightarrow_{E\lambda\beta}$ (M_1[Q/x]) . D'où λy.(M_1[P/x]) $\rightarrow_{E\lambda\beta}$ λy.(M_1[Q/x]) .

4. Si M ≡ (Imp M_1 M_2) alors ((Imp M_1 M_2) [P / x]) = Imp (M_1[P/x]) (M_2[P/x]) et ((Imp M_1 M_2) [Q / x]) = Imp (M_1[Q/x]) (M_2[Q/x]). Par hypothèse de récurrence on déduit que (M_1[P/x]) $\rightarrow_{E\lambda\beta}$($M_1$[Q/x]) et ($M_2$[P/x]) $\rightarrow_{E\lambda\beta}$ (M_2[Q/x]).
   D'où par l'application de la propriété de transitivité de la relation $\rightarrow_{E\lambda\beta}$ on déduit que ((Imp M_1 M_2) [P / x]) $\rightarrow_{E\lambda\beta}$ ((Imp M_1 M_2) [Q / x]).

5. Si M≡ (PI M_1) alors on a ((PI M_1) [P/x]) = PI (M_1 [P/x]) et ((PI M_1) [Q/x]) = PI (M_1 [Q/x]). Par hypothèse de récurrence on déduit que (M_1 [P/x]) $\rightarrow_{E\lambda\beta}$ (M_1 [Q/x]). D'où PI (M_1 [P/x]) $\rightarrow_{E\lambda\beta}$ PI (M_1 [Q/x]).

Proposition 4 :

Pour tout termes M N N' et pour toute variable x y, si x ≠ y et x ∉ FV(N'), alors M [N/x][y/N'] = M[y/N'][x/N[y/N']].

La spécification formelle de cette proposition est :

Lemma sub_distrib: forall n1 n2 a b p q:nat, forall P Q M M1 M2 N1 N2 N3: c_term,

var(n1,n2) <> var(a,b) -> ~(is_free (var(a,b)) P) -> ((level P)<= n2)

->((level Q) <= b) -> (substitution (var(a,b)) Q M M1)

-> (substitution (var(n1,n2)) P M1 M2) -> (substitution (var(n1,n2)) P Q N1)

-> (substitution (var(n1,n2)) P M N2) -> (substitution (var(a,b)) N1 N2 N3)

-> M2 = N3.

La démonstration est par récurrence sur la structure de M ; Soient D et G les membres droit et gauche de l'égalité.

1. Si M ≡ x, alors G = N[y/N'] et D = N[y/N'] car x ≠ y.
2. Si M ≡ y, alors G = N' et D = N'[x/N[y/N']] = N ' car x ≠ y et x ∉ FV(N').
3. Si M ≡ z où x ≠ z et y ≠ z, alors G = z = D.
4. Si M ≡ λz. P ; on démontre ce cas par l'utilisation de la l'hypothèse de récurrence.
5. Si M ≡ App P Q ; on démontre ce cas par l'utilisation de l'hypothèse de récurrence.
6. Si M ≡ Imp P Q ; on démontre ce cas par l'utilisation de l'hypothèse de récurrence.
7. Si M ≡ PI Q ; on démontre ce cas par l'utilisation de la l'hypothèse de récurrence.

Proposition 5 :

Pour tout termes M X et pour toute variable x, si x ∉ FV(M) alors (M[x/X]) ≡ M

La spécification formelle de cette proposition dans le système coq est :

Lemma sub_var_nfree: forall M p q X, ((level X) <= q) -> ~(is_free (var(p,q)) M) -> (substitution (var(p,q)) X M M).

La preuve est par récurrence sur le terme M.

Proposition 6 :

Pour tout termes M NP Q :c_term et pour toute variable x, si (N[M/x]) ≡ P et (N[M/x]) ≡ Q alors P = Q.

La spécification formelle de cette proposition dans le système coq est :

Lemma subst_term_eq: forall M N P Q: c_term, forall n1 n2:nat,

(substitution (var(n1,n2)) M N P) -> (substitution (var(n1,n2)) M N Q)

-> P = Q.

La preuve est par récurrence sur le terme N.

Proposition 7 :

Pour tout termes M0 M1 M2 Z Z0 Z1 Z2 et pour toute variable x, si (M1[M0/x]≢ Z et (M2[M1/x]) ≡ Z0 et (Z0[M0/x]) ≡ Z1 et (M2[Z/x]) ≡ Z2 alors Z2 = Z1.

La spécification formelle de cette proposition dans le système coq est comme suit :

Lemma sub_eq_Z1_Z2: forall M2 Z Z0 Z1 Z2 M0 M1 n1 n2 , ((level M0) <= n2) ->

((level M1) <= n2) -> (substitution (var(n1,n2)) M0 M1 Z)

-> (substitution (var(n1,n2)) M1 M2 Z0) ->(substitution (var(n1,n2)) M0 Z0 Z1) ->

(substitution (var(n1,n2)) Z M2 Z2) -> Z2 = Z1.

La preuve est par récurrence sur la structure du terme M2 en utilisant le lemme sub_var_nfree et le lemme subst_term_eq.

Proposition 8 :

Pour tout termes M P Q P1 Q1 et pour toute variable x, si P$\not\rightarrow_{E\lambda\beta}$ Q et si (P[M/x])≡P1 et si (Q[M/x]) ≡ Q1 alors P1 $\rightarrow_{E\lambda\beta}$ Q1.

La spécification formelle de cette proposition dans le système coq est comme suit :

Lemma red_subst_red: forall P Q, (Ebeta_reduc P Q)-> forall M P1 Q1 n1 n2,

((level M) <= n2) -> (substitution (var(n1,n2)) M P P1)->

(substitution (var(n1,n2)) M Q Q1) -> (Ebeta_reduc P1 Q1).

La preuve de cette proposition est par induction sur la longueur da la réduction P$\rightarrow_{E\lambda\beta}$ Q, en utilisant les lemmes subst_term_eq, sub_var_nfree et le lemme sub_eq_Z1_Z2.

Proposition 9 :

Pour tout termes N P Q x1 x2 x3 x4 et pour toute variable x, si (Q[P/x]≢ x1 et (N[x1/x]) ≡ x2 et (N[Q/x]) ≡ x3 et (x3[P/x]) ≡ x4 alors x2 = x4.

La spécification formelle de cette proposition dans le système coq est comme suit :

Lemma sub_x2_eq_x4: forall Q N P x1 x2 x3 x4 p q, substitution (var(p,q)) P Q x1 ->

(substitution (var(p,q)) x1 N x2) -> (substitution (var(p,q)) Q N x3)

-> (substitution (var(p,q)) P x3 x4) -> x2 = x4.

La preuve de cette proposition est par double induction sur la structure des termes Q et N, en utilisant les lemmes sub_var_nfree et subst_term_eq.

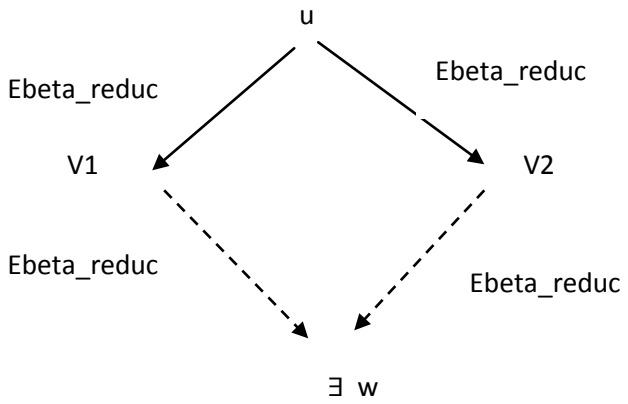
Après la présentation des lemmes intermédiaires les plus intéressants, on peut montrer la confluence de Elambda-beta-réduction.

3.7. La confluence d'Elambda-beta-réduction :

Dans cette section, on va démontrer que l'Elamda_beta_réduction (Ebeta_reduc) vérifie la propriété de Church-Rosser, c'est-à-dire :

Pour tout Eλ-terme u v_1 v_2, si u→ $_{Ebeta\_reduc}$ v_1 et u→ $_{Ebeta\_reduc}$ v_2 alors il existe un Eλ_term w tel que, v_1 → $_{Ebeta\_reduc}$ w et v_2 → $_{Ebeta\_reduc}$ w. ……▶ (1)

Pour démontrer ce théorème on utilise la méthode des réductions parallèles.



3.7.1. Résidus :

Soient P un Elambda_terme et R, S des beta_redex dans P. Lorsque le redex R est contracté, P se réduit à P'. **Les résidus** de S dans P' sont des redex définis comme suit :

Cas 1 : Si R et S sont disjoints dans P, alors lorsque R est contracté S reste inchangé. Donc le résidu de S dans P' est S.

Cas 2 : Si R ≡ S, alors contracter R est pareille à contracter S ; Donc S n'a pas de résidus dans P'.

Cas 3 : Si R est contenu dans S, c.-à-d. S a la forme (λx.M)N et R est dans M ou dans N. la réduction de R change M à M' ou N à N'. alors S se change à (λx.M')N ou (λx.M)N' qui est le résidu de S dans P' .

Cas 4 : Si S est contenu dans R, c.-à-d. R a la forme (λx.M)N et S est dans M ou dans N.

Sous cas 4a : S est dans M, Le résidu de S dans P' prend une des formes :

S [N/x], S [N/x] $[z_1/y]\ldots[z_n/y_n]$, S

Sous cas 4b: S est dans N, Lorsque le terme M [N/x] est construit ; le redex S est dupliqué en autant de copies qu'il y a d'occurrences libres de x dans M et qui sont les résidus de S dans P'.

3.7.2. Introduction à la réduction parallèle

La méthode de démonstration que l'on va utiliser est due à William Tait et Per Martin-Löf et a été encore simplifiée par Masako Takahashi en 1995. Avant de passer à la démonstration proprement dite, on va décrire informellement l'idée de cette méthode en analysant ce qui se passe mal avec l'Elambda-calcul.

Premièrement, on a songé à démontrer que la relation beta_red (réduction en une étape) est fortement confluente ; c.-à-d :

Pour tout Eλ-terme u v_1 v_2, si u $\rightarrow_{beta\_red}$ v_1 et u $\rightarrow_{beta\_red}$ v_2 alors il existe un Eλ-terme w tel que $v_1 \rightarrow_{beta\_red}$ w et $v_2 \rightarrow_{beta\_red}$ w .....▶ (2)

Si (2) est vérifiée, alors on peut facilement déduire (1). Mais malheureusement (2) n'est pas toujours vérifiée. Par exemple ; soit P ≡ (λy.uyy)(**I**z) avec **I** ≡ (λx.x) ;

P $\rightarrow_{beta\_red}$ u(**I**z)(**I**z) et P $\rightarrow_{beta\_red}$ (λy.uyy)z , et (λy.uyy)z $\rightarrow_{beta\_red}$ u z z, mais u (I z) (I z) ne peut être réduit à uzz par une seule étape. u (I z) (I z) peut être réduit à u z z par deux étapes disjointes.

On va donc chercher à construire une relation, notée MCD (qui représentera la réduction parallèle) qui soit contenu dans Ebeta_reduc et qui est fortement confluente. Par ailleurs, si on a MCD ⊂ $\rightarrow_{Ebeta\_reduc,}$ alors on constate immédiatement que $MCD^* = \rightarrow_{Ebeta\_reduc}$ et donc si on prouve la confluence de MCD suffira à obtenir le résultat souhaité.

En disant cela, on n'a évidemment pas résolu le problème, mais on a déjà fait un grand pas. Le second pas consiste à se demander ce qu'on va mettre dans MCD.

Soit M un Elambda-terme et $\{R_1, R2,\ldots R_i,\ldots R_n\}$ est l'ensemble des redex présents dans M. On dit que R_i est un redex minimal s'il ne contient aucun redex Rj de l'ensemble des redex de M. Le principe de la réduction parallèle est de commencer à contracter d'abord le redex minimal R_i puis contracter le résidu minimal de $R_1,\ldots$, Rn et refaire l'opération de réduction jusqu'à contracter tous les résidus.

Définition 3.9 : *La réduction parallèle*

La réduction parallèle, notée « $\rightarrow_{MCD}$ », est définie comme suit :

- Si x Є V_i alors $x \rightarrow_{MCD} x$
- Si $M \rightarrow_{MCD} M'$ et $N \rightarrow_{MCD} N'$ et $M'[N'/x] = P$, alors $(\lambda x.M)N \rightarrow_{MCD} P$
- Si $M \rightarrow_{MCD} M'$ et $N \rightarrow_{MCD} N'$, alors $M\ N \rightarrow_{MCD} M'\ N'$
- Si $M \rightarrow_{MCD} M'$ et $N \rightarrow_{MCD} N'$, alors $P(M\ N) \rightarrow_{MCD} P(M'\ N')$
- Si $M \rightarrow_{MCD} M'$, alors $\Pi\ M \rightarrow_{MCD} \Pi\ M'$.

La réduction parallèle est définie en coq de manière inductive comme suit :

Inductive MCD: c_term -> c_term -> Prop:=

|mcd_cvar: forall n1 n2:nat, (MCD (cvar(var(n1,n2))) (cvar(var(n1,n2))))

|mcd_redex: forall n1 n2:nat, forall M N M' N' P:c_term, (MCD M M') -> (MCD N N')

-> (substitution (var(n1,n2)) N' M' P)->(MCD (App(Abs(n1,n2)M)N) P)

|mcd_app: forall M N M' N':c_term, (MCD M M') -> (MCD N N') -> (MCD (App M N) (App M' N'))

|mcd_imp: forall M N M' N':c_term, (MCD M M') -> (MCD N N') -> (MCD (Imp M N) (Imp M' N'))

|mcd_abs: forall M M':c_term, forall n1 n2:nat, (MCD M M') -> (MCD (Abs(n1,n2)M) (Abs(n1,n2)M'))

|mcd_pi: forall M M':c_term, (MCD M M') -> (MCD (PI M) (PI M')).

Remarque préliminarie:

- Si $\rightarrow_{MCD}$ est fortement confluente, alors $\rightarrow_{MCD}$ est confluente

3.7.3. Démonstration du théorème de Church-Rosser:

Pour démontrer que la relation $\rightarrow_{Ebeta\_reduc}$ est confluente on doit démontrer les points suivants :

I. $\rightarrow_{beta\_red} \subset \rightarrow_{MCD} \subset \rightarrow_{Ebeta\_reduc}$.
II. $\rightarrow_{MCD}$ est fortement confluente.
III. Créer la clôture transitive et réflexive de la relation $\rightarrow_{MCD}$ qu'on va nommer $\rightarrow_{mcd\_star}$
IV. Démontrer que si $\rightarrow_{MCD}$ est fortement confluente alors $\rightarrow_{MCD}$ est confluente.
V. Démontrer que $\rightarrow_{mcd\_star} \subset \rightarrow_{Ebeta\_reduc}$ et $\rightarrow_{Ebeta\_reduc} \subset \rightarrow_{mcd\_star}$, c.-à-d. que la relation $\rightarrow_{mcd\_star}$ est équivalente à la relation $\rightarrow_{Ebeta\_reduc}$.

- Pour démontrer que $\rightarrow_{beta\_red} \subset \rightarrow_{MCD}$, on doit d'abord démontrer que la relation $\rightarrow_{MCD}$ est une relation réflexive.

Lemme 1: la relation $\rightarrow_{MCD}$ est une relation réflexive. La preuve de ce lemme est par induction sur la structure du terme M.

Son énoncé dans le système coq est comme suit :

```
Lemma mcd_ref: forall M:c_term, MCD M M.
Proof.
induction M0. induction e. induction p. apply (mcd_cvar a b).
apply (mcd_app M0_1 M0_2 M0_1 M0_2 IHM0_1 IHM0_2).
case p. intros. apply(mcd_abs M0 M0 n n0 IHM0).
apply (mcd_imp M0_1 M0_2 M0_1 M0_2 IHM0_1 IHM0_2).
apply(mcd_pi M0 M0 IHM0).
Qed.
```

3.7.3.1. La preuve de $\rightarrow_{beta\_red} \subset \rightarrow_{MCD} \subset \rightarrow_{Ebeta\_reduc}$:

Lemme 2: $\rightarrow_{beta\_red} \subset \rightarrow_{MCD}$. La preuve de ce lemme est par induction sur la longueur de la réduction $\rightarrow$*beta_red*. Son énoncé dans le système coq est :

```
Lemma beta_mcd: forall M N:c_term, (beta_red M N) -> (MCD M N).
intros.
inversion H. generalize(mcd_ref N1); intro. generalize(mcd_ref M1); intro.
```

apply(mcd_redex p q M1 N1 M1 N1 N0 H4 H3 H0).

Qed.

Lemme 3: $\rightarrow_{\mathbf{MCD}} \subset \rightarrow_{\mathbf{Ebeta\_reduc}}$. La preuve de ce lemme est par induction sur la longueur de la réduction $\rightarrow_{MCD}$ .Son énoncé dans le système coq est:

Lemma mcd_Ebeta: forall M N:c_term, (MCD M N) -> (Ebeta_reduc M N).

Proof.

induction 1. apply(refl_reduc (cvar(var(n3,n4)))).

generalize(Abs_reduc M0 M' n3 n4 IHMCD1); intro.

generalize(Appr_reduc (Abs(n3,n4)M0) (Abs(n3,n4)M') N0 H2); intro.

generalize(Appl_reduc N0 N' (Abs(n3,n4)M') IHMCD2); intro.

generalize(trans_reduc (App(Abs(n3,n4)M0)N0) (App(Abs(n3,n4)M')N0) (App(Abs(n3,n4)M')N') H3 H4); intro.

generalize(beta_red0 n3 n4 N' M' P H1); intro.

generalize(beta_reduc (App(Abs(n3,n4)M')N') P H6); intro.

apply(trans_reduc (App(Abs(n3,n4)M0)N0) (App(Abs(n3,n4)M')N') P H5 H7).

generalize(Appr_reduc M0 M' N0 IHMCD1); intro.

generalize(Appl_reduc N0 N' M' IHMCD2); intro.

apply(trans_reduc (App M0 N0) (App M' N0) (App M' N') H1 H2).

generalize(Impr_reduc M0 M' N0 IHMCD1); intro generalize(Impl_reduc N0 N' M' IHMCD2); intro.

apply(trans_reduc (Imp M0 N0) (Imp M' N0) (Imp M' N') H1 H2).

apply(Abs_reduc M0 M' n3 n4 IHMCD). apply(PI_reduc M0 M' IHMCD).

Qed.

- Pour démontrer que la relation $\rightarrow_{MCD}$ est fortement confluente, on doit d'abord démontrer le lemme suivant :

Lemme 4 : Si $M \rightarrow_{MCD} M'$ et $N \rightarrow_{MCD} N'$, alors $M[x/ N] \rightarrow_{MCD} M'[x/ N']$.

