République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université M'Hamed Bougarra Boumerdès

Faculté des Sciences

Département d'Informatique

Domaine : Mathématique et Informatique

Filière : Informatique

Spécialité : Ingénierie de logiciels et traitement de l'information

Projet de Fin d'études Master

Thème

Spécification et preuve dans Coq de la propriété de Church-Rosser de la E$\lambda\beta$-réduction

---

Réalisé par :

BELHI Abdelhak

ELHADDAD Mohamed Yacine

Encadré par : Pr. MEZGHICHE Mohamed

Soutenu le : 23-06-2016

Devant le jury composé de :

-Mr AIT-BOUZIAD Ahmed — Président

-Mr MEZGHICHE Mohamed — Encadreur

-Mr CHAABANI Mohamed — Membre

Année universitaire 2015/2016

Table des matières

Table des figures

Introduction

Le λ-calcul est un formalisme introduit par Church dans les années 1930 dans le but de fournir un fondement aux mathématiques comme alternative à la théorie des ensembles. De nos jours,il est utilisé dans la théorie de la calculabilité pour exprimer les fonctions calculables. Il sert aussi comme modèle de base dans l'implémentation des langages de programmation fonctionnelle. Les théoriciens et les mathématiciens constructivistes l'utilisent pour exprimer la logique d'ordre supérieur. Cependant, pour exprimer cette dernière, un groupe de chercheurs s'est intéressé au λ-calcul typé, alors que l'école Curry s'est intéressé au λ-calcul pur et à la logique combinatoire illative (ICL). En effet, Curry a découvert que le λ-calcul étendu sans restriction dans les règles d'introduction et d'élimination de l'implication est inconsistant (paradoxe de Curry)[Hin08][Mez02].

Le système Eλ introduit par M.Mezghiche et C.Ben-yelles[Mez02] est une extension du λ-calcul pur pouvant interpréter la logique d'ordre supérieur. La consistance du système est garantie grâce à l'introduction de la notion de niveau d'un Eλ-terme. En effet, le terme [N/x] M est défini seulement si *niveau(N)* $\leq$ *niveau(x)*. Cette restriction entraine la définition d'une nouvelle relation de réduction dans $E\lambda$ qui a la propriété de Church-Rosser.

Le but de notre projet est de spécifier le système Eλ dans l'assistant de preuve Coq, de représenter les Eλ-termes avec la notation de De Bruijn[Deb72] et de formaliser la preuve de la confluence de la relation de réduction E$\lambda\beta$-réduction.

Dans ce rapport, nous présenterons en premier lieu le système Coq et la théorie du λ-calcul. Nous présenterons également les preuves existantes du théorème de Curch-Rosser, qui est un résultat très important dans la théorie du λ-calcul. Ceci est suivi par une brève définition du système Eλ, sa syntaxe et ses règles de réécriture. Par la suite, nous décrivons notre démarche pour spécifier et démontrer la confluence de la E$\lambda\beta$-réduction et nous clôturons notre travail par une conclusion générale.

Chapitre 1

L'assistant de preuve Coq

Dans ce chapitre nous allons présenter l'assistant de preuve Coq, la démarche de son utilisation avec quelques exemples.

1.1 Introduction

Pourquoi valider les preuves par des ordinateurs ? La réponse à cette question est que la preuve écrite sur papier n'est pas toujours détaillée, et même si on détaille tout, la preuve devient illisible et peut contenir des erreurs. L'ordinateur par contre apporte une solution à ces problèmes.

La preuve élaborée avec un assistant de preuve est formelle. L'assistant vérifie tous les cas avec un niveau de rigueur très élevé. Alors que les preuves manuelles sont construites grâce à l'intuition humaine. Les assistants de preuve sont basés sur des outils permettant d'effectuer des preuves formelles et rigoureuses. Ces outils sont construits sur un noyau de très petite taille facile à vérifier à la main. Certes la preuve devient plus longue mais l'effort est justifié car cette preuve sera certifiée.

Le système Coq est conçu pour rédiger des preuves formelles mais aussi pour écrire des spécifications formelles et vérifier la correction des programmes par rapport a leurs spécifications. Il est basé sur la logique d'ordre supérieur, ce qui permet le développement de programmes informatiques avec leurs spécifications formelles. Par ailleurs, Coq utilise un mécanisme d'extraction pour extraire les programme validés vers d'autres langages de programmation comme OCaml ou Haskell.

Pour toutes ces raisons, Coq est de plus en plus utilisé par les chercheurs pour valider des raisonnements et des théories. On le trouve également dans l'industrie de

développement des systèmes informatiques où la tolérance aux erreurs est quasi nulle. En effet, de nos jours on constante que dans tous ces domaines on a besoin de programmes fiables, ce qui justifie le coût et l'effort investis dans la conception d'un tel système de démonstration formelle.

Le projet Coq est le fruit de plus de 30 ans de recherches, lancé en 1984 sur une implémentation du calcul de construction par Thierry Coquand et Gérard Huet. En 1991, Christine Paulin l'a étendu au calcul de constructions inductives. Depuis, plusieurs autres chercheurs à l'INRIA et a l'Université Paris Diderot ont également apporté une contribution au développement du système. De nos jours il est maintenu par l'équipe πr^2 au sein de l'INRIA en partenariat avec l'Université Paris Diderot, l'école polytechnique, l'Université Paris-Sud, l'École normale supérieure de Lyon.

Coq est un projet open-source écrit en OCaml et en C [Cdt16][Pal15][Ber15]. Dans notre projet nous allons utiliser la version 8.5pl1.

1.2 Les succès de Coq

Parmi les succès les plus connus de Coq nous avons[Ber15] :

- La preuve du théorème de Feit et Thompson relevant du domaine de l'algèbre.
- L'élaboration de CompCert C un compilateur optimisant le C (langage) qui est entièrement programmé et prouvé en Coq.
- GeoProof qui permet la formalisation et l'automatisation du raisonnement géométrique au sein de l'assistant de preuve Coq.

Dans le tableau suivant nous présentons une étude comparative entre le système Coq, et d'autres systèmes de preuve[Ber15].

Name	Latest version	Developer(s)	Implementation language	Features					
				Higher-order logic	Dependent types	Small kernel	Proof automation	Proof by reflection	Code generation
ACL2	5.0	Matt Kaufmann and J Strother Moore	Common Lisp	No	Untyped	No	Yes	Yes[1]	Already executable
Agda	2.4.2.1	Ulf Norell (Chalmers)	Haskell	Yes	Yes	Yes	No	Partial	Already executable
Coq	8.4	INRIA	OCaml	Yes	Yes	Yes	Yes	Yes	Yes
HOL Light	repository	John Harrison	OCaml	Yes	No	Yes	Yes	No	No
HOL4	Kananaskis-8 (or repo)	Michael Norrish, Konrad Slind, and others	Standard ML	Yes	No	Yes	Yes	No	Yes
Isabelle	2013 (or repo)	Larry Paulson (Cambridge), Tobias Nipkow (München) and Makarius Wenzel (Paris-Sud)	Standard ML	Yes	No	Yes	Yes	Yes	Yes
LEGO	1.3.1	Randy Pollack (Edinburgh)	Standard ML	Yes	Yes	Yes	No	No	No
Mizar	8.1.02	Białystok University	Free Pascal	Partial	Yes	No	No	No	No
NuPRL	5	Cornell University	Common Lisp	Yes	Yes	Yes	Yes	Unknown	Yes
PVS	5.0	SRI International	Common Lisp	Yes	Yes	No	Yes	No	Unknown
Twelf	1.7.1	Frank Pfenning and Carsten Schürmann	Standard ML	Yes	Yes	Unknown	No	No	Unknown

FIGURE 1.1 – Tableau comparatif entre Coq et les autres assistants de preuve

1.3 Fonctionnement du système Coq

Étant basé sur le calcul de constructions inductives et la théorie des types. Coq utilise l'isomorphisme de Curry-Howard pour son mécanisme de validation. Il fait correspondre à une propriété (ou théorème) un type. La preuve de cette propriété est un habitant du type (un terme ayant comme type le type de la propriété) ; c'est la vision intuitionniste. Les tactiques fournies avec Coq permettent la construction du terme et donc de la preuve. En plus, l'utilisateur a la possibilité de construire de nouvelle tactiques. L'exemple suivant explique le fonctionnement de principe de Curry-Howard[Ber15][Cdt16][Ber15].

Exemple . Démontrons la relation de transitivité de l'implication.

`Theorem` *Trans_Imp* :$\forall\ P\ Q\ R$: `Prop`, $(P \to Q) \to (Q \to R) \to P \to R$.

dans Coq la preuve se déroulera comme suit :

`Proof`.
`intros` *P Q R H1 H2 H3*.
`apply` *H2*.
`apply` *H1*.
`assumption`.

```
Qed.
```

On peut imprimer le terme qui prouve le théorème ci-dessus avec la commande *Print*

`Print` *Trans_Imp*.

```
Trans_Imp =
fun (P Q R : Prop) (H1 : P -> Q) (H2 : Q -> R) (H3 : P) => H2 (H1 H3)
     : forall P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R
```

On voit clairement que la preuve est une fonction à 3 arguments qui sont H1 H2 et H3. Cette fonction est une composition de fonctions *(H2 (H1 H3))* qui a comme type :

$$\forall\, P\ Q\ R : \texttt{Prop}, (P \to Q) \to (Q \to R) \to P \to R$$

1.4 Les composants essentiels de Coq

Le système Coq est composé de deux outils essentiels[Cha03] :

- Le langage de spécification (Gallina).
- L'outil de démonstration (les tactiques).

1.5 Le langage de spécification Gallina

Gallina est un langage fonctionnel basé sur le calcul de constructions inductives. C'est une variante du λ-calcul typé. Gallina permet la spécification des programmes informatique, mais aussi le développement des théories mathématiques. Il est muni d'un mécanisme de typage très strict. Dans Gallina, tout objet manipulé a un type et même les types ont un type. La notion de typage est divisée en sortes. La sorte *Prop* est réservée pour le typage des propositions logique, la sorte *Set* pour le typage des spécifications, des ensembles et des structures de données. Les sortes peuvent être manipulées comme des termes. Par conséquent elles ont un type. Autrement dit, si on suppose que *Set* a comme type *Set*, cela conduit vers une inconsistance (Paradoxe de Girard). Les sortes *Set* et *Prop* ont comme type *Type*, mais quel type donner pour la sorte *Type*?. Les concepteurs du système Coq ont introduit une hiérarchie dans la sorte *Type*, C'est-à-dire pour $i \neq j$, *Type(i)* a comme type *Type (j)*[Cdt16].

