Démonstration Automatique dans le Calcul des Constructions

Gilles Dowek

THESE DE DOCTORAT

présentée

A L'UNIVERSITE PARIS 7

Spécialité : Informatique Fondamentale

par

Gilles DOWEK

Sujet de la thèse :

Démonstration Automatique dans le Calcul des Constructions

Soutenue le 16 décembre 1991 devant la Commission d'examen composée de :

MM.	**Dominique Perrin**	**Président**
	Thierry Coquand	**Rapporteurs**
	Michel Parigot	
	Hassan Aït-Kaci	**Examinateurs**
	Gérard Huet	
	Gordon Plotkin	
	Christian Queinnec	

Je tiens, avant toute chose, à remercier Monsieur le Professeur Dominique Perrin de l'honneur qu'il me fait en présidant le jury de cette thèse. Je voudrais également remercier Messieurs Thierry Coquand et Michel Parigot qui ont lu ces pages avec une grande patience et Messieurs Hassan Aït-Kaci, Gordon Plotkin et Christian Queinnec de l'intérêt qu'ils portent à ce travail en formant le jury qui le sanctionne.

Gérard Huet m'a tout d'abord proposé ce sujet de recherche, il en a ensuite constamment suivi le déroulement, modérant de sa lucidité mes excès d'enthousiasme et de découragement. J'aimerais qu'il sache combien je l'en remercie.

Nombre des idées présentées ici m'ont été suggérées par des membres du projet Formel de l'INRIA, qui m'a accueilli pour ce travail. En particulier, l'idée d'utiliser des termes marqués pour prouver le résultat du chapitre 2 m'a été donnée par Gérard Huet et celle d'utiliser un codage à la Harper-Honsel-Plotkin pour prouver la normalisation de la réduction sur ces termes par Christine Paulin-Mohring. Il va sans dire que l'influence de cet exceptionnel environnement de travail a été déterminante.

Table des matières

Pour projeter un livre, la première
chose est de savoir exclure.
Italo Calvino

La démonstration automatique a pour projet la conception de méthodes qui, partant d'une proposition, en construisent automatiquement une preuve, c'est-à-dire une argumentation en justifiant la vérité.

Un sens est donné à ce projet par le caractère grammatical (ou formel) de la notion de preuve. En effet si on considère que la suite de symboles "Tous les hommes sont mortels, Socrate est un homme donc Socrate est mortel" est une preuve de la proposition "Socrate est mortel", ce n'est pas en raison de la signification des mots "Socrate", "homme", "mortel" mais uniquement en raison de la façon dont ces mots sont assemblés dans cette phrase. Cette remarque permet de distinguer deux étapes dans le projet de construction automatique de preuves. La première est la formalisation des règles qui distinguent les preuves au sein des suites de symboles, la seconde est leur automatisation. La première de ces tâches est l'objet de la *théorie de la démonstration*, théorie qui dépasse largement le cadre de la démonstration automatique et qui lui est bien antérieure, la seconde est l'objet de la *démonstration automatique* proprement dite.

Dans son projet de formalisation du raisonnement, la théorie de la démonstration n'aboutit pas à un système formel unique, mais plusieurs peuvent prétendre être une formalisation du raisonnement naturel. Tous ces systèmes ont plus ou moins en commun une organisation qui distingue un langage de description des objets du discours, un langage d'expression des propositions concernant ces objets et des règles de déductions qui permettent de démontrer certaines de ces propositions. Les choix qui orientent vers un système ou un autre concernent deux points.

Le premier est le choix des connecteurs logiques et des règles de déduction régissant ces connecteurs. Aux connecteurs habituels (implication, quantification universelle et existentielle, conjonction, disjonction, négation) peuvent s'ajouter des modalités comme la nécessité ou la possibilité. De plus une certaine liberté est laissée dans le choix des règles régissant certains de ces connecteurs, notamment des règles régissant la négation et des règles structurelles. On parle alors de logique *classique*, *intuitionniste*, *minimale*, *linéaire*, *relevante*, *modale*, *temporelle*, etc.

L'autre point est le langage des objets du discours. Par exemple en arithmétique, on peut choisir de traiter uniquement des entiers ou alors de traiter aussi des collections d'entiers, des fonctions des

entiers dans les entiers, etc. On parle alors de logique du *premier ordre*, du *deuxième ordre*, d'*ordre supérieur*, de *systèmes de types*, etc. Trois grandes étapes peuvent être distinguées dans l'histoire de ces langages :

• En logique du premier ordre, les objets sont représentés par des arbres. Il n'est pas possible de parler de collections ou de fonctions sans passer par un codage, tel que celui proposé par la théorie des ensembles.

• En logique d'ordre supérieur, les objets sont représentés par des λ-termes simplement typés, on peut alors parler de collections et de fonctions, mais dans un formalisme très pauvre.

• Dans les systèmes de types, on enrichit le λ-calcul simplement typé par des types dépendants, des types polymorphes, des constructeurs de types et des types inductifs. On a alors deux propriétés essentielles : le *principe de compréhension* (toute fonction dont la totalité peut être prouvée peut être représentée par un terme) et la *représentabilité des preuves* (toute preuve peut être représentée par un terme au sens de la sémantique de Heyting et de l'isomorphisme de Curry-Howard).

Les choix relatifs aux connecteurs et aux règles de déduction sont peu explorés en démonstration automatique, on se limite en général aux logiques classique et intuitionniste. Les choix relatifs au langage des termes sont en revanche source de nombreux travaux. Une méthode de construction automatique de preuves pour la logique du premier ordre a été décrite en 1965 par Robinson sous le nom de *résolution*. Cette méthode à été généralisée par Huet en 1972 à la logique d'ordre supérieur. Nous la généralisons ici aux systèmes de types. Nous ne considérons que les systèmes de types avec types dépendants, types polymorphes et constructeurs de types, c'est-à-dire les huit systèmes du cube de Barendregt, en particulier le Calcul des Constructions. Nous n'abordons pas le problème des types inductifs mais les idées développées ici peuvent vraisemblablement se généraliser à ces systèmes.

En un certain sens, les systèmes de types étant une généralisation de la logique d'ordre supérieur, le problème de la démonstration automatique y est plus compliqué et les méthodes développées plus inefficaces. Néanmoins le fait que ces systèmes vérifient le principe de compréhension, c'est-à-dire leur richesse calculatoire, permet de rendre triviales certaines preuves d'égalités, comme $(fact\ 6) = 720$, puisque démontrer cette proposition dans un système de types revient simplement à normaliser ces deux termes et remarquer que leurs formes normales sont identiques, alors que cela demande l'utilisation répétée d'axiomes concernant la factorielle dans les systèmes précédents. Par ailleurs, nous soutenons que l'identification qui est faite dans ces systèmes entre preuves et objets et qui simplifie leur expression, simplifie également les algorithmes de construction de preuves. En particulier la distinction habituelle entre résolution et unification n'y est plus pertinente.

Après une rapide présentation des systèmes de types au *chapitre 1*, le *chapitre 2* présente un résultat de bonne fondation utilisé à plusieurs reprises par la suite. Ce chapitre peut être omis lors d'une première lecture.

La méthode de démonstration automatique est ensuite présentée dans les *chapitres 3 et 4*. Cette méthode n'utilise pas un algorithme d'unification, mais celui-ci est en quelque sorte uniformément réparti dans l'algorithme de résolution. Une restriction de cette méthode est présentée au *chapitre 5* ainsi que des applications à la synthèse partielle de preuves et à l'analyse de preuves partielles.

Les chapitres suivants sont consacrés à la résolution d'équations. Nous montrons au *chapitre 6* qui si nous n'avons pas besoin d'un algorithme d'unification pour notre méthode de synthèse de preuves, celle-ci peut en revanche être utilisée comme algorithme d'unification, tout au moins comme méthode de recherche d'unificateurs fermés. Quelques pistes vers un algorithme de recherche d'unificateurs ouverts sont étudiées bien que l'intérêt de ce problème soit discutable. Deux cas particuliers

de l'unification sont connus comme décidables dans le λ-calcul simplement typé : l'*unification du premier ordre* (récemment généralisée par Miller à l'*unification de termes à arguments restreints*) et le *filtrage du deuxième ordre*. Le premier a été généralisé aux systèmes de types par Pfenning, nous généralisons le deuxième au *chapitre 7*. Enfin au *chapitre 8* nous étudions quelques cas particuliers indécidables de l'unification : l'unification du deuxième ordre est indécidable comme dans le λ-calcul simplement typé, nous montrons également que, contrairement à ce qui se passe dans le λ-calcul simplement typé (du moins dans l'état actuel de nos connaissances), le filtrage est indécidable dès qu'on a soit des types dépendants, soit des constructeurs de types, soit des types polymorphes, soit des types inductifs.

Préliminaires

Chapitre 1

Trois Formalismes Logiques

Nous présentons successivement trois systèmes logiques qui formalisent le raisonnement naturel. Dans le premier d'entre eux, la *logique du premier ordre*, tous les objets du discours appartiennent à une même collection, par exemple, en arithmétique, les nombres entiers. Le deuxième, la *logique d'ordre supérieur*, est une extension de ce système qui permet de traiter également des fonctions de cette collection dans elle-même et des parties de cette collection, mais dans un formalisme assez pauvre. Enfin, les *systèmes de types* généralisent la logique d'ordre supérieur en enrichissant le formalisme de description des fonctions et des collections.

1.1 Logique du Premier Ordre

1.1.1 Termes

On se donne un ensemble de variables et un ensemble de symboles de fonctions. A chaque symbole de fonction on associe un entier : son arité. En arithmétique par exemple on se donne les symboles de fonction 0, S, $+$ et $*$ (lire *zéro*, *successeur*, *plus* et *fois*).

Définition 1. *: Terme*
L'ensemble des termes est défini inductivement par les règles suivantes :
• *les variables sont des termes,*
• *si $t_1, ..., t_n$ sont des termes et F un symbole de fonction d'arité n, alors $(F\ t_1\ ...\ t_n)$ est un terme.*

Par exemple 0, $(S\ 0)$, $(S\ (S\ 0))$, $(+\ (*\ x\ x)\ (S\ (S\ (S\ 0))))$ sont des termes.

1.1.2 Propositions

On se donne un ensemble de symboles de prédicats. A chaque symbole de prédicat on associe un entier : son arité. Par exemple, en arithmétique, on se donne les prédicats $=$ et $\leq$ (lire *égal* et *inférieur ou égal*).

Définition 2. *: Proposition*
L'ensemble des propositions est défini inductivement par :
• *si $t_1, ..., t_n$ sont des termes et P un symbole de prédicat d'arité n, alors $(P\ t_1\ ...\ t_n)$ est une proposition,*

- *si P et Q sont des propositions, alors $P \to Q$ est une proposition (lire P implique Q),*
- *si P et Q sont des propositions, alors $P \wedge Q$ est une proposition (lire P et Q),*
- *si P et Q sont des propositions, alors $P \vee Q$ est une proposition (lire P ou Q),*
- *si P est une proposition, alors $\neg P$ est une proposition (lire non P),*
- *$\bot$ est une proposition (lire absurde),*
- *si P est une proposition, alors $\forall x.P$ est une proposition (lire pour tout x, P),*
- *si P est une proposition, alors $\exists x.P$ est une proposition (lire il existe x tel que P).*

Par exemple $\exists x.(= \ (* \ x \ x)(S \ (S \ (S \ (S \ 0)))))$ et $\exists x.(= \ (+ \ (* \ x \ x) \ (S \ 0)) \ 0)$ sont deux propositions exprimant respectivement l'existence d'un entier dont le carré est égal à quatre et l'existence d'un entier dont le carré plus un est nul.