La preuve de ce lemme est par induction sur la structure du terme M, on se sert du lemme de la distributivité de la substitution (sub_distrib , le preuve de ce lemme est dans l'annexe B) et

les lemmes mcd_var_nfree et inv_mcd_redex où l'énoncé de preuve de ces lemmes est dans l'annexe C.

L'énoncé du lemme 4 dans le système coq est comme suit :

```
Lemma mcd_subst_mcd: forall P Q P' Q' P1 P2: c_term, forall n1 n2 q:nat,
 (level Q q) -> (q <= n2) -> (MCD P P') -> (MCD Q Q') ->
 (substitution (var(n1,n2)) Q P P1) ->   (substitution (var(n1,n2)) Q' P' P2) -> (MCD P1 P2).
Proof.
induction P. intros.
(********************************** cvar ********************************)
induction e. induction p.
inversion H1. rewrite <- H5 in H4. generalize (evar_eqdec n3 n4 a b); intro.
inversion H8.  rewrite H9 in H3; rewrite H9 in H4.
generalize (subst_var_eq Q P1 a b H3); intro. rewrite <- H10.
generalize (subst_var_eq Q' P2 a b H4); intro.
rewrite <- H11; exact H2. generalize (subst_var_neq Q P1 n3 n4 a b H9 H3); intro.
rewrite H10; clear H10. generalize (subst_var_neq Q' P2 n3 n4 a b H9 H4); intro.
rewrite H10; clear H10. apply mcd_cvar.
(************************************ App ********************************)
intros.  inversion H1. inversion H3.
generalize(exist_level Q'); intro. inversion_clear H21.
generalize(level_subst Q' P' P3 n3 n4 x0 H22 H4); intro.
generalize(exist_term N' Q' n3 n4 x0 H22 H21); intro. inversion_clear H23.
generalize(exist_term M' Q' n3 n4 x0 H22 H21); intro. inversion_clear H23.
generalize(exist_level x1); intro. inversion_clear H23.
generalize(exist_level N'); intro. inversion_clear H23.
generalize(level_subst N' M' P' n5 n6 x4 H27 H10); intro.
generalize(subst_level N' Q' x1 n3 n4 x0 x4 x3 H22 H27 H26 H21 H24); intro.
generalize(le_trans x3 x4 n6 H28 H23); intro. rewrite <- H5 in H19; inversion H19.
```

generalize(IHP2 Q N' Q' N2 x1 n3 n4 x H15 H17 H8 H2 H20 H24); intro.

generalize(exist_level x1); intro. inversion_clear H40.

generalize(subst_level N' Q' x1 n3 n4 x0 x4 x6 H22 H27 H41 H21 H24); intro.

rewrite <- H32 in H10; rewrite <- H35 in H10. rewrite <- H32; rewrite <- H35.

generalize(level_subst N' M' P' n3 n4 x4 H27 H10); intro.

assert(x6 <= n4). apply(le_trans x6 x4 n4 H40 H42).

generalize(exist_term M' x1 n3 n4 x6 H41 H43); intro. inversion_clear H44.

generalize(sub_x2_eq_x4 N' M' Q' x1 x7 P' P3 n3 n4 H24 H45 H10 H4); intro.

rewrite H44 in H45; clear H44. apply(mcd_redex n3 n4 M0 N2 M' x1 P3 H7 H39 H45).

rewrite <- H38 in H19. generalize(mcd_var_nfree Q Q' n5 n6 H2 H35); intro.

generalize(sub_abs_nfree1 n3 n4 n5 n6 M' x0 x2 Q' H42 H22 H21 H40 H25); intro.

rewrite <- H5 in IHP1. generalize(mcd_abs M0 M' n5 n6 H7); intro.

generalize(IHP1 Q (Abs(n5,n6)M') Q' (Abs(n5,n6)Z) (Abs(n5,n6)x2) n3 n4 q H H0 H44 H2 H19 H43); intro.

generalize(inv_mcd_redex Z x2 n5 n6 H45); intro.

generalize(IHP2 Q N' Q' N2 x1 n3 n4 q H H0 H8 H2 H20 H24); intro.

generalize(exist_term x2 x1 n5 n6 x3 H26 H29); intro. inversion_clear H48.

generalize(neq_var n3 n4 n5 n6 H40); intro.

generalize(sub_distrib n3 n4 n5 n6 x0 x4 Q' N' M' P' P3 x1 x2 x6 H48 H42 H22 H21 H27 H23 H10 H4 H24 H25 H49); intro.

rewrite <- H50 in H49. apply(mcd_redex n5 n6 Z N2 x2 x1 P3 H46 H47 H49).

generalize(neq_var n3 n4 n5 n6 H41); intro.

generalize(IHP2 Q N' Q' N2 x1 n3 n4 q H H0 H8 H2 H20 H24); intro.

generalize(exist_term x2 x1 n5 n6 x3 H26 H29); intro. inversion_clear H44.

generalize(mcd_var_nfree Q Q' n5 n6 H2 H40); intro.

generalize(subst_distrib n3 n4 n5 n6 x0 x4 Q' N' M' P' P3 x1 x2 x6 H42 H44 H22 H21 H27 H23 H10 H4 H24 H25 H45); intro.

rewrite <- H46 in H45.

generalize(sub_abs_nfree1 n3 n4 n5 n6 M' x0 x2 Q' H44 H22 H21 H41 H25); intro.

rewrite <- H5 in IHP1.

generalize(mcd_abs M0 M' n5 n6 H7); intro. rewrite <- H38 in H19.

generalize(IHP1 Q (Abs(n5,n6)M') Q' (Abs(n5,n6)M0) (Abs(n5,n6)x2) n3 n4 q H H0 H48 H2 H19 H47); intro.

generalize(inv_mcd_redex M0 x2 n5 n6 H49); intro.

apply(mcd_redex n5 n6 M0 N2 x2 x1 P3 H50 H43 H45).

inversion H3. rewrite <- H8 in H4. inversion H4.

generalize (IHP1 Q M' Q' M2 M4 n3 n4 q H H0 H7 H2 H18 H28); intro.

generalize (IHP2 Q N' Q' N2 N4 n3 n4 q H H0 H9 H2 H19 H29); intro.

apply (mcd_app M2 N2 M4 N4 H30 H31).

*(********** Abs **********)*

intros.

inversion H1. rewrite <- H5 in H3; rewrite <- H6 in H4.

inversion H3; inversion H4. rewrite <- H20 in H3; rewrite <- H23 in H3.

rewrite <- H20 in H4; rewrite <- H23 in H4.

generalize (mcd_abs P M' n5 n6 H8); intro. exact H27.

generalize (eq_couple n3 n4 n5 n6 H11 H14); intro.

generalize (neq_couple n3 n4 n5 n6 H28); intro. contradiction.

apply(mcd_abs P M' n5 n6 H8).

generalize (eq_couple n3 n4 n5 n6 H23 H26); intro.

generalize (neq_couple n3 n4 n5 n6 H19); intro. contradiction.

generalize (IHP Q M' Q' Z Z0 n3 n4 q H H0 H8 H2 H20 H32); intro.

apply (mcd_abs Z Z0 n5 n6 H33).

generalize(sub_var_nfree M' n3 n4 x0 Q' H28 H30 H26); intro.

generalize(IHP Q M' Q' Z M' n3 n4 q H H0 H8 H2 H20 H33); intro.

apply(mcd_abs Z M' n5 n6 H34). apply(mcd_abs P M' n5 n6 H8).

generalize(mcd_var_nfree P M' n3 n4 H8 H14); intro.

generalize(sub_var_nfree M' n3 n4 x0 Q' H28 H30 H33); intro.

generalize(subst_term_eq Q' M' Z M' n3 n4 H32 H34); intro.

rewrite H35; rewrite <- H5 in H1; rewrite <- H6 in H1; assumption.

```
rewrite <- H5 in H1; rewrite <- H6 in H1; assumption.
(********************************** Imp ************************************)
intros.
inversion H1. rewrite <- H8 in H4. inversion H3. inversion H4.
generalize (IHP1 Q M' Q' M2 M4 n3 n4 q H H0 H7 H2 H18 H28); intro.
generalize (IHP2 Q N' Q' N2 N4 n3 n4 q H H0 H9 H2 H19 H29); intro.
apply mcd_imp; assumption.
(********************************** PI ************************************)
intros.
inversion H1. rewrite <- H7 in H4; clear H7. inversion H3; inversion H4.
generalize (IHP Q M' Q' N0 N1 n3 n4 q H H0 H6 H2 H14 H22); intro.
apply mcd_pi; exact H23.
Qed.
```

Maintenant, on peut donner la preuve que la relation $\twoheadrightarrow_{MCD}$ est fortement confluente, on se sert des lemmes qu'on a démontré précédemment.

3.7.3.2. La relation $\rightarrow_{MCD}$ est fortement confluente :

Lemme 5 : Pour tout Eλ-terme P X Y, Si P $\rightarrow_{MCD}$ X et P $\rightarrow_{MCD}$ Y alors il existe un Eλ-terme T, tel que X $\rightarrow_{MCD}$ T et Y $\rightarrow_{MCD}$ T

La preuve de ce lemme est par induction sur la structure du terme P. l'énoncé du ce lemme dans le système Coq est comme suit :

```
Lemma MCD_strong_conf: forall P X Y: c_term, (MCD P X) -> (MCD P Y)->
    (exists T:c_term, (MCD X T) /\ (MCD Y T)).
Proof.
(********************************** cvar ************************************)
induction P. induction e. induction p.
intros. generalize (MCD_cvar a b X H); intro.
```

generalize (MCD_cvar a b Y H0); intro. rewrite H1; rewrite H2.

exists (cvar(var(a,b))). split; repeat apply mcd_cvar.

*(********** App **********)*

intros.

inversion H; inversion H0. rewrite <- H1 in IHP1.

rewrite <- H7 in H1; inversion H1. rewrite <- H16 in H9.

generalize(mcd_abs M0 M' n3 n4 H3); intro. generalize(mcd_abs M0 M'0 n3 n4 H9); intro.

generalize(IHP1 (Abs (n3, n4) M') (Abs (n3, n4) M'0) H13 H17); intro.

inversion_clear H18. generalize(IHP2 N' N'0 H4 H10); intro.

inversion_clear H18. inversion_clear H19; inversion_clear H20.

inversion H18; inversion H21. rewrite <- H28 in H23. inversion H23. rewrite <- H33 in H31.

generalize(exist_level N'); intro. inversion_clear H32.

generalize(exist_level x0); intro. inversion_clear H32.

generalize(level_subst N' M' X n3 n4 x1 H34 H6); intro. generalize (mcd_Ebeta N' x0 H19); intro.

generalize(red_level_decr N' x0 H36 x1 x2 H34 H35); intro.

generalize(le_trans x2 x1 n4 H37 H32); intro. generalize(exist_term M'1 x0 n3 n4 x2 H35 H38); intro.

inversion_clear H39. rewrite <- H14 in H12; rewrite <- H15 in H12.

generalize(exist_level N'0); intro. inversion_clear H39.

generalize(level_subst N'0 M'0 Y n3 n4 x4 H41 H12); intro.

generalize(mcd_subst_mcd M' N' M'1 x0 X x3 n3 n4 x1 H34 H32 H26 H19 H6 H40); intro.

generalize(mcd_subst_mcd M'0 N'0 M'1 x0 Y x3 n3 n4 x4 H41 H39 H31 H22 H12 H40); intro.

exists x3; split; assumption.

rewrite <- H1 in H9; inversion H9.rewrite <- H13 in H10. rewrite <- H1 in IHP1.

generalize(mcd_abs M0 M' n3 n4 H3); intro. generalize(mcd_abs M0 M'1 n3 n4 H16); intro.

generalize(IHP1 (Abs(n3,n4)M') (Abs(n3,n4)M'1) H17 H18); intro.

inversion_clear H19. inversion_clear H20. inversion H19; inversion H21.

rewrite <- H22 in H27. inversion H27. rewrite H32 in H30. clear H32.

generalize(IHP2 N' N'0 H4 H11); intro.

inversion_clear H31. inversion_clear H32.

generalize(exist_level N'); intro. inversion_clear H32.

generalize(exist_level x0); intro. inversion_clear H32.

generalize(exist_level N'0); intro. inversion_clear H32.

generalize(level_subst N' M' X n3 n4 x1 H34 H6); intro.

generalize(mcd_Ebeta N' x0 H31); intro.

generalize(red_level_decr N' x0 H37 x1 x2 H34 H35); intro.

assert(x2 <= n4). apply(le_trans x2 x1 n4 H38 H32).

generalize(exist_term M'2 x0 n3 n4 x2 H35 H39); intro. inversion_clear H40. exists x4; split.

apply(mcd_subst_mcd M' N' M'2 x0 X x4 n3 n4 x1 H34 H32 H25 H31 H6 H41).

apply(mcd_redex n3 n4 M'1 N'0 M'2 x0 x4 H30 H33 H41). rewrite <- H6 in H3; inversion H3.

generalize(mcd_abs M1 M'0 n3 n4 H8); intro.

generalize(mcd_abs M1 M'1 n3 n4 H16); intro. rewrite <- H6 in IHP1.

generalize(IHP1 (Abs (n3, n4) M'0) (Abs (n3, n4) M'1) H17 H18); intro.

inversion_clear H19. inversion_clear H20. inversion H19; inversion H21.

rewrite <- H27 in H22; inversion H22. rewrite <- H32 in H30; clear H32.

generalize(IHP2 N' N'0 H5 H9); intro. inversion_clear H31. inversion_clear H32.

generalize(exist_level N'); intro. inversion_clear H32. generalize(exist_level x0); intro.

inversion_clear H32. generalize(exist_level N'0); intro. inversion_clear H32.

generalize(level_subst N'0 M'0 Y n3 n4 x3 H36 H11); intro. generalize(mcd_Ebeta N'0 x0 H33); intro.

generalize(red_level_decr N'0 x0 H37 x3 x2 H36 H35); intro.

assert(x2 <= n4). apply(le_trans x2 x3 n4 H38 H32).

generalize(exist_term M'2 x0 n3 n4 x2 H35 H39); intro.

inversion_clear H40. exists x4; split. apply(mcd_redex n3 n4 M'1 N' M'2 x0 x4 H30 H31 H41).

apply(mcd_subst_mcd M'0 N'0 M'2 x0 Y x4 n3 n4 x3 H36 H32 H25 H33 H11 H41).

generalize (IHP1 M' M'0 H3 H8); intro. generalize (IHP2 N' N'0 H5 H10); intro.

elim H11; intros; elim H12; intros. elim H13; intros; elim H14; intros.

exists (App x x0); split; apply mcd_app; [exact H15 |exact H17 | exact H16 |exact H18].

*(********** Abs **********)*

intros. induction p. inversion H. inversion H0. generalize (IHP M' M'0 H5 H10); intro.

elim H11; intros. elim H12; intros. generalize (mcd_abs M' x a b H13); intro.

generalize (mcd_abs M'0 x a b H14); intro. exists (Abs(a,b)x); split; [exact H15 |exact H16].

*(********** Imp **********)*

intros. inversion H; inversion H0.

generalize (IHP1 M' M'0 H3 H8); intro. generalize (IHP2 N' N'0 H5 H10); intro.

elim H11; intros; elim H12; intros. elim H13; intros; elim H14; intros.

exists (Imp x x0); split. apply mcd_imp; [exact H15 | exact H17].

apply mcd_imp; [exact H16 | exact H18].

(********** PI **********)

intros. inversion H; inversion H0. generalize (IHP M' M'0 H2 H5); intro.

elim H7; intros. elim H8; intros. exists(PI x); split. apply mcd_pi; exact H9.

apply mcd_pi; exact H10.

Qed.

Maintenant, c'est le moment de créer la fermeture transitive et réflexive de la relation $\rightarrow_{MCD}$ et de démontrer que $\rightarrow_{MCD}$ est confluente.

3.7.3.3. Créer la clôture transitive et réflexive de la relation $\rightarrow_{MCD}$

La clôture transitive et réflexive de la relation MCD est nommée mcd_star. Elle est définit dans le système Coq comme suit :

Inductive mcd_star: c_term -> c_term -> Prop:=

|refl_red: forall x:c_term, mcd_star x x

|trans_red: forall x y z:c_term, MCD x y -> mcd_star y z -> mcd_star x z.

On définit quelques fonctions qui nous aident pour démontrer que la relation $\rightarrow_{mcd\_star}$ est confluente.

Definition coherent (x y:c_term): Prop:=

exists z:c_term, mcd_star x z /\ mcd_star y z.

Definition confluent (x:c_term): Prop:=

forall y z:c_term, mcd_star x y -> mcd_star x z -> coherent y z.

Definition Confluent: Prop:= forall x:c_term, confluent x.

Definition strongly_confluent(x:c_term): Prop:=

forall y z:c_term, MCD x y -> MCD x z -> exists w:c_term, MCD y w /\ MCD z w.

Definition Strongly_confluent: Prop:= forall x:c_term, strongly_confluent x.

3.7.3.4. Démontrer que la relation $\rightarrow_{MCD}$ est confluente :

Lemme 6 : Si la relation$\rightarrow_{MCD}$ est fortement confluente, alors$\rightarrow_{MCD}$ est confluente c.-à-d. : Si (Pour tout Eλ-terme P X Y, Si P $\rightarrow_{MCD}$ X et P $\rightarrow_{MCD}$ Y alors il existe un Eλ-terme T, tel que X $\rightarrow_{MCD}$ T et Y $\rightarrow_{MCD}$ T) alors (Pour tout Eλ-terme P X Y, Si P $\rightarrow_{mcd\_star}$ X et P $\rightarrow_{mcd\_star}$ Y alors il existe un Eλ-terme T, tel que X $\rightarrow_{mcd\_star}$ T et Y $\rightarrow_{mcd\_star}$ T)

L'énoncé de ce lemme dans le système Coq est comme suit :

Lemma strog_conf_conf: Strongly_confluent -> Confluent.

Proof.

*intro H'; red in |- *. intro x; red in |- *. intros y z H'0.*

unfold coherent. red in H'.

generalize z; clear z.

elim H'0; clear H'0. intros x0 z H'1. exists z. split; [assumption | apply refl_red].

intros x0 y0 z H'1 H'2 H'3 z0 H'4. cut (ex (fun t:c_term => mcd_star y0 t /\ MCD z0 t)).

intro h; elim h. intros t h0; elim h0. intros H'0 H'5; clear h h0. generalize (H'3 t); intro h. lapply h;

[intro h0; elim h0; intros z1 h1; elim h1; intros H'6 H'7; clear h h0 h1 | clear h]; auto with sets.

exists z1; split; [assumption | idtac]. apply trans_red with t; auto with sets.

generalize H'1; generalize y0; clear H'1. elim H'4.

intros x1 y1 H'0; exists y1; split; [apply refl_red | assumption].

intros x1 y1 z1 H'0 H'1 H'5 y2 H'6. red in H'.

generalize (H' x1 y1 y2); intro h; lapply h;

[intro H'7; lapply H'7;

[intro h0; elim h0; intros z2 h1; elim h1; intros H'8 H'9; clear h H'7 h0 h1 | clear h]

| clear h]; auto with sets.

generalize (H'5 z2); intro h; lapply h;

[intro h0; elim h0; intros t h1; elim h1; intros H'7 H'10; clear h h0 h1 | clear h]; auto with sets.

exists t; split; auto with sets.

apply trans_red with z2; auto with sets.