1.5.1 Navigation dans Gallina

Tous les services du système Coq sont accessibles grâce à un ensemble de commandes appelé *vernacular* (langage de commandes). Tous les mots clé de *vernacular* commencent par une lettre majuscule[Cdt16].

Exemple . *Definition , Eval , Theorem , Print , Fixpoint , Check ....*

1.5.2 Les Commandes essentiels

Dans cette partie nous allons décrire brièvement quelques composants essentiels de Coq :

a- Les Définitions

Les définitions étendent l'environnement, en donnant des noms à des termes
Syntaxe :

Definition ident := `<term>`.

Exemple . `Definition` *deux* := 0+2 .

b- Les Définitions de types inductifs

Les types inductifs permettent la formalisation des définitions par récurrence pour décrire des ensembles de termes, des structures de données ou des relations définies par induction. Chaque clause d'une définition inductive est appelée constructeur. Le constructeur a un nom et un type. Chaque définition retenue par le système se voit attribuée un principe d'induction qui sera utilisé lors du raisonnement par induction sur un terme du type défini[Cdt16][Ber15][Pal15].

Exemple .

- **Les entiers naturels** : les entiers naturels de Peano sont définis avec une structure inductive comprenant le 0 et le successeur d'un entier. La définition dans coq est la suivante :

  `Inductive` *nat* : `Set` :=
  $|O : nat$
  $|S : nat \rightarrow nat$.

  Le principe d'induction généré :
  Check `nat_ind`.

```
nat_ind
  : forall P : nat -> Prop,
  P O -> (forall n : nat, P n->P(S n)) ->forall n : nat,P n
```

– **Les booléens** : Le type booléen est le plus petit type avec deux constructeurs , le $true$ et le $false$:
Inductive *bool* : Set := *true* | *false*.

– **La fermeture transitive réflexive** : Cet exemple décrit la définition inductive de la fermeture réflexive et transitive d'une relation binaire R sur un ensemble quelconque A[Pie16] :
Inductive *fermR* {*A* : Type} (*R* : *relation A*) : *relation A* :=
|*base* : ∀ *x y*, *R x y* → *fermR R x y*
|*ref* : ∀ *x*, *fermR R x x*
|*trans* : ∀ *x y z*, *fermR R x y* → *fermR R y z* → *fermR R x z*.

c- Les fonctions récursives

Les fonctions récursives sont définies sur les structures de données inductives avec la syntaxe suivante :

```
Fixpoint ident params struct ident0  : type0 := term0
```

Comme la plupart des langages de programmation fonctionnelle, Coq permet la définition des fonctions récursives, en y ajoutant le pouvoir d'exprimer des types avec des fonctions récursives (types dépendants), avec la seule contrainte que la fonction doit se terminer. En effet, Coq essaie de trouver une récursion structurelle sur un argument dans la définition de la fonction (si elle n'est pas vulgarisée avec le mot clé struct), puis il cherche si les valeurs de cet argument sont décrémentées dans l'appel récursif. Si oui, la définition sera correcte (si le typage est bon) et un message confirme la décrémentation sur l'argument, sinon l'opération échoue. Ce principe est à la fois bon et pas bon car il renforce la sureté de la programmation en Coq, mais il affaibli le pouvoir d'expressivité. Les fonctions récursives ne pouvant pas être traduites en Coq, peuvent toutefois être spécifiées par des prédicats inductifs à la Prolog.

Exemple : La fonction plus :

Fixpoint *add* (*n m* :*nat*) : *nat* :=
match *n* with

| $O \Rightarrow m$

| $S\ p \Rightarrow S$ (*plus p m*)

`end`.

d- Les propriétés

Une propriété est une proposition logique de la sorte $Prop$. Elle est utilisée pour valider la spécification. Une propriété commence avec l'une des commandes suivantes : *Theorem*, *Lemma*, *fact* , *Remark* , *Examlpe*, sa syntaxe est la suivante :

Theorem nom : `<enoncé>`.

Une fois la propriété écrite, elle passe par le *type checker* de coq, qui va générer son type.

e- D'autres commandes

D'autres commandes sont essentielles pour l'utilisation du système Coq :

- la commande *Check* : pour vérifier le type d'un terme , par exemple : *Check* 2. 2 : nat
- la commande *Eval* : pour calculer une expression, exemple : *Eval* compute in (3+6) 9 : nat.
- la commande *Print* : pour afficher la définition d'un terme.

1.6 La démonstration des propriétés

Lorsque l'utilisateur veut valider sa spécification, il doit d'abord rédiger une ou plusieurs propriétés sur celle-ci. Ensuite, il doit démontrer ces propriétés une par une. Il commence par écrire la propriété qui peut être vue comme un théorème ou lemme. À la fin de cette étape, le système Coq entre dans le mode de preuve. Dans ce mode, l'utilisateur utilise un langage spécial, qui repose sur des tactiques. Une tactique prend en entrée un sous-but à prouver et selon le cas, le résout (l'élimine), le modifie, génère d'autres sous-buts, ou bien échoue en générant une erreur.

Pour prouver une propriété, il existe dans Coq un autre environnement qui est activé dès que l'écriture d'une propriété est terminée. Dans cet environnement, la section de la preuve commence par le mot clé `Proof` et se termine par le mot clé `Qed` ou `Save`. Une fois la preuve terminée, Coq affiche le message **No more subgoals**. Après

l'écriture de la commande `Qed`, le système Coq revérifie la totalité de la preuve et sauvegarde le terme construit suite à cette dernière.

Parmi les tactiques les plus utilisées, nous avons :`intros` , `simpl` , `reflexivity` , `apply` , `destruct` , `induction` , `rewrite`. Le système Coq est fourni avec un grand nombre de tactiques, nous présentons ci-dessous quelques unes [Cdt16] :

• `intro` : la tactique $intro$ permet d'introduire des hypothèses dans le contexte du but courant et transformer un but quantifié universellement en un but sans quantification universelle.

• `apply term` : cette tactique essaie de faire correspondre le but courant avec la conclusion du type de `term` (un terme dans le contexte local). Si elle réussit, elle retourne autant de sous-buts qu'il y a de prémisses dans `term`.

• `split` : la tactique qui permet de décomposer un but dont l'énoncé est une conjonction est la tactique `split`. Elle permet de passer d'un but de la forme $A \wedge B$ à deux buts plus simples de la forme A et B.

• `left` : lorsqu'on cherche à prouver une expression de la forme $A \vee B$, il suffit parfois de ne prouver que A. C'est cette étape de raisonnement qui est fournie par la tactique left. Cette tactique permet de remplacer un but de la forme $A \vee B$ par un seul but de la forme A.

• `right` : le comportement de cette tactique est exactement symétrique à celui de la tactique `left`. Elle permet de transformer un but de la forme $A \vee B$ en un seul but de la forme B.

• `assumption` : l'activation de la tactique $assumption$ contrôle la convertibilité entre l'hypothèse et l'énoncé à prouver.

• `genaralize` : elle permet d'ajouter au but courant une quantification universelle. Si le but est de la forme $(p\ x)$ et que x est un objet défini de type T, alors, $generalize\ x$ transforme le but en : `forall x : t, (P x).`

• `exists` : lorsque le but est de la forme `exists x : T, A x`, la tactique $exists$ M (où M est un terme de type T) remplace le but courant par un but de la forme $A\ M$.

• `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 $M = M$.

• `induction term` : elle permet de faire une preuve par induction. L'argument

$term$ doit être une constante inductive. Cette tactique génère un sous-but pour chaque constructeur du type inductif concerné et remplace chaque occurrence de $term$ dans la conclusion et les hypothèses du but en rajoutant au contexte local des hypothèses d'induction.

• `auto` : le principe de cette tactique est simple, auto utilise une base de données de tactiques(*Hints*), lesquelles sont appliquées au but initial, ainsi qu'à tous les sous buts engendrés, et ce répétitivement, jusqu'à la résolution de tous les buts. Si ces buts ne peuvent être tous résolus, `auto` annule tous ses efforts et laisse le but initial inchangé.

• `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.

1.7 Utilisation du système Coq

Pour utiliser Coq il y a deux choix, soit en ligne de commande avec la commande `coqtop` , soit avec l'interface graphique. Pour utiliser Coq avec l'interface graphique, on peut opter pour celle fournie par les développeurs de Coq (CoqIDE), ou bien on utilise l'éditeur Emacs via le paquet ProofGeneral.

1.7.1 L'interface CoqIDE

Dans cette section, nous allons décrire l'interface de CoqIDE du système Coq. Elle est très modestement élaborée. L'utilisateur rédige son programme dans la partie gauche. Ensuite, il peut l'exécuter pas à pas interactivement à l'aide des flèches de la barre d'outils. Les résultats du top-level vont apparaitre en bas à droite. La partie droite haute est réservée pour le mode de preuve. Elle va contenir les buts à prouver, les hypothèses utilisables et des informations sur la preuve elle-même (par exemple le nombre de sous-buts).

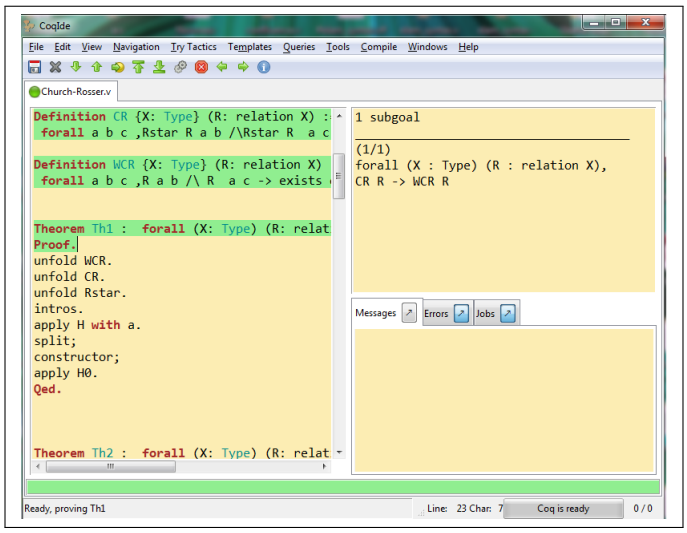

FIGURE 1.2 – Interface de CoqIDE

1.7.2 ProofGeneral

ProofGeneral est un plugin pour le célèbre éditeur de texte Emacs. il a été développé dans l'Université d'Edinburgh. Il offre un environnement graphique avec une interface simple, personnalisable et un coloriage syntaxique. En plus il optimise le développement en ajoutant la possibilité de créer des macros clavier.