1.1.3 Déduction Naturelle

La *déduction naturelle* [23] est une description des règles qui permettent de déduire une proposition (théorème) à partir d'autres propositions (axiomes ou hypothèses) . Soit Γ un ensemble de propositions et P une proposition, on note $\Gamma \vdash P$ le jugement P *est démontrable sous les hypothèses* Γ. On définit inductivement ce jugement par un ensemble de règles. Par exemple la règle :

$$\frac{\Gamma \vdash P \to Q \quad \Gamma \vdash P}{\Gamma \vdash Q}$$

indique que si on a déjà prouvé les propositions $P \to Q$ et P sous les hypothèses Γ, alors on peut en déduire la proposition Q sous ces mêmes hypothèses.

Définition 3. *: Variable Libre, Substitution*

Soit x une variable et t un terme, la variable x est dite libre dans le terme t si $t = x$ ou $t = (F \ t_1 \ ... \ t_n)$ et x est libre dans l'un des t_i.

Soit x une variable et Q une proposition, la variable x est dite libre dans la proposition Q si l'une des conditions suivantes est vérifiée :

- *$Q = (P \ t_1 \ ... \ t_n)$ et x est libre dans un des termes t_i,*
- *$Q = P_1 \to P_2$, $Q = P_1 \wedge P_2$ ou $Q = P_1 \vee P_2$, et x est libre dans P_1 ou dans P_2,*
- *$Q = \neg P$ et x est libre dans P*
- *$Q = \exists y.P$ ou $Q = \forall y.P$, $y \neq x$ et x est libre dans P.*

Soit P une proposition, x une variable et t un terme, la proposition $P[x \leftarrow t]$ est obtenue en substituant le terme t à toutes les occurrences libres de x dans P.

Remarque : On suppose que toutes les variables muettes ont un nom différent. De ce fait, les substitutions n'introduisent pas de captures de variables. De même on ne distingue pas les propositions $\forall x.(P \ x)$ et $\forall y.(P \ y)$ (α-équivalence). Une présentation rigoureuse de tous les formalismes présentés ici utiliserait, par exemple, des indices de de Bruijn [2].

Définition 4. *: Règles de Déduction*

$$\frac{P \in \Gamma}{\Gamma \vdash P} \, lien \ axiome$$

$$\frac{\Gamma[P] \vdash Q}{\Gamma \vdash P \to Q} \to -intro$$

$$\frac{\Gamma \vdash P \to Q \quad \Gamma \vdash P}{\Gamma \vdash Q} \to -elim$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \wedge -intro$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \wedge -elim$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \wedge -elim$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \vee -intro$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \vee -intro$$

$$\frac{\Gamma \vdash P \vee Q \quad \Gamma \vdash P \to R \quad \Gamma \vdash Q \to R}{\Gamma \vdash R} \vee -elim$$

$$\frac{\Gamma[P] \vdash \bot}{\Gamma \vdash \neg P} \neg -intro$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash \neg P}{\Gamma \vdash \bot} \neg -elim$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \bot -elim$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x.P} \forall -intro \quad x \text{ non libre dans une proposition de } \Gamma$$

$$\frac{\Gamma \vdash \forall x.P}{\Gamma \vdash P[x \leftarrow t]} \forall -elim$$

$$\frac{\Gamma \vdash P[x \leftarrow t]}{\Gamma \vdash \exists x.P} \exists -intro$$

$$\frac{\Gamma \vdash \exists x.P \quad \Gamma \vdash \forall x.(P \to Q)}{\Gamma \vdash Q} \exists -elim \quad x \text{ non libre dans } Q$$

Cet ensemble de règles définit la logique *intuitionniste*. On définit la logique *classique* en ajoutant la règle :

$$\frac{}{\Gamma \vdash P \vee \neg P} \text{ classique}$$

1.2 Logique d'Ordre Supérieur

1.2.1 λ-calcul

λ-notation

En logique du premier ordre, les termes sont des arbres. Les termes sans variables forment une collection (par exemple en arithmétique les entiers). Il n'est pas possible d'exprimer des propositions concernant les fonctions de cette collection dans elle-même, ni les parties de cette collection sans passer par un codage tel que la théorie des ensembles qui considère une collection unique qui contient pêle-mêle les objets de base, les fonctions de la collection de ces objets dans elle-même et les parties de cette collection.

Le λ-calcul [7] est une extension du formalisme des arbres dans lequel un terme sans variables est un objet de base (par exemple un entier), une fonction de l'ensemble des objets de base dans lui-même ou une partie de cet ensemble.

Ordinairement on décrit une fonction f par la notation $f : x \mapsto t$ où x est une variable (dite *muette*) et t un terme. L'expression $x \mapsto t$ ne peut être utilisée que lors d'une telle définition, il n'est pas d'usage de l'utiliser à l'intérieur d'un terme.

En λ-calcul on note cette fonction $[x]t$ (ou dans la notation originale $\lambda x.t$). Par exemple $[x](+\ (*\ x\ x)\ (S\ (S\ 0))$ est la fonction qui associe à tout entier x l'entier $(+\ (*\ x\ x)\ (S\ (S\ 0))$. Contrairement à l'expression $x \mapsto (+\ (*\ x\ x)\ (S\ (S\ 0))$, le terme $[x](+\ (*\ x\ x)\ (S\ (S\ 0)))$ peut à son tour être utilisé à l'intérieur d'un autre terme. Par exemple, on peut appliquer ce terme au terme $(+\ y\ y)$ de façon à former $([x](+\ (*\ x\ x)\ (S\ (S\ 0)))\ (+\ y\ y))$, dans lequel on peut ensuite abstraire la variable y, ce qui donne le terme $[y]([x](+\ (*\ x\ x)\ (S\ (S\ 0)))\ (+\ y\ y))$ et ainsi de suite.

Définition 5. *: Terme du λ-calcul*

$$T ::= x \mid (T\ T) \mid [x]T$$

Les termes de la forme $(t\ t')$ s'appellent des *applications* et les termes de la forme $[x]t$ des *abstractions* .

Remarque : En λ-calcul, les symboles de fonctions ne forment plus une catégorie syntaxique à part, mais sont des variables ordinaires.

Réduction et Equivalence

Quand on applique le terme $[x](+\ (*\ x\ x)\ (S\ (S\ 0)))$ à l'entier $(S\ 0)$, on veut en fait substituer à l'argument formel x, l'argument effectif $(S\ 0)$ pour obtenir le terme $(+\ (*\ (S\ 0)\ (S\ 0))\ (S\ (S\ 0)))$. On dit que le terme $([x](+\ (*\ x\ x)\ (S\ (S\ 0)))\ (S\ 0))$ se β-réduit sur le terme $(+\ (*\ (S\ 0)\ (S\ 0))\ (S\ (S\ 0)))$. Le terme $(+\ (*\ (S\ 0)\ (S\ 0))\ (S\ (S\ 0)))$ ne peut plus se réduire, il est dit en forme β-normale. On dit également que les termes $([x](+\ (*\ x\ x)\ (S\ (S\ 0)))\ (S\ 0))$ et le terme $(+\ (*\ (S\ 0)\ (S\ 0))\ (S\ (S\ 0)))$ sont β-équivalents.

Définition 6. *: Réduction et Equivalence*

La relation de β-réduction (en une étape) $\rhd$ est la plus petite relation telle que :

— $([x]t\ u) \rhd t[x \leftarrow u]$,

— *si* $t_1 \rhd t_2$, *alors* $(t_1\ u) \rhd (t_2\ u)$,

— *si* $t_1 \rhd t_2$, *alors* $(u\ t_1) \rhd (u\ t_2)$,

— *si* $t_1 \rhd t_2$, *alors* $[x]t_1 \rhd [x]t_2$.

La relation de β*-réduction* $\rhd^*$ *est la plus petite relation réflexive et transitive qui contient la relation* $\rhd$. *La relation de* β*-équivalence* $\equiv$ *est la plus petite relation d'équivalence qui contient la relation* $\rhd$. *Un terme* t *est dit normal s'il n'existe pas de terme* t' *tel que* $t \rhd t'$.

Une autre forme similaire de réduction est la η*-réduction. Si* f *est un terme tel que* x *ne soit pas libre dans* f, *on veut que le terme* $[x](f\ x)$ *se réduise sur le terme* f. *On ajoute pour cela la règle de réduction :*

— $[x](f\ x) \rhd f$ *si* x *non libre dans* f.

Types Simples

Les règles de formation des termes énoncées ci-dessus permettent de former le terme $\delta = [x](x\ x)$. Il est difficile de comprendre ce terme, qui est une fonction qui prend en argument une fonction et l'applique à elle-même. De plus si on applique ce terme à lui-même $(\delta\ \delta) = ([x](x\ x)\ [x](x\ x))$ on obtient un terme qui se réduit sur lui-même et n'a donc pas de forme normale. Il est encore plus difficile de donner un sens à ce terme qui désigne en quelque sorte un objet en en donnant une méthode de calcul, laquelle méthode ne termine pas.

Si ces objets pathologiques peuvent être construits, c'est parce que nos règles de formation des termes sont trop laxistes. En effet la règle d'application permet d'appliquer n'importe quel terme t à n'importe quel terme t', sans s'assurer que t est bien une fonction et t' un objet du domaine de cette fonction. C'est ce que l'on fait en associant à chaque objet un type [6] . Les types sont formés grâce à deux règles :

— Nat, $Bool$, etc. sont des types,

— si T et T' sont deux types, $T \to T'$ est un type (lire *le type des fonctions de* T *dans* T')

On associe à chaque variable un type qu'on indique, pour les variables libres, dans un *contexte de déclarations* et pour les variables muettes là où elles sont liées.

Définition 7. *: Terme du* λ*-calcul simplement typé*

$$T ::= x \mid (T\ T) \mid [x : U]T$$

où U *est un type.*

Par exemple le terme $[x : Nat](+\ (*\ x\ x)\ (S\ (S\ 0)))$ est de type $Nat \to Nat$ dans le contexte $[0 : Nat; S : Nat \to Nat; + : Nat \to Nat \to Nat; * : Nat \to Nat \to Nat]$.

Ces types généralisent la notion d'arité. Remarquons que, à la différence des termes du premier ordre, dans un terme du λ-calcul, il n'est pas nécessaire d'appliquer un symbole de fonction à tous ses arguments pour obtenir un terme bien formé, par exemple les termes $+$, $(+\ 0)$ et $(+\ 0\ 0)$ sont bien formés et leurs types respectifs sont $Nat \to Nat \to Nat$, $Nat \to Nat$ et Nat.

Plus précisément pour éviter de fixer une fois pour toutes les types de base Nat, $Bool$, etc. On décide que tous les types sont eux mêmes des termes et que leur type est un unique type de base Set.

Définition 8. *: Terme du* λ*-calcul simplement typé*

$$T ::= Set \mid T \to T \mid x \mid (T\ T) \mid [x : T]T$$

Le contexte ci-dessus doit maintenant s'écrire :

$$[Nat : Set; 0 : Nat; S : Nat \to Nat; + : Nat \to Nat \to Nat; * : Nat \to Nat \to Nat]$$

Remarquons que si Set est le type de tous les types, il n'est pas lui-même de type Set.

Définition 9. *: λ-calcul Simplement Typé*

Soit Γ un contexte, on note $\Gamma \vdash t : T$ le jugement t a le type T dans le contexte Γ et Γ bien formé le jugement de bonne formation de Γ.
Le contexte vide est bien formé :

$$\frac{}{[\,]\ \textit{bien formé}}$$

Déclaration d'une variable de type :

$$\frac{\Gamma\ \textit{bien formé}}{\Gamma[A : Set]\ \textit{bien formé}}$$

Les variables de type sont des types :

$$\frac{A : Set \in \Gamma}{\Gamma \vdash A : Set}$$

Formation des types fonctionnels :

$$\frac{\Gamma \vdash A : Set \quad \Gamma \vdash B : Set}{\Gamma \vdash A \to B : Set}$$

Déclaration d'une variable :

$$\frac{\Gamma \vdash A : Set}{\Gamma[x : A]\ \textit{bien formé}}$$

Les variables sont des termes :

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

Application :

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : B}$$

Abstraction :

$$\frac{\Gamma \vdash A : Set \quad \Gamma[x : A] \vdash B : Set \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash [x : A]t : A \to B}$$

Remarque : Comme précédemment, on suppose que les variables muettes ont des noms différents. De ce fait, dans les règles de déclaration de variables, on suppose que le nom de la variable déclarée est un nouveau nom, qui n'est pas déjà pris par une autre variable du contexte.