Qed.

3.7.3.5. L'équivalence entre →Ebeta_reduc et →mcd_star :

Finalement, on est arrivé à démontrer que la relation→mcd_star est équivalente à la relation →Ebeta_reduc. Pour cela on a besoin de démontrer d'autres lemmes intermédiaires.

L'énoncé de preuves de ces lemmes dans le système coq se trouve dans l'annexe C :

Proposition1 : Pour tout termes M N Z, si M →mcd_star N alors (App Z M) →mcd_star (App Z N)

Proposition 2: Pour tout termes M N Z, si M →mcd_star N alors (App M Z) →mcd_star (App N Z).

Proposition3 : Pour tout termes M N Z, si M →mcd_star N alors (Imp Z M) →mcd_star (Imp Z N).

Proposition4 : Pour tout termes M N Z, si M →mcd_star N alors (Imp M Z) →mcd_star (Imp N Z).

Proposition 5 : Pour tout termes M et N et pour toute variable x, si M →mcd_star N alors (Abs x. M) →mcd_star (Abs x. N).

Proposition 6 : Pour tout terme M, si M →mcd_star N, alors (PI M) →mcd_star (PI N).

Proposition 7 : Pour tout termes M N, si M $\rightarrow_{MCD}$ N, alors M →mcd_star N.

Proposition 8 : Pour tout termes M N P, si M →mcd_star N et N →mcd_star P alors

M →mcd_star P.

3.7.3.5.1. L'énoncé du lemme démontrant que la relation →mcd_star ⊂ →Ebeta_reduc est comme suit :

Lemma contain_mcd_star_Ebeta: forall x y:c_term, mcd_star x y -> Ebeta_reduc x y.

Proof.

induction 1; intros. apply (refl_reduc x).

generalize(mcd_Ebeta x y H); intro.

apply(trans_reduc x y z H1 IHmcd_star).

Qed.

3.7.3.5.2. L'énoncé du lemme démontrant que la relation →Ebeta_reduc ⊂ →mcd_star est comme suit :

Lemma contain_Ebeta_mcd_star: forall x y:c_term, Ebeta_reduc x y -> mcd_star x y.

Proof.

induction 1; intros.

apply (refl_red M0).

apply (mcd_star_appl M0 N0 IHEbeta_reduc Z).

apply(mcd_star_appr M0 N0 IHEbeta_reduc Z).

apply (mcd_star_impl M0 N0 IHEbeta_reduc X).

apply(mcd_star_impr M0 N0 IHEbeta_reduc X).

generalize(beta_mcd M0 N0 H); intro.

apply(inc_mcd_mcd_star M0 N0 H0).

apply(mcd_star_abs M0 N0 IHEbeta_reduc p q).

apply(mcd_star_pi M0 N0 IHEbeta_reduc).

apply (mcd_star_trans M0 N0 T IHEbeta_reduc1 IHEbeta_reduc2).

Qed.

D'où d'après le lemme contain_mcd_star_Ebeta et le lemme contain_Ebeta_mcd_star, on déduit que la relation →Ebeta_reduc est équivalente à la relation →mcd_star. Donc la relation →Ebeta_reduc est confluente.

Conclusion :

L'objectif de ce mémoire est de formaliser le système Elambda en définissant la notion de termes, la relation de substitution et la relation de réduction(Ebeta_reduc) et les règles d'inférence. Ainsi de démontrer qu'Ebeta_reduc est confluente, en utilisant l'assistant de preuve Coq.

Dans Coq, programmer revient à écrire des spécifications et de démontrer des propriétés. Il s'agit d'une nouvelle méthodologie de la programmation, qui peut sembler déroutante au premier abord, mais elle offre un avantage important car elle garantit l'absence d'erreurs dans le programme obtenu.

La preuve de la spécification a nécessité plusieurs lemmes intermédiaires. L'introduction de ces résultats intermédiaires permet d'éviter de démontrer la même propriété plusieurs fois. La preuve du résultat principal est ainsi écourtée. Elle est plus lisible et simple à comprendre. Pour démontrer la confluence d' Ebeta_reduc, on a suivi ces étapes :

- On a créé une nouvelle relation MCD qui réduit tout les redex présents dans un terme (réduction parallèle) et on a démontré que cette nouvelle relation est fortement confluente.
- On a crée la fermeture réflexive et transitive de la relation MCD, nommée mcd_star d'une manière qu'elle soit équivalente à la relation Ebeta_reduc.
- On a démontré que MCD est confluente. Donc Ebeta_reduc est confluente.

Bibliographie

[1]. H. Barendregt. The lambda-calculus : Its syntax and semantics Series "studies in logic", North Holland Co. Amsterdam (1984).

[2]. H.P. Barendregt. The impact of the lambda calculus in logic and computer science. (1997). Available by ftp.

[3]. Y. Bertot, P. Castéran. Le Coq' Art (V8). 26 janvier 2011.

[4]. M.W Bunder. The inconsistency of higher order predicate calculus and set theory based on combinatory logic. Mathematical logic in Latin America(proceedings of the fourth symposium) A.I. Arruda et al.(Editors) North Holland company Amsterdam 1980, pp 99-107.

[5]. M.W Bunder. Some consistency proofs and a charaterization of inconsistency proofs in illative combinatory logic, The Journal of Symbolic Logic, Volume 52, Number 1 (1987) pp 89-110.

[6]. M. W. Bunder. A paradox in illative combinatory logic. Notre Dame Journal of Formal Logic;Volume XI, Number 4, October 1970.

[7]. W. Carnielli, M.Coniglio, J.Marcos. Logics of formal inconsistency. Handbook of philosophical logic; 2nd edition, v.14, 2005.

[8]. M. Chaabani. Spécification, Preuve et Extraction Automatique des programmes en Coq ; Cas : l'algorithme $\lambda c\beta+$ - réduction. Mémoire de magister ; Université de Boumerdes.

[9]. H. Curry, R. Hindley and J. Seldin. Combinatory logic II. North Holland compagny Amsterdam, 1972.

[10]. L. Czajka, Higher-order illative combinatory logic. Institute of informatics, University of Warsaw. The Journal of Symbolic Logic 2013.

[11]. R. David. Une preuve simple de résultats simple en λ-calcul. Comptes Rendus de l Académie des Sciences - Series I - Mathematics 320 (1995) p 1401-1406.

[12]. W. Dekkers, M. Bunder, H.P. Barendregt. Completeness of two systems of illative combinatory logic for first-order propositional and predicate calculus. The Journal of Symbolic logic (1997).

[13]. C. Dubois. Les programmes vus comme des preuves, les types vus comme des propositions. Notes de cours DEA Informatique DRAFT, Octobre 2001.

[14]. K. Grue. Map Theory, Theoretical computer science 101 (1990).

[15]. R. Hindley and J. Seldin. Introduction to combinators and lambda-calculus. Cambridge University Press (1986).

[16]. C. Houtmann. Représentation et intéraction des preuves en super-déduction modulo. Thèse de doctorat ; Université Henri Poincaré - Nancy 1.

[17]. W.A. Howard the formulae-as-types notion of construction. In Hindley and seldin, editor, To H.B curry : Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479-490.Academic Press,London, 1980.

[18]. J.-L.Krivine . Lambda-calcul types et modèles. Etudes et recherche en informatique. Masson, 1990.

[19]. Markov, Essai de construction d'une logique de la mathématique constructive; Revue internationale de philosophie 25eme année 1971 fasicule 4 pp.477-507.

[20]. E. Mendelson. Introduction to Mathematical Logic, van Nostrand New York 1967.

[21]. M.Mezghiche. Logique constructive, spécification et programmation sûre. Notes de cours ; Laboratoire d'Informatique Fondamentale et Appliquée, Université de Boumerdès, 2008.

[22]. M.Mezghiche, C.Ben yelles. E-lambda-calculus: An illative combinatory logic interpreting higher order logic. Workshop on 35 years of Automath Heriot-Watt University, Edinburgh, April 2002.

[23]. C.Riba. Définitions par réécriture dans le lambda-calcul : confluence, réductibilité et typage. Thèse de doctorat ; Ecole doctorale IAEM Loraine, Université de Nancy.

[24]. The Coq Development Team.The standard library. February 8, 2007; Version v8.1.

[25]. The Coq Development Team. Reference Manual.December 19, 2011;Version 8.3pl3.

[26]. The Coq Development Team. The Coq reference manual. LogiCal Project, http://coq.inria.fr/.

Annexe A

- **Le lemme démontrant que chaque terme possède un niveau (level)**

Lemma exist_level: forall N, exists n, level N n.

Proof.

induction N.

case e. intro p. case p; intros.

exists n0. apply cvar_level.

elim IHN1. intros.

elim IHN2. intros.

exists (max x x0).

apply (App_level x x0 N1 N2 H H0).

case p.

elim IHN; intros.

exists (max x n0).

apply (Abs_level n n0 x N H).

elim IHN1; intros.

elim IHN2; intros.

exists (S (max x x0)).

apply (Imp_level x x0 N1 N2 H H0).

elim IHN; intros.

exists (max 1 x).

apply (PI_level x N H).

Qed.

- **Le lemme démontrant que chaque terme possède un niveau unique :**

Lemma ident_level: forall X: c_term, forall m n:nat, (level X = m) -> (level X = n) ->

(m = n).

Proof.

induction X; intros m n H0 H1; omega.

Qed.

- **Le lemme démontrant que chaque terme M peut avoir des occurrences libres ou liées d'une variable x**

Lemma occur_lib_lié: forall (M:c_term) (n1 n2:nat), (is_free (var(n1,n2))M) \/ ~(is_free (var(n1,n2)) M).

Proof.

induction M. intros.

case e; intro. case p; intros.

generalize(couple_eqdec n1 n2 n n0); intro.

inversion_clear H. inversion_clear H0.

left. apply (cvar_free n n0). generalize(inv_neq_couple n1 n2 n n0 H0); intro.

right. red. intro. inversion H1. generalize(eq_couple n1 n2 n n0 H5 H6); intro.

generalize(neq_couple n1 n2 n n0 H); intro. contradiction.

intros. elim (IHM1 n1 n2); elim (IHM2 n1 n2). intros. left.

generalize (App_free n1 n2 M1 M2); intro.

apply H1; right; exact H. intros; left.

generalize (App_free n1 n2 M1 M2); intro.

apply H1; left; exact H0. intros. left.

generalize (App_free n1 n2 M1 M2); intro. apply H1; right; exact H. intros; right.

unfold not. intro. inversion H1. inversion H3; contradiction.

intros. case p. intros. generalize(couple_eqdec n1 n2 n n0); intro.

inversion_clear H. inversion H0.

right. unfold not; intro. inversion H.

generalize(ref_eq_couple n n0); intro.

generalize(neq_couple n n0 n n0 H6); intro. contradiction.

generalize(inv_neq_couple n1 n2 n n0 H0); intro.

elim (IHM n1 n2); intro. left; apply (Abs_free n1 n2 n n0 M H H1).

right; red;intro. inversion H2. contradiction. intros. elim (IHM1 n1 n2); elim (IHM2 n1 n2).

intros; left. generalize(Imp_free n1 n2 M1 M2); intro. apply H1. left; exact H0.

intros; left. generalize(Imp_free n1 n2 M1 M2); intro.

apply H1. left; exact H0. intros; left.

generalize(Imp_free n1 n2 M1 M2); intro. apply H1. right; exact H.

intros; right. red; intro. inversion H1. inversion H3; contradiction.

intros. elim (IHM n1 n2); intro. left; apply (PI_free n1 n2 M H). right; red;intro.

inversion H0. case (H H2).

Qed.

Annexe B

Des lemmes démontrant la compatibilité entre la substitution et la réduction :

Lemme1 : (Proposition 2 de chapitre 3)

```
Lemma ebeta_red_level_decr: forall M N:c_term, (Ebeta_reduc M N) ->
    (level N ) <= (level M ) .
 Proof.
induction 1.
(**********************reflexivity**********************************)
intros. generalize (ident_level M p q H H0); intro; omega.
 (*************************** application à gauche ********************)
intros. inversion_clear H0. inversion_clear H1.
generalize (ident_level Z m m0 H2 H0); intro. rewrite <- H1; clear H1.
apply (max_m_n m n0 m n). apply le_refl. apply IHEbeta_reduc; assumption.
(************************** application à droite********************)
intros. inversion_clear H0; inversion_clear H1.
generalize (ident_level Z n n0 H3 H4); intro. rewrite <- H1; clear H1.
apply (max_m_n m0 n m n). apply IHEbeta_reduc; assumption. apply le_refl.
(************************ implication à gauche**********************)
intros. inversion_clear H0; inversion_clear H1.
generalize (ident_level X m m0 H2 H0); intro. rewrite <- H1; clear H1.
apply le_n_S. apply max_m_n. apply le_refl. apply IHEbeta_reduc; assumption.
(*************************** implication à droite********************)
intros. inversion_clear H0; inversion_clear H1.
generalize (ident_level X n n0 H3 H4); intro. rewrite <- H1; clear H1.
apply le_n_S. apply max_m_n. apply IHEbeta_reduc; assumption. apply le_refl.
(****************** beta réduction**********************************)
intros. inversion H. rewrite <- H3 in H0. inversion H0. inversion H7.
```

generalize (level_subst N0 M1 N p0 q0 n H9 H2). intro.

assert (max (max m0 q0) n = max m0 q0). apply assoc. assumption.

rewrite H16. generalize (subst_level M1 N0 N p0 q0 n m0 q H9 H14 H1 H15 H2); intro.

apply max_m_l. assumption.

*(********** Abs**********)*

intros. inversion H0; inversion H1.

apply (max_m_n m0 q m q). apply IHEbeta_reduc; assumption. apply le_refl.

*(**********PI**********)*

intros. inversion H0; inversion H1. apply (max_m_n 1 m0 1 m).

apply le_refl. apply IHEbeta_reduc; assumption.

*(**********transitivité**********)*

intros. assert (exists n, level N n). apply (exist_level N).

inversion_clear H3. generalize(IHEbeta_reduc1 p x H1 H4); intros.

generalize(IHEbeta_reduc2 x q H4 H2); intros. omega.

Qed.

Lemme2 : (Proposition 3 de chapitre 3)

Lemma Ebeta_subst_Ebeta: forall P Q M P1 Q1:c_term, forall n1 n2:nat, (Ebeta_reduc P Q)

-> (substitution (var(n1,n2)) P M P1) -> (substitution (var(n1,n2)) Q M Q1) -> (Ebeta_reduc P1 Q1).

Proof.

induction M; intros.

*(********** cvar **********)*

inversion H0; inversion H1. rewrite <- H7; rewrite <- H14. exact H.

rewrite <- H11 in H4. inversion H4. generalize(eq_couple n1 n2 m n H18 H19); intro.

generalize(neq_couple n1 n2 m n H16); intro. contradiction.

rewrite <- H4 in H12; inversion H12.

generalize(eq_couple n1 n2 m n H18 H19); intro.

generalize(neq_couple n1 n2 m n H9); intro. contradiction.

rewrite <- H4 in H12; inversion H12. apply refl_reduc.

*(****************************** application ******************************)*

inversion H0. inversion H1. generalize (IHM1 M0 M4 n1 n2 H H10 H20). intro.

generalize (IHM2 N1 N2 n1 n2 H H11 H21). intro.

generalize (Appl_reduc N1 N2 M0 H23). intro.

generalize (Appr_reduc M0 M4 N2 H22). intro.

generalize (trans_reduc (App M0 N1) (App M0 N2) (App M4 N2) H24 H25); intro.

assumption.

*(****************************** Abs ***********************************)*

induction p. generalize (couple_eqdec n1 n2 a b). intro. inversion H2. inversion H3.

rewrite <- H5 in H0;rewrite <- H6 in H0. rewrite <- H5 in H1; rewrite <- H6 in H1.

inversion H0; inversion H1. apply refl_reduc. generalize(ref_eq_couple n1 n2); intro.

generalize(neq_couple n1 n2 n1 n2 H23); intro. contradiction. apply refl_reduc.

generalize(ref_eq_couple n1 n2); intro. generalize(neq_couple n1 n2 n1 n2 H16); intro.

contradiction. generalize(ref_eq_couple n1 n2); intro.

generalize(neq_couple n1 n2 n1 n2 H16); intro. contradiction.

generalize(ref_eq_couple n1 n2); intro. generalize(neq_couple n1 n2 n1 n2 H16); intro.

contradiction. apply refl_reduc. generalize(ref_eq_couple n1 n2); intro.

generalize(neq_couple n1 n2 n1 n2 H28); intro. contradiction. apply refl_reduc.

inversion H0; inversion H1. apply refl_reduc.

generalize(eq_couple n1 n2 a b H6 H9); intro. contradiction. apply refl_reduc.

generalize(eq_couple n1 n2 a b H18 H21); intro.

generalize(neq_couple n1 n2 a b H14); intro. contradiction.

generalize (IHM Z Z0 n1 n2 H H15 H27); intro. apply (Abs_reduc Z Z0 a b H28).

generalize(sub_var_nfree M n1 n2 x P H11 H13 H21); intro.

generalize(subst_term_eq P M Z M n1 n2 H15 H28); intro. rewrite H29; apply refl_reduc.

apply refl_reduc. generalize(sub_var_nfree M n1 n2 x0 Q H23 H25 H9); intro.

generalize(subst_term_eq Q M Z M n1 n2 H27 H28); intro. rewrite H29; apply refl_reduc.

apply refl_reduc.

*(********** Implication **********)*

inversion H0. inversion H1. generalize (IHM1 M0 M4 n1 n2 H H10 H20); intro.

generalize (IHM2 N1 N2 n1 n2 H H11 H21); intro.

generalize (Impr_reduc M0 M4 N1 H22); intro.

generalize (Impl_reduc N1 N2 M4 H23); intro.

generalize(trans_reduc (Imp M0 N1) (Imp M4 N1) (Imp M4 N2) H24 H25);intro.

assumption.

*(********** PI **********)*

inversion H0. inversion H1. generalize (IHM N N0 n1 n2 H H9 H17); intro.

generalize (PI_reduc N N0 H18); intro. exact H19.

Qed.

Lemme 3: (Proposition 4 de chapitre 3)

Lemma sub_distrib: forall n1 n2 a b: nat, forall P Q M M1 M2 N1 N2 N3: c_term,

var(n1,n2) <> var(a,b) -> ~(is_free (var(a,b)) P) ->((level P) <= n2)

->((level Q) <= b) -> (substitution (var(a,b)) Q M M1)

-> (substitution (var(n1,n2)) P M1 M2) -> (substitution (var(n1,n2)) P Q N1)

-> (substitution (var(n1,n2)) P M N2) -> (substitution (var(a,b)) N1 N2 N3)

-> M2 = N3.