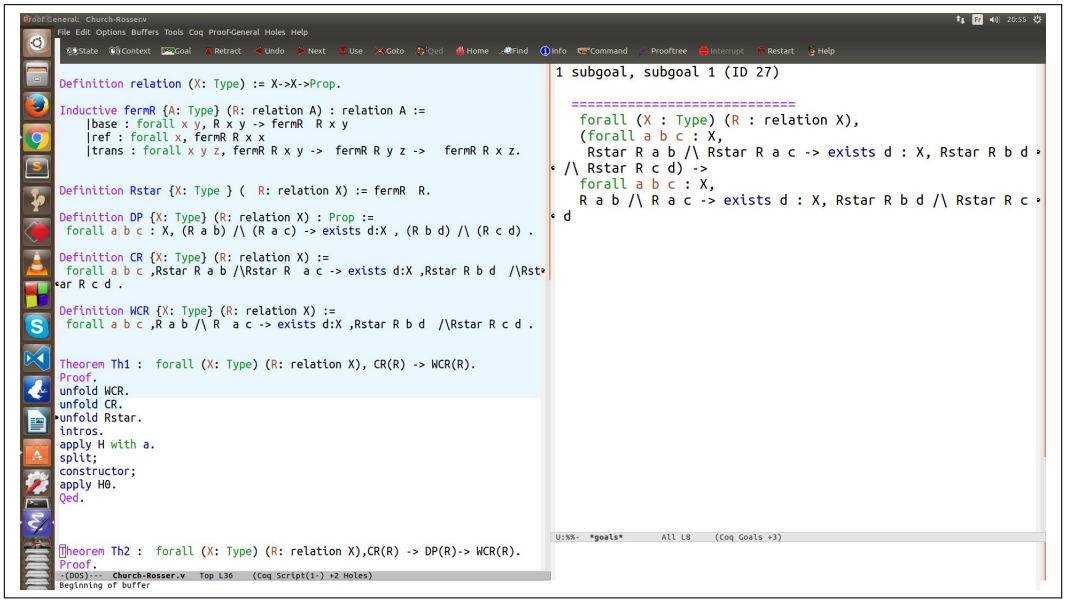

FIGURE 1.3 – Interface de ProofGeneral

1.8 Conclusion

Dans ce chapitre nous avons donné des généralités sur le système Coq qui est un assistant de preuves hybride combinant preuve et calculs. Nous avons présenté et illustré par des exemples son langage de spécification Gallina et son mécanisme de validation. Enfin nous avons décrit les interfaces CoqIDE et ProofGeneral.

Dans le chapitre suivant nous allons donner des généralités sur la théorie du λ-calcul.

Chapitre 2

λ-calcul

2.1 Introduction

Dans ce chapitre, nous présentons la théorie du λ-calcul, son alphabet, sa syntaxe et ses règles de calcul.

2.2 Le λ-calcul

Le λ-calcul est un petit langage de programmation. Sa syntaxe est basée sur des entités syntaxiques que l'on appelle des λ-termes (ou λ-expressions). Elles se divisent en trois catégories[Cha03] :

— Les variables : x, y, ... sont des λ-termes ;

— Les applications : $u\ v$ est un λ-terme si u et v sont des λ-termes ;

— Les abstractions : $\lambda x.v$ est un λ–terme si x est une variable et v un λ–terme.

2.3 Variables libres et variables liées

2.3.1 Les variables libres

Les occurrences libres d'une variable x dans un terme t sont définies par induction sur la longueur de t :

— Si t est une variable x, l'occurrence de x dans t est libre.

— Si $t = (uv)$, les occurrences libres de x dans t sont celles de x dans u et celles dans v.

— Si $t = \lambda y.u$, les occurrences libres de x dans t sont celles de x dans u, sauf si

$x = y$; dans ce cas, x n'a pas d'occurrence libre dans t.

2.3.2 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 :

— $BV(x) \equiv \varnothing$.

— $BV(MN) \equiv BV(M) \cup BV(N)$.

— $BV(\lambda x.M) \equiv BV(M) \cup \{x\}$.

BV : Bound Variable (variable liée)

2.4 La substitution

Dans le λ-calcul, un ensemble de règles de réécriture est défini pour décrire l'évaluation d'une λ-expression. Ces règles sont basées sur le concept de la substitution[Bar84][Cha03].

Remarque. $M[x := u]$ signifie que l'on remplace les occurrences libres de x dans M par u.

$$
\begin{array}{|l|}
\hline
\quad x[x := u] = u \\
\quad y[x := u] = y \quad si \quad x \neq y \\
(\lambda x.t')[x := u] = \lambda x.t' \\
(\lambda y.t')[x := u] = \lambda y.(t'[x := u]) \quad si \quad x \neq y \quad et \quad y \notin FV(u). \\
(\lambda y.t')[x := u] = \lambda z.(t'[y := z][x := u]) \; si \; x \neq y, \; y \in FV(u), \; z \notin FV(t') \; et \; z \notin FV(u) \\
\quad (t't'')[x := u] = (t'[x := u]t''[x := u]) \\
\hline
\end{array}
$$

2.5 Les réductions

2.5.1 α-conversion

La α-conversion permet de renommer un paramètre formel d'une λ-expression. Soit $a = (\lambda x.x)$ et $b = (\lambda y.y)$; ces termes sont α-équivalents

2.5.2 β-réduction

La seconde réduction, nommée β est la plus importante : c'est elle qui explique comment l'application d'une fonction à un argument peut se réduire en un résultat :

$$(\beta)\ (\lambda x.e)M \leftrightarrow [M/x]e$$

Un redexe est un terme qui admet une ou plusieurs β-réductions[Cha03].

2.5.3 Forme normale

Un λ-terme t est dit en forme normale si aucune β-réduction ne peut lui être appliquée ; c'est-à-dire si t ne contient aucun redexe. Dans le cas contraire, on dit que t est normalisable. Si de plus toutes les réductions à partir de t sont finies, alors on dit que t est fortement normalisable. Exemples : $(\lambda x.x)((\lambda y.y)z)$ est fortement normalisable. $(\lambda x.xxx)(\lambda x.xxx)$ ne fait que créer des termes de plus en plus grands [Bar84][Cha03].

2.6 D'autres représentations

Il existe d'autres méthodes pour la représentation des termes du λ-calcul. Parmi celles-ci on trouve la notation de de Bruijn[Deb72] et la représentation avec les combinateurs.

2.6.1 Les indices de DeBruijn

Dans la représentation par les indices de DeBruijn, l'idée consiste à remplacer toute variable liée x par un pointeur vers l'abstraction qui le lie. Ce pointeur est représenté par un entier qui sert à compter le nombre de symboles λ rencontrés lorsqu'on remonte l'arbre de syntaxe abstraite à partir de l'occurrence considérée de x vers le λ qui la lie[Deb72]. Les noms des variables liées sont inutiles, et peuvent être par conséquent effacés. Les variables libres peuvent être conservées comme elles sont. La syntaxe des indices de DeBruijn sera donc :

- $x, y, z, ...$: Représente les variables libres .
- $0, 1, 2, 3, ...$: Représente les indices de DeBruijn.
- λM : Abstraction
- MN : Application

Lors de la substitution, les indices doivent être mis à jour. La fonction de décalage est définie comme suit[Deb72] :

$$\begin{aligned}
\uparrow_d^c (i) = &\quad i \quad \textbf{si} \quad i < c \\
\uparrow_d^c (i) = &\quad i + d \quad \textbf{si} \ \ i \geq c \\
\uparrow_d^c (\lambda.t) = &\quad \lambda.\uparrow_d^{c+1} (t) \\
\uparrow_d^c (t_1\, t_2) = &\quad (\uparrow_d^c (t_1)) \ \ (\uparrow_d^c (t_2))
\end{aligned}$$

la substitution est définie comme suit[Deb72] :

$$\begin{aligned}
[t/j]\, i = &\quad t \quad \textbf{si} \quad i = j \\
[t/j]\, i = &\quad i \quad \textbf{si} \quad i \neq j \\
[t/j]\, (\lambda.t') = &\ \lambda.[\uparrow^1 (t)/j+1]t' \\
[t/j]\, (t_1 t_2) = &\ ([t/j]\, t_1) \ \ ([t/j]\, t_2)
\end{aligned}$$

la β-réduction est définie comme suit[Deb72] :

$$(\lambda.t_1)t_2 = \uparrow^{-1} ([\uparrow^1 (t_2)/0]t_1)$$

2.7 Conclusion

Dans ce chapitre, nous avons présenté des généralités sur la théorie du λ-calcul avec la notation classique et la notation de De Bruijn. le chapitre suivant sera consacré à l'illustration des méthodes existantes pour démontrer la confluence de la β-réduction.

Chapitre 3

La propriété de Church-Rosser (Confluence)

3.1 Introduction

La confluence est une propriété associée souvent aux systèmes de réécriture. Un système muni avec une telle propriété est considéré comme déterministe. C'est-à-dire deux chemins de calcul ou de réduction différents doivent toujours pouvoir se recroiser[Sau13][Ber06].

La confluence implique la consistance du système λ-calcul et l'unicité de la forme normale d'un λ-terme, si elle existe[Bar84].

3.2 Définitions

Afin d'expliquer la propriété de confluence dans le λ-calcul, quelques notions préliminaires doivent être introduites.