Définition 10. *: Terme Bien Typé*

Un terme t est bit bien typé dans un contexte Γ s'il existe un type T tel que $\Gamma \vdash t : T$.

Proposition 1. *: La $\beta\eta$-réduction est fortement normalisable et confluente, sur les termes typés. C'est-à-dire que si t est un terme bien typé dans un contexte Γ, alors toute réduction issue de ce terme termine sur une forme normale et deux réductions issues d'un même terme terminent sur la même forme normale.*

Démonstration : Voir [63] [61].

1.2.2 Logique d'Ordre Supérieur

La logique d'ordre supérieur [6] est une extension de la logique du premier ordre dans laquelle les termes ne sont plus des arbres mais des λ-termes simplement typés.

En logique d'ordre supérieur, un prédicat tel que $\leq$ ne peut plus comme au premier ordre s'appliquer à n'importe quel terme mais uniquement à des termes de type Nat. On exprime cela en lui associant un type. On pose un nouveau type de base $Prop$, le type des propositions et on ajoute la règle de typage :

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash Prop : Set}$$

Le prédicat $\leq$ est désormais simplement une variable de type $Nat \to Nat \to Prop$.

On ajoute les règles suivantes pour les connecteurs :

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash \to : Prop \to Prop \to Prop}$$

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash \wedge : Prop \to Prop \to Prop}$$

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash \vee : Prop \to Prop \to Prop}$$

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash \neg : Prop \to Prop}$$

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash \bot : Prop}$$

Quand on écrit la proposition $\forall x.(\leq \ 0 \ x)$ on veut pouvoir substituer à x non pas, comme au premier ordre, n'importe quel terme, mais uniquement un terme de type Nat. On écrit donc $\forall_{Nat}([x : Nat] \ (\leq \ 0 \ x))$ (ou de façon équivalente $\forall x : Nat.(\leq \ 0 \ x)$). On se donne donc un quantificateur universel par type et les règles :

$$\frac{\Gamma \vdash T : Set}{\Gamma \vdash \forall_T : (T \to Prop) \to Prop}$$

$$\frac{\Gamma \vdash T : Set}{\Gamma \vdash \exists_T : (T \to Prop) \to Prop}$$

Remarquons que le fait de pouvoir quantifier sur une variable de n'importe quel type autorise la quantification sur les variables fonctionnelles et aussi sur les variables de prédicat.

On considère une notion de contexte étendu où sont déclarés les variables et les axiomes. La règle de déclaration d'un axiome est la suivante :

$$\frac{\Gamma \vdash P : Prop}{\Gamma[P] \text{ bien formé}}$$

Définition 11. *: Logique d'Ordre Supérieur*

Le jugement Γ bien formé et $\Gamma \vdash t : T$ sont définis comme en λ-calcul simplement typé sauf qu'on ajoute les règles ci-dessus pour les propositions. Le jugement $\Gamma \vdash P$ (P est démontrable dans

le contexte Γ*) est défini par les mêmes règles qu'au premier ordre avec une légère modification pour les règles des quantificateurs :*

$$\frac{\Gamma[x:T] \vdash P}{\Gamma \vdash \forall x:T.P} \forall - intro$$

$$\frac{\Gamma \vdash \forall x:T.P \quad \Gamma \vdash t:T}{\Gamma \vdash P[x \leftarrow t]} \forall - elim$$

$$\frac{\Gamma \vdash P[x \leftarrow t] \quad \Gamma \vdash t:T}{\Gamma \vdash \exists x:T.P} \exists - intro$$

$$\frac{\Gamma \vdash \exists x:T.P \quad \Gamma \vdash \forall x:T.(P \to Q) \quad \Gamma \vdash Q:Prop}{\Gamma \vdash Q} \exists - elim$$

Remarque : On prendra soin de ne pas confondre $\Gamma \vdash P : Prop$ qui est le jugement que P est une proposition bien formée indépendamment de tout critère de vérité et $\Gamma \vdash P$ qui est le jugement que P est démontrable sous les axiomes de Γ.

Remarque : En logique d'ordre supérieur, comme au premier ordre, on considère que tous les types sont habités même si on ne peut pas en exhiber un habitant. Cela peut s'exprimer en ajoutant la règle de déduction :

$$\frac{\Gamma \vdash P \qquad \vdash T : Set}{\Gamma \vdash \exists x:T.P}$$

où x n'est pas une variable libre de P.

Remarque : En logique d'ordre supérieur, les symboles de prédicat ne forment plus une catégorie syntaxique à part, mais sont des variables ordinaires. De même les propositions sont simplement les termes de type $Prop$.

Prédicativité - Imprédicativité

En logique du premier ordre, on ne peut pas exprimer le principe de récurrence sur les entiers par un axiome unique, mais on a besoin de poser un nombre infini d'axiomes :

$$(P\ 0) \to \forall x.((P\ x) \to (P\ (S\ x))) \to \forall x.(P\ x)$$

où P décrit l'ensemble des prédicats sur les entiers. A l'ordre supérieur, la possibilité de quantifier sur les prédicats permet de poser un axiome unique :

$$\forall P : Nat \to Prop.((P\ 0) \to \forall x : Nat.((P\ x) \to (P\ (S\ x))) \to \forall x : Nat.(P\ x))$$

Une question essentielle se pose alors : cet axiome est-il lui-même une proposition, ou est-il une méta-proposition. Si on décide que cet axiome R est lui-même une proposition, alors le terme $[x : Nat]R$ a pour type $Nat \to Prop$ et est donc susceptible d'être substitué à P dans R. On voit ici que décréter R de type $Prop$ permet de donner à cette phrase un sens autoréférent.

Un système qui permet l'application d'un schéma d'axiome à lui-même est dit *imprédicatif*, un système qui l'interdit est dit *prédicatif*. La logique d'ordre supérieur telle que nous l'avons présentée est imprédicative. L'imprédicativité est due au fait que les quantificateurs $\forall_T$ et $\exists_T$ ont le type $(T \to Prop) \to Prop$ *y compris quand le symbole* $Prop$ *a une occurrence dans* T.

Un moyen d'exprimer le principe de récurrence tout en restant prédicatif est de décréter une stratification entre propositions du premier niveau auxquelles on est susceptible d'appliquer le principe de récurrence et méta-propositions dont fait partie l'axiome de récurrence.

Quand le symbole $Prop$ a une occurrence dans le type T, alors les symboles $\forall_T$ et $\exists_T$ ont le type $(T \to Prop) \to M\acute{e}taProp$. On a également besoin de quantificateurs $\forall'_T$ et $\exists'_T$ de type $(T \to M\acute{e}taProp) \to M\acute{e}taProp$ pour quantifier dans les méta-propositions.

Le terme $\forall P : Nat \to Prop.((P\ 0) \to \forall x : Nat.((P\ x) \to (P\ (S\ x))) \to \forall x : Nat.(P\ x))$ est alors de type $M\acute{e}taProp$. Remarquons que dans ce système on peut quantifier sur une variable de n'importe quel type, à condition que ce type ne contienne pas d'occurrence du symbole $M\acute{e}taProp$.

Une autre possibilité est d'interdire complètement la quantification sur les variables de prédicats (c'est-à-dire les variables de type T tel que le symbole $Prop$ ait une occurrence dans T) et d'inclure le principe de récurrence dans les règles de déduction en ajoutant la règle :

$$\frac{(P\ 0) \quad \forall x : Nat.((P\ x) \to (P\ (S\ x)))}{\forall x : Nat.(P\ x)}$$

Depuis le paradoxe de Russell (et en fait depuis le paradoxe d'Epinemides) les logiciens ont appris à se méfier des phrases autoréférentes comme source d'incohérences dans leurs systèmes. Pour cette raison, certains logiciens contemporains refusent les systèmes imprédicatifs. D'autres considèrent que l'imprédicativité n'est pas source de paradoxes. Un résultat qui vient étayer leur thèse est le théorème de cohérence de la logique d'ordre supérieur (imprédicative).

Néanmoins, la preuve de ce théorème est ordinairement formulée dans la théorie des ensembles (qui est imprédicative) et elle ne peut pas se formuler dans un système prédicatif. Selon les tenants de la prédicativité ce résultat ne démontre rien car la théorie des ensembles étant elle-même imprédicative, ils n'ont aucune raison de la supposer cohérente. Ces problèmes de fondement n'étant pas directement en rapport avec notre étude, nous nous garderons d'émettre une opinion.

1.2.3 Limites de la Logique d'Ordre Supérieur

Nous allons voir dans ce paragraphe deux limitations de la logique d'ordre supérieur. Ces limitations sont dues à la faible expressivité du langage des termes de cette logique. Par la suite, nous considérerons des extensions du λ-calcul simplement typé et des extensions de la logique d'ordre supérieur qui auront ces calculs comme langage des termes.

Fonctions Représentables en λ-calcul Simplement Typé

Soit le terme $plus\_deux = [x : Nat](S\ (S\ x))$. Le terme $(plus\_deux\ (S\ 0))$ se réduit sur le terme $(S\ (S\ (S\ 0)))$. On peut de même chercher dans le contexte $[Nat : Set; 0 : Nat; S : Nat \to Nat]$ un terme $plus : Nat \to Nat \to Nat$ tel que si a et b sont des entiers, le terme $(plus\ a\ b)$ se réduise sur la somme de a et b. Nous montrerons par la suite qu'un tel terme n'existe pas.

En revanche, on peut ajouter au contexte des variables + et = et considérer les axiomes :

$$\forall y : Nat.(+\ 0\ y) = y \quad \text{et} \quad \forall x : Nat.\forall y : Nat.(+\ (S\ x)\ y) = (S\ (+\ x\ y))$$

En utilisant ces axiomes, on peut montrer la proposition :

$$(+\ (S\ (S\ 0))\ (S\ (S\ 0))) = (S\ (S\ (S\ (S\ 0))))$$

mais le terme $(+\ (S\ (S\ 0))\ (S\ (S\ 0)))$ ne se réduit pas pour autant sur $(S\ (S\ (S\ (S\ 0))))$.

Définition 12. *: Fonction des Entiers dans les Entiers*

On appelle fonction des entiers dans les entiers un terme f de type $Nat \to ... \to Nat \to Nat$. Le terme b est appelé l'image des termes $a_1, ..., a_n$ si on peut démontrer la proposition $b = (f\ a_1\ ...\ a_n)$.

Définition 13. *: Entier*

Un entier est un terme de type Nat qui a la forme $(S...(S\ 0)...)$.

Définition 14. *: Fonction Représentable par un Terme*

Une fonction f est dite représentée par un terme t si pour tout n-uplets d'entiers $< a_1, ..., a_n >$, le terme $(t\ a_1\ ...\ a_n)$ se réduit sur un entier b tel qu'on puisse démontrer la proposition $b = (f\ a_1\ ...\ a_n)$.

Proposition 2. *: En logique d'ordre supérieur, les seules fonctions représentables par un terme sont les fonctions constantes et les fonctions qui ajoutent une constante à l'un de leurs arguments.*

Un corollaire de cette proposition est que la somme et le produit ne sont pas représentables par des termes en logique d'ordre supérieur.