Proof.

induction M. intros.

*(********** cvar **********)*

induction e. induction p0. inversion H5.

rewrite <- H12 in H5; rewrite <- H15 in H5.

rewrite <- H12 in H8; rewrite <- H15 in H8.

rewrite <- H14 in H5; rewrite <- H14 in H6. inversion H8.

generalize (eq_var n1 n2 a b H20 H23); intro. contradiction.

generalize (subst_term_eq P Q M2 N1 n1 n2 H6 H7); intro.

rewrite <- H24 in H9; rewrite <- H27 in H9. assert(exists u:nat, level M2 u).

apply(exist_level M2). inversion_clear H28.

generalize(level_subst M2 (cvar(var(a,b))) N3 a b x1 H29 H9); intro.

generalize(sub_var_eq a b x1 M2 H29 H28); intro.

generalize(subst_term_eq M2 (cvar(var(a,b))) N3 M2 a b H9 H30); intro.

rewrite H31; reflexivity. rewrite <- H16 in H6; inversion H6. rewrite <- H23; rewrite <- H23 in H6.

generalize(subst_term_eq P (cvar(var(a0,b0))) P N2 n1 n2 H6 H8); intro.

rewrite <- H27 in H9. assert(exists u:nat, level N1 u). apply(exist_level N1). inversion_clear H28.

generalize(level_subst N1 P N3 a b x1 H29 H9); intro. generalize(sub_var_nfree P a b x1 N1 H29 H28 H0); intro.

generalize(subst_term_eq N1 P N3 P a b H9 H30); intro. rewrite H31; reflexivity.

generalize(subst_term_eq P (cvar(var(a0,b0))) M2 N2 n1 n2 H6 H8); intro.

rewrite <- H25 in H28. rewrite <- H28 in H9. assert(exists u:nat, level N1 u).

apply(exist_level N1). inversion_clear H29.

generalize(level_subst N1 (cvar(var(a0,b0))) N3 a b x1 H30 H9); intro.

generalize(sub_var_neq a b a0 b0 x1 N1 H30 H29 H18); intro.

generalize(subst_term_eq N1 (cvar(var(a0,b0))) N3 (cvar(var(a0,b0))) a b H9 H31); intro.

rewrite H32; reflexivity.

*(****************************** App ************************************)*

intros. inversion H5. rewrite <- H17 in H6. clear H17. inversion H6.

inversion H8. rewrite <- H36 in H9; clear H36. inversion H9.

```
generalize (IHM1 M4 M6 N1 M8 M10 H H0 H1 H2 H3 H4 H18 H27 H7 H37 H46); intro.
rewrite <- H48. clear H48.
generalize (IHM2 N0 N5 N1 N7 N9 H H0 H1 H2 H3 H4 H19 H28 H7 H38 H47); intro.
rewrite <- H48; clear H48. reflexivity.
(****************************** Abs *************************************)
intros. inversion H5. rewrite <- H14 in H6. clear H14. inversion H6.
generalize (eq_var n1 n2 a b H19 H22); intro. contradiction.
rewrite <- H12 in H8. inversion H8. generalize (eq_var n1 n2 a b H31 H34); intro.
contradiction. generalize (subst_term_eq P M Z Z0 n1 n2 H28 H40); intro.
rewrite <- H41 in H37; clear H41. rewrite <- H37 in H9. inversion H9.
reflexivity.  generalize (neq_couple a b a b H51); intro.
generalize (ref_eq_couple a b); intro. contradiction. reflexivity.
generalize(sub_var_nfree M n1 n2 x0 P H24 H26 H34); intro.
generalize(subst_term_eq P M Z M n1 n2 H28 H41); intro.
rewrite H42; clear H42. rewrite <- H37 in H9. assert(exists v:nat, level N1 v).
apply(exist_level N1).  inversion_clear H42.
generalize (level_subst N1 (Abs(a,b)M) N3 a b x2 H43 H9); intro.
generalize (sub_abs_nfree0 a b M x2 N1 H43 H42); intro.
generalize(subst_term_eq N1 (Abs(a,b)M) N3 (Abs(a,b)M) a b H9 H44); intro.
rewrite H45; reflexivity. rewrite <- H12 in H8.
generalize (sub_abs_nfree2 n1 n2 a b M x0 P H22 H24 H26 H27 H28); intro.
generalize (subst_term_eq P (Abs (a,b)M) N2 (Abs(a,b)M) n1 n2 H8 H29); intro.
rewrite H30 in H9. assert(exists v:nat, level N1 v). apply(exist_level N1).
inversion_clear H31. generalize(level_subst N1 (Abs(a,b)M) N3 a b x1 H32 H9); intro.
generalize(sub_abs_nfree0 a b M x1 N1 H32 H31); intro.
generalize(subst_term_eq N1 (Abs(a,b)M) N3 (Abs(a,b)M) a b H9 H33); intro.
rewrite H34; reflexivity. rewrite <- H12 in H5; rewrite <- H12 in H8.
```

rewrite <- H18 in H6; clear H18. inversion H6. rewrite H22 in H8; rewrite H25 in H8.

generalize (sub_abs_nfree0 m n M x0 P H27 H28); intro.

generalize (subst_term_eq P (Abs (m,n)M) N2 (Abs(m,n)M) m n H8 H29); intro.

rewrite H30 in H9; clear H30. rewrite H22 in H7; rewrite H25 in H7.

generalize (sub_var_nfree Q m n x0 P H27 H28 H14); intro.

generalize (subst_term_eq P Q N1 Q m n H7 H30); intro. rewrite H31 in H9.

generalize(sub_abs_nfree1 a b m n M x Z Q H14 H15 H17 H19 H20); intro.

generalize (subst_term_eq Q (Abs (m,n)M) N3 (Abs(m,n)Z) a b H9 H32); intro.

rewrite H33; reflexivity. inversion H8. generalize (eq_couple n1 n2 m n H34 H37); intro.

generalize (neq_couple n1 n2 m n H30); intro. contradiction.

rewrite <- H40 in H9; clear H40. inversion H9. generalize (eq_couple a b m n H45 H48); intro.

generalize (neq_couple a b m n H19); intro. contradiction.

generalize (IHM Z Z0 N1 Z1 Z2 H H0 H1 H2 H3 H4 H20 H31 H7 H43 H54); intro.

rewrite H55; reflexivity. generalize (sub_var_nfree Z1 a b x2 N1 H50 H52 H48); intro.

generalize (IHM Z Z0 N1 Z1 Z1 H H0 H1 H2 H3 H4 H20 H31 H7 H43 H55); intro.

rewrite H56; reflexivity. rewrite <- H40 in H9. inversion H9. generalize (eq_couple a b m n H46 H49); intro.

generalize (neq_couple a b m n H19); intro. contradiction.

generalize (sub_var_nfree M n1 n2 x1 P H39 H41 H37); intro.

generalize (IHM Z Z0 N1 M Z1 H H0 H1 H2 H3 H4 H20 H31 H7 H56 H55); intro.

rewrite H57; reflexivity. generalize (sub_var_nfree M a b q Q H3 H4 H49); intro.

generalize (subst_term_eq Q M Z M a b H20 H56); intro. rewrite H57 in H6; clear H57.

generalize (sub_abs_nfree2 n1 n2 m n M x0 P H37 H27 H29 H42 H43); intro.

generalize (subst_term_eq P (Abs(m,n)M) M2 (Abs(m,n)M) n1 n2 H6 H57); intro.

rewrite H58 in H28; clear H58. rewrite H28; reflexivity.

inversion H8. rewrite H34 in H7; rewrite H37 in H7.

generalize(sub_var_nfree Q m n x1 P H39 H40 H14); intro.

generalize(subst_term_eq P Q N1 Q m n H7 H41); intro. rewrite H42 in H9; clear H42.

rewrite H34 in H8; rewrite H37 in H8. generalize(sub_abs_nfree0 m n M x1 P H39 H40); intro.

generalize(subst_term_eq P (Abs(m,n)M) N2 (Abs(m,n)M) m n H8 H42); intro.

rewrite H43 in H9; clear H43.

generalize(sub_abs_nfree1 a b m n M x Z Q H14 H15 H17 H19 H20); intro.

generalize(subst_term_eq Q (Abs(m,n)M) N3 (Abs(m,n)Z) a b H9 H43); intro.

rewrite H44; reflexivity. rewrite <- H40 in H9; inversion H9.

generalize (eq_couple a b m n H46 H49); intro.

generalize (neq_couple a b m n H19); intro. contradiction.

generalize (sub_var_nfree Z n1 n2 p P H1 H2 H25); intro.

generalize (IHM Z Z N1 Z0 Z1 H H0 H1 H2 H3 H4 H20 H56 H7 H43 H55); intro.

rewrite H57; reflexivity.

generalize (sub_var_nfree Z0 a b x2 N1 H51 H53 H49); intro.

generalize (sub_var_nfree Z n1 n2 x0 P H27 H29 H25); intro.

generalize (IHM Z Z N1 Z0 Z0 H H0 H1 H2 H3 H4 H20 H57 H7 H43 H56); intro.

rewrite H58; reflexivity. rewrite <- H40 in H9; inversion H9.

generalize (eq_couple a b m n H46 H49); intro.

generalize (neq_couple a b m n H19); intro. contradiction.

generalize (sub_var_nfree Z n1 n2 x0 P H27 H29 H25); intro.

generalize (sub_var_nfree M n1 n2 x1 P H39 H41 H37); intro.

generalize (IHM Z Z N1 M Z0 H H0 H1 H2 H3 H4 H20 H56 H7 H57 H55); intro.

rewrite H58; reflexivity. generalize (sub_var_nfree M a b q Q H3 H4 H49); intro.

generalize (subst_term_eq Q M Z M a b H20 H56); intro. rewrite H57 in H6; clear H57.

generalize (sub_abs_nfree2 n1 n2 m n M x1 P H37 H39 H41 H42 H43); intro.

generalize (subst_term_eq P (Abs(m,n)M) M2 (Abs(m,n)M) n1 n2 H6 H57); intro.

rewrite H58 in H28; inversion_clear H28. reflexivity.

rewrite <- H12 in H5; rewrite <- H12 in H8. rewrite <- H18 in H6; inversion H6.

rewrite H23 in H8; rewrite H26 in H8. generalize(sub_abs_nfree0 m n M x0 P H28 H29); intro.

generalize(subst_term_eq P (Abs(m,n)M) N2 (Abs(m,n)M) m n H8 H30); intro.

rewrite H31 in H9. inversion H9. generalize(eq_couple a b m n H34 H37); intro.

generalize(neq_couple a b m n H20); intro. contradiction.

generalize(sub_var_nfree M a b x1 N1 H39 H41 H14); intro.

generalize(subst_term_eq N1 M Z M a b H43 H44); intro. rewrite H45; trivial.

reflexivity. rewrite <- H29 in H6.

generalize(subst_term_eq P (Abs(m,n)M) (Abs(m,n)Z) N2 n1 n2 H6 H8); intro.

rewrite <- H33 in H9. inversion H9. generalize(neq_couple a b m n H20); intro.

generalize(eq_couple a b m n H36 H39); intro. contradiction.

generalize(sub_var_nfree M a b x Q H15 H17 H14); intro.

generalize(IHM M Z N1 Z Z0 H H0 H1 H2 H3 H4 H46 H32 H7 H32 H45); intro.

rewrite H47; reflexivity. reflexivity.

generalize(sub_abs_nfree2 n1 n2 m n M x0 P H26 H28 H30 H31 H32); intro.

generalize(subst_term_eq P (Abs(m,n)M) (Abs(m,n)M) N2 n1 n2 H33 H8); intro.

rewrite <- H34 in H9; clear H34. assert(exists v:nat, level N1 v). apply(exist_level N1).

inversion_clear H34. generalize (level_subst N1 (Abs(m,n)M) N3 a b x1 H35 H9); intro.

generalize(nfree_M_N_P n1 n2 m n P Q N1 H31 H19 H7); intro.

generalize(sub_abs_nfree2 a b m n M x1 N1 H14 H35 H34 H36 H20); intro.

generalize(subst_term_eq N1 (Abs(m,n)M) (Abs(m,n)M) N3 a b H37 H9); intro.

rewrite <- H38; reflexivity.

*(********** Imp **********)*

intros. inversion H5. rewrite <- H17 in H6. inversion H6.

inversion H8. rewrite <- H37 in H9. inversion H9.

generalize (IHM1 M4 M6 N1 M8 M10 H H0 H1 H2 H3 H4 H18 H28 H7 H38 H48); intro.

rewrite <- H50; clear H50.

generalize (IHM2 N0 N5 N1 N7 N9 H H0 H1 H2 H3 H4 H19 H29 H7 H39 H49); intro.

rewrite <- H50; clear H50. reflexivity.

*(********** PI **********)*

intros. inversion H5. rewrite <- H16 in H6. inversion H6. inversion H8. rewrite <- H32 in H9. inversion H9.

generalize (IHM N N0 N1 N4 N5 H H0 H1 H2 H3 H4 H17 H25 H7 H33 H41); intro.

rewrite <- H42; reflexivity.

Qed.

Lemme 4: (Proposition 8 de chapitre 3)

Lemma red_subst_red: forall P Q,(Ebeta_reduc P Q)-> forall M P1 Q1 n1 n2,

((level M) <= n2) ->(substitution (var(n1,n2)) M P P1)->

(substitution (var(n1,n2)) M Q Q1)-> (Ebeta_reduc P1 Q1).

Proof.

induction 1.

*(**********reflexivité**********)*

intros. generalize (subst_term_eq M0 M P1 Q1 n1 n2 H0 H1); intro.

rewrite H2. apply refl_reduc.

*(********** application à gauche **********)*

intros. inversion H1. inversion H2.

generalize (subst_term_eq M0 Z M2 M4 n1 n2 H10 H19); intro. rewrite <- H21; clear H21.

generalize (IHEbeta_reduc M0 N1 N3 n1 n2 H0 H11 H20); intro.

apply Appl_reduc. exact H21.

*(********** application à droite **********)*

intros. inversion H1. inversion H2.

generalize (subst_term_eq M0 Z N1 N3 n1 n2 H11 H20); intro. rewrite <- H21; clear H21.

generalize (IHEbeta_reduc M0 M2 M4 n1 n2 H0 H10 H19); intro.

generalize (Appr_reduc M2 M4 N1 H21); intro. exact H22.

*(************************ implication à gauche ***************************)*

intros. inversion H1. inversion H2.

generalize (subst_term_eq M0 X M2 M4 n1 n2 H10 H19); intro. rewrite <- H21; clear H21.

generalize (IHEbeta_reduc M0 N1 N3 n1 n2 H0 H11 H20); intro.

generalize (Impl_reduc N1 N3 M2 H21); intro. exact H22.

*(****************************** implication à droite ***********************)*

intros. inversion H1. inversion H2.

generalize (subst_term_eq M0 X N1 N3 n1 n2 H11 H20); intro.

rewrite <- H21; clear H21. generalize (IHEbeta_reduc M0 M2 M4 n1 n2 H0 H10 H19); intro.

generalize (Impr_reduc M2 M4 N1 H21); intro. exact H22.

*(****************************** béta réduction ***************************)*

intros. inversion H. rewrite <- H4 in H1. inversion H1. inversion H13.

inversion H14. rewrite <- H27. rewrite <- H26 in H3; inversion H3.

rewrite <- H33 in H2; inversion H2. rewrite <- H39; clear H39.

generalize(sub_var_eq p q M0 H22); intro.

generalize(beta_red0 M0 (cvar(var(p,q))) M0 p q H39); intro.

apply(beta_reduc (App (Abs (var(p, q)) (cvar (var (p, q)))) M0) M0). exact H40.

generalize (ref_eq_couple n1 n2); intro. contradiction.

rewrite <- H34 in H2; inversion H2. rewrite <- H39; clear H39.

rewrite <- H19 in H35; rewrite <- H20 in H35.

generalize (eq_couple n1 n2 m n H40 H41); intro. contradiction.

generalize (sub_var_neq p q m n M0 H22 H35); intro.

generalize(beta_red0 M0 (cvar(var(m,n))) (cvar(var(m,n))) p q H44); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (m, n)))) M0) (cvar (var (m, n)))).

exact H45. rewrite <- H33 in H2; inversion H2.

generalize (sub_abs_nfree0 p q M5 M0 H42); intro.

generalize(beta_red0 M0 (Abs(var(p,q))M5) (Abs(var(p,q))M5) p q H43); intro.

apply(beta_reduc(App (Abs (var(p, q)) (Abs(var(p, q)) M5)) M0) (Abs(var(p, q)) M5)).

exact H44.

generalize (eq_couple n1 n2 p q H19 H20); intro. contradiction.

rewrite H20 in H43. generalize(sub_abs_nfree0 p q M5 M0 H43); intro.

generalize(beta_red0 M0 (Abs(var (p, q)) M5)(Abs(var (p, q)) M5) p q H46); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M5)) M0) (Abs(var (p, q)) M5) H47).

rewrite <- H36 in H2; inversion H2. rewrite <- H19 in H33; rewrite <- H20 in H33.

generalize (eq_couple n1 n2 m n H42 H43); intro. contradiction.

rewrite <- H19 in H37; rewrite <- H20 in H37.

generalize(subst_ident_var M5 Z n1 n2 H37); intro. rewrite <- H49 in H48; clear H49.

rewrite H19 in H48; rewrite H20 in H48.

generalize(sub_abs_nfree1 p q m n M5 Z0 M0 H44 H22 H33 H48); intro.

generalize(beta_red0 M0 (Abs(var (m, n)) M5) (Abs(var(m, n)) Z0) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M5)) M0) (Abs(var (m, n)) Z0) H50).

rewrite <- H19 in H37; rewrite <- H20 in H37.

generalize(subst_ident_var M5 Z n1 n2 H37); intro.

rewrite <- H49 in H44; rewrite <- H49; clear H49. rewrite H19 in H44; rewrite H20 in H44.

generalize(sub_abs_nfree2 p q m n M5 M0 H44 H22 H47 H33); intro.

generalize(beta_red0 M0 (Abs(var (m, n)) M5) (Abs(var (m, n)) M5) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M5)) M0) (Abs(var (m, n)) M5) H50).

rewrite <- H36 in H2; inversion H2. rewrite H42 in H19; rewrite H43 in H20.

rewrite H19; rewrite H20. generalize(sub_abs_nfree0 p q M5 M0 H22); intro.

generalize(beta_red0 M0 (Abs(var (p, q)) M5) (Abs(var (p, q)) M5) p q H46); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M5)) M0) (Abs(var (p, q)) M5) H47).

rewrite H19 in H48; rewrite H20 in H48.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H44 H22 H37 H48); intro.