3.2.1 Relation binaire

Soit R une relation binaire entre les éléments d'un ensemble quelconque E. Sa fermeture réflexive transitive notée R^* est définie inductivement comme suit[Bar84][Sau13][Pol95] :

— $a, b \in E, a\ R\ b \vDash a\ R^*\ b$

— $a \in E, a\ R^*\ a$

— $a, b, c \in E\ , a\ R^*\ b \wedge b\ R^*\ c \vDash a\ R^*\ c$

3.3 Les formes de confluence

3.3.1 Confluence Globale (CR)

Une relation binaire R est confluente ou globalement confluente (en anglais *Church-Rosser*)[Bar84][Sau13] si :

$$\forall a, b, c \in E, aR^*b \wedge aR^*c \implies \exists d \in E, bR^*d \wedge cR^*d$$

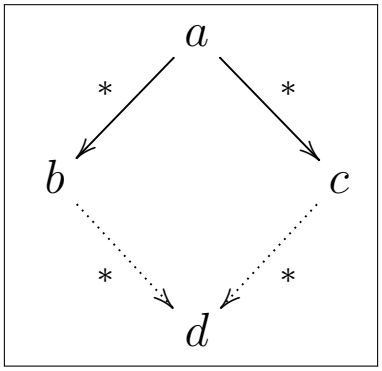

FIGURE 3.1 – Confluence globale

3.3.2 Confluence Locale (WCR)

Une relation binaire R est localement confluente (en anglais *Weakly Church-Rosser*)[Bar84][Sau13] si :

$$\forall a, b, c \in E, aRb \wedge aRc \implies \exists d \in E, bR^*d \wedge cR^*d$$

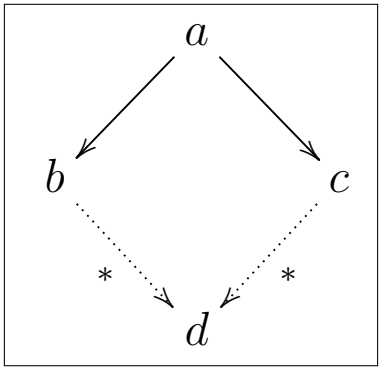

FIGURE 3.2 – Confluence locale

3.3.3 Confluence Forte (DP)

Une relation binaire R est fortement confluente ou possède la propriété du diamant (en anglais *Diamond Property*)[Bar84][Sau13] si :

$$\forall a, b, c \in E, aRb \wedge aRc \implies \exists d \in E, bRd \wedge cRd$$

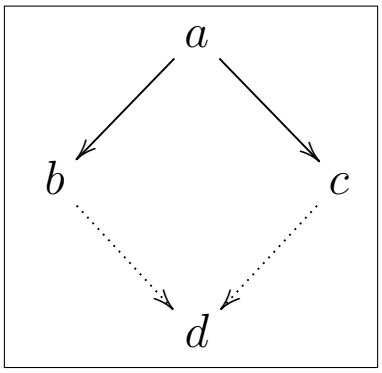

FIGURE 3.3 – Confluence forte

3.3.4 Relation entre les différentes formes de confluence

La relation entre les différentes formes de confluence se présente sous la forme de l'implication suivante :

$$\text{DP} \implies \text{CR} \implies \text{WCR}$$

3.4 Théorème de Church-Rosser

Le théorème de Church-Rosser est un théorème fondamental dans la théorie du λ-calcul. Il a été élaboré par Alonzo Church et Barkley Rosser en 1936. Il affirme la confluence de la relation de réduction β-réduction. Ce résultat entraine la consistance du λ-calcul et l'unicité de la forme normale d'un λ-terme si elle existe.

Nous allons noter la relation β-réduction du λ-calcul comme suit : $(\rightarrow_\beta)$.

L'énoncé du théorème de Church-Rosser selon Barendregt[Bar84] est le suivant :

La β-réduction est confluente.

Les conséquences du Théorème de Church-Rosser sont décrites par les corollaires ci-dessous.

Corollaire. Les formes normales sont uniques (si elles existent) i.e. (soit a, b, c, d des λ-termes, si $a \rightarrow_\beta^* b, a \rightarrow_\beta^* c, b$ et c sont à la forme normale , alors $b \equiv_\alpha c$ (α-équivalents))[Pol95].

Preuve. Avec la confluence forte de $(\rightarrow_\beta^*)$, b et c se réduisent à un terme d, mais puisque b et c sont déjà à la forme normale,alors $b \equiv_\alpha c \equiv_\alpha d$. □.

Corollaire. Le λ-calcul est consistant[Pol95].

Preuve. Par contradiction, on suppose que λ-calcul est inconsistant. Alors, on a par exemple : $X \rightarrow_\beta^* 1$ et $X \rightarrow_\beta^* 0$, donc avec DP($\rightarrow_\beta^*$) on a $\exists\, d, 0 \rightarrow_\beta^* d$ et $1 \rightarrow_\beta^* d$ mais 0 et 1 sont déjà à la forme normale, par conséquent $0 = 1 = d$, mais $0 \neq 1$ (contradiction !) d'où le λ-calcul est consistant. □.

Ces deux résultats sont très importants , d'où la nécessité de la démonstration d'un tel théorème.

3.5 Preuves existantes du théorème de Church-Rosser

Pour prouver la confluence du λ-calcul, il existe plusieurs méthodes. Parmi celles-ci, la preuve originale due à Church et Rosser est reportée comme étant longue et difficile à comprendre [Sha88]. De même, en 1971 , Per Martin-Löf a publié sa célèbre preuve de ce théorème[Bar84][Hin08] basée sur les travaux de William Tait 1965. Cette preuve utilise la notion des réductions parallèles ou *minimal complet developpement* selon Hindley[Hin08]. Elle compte plusieurs améliorations, dont la plus importante revient à Masako Takahashi[Tak95] qui introduit la notion de développement complet. Dans ce qui suit nous allons présenter la méthode de preuve due à Tait et Martin-Löf avec les réductions parallèles, ainsi que son amélioration avec les développements complets.

3.6 La preuve de Tait et Martin-Löf

Dans cette partie nous allons décrire la démarche suivie pour démontrer la confluence en utilisant la méthode des réductions parallèles. Nous suivrons la démarche décrite dans le livre de Barendregt[Bar84] et le cour de [Sau13].

D'abord, on rappelle la définition inductive de $(\rightarrow_\beta)$:

- $(\lambda x.M)N \rightarrow_\beta [N/x]M$
- $M \rightarrow_\beta N \implies \lambda x.M \rightarrow_\beta \lambda x.N$

- $M \to_\beta M' \implies MN \to_\beta M'N$

- $N \to_\beta N' \implies MN \to_\beta MN'$

Ça aurait été facile de prouver la confluence du λ-calcul si $(\to_\beta)$ vérifie la confluence forte, mais malheureusement ce n'est pas le cas car $(\to_\beta)$ n'est pas réflexive et elle ne réduit pas en parallèle :

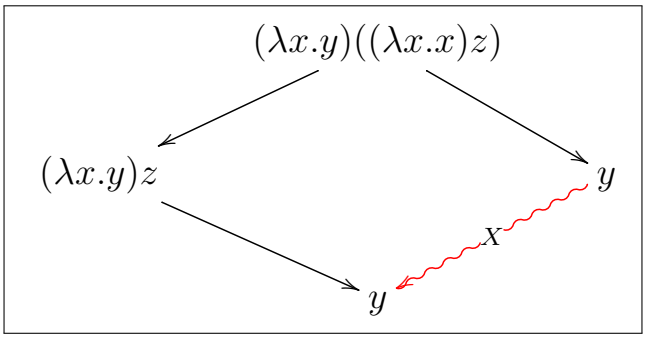

FIGURE 3.4 – β-réduction non réflexive

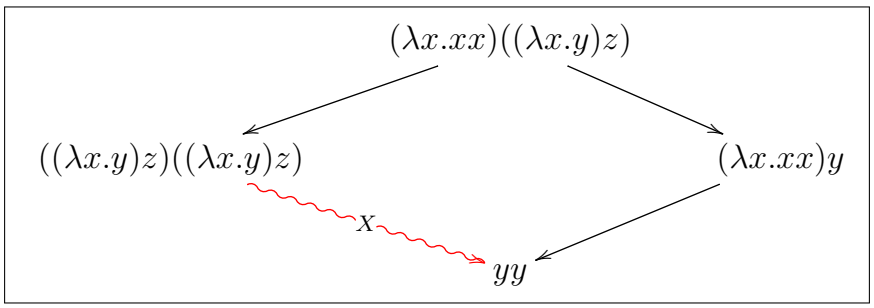

FIGURE 3.5 – β-réduction ne réduit pas en parallèle

La fermeture réflexive et transitive $(\to_\beta^*)$ de $(\to_\beta)$ vérifie la propriété du diamant (confluence forte), mais elle est très compliquée pour être analysée vu que c'est la fermeture réflexive et transitive d'une relation définie inductivement[Sau13].

Donc l'idée consiste à créer une nouvelle relation qui est réflexive et peut réduire en parallèle. Cette relation se trouve entre $(\to_\beta)$ et $(\to_\beta^*)$, vérifie la propriété du diamant et sa fermeture transitive est équivalente à $(\to_\beta^*)$. Elle est nommée réduction parallèle.

3.6.1 Réduction parallèle

Notation. Nous allons noter la réduction parallèle avec le symbole suivant :$\not\Rightarrow$[Pol95]

la relation de réduction parallèle est définie inductivement comme suit[Bar84] :

- $M \not\Rightarrow M$
- $M \not\Rightarrow M' \implies \lambda x.M \not\Rightarrow \lambda x.M'$
- $M \not\Rightarrow M' \wedge N \not\Rightarrow N' \implies MN \not\Rightarrow M'N'$
- $M \not\Rightarrow M' \wedge N \not\Rightarrow N' \implies (\lambda x.M)N \not\Rightarrow [N'/x]M'$

Sa fermeture transitive est notée : $(\not\Rightarrow^*)$.

3.6.2 La démarche de preuve

Le travail consiste à démontrer la confluence forte de $(\rightarrow_\beta^*)$ pour cela on commence par démontrer la confluence forte de $(\not\Rightarrow)$, ensuite, on démontrera que si $(\not\Rightarrow)$ est fortement confluente alors $(\not\Rightarrow^*)$ est aussi fortement confluente, et finalement on démontre que $(\not\Rightarrow^*)$ est équivalente à $(\rightarrow_\beta^*)$. Le théorème de Church-Rosser sera démontré par une simple application de la règle Modus-Ponens[Bar84][Sau13].