On peut opposer deux façons de démontrer une égalité : *raisonner* en utilisant des axiomes et *calculer* en réduisant deux termes et en constatant que leurs formes normales sont égales. Quand un objet est *déclaré* dans le contexte et ses propriétés *axiomatisées* on ne peut que raisonner pour prouver des théorèmes concernant cet objet, quand il est *défini* comme un terme, certains théorèmes le concernant peuvent être *calculés*. Un système logique est d'autant mieux adapté à la démonstration automatique qu'un maximum de fonctions sont représentables et qu'il permet de ce fait de faire un maximum de définitions et un minimum de déclarations.

La faible expressivité du λ-calcul simplement typé du point de vue de la représentabilité des fonctions est donc une limitation de la logique d'ordre supérieur.

Représentation des Preuves par des Termes

Considérons les propositions construites uniquement avec des variables propositionnelles et l'implication : $\to$. Pour écrire les preuves nous disposons de deux règles : le lien axiome, l'élimination et l'introduction de l'implication.

La sémantique de Heyting (due à Heyting et Kolmogorov) propose d'associer à chaque variable un ensemble : l'ensemble de ses preuves (cet ensemble est vide quand la proposition ne peut être démontrée) et de définir récursivement l'ensemble des preuves de $P \to Q$ comme l'ensemble des fonctions des preuves de P dans les preuves de Q (ici aussi cet ensemble peut être vide). En effet, une preuve de $P \to Q$ est un objet qui permet de construire une preuve de Q dès que l'on a une preuve de P, c'est donc essentiellement une fonction des preuves de P dans les preuves de Q.

Comme nous représentons les fonctions par des λ-termes simplement typés nous associons à chaque variable propositionnelle A un type de base (le *type de ses preuves*) également noté A. Le type des preuves de $P \to Q$ est le type des fonctions de P dans Q, c'est-à-dire le type $P \to Q$. On peut donc identifier types et propositions en identifiant la flèche des types et l'implication.

Quand on déclare un axiome P, on suppose qu'il y a un élément dans le type des preuves de P. Quand on veut prouver un théorème Q on cherche si on peut exhiber, en utilisant les éléments donnés avec les axiomes, un élément du type des preuves de cette proposition.

Par exemple si on se donne les deux axiomes A et $A \to B$ et que l'on veut prouver B, on se donne a dans le type des preuves de A, c'est-à-dire un élément de type A et f dans le type des preuves de $A \to B$, c'est-à-dire une fonction du type des preuves de A dans le type des preuves de

B, c'est-à-dire un élément de type $A \to B$. Le terme $(f\ a)$ est une preuve de B, c'est un élément du type des preuves de B, c'est un élément de type B.

Dans un autre exemple, on ne se donne aucun axiome, la fonction identité $[x : A]x$ qui associe une preuve de A à toute preuve de A est une preuve de $A \to A$.

Proposition 3. *: Soit Γ un contexte contenant des déclarations de variables de type $Prop$ et des axiomes qui sont des propositions formées uniquement avec ces variables propositionnelles et l'implication. Soit Γ' le contexte obtenu en remplaçant dans Γ toutes les déclarations $A : Prop$ par $A : Set$ et les axiomes P par la déclaration d'une variable $x : P$, alors si Q est une proposition démontrable dans Γ, c'est un type habité dans Γ'.*

Démonstration : Par récurrence sur la longueur de la preuve de Q.

Remarque : Réciproquement, chaque terme de type Q dans Γ' peut être traduit en une preuve de Q dans Γ. Preuves et termes peuvent ainsi être identifiés. Cette correspondance est appelée *isomorphisme de Curry-Howard* [13] [36] [3].

Le λ-calcul simplement typé permet donc de représenter par des termes les preuves de la logique propositionnelle réduite à l'implication, il est malheureusement trop peu expressif pour permettre de représenter toutes les preuves de la logique d'ordre supérieur. Nous allons dans le paragraphe suivant considérer des extensions de ce calcul qui permettent de représenter toutes ces preuves. Règles de typage et de déduction feront alors double emploi. Nous pourrons abandonner les règles de déduction. Déclarer un axiome P sera alors simplement déclarer une variable de type P et prouver une proposition P sera simplement donner un terme de type P.

1.3 Systèmes de Types

1.3.1 Quelques Systèmes de Types

Types Dépendants

L'identification entre types et propositions nous amène à identifier les symboles Set et $Prop$. Cette identification nous oblige à abandonner la règle $Prop : Set$, qui devrait alors s'écrire $Prop : Prop$ et qui nous mènerait à un système non normalisable [28]. Le type $Nat \to Nat \to Prop$ n'est donc plus un terme correct dans le λ-calcul simplement typé puisque c'est devenu le type d'une fonction des entiers dans les types. Nous allons donc considérer une extension de ce calcul : le λ-calcul avec types dépendants ($\lambda\Pi$-calcul) [3] [36] [33] qui autorise de telles fonctions.

Les types dépendants peuvent être considérés indépendamment de l'isomorphisme de Curry-Howard. Par exemple, quand on considère le type des listes d'entiers de longueur n, on introduit une fonction $list$ des entiers dans les types, $(list\ 0)$ est le type des listes de longueur 0, $(list\ 1)$ le type des listes de longueur 1, etc.

Nous devons tout d'abord ajouter des règles qui permettent de considérer des termes comme $Nat \to Prop$, $Nat \to Nat \to Prop$, etc. Donner à ces termes le type $Prop$ mène à un système non normalisable [28]. Nous allons donc introduire un nouveau type de base : $Type$ et poser $Prop : Type$, $Nat \to Prop : Type$, etc.

Nous introduisons donc une nouvelle règle de typage pour $Prop$:

$$\frac{\Gamma \text{ bien formé}}{\Gamma \vdash Prop : Type}$$

et une règle qui permet de former les types $Nat \to Prop$, $Nat \to Nat \to Prop$, etc. :

$$\frac{\Gamma \vdash A : Prop \quad \Gamma \vdash B : Type}{\Gamma \vdash A \to B : Type}$$

Nous nous donnons aussi une règle qui permet de déclarer une variable telle que $list$ ou $\leq$ dont le type du type est $Type$ et non $Prop$.

$$\frac{\Gamma \vdash T : Type}{\Gamma[x : T] \text{ bien formé}}$$

Ces règles permettent de considérer le contexte :

$$[Nat : Prop; 0 : Nat; S : Nat \to Nat; \leq : Nat \to Nat \to Prop]$$

Il nous faut encore une règle pour construire les termes dont le type du type est $Type$ et non $Prop$ comme le terme $[x : Nat](\leq \ x \ (S \ (S \ (S \ (S \ (S \ 0))))))) : Nat \to Prop$ qui est un prédicat caractérisant les entiers inférieurs à cinq :

$$\frac{\Gamma \vdash A : Prop \quad \Gamma[x : A] \vdash B : Type \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash [x : A]t : A \to B}$$

Dans un calcul avec types dépendants, le type du résultat d'une fonction peut dépendre de la valeur des arguments de cette fonction. Par exemple, on peut considérer la fonction qui prend en argument un entier n et retourne la liste de longueur n contenant n fois l'entier 0. On doit donc généraliser la notation $f : A \to B$ en une notation $f : (x : A)B$ qui signifie que la fonction f appliquée à un objet a de type A donnera un résultat dont le type est B dans lequel on a substitué a à x. Par exemple, le type de la fonction qui prend en argument un entier n et retourne la liste de longueur n contenant n fois l'entier 0 est $(n : Nat)(list \ n)$. De même le type de la fonction qui ajoute un entier en tête d'une liste est $(n : Nat)(l : (list \ n))(x : Nat)(list(+ \ n \ 1))$. La notation $A \to B$ devient une abréviation pour $(x : A)B$ quand x n'apparaît pas dans B.

Les règles de formation des produits et des abstractions sont donc modifiées en :

$$\frac{\Gamma \vdash A : Prop \quad \Gamma[x : A] \vdash B : Prop}{\Gamma \vdash (x : A)B : Prop}$$

$$\frac{\Gamma \vdash A : Prop \quad \Gamma[x : A] \vdash B : Type}{\Gamma \vdash (x : A)B : Type}$$

$$\frac{\Gamma \vdash A : Prop \quad \Gamma[x : A] \vdash B : Prop \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash [x : A]t : (x : A)B}$$

$$\frac{\Gamma \vdash A : Prop \quad \Gamma[x : A] \vdash B : Type \quad \Gamma[x : A] \vdash t : B}{\Gamma \vdash [x : A]t : (x : A)B}$$

Il nous faut encore ajouter une règle pour que deux termes $\beta\eta$-équivalents aient les mêmes propriétés vis-à-vis du typage. On peut démontrer que si $\Gamma \vdash t : T$ et $t \equiv t'$, alors $\Gamma \vdash t' : T$, en revanche on a besoin de la règle :

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad T \equiv T'}{\Gamma \vdash t : T'} \ s \in \{Prop, Type\}$$

Résumons maintenant le définition de ce calcul.

Définition 15. *: Terme*

$$T \ ::= \ Prop \mid Type \mid x \mid (T\ T) \mid [x:T]T \mid (x:T)T$$

Définition 16. *: $\lambda\Pi$-calcul*

Si t et t' sont deux termes et x une variable, on note $t[x \leftarrow t']$ le terme obtenu en substituant t' à x dans t. On note $t \equiv t'$ le fait que t et t' soient $\beta\eta$-équivalents. Les termes $Prop$ et $Type$ sont appelés les sortes du calcul.

Un contexte est une liste de couples $< x, T >$ (notées $x : T$) où x est une variable et T un terme.

Les règles de typage sont réparties en deux catégories : les règles générales qui sont une reformulation des règles du λ-calcul simplement typé et les règles spécifiques au λ-calcul avec types dépendants. Règles générales :

$$\frac{}{[\,]\ \textit{bien formé}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x:T]\ \textit{bien formé}}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma\ \textit{bien formé}}{\Gamma \vdash Prop : Type}$$

$$\frac{\Gamma\ \textit{bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : Prop \quad \Gamma[x:T] \vdash T' : Prop}{\Gamma \vdash (x:T)T' : Prop}$$

$$\frac{\Gamma \vdash (x:T)T' : Prop \quad \Gamma[x:T] \vdash t : T'}{\Gamma \vdash [x:T]t : (x:T)T'}$$

$$\frac{\Gamma \vdash t : (x:T)T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad T \equiv T'}{\Gamma \vdash t : T'}\ s \in \{Prop, Type\}$$

Règles spécifiques au λ-calcul avec types dépendants :

$$\frac{\Gamma \vdash T : Prop \quad \Gamma[x:T] \vdash T' : Type}{\Gamma \vdash (x:T)T' : Type}$$

$$\frac{\Gamma \vdash T : Prop \quad \Gamma[x:T] \vdash T' : Type \quad \Gamma[x:T] \vdash t : T'}{\Gamma \vdash [x:T]t : (x:T)T'}$$

Dans le $\lambda\Pi$-calcul on peut représenter les preuves de la logique d'ordre supérieur restreinte à l'implication et à la quantification universelle limitée aux variables dont le type du type est $Prop$ et non $Type$.

En effet la sémantique de Heyting suggère de représenter une preuve de $\forall x : T.P$ comme une fonction qui associe à tout élément de a une preuve de $P[x \leftarrow a]$, c'est-à-dire comme un objet de type $(x : T)P$. En identifiant la notation $\forall x : T.P$ et $(x : T)P$ on étend l'isomorphisme entre propositions et types. On montre que toute preuve de P écrite en déduction naturelle peut se traduire en un λ-terme de type P et que réciproquement tout terme de type P peut s'interpréter comme une preuve de P.