generalize(beta_red0 M0 (Abs(var (m, n)) M5) (Abs(var (m, n)) Z) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M5)) M0) (Abs(var (m, n)) Z) H50).

generalize(sub_abs_nfree2 p q m n M5 M0 H31 H22 H47 H37); intro.

generalize(beta_red0 M0 (Abs(var (m, n)) M5) (Abs(var (m, n)) M5) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n))M5)) M0) (Abs(var (m, n)) M5) H50).

rewrite <- H35 in H2; inversion H2. rewrite H19 in H32; rewrite H20 in H32.

rewrite H19 in H36; rewrite H20 in H36. generalize(subst_ident_var M5 M6 p q H32); intro.

generalize(subst_ident_var N3 N4 p q H36); intro.

rewrite <- H46 in H44; rewrite <- H47 in H45.

generalize(sub_app n1 n2 M5 N3 M8 N6 M0 H42 H44 H45); intro.

rewrite H19 in H48; rewrite H20 in H48.

generalize(beta_red0 M0 (App M5 N3) (App M8 N6) p q H48); intro.

apply (beta_reduc (App (Abs(var (p, q)) (App M5 N3)) M0) (App M8 N6) H49).

rewrite <- H35 in H2; inversion H2. rewrite H19 in H32; rewrite H20 in H32.

rewrite H19 in H36; rewrite H20 in H36. generalize(subst_ident_var M5 M6 p q H32); intro.

generalize(subst_ident_var N3 N4 p q H36); intro.

rewrite <- H46 in H44; rewrite <- H47 in H45.

generalize(sub_imp n1 n2 M5 N3 M8 N6 M0 H42 H44 H45); intro.

rewrite H19 in H48; rewrite H20 in H48.

generalize(beta_red0 M0 (Imp M5 N3) (Imp M8 N6) p q H48); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Imp M5 N3)) M0) (Imp M8 N6) H49).

rewrite <- H34 in H2; inversion H2. rewrite H19 in H35; rewrite H20 in H35.

generalize(subst_ident_var M5 N3 p q H35); intro. rewrite <- H43 in H42.

rewrite H19 in H42; rewrite H20 in H42. generalize(sub_pi p q M5 M0 N4 H22 H42); intro.

generalize (beta_red0 M0 (PI M5) (PI N4) p q H44); intro.

apply (beta_reduc (App (Abs(var (p, q)) (PI M5)) M0) (PI N4) H45).

rewrite <- H27 in H3; inversion H3. rewrite <- H34 in H2; inversion H2.

generalize(eq_couple n1 n2 m n H40 H41); intro. contradiction.

generalize (sub_var_eq p q (cvar(var(m,n))) H35); intro.

generalize(beta_red0 (cvar (var (m, n))) (cvar (var (p, q))) (cvar (var (m, n))) p q H44); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (cvar (var (m, n))))(cvar (var (m, n))) H45).

rewrite <- H35 in H2; inversion H2. rewrite <- H19 in H36; rewrite <- H20 in H36.

generalize(eq_couple n1 n2 m0 n0 H41 H42); intro. contradiction.

generalize(sub_var_neq p q m0 n0 (cvar(var(m,n))) H32 H36); intro.

generalize(beta_red0 (cvar (var (m, n))) (cvar (var (m0, n0))) (cvar (var (m0, n0))) p q H45); intro.

apply (beta_reduc (App (Abs(var (p, q)) (cvar (var (m0, n0)))) (cvar (var (m, n)))) (cvar (var (m0, n0))) H46).

rewrite <- H34 in H2; inversion H2. generalize(sub_abs_nfree0 p q M5 (cvar(var(m,n))) H35); intro.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (p, q)) M5)(Abs(var (p, q)) M5) p q H44); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M5)) (cvar (var (m, n)))) (Abs(var (p, q)) M5) H45).

generalize(eq_couple n1 n2 p q H19 H20); intro. contradiction.

generalize(sub_abs_nfree0 p q M5 (cvar(var(m,n))) H35); intro.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (p, q)) M5)(Abs(var (p, q)) M5) p q H47); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M5)) (cvar (var (m, n)))) (Abs(var (p, q)) M5) H48).

rewrite <- H37 in H2; inversion H2. rewrite <- H19 in H34; rewrite <- H20 in H34.

generalize(eq_couple n1 n2 m0 n0 H43 H44); intro. contradiction.

rewrite H19 in H29; rewrite H20 in H29. generalize(inv_neq_couple p q m n H29); intro.

generalize(sub_var_t1_nlib_t2 M5 Z p q m n H50 H38); intro.

rewrite H19 in H49; rewrite H20 in H49.

generalize(sub_var_nfree Z p q M0 H22 H51); intro.

generalize(subst_term_eq M0 Z Z0 Z p q H49 H52); intro. rewrite H53; clear H53.

generalize (sub_abs_nfree1 p q m0 n0 M5 Z (cvar(var(m,n))) H32 H33 H34 H38); intro.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (m0, n0)) M5) (Abs(var (m0, n0)) Z) p q H53); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M5)) (cvar (var (m, n)))) (Abs(var (m0, n0)) Z) H54).

generalize (sub_abs_nfree1 p q m0 n0 M5 Z (cvar(var(m,n))) H32 H33 H34 H38); intro.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (m0, n0)) M5) (Abs(var (m0, n0)) Z) p q H50); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M5)) (cvar (var (m, n)))) (Abs(var (m0, n0))Z) H51).

rewrite <- H37 in H2; inversion H2.

generalize(sub_abs_nfree2 p q m0 n0 M5 (cvar(var(m,n))) H32 H33 H34 H38); intro.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (m0, n0)) M5) (Abs(var (m0, n0)) M5) p q H47); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M5)) (cvar (var (m, n)))) (Abs(var(m0, n0)) M5) H48).

generalize(sub_abs_nfree2 p q m0 n0 M5 (cvar(var(m,n))) H32 H33 H34 H38); intro.

rewrite H19 in H49; rewrite H20 in H49. generalize(sub_var_nfree M5 p q M0 H22 H32); intro.

generalize(subst_term_eq M0 M5 Z M5 p q H49 H51); intro. rewrite H52; clear H52.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (m0, n0)) M5) (Abs(var (m0, n0)) M5) p q H50); intro.

apply (beta_reduc (App (Abs (var(p, q)) (Abs(var (m0, n0)) M5)) (cvar (var (m, n)))) (Abs(var (m0, n0)) M5) H52).

generalize(sub_abs_nfree2 p q m0 n0 M5 (cvar(var(m,n))) H32 H33 H34 H38); intro.

generalize(beta_red0 (cvar (var (m, n))) (Abs(var (m0, n0)) M5) (Abs(var (m0, n0)) M5) p q H50); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M5)) (cvar (var (m, n)))) (Abs(var (m0, n0)) M5) H51).

rewrite <- H36 in H2; inversion H2. rewrite H19 in H29; rewrite H20 in H29.

generalize(inv_neq_couple p q m n H29); intro.

generalize(sub_var_t1_nlib_t2 M5 M6 p q m n H47 H33); intro.

generalize(sub_var_t1_nlib_t2 N3 N4 p q m n H47 H37); intro.

rewrite H19 in H45; rewrite H20 in H45.

rewrite H19 in H46; rewrite H20 in H46.

generalize (sub_var_nfree M6 p q M0 H22 H48); intro.

generalize(subst_term_eq M0 M6 M8 M6 p q H45 H50); intro.

rewrite H51; clear H51.

generalize (sub_var_nfree N4 p q M0 H22 H49); intro.

generalize(subst_term_eq M0 N4 N6 N4 p q H46 H51); intro. rewrite H52; clear H52.

generalize(sub_app p q M5 N3 M6 N4 (cvar(var(m,n))) H32 H33 H37); intro.

generalize(beta_red0 (cvar (var (m, n))) (App M5 N3) (App M6 N4) p q H52); intro.

apply (beta_reduc (App (Abs (var(p, q)) (App M5 N3)) (cvar (var (m, n)))) (App M6 N4) H53).

rewrite <- H36 in H2; inversion H2. rewrite H19 in H29; rewrite H20 in H29.

generalize(inv_neq_couple p q m n H29); intro.

generalize(sub_var_t1_nlib_t2 M5 M6 p q m n H47 H33); intro.

generalize(sub_var_t1_nlib_t2 N3 N4 p q m n H47 H37); intro.

rewrite H19 in H45; rewrite H20 in H45. rewrite H19 in H46; rewrite H20 in H46.

generalize (sub_var_nfree M6 p q M0 H22 H48); intro.

generalize(subst_term_eq M0 M6 M8 M6 p q H45 H50); intro.

rewrite H51; clear H51. generalize (sub_var_nfree N4 p q M0 H22 H49); intro.

generalize(subst_term_eq M0 N4 N6 N4 p q H46 H51); intro. rewrite H52; clear H52.

generalize(sub_imp p q M5 N3 M6 N4 (cvar(var(m,n))) H32 H33 H37); intro.

generalize(beta_red0 (cvar (var (m, n))) (Imp M5 N3) (Imp M6 N4) p q H52); intro.

apply (beta_reduc (App (Abs (var(p, q)) (Imp M5 N3)) (cvar (var (m, n)))) (Imp M6 N4) H53).

rewrite <- H35 in H2; inversion H2. rewrite H19 in H29; rewrite H20 in H29.

generalize(inv_neq_couple p q m n H29); intro.

generalize (sub_var_t1_nlib_t2 M5 N3 p q m n H44 H36); intro.

rewrite H19 in H43; rewrite H20 in H43. generalize (sub_var_nfree N3 p q M0 H22 H45); intro.

generalize(subst_term_eq M0 N3 N4 N3 p q H43 H46); intro.rewrite H47; clear H47.

generalize (sub_pi p q M5 (cvar(var(m,n))) N3 H32 H36); intro.

generalize(beta_red0 (cvar (var (m, n))) (PI M5) (PI N3) p q H47); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M5)) (cvar (var (m, n)))) (PI N3) H48).

rewrite <- H26 in H3; inversion H3. rewrite <- H33 in H2; inversion H2.

generalize(sub_var_eq p q (Abs(var(n1,n2))M5) H34); intro.

generalize(beta_red0 (Abs(var (n1, n2)) M5) (cvar (var (p, q))) (Abs(var (n1, n2)) M5) p q H41); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Abs(var (n1, n2)) M5))(Abs(var (n1, n2)) M5) H42).

generalize(ref_eq_couple n1 n2); intro. contradiction.

rewrite H19; rewrite H20. rewrite H19 in H34; rewrite H20 in H34.

generalize(sub_var_eq p q (Abs(var(p,q))M5) H34); intro.

generalize(beta_red0 (Abs(var (p, q)) M5) (cvar(var(p,q))) (Abs(var (p, q)) M5) p q H46); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Abs(var (p, q)) M5))(Abs(var (p, q)) M5) H47).

rewrite <- H34 in H2; inversion H2. rewrite <- H19 in H35; rewrite <- H20 in H35.

generalize(eq_couple n1 n2 m n H40 H41); intro. contradiction.

generalize(sub_var_neq p q m n (Abs(var(n1,n2))M5) H31 H35); intro.

generalize(beta_red0 (Abs(var (n1, n2)) M5) (cvar (var (m, n))) (cvar (var (m, n))) p q H44); intro.

apply (beta_reduc (App (Abs(var (p, q)) (cvar (var (m, n)))) (Abs(var (n1, n2)) M5)) (cvar (var (m, n))) H45).

rewrite <- H33 in H2; inversion H2. rewrite H39 in H34; rewrite H40 in H34.

generalize(sub_abs_nfree0 p q M6 (Abs(var(p,q))M5) H34); intro.

generalize(beta_red0 (Abs(var (p, q)) M5) (Abs(var (p, q)) M6) (Abs(var (p, q)) M6) p q H43); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M6)) (Abs(var (p, q)) M5)) (Abs(var (p, q)) M6) H44).

generalize(eq_couple n1 n2 p q H19 H20); intro. contradiction.

generalize(sub_abs_nfree0 p q M6 (Abs(var(n1,n2))M5) H34); intro.

generalize(beta_red0 (Abs(var (n1, n2)) M5) (Abs(var (p, q)) M6) (Abs(var (p, q)) M6) p q H46); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M6)) (Abs(var (n1, n2)) M5)) (Abs(var (p, q)) M6) H47).

rewrite <- H36 in H2; inversion H2. rewrite H19 in H42; rewrite H20 in H43.

generalize(eq_couple p q m n H42 H43); intro. contradiction.

rewrite H19 in H37; rewrite H20 in H37.

generalize(sub_abs_var_t1_nlib_t2 M6 p q M5 Z H37); intro.

rewrite H19 in H48; rewrite H20 in H48. generalize(sub_var_nfree Z p q M0 H22 H49); intro.

generalize(subst_term_eq M0 Z Z0 Z p q H48 H50); intro. rewrite H51; clear H51.

rewrite H19; rewrite H20. rewrite H19 in H31; rewrite H20 in H31.

rewrite H19 in H32; rewrite H20 in H32.

generalize(sub_abs_nfree1 p q m n M6 Z (Abs(var(p,q))M5) H31 H32 H33 H37); intro.

generalize(beta_red0 (Abs (var(p, q)) M5) (Abs(var (m, n)) M6) (Abs(var (m, n)) Z) p q H51); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n))M6)) (Abs(var (p, q)) M5)) (Abs(var (m, n)) Z) H52).

generalize(sub_abs_nfree1 p q m n M6 Z (Abs(var(n1,n2))M5) H31 H32 H33 H37); intro.

generalize(beta_red0 (Abs(var (n1, n2)) M5) (Abs(var (m, n)) M6) (Abs(var (m, n)) Z) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (Abs(var (n1, n2)) M5)) (Abs(var (m, n)) Z) H50).

rewrite <- H36 in H2; inversion H2. rewrite H19 in H42; rewrite H20 in H43.

rewrite <- H42; rewrite <- H43. rewrite H19 in H32; rewrite H20 in H32.

generalize(sub_abs_nfree0 p q M6 (Abs(var(p,q))M5) H32); intro.

generalize(beta_red0 (Abs(var (p, q)) M5) (Abs(var (p, q)) M6) (Abs(var (p, q)) M6) p q H46); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M6)) (Abs(var (p, q)) M5)) (Abs(var (p, q)) M6) H47).

rewrite H19 in H48; rewrite H20 in H48.

generalize(sub_var_nfree M6 p q M0 H22 H31); intro.

generalize(subst_term_eq M0 M6 Z M6 p q H48 H49); intro. rewrite H50; clear H50.

generalize(sub_abs_nfree2 p q m n M6 (Abs(var(n1,n2))M5) H31 H32 H33 H37); intro.

generalize(beta_red0 (Abs(var (n1, n2)) M5) (Abs(var (m, n)) M6) (Abs(var (m, n)) M6)p q H50); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (Abs(var (n1, n2)) M5)) (Abs(var (m, n)) M6) H51).

generalize(sub_abs_nfree2 p q m n M6 (Abs(var(n1,n2))M5) H31 H32 H33 H37); intro.

generalize(beta_red0 (Abs(var (n1, n2)) M5) (Abs (var(m, n)) M6) (Abs(var (m, n)) M6) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (Abs(var (n1, n2)) M5)) (Abs(var (m, n)) M6) H50).

rewrite <- H35 in H2; inversion H2. rewrite H19 in H44; rewrite H20 in H44.

rewrite H19 in H45; rewrite H20 in H45. rewrite H19 in H32; rewrite H20 in H32.

rewrite H19 in H36; rewrite H20 in H36.

generalize(sub_abs_var_t1_nlib_t2 M6 p q M5 M7 H32); intro.

generalize(sub_abs_var_t1_nlib_t2 N3 p q M5 N4 H36); intro.

generalize(sub_var_nfree M7 p q M0 H22 H46); intro.

generalize(sub_var_nfree N4 p q M0 H22 H47); intro.

generalize(subst_term_eq M0 M7 M9 M7 p q H44 H48); intro.

generalize(subst_term_eq M0 N4 N6 N4 p q H45 H49); intro.

rewrite H50; clear H50; rewrite H51; clear H51.

rewrite H19; rewrite H20. rewrite H19 in H31; rewrite H20 in H31.

generalize (sub_app p q M6 N3 M7 N4 (Abs(var(p,q))M5) H31 H32 H36); intro.

generalize(beta_red0 (Abs(var (p, q)) M5) (App M6 N3) (App M7 N4) p q H50); intro.

apply (beta_reduc (App (Abs (var(p, q)) (App M6 N3)) (Abs (var(p, q)) M5)) (App M7 N4) H51).

rewrite <- H35 in H2; inversion H2.

rewrite H19 in H44; rewrite H20 in H44. rewrite H19 in H45; rewrite H20 in H45.

rewrite H19 in H32; rewrite H20 in H32. rewrite H19 in H36; rewrite H20 in H36.

generalize(sub_abs_var_t1_nlib_t2 M6 p q M5 M7 H32); intro.

generalize(sub_abs_var_t1_nlib_t2 N3 p q M5 N4 H36); intro.

generalize(sub_var_nfree M7 p q M0 H22 H46); intro.

generalize(sub_var_nfree N4 p q M0 H22 H47); intro.

generalize(subst_term_eq M0 M7 M9 M7 p q H44 H48); intro.

generalize(subst_term_eq M0 N4 N6 N4 p q H45 H49); intro.

rewrite H50; clear H50; rewrite H51; clear H51.

rewrite H19; rewrite H20. rewrite H19 in H31; rewrite H20 in H31.

generalize (sub_imp p q M6 N3 M7 N4 (Abs(var(p,q))M5) H31 H32 H36); intro.

generalize(beta_red0 (Abs(var (p, q)) M5) (Imp M6 N3) (Imp M7 N4) p q H50); intro.

apply (beta_reduc (App (Abs (var(p, q)) (Imp M6 N3)) (Abs (var(p, q)) M5)) (Imp M7 N4) H51).

rewrite <- H34 in H2; inversion H2. rewrite H19 in H35; rewrite H20 in H35.

generalize(sub_abs_var_t1_nlib_t2 M6 p q M5 N3 H35); intro.

rewrite H19 in H42; rewrite H20 in H42.

generalize(sub_var_nfree N3 p q M0 H22 H43); intro.

generalize(subst_term_eq M0 N3 N4 N3 p q H42 H44); intro.

rewrite H45; clear H45. rewrite H19; rewrite H20.

rewrite H19 in H31; rewrite H20 in H31.

generalize(sub_pi p q M6 (Abs(var(p,q))M5) N3 H31 H35); intro.

generalize(beta_red0 (Abs(var (p, q)) M5) (PI M6) (PI N3) p q H45); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M6)) (Abs(var (p, q)) M5)) (PI N3) H46).

rewrite <- H29 in H3; inversion H3. rewrite <- H36 in H2; inversion H2.

generalize(eq_couple n1 n2 m n H42 H43); intro. contradiction.