La preuve a donc trois parties qui sont :

— Si $(\not\Rightarrow)$ est fortement confluente alors sa fermeture transitive $(\not\Rightarrow^*)$ est fortement confluente (DP($\not\Rightarrow$) $\implies$ DP($\not\Rightarrow^*$)). (lemme.1 *strip lemma*)

— $(\not\Rightarrow)$ est fortement confluente (DP($\not\Rightarrow$)). (lemme.2)

— La fermeture transitive de la réduction parallèle est équivalente à la fermeture réflexive et transitive de la β-réduction ($\not\Rightarrow^* \iff \rightarrow_\beta^*$). (lemme.3)

3.7 Amélioration avec les développements complets

D'abord nous devons introduire la notion de résidu.

Définition. Un résidu est un redexe généré après l'application d'une β-réduction, exemple :

$(\lambda x.xx)((\lambda x.y)z) \rightarrow_\beta ((\lambda x.y)z)((\lambda x.y)z)$ les deux termes $((\lambda x.y)z)$ sont appelés résidus de $(\lambda x.xx)((\lambda x.y)z)$ après l'application de la β-réduction.

La confluence forte de la relation réduction parallèle (DP($\not\Rightarrow$)) peut être facilement prouvée en utilisant la notion des développements complets due à Takahashi (1995)[Tak95] et [Pol95]. La méthode consiste à ne pas analyser les différences entre les réductions, mais de chercher l'étape de réduction parallèle "maximum" qui contracte tous les redexes.

Par exemple : pour $a \not\Rightarrow b$, l'utilisation d'une réduction parallèle maximum va nous permettre de contracter tous les résidus dans b des redexes de a (nouveaux re-

dexes générés avec ($\#\!\!\!\rightarrow$)). Ce qui va nous permettre de toujours dépasser la réduction parallèle et ainsi de fermer les triangles gauche et droit indépendamment comme le montre la figure ci-dessous.

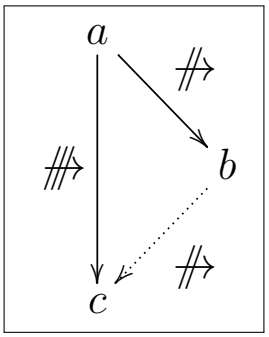

FIGURE 3.6 – Réduction parallèle maximum

Remarque. La différence entre la réduction parallèle et le développement complet réside seulement dans la règle de l'application. Autrement dit, dans l'application MN, M ne peut pas être une abstraction avec le développement complet.

Exemple. Avec la réduction parallèle , le terme $(\lambda x.x)N$ peut-être réduit de 3 façons différentes (réflexivité, application et β-réduction) mais seulement l'une d'elles le contracte (β-réduction). Avec le développement complet il peut-être réduit d'une seule façon et celle-ci le contracte (β-réduction).

Notation. Nous allons noter la relation développement complet avec le symbole suivant : ($\#\!\!\!\rightarrow$).

Cette relation est définie inductivement comme suit :

- $x \#\!\!\!\rightarrow x$
- $M \#\!\!\!\rightarrow M' \implies \lambda x.M \#\!\!\!\rightarrow \lambda x.M'$
- $M \#\!\!\!\rightarrow M' \wedge N \#\!\!\!\rightarrow N' \implies MN \#\!\!\!\rightarrow M'N'$ (M n'est pas une abstraction [1])
- $M \#\!\!\!\rightarrow M' \wedge N \#\!\!\!\rightarrow N' \implies (\lambda x.M)N \#\!\!\!\rightarrow [N'/x]M'$

Sa fermeture transitive est notée : ($\#\!\!\!\rightarrow^*$).

Pour prouver la confluence forte de ($\#\!\!\!\rightarrow$) on devra démontrer deux lemmes intermédiaires

1. Dans le cas du Eλ-calcul, M peut être une abstraction mais il faudra imposer un autre type de contraintes.

Lemme 4 : existence du développement complet . $\forall a \exists d, \quad a \nRightarrow d$

Preuve. Avec une induction simple sur la structure de a [Pol95]. □.

Lemme 5 : $\forall a, b, d \quad a \nRightarrow d \wedge a \not\Rightarrow b \implies b \not\Rightarrow d$

Preuve. Avec une induction structurelle sur la relation ($a \nRightarrow d$) [Pol95]. □.

Enfin, pour prouver la confluence forte de la réduction parallèle (DP($\not\Rightarrow$)), il suffit d'appliquer le lemme 2 deux fois (sur la gauche et sur la droite) [Pol95].

3.8 Conclusion

Dans ce chapitre nous avons présenté deux démarches existantes pour démontrer la confluence de la β-réduction, on note que les démarches ci-dessus sont valables pour la notation de De bruijn. Dans le chapitre suivant nous allons définir le système Eλ, sa syntaxe et ses règles de réécriture.

Chapitre 4

Le système Eλ

4.1 Introduction

Le système Eλ est une extension du λ-calcul pur pouvant interpréter la logique d'ordre supérieur. Deux constantes " P et Π " sont introduites représentant respectivement l'implication et la quantification universelle. Afin d'éviter le paradoxe de Curry et assurer le maintien de la consistance du système, la notion de niveau d'un Eλ-terme a été introduite pour ajouter une contrainte à l'opération de substitution. Le système obtenu devient assez riche grâce aux deux constantes introduites qui permettent de définir et d'exprimer le reste des connecteurs logiques. Il est par ailleurs important de noter que le système Eλ peut interpréter la logique d'ordre supérieur. Ce qui en fait une bonne théorie de base pour l'implémentation d'un assistant de preuve[Mez02].

4.2 L'ensemble des Eλ-termes

L'ensemble des termes du système $E\lambda$ noté C_w est l'union des ensembles C_0, C_1, C_2.... Chaque ensemble $C_i (i \geq 1)$ est construit en ajoutant successivement 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 λ-termes classiques où chaque terme possède les variables de la forme x^0, y^0, z^0.... L'ensemble C_0 ne contient pas les constantes P et Π[Mez02].

La définition de l'ensemble C_0 : L'ensemble C_0 est construit à partir d'un ensemble infini de variables $V_0 = \{x^0, y^0, z^0...\}$ en utilisant l'application et la λ-abstraction[Mez02].

— Si $x \in V_0$ alors $x \in C_0$

— Si $M, N \in C_0$ alors $(MN) \in C_0$

— Si $M \in C_0, x \in V_0$ alors $(\lambda x.M) \in C_0$.

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

L'ensemble des termes C_i est construit inductivement à partir d'un ensemble infini de variables $V_i = V_{i-1} \cup \{x^i, y^i, z^i, ...\}$ et des deux constantes P et Π[Mez02] :

— Si $x \in V_i$ alors $x \in C_i$

— Si $M \in C_i, N \in C_j$ alors $(MN) \in C_{max(i,j)}$

— 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 $PXY \in C_{max(i,j)+1}$.

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

Définition L'ensemble des $E\lambda$-termes C_w est défini par $C_w = \cup_{i<w} C_i$.

Dans notre cas, nous avons défini en Coq, l'ensemble des Eλ-termes de la façon suivante :

Formalisation.

`Inductive` *elambda* : `Set` :=

| *Var* : *nat* × *nat* → *elambda*

| *Abs* : *elambda* → *nat* → *elambda*

| *App* : *elambda* → *elambda* → *elambda*

| *Imp* : *elambda* → *elambda* → *elambda*

| *Pi* : *elambda* → *elambda*.

Maintenant, nous allons définir le niveau d'un $E\lambda$-terme qui introduit une contrainte dans l'opération de la substitution en permettant à chaque variable du niveau k d'être substituée par un terme du niveau inférieur ou égal à k. Cette notion joue un rôle important pour éviter le paradoxe de Curry et servir par conséquent, dans la preuve de la consistance du système $E\lambda$[Mez02].

Définition (niveau d'un Eλ-terme) Le niveau d'un $E\lambda$-terme est calculé en suivant les règles ci dessous [Mez02] :

– *niveau*(x^i) = i

– *niveau*$(\lambda x.M)$ = *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)).*

Formalisation. Nous avons défini en Coq, la fonction qui calcule le niveau des Eλ-termes de la façon suivante :

```
Fixpoint level (M : elambda) : nat :=
 match M with
 | Var x ⇒ snd x
 | Abs M' l ⇒ if (le_lt_dec (level M') l) then l else (level M')
 | App A B ⇒ if (le_lt_dec (level A) (level B)) then (level B) else (level A)
 | Imp A B ⇒ S(if (le_lt_dec (level A) (level B)) then (level B) else (level A))
 | Pi M' ⇒ if (le_lt_dec (S O) (level M')) then (level M') else 1
end.
```

Notation : Pour rendre le texte plus lisible, nous écrivons "$X \supset Y$" au lieu de "P X Y". Ainsi, nous allons écrire X^k pour dire que X est un Eλ-terme et k son niveau[Mez02].

Définition Le système $E\lambda$ est défini comme suit :

1. L'ensemble des $E\lambda$-termes est C_w
2. La notion de la $E\lambda\beta$-réduction est défini en se basant sur les règles suivantes [Mez02] :

   $(\rho)\, M \to_{E\lambda\beta} M$

   $(\mu)\, M \to_{E\lambda\beta} N \implies ZM \to_{E\lambda\beta} ZN$

   $(\mu')\, M \to_{E\lambda\beta} N \implies X \supset M \to_{E\lambda\beta} X \supset N$

   $(\mu'')\, M \to_{E\lambda\beta} N \implies \Pi M \to_{E\lambda\beta} \Pi N$

   $(\nu)\, M \to_{E\lambda\beta} N \implies MZ \to_{E\lambda\beta} NZ$

   $(\nu')\, M \to_{E\lambda\beta} N \implies M \supset X \to_{E\lambda\beta} N \supset X$

   $(\tau)\, M \to_{E\lambda\beta} N, N \to_{E\lambda\beta} T$ alors $M \to_{E\lambda\beta} T$

   $(E\alpha)\, \lambda x.M \to_{E\lambda\beta} \lambda y.[y/x]M$ si niveau(x) = niveau(y) et $y \notin FV(M)$

   $(E\beta)\, (\lambda x.M)N \to_{E\lambda\beta} [N/x]M$ si niveau(N) $\leq$ niveau(x) ou niveau($\lambda x.M$)=0.