Polymorphisme

Dans le $\lambda\Pi$-calcul, le terme $(x : T)T'$ est bien typé si T est de type $Prop$ et T' de type $Prop$ ou $Type$. Pour pouvoir quantifier sur les prédicats dans les propositions et représenter les preuves de ces propositions par des λ-termes, il faut permettre la formation de types $(x : T)T'$ où T' est de type $Prop$ et T de type $Type$ et la formation de termes $[x : T]t$ où t est de type T', lui-même de type $Prop$ et T de type $Type$. Un tel calcul est dit polymorphe.

Définition 17. *: λ-calcul Polymorphe*

Le λ-calcul polymorphe [28] est obtenu en ajoutant aux règles de base, les règles spécifiques au λ-calcul polymorphe :

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x : T] \vdash T' : Prop}{\Gamma \vdash (x : T)T' : Prop}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x : T] \vdash T' : Prop \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'}$$

Par exemple, le terme $[P : Prop][x : P]x$ est une preuve de $(P : Prop)(P \to P)$. Qui se lit : "Pour toute proposition P, P implique P". Cette phrase est de type $Prop$ (imprédicativité), elle affirme donc en particulier qu'elle s'implique elle-même.

Naturellement, en considérant les règles de base, celles spécifiques au λ-calcul avec types dépendants et au λ-calcul polymorphe, on obtient le *λ-calcul polymorphe avec types dépendants.*

Types Inductifs

Soit R l'axiome de récurrence sur les entiers :

$$R : \forall P : Nat \to Prop.((P\ 0) \to \forall x : Nat.((P\ x) \to (P\ (S\ x))) \to \forall x : Nat.(P\ x))$$

Quand on a un prédicat P et les preuves $a : (P\ 0)$, $b : \forall x : Nat.((P\ x) \to (P\ (S\ x)))$, on peut en déduire $(R\ P\ a\ b) : \forall x : Nat.(P\ x)$ puis appliquer cette proposition à un entier n pour obtenir une preuve de la proposition $(P\ n)$. Cette proposition a aussi une autre preuve beaucoup plus concrète qui consiste à appliquer une fois l'hypothèse a puis n fois l'hypothèse b pour prouver successivement $(P\ 0)$, $(P\ 1)$, $(P\ 2)$, ..., $(P\ n)$. Par exemple, le terme $(b\ 4\ (b\ 3\ (b\ 2\ (b\ 1\ (b\ 0\ a)))))$ est une preuve de $(P\ 5)$.

Le λ-calcul avec types inductifs permet d'identifier ces deux preuves de la même proposition.

Définition 18. *: Types Inductifs*

Le λ-calcul avec types inductifs [30] [50] est obtenu en ajoutant au λ-calcul polymorphe les règles de réduction :

- $(R\ P\ a\ b\ 0) \rhd a$,
- $(R\ P\ a\ b\ (S\ x)) \rhd (b\ x\ (R\ P\ a\ b\ x))$

et des règles similaires pour les types autres que Nat.

Remarque : Dans un système prédicatif où la quantification sur les prédicats est interdite et le principe de récurrence traduit par une règle :

$$\frac{(P\ 0) \quad \forall x : Nat.((P\ x) \to (P\ (S\ x)))}{\forall x : Nat.(P\ x)}$$

on peut représenter les preuves par des termes en ajoutant un symbole primitif R et une règle de typage :

$$\frac{\Gamma \vdash a : (P\ 0) \quad \Gamma \vdash b : \forall x : Nat.((P\ x) \to (P\ (S\ x)))}{\Gamma \vdash (R\ a\ b) : \forall x : Nat.(P\ x)}$$

Un tel système (même s'il n'est pas polymorphe) peut s'étendre par l'adjonction de types inductifs en ajoutant les règles de réduction :

- $(R\ a\ b\ 0) \rhd a$,
- $(R\ a\ b\ (S\ x)) \rhd (b\ x\ (R\ a\ b\ x))$.

Constructeurs de Types

Un prédicat peut être vu comme un ensemble : l'ensemble des valeurs qui vérifient ce prédicat. Quantifier sur les prédicats revient donc à quantifier sur les ensembles. La possibilité suivante, qui est donnée en logique d'ordre supérieur, est de quantifier sur les ensembles d'ensembles, c'est-à-dire quantifier sur les objets du type $(Nat \to Prop) \to Prop$ par exemple. Cela demande simplement de donner à ces objets le type $Type$. Si T et T' sont de type $Type$ on décide donc que $(x : T)T'$ est aussi de type $Type$. De même, pour former des objets de ce type, il faut autoriser les abstractions correspondantes. Un tel calcul est dit admettre des constructeurs de types [28].

Définition 19. *: Constructeurs de Types*
Le λ-calcul avec constructeurs de types [28] est obtenu en ajoutant aux règles de base les règles :

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x : T] \vdash T' : Type}{\Gamma \vdash (x : T)T' : Type}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x : T] \vdash T' : Type \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'}$$

1.3.2 Classification des Systèmes de Types

Cube de Barendregt

L'extension du λ-calcul simplement typé avec types dépendants, types polymorphes et constructeurs de types est appelée le *Calcul des Constructions* [8] [11].

Les règles de ce calcul peuvent être ainsi agrégées :

Définition 20. *: Calcul des Constructions*

$$\frac{}{[\,]\ \textit{bien formé}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \textit{bien formé}}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma\ \textit{bien formé}}{\Gamma \vdash Prop : Type}$$

$$\frac{\Gamma\ \textit{bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma[x:T] \vdash T' : s'}{\Gamma \vdash (x:T)T' : s'} < s, s' > \in R$$

$$\frac{\Gamma \vdash (x:T)T' : s \quad \Gamma[x:T] \vdash t : T'}{\Gamma \vdash [x:T]t : (x:T)T'} \, s \in \{Prop, Type\}$$

$$\frac{\Gamma \vdash t : (x:T)T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'} \, s \in \{Prop, Type\}$$

où $R = \{Prop, Type\}^2$.

Dans la règle de formation des produits, le couple $< s, s' >=< Prop, Type >$ permet la formation de types dépendants, le couple $< s, s' >=< Type, Prop >$ permet la formation de types polymorphes, le couple $< s, s' >=< Type, Type >$ permet la formation de constructeurs de types.

En prenant pour R non plus l'ensemble $\{Prop, Type\}^2$ mais uniquement une partie de cet ensemble on choisit d'inclure ou non chacune de ces trois propriétés au calcul que l'on définit. Ces trois choix binaires déterminent huit systèmes appelés les systèmes du cube de Barendregt [1]. Chacun de ces systèmes peut s'étendre en ajoutant des types inductifs.

Types Dépendants	Polymorphisme	Constructeurs de Types	Types Inductifs	
N	N	N	N	Simplement Typé (Church [6])
O	N	N	N	$\lambda\Pi$ (de Bruijn [3] / Howard [36] / Harper-Honsel-Plotkin [33])
N	O	N	N	F (Girard [28])
O	O	N	N	$\lambda P2$
N	O	O	N	F_ω (Girard [28])
O	O	O	N	CoC(Coquand-Huet [8] [11])
N	N	N	O	T(Gödel [30])
O	N	N	O	ITT (Martin-Löf [50])
N	O	N	O	F^{ind}
O	O	N	O	$\lambda P2^{ind}$
N	O	O	O	F_ω^{ind}
O	O	O	O	CIC(Coquand-Paulin [12])

Sur ces seize systèmes nous n'en avons présentés que douze, les constructeurs de types sans les types polymorphes menant à des systèmes de peu d'intérêt.

Systèmes de Types Purs

Nous avons vu que des restrictions du Calcul des Constructions pouvaient être obtenues en restreignant la règle de formation des produits. On peut donc paramétrer un système de types par un ensemble de règles pour obtenir les huit systèmes du cube de Barendregt. On peut également le

paramétrer par l'ensemble des sortes considéré et par les règles telles que $Prop : Type$. Cette idée est due à Geuvers, Berardi et Terlouw.

Un système de types est donc défini par un ensemble S de sortes, un ensemble Ax de couples de sortes (appelés *axiomes*), un ensemble R de triplets de sortes (appelés *règles*).

Définition 21. *: Terme*

$$T \ ::= \ s \mid x \mid (T\ T) \mid [x : T]T \mid (x : T)T$$

Définition 22. *: Règles de Typage*

$$\frac{}{[\,]\ \textit{bien formé}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \textit{bien formé}}\, s \in S$$

$$\frac{\Gamma\ \textit{bien formé}}{\Gamma \vdash s : s'} < s, s' > \in Ax$$

$$\frac{\Gamma\ \textit{bien formé} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma[x : T] \vdash T' : s'}{\Gamma \vdash (x : T)T' : s''} < s, s', s'' > \in R$$

$$\frac{\Gamma \vdash (x : T)T' : s \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'}\, s \in S$$

$$\frac{\Gamma \vdash t : (x : T)T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'}\, s \in S$$

Exemples de Systèmes de Types Purs

Dans le Calcul des Constructions le terme $Type$ n'est pas bien typé. On peut étendre ce système en un système *le Calcul des Constructions avec Univers* où tout terme type d'un terme est bien typé.

Définition 23. *: Calcul des Constructions avec Univers*

On se donne un nombre infini de sortes : $Prop, Type_0 (= Type), Type_1, Type_2, ...,$ *les axiomes* $Prop : Type_0$ *et* $Type_i : Type_{i+1}$ *et les règles :*

$$< Prop, Prop, Prop >$$

$$< Prop, Type_i, Type_i >$$

$$< Type_i, Prop, Prop >$$

$$< Type_i, Type_j, Type_{max\{i,j\}} >$$

Remarque : La règle $< Type_i, Prop, Prop >$ donne à ce système un caractère imprédicatif "au niveau $Prop$", mais la règle $< Type_i, Type_j, Type_{max\{i,j\}} >$ lui donne un caractère prédicatif "au niveau $Type$". En effet, si P est de type $Type_j$, $(x : Type_i)P$ est de type $Type_{max\{i+1,j\}}$ donc n'est pas de type $Type_i$. Un système imprédicatif à deux niveaux (comme le système avec la règle $< Type_i, Type_j, Type_j >$) n'est pas normalisable [28].

Enfin ce système peut être étendu en ajoutant une règle de cumulativité qui permet d'inclure le type $Type_i$ dans le type $Type_{i+1}$, voir [9] [49].

Un deuxième exemple de système de types pur est le système $Méta$ que nous utiliserons dans l'expression de nos algorithmes de synthèse de preuves et de résolution d'équations.

Définition 24. *: Méta*

$$Méta =< S', Ax', R' >$$

$$S' = \{Prop, Type, Extern\}$$

$$Ax' = \{< Prop, Type >, < Type, Extern >\}$$

$$R' = \{< Prop, Prop, Prop >, < Prop, Type, Type >, < Type, Prop, Prop >,$$

$$< Type, Type, Type >, < Prop, Extern, Extern >, < Type, Extern, Extern >\}$$

Remarque : Ce système est un sous-système du Calcul des Constructions avec Univers, dans lequel la sorte $Type_1$ est notée $Extern$.

Remarque : L'intérêt de ce système vient du fait que si $\mathcal{T}$ un système de types du Cube, alors pour tout terme t bien typé dans le système $\mathcal{T}$, il est possible de déclarer dans le système $Méta$ une variable qui a le type de t (par exemple avec $t = Prop$, il est possible de déclarer dans le système $Méta$ une variable de type $Type$). De plus, si $t : T'$ est bien typé dans le système $Méta$ dans le contexte $\Gamma[x : T]$ et $T' \neq Extern$, alors $f = [x : T]t$ est également bien typé.