generalize(subst_term_eq M0 M5 Z Z0 n1 n2 H31 H48); intro.

rewrite H49; clear H49. rewrite H19 in H47; rewrite H20 in H47.

rewrite H19 in H48; rewrite H20 in H48.

generalize(sub_abs_nfree1 p q m n M5 Z0 M0 H44 H22 H47 H48); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z0) p q H22 H49); intro.

assert((level (Abs (var (m, n)) Z0)) <= q). apply (le_trans (level (Abs (var (m, n)) Z0)) (level (Abs (var (m, n)) M5)) q H50 H37).

generalize(sub_var_eq p q (Abs(var(m,n))Z0) H51); intro.

generalize (beta_red0 (Abs(var (m, n)) Z0) (cvar (var (p, q))) (Abs(var (m, n)) Z0) p q H52); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Abs(var (m, n)) Z0)) (Abs(var (m, n)) Z0) H53).

generalize(sub_var_nfree M5 n1 n2 M0 H46 H44); intro.

generalize(subst_term_eq M0 M5 Z M5 n1 n2 H31 H49); intro.

rewrite H50; clear H50. generalize (sub_var_eq p q (Abs(var(m,n))M5) H37); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (cvar (var (p, q))) (Abs(var (m, n)) M5) p q H50); intro.

apply (beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Abs(var (m, n)) M5)) (Abs(var (m, n)) M5) H51).

rewrite <- H37 in H2; inversion H2. rewrite <- H19 in H38; rewrite <- H20 in H38.

generalize(eq_couple n1 n2 m0 n0 H43 H44); intro. contradiction.

rewrite H19 in H27; rewrite H20 in H27; rewrite H19 in H31; rewrite H20 in H31.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H47); intro.

assert((level (Abs (var (m, n)) Z)) <= q). apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H48 H34).

generalize(sub_var_neq p q m0 n0 (Abs(var(m,n))Z) H49 H38); intro.

generalize(beta_red0 (Abs (var(m, n)) Z) (cvar (var (m0, n0))) (cvar (var (m0, n0))) p q H50); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (m0, n0)))) (Abs(var (m, n)) Z)) (cvar (var (m0, n0))) H51).

rewrite <- H36 in H2. rewrite H19 in H2; rewrite H20 in H2.

generalize(sub_abs_nfree0 p q M6 M0 H22); intro.

generalize(subst_term_eq M0 (Abs(var(p,q))M6) Q1 (Abs(var(p,q))M6) p q H2 H38); intro.

rewrite H39; clear H39. rewrite H19 in H27; rewrite H20 in H27.

rewrite H19 in H31; rewrite H20 in H31.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H39); intro.

assert((level (Abs (var (m, n)) Z)) <= q). apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H40 H37).

generalize(sub_abs_nfree0 p q M6 (Abs(var(m,n))Z) H41); intro.

generalize(beta_red0 (Abs(var (m, n)) Z) (Abs(var (p, q)) M6) (Abs(var (p, q)) M6) p q H42); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M6)) (Abs(var (m, n)) Z)) (Abs(var (p, q)) M6) H43).

rewrite <- H39 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

generalize(eq_couple p q m0 n0 H45 H46); intro. contradiction.

rewrite H19 in H31; rewrite H20 in H31. rewrite H19 in H27; rewrite H20 in H27.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

generalize(sub_abs_nfree1 p q m0 n0 M6 Z0 (Abs(var(m,n))M5) H34 H35 H36 H40); intro.

generalize(sub_abs_nfree1 p q m0 n0 Z0 Z1 M0 H47 H49 H50 H51); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H49 H52); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H55 H35).

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) (Abs(var(m0,n0))M6) T).

apply (exist_term (Abs(var(m0,n0))M6) (Abs(var(m,n))Z) p q H56).

inversion_clear H57.

generalize(sub_eq_Z1_Z2 (Abs(var(m0,n0))M6) (Abs(var(m,n))Z) (Abs(var(m0,n0))Z0) (Abs(var(m0,n0))Z1)

x M0 (Abs(var(m,n))M5) p q H49 H35 H52 H53 H54 H58); intro.

rewrite H57 in H58; clear H57.

generalize(beta_red0 (Abs(var (m, n)) Z) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0)) Z1) p q H58); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) Z)) (Abs(var (m0, n0)) Z1) H57).

rewrite H19 in H31; rewrite H20 in H31. rewrite H19 in H27; rewrite H20 in H27.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

generalize(sub_abs_nfree1 p q m0 n0 M6 Z0 (Abs(var(m,n))M5) H34 H35 H36 H40); intro.

generalize(sub_abs_nfree2 p q m0 n0 Z0 M0 H47 H49 H50 H51); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H49 H52); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H55 H35).

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) (Abs(var(m0,n0))M6) T).

apply(exist_term (Abs(var(m0,n0))M6) (Abs(var(m,n))Z) p q H56); intro.

inversion_clear H57. Print sub_eq_Z1_Z2.

generalize(sub_eq_Z1_Z2 (Abs(var(m0,n0))M6) (Abs(var(m,n))Z) (Abs(var(m0,n0))Z0) (Abs(var(m0,n0))Z0)

x M0 (Abs(var(m,n))M5) p q H49 H35 H52 H53 H54 H58); intro.

rewrite H57 in H58; clear H57.

generalize(beta_red0 (Abs(var (m, n)) Z) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0)) Z0) p q H58); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) Z)) (Abs(var (m0, n0)) Z0) H57).

rewrite H19 in H31; rewrite H20 in H31. rewrite H19 in H27; rewrite H20 in H27.

rewrite <- H39 in H2. inversion H2. rewrite H19 in H45; rewrite H20 in H46.

generalize(eq_couple p q m0 n0 H45 H46); intro. contradiction.

rewrite <- H19 in H34; rewrite <- H20 in H34.

generalize(sub_var_nfree M6 n1 n2 M0 H49 H34); intro.

generalize(subst_term_eq M0 M6 Z0 M6 n1 n2 H51 H52); intro. rewrite H53; clear H53.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

rewrite H19 in H34; rewrite H20 in H34.

generalize(sub_abs_nfree2 p q m0 n0 M6 (Abs(var(m,n))M5) H34 H35 H36 H40); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H53); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H55 H35).

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) (Abs(var(m0,n0))M6) T).

apply(exist_term (Abs(var(m0,n0))M6) (Abs(var(m,n))Z) p q H56).

inversion_clear H57.

generalize(sub_abs_nfree1 n1 n2 m0 n0 M6 M6 M0 H47 H49 H50 H52); intro.

rewrite H19 in H57; rewrite H20 in H57.

generalize(sub_eq_Z1_Z2 (Abs(var(m0,n0))M6) (Abs(var(m,n))Z) (Abs(var(m0,n0))M6) (Abs(var(m0,n0))M6)

x M0 (Abs(var(m, n))M5) p q H22 H35 H53 H54 H57 H58); intro.

rewrite H59 in H58; clear H59.

generalize(beta_red0 (Abs(var (m, n)) Z) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0)) M6) p q H58); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) Z)) (Abs(var (m0, n0)) M6) H59).

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H52); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H53 H35).

rewrite H19 in H47; rewrite H20 in H47.

generalize(var_nfree_M_nfree_Abs p q m0 n0 M6 H47); intro.

generalize(sub_var_nfree (Abs(var (m0, n0)) M6) p q (Abs(var(m,n))Z) H54 H55); intro.

generalize(beta_red0 (Abs(var (m, n)) Z) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0)) M6) p q H56); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) Z)) (Abs(var (m0, n0)) M6) H57).

rewrite <- H38 in H2. rewrite H19 in H2; rewrite H20 in H2; inversion H2.

generalize(sub_abs_nfree1 n1 n2 m n M5 Z M0 H25 H26 H27 H31); intro.

rewrite H19 in H49; rewrite H20 in H49.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H49); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H50 H34).

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) M6 T).

apply(exist_term M6 (Abs(var(m,n))Z) p q H51). inversion_clear H52.

generalize(sub_eq_Z1_Z2 M6 (Abs(var(m,n))Z) M7 M9 x M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H35 H47 H53); intro.

rewrite H52 in H53; clear H52.

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) N3 T).

apply(exist_term N3 (Abs(var(m,n))Z) p q H51). inversion_clear H52.

generalize(sub_eq_Z1_Z2 N3 (Abs(var(m,n))Z) N4 N6 x0 M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H39 H48 H54); intro.

rewrite H52 in H54; clear H52.

generalize(sub_app p q M6 N3 M9 N6 (Abs(var(m,n))Z) H51 H53 H54); intro.

generalize(beta_red0 (Abs(var (m, n)) Z) (App M6 N3) (App M9 N6) p q H52); intro.

apply (beta_reduc (App (Abs(var (p, q)) (App M6 N3)) (Abs(var (m, n)) Z)) (App M9 N6) H55).

rewrite <- H38 in H2. rewrite H19 in H2; rewrite H20 in H2; inversion H2.

generalize(sub_abs_nfree1 n1 n2 m n M5 Z M0 H25 H26 H27 H31); intro.

rewrite H19 in H49; rewrite H20 in H49.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H49); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H50 H34).

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) M6 T).

apply(exist_term M6 (Abs(var(m,n))Z) p q H51). inversion_clear H52.

generalize(sub_eq_Z1_Z2 M6 (Abs(var(m,n))Z) M7 M9 x M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H35 H47 H53); intro.

rewrite H52 in H53; clear H52.

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) N3 T).

apply(exist_term N3 (Abs(var(m,n))Z) p q H51). inversion_clear H52.

generalize(sub_eq_Z1_Z2 N3 (Abs(var(m,n))Z) N4 N6 x0 M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H39 H48 H54); intro.

rewrite H52 in H54; clear H52.

generalize(sub_imp p q M6 N3 M9 N6 (Abs(var(m,n))Z) H51 H53 H54); intro.

generalize(beta_red0 (Abs(var (m, n)) Z) (Imp M6 N3) (Imp M9 N6) p q H52); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Imp M6 N3)) (Abs(var (m, n)) Z)) (Imp M9 N6) H55).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H31; rewrite H20 in H31. rewrite H19 in H27; rewrite H20 in H27.

generalize(sub_abs_nfree1 p q m n M5 Z M0 H25 H22 H27 H31); intro.

generalize(sub_pi p q M6 (Abs(var(m,n))M5) N3 H34 H38); intro.

generalize(sub_pi p q N3 M0 N4 H22 H45); intro.

generalize(subst_level (Abs(var(m,n))M5) M0 (Abs(var(m,n))Z) p q H22 H46); intro.

assert((level (Abs (var (m, n)) Z)) <= q).

apply (le_trans (level (Abs (var (m, n)) Z)) (level (Abs (var (m, n)) M5)) q H49 H34).

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))Z) (PI M6) T).

apply(exist_term (PI M6) (Abs(var(m,n))Z) p q H50).

inversion_clear H51.

generalize(sub_eq_Z1_Z2 (PI M6) (Abs(var(m,n))Z) (PI N3) (PI N4) x M0 (Abs(var(m,n))M5)

p q H22 H34 H46 H47 H48 H52); intro.

rewrite H51 in H52; clear H51.

generalize(beta_red0 (Abs(var (m, n)) Z) (PI M6) (PI N4) p q H52); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M6)) (Abs(var (m, n)) Z)) (PI N4) H51).

rewrite <- H29 in H3. inversion H3. rewrite <- H36 in H2. inversion H2.

generalize(eq_couple n1 n2 m n H42 H43); intro. contradiction.

generalize(sub_var_nfree M5 n1 n2 M0 H46 H25); intro.

generalize(subst_term_eq M0 M5 Z M5 n1 n2 H48 H49); intro. rewrite H50; clear H50.

generalize(sub_var_eq p q (Abs(var(m,n))M5) H37); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (cvar (var (p, q))) (Abs(var (m, n)) M5) p q H50); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Abs(var (m, n)) M5)) (Abs(var (m, n)) M5) H51).

generalize(sub_var_eq p q(Abs(var(m,n))M5) H37); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (cvar (var (p, q))) (Abs(var (m, n)) M5) p q H49); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Abs(var (m, n)) M5)) (Abs(var (m, n)) M5) H50).

rewrite <- H37 in H2;rewrite H19 in H2; rewrite H20 in H2.

generalize(sub_var_neq p q m0 n0 M0 H22 H38); intro.

generalize(subst_term_eq M0 (cvar(var(m0,n0))) Q1 (cvar(var(m0,n0))) p q H2 H39); intro.

rewrite H40; clear H40.

generalize(sub_var_neq p q m0 n0 (Abs(var(m,n))M5) H34 H38); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (cvar (var (m0, n0))) (cvar (var (m0, n0))) p q H40); intro.

apply (beta_reduc (App (Abs(var (p, q)) (cvar (var (m0, n0)))) (Abs(var (m, n)) M5)) (cvar (var (m0, n0))) H41).

rewrite <- H36 in H2; rewrite H19 in H2; rewrite H20 in H2.

generalize(sub_abs_nfree0 p q M6 M0 H22); intro.

generalize(subst_term_eq M0 (Abs(var(p,q))M6) Q1 (Abs(var(p,q))M6) p q H2 H38); intro.

rewrite H39; clear H39.

generalize(sub_abs_nfree0 p q M6 (Abs(var(m,n))M5) H37); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (Abs(var (p, q)) M6) (Abs(var (p, q)) M6) p q H39); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M6)) (Abs(var (m, n)) M5)) (Abs(var (p, q)) M6) H40).

rewrite <- H39 in H2. inversion H2. rewrite H19 in H45; rewrite H20 in H46.

generalize(eq_couple p q m0 n0 H45 H46); intro. contradiction.

generalize(sub_abs_nfree2 n1 n2 m n M5 M0 H25 H26 H27 H31); intro.

generalize(sub_abs_nfree1 p q m0 n0 M6 Z (Abs(var(m,n))M5) H34 H35 H36 H40); intro.

generalize(sub_abs_nfree1 n1 n2 m0 n0 Z Z0 M0 H47 H49 H50 H51); intro.

rewrite H19 in H52; rewrite H20 in H52. rewrite H19 in H54; rewrite H20 in H54.

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))M5) (Abs(var(m0,n0))M6) T).

apply(exist_term (Abs(var(m0,n0))M6) (Abs(var(m,n))M5) p q H35). inversion_clear H55.

generalize(sub_eq_Z1_Z2 (Abs(var(m0,n0))M6) (Abs(var(m,n))M5) (Abs(var(m0,n0))Z) (Abs(var(m0,n0))Z0) x

M0 (Abs(var(m,n))M5) p q H22 H35 H52 H53 H54 H56); intro.

rewrite H55 in H56; clear H55.

generalize(beta_red0 (Abs(var (m, n)) M5) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0))Z0) p q H56); intro.

apply(beta_reduc (App (Abs (var(p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) M5)) (Abs(var (m0, n0)) Z0) H55).

generalize(sub_abs_nfree2 n1 n2 m n M5 M0 H25 H26 H27 H31); intro.

generalize(sub_abs_nfree1 p q m0 n0 M6 Z (Abs(var(m,n))M5) H34 H35 H36 H40); intro.

generalize(sub_abs_nfree2 n1 n2 m0 n0 Z M0 H47 H49 H50 H51); intro.

rewrite H19 in H52; rewrite H20 in H52.

rewrite H19 in H54; rewrite H20 in H54.

assert(exists T: c_term, substitution (var(p,q)) (Abs(var(m,n))M5) (Abs(var(m0,n0))M6) T).

apply (exist_term (Abs(var(m0,n0))M6) (Abs(var(m,n))M5) p q H35). inversion_clear H55.

generalize(sub_eq_Z1_Z2 (Abs(var(m0,n0))M6) (Abs(var(m,n))M5) (Abs(var(m0,n0))Z) (Abs(var(m0,n0))Z) x

M0 (Abs(var(m,n))M5) p q H22 H35 H52 H53 H54 H56); intro.

rewrite H55 in H56; clear H55.

generalize(beta_red0 (Abs(var (m, n)) M5) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0)) Z) p q H56); intro.

apply(beta_reduc (App (Abs (var(p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) M5)) (Abs(var (m0, n0)) Z) H55).

rewrite <- H39 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

generalize(eq_couple p q m0 n0 H45 H46); intro. contradiction.

generalize(sub_var_nfree M6 p q M0 H49 H34); intro.

generalize(subst_term_eq M0 M6 Z M6 p q H51 H52); intro. rewrite H53; clear H53.

generalize(sub_abs_nfree2 p q m0 n0 M6 (Abs(var(m,n))M5) H34 H35 H36 H40); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (Abs(var (m0, n0)) M6) (Abs(var (m0, n0)) M6) p q H53); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m0, n0)) M6)) (Abs(var (m, n)) M5)) (Abs(var (m0, n0)) M6) H54).

generalize(var_nfree_M_nfree_Abs p q m0 n0 M6 H47); intro.

generalize(sub_var_nfree (Abs(var (m0, n0))M6) p q (Abs(var (m, n)) M5) H35 H52); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (Abs(var (m0, n0)) M6) (Abs (var(m0, n0)) M6) p q H53); intro.

apply(beta_reduc (App(Abs(var (p, q)) (Abs(var (m0, n0))M6)) (Abs (var(m, n))M5))(Abs(var (m0, n0)) M6) H54).

rewrite <- H38 in H2; rewrite H19 in H2; rewrite H20 in H2.

inversion H2. rewrite H19 in H25; rewrite H20 in H25.

rewrite H19 in H31; rewrite H20 in H31.

generalize(sub_abs_nfree2 p q m n M5 M0 H25 H22 H27 H31); intro.

assert (exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))M5) M6 T).

apply(exist_term M6 (Abs(var(m,n))M5) p q H34). inversion_clear H50.

generalize(sub_eq_Z1_Z2 M6 (Abs(var(m,n))M5) M7 M9 x M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H35 H47 H51); intro.

assert(exists T: c_term, substitution (var(p,q)) (Abs(var(m,n))M5) N3 T).

apply(exist_term N3 (Abs(var(m,n))M5) p q H34). inversion_clear H52.

generalize(sub_eq_Z1_Z2 N3 (Abs(var(m,n))M5) N4 N6 x0 M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H39 H48 H53); intro.

rewrite H50 in H51; clear H50. rewrite H52 in H53; clear H52.

generalize(sub_app p q M6 N3 M9 N6 (Abs(var(m,n))M5) H34 H51 H53); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (App M6 N3) (App M9 N6) p q H50);intro.

apply(beta_reduc (App (Abs(var (p, q)) (App M6 N3)) (Abs (var(m, n)) M5)) (App M9 N6) H52).

rewrite <- H38 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

rewrite H19 in H25; rewrite H20 in H25. rewrite H19 in H31; rewrite H20 in H31.

generalize(sub_abs_nfree2 p q m n M5 M0 H25 H22 H27 H31); intro.

assert (exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))M5) M6 T).