   $(E\xi)\, M \to_{E\lambda\beta} N \implies \lambda x.M \to_{E\lambda\beta} \lambda x.N$

4.3 Conclusion

Dans ce chapitre nous avons présenté le système $E\lambda$ qui est un système consistant pouvant interpréter la logique d'ordre supérieur. La consistance du système est garantie grâce à la notion du niveau qui permet de faire une restriction sur la règle de substitution pour éviter le paradoxe de Curry. Ce système peut être utilisé pour implémenter les assistants de preuves. Et comme $\rightarrow_{E\lambda\beta}$ est une relation de réécriture, montrer qu'elle a la propriété de Church-Rosser s'avère important car elle assure la consistance et le déterminisme du système. Le chapitre suivant sera consacré à la preuve de la propriété de Church-Rosser pour la E$\lambda\beta$-réduction.

Chapitre 5

Confluence de la E$\lambda\beta$-réduction et sa formalisation

5.1 Introduction

Dans les chapitres 3 et 4, nous avons décrit la propriété de Church-Rosser et le system Eλ. Nous avons aussi expliqué l'importance d'avoir cette propriété pour un système de réécriture tel que la E$\lambda\beta$-réduction.

Dans cette partie nous détaillons la démarche suivie pour démontrer la propriété de Church-Rosser pour la E$\lambda\beta$-réduction, et nous validons les spécifications formelles (code Coq) utilisées.

5.2 Quelques propriétés

Afin de démontrer la propriété de Church-Rosser de la $E\lambda\beta$-réduction, quelques propositions sont nécessaires et doivent être démontrées avant d'entamer la preuve[Mez02] :

1. **Proposition .** Si *niveau(M)* $\leq$ *niveau(x)* alors *niveau([M/x]P)* $\leq$ *niveau(P)*
   Preuve. induction sur P, tous les cas sont simples à prouver sauf si P $\equiv \lambda y^i.Q^j$ alors :
   - *niveau(P) = niveau*($\lambda y^i.Q^j$) = *max*(i, j)
   - *niveau([M/x]P) = niveau*([M/x]($\lambda y^i.Q^j$))
   - Si $x = y$ alors *niveau([M/x]P) = niveau([M/*x*]Q) = max*(i,j) et donc on a *max*(i, j) $\leq$ *max*(i, j).

- Si $x \neq y$ alors *niveau([M/x]P) = niveau(λy^i.[M/x]Q^j) = max(i, niveau([M/x]Q^i)* et on a *niveau([M/x]Q^i)* $\leq$ *niveau(x)* par l'hypothèse d'induction donc *niveau([M/x]P)* $\leq$ *niveau(x)* □.

2. **Proposition.** Si P $\rightarrow_{E\lambda\beta}$ Q alors *niveau(P)* $\geq$ *niveau(Q)*
   Preuve. par une simple induction sur la réduction $P \rightarrow_{E\lambda\beta} Q$ □.

3. **Proposition .** Si $P \rightarrow_{E\lambda\beta} Q$ alors $[P/x]M \rightarrow_{E\lambda\beta} [Q/x]M$
   Preuve. par induction sur M, tous les cas sont simple à prouver sauf le cas ou $M \equiv AB$ on utilise la transitivité de la relation $\rightarrow_{E\lambda\beta}$:
   $[P/x](AB) \equiv [P/x]A[P/x]B$
   $[Q/x](AB) \equiv [Q/x]A[Q/x]B$
   On a par les hypothèses d'induction les réductions suivantes :
   $[P/x]A \rightarrow_{E\lambda\beta} [Q/x]A$
   $[P/x]B \rightarrow_{E\lambda\beta} [Q/x]B$
   Donc : $[P/x]A[P/x]B \rightarrow_{E\lambda\beta} [Q/x]A[P/x]B$ et $[Q/x]A[P/x]B \rightarrow_{E\lambda\beta} [Q/x]A[Q/x]B$ par transitivité on aura :
   $[P/x]A[P/x]B \rightarrow_{E\lambda\beta} [Q/x]A[Q/x]B$ □.
4. **Proposition.** *Si $x \neq y$, $y \notin$ FV(P), niveau(P)* $\leq$ *niveau(x)* et *niveau(Q)* $\leq$ *niveau(y)* alors $[P/x][Q/y]M \equiv [[P/x]Q/y][P/x]M$
   Preuve. par une simple induction sur M □.

5. **Proposition .** Si *niveau(M)* $\leq$ *niveau(x)* et $P \rightarrow_{E\lambda\beta} Q$ alors $[M/x]P \rightarrow_{E\lambda\beta} [M/x]Q$
   Preuve. par induction sur la relation $P \rightarrow_{E\lambda\beta} Q$ □.

5.3 La preuve de Tait et Martin-Löf pour la E$\lambda\beta$-réduction

Dans cette partie nous allons expliquer les étapes de la preuves de Tait et Martin-Löf pour le système Eλ. Nous allons rependre les mêmes étapes décrites dans le chapitre 3, détailler les preuves et donner la formalisation utilisé dans le système Coq. D'abord, on rappelle la définition inductive de ($\rightarrow_{E\lambda\beta}$)[Mez02] :

$(\rho)M \rightarrow_{E\lambda\beta} M$

$(\mu)M \to_{E\lambda\beta} N \implies ZM \to_{E\lambda\beta} ZN$

$(\mu')M \to_{E\lambda\beta} N \implies X \supset M \to_{E\lambda\beta} X \supset N$

$(\mu'')M \to_{E\lambda\beta} N \implies \Pi M \to_{E\lambda\beta} \Pi N$

$(\nu)M \to_{E\lambda\beta} N \implies MZ \to_{E\lambda\beta} NZ$

$(\nu')M \to_{E\lambda\beta} N \implies M \supset X \to_{E\lambda\beta} N \supset X$

$(E\beta)(\lambda x.M)N \to_{E\lambda\beta} [N/x]M$ si *niveau(N)* $\leq$ *niveau(x)*

$(E\xi)M \to_{E\lambda\beta} N \implies \lambda x.M \to_{E\lambda\beta} \lambda x.N$

Formalisation. Nous avons défini en Coq, la relation de réduction $\to_{E\lambda\beta}$ de la façon suivante :

`Inductive` *ebeta_red* : *elambda* → *elambda* → `Prop` :=
| *beta_exist* : ∀ *M N l*, `level` *N* ≤ *l* → *ebeta_red* (*App* (*Abs M l*) *N*) (`subst` *M N* 0)
| *abs_red* :∀ *M N l*, (*ebeta_red M N*) → (*ebeta_red* (*Abs M l*) (*Abs N l*))
| *app_red_l* : ∀ *M1 N1 M* ,(*ebeta_red M1 N1*) → (*ebeta_red* (*App M1 M*) (*App N1 M*))
| *app_red_r* :∀ *M1 N1 M* ,(*ebeta_red M1 N1*) → (*ebeta_red* (*App M M1*) (*App M N1*))
| *imp_red_r* : ∀ *M1 N1 M*,(*ebeta_red M1 N1*) → (*ebeta_red* (*Imp M M1*)(*Imp M N1*))
| *imp_red_l* : ∀ *M1 N1 M* ,(*ebeta_red M1 N1*) → (*ebeta_red* (*Imp M1 M*)(*Imp N1 M*))
| *pi_red* : ∀ *M1 N1*,(*ebeta_red M1 N1*) → (*ebeta_red* (*Pi M1*)(*Pi N1*)).

`Inductive` *ebeta_star* : *elambda* → *elambda* → `Prop` :=
| *ebeta_base* : ∀ *M N* : *elambda*, (*ebeta_red M N*) → (*ebeta_star M N*)
| *ebeta_reflexive* : ∀ *M* :*elambda*, (*ebeta_star M M*)
| *ebeta_transitive* : ∀ *A B C* :*elambda*,

la relation de réduction parallèle est définie inductivement comme suit[Bar84] :

- $M \not\Rightarrow M$
- $M \not\Rightarrow M' \implies \lambda x.M \not\Rightarrow \lambda x.M'$
- $M \not\Rightarrow M' \wedge N \not\Rightarrow N' \implies MN \not\Rightarrow M'N'$
- $M \not\Rightarrow M' \wedge N \not\Rightarrow N' \implies (\lambda x.M)N \not\Rightarrow [N'/x]M'$ si $niveau(N') \leq niveau(x)$
- $M \not\Rightarrow M' \wedge N \not\Rightarrow N' \implies M \supset N \not\Rightarrow M' \supset N'$
- $M \not\Rightarrow M' \implies \Pi M \not\Rightarrow \Pi M'$

Sa fermeture transitive est notée : $(\not\Rightarrow^*)$.

Formalisation. Nous avons définie en Coq, la relation de réduction $\not\Rightarrow$ de la façon suivante :

`Inductive` *par_red1* : *elambda* → *elambda* → `Prop` :=
par_beta_exist : ∀ *M M' N N' l*, `level` *N'* ≤ *l* → (*par_red1 M M'*) → (*par_red1 N N'*) → (*par_red1* (*App* (*Abs M l*) *N*) (`subst` *M' N'* 0))
|*ref_par_red* : ∀ *n p* :*nat*, (*par_red1* (*Var* (*n*,*p*)) (*Var* (*n*,*p*)))
|*abs_par_red* : ∀ *M M' l* , (*par_red1 M M'*) → (*par_red1* (*Abs M l*) (*Abs M' l*))
|*app_par_red* : ∀ *M M' N N'* :*elambda*, (*par_red1 M M'*) → (*par_red1 N N'*) -¿(*par_red1* (*App M N*) (*App M' N'*))
|*imp_par_red* : ∀ *M M' N N'* :*elambda*, (*par_red1 M M'*) → (*par_red1 N N'*) -¿(*par_red1* (*Imp M N*) (*Imp M' N'*))
|*pi_par_red* : ∀ *M M'* :*elambda*, (*par_red1 M M'*) → (*par_red1* (*Pi M*) (*Pi M'*)).