1.3.3 Propriétés des Systèmes de Types Purs

Définition 25. *: Système de Types Fonctionnel*

Un système de types est dit fonctionnel si :

$$< s, s' >\in Ax \text{ et } < s, s'' >\in Ax \text{ implique } s' = s''$$

$$< s, s', s'' >\in R \text{ et } < s, s', s''' >\in R \text{ implique } s'' = s'''$$

Proposition 4. *: Les Calculs du Cube, le système Méta et le Calcul des Constructions avec Univers sont fonctionnels.*

Définition 26. *: Terme Bien Typé*

Un terme t est dit bien typé dans un contexte Γ s'il existe un terme T tel que $\Gamma \vdash t : T$.

Proposition 5. *: Dans un système de types fonctionnel un terme t bien typé dans un contexte Γ a un type unique modulo $\beta\eta$-équivalence.*

Démonstration : Par récurrence sur la structure de t.

Théorème 1. *: Normalisation Forte et Confluence de la Réduction*
Tout terme t bien typé dans un contexte Γ*, dans un calcul du cube, dans le système Méta ou dans le Calcul des Constructions avec Univers a une forme normale unique.*

Démonstration : Voir [9] [60] [25].

Remarque : Le théorème précédent ne s'applique qu'aux calculs du cube, au système *Méta* et au Calcul des Constructions avec Univers et non à tous les systèmes de types purs. Par exemple, dans le système avec une unique sorte $Prop$, l'axiome $Prop : Prop$ et la règle $< Prop, Prop, Prop >$, on peut typer des termes non normalisables [28].

Définition 27. *: Type*
Un terme T *est dit être un type dans un contexte* Γ *s'il existe une sorte* s *telle que* $\Gamma \vdash T : s$.

Proposition 6. *: Si* $\Gamma \vdash t : T$*, alors* T *est un type ou une sorte.*

Démonstration : Par récurrence sur la longueur de la dérivation de $\Gamma \vdash t : T$.

Définition 28. *: Terme Atomique*
Un terme t est dit atomique s'il est de la forme $(x\ c_1\ ...\ c_n)$ *où* x *est une variable ou une sorte. Le symbole* x *est appelé la tête du terme t.*

Proposition 7. *: Soit t un terme normal et bien typé, t est une abstraction, un produit ou un terme atomique.*

Démonstration : Si le terme t n'est ni une abstraction ni un produit, alors il peut être écrit d'une façon unique :

$$t = (u\ c_1\ ...\ c_n)$$

où u n'est pas une application.

Le terme u n'est pas un produit (si $n \neq 0$ parce qu'un produit est de type s pour une sorte s et ne peut donc pas être appliqué et si $n = 0$ parce que t n'est pas un produit). Ce n'est pas une abstraction (si $n \neq 0$ parce que t est en forme normale et si $n = 0$ parce que t n'est pas une abstraction). C'est donc une variable ou une sorte.

Proposition 8. *: Soit* T *un type normal,* T *s'écrit de manière unique* $T = (x_1 : P_1)...(x_n : P_n)P$ *avec* P *atomique.*

Démonstration : Par récurrence sur la structure de T.

Définition 29. *: Forme* η*-longue*
Soit Γ *un contexte et t un terme* $\beta\eta$*-normal bien typé dans* Γ*. La forme* η*-longue de ce terme est définie de la façon suivante :*

- *Si* $t = [x : U]u$*, alors soit* U' *la forme* η*-longue de* U *dans* Γ *et* u' *celle de* u *dans* $\Gamma[x : U]$*, on définit la forme* η*-longue de t comme le terme* $[x : U']u'$*,*
- *Si* $t = (x : U)V$*, alors soit* U' *la forme* η*-longue de* U *dans* Γ *et* V' *celle de* V *dans* $\Gamma[x : U]$*, on définit la forme* η*-longue de t comme le terme* $(x : U')V'$*,*

— *Si* $t = (w\ c_1\ ...\ c_p)$*, alors soit* $T = (x_1 : P_1)...(x_n : P_n)P$ *(*P *atomique) la forme normale du type de* t*. Soit* c'_i *la forme* η*-longue de* c_i *dans* Γ*,* P'_i *celle de* P_i *dans* $\Gamma[x_1 : P_1; ...; x_{i-1} : P_{i-1}]$ *et* x'_i *celle de* x_i *dans* $\Gamma[x_1 : P_1; ...; x_i : P_i]$*. On définit la forme* η*-longue de* t *comme le terme* $[x_1 : P'_1]...[x_n : P'_n](w\ c'_1\ ...\ c'_p\ x'_1\ ...\ x'_n)$*.*

La bonne fondation de cette définition est prouvée au chapitre suivant.

De façon évidente, la forme η-longue d'un terme $\beta\eta$-normal est β-normale, ce qui motive la définition :

Définition 30. *: Forme* β*-normale* η*-longue*

Soit t *un terme bien typé dans un contexte* Γ*, la forme* β*-normale* η*-longue de* t *est la forme* η*-longue de sa forme* $\beta\eta$*-normale.*

Par la suite, tous les termes seront supposés en forme β-normale η-longue.

Fonctions Représentables

Nous allons maintenant voir que ces extensions du λ-calcul simplement typé qui ont été motivées par des considérations de représentabilité des preuves sont également plus riches du point de vue de la représentabilité des fonctions.

Dans le λ-calcul simplement typé, les seules fonctions des entiers dans les entiers représentables sont les fonctions constantes et celles qui ajoutent une constante à l'un de leurs arguments. Cet ensemble de fonctions peut être défini comme le plus petit ensemble qui contient les projections, la fonction nulle, la fonction successeur et qui est clos par composition. Une construction manque pour obtenir les fonctions récursives primitives : la définition par récurrence.

Une première tentative pour obtenir cette construction est de définir les entiers non plus avec deux variables pour zéro et le successeur, mais comme des itérateurs : les entiers de Church. On se donne un type T de base, on définit l'entier n comme le terme :

$$n = [x : T][f : T \to T](f\ ...\ (f\ x)\ ...\ )\ (n \text{ fois})$$

Les entiers sont alors des termes de type $T \to (T \to T) \to T$. Quand on a une fonction g et un entier n le terme $[x : T](n\ x\ g)$ est la fonction g itérée n fois $g \circ g \circ ... \circ g$. Malheureusement cette possibilité d'itération est très restreinte : il n'est possible d'itérer que des fonctions de type $T \to T$ et pas celles de type $Nat \to Nat$ et, de ce fait, il n'est pas possible de représenter toutes les fonctions primitives récursives par un λ-terme simplement typé :

Proposition 9. *: Les fonctions sur les entiers de Church représentables comme des termes dans le* λ*-calcul simplement typé sont les polynômes étendus par le test à zéro.*

Démonstration : Voir [62].

En revanche dans les systèmes polymorphes, on peut définir l'entier n comme le terme :

$$n = [T : Prop][x : T][f : T \to T](f\ ...\ (f\ x)\ ...)\ (n \text{ fois})$$

et ainsi itérer des fonctions de type $T \to T$ pour n'importe quel type T, en particulier le type Nat lui-même et les types où Nat a une occurrence (imprédicativité).

Proposition 10. *: Les fonctions récursives primitives sont représentables par des termes dans le λ-calcul polymorphe, et également les fonctions récursives primitives avec paramètres fonctionnels comme la fonction d'Ackermann.*

Démonstration : Voir [28] [29].

Un résultat plus fort est le suivant :

Proposition 11. *: Les fonctions représentables dans le λ-calcul polymorphe sont les fonctions de l'arithmétique du deuxième ordre. Les fonctions représentables dans le λ-calcul polymorphe avec constructeurs de types sont les fonctions de l'arithmétique d'ordre supérieur.*

Démonstration : Voir [28].

Dans les systèmes avec types inductifs, en revanche, il n'est plus nécessaire de définir les entiers comme des itérateurs pour pouvoir représenter les fonctions récursives primitives. En effet, si a est un terme de type Nat et b un terme de type $Nat \to Nat \to Nat$, en posant $P = [x : Nat]Nat$ et $f = (R\ P\ a\ b)$, le terme $(f\ 0)$ se réduit sur a et $(f\ (S\ x))$ sur $(b\ x\ (f\ x))$ (les cas de récursion avec paramètres sont très similaires). Ainsi :

Proposition 12. *: Les fonctions primitives récursives sont représentables par des termes dans le λ-calcul avec type inductif, et également les fonctions récursives primitives avec paramètres fonctionnels.*

Démonstration : Voir [30] [29].

Remarque : Si on s'intéresse à présent non plus aux fonctions représentables mais aux *algorithmes* représentables, on s'aperçoit que le formalisme avec types inductifs est plus riche que le λ-calcul polymorphe.

1.3.4 Une Extension de la Logique d'Ordre Supérieur : Le Calcul des Constructions

Si on étend la logique d'ordre supérieur restreinte à l'implication et à la quantification universelle, en prenant comme langage des objets non plus le λ-calcul simplement typé mais le Calcul des Constructions, on obtient un système où toute preuve d'une proposition P peut être traduite en un terme de type P et vice versa. Règles de typage et de déduction font double emploi, on peut donc abandonner les règles de déduction. Déclarer un axiome P est simplement déclarer une variable de type P et prouver une proposition P est simplement donner un terme de type P. Le système logique se réduit donc ainsi à un λ-calcul typé.

Connecteurs

Jusqu'à maintenant nous nous sommes restreints à ne considérer qu'un connecteur et un quantificateur : l'implication et le quantificateur universel. Cela vient du fait qu'une fois que l'on dispose de ce connecteur et de ce quantificateur on peut les utiliser pour coder les règles de déduction concernant les autres connecteurs et quantificateurs comme des propositions [28]. En effet, ces règles permettent de déduire des propositions d'autres propositions, ce sont donc essentiellement des implications. Par

ailleurs elles contiennent des variables libres qui sont implicitement universellement quantifiées. Par exemple, on peut déclarer la conjonction par :

$$and : Prop \to Prop \to Prop$$

et poser les règles de déduction comme des axiomes exprimés avec l'implication et la quantification universelle :

$$and\_intro : (A : Prop)(B : Prop)(A \to B \to (and\ A\ B))$$

$$and\_elim1 : (A : Prop)(B : Prop)((and\ A\ B) \to A)$$

$$and\_elim2 : (A : Prop)(B : Prop)((and\ A\ B) \to B)$$

En fait, il est même possible de définir le symbole *and* et de prouver ces règles.

Définition 31. *: Connecteurs*
On pose :

$$and = [A : Prop][B : Prop](C : Prop)((A \to B \to C) \to C)$$

$$and\_intro = [A : Prop][B : Prop][x : A][y : B][C : Prop][u : A \to B \to C](u\ x\ y)$$

$$and\_elim1 = [A : Prop][B : Prop][u : (and\ A\ B)](u\ A\ [x : A][y : B]x)$$

$$and\_elim2 = [A : Prop][B : Prop][u : (and\ A\ B)](u\ B\ [x : A][y : B]y)$$

$$or = [A : Prop][B : Prop](C : Prop)((A \to C) \to (B \to C) \to C)$$

$$or\_intro1 = [A : Prop][B : Prop][x : A][C : Prop][u : A \to C][v : B \to C](u\ x)$$

$$or\_intro2 = [A : Prop][B : Prop][x : B][C : Prop][u : A \to C][v : B \to C](v\ x)$$

$$or\_elim = [A : Prop][B : Prop][C : Prop][x : (or\ A\ B)][u : (A \to C)][v : B \to C](x\ C\ u\ v)$$

$$bot = (C : Prop)C$$

$$bot\_elim = [C : Prop][u : bot](u\ C)$$

$$ex = [T : Prop][P : T \to Prop](C : Prop)(((x : T)(P\ x) \to C) \to C)$$

$$ex\_intro = [T : Prop][P : T \to Prop][t : T][u : (P\ t)][C : Prop][v : ((x : T)(Px) \to C)](v\ t\ u)$$

$$ex\_elim = [T : Prop][P : T \to Prop][C : Prop][u : (exTP)][v : ((x : T)(Px) \to C)](u\ C\ v)$$

Proposition 13. *:*
On a :

$$and\_intro : (A : Prop)(B : Prop)(A \to B \to (and\ A\ B))$$

$$and\_elim1 : (A : Prop)(B : Prop)((and\ A\ B) \to A)$$

$$and\_elim2 : (A : Prop)(B : Prop)((and\ A\ B) \to B)$$

$$or\_intro1 : (A : Prop)(B : Prop)(A \to (or\ A\ B))$$

$$or\_intro2 : (A : Prop)(B : Prop)(B \to (or\ A\ B))$$

$$or\_elim : (A : Prop)(B : Prop)(C : Prop)((or\ A\ B) \to (A \to C) \to (B \to C) \to C)$$

$$bot\_elim : (C : Prop)(bot \to C)$$

$$ex\_intro : (T : Prop)(P : T \to Prop)(t : T)((P\ t) \to (ex\ T\ P))$$

$$ex\_elim : (T : Prop)(P : T \to Prop)(C : Prop)((ex\ T\ P) \to ((x : T)(P\ x) \to C) \to C)$$

Remarque : Pour la négation, on pose $(\neg A) = A \to \bot$, les règles $\neg$-intro et $\neg$-elim sont des cas particuliers de $\to$-intro et $\to$-elim.