apply(exist_term M6 (Abs(var(m,n))M5) p q H34). inversion_clear H50.

generalize(sub_eq_Z1_Z2 M6 (Abs(var(m,n))M5) M7 M9 x M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H35 H47 H51); intro.

assert(exists T: c_term, substitution (var(p,q)) (Abs(var(m,n))M5) N3 T).

apply(exist_term N3 (Abs(var(m,n))M5) p q H34). inversion_clear H52.

generalize(sub_eq_Z1_Z2 N3 (Abs(var(m,n))M5) N4 N6 x0 M0 (Abs(var(m,n))M5) p q

H22 H34 H49 H39 H48 H53); intro.

rewrite H50 in H51; clear H50. rewrite H52 in H53; clear H52.

generalize(sub_imp p q M6 N3 M9 N6 (Abs(var(m,n))M5) H34 H51 H53); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (Imp M6 N3) (Imp M9 N6) p q H50);intro.

apply(beta_reduc (App (Abs(var (p, q)) (Imp M6 N3)) (Abs (var(m, n)) M5)) (Imp M9 N6) H52).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H25; rewrite H20 in H25. rewrite H19 in H31; rewrite H20 in H31.

generalize(sub_abs_nfree2 p q m n M5 M0 H25 H22 H27 H31); intro.

assert(exists T:c_term, substitution (var(p,q)) (Abs(var(m,n))M5) M6 T).

apply(exist_term M6 (Abs(var(m,n))M5) p q H34). inversion_clear H47.

generalize(sub_eq_Z1_Z2 M6 (Abs(var(m,n))M5) N3 N4 x M0 (Abs(var(m,n))M5) p q

H22 H34 H46 H38 H45 H48); intro.

rewrite H47 in H48; clear H47.

generalize(sub_pi p q M6 (Abs(var(m,n))M5) N4 H34 H48); intro.

generalize(beta_red0 (Abs(var (m, n)) M5) (PI M6) (PI N4) p q H47); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M6)) (Abs(var (m, n)) M5)) (PI N4) H49).

rewrite <- H28 in H3; inversion H3. rewrite <- H35 in H2; inversion H2.

generalize(subst_term_eq M0 M5 M6 M8 n1 n2 H26 H44); intro.

generalize(subst_term_eq M0 N3 N4 N6 n1 n2 H30 H45); intro. rewrite H46; rewrite H47.

rewrite H19 in H44; rewrite H20 in H44; rewrite H19 in H45; rewrite H20 in H45.

generalize(sub_app p q M5 N3 M8 N6 M0 H22 H44 H45); intro.

generalize(subst_level (App M5 N3) M0 (App M8 N6) p q H22 H48); intro.

assert (level(App M8 N6) <= q). apply(le_trans (level(App M8 N6)) (level (App M5 N3)) q H49 H36).

generalize(sub_var_eq p q (App M8 N6) H50); intro.

generalize(beta_red0 (App M8 N6) (cvar (var (p, q))) (App M8 N6) p q H51); intro.

apply (beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (App M8 N6)) (App M8 N6) H52).

rewrite<- H36 in H2; inversion H2. rewrite H19 in H42; rewrite H20 in H43.

generalize(eq_couple p q m n H42 H43); intro. contradiction.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H46); intro.

assert(level (App M6 N4) <= q). apply(le_trans (level (App M6 N4)) (level (App M5 N3)) q H47 H33).

generalize(sub_var_neq p q m n (App M6 N4) H48 H37); intro.

generalize(beta_red0 (App M6 N4) (cvar (var (m, n))) (cvar (var (m, n))) p q H49); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (m, n)))) (App M6 N4)) (cvar (var (m, n))) H50).

rewrite <- H35 in H2; rewrite H19 in H2; rewrite H20 in H2.

generalize (sub_abs_nfree0 p q M7 M0 H22); intro.

generalize(subst_term_eq M0 (Abs(var(p,q))M7) Q1 (Abs(var(p,q))M7) p q H2 H37); intro.

rewrite H38; clear H38. rewrite H19 in H26; rewrite H20 in H26.

rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H38); intro.

assert(level (App M6 N4) <= q). apply(le_trans (level (App M6 N4)) (level (App M5 N3)) q H39 H36).

generalize(sub_abs_nfree0 p q M7 (App M6 N4) H40); intro.

generalize(beta_red0 (App M6 N4) (Abs (var(p, q)) M7) (Abs(var (p, q)) M7) p q H41); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M7)) (App M6 N4)) (Abs(var (p, q)) M7) H42).

rewrite <- H38 in H2. rewrite H19 in H2; rewrite H20 in H2; inversion H2.

generalize(eq_couple p q m n H44 H45); intro. contradiction.

rewrite H19 in H26; rewrite H20 in H26.

rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(sub_abs_nfree1 p q m n M7 Z (App M5 N3) H33 H34 H35 H39); intro.

generalize(sub_abs_nfree1 p q m n Z Z0 M0 H46 H48 H49 H50); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H51); intro.

assert(level (App M6 N4) <= q). apply(le_trans (level (App M6 N4)) (level (App M5 N3)) q H54 H34).

assert (exists T:c_term, substitution (var(p,q)) (App M6 N4) (Abs(var(m,n))M7) T).

apply(exist_term (Abs(var(m,n))M7) (App M6 N4) p q H55). inversion_clear H56.

generalize(sub_eq_Z1_Z2 (Abs(var(m,n))M7) (App M6 N4) (Abs(var(m,n))Z) (Abs(var(m,n))Z0) x

M0 (App M5 N3) p q H22 H34 H51 H52 H53 H57); intro.

rewrite H56 in H57; clear H56.

generalize(beta_red0 (App M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n)) Z0) p q H57); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M7)) (App M6 N4)) (Abs(var (m, n))Z0) H56).

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(sub_abs_nfree1 p q m n M7 Z (App M5 N3) H33 H34 H35 H39); intro.

generalize(sub_abs_nfree2 p q m n Z M0 H46 H48 H49 H50); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H51); intro.

assert (level (App M6 N4) <= q). apply(le_trans (level(App M6 N4)) (level (App M5 N3)) q H54 H34).

assert(exists T:c_term, substitution (var(p,q)) (App M6 N4) (Abs(var(m,n))M7) T).

apply(exist_term (Abs(var(m,n))M7) (App M6 N4) p q H55). inversion_clear H56.

generalize(sub_eq_Z1_Z2 (Abs(var(m,n))M7) (App M6 N4) (Abs(var(m,n))Z) (Abs(var(m,n))Z)

x M0 (App M5 N3) p q H22 H34 H51 H52 H53 H57); intro.

rewrite H56 in H57; clear H56.

generalize(beta_red0(App M6 N4) (Abs (var(m, n)) M7) (Abs(var (m, n))Z) p q H57); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M7)) (App M6 N4)) (Abs(var (m, n)) Z) H56).

rewrite <- H38 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

generalize(eq_couple p q m n H44 H45); intro. contradiction.

generalize(sub_var_nfree M7 p q M0 H22 H33); intro.

generalize(subst_term_eq M0 M7 Z M7 p q H50 H51); intro.

rewrite H52; clear H52. rewrite H19 in H26; rewrite H20 in H26.

rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H52); intro.

assert(level (App M6 N4) <= q). apply(le_trans (level (App M6 N4)) (level (App M5 N3)) q H53 H34).

generalize(var_nfree_M_nfree_Abs p q m n M7 H33); intro.

generalize(sub_var_nfree (Abs(var(m,n))M7) p q (App M6 N4) H54 H55); intro.

generalize(beta_red0 (App M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n)) M7) p q H56); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M7)) (App M6 N4)) (Abs (var(m, n)) M7) H57).

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H51); intro.

assert(level(App M6 N4) <= q). apply(le_trans (level (App M6 N4)) (level(App M5 N3)) q H52 H34).

generalize(var_nfree_M_nfree_Abs p q m n M7 H33); intro.

generalize(sub_var_nfree (Abs(var(m,n))M7) p q (App M6 N4) H53 H54); intro.

generalize(beta_red0 (App M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n)) M7) p q H55); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M7)) (App M6 N4)) (Abs(var (m, n)) M7) H56).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H48); intro.

assert(level(App M6 N4) <= q). apply (le_trans (level (App M6 N4)) (level(App M5 N3)) q H49 H33).

assert(exists T:c_term, substitution (var(p,q)) (App M6 N4) M7 T).

apply (exist_term M7 (App M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 M7 (App M6 N4) M8 M10 x M0 (App M5 N3) p q

H22 H33 H48 H34 H46 H52); intro.

rewrite H51 in H52; clear H51. assert(exists T:c_term, substitution (var(p,q)) (App M6 N4) N5 T).

apply(exist_term N5 (App M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 N5 (App M6 N4) N6 N8 x0 M0 (App M5 N3) p q

H22 H33 H48 H38 H47 H53); intro. rewrite H51 in H53; clear H51.

generalize(sub_app p q M7 N5 M10 N8 (App M6 N4) H50 H52 H53); intro.

generalize(beta_red0 (App M6 N4) (App M7 N5) (App M10 N8) p q H51); intro.

apply (beta_reduc (App (Abs(var (p, q)) (App M7 N5)) (App M6 N4)) (App M10 N8) H54).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H48); intro.

assert(level(App M6 N4) <= q). apply (le_trans (level (App M6 N4)) (level(App M5 N3)) q H49 H33).

assert(exists T:c_term, substitution (var(p,q)) (App M6 N4) M7 T).

apply(exist_term M7 (App M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 M7 (App M6 N4) M8 M10 x M0 (App M5 N3) p q

H22 H33 H48 H34 H46 H52); intro.

rewrite H51 in H52; clear H51.

assert(exists T:c_term, substitution (var(p,q)) (App M6 N4) N5 T).

apply(exist_term N5 (App M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 N5 (App M6 N4) N6 N8 x0 M0 (App M5 N3) p q

H22 H33 H48 H38 H47 H53); intro.

rewrite H51 in H53; clear H51.

generalize(sub_imp p q M7 N5 M10 N8 (App M6 N4) H50 H52 H53); intro.

generalize(beta_red0 (App M6 N4) (Imp M7 N5) (Imp M10 N8) p q H51); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Imp M7 N5)) (App M6 N4)) (Imp M10 N8) H54).

rewrite <- H36 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_app p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (App M5 N3) M0 (App M6 N4) p q H22 H45); intro.

assert(level (App M6 N4) <= q). apply (le_trans (level (App M6 N4)) (level(App M5 N3)) q H46 H33).

assert(exists T:c_term, substitution (var(p,q)) (App M6 N4) M7 T).

apply(exist_term M7 (App M6 N4) p q H47). inversion_clear H48.

generalize(sub_eq_Z1_Z2 M7 (App M6 N4) N5 N6 x M0 (App M5 N3) p q

H22 H33 H45 H37 H44 H49); intro.

rewrite H48 in H49; clear H48.

generalize(sub_pi p q M7 (App M6 N4) N6 H47 H49); intro.

generalize(beta_red0 (App M6 N4) (PI M7) (PI N6) p q H48); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M7)) (App M6 N4)) (PI N6) H50).

rewrite <- H28 in H3; inversion H3. rewrite <- H35 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2. rewrite H19 in H26; rewrite H20 in H26.

rewrite H19 in H30; rewrite H20 in H30.

generalize(subst_term_eq M0 M5 M6 M8 p q H26 H44); intro.

generalize(subst_term_eq M0 N3 N4 N6 p q H30 H45); intro.

rewrite H46; clear H46. rewrite H47; clear H47.

generalize(sub_imp p q M5 N3 M8 N6 M0 H42 H44 H45); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M8 N6) p q H42 H46); intro.

assert(level(Imp M8 N6) <= q). apply(le_trans (level(Imp M8 N6)) (level(Imp M5 N3)) q H47 H36).

generalize(sub_var_eq p q (Imp M8 N6) H48); intro.

generalize(beta_red0 (Imp M8 N6) (cvar (var (p, q))) (Imp M8 N6) p q H49); intro.

apply (beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (Imp M8 N6))(Imp M8 N6) H50).

rewrite <- H36 in H2; rewrite H19 in H2; rewrite H20 in H2.

generalize(sub_var_neq p q m n M0 H22 H37); intro.

generalize(subst_term_eq M0 (cvar(var(m,n))) Q1 (cvar(var(m,n))) p q H2 H38); intro.

rewrite H39; clear H39. rewrite H19 in H26; rewrite H20 in H26.

rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H39); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H40 H33).

generalize(sub_var_neq p q m n (Imp M6 N4) H41 H37); intro.

generalize(beta_red0 (Imp M6 N4) (cvar (var (m, n))) (cvar (var (m, n))) p q H42); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (m, n)))) (Imp M6 N4)) (cvar (var (m, n))) H43).

rewrite <- H35 in H2; rewrite H19 in H2; rewrite H20 in H2.

generalize(sub_abs_nfree0 p q M7 M0 H22); intro.

generalize(subst_term_eq M0 (Abs(var(p,q))M7) Q1 (Abs(var(p,q))M7) p q H2 H37); intro.

rewrite H38; clear H38. rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H38); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H39 H36).

generalize(sub_abs_nfree0 p q M7 (Imp M6 N4) H40); intro.

generalize(beta_red0 (Imp M6 N4) (Abs(var (p, q)) M7) (Abs(var (p, q)) M7) p q H41); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M7)) (Imp M6 N4)) (Abs(var (p, q)) M7) H42).

rewrite <- H38 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

generalize(eq_couple p q m n H44 H45); intro. contradiction.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(sub_abs_nfree1 p q m n M7 Z (Imp M5 N3) H33 H34 H35 H39); intro.

generalize(sub_abs_nfree1 p q m n Z Z0 M0 H46 H48 H49 H50); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H51); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H54 H34).

assert (exists T: c_term, substitution (var(p,q)) (Imp M6 N4) (Abs(var(m,n))M7) T).

apply(exist_term (Abs(var(m,n))M7) (Imp M6 N4) p q H55).

inversion_clear H56.

generalize(sub_eq_Z1_Z2 (Abs(var(m,n))M7) (Imp M6 N4) (Abs(var(m,n))Z) (Abs(var(m,n))Z0)

x M0 (Imp M5 N3) p q H22 H34 H51 H52 H53 H57); intro.

rewrite H56 in H57; clear H56.

generalize(beta_red0 (Imp M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n)) Z0) p q H57); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M7)) (Imp M6 N4)) (Abs(var (m, n))Z0) H56).

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(sub_abs_nfree1 p q m n M7 Z (Imp M5 N3) H33 H34 H35 H39); intro.

generalize(sub_abs_nfree2 p q m n Z M0 H46 H48 H49 H50); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H51); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level (Imp M6 N4)) (level(Imp M5 N3)) q H54 H34).

assert (exists T: c_term, substitution (var(p,q)) (Imp M6 N4) (Abs(var(m,n))M7) T).

apply(exist_term (Abs(var(m,n))M7) (Imp M6 N4) p q H55). inversion_clear H56.

generalize(sub_eq_Z1_Z2 (Abs(var(m,n))M7) (Imp M6 N4) (Abs(var(m,n))Z) (Abs(var(m,n))Z)

x M0 (Imp M5 N3) p q H22 H34 H51 H52 H53 H57); intro.

rewrite H56 in H57; clear H56.

generalize(beta_red0 (Imp M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n))Z) p q H57); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs (var(m, n)) M7)) (Imp M6 N4)) (Abs(var (m, n)) Z) H56).

rewrite <- H38 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

generalize(eq_couple p q m n H44 H45); intro. contradiction.

generalize(sub_var_nfree M7 p q M0 H22 H33); intro.

generalize(subst_term_eq M0 M7 Z M7 p q H50 H51); intro.

rewrite H52; clear H52. rewrite H19 in H26; rewrite H20 in H26.

rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H52); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H53 H34).

generalize(var_nfree_M_nfree_Abs p q m n M7 H33); intro.

generalize(sub_var_nfree (Abs(var (m, n)) M7) p q (Imp M6 N4) H54 H55); intro.

generalize(beta_red0 (Imp M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n)) M7) p q H56); intro.

apply (beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M7)) (Imp M6 N4)) (Abs(var (m, n))M7) H57).