5.3.1 La démarche de preuve

Comme décrit dans le chapitre 3, la preuve de Tait et Martin-Löf comporte 3 parties qui sont :

- Si ($\not\Rightarrow$) est fortement confluente alors sa fermeture transitive ($\not\Rightarrow^*$) est fortement confluente (DP($\not\Rightarrow$) $\implies$ DP($\not\Rightarrow^*$)). (lemme.1 *strip lemma*)
- ($\not\Rightarrow$) est fortement confluente (DP($\not\Rightarrow$)). (lemme.2)
- La fermeture transitive de la réduction parallèle est équivalente à la fermeture réflexive et transitive de la β-réduction ($\not\Rightarrow^* \iff \rightarrow^*_{E\lambda\beta}$). (lemme.3)

Dans ce qui suit, nous allons donner des preuves informelles des lemmes donnés ci-dessus avec leurs formalisations.

5.3.2 La preuve détaillée

Dans cette section, nous présentons les démonstrations des lemmes nécessaires pour démontrer le théorème de Churche-Rosser. On note que les preuves sont principalement dues à [Bar84],[Mez02],[Pol95] et [Sar13] avec quelques modifications.

Lemme 1 : ***Strip lemma***

$$\text{DP}(\not\Rightarrow) \implies \text{DP}(\not\Rightarrow^*)$$

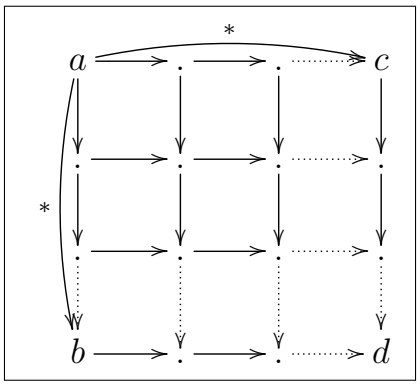

FIGURE 5.1 – Strip Lemma

Preuve. On parcourt le diagramme avec une double induction (Horizontalement et verticalement) comme le montre la figure ci-dessus [Bar84]. □.

Formalisation. Pour démontrer ce lemme en coq , nous l'avons divisé en deux sous cas :

`Definition` *confluence* (*A* :`Set`)(*R* : $A \to A \to$ `Prop`) :=

$\forall$ *x y* :*A*, (*R x y*) $\to \forall$ *z*, (*R x z*) $\to \exists$ *u* :*A*, ((*R y u*) $\wedge$ (*R z u*)).

`Definition` *strip* := $\forall$ *x y* :*elambda*, (*ebeta_par_red x y*) $\to$

$\forall$ *z* :*elambda*, (*par_red1 x z*) $\to \exists$ *u* :*elambda*, (*par_red1 y u*) $\wedge$ (*ebeta_par_red z u*).

`Lemma` *strip_lemma_r* : (*confluence elambda par_red1*) $\to$ *strip*.

`Lemma` *strip_lemme_l* : *strip* $\to$ (*confluence elambda ebeta_par_red*).

Lemme de substitution

Ce lemme est nécessaire pour prouver la confluence forte de la réduction parallèle.

Soient M, N, M', N' des λ-termes et x une variable, Si $M \not\Rightarrow M'$ et $N \not\Rightarrow N'$ et *niveau(N)* $\leq$ *niveau(x)* alors $[N/x]M \not\Rightarrow [N'/x]M'$.

Preuve. Par induction sur la définition de $M \not\Rightarrow M'$[Bar84] :

- Cas 1. $M \not\Rightarrow M' \equiv M \not\Rightarrow M$. Alors on démontre que $[N/x]M \not\Rightarrow [N'/x]M$ par induction sur la structure de M comme le montre le tableau ci-dessous [Bar84] :
- Cas 2. $M \not\Rightarrow M' \equiv \lambda y.P \not\Rightarrow \lambda y.P'$ et qui constitue une conséquence directe de $P \not\Rightarrow P'$. Donc Avec l'hypothèse d'induction on a $[N/x]P \not\Rightarrow [N'/x]P'$. Donc $\lambda y.[N/x]P \not\Rightarrow \lambda y.[N'/x]P'$, i.e. $[N/x]M \not\Rightarrow [N'/x]M'$.
- Cas 3. $M \not\Rightarrow M' \equiv PQ \not\Rightarrow P'Q'$ et qui constitue une conséquence directe de $P \not\Rightarrow P', Q \not\Rightarrow Q'$, alors :

M	Partie gauche	Partie droite	Commentaire
x	N	N'	Trivial
y	y	y	Trivial
PQ	$[\,]P[\,]Q$	$[\,']P\,[\,']Q$	En utilisant l'hypothèse d'induction
$\lambda y.P$	$\lambda.[\,]P$	$\lambda y.[\,']P$	En utilisant l'hypothèse d'induction
$P \supset Q$	$[\,]P \supset [\,]Q$	$[\,']P \supset [\,']Q$	En utilisant l'hypothèse d'induction
ΠP	$\Pi[\,]P$	$\Pi[\,']P$	En utilisant l'hypothèse d'induction

$[N/x]M \equiv [N/x]P[N/x]Q \not\Rightarrow [N'/x]P'[N'/x]Q'$, par l'hypothèse d'induction.

$\equiv [N'/x]M'$

— Cas 4. $M \not\Rightarrow M' \equiv P \supset Q \not\Rightarrow P' \supset Q'$ et qui constitue une conséquence directe de $P \not\Rightarrow P', Q \not\Rightarrow Q'$, alors :

$[N/x]M \equiv [N/x]P \supset [N/x]Q \not\Rightarrow [N'/x]P' \supset [N'/x]Q'$, par l'hypothèse d'induction.

$\equiv [N'/x]M'$

— Cas 5. $M \not\Rightarrow M' \equiv \Pi.P \not\Rightarrow \Pi.P'$ et qui constitue une conséquence directe de $P \not\Rightarrow P'$. En utilisant l'hypothèse d'induction on a $[N/x]P \not\Rightarrow [N'/x]P'$. Donc $\Pi.[N/x]P \not\Rightarrow \Pi.[N'/x]P'$, i.e. $[N/x]M \not\Rightarrow [N'/x]M'$.

— Cas 6. $M \not\Rightarrow M' \equiv (\lambda y.P)Q \not\Rightarrow [Q'/y]P'$ avec $niveau(Q) \leq niveau(y)$ et qui constitue une conséquence directe de $P \not\Rightarrow P', Q \not\Rightarrow Q'$ alors :
$[N/x]((\lambda y.P)Q) \not\Rightarrow [N/x][Q'/y]P' \equiv (\lambda y.[N/x]P)[N/x]Q \not\Rightarrow [N'/x][Q'/y]P'$
$\equiv (\lambda y.[N/x]P)[N/x]Q \not\Rightarrow [[N'/x]Q'/y][N'/x]P'$
Posons $A = [N/x]P$ et $B = [N/x]Q$ alors $A' = [N'/x]P'$ et $B' = [N'/x]Q'$ avec *niveau(N)* $\leq$ *niveau(x)* et *niveau([N/x]Q)* $\leq$ *niveau(y)* Qui est obtenu en utilisant la proposition 1 à partir de *niveau(Q)* $\leq$ *niveau(y))*
On a alors : $(\lambda y.A)B \not\Rightarrow [B'/y]A'$ avec *niveau(B)* $\leq$ *niveau(y)* (Constructeur de $\not\Rightarrow$)
□.

Formalisation. Nous avons formalisé en Coq,le lemme de substitution de la façon suivante : `Lemma` *subst_lemma* : ∀ *M M' N N' l x* , `level` (*N*) ≤ *l* → *par_red1 M M'* → *par_red1 N N'* → *par_red1* (`subst` *M N x*) (`subst` *M' N' x*).

Lemme 2 : La Confluence forte de la réduction parallèle

$$\mathrm{DP}(\not\Rightarrow)$$

Preuve. La preuve se fait par induction sur la définition de $M \not\Rightarrow M_1$, on va montrer que pour toute $M \not\Rightarrow M_2$ il y a un M_3 où $M_1 \not\Rightarrow M_3, M_2 \not\Rightarrow M_3$[Bar84].

- Cas 1. Lorsqu'on a $M \not\Rightarrow M_1$ avec $M \equiv M_1$ alors on prend $M_3 \equiv M_2$(réflexivité).
- Cas 2. Quand $M \not\Rightarrow M_1$ équivaut à $(\lambda x.P)Q \not\Rightarrow [Q'/x]P'$ avec *niveau(Q')* $\leq$ *niveau(x)* et c'est une conséquence directe de $P \not\Rightarrow P', Q \not\Rightarrow Q'$. En utilisant le lemme d'abstraction[Bar84], on distingue deux sous cas :
  - Sous cas 2.1. Lorsqu'on a $M_2 \equiv (\lambda x.P'')Q''$ avec $P \not\Rightarrow P'', Q \not\Rightarrow Q''$. En utilisant l'hypothèse d'induction on aura les termes P''', Q''' avec $P' \not\Rightarrow P''', P'' \not\Rightarrow P'''$ (et $Q' \not\Rightarrow Q''', Q'' \not\Rightarrow Q'''$), donc avec le lemme de substitution on prend $M_3 \equiv [Q'''/x]P'''$ avec *niveau(Q'')* $\leq$ *niveau(x)* car *niveau(Q')* $\leq$ *niveau(x)* et $Q' \not\Rightarrow Q'''$ (Proposition 2).
  - Sous cas 2.2. Lorsqu'on a $M_2 \equiv [Q''/x]P''$ avec $P \not\Rightarrow P'', Q \not\Rightarrow Q''$. En utilisant l'hypothèse d'induction, on prend $M_3 \equiv [Q'''/x]P'''$ avec *niveau(Q'')* $\leq$ *niveau(x)* car *niveau(Q')* $\leq$ *niveau(x)* et $Q' \not\Rightarrow Q'''$ (Proposition 2).
- Cas 3. Lorsque $M \not\Rightarrow M_1$ équivaut à $PQ \not\Rightarrow P'Q'$, ceci constitue une conséquence directe de $P \not\Rightarrow P', Q \not\Rightarrow Q'$. On distingue encore deux sous cas :
  - Sous cas 3.1. Lorsqu'on a $M_2 \equiv P''Q''$ avec $P \not\Rightarrow P'', Q \not\Rightarrow Q''$. Alors, en utilisant l'hypothèse d'induction, on prend $M_3 \equiv P'''Q'''$.
  - Sous cas 3.2. Lorsqu'on a $P \equiv (\lambda x.P_1), M_2 \equiv [Q''/x]P_1''$ avec $P_1 \not\Rightarrow P_1'', Q \not\Rightarrow Q''$ et *niveau(Q'')* $\leq$ *niveau(x)*. En utilisant le lemme d'abstraction[Bar84] on aura $P' \equiv \lambda x.P_1'$ avec $P_1 \not\Rightarrow P_1'$. En utilisant l'hypothèse d'induction, on peut prendre $M_3 \equiv [Q'''/x]P_1'''$ avec *niveau(Q'')* $\leq$ *niveau(x)*(Proposition 2) comme le montre la figure ci-dessous :