Cohérence

Proposition 14. *: Dans le contexte vide, le type $\bot = (P : Prop)P$ est vide.*

Démonstration : S'il existait un terme de type $(P : Prop)P$, il en existerait aussi un normal, donc un terme de type P dans le contexte $[P : Prop]$. D'après la proposition 7, ce terme est une abstraction, un produit ou un terme atomique. Ce ne peut être une abstraction ou un produit pour des raisons évidentes de type, c'est donc un terme atomique. Sa tête ne peut être ni le symbole $Prop$ ni $Type$ ni P pour des raisons évidentes de type, d'où contradiction.

Un corollaire est qu'on ne peut pas démontrer à la fois P et $\neg P$ dans le contexte vide. Cette propriété s'appelle la *cohérence* du système logique.

Remarque : L'apparente simplicité de cette preuve de cohérence ne doit pas masquer le fait qu'elle s'appuie sur le théorème de normalisation du calcul qui est, quant à lui, difficile.

Représentabilité des Fonctions

Le Calcul des Constructions, comme le λ-calcul polymorphe avec constructeurs de types, permet de représenter les fonctions de l'arithmétique d'ordre supérieur. Mais comme ces fonctions sont également celles du Calcul des Constructions, on obtient une relation très étroite entre les aspects logique et calculatoire de ce système :

Théorème 2. *: Principe de Compréhension*

Pour tout prédicat $P : Nat \to ... \to Nat \to Prop$ *tel qu'on puisse prouver la proposition* $\forall x_1 : Nat....\forall x_n : Nat.\exists y : Nat.(P\ x_1\ ...\ x_n\ y)$*, dans le Calcul des Constructions et sous les axiomes de l'arithmétique, il existe un terme* f *tel que pour tout n-uplets d'entiers* $< a_1, ..., a_n >$*, le terme* $(f\ a_1\ ...\ a_n)$ *se réduise sur un entier* b *tel que* $(P\ a_1\ ...\ a_n\ b)$*.*

Quelques Variantes

Nous avons vu précédemment que l'isomorphisme de Curry-Howard nous poussait à identifier $Prop$ et Set et donc à abandonner la règle $Prop : Set$. Une autre possibilité consiste à ne pas appliquer l'isomorphisme de Curry-Howard à la lettre et à identifier Set et $Type$. Ainsi, on peut garder la règle $Prop : Set$. Dans ce cas, on a besoin de donner un type au terme $Type$ $(= Set)$ pour pouvoir déclarer des variables de type Set. On se place donc dans un sous-système du Calcul des Constructions avec Univers, qui contient au moins les sortes $Prop$, $Type$ et $Type_1$. Dans ce système le langage des preuves est imprédicatif, mais celui des termes prédicatif.

Enfin une autre variante consiste à garder les symboles $Prop$, Set et $Type$ avec les règles $Prop : Type$ et $Set : Type$. Dans ce cas, preuves et termes se correspondent par isomorphisme mais ne sont pas identifiés. Dans ce système, aussi bien le langage des preuves que le langage des termes est imprédicatif.

Ces systèmes sont comparés dans [24].

Chapitre 2

Bonne Fondation de la Récursion Simultanée sur le Type et les Sous-Termes Stricts d'un Terme

Nous nous plaçons dans ce chapitre dans un quelconque sous-système du Calcul des Constructions avec Univers. Par exemple un système du cube, le système *Méta* ou le Calcul des Constructions avec Univers lui-même.

Considérons un programme qui prend en argument un terme normal t et s'appelle lui-même sur un sous-terme strict de t. Ce programme termine de façon évidente. Considérons de même un programme qui prend en argument un terme normal t et s'appelle lui-même sur la forme normale du type de t (si t n'est pas une sorte). Ce programme termine également car si t n'est pas une sorte, son type est une sorte ou le type de son type en est une. Si maintenant le programme s'appelle lui-même soit sur un sous-terme strict de t soit sur le type de t, il n'est plus évident qu'il termine. En effet, du fait de la présence de types dépendants, un sous-terme d'un type n'est pas nécessairement un type. Par exemple en partant de la liste vide [ ] (de type $(list\ 0)$), on engendre la suite de termes [ ], $(list\ 0)$, 0, Nat, $Prop$. Cette suite est finie. Nous allons montrer dans ce chapitre que c'est toujours le cas, c'est-à-dire que la récursion simultanée sur le type et les sous-termes stricts d'un terme est bien fondée.

Ce résultat de bonne fondation sera utilisé par la suite pour prouver la complétude d'une méthode de démonstration automatique (chapitre 4). En effet, pour synthétiser un terme t de type T, nous aurons comme tâche intermédiaire à synthétiser des sous-termes de t et aussi les types de certains de ces sous-termes. La complétude de cette méthode repose sur le fait que ces tâches intermédiaires ne forment pas une régression infinie, mais sont ordonnées de façon bien fondée. De même un algorithme de résolution d'équation (au chapitre 7) demandera, pour résoudre une équation $a = b$, de résoudre d'abord des équations entre des sous-termes de a et b et également entre les types de certains de ces sous-termes. La complétude et la terminaison de cet algorithme reposent encore sur le fait que ces équations sont ordonnées de façon bien fondée.

Définition 32. *: Sous-Terme*

On considère des termes bien typés normaux indicés par le contexte dans lequel ils sont bien typés : t_Γ.

Soit t_Γ *un tel terme, on définit par récurrence sur la structure de* t_Γ *l'ensemble des sous-termes*

stricts de t_Γ *:*

— *si* t *est une sorte ou une variable, alors* $Sub(t_\Gamma) = \{\}$,
— *si* $t = (u\ v)$, *alors* $Sub(t_\Gamma) = \{u_\Gamma, v_\Gamma\} \cup Sub(u_\Gamma) \cup Sub(v_\Gamma)$,
— *si* $t = [x : P]u$, *alors* $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$,
— *si* $t = (x : P)u$, *alors* $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$.

Définition 33. *: La Relation* $<$

Soit $<$ *la plus petite relation transitive définie sur les termes normaux telle que :*

— *si* t_Γ *est un sous-terme strict de* t'_Δ, *alors* $t_\Gamma < t'_\Delta$,
— *si* T_Γ *est la forme* β*-normale* η*-longue du type de* t_Γ *dans* Γ *et le terme* t *n'est pas une sorte, alors* $T_\Gamma < t_\Gamma$.

Nous allons montrer dans ce chapitre que la relation $<$ est bien fondée.

2.1 Termes Marqués

On définit une syntaxe pour les systèmes de types dans laquelle chaque terme est marqué par son type. En fait tous les termes sont marqués sauf les sortes.

Définition 34. *: Termes Marqués*

$$T \ ::= \ s \mid x^T \mid (T\ T)^T \mid ([x : T]T)^T \mid ((x : T)T)^T$$

Soit t *et* u *deux termes marqués et* x *une variable. On note* $t[x \leftarrow u]$ *le terme obtenu en substituant* u *à* x *dans* t. *Remarquons que comme la variables* x *peut apparaître dans le terme même et dans les marques, la substitution doit être effectuée aussi bien dans le terme lui-même que dans les marques.*

On note $t \equiv u$ *la relation de* $\beta\eta$*-équivalence entre les termes* t *et* u. *Ici aussi, des conversions peuvent être effectuées aussi bien dans le terme que dans les marques.*

Un contexte marqué est une liste de couples $< x, T >$ *(notées* $x : T$*) où* x *est une variable et* T *un terme marqué.*

Définition 35. *: Règles de Typage*

On définit inductivement deux jugements : Γ *est bien formé et* t *a le type* T *dans* Γ *(*$\Gamma \vdash t : T$*) où* Γ *est un contexte marqué et* t *et* T *deux termes marqués.*

$$\frac{}{[\,]\ \textit{bien formé}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \textit{bien formé}}\, s \in S$$

$$\frac{\Gamma\ \textit{bien formé}}{\Gamma \vdash s : s'} < s, s' > \in Ax$$

$$\frac{\Gamma\ \textit{bien formé}\ \ x : T \in \Gamma}{\Gamma \vdash x^T : T}$$

$$\frac{\Gamma \vdash T : s\ \ \Gamma[x : T] \vdash U : s'}{\Gamma \vdash ((x : T)U)^{s''} : s''} < s, s', s'' > \in R$$

$$\frac{\Gamma \vdash ((x:T)U)^s : s \quad \Gamma[x:T] \vdash t : U}{\Gamma \vdash ([x:T]t)^{((x:T)U)^s} : ((x:T)U)^s}\, s \in S$$

$$\frac{\Gamma \vdash t : ((x:T)U)^s \quad \Gamma \vdash u : T}{\Gamma \vdash (t\ u)^{U[x \leftarrow u]} : U[x \leftarrow u]}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T \equiv U}{\Gamma \vdash t : U}\, s \in S$$

Proposition 15. *: Soit Γ un contexte et t et T deux termes tels que t ne soit pas une sorte et $\Gamma \vdash t : T$. Le terme T est alors équivalent à la marque la plus externe de t. Par exemple, si $t = (u\ v)^U$, alors T est équivalent à U.*

Démonstration : Par récurrence sur la longueur de la dérivation de $\Gamma \vdash t : T$.

Définition 36. *: Terme Marqué Bien Typé*
Un terme t est dit bien typé dans un contexte Γ s'il existe un terme T tel que $\Gamma \vdash t : T$.

Définition 37. *: Contenu d'un Terme Marqué*
Soit t un terme marqué, le contenu de t est le terme sans marques $t^\#$ défini par récurrence sur la structure de t :

- *si t est une sorte, alors $t^\# = t$,*
- *si $t = x^T$, alors $t^\# = x$,*
- *si $t = (u\ v)^T$, alors $t^\# = (u^\#\ v^\#)$,*
- *si $t = ([x:P]u)^T$, alors $t^\# = [x:P^\#]u^\#$,*
- *si $t = ((x:P)U)^T$, alors $t^\# = (x:P^\#)U^\#$.*

Définition 38. *: Contenu d'un Contexte*
Soit $\Gamma = [x_1 : P_1; ...; x_n : P_n]$ un contexte marqué, le contenu de Γ est le contexte :

$$\Gamma^\# = [x_1 : P_1^\#; ...; x_n : P_n^\#]$$

Proposition 16. *: Soit t un terme marqué de type T dans un contexte Γ. Le terme $t^\#$ est bien typé et de type $T^\#$ dans $\Gamma^\#$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $\Gamma \vdash t : T$.