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H51); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H52 H34).

generalize(var_nfree_M_nfree_Abs p q m n M7 H46); intro.

generalize(sub_var_nfree (Abs(var (m, n)) M7) p q (Imp M6 N4) H53 H54); intro.

generalize(beta_red0 (Imp M6 N4) (Abs(var (m, n)) M7) (Abs(var (m, n)) M7) p q H55); intro.

apply (beta_reduc (App (Abs (var(p, q)) (Abs(var (m, n)) M7)) (Imp M6 N4)) (Abs(var (m, n))M7) H56).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H48); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H49 H33).

assert(exists T:c_term, substitution (var(p,q)) (Imp M6 N4) M7 T).

apply(exist_term M7 (Imp M6 N4) p q H50).

inversion_clear H51.

generalize(sub_eq_Z1_Z2 M7 (Imp M6 N4) M8 M10 x M0 (Imp M5 N3) p q

H22 H33 H48 H34 H46 H52); intro.

rewrite H51 in H52; clear H51.

assert(exists T:c_term, substitution (var(p,q)) (Imp M6 N4) N5 T).

apply(exist_term N5 (Imp M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 N5 (Imp M6 N4) N6 N8 x0 M0 (Imp M5 N3) p q

H22 H33 H48 H38 H47 H53); intro.

rewrite H51 in H53; clear H51.

generalize(sub_app p q M7 N5 M10 N8 (Imp M6 N4) H50 H52 H53);intro.

generalize(beta_red0 (Imp M6 N4) (App M7 N5) (App M10 N8) p q H51); intro.

apply(beta_reduc (App (Abs(var (p, q)) (App M7 N5)) (Imp M6 N4)) (App M10 N8) H54).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H48); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H49 H33).

assert(exists T:c_term, substitution (var(p,q)) (Imp M6 N4) M7 T).

apply(exist_term M7 (Imp M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 M7 (Imp M6 N4) M8 M10 x M0 (Imp M5 N3) p q

H22 H33 H48 H34 H46 H52); intro.

rewrite H51 in H52; clear H51.

assert(exists T:c_term, substitution (var(p,q)) (Imp M6 N4) N5 T).

apply(exist_term N5 (Imp M6 N4) p q H50). inversion_clear H51.

generalize(sub_eq_Z1_Z2 N5 (Imp M6 N4) N6 N8 x0 M0 (Imp M5 N3) p q

H22 H33 H48 H38 H47 H53); intro.

rewrite H51 in H53; clear H51.

generalize(sub_imp p q M7 N5 M10 N8 (Imp M6 N4) H50 H52 H53); intro.

generalize(beta_red0 (Imp M6 N4) (Imp M7 N5) (Imp M10 N8) p q H51); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Imp M7 N5)) (Imp M6 N4)) (Imp M10 N8) H54).

rewrite <- H36 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H26; rewrite H20 in H26. rewrite H19 in H30; rewrite H20 in H30.

generalize(sub_imp p q M5 N3 M6 N4 M0 H22 H26 H30); intro.

generalize(subst_level (Imp M5 N3) M0 (Imp M6 N4) p q H22 H45); intro.

assert(level(Imp M6 N4) <= q). apply(le_trans (level(Imp M6 N4)) (level(Imp M5 N3)) q H46 H33).

assert(exists T:c_term, substitution (var(p,q)) (Imp M6 N4) M7 T).

apply(exist_term M7 (Imp M6 N4) p q H47). inversion_clear H48.

generalize(sub_eq_Z1_Z2 M7 (Imp M6 N4) N5 N6 x M0 (Imp M5 N3) p q

H22 H33 H45 H37 H44 H49); intro.

rewrite H48 in H49; clear H48.

generalize(sub_pi p q M7 (Imp M6 N4) N6 H47 H49); intro.

generalize(beta_red0 (Imp M6 N4) (PI M7) (PI N6)p q H48); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M7)) (Imp M6 N4)) (PI N6) H50).

rewrite <- H27 in H3; inversion H3.

rewrite <- H34 in H2; rewrite H19 in H2; rewrite H20 in H2.

rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_term_eq M0 (PI M5) Q1 (PI N3) p q H2 H36); intro. rewrite H37; clear H37.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H36); intro.

assert(level (PI N3) <= q). apply(le_trans (level(PI N3)) (level(PI M5)) q H37 H35).

generalize(sub_var_eq p q (PI N3) H38); intro.

generalize(beta_red0 (PI N3) (cvar (var (p, q))) (PI N3) p q H39); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (p, q)))) (PI N3)) (PI N3) H40).

rewrite <- H35 in H2; rewrite H19 in H2; rewrite H20 in H2.

generalize(sub_var_neq p q m n M0 H22 H36); intro.

generalize(subst_term_eq M0 (cvar(var(m,n))) Q1 (cvar(var(m,n))) p q H2 H37); intro.

rewrite H38; clear H38. rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H38); intro.

assert(level (PI N3) <= q). apply(le_trans (level(PI N3)) (level(PI M5)) q H39 H32).

generalize(sub_var_neq p q m n (PI N3) H40 H36); intro.

generalize(beta_red0 (PI N3) (cvar (var (m, n))) (cvar (var (m, n))) p q H41); intro.

apply(beta_reduc (App (Abs(var (p, q)) (cvar (var (m, n)))) (PI N3)) (cvar (var (m, n))) H42).

rewrite <- H34 in H2; rewrite H19 in H2; rewrite H20 in H2.

generalize (sub_abs_nfree0 p q M6 M0 H22); intro.

generalize(subst_term_eq M0 (Abs(var(p,q))M6) Q1(Abs(var(p,q))M6) p q H2 H36); intro.

rewrite H37; clear H37. rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H37); intro.

assert(level (PI N3) <= q). apply(le_trans (level(PI N3)) (level(PI M5)) q H38 H35).

generalize(sub_abs_nfree0 p q M6 (PI N3) H39); intro.

generalize(beta_red0 (PI N3) (Abs(var (p, q)) M6) (Abs(var (p, q)) M6) p q H40); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (p, q)) M6)) (PI N3)) (Abs(var (p, q)) M6) H41).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

generalize(eq_couple p q m n H43 H44); intro. contradiction.

rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(sub_abs_nfree1 p q m n M6 Z (PI M5) H32 H33 H34 H38); intro.

generalize(sub_abs_nfree1 p q m n Z Z0 M0 H45 H47 H48 H49); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H50); intro.

assert(level (PI N3) <= q). apply(le_trans (level(PI N3)) (level(PI M5)) q H53 H33).

assert(exists T:c_term, substitution (var(p,q)) (PI N3) (Abs(var(m,n))M6) T).

apply(exist_term (Abs(var(m,n))M6) (PI N3) p q H54).

inversion_clear H55.

generalize(sub_eq_Z1_Z2 (Abs(var(m,n))M6) (PI N3) (Abs(var(m,n))Z) (Abs(var(m,n))Z0) x

M0 (PI M5) p q H22 H33 H50 H51 H52 H56); intro.

rewrite H55 in H56; clear H55.

generalize(beta_red0 (PI N3) (Abs(var (m, n)) M6) (Abs(var (m, n)) Z0) p q H56); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (PI N3)) (Abs(var (m, n)) Z0) H55).

rewrite H19 in H29; rewrite H20 in H29. generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(sub_abs_nfree1 p q m n M6 Z (PI M5) H32 H33 H34 H38); intro.

generalize(sub_abs_nfree2 p q m n Z M0 H45 H47 H48 H49); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H50); intro. assert(level (PI N3) <= q).

apply(le_trans (level(PI N3)) (level(PI M5)) q H53 H33).

assert(exists T:c_term, substitution (var(p,q)) (PI N3) (Abs(var(m,n))M6) T).

apply (exist_term (Abs(var(m,n))M6) (PI N3) p q H54). inversion_clear H55.

generalize(sub_eq_Z1_Z2 (Abs(var(m,n))M6) (PI N3) (Abs(var(m,n))Z) (Abs(var(m,n))Z) x M0

(PI M5) p q H22 H33 H50 H51 H52 H56); intro.

rewrite H55 in H56; clear H55.

generalize(beta_red0(PI N3) (Abs(var (m, n)) M6) (Abs(var (m, n)) Z) p q H56); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (PI N3)) (Abs(var (m, n)) Z) H55).

rewrite <- H37 in H2; rewrite H19 in H2; rewrite H20 in H2. inversion H2.

generalize(eq_couple p q m n H43 H44); intro. contradiction.

generalize(sub_var_nfree M6 p q M0 H22 H32); intro.

generalize(subst_term_eq M0 M6 Z M6 p q H49 H50); intro.

rewrite H51; clear H51. rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H51); intro.

assert(level (PI N3) <= q).

apply(le_trans (level(PI N3)) (level(PI M5)) q H52 H33).

generalize(var_nfree_M_nfree_Abs p q m n M6 H32); intro.

generalize(sub_var_nfree (Abs(var (m, n)) M6) p q (PI N3) H53 H54); intro.

generalize(beta_red0 (PI N3) (Abs (var(m, n)) M6) (Abs(var (m, n)) M6) p q H55);intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (PI N3)) (Abs(var (m, n)) M6) H56).

rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H50); intro.

assert(level (PI N3) <= q).

apply(le_trans (level(PI N3)) (level(PI M5)) q H51 H33).

generalize(var_nfree_M_nfree_Abs p q m n M6 H45); intro.

generalize(sub_var_nfree (Abs(var (m, n)) M6) p q(PI N3) H52 H53); intro.

generalize(beta_red0(PI N3) (Abs(var (m, n))M6) (Abs(var (m, n)) M6) p q H54);intro.

apply(beta_reduc (App (Abs(var (p, q)) (Abs(var (m, n)) M6)) (PI N3)) (Abs(var (m, n)) M6) H55).

rewrite <- H36 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H47); intro.

assert(level (PI N3) <= q). apply(le_trans (level(PI N3)) (level(PI M5)) q H48 H32).

assert(exists T:c_term, substitution (var(p,q)) (PI N3) M6 T).

apply(exist_term M6 (PI N3) p q H49).

inversion_clear H50.

generalize(sub_eq_Z1_Z2 M6 (PI N3) M7 M9 x M0 (PI M5) p q H22 H32 H47 H33 H45 H51); intro.

rewrite H50 in H51; clear H50.

assert(exists T: c_term, substitution (var(p,q)) (PI N3) N4 T).

apply(exist_term N4 (PI N3) p q H49).

inversion_clear H50.

generalize(sub_eq_Z1_Z2 N4 (PI N3) N5 N7 x0 M0 (PI M5) p q H22

H32 H47 H37 H46 H52); intro.

rewrite H50 in H52; clear H50.

generalize(sub_app p q M6 N4 M9 N7 (PI N3) H49 H51 H52); intro.

generalize(beta_red0 (PI N3) (App M6 N4) (App M9 N7) p q H50); intro.

apply(beta_reduc (App(Abs(var (p, q))(App M6 N4)) (PI N3)) (App M9 N7) H53).

rewrite <- H36 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H47); intro.

assert(level (PI N3) <= q).

apply(le_trans (level(PI N3)) (level(PI M5)) q H48 H32).

assert(exists T:c_term, substitution (var(p,q)) (PI N3) M6 T).

apply(exist_term M6 (PI N3) p q H49).

inversion_clear H50.

generalize(sub_eq_Z1_Z2 M6 (PI N3) M7 M9 x M0 (PI M5) p q

H22 H32 H47 H33 H45 H51); intro.

rewrite H50 in H51; clear H50.

assert(exists T: c_term, substitution (var(p,q)) (PI N3) N4 T).

apply(exist_term N4 (PI N3) p q H49).

inversion_clear H50.

generalize(sub_eq_Z1_Z2 N4 (PI N3) N5 N7 x0 M0 (PI M5) p q H22

H32 H47 H37 H46 H52); intro.

rewrite H50 in H52; clear H50.

generalize(sub_imp p q M6 N4 M9 N7 (PI N3) H49 H51 H52); intro.

generalize(beta_red0 (PI N3) (Imp M6 N4) (Imp M9 N7) p q H50); intro.

apply(beta_reduc (App (Abs(var (p, q)) (Imp M6 N4)) (PI N3)) (Imp M9 N7) H53).

rewrite <- H35 in H2; rewrite H19 in H2; rewrite H20 in H2; inversion H2.

rewrite H19 in H29; rewrite H20 in H29.

generalize(sub_pi p q M5 M0 N3 H22 H29); intro.

generalize(subst_level (PI M5) M0 (PI N3) p q H22 H44); intro.

assert(level (PI N3) <= q).

apply(le_trans (level(PI N3)) (level(PI M5)) q H45 H32).

generalize(sub_pi p q M6 (PI M5) N4 H32 H36);intro.

generalize(sub_pi p q N4 M0 N5 H22 H43); intro.

assert(exists T:c_term, substitution (var(p,q)) (PI N3) (PI M6) T).

apply(exist_term (PI M6) (PI N3) p q H46). inversion_clear H49.

generalize(sub_eq_Z1_Z2 (PI M6) (PI N3) (PI N4) (PI N5) x M0 (PI M5)

p q H22 H32 H44 H47 H48 H50); intro.

rewrite H49 in H50; clear H49.

generalize(beta_red0 (PI N3) (PI M6) (PI N5) p q H50); intro.

apply(beta_reduc (App (Abs(var (p, q)) (PI M6)) (PI N3)) (PI N5) H49).

generalize(level_subst M1 N0 N p q H3); intro.

generalize(subst_level N0 M0 N2 n1 n2 H11 H14); intro.

assert(level N2 <= q). apply(le_trans (level N2) (level N0) q H27 H26).

assert(exists T:c_term, substitution (var(p,q)) N2 Z T).

apply (exist_term Z N2 p q H28). inversion_clear H29.

generalize(inv_neq_couple n1 n2 p q H24); intro.

generalize(neq_var n1 n2 p q H29);intro.

generalize(sub_distrib n1 n2 p q M0 N0 M1 N Q1 N2 Z x H31 H21 H23

H26 H3 H2 H14 H25 H30);intro.

rewrite H32; clear H32. generalize(beta_red0 N2 Z x p q H30); intro.

apply(beta_reduc (App (Abs(var (p, q)) Z) N2) x H32). inversion H13.

generalize(eq_couple n1 n2 p q H30 H31); intro. contradiction.

generalize(sub_var_nfree M1 n1 n2 M0 H11 H21); intro.

generalize(level_subst M1 N0 N p q H3); intro.

generalize(subst_level N0 M0 N2 n1 n2 H34 H14); intro.

assert(level N2 <= q). apply(le_trans (level N2) (level N0) q H39 H38).

assert(exists T:c_term, substitution (var(p,q)) N2 M1 T).

apply(exist_term M1 N2 p q H40). inversion_clear H41.

generalize(inv_neq_couple n1 n2 p q H25); intro.

generalize(neq_var n1 n2 p q H41); intro.

generalize(sub_distrib n1 n2 p q M0 N0 M1 N Q1 N2 M1 x H43 H32 H34

H38 H3 H2 H14 H37 H42); intro.

rewrite H44; clear H44. generalize(beta_red0 N2 M1 x p q H42); intro.

apply (beta_reduc (App (Abs(var (p, q)) M1) N2) x H44).

generalize(sub_var_nfree M1 n1 n2 M0 H11 H21); intro.

generalize(subst_level N0 M0 N2 n1 n2 H11 H14); intro.

generalize(level_subst M1 N0 N p q H3); intro.

assert(level N2 <= q). apply(le_trans (level N2) (level N0) q H38 H39).

generalize(exist_term M1 N2 p q H40); intro. inversion_clear H41.

generalize(inv_neq_couple n1 n2 p q H36); intro.

generalize(neq_var n1 n2 p q H41); intro.

generalize(sub_distrib n1 n2 p q M0 N0 M1 N Q1 N2 M1 x H43 H35 H34 H39 H3 H2 H14 H37 H42); intro.

rewrite <- H44 in H42. generalize(beta_red0 N2 M1 Q1 p q H42); intro.

apply(beta_reduc (App (Abs(var (p, q))M1) N2) Q1 H45).

*(****************************** Abs ***************************************)*

intros. inversion H1; inversion H2. apply (Abs_reduc M N p q H).

generalize (eq_couple n1 n2 p q H7 H8); intro. contradiction. apply(Abs_reduc M N p q H).

generalize (eq_couple n1 n2 p q H18 H19); intro. contradiction.

generalize (IHEbeta_reduc M0 Z Z0 n1 n2 H0 H13 H24); intro. apply Abs_reduc. exact H25.

generalize (sub_var_nfree N n1 n2 M0 H22 H20); intro.

generalize (IHEbeta_reduc M0 Z N n1 n2 H0 H13 H25); intro.

apply (Abs_reduc Z N p q H26). apply(Abs_reduc M N p q H).

generalize (sub_var_nfree M n1 n2 M0 H22 H9); intro.

generalize (IHEbeta_reduc M0 M Z n1 n2 H0 H25 H24); intro.

apply (Abs_reduc M Z p q H26). generalize (sub_var_nfree M n1 n2 M0 H0 H9); intro.

generalize (sub_var_nfree N n1 n2 M0 H0 H20); intro.

generalize (IHEbeta_reduc M0 M N n1 n2 H0 H25 H26); intro. apply (Abs_reduc M N p q H27).

*(********************************** PI ************************************)*

intros. inversion H1. inversion H2.

generalize (IHEbeta_reduc M0 N0 N1 n1 n2 H0 H9 H16); intro.

apply PI_reduc. exact H17.

*(****************************** transitivité ******************************)*

intros. assert(exists N':c_term, substitution (var(n1,n2)) M0 N N').

apply(exist_term N M0 n1 n2 H1). inversion_clear H4.

generalize(IHEbeta_reduc1 M0 P1 x n1 n2 H1 H2 H5); intro.

generalize(IHEbeta_reduc2 M0 x Q1 n1 n2 H1 H5 H3); intro.

apply(trans_reduc P1 x Q1 H4 H6).

Qed.

Annexe C

Les lemmes intermédiaires qui aident à démontrer l'équivalence entre la relation Ebeta_reduc et →mcd_star :

Lemme 1 : (Proposition 1 de chapitre 3)

Lemma mcd_star_appl: forall M N:c_term, mcd_star M N -> forall Z:_, mcd_star (App Z M) (App Z N).

Proof.

induction 1. intro Z ; apply refl_red.

intro Z. assert(MCD Z Z). apply (mcd_ref Z).

generalize(mcd_app Z x Z y H1 H); intro. generalize (IHmcd_star Z); intro.

apply (trans_red (App Z x) (App Z y) (App Z z) H2 H3).

Qed.

Lemme 2 : (Proposition 2 de chapitre 3)

Lemma mcd_star_appr: forall M N:c_term, mcd_star M N -> forall Z: _, mcd_star (App M Z) (App N Z).

Proof.

induction 1. intro Z ; apply refl_red. intro Z. assert(MCD Z Z). apply (mcd_ref Z).

generalize(mcd_app x Z y Z H H1); intro. generalize (IHmcd_star Z); intro.

apply (trans_red (App x Z) (App y Z) (App z Z) H2 H3).

Qed.

Lemme 3: (Proposition 3 de chapitre 3)

Lemma mcd_star_impl: forall M N:c_term, mcd_star M N -> forall Z:_, mcd_star (Imp Z M) (Imp Z N).

Proof.

induction 1. intro Z ; apply refl_red. intro Z. assert(MCD Z Z). apply (mcd_ref Z).

generalize(mcd_imp Z x Z y H1 H); intro. generalize (IHmcd_star Z); intro.

apply (trans_red (Imp Z x) (Imp Z y) (Imp Z z) H2 H3). Qed.

Lemme 4 : (Proposition 4 de chapitre 3)

Lemma mcd_star_impr: forall M N:c_term, mcd_star M N -> forall Z: _, mcd_star (Imp M Z) (Imp N Z).

Proof.

induction 1. intro Z ; apply refl_red. intro Z. assert(MCD Z Z). apply (mcd_ref Z).

generalize(mcd_imp x Z y Z H H1); intro. generalize (IHmcd_star Z); intro.

apply (trans_red (Imp x Z) (Imp y Z) (Imp z Z) H2 H3).

Qed.

Lemme 5 : (Proposition 5 de chapitre 3)

Lemma mcd_star_abs: forall M N:c_term, mcd_star M N -> forall p q:_, mcd_star (Abs(var(p,q)) M) (Abs(var(p,q)) N).

Proof.

induction 1;intros. apply (refl_red (Abs(var(p,q))x)).

generalize(mcd_abs x y p q H); intro. generalize(IHmcd_star p q); intro.

apply(trans_red (Abs (var(p, q)) x) (Abs(var (p, q)) y) (Abs(var (p, q)) z) H1 H2).

Qed.

Lemme 6 : (Proposition 6 de chapitre 3)

Lemma mcd_star_pi: forall M N:_, mcd_star M N -> mcd_star (PI M) (PI N).

Proof.

induction 1. apply(refl_red (PI x)).

generalize(mcd_pi x y H); intro. apply(trans_red (PI x) (PI y) (PI z) H1 IHmcd_star).

Qed.

Lemme 7 : (Proposition 7 de chapitre 3)

Lemma inc_mcd_mcd_star: forall x y:c_term, MCD x y -> mcd_star x y.

Proof.

intros. assert(mcd_star y y -> mcd_star x y). apply trans_red. assumption.

apply H0; apply (refl_red y).

Qed.

Lemme 8 : (Proposition 8 de chapitre 3)

Lemma mcd_star_trans: forall x y z:c_term, mcd_star x y -> mcd_star y z -> mcd_star x z.

Proof.

induction 1; intros. assumption.

apply trans_red with y. assumption.

apply IHmcd_star. exact H1.

Qed.