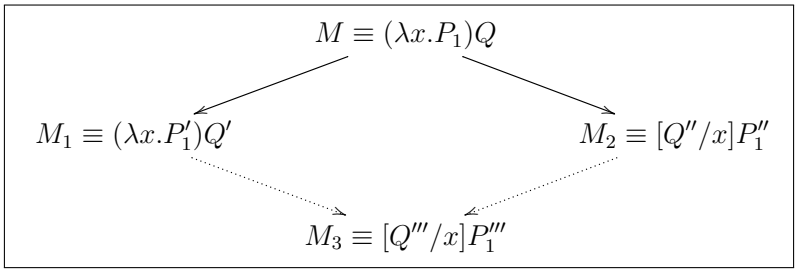

FIGURE 5.2 – Preuve confluence forte

— Cas 4. Lorsqu'on a $M \not\Rightarrow M_1$ équivaut à $\lambda x.P \not\Rightarrow \lambda x.P'$, et qui constitue une conséquence directe de $P \not\Rightarrow P'$. Alors $M_2 \equiv \lambda x.P''$. Par l'hypothèse d'induction, on peut prendre $M_3 \equiv \lambda x.P'''$.

— Cas 5. Lorsqu'on a $M \not\Rightarrow M_1$ équivaut à $P \supset Q \not\Rightarrow P' \supset Q'$, et qui constitue une conséquence directe de $P \not\Rightarrow P'$ et $Q \not\Rightarrow Q'$. Alors $M_2 \equiv P'' \supset Q''$. En utilisant l'hypothèse d'induction, on peut prendre $M_3 \equiv P''' \supset Q'''$.

— Cas 6. Lorsqu'on a $M \not\Rightarrow M_1$ équivaut à $\Pi.P \not\Rightarrow \Pi P'$, et qui constitue une conséquence directe de $P \not\Rightarrow P'$. Alors $M_2 \equiv \Pi P''$. Par l'hypothèse d'induction, on peut prendre $M_3 \equiv \Pi P'''$. □.

Formalisation. Nous avons énoncé en Coq, le lemme de la confluence forte de la relation de réduction ($\not\Rightarrow$) de la façon suivante :

`Lemma` *lemma2* : ∀ *M M1* : *elambda*,
par_red1 M M1 → ∀ *M2* , *par_red1 M M2* → ∃ *M3* : *elambda*, *par_red1 M1 M3* ∧ *par_red1 M2 M3*.

Lemme 3 : L'équivalence $\not\Rightarrow^* \iff \rightarrow^*_{E\lambda\beta}$

Preuve. On démontre les deux formules suivantes :

- $\not\Rightarrow^* \implies \rightarrow^*_{E\lambda\beta}$ (simple induction sur la relation $\not\Rightarrow^*$).
- $\rightarrow^*_{E\lambda\beta} \implies \not\Rightarrow^*$ (simple induction sur la relation $\rightarrow^*_\beta$). □.

Formalisation. Nous avons énoncé en Coq, l'équivalence entre $\not\Rightarrow^*$ et $\rightarrow^*_{E\lambda\beta}$ de la façon suivante :

`Lemma` *ebeta_red_par_red* : ∀ *M N* :*elambda*, (*ebeta_star M N*) → (*ebeta_par_red M N*).
`Lemma` *par_red_ebeta_par* : ∀ *M N* :*elambda*, (*ebeta_par_red M N*) → (*ebeta_star M N*).

5.4 Conclusion

Dans ce chapitre nous avons expliqué la démarche que nous avons suivie pour démontrer la confluence de la E$\lambda\beta$-réduction. On note que la partie la plus difficile est le lemme de substitution, qui nécessite une très longue analyse.

Conclusion

La confluence est une propriété fondamentale des systèmes de réécriture. D'où l'importance accordée à ce résultat dans la théorie du λ-calcul. Notre projet consiste en la formalisation de la preuve de la confluence de la E$\lambda\beta$-réduction dans l'assistant de preuve Coq.

Pour cela, nous avons commencé avec une présentation générale sur le système Coq. Nous avons également introduit la théorie du λ-calcul, sa syntaxe et ses règles de calcul, puis présenté la théorie Eλ, qui est une extension du λ-calcul pur, pouvant interpréter la logique d'ordre supérieur. Par la suite, nous avons décrit la démarche suivie pour démonter la confluence de la relation de réduction E$\lambda\beta$-réduction, ainsi que sa formalisation dans le système Coq.

À travers ce travail, nous avons acquis des connaissances avancées sur le système Coq et sur la logique d'ordre supérieur. Nous avons étudié le système Eλ, touché de prés le domaine de la spécification/vérification formelle et compris que savoir poser un problème le résout à 80%, d'où la nécessité d'avoir une bonne spécification formelle au départ d'un tel projet.

Bibliographie

[Bar84] H. P. Barendregt, *The Lambda Calculus : Its Syntaxe and Semantics, volume 103 of Studies in logic and the Foundations of Mathematics*. North-Holland, Amesterdam, revised edition, 1984.

[Ber06] Yves Bertot, *introduction au lambda-calcul pur* . INRIA, 2006. Consulté le : 09/05/2016. Disponible à : *http://www-sop.inria.fr/members/Yves.Bertot/courses/lambda-pur.pdf*.

[Ber15] Yves Bertot, Pierre Castéran *Le Coq' Art (V8)* , 2015 Consulté le : 17/04/2016. Disponible à : *https://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf*.

[Ber09] Gérard Berry. *Le lambda-calcul : réductions, causalité et déterminisme* Séminaire : *Penser, modéliser et maîtriser le calcul informatique (Chaire Informatique et sciences numériques)*. Collège de France. Consulté le : 04/04/2016. Disponible à : *https://www.college-de-france.fr/site/gerard-berry/course-2009-12-02-10h00.htm*.

[Ber15] Gérard Berry. *Des logiques d'ordre supérieur à la programmation vérifiée en Coq* Séminaire : *Prouver les programmes : pourquoi, quand, comment*. Collège de France. Consulté le : 04/04/2016. Disponible à : *https://www.college-de-france.fr/site/gerard-berry/course-2015-03-18-16h00.htm*.

[Cdt16] The Coq Development Team. *The Coq Proof Assistant Reference Manual version 8.5pl1*. April 2016.

[Cha03] Mohamed Chaabani , *Spécification, Preuve et Extraction Automatique des programmes en Coq Cas : l'algorithme $\lambda c\beta^{+}$-réduction* , UMBB , 2003.

[Chl16] Adam Chlipala. *Certified Programming with Dependent Types* MIT, March 2016. Consulté le : 10/04/2016. Disponible à : *http://adam.chlipala.net/cpdt/cpdt.pdf*.

[Deb72] Nicolas G. de Bruijn. *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. Indag*. Math., 34(5), 1972.

[Gou14] Jean Goubault-Larrecq, *Lambda-calcul et langages fonctionnels*, ENS Cachant 2014. Consulté le : 17/03/2016. Disponible à : *http://www.lsv.ens-cachan.fr/~goubault/Lambda/lambda.pdf*.

[Hin08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators, an introduction*, Cambridge University Press, 2008.

[Hue91] Gérard Huet. *Introduction au λ-calcul pur*. NATO ASI Series Vol F.79 :153-200, 1991.

[Hue94] Gérard Huet. *Residual theory in λ-calculus : A formal development*. Journal of Functional Programming, 4(3) :371-394, July 1994.

[Hue09] Gérard Huet. *Lambda-calcul, logique et linguistique* Séminaire : *Penser, modéliser et maîtriser le calcul informatique (Chaire Informatique et sciences numériques)*. Collège de France. Consulté le : 23/03/2016. Disponible à : *https://www.college-de-france.fr/site/gerard-berry/seminar-2009-12-02-11h00.htm*.

[Mez02] Mohamed Mezghiche, Choukri Ben-yelles. *Elambda-calculus : A higher order logic extension of untyped λ-calculus. Thirty Five years of Automath*, Heriot-Watt University, Edinburgh April 2002.

[Mez09] Mohamed Mezghiche, Support de Cours : *Logique constructive, spécification et programmation sûre*. UMBB, 2009.

[Pal15] Christine Paulin. *Langages et systèmes pour la preuve interactive* Séminaire : *Prouver les programmes : pourquoi, quand, comment*. Collège de France. Consulté le : 29/05/2016. Disponible à : *https://www.college-de-france.fr/site/gerard-berry/seminar-2015-03-18-17h30.htm*.

[Pie16] Benjamin C. Pierce et al. *Software Foundations* Penn Engineering, May 2016 [En Ligne]. Consulté le 02/06/2016. Disponible a l'adresse : *https://www.cis.upenn.edu/~bcpierce/sf/current/index.html*.

[Pol95] Robert Pollack. *Polishing up the Tait-Martin-Löf proof of the Church-Rosser theorem*. Chalmers Univ. of Technology, Göteborg, Sweden, January 1995.

[Sau13] Alexis Saurin. *Notes de cours λ-calcul : des abstractions aux applications*, Univ Paris Diderot , 2013 *https://www.irif.univ-paris-diderot.fr/users/saurin/Enseignement/LMFI/2013-14/notes_LMFI.pdf*.

[Sha88] N. Shankar. *A mechanical proof of the Church-Rosser theorem*. Journal of the ACM, 35(3) :475-522, July 1988.

[Tak95] Masako Takahashi. *Parallel reduction in λ-calculus* (revised version). Information and Computation, 118(1) :120-127, April 1995.