2.2 Normalisation des Termes Marqués

Définition 39. *: $\beta\eta$-réduction*
On note $\triangleright$ la relation de $\beta\eta$-réduction (en une étape). Le radical réduit peut aussi bien se trouver dans le terme que dans les marques.
On note $\triangleright^*$ la relation de $\beta\eta$-réduction en un nombre quelconque d'étapes et $\triangleright^+$ la relation de $\beta\eta$-réduction en au moins une étape.

Définition 40. *: Traduction d'un Terme Marqué en un Terme non Marqué.*
Soit t un terme marqué qui est bien typé ou une sorte, soit T son type dans le Calcul des Constructions avec Univers et s le type de T. On définit par récurrence sur la structure de t un terme t° du Calcul des Constructions avec Univers :

— *si t est une sorte on pose $t^\circ = t$,*
— *si $t = x^T$, alors on pose $t^\circ = (([z : s]x)\ T^\circ)$,*
— *si $t = (u\ v)^T$, alors on pose $t^\circ = (([z : s](u^\circ\ v^\circ))\ T^\circ)$,*
— *si $t = ([x : P]u)^T$, alors on pose $t^\circ = (([z : s][x : P^\circ]u^\circ)\ T^\circ)$,*
— *si $t = ((x : P)Q)^T$, alors on pose $t^\circ = (([z : s](x : P^\circ)Q^\circ)\ T^\circ)$.*
Cette traduction est similaire à celles définies dans [33] [26].

Définition 41. *: Traduction d'un Contexte Marqué en un Contexte non Marqué*
Soit $\Gamma = [x_1 : P_1; ...; x_n : P_n]$ un contexte marqué, on pose $\Gamma^\circ = [x_1 : P_1^\circ; ...; x_n : P_n^\circ]$

Proposition 17. *: Soit Γ un contexte marqué et t et T deux termes marqués tels que $\Gamma \vdash t : T$. On a $\Gamma^\circ \vdash t^\circ : T^\circ$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $\Gamma \vdash t : T$.

Proposition 18. *: $a^\circ[x \leftarrow b^\circ] \rhd^* (a[x \leftarrow b])^\circ$*

Démonstration : Par récurrence sur la structure de a.
Si $a = x^T$ alors :

$$a^\circ = [z : s]x\ T^\circ$$

$$a^\circ[x \leftarrow b^\circ] = [z : s]b^\circ\ T^\circ[x \leftarrow b^\circ] \rhd^* b^\circ = (x^T[x \leftarrow b])^\circ = (a[x \leftarrow b])^\circ$$

Les autres cas sont une simple application de l'hypothèse de récurrence. Par exemple si a est une application $a = (t\ u)^T$, on a :

$$a^\circ = [z : s](t^\circ\ u^\circ)T^\circ$$

Donc :

$$a^\circ[x \leftarrow b^\circ] = [z : s](t^\circ[x \leftarrow b^\circ]\ u^\circ[x \leftarrow b^\circ])T^\circ[x \leftarrow b^\circ]$$

Par hypothèse de récurrence on a :

$$t^\circ[x \leftarrow b^\circ] \rhd^* (t[x \leftarrow b])^\circ$$

$$u^\circ[x \leftarrow b^\circ] \rhd^* (u[x \leftarrow b])^\circ$$

$$T^\circ[x \leftarrow b^\circ] \rhd^* (T[x \leftarrow b])^\circ$$

Donc :

$$a^\circ[x \leftarrow b^\circ] \rhd^* [z : s]((t[x \leftarrow b])^\circ\ (u[x \leftarrow b])^\circ)(T[x \leftarrow b])^\circ$$
$$= ((t[x \leftarrow b]\ u[x \leftarrow b])^{T[x \leftarrow b]})^\circ = (a[x \leftarrow b])^\circ$$

Proposition 19. *: Si $a \rhd b$, alors $a^\circ \rhd^+ b^\circ$.*

Démonstration : Si $a = (([x : P]t)^T u)^U$ et $b = t[x \leftarrow u]$,

$$a^\circ = ([z : s]((([z' : s'][x : P^\circ]t^\circ)\ T^\circ)\ u^\circ)\ U^\circ)$$

$$b^\circ = (t[x \leftarrow u])^\circ$$

Dans a°, on réduit d'abord trois β-radicaux pour obtenir le terme $t^\circ[x \leftarrow u^\circ]$, puis un certain nombre de radicaux pour obtenir le terme b°.

Si $a = ([x : P](t\ x^P)^T)^U$ et $b = t$,

$$a^\circ = ([z : s][x : P^\circ]([z' : s'](t^\circ\ ([z'' : s'']x)\ P^\circ))\ T^\circ)U^\circ$$

$$b^\circ = t^\circ$$

On réduit trois β-radicaux et un η-radical pour obtenir le terme b°.

Et de même si on réduit un radical dans un sous-terme.

Lemme 1. *: La $\beta\eta$-réduction sur les termes marqués est fortement normalisable.*

Démonstration : S'il existait une réduction infinie issue d'un terme marqué t, il en existerait aussi une issue de t°, ce qui est en contradiction avec la normalisation forte du Calcul des Constructions avec Univers.

Proposition 20. *: Soient a et b deux termes marqués normaux bien typés dans un contexte marqué Γ. Si $a^\# = b^\#$, alors $a = b$.*

Démonstration : Par récurrence sur la structure de a. Comme a et b ont le même contenu, ce sont ou bien deux sortes, ou bien deux variables, ou bien deux abstractions, ou bien deux produits ou bien deux applications.

Si ce sont par exemple deux applications, $a = (t\ u)^T$, $b = (v\ w)^U$, alors les termes t et v bien typés et normaux dans Γ et ont même contenu, ils sont donc égaux. De même les termes u et w sont bien typés dans Γ et ont même contenu, ils sont donc égaux. Soit V le type de $a^\# = b^\#$ dans $\Gamma^\#$. Les termes T et U sont bien typés et normaux dans Γ, ils ont tous deux le contenu V, ils sont donc égaux. Donc $a = b$.

Et de même si a et b sont deux sortes, variables, abstractions ou produits.

Proposition 21. *: Soit a et b deux termes marqués. Si $a \rhd^* b$, alors $a^\# \rhd^* b^\#$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $a \rhd^* b$.

Proposition 22. *: Unicité de la Forme Normale*

Soit Γ un contexte et t, a et b des termes bien typés dans Γ tels que a et b sont normaux et t se réduit (en un nombre arbitraire d'étapes) sur a et b, alors $a = b$.

Démonstration : Les termes a et b sont bien typés dans Γ et leur contenu est la forme normale de $t^\#$.

2.3 Terme Marqué Associé à un Terme non Marqué

On veut associer à chaque terme non marqué un terme marqué. On pourrait considérer le terme non marqué et marquer chacun de ses sous-termes par son type, mais nous devrions alors marquer les nouveaux sous-termes introduits comme marques et prouver que ce processus termine. Il est en fait plus simple de construire le terme marqué par récurrence sur la longueur de la dérivation de typage du terme non marqué.

Proposition 23. *: Soit a et b deux termes marqués. Si $a \equiv b$, alors $a^{\#} \equiv b^{\#}$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $a \equiv b$.

Proposition 24. *: Soit a et b deux termes marqués bien typés dans le contexte marqué Γ. Si $a^{\#} \equiv b^{\#}$, alors $a \equiv b$.*

Démonstration : Soit c la forme normale de a et d celle de b. Le terme $c^{\#}$ est la forme normale de $a^{\#}$ et $d^{\#}$ la forme normale de $b^{\#}$. Comme $a^{\#} \equiv b^{\#}$, on a $c^{\#} = d^{\#}$, donc $c = d$ et de ce fait $a \equiv b$.

Définition 42. *: Traduction d'un Terme non Marqué en un Terme Marqué*

Par récurrence sur la longueur de la dérivation non marquée $\Delta \vdash a : A$ ou Δ bien formé, on construit un contexte marqué Δ^* et des termes marqués a^* et A^* tels que $\Delta^* \vdash a^* : A^*$ et ${\Delta^*}^{\#} = \Delta$, ${a^*}^{\#} = a$ et ${A^*}^{\#} = A$ ou un contexte bien formé Δ^* tel que ${\Delta^*}^{\#} = \Delta$.

— *Si la dernière règle est :*

$$\frac{}{[\,]\ \textit{bien formé}}$$

on pose $\Delta^* = [\,]$.

— *Si la dernière règle est :*

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \textit{bien formé}}$$

alors par hypothèse de récurrence on a construit Γ^* et T^*. On pose $\Delta^* = \Gamma^*[x : T^*]$.

— *Si la dernière règle est :*

$$\frac{\Gamma\ \textit{bien formé}}{\Gamma \vdash s : s'}$$

alors par hypothèse de récurrence on a construit Γ^*. On pose $\Delta^* = \Gamma^*$, $a^* = s$ et $A^* = s'$.

— *Si la dernière règle est :*

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

alors par hypothèse de récurrence on a construit Γ^* et T^* et $x : T^* \in \Gamma^*$. On pose $\Delta^* = \Gamma^*$, $a^* = x^{T^*}$ et $A^* = T^*$.

— *Si la dernière règle est :*

$$\frac{\Gamma \vdash T : s \quad \Gamma[x : T] \vdash U : s'}{\Gamma \vdash (x : T)U : s''}$$

alors par hypothèse de récurrence on a construit Γ^*, T^* et U^*.
On pose $\Delta^* = \Gamma^*$, $a^* = ((x : T^*)U^*)^{s''}$ et $A^* = s''$.

— *Si la dernière règle est :*

$$\frac{\Gamma \vdash ((x : T)U) : s \quad \Gamma[x : T] \vdash t : U}{\Gamma \vdash [x : T]t : (x : T)U}$$

alors par hypothèse de récurrence on a construit Γ^*, $((x : T)U)^*$, T^*, t^* et U^*. On pose $\Delta^* = \Gamma^*$, $a^* = ([x : T^*]t^*)^{((x:T^*)U^*)^s}$ et $A^* = ((x : T^*)U^*)^s$.

— *Si la dernière règle est :*

$$\frac{\Gamma \vdash t : (x : T)U \quad \Gamma \vdash u : T}{\Gamma \vdash (t\ u) : U[x \leftarrow u]}$$

alors par hypothèse de récurrence on a construit Γ^*, t^*, $((x : T)U)^*$, u^* *et* T^*. *Comme* $((x : T)U)^{*\ \#} = ((x : T)U)$ *le terme* $((x : T)U)^*$ *est un produit* $((x : T)U)^* = ((x : P)Q)^s$ *avec* $P^\# = T$ *et* $Q^\# = U$. *On pose* $\Delta^* = \Gamma^*$, $a^* = (t^*\ u^*)^{Q[x \leftarrow u^*]}$ *et* $A^* = Q[x \leftarrow u^*]$.

— *Si la dernière règle est :*

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash U : s \quad \Gamma \vdash t : T \quad T \equiv U}{\Gamma \vdash t : U}$$

alors par hypothèse de récurrence on a construit Γ^*, t^*, T^* *et* U^*. *On a* $T^{*\ \#} \equiv U^{*\ \#}$, *donc* $T^* \equiv U^*$. *On pose* $\Delta^* = \Gamma^*$, $a^* = t^*$ *et* $A^* = U^*$.

Remarque : Soit Δ un contexte et a et A deux termes tels que $\Delta \vdash a : A$. Deux dérivations de $\Delta \vdash a : A$ peuvent mener à construire deux terme a_1^* et a_2^* distincts. Mais on a $a_1^{*\ \#} = a_2^{*\ \#} = a$. Donc d'après la proposition 24, $a_1^* \equiv a_2^*$ et donc a_1^* et a_2^* ont même forme normale.

2.4 Forme η-longue d'un Terme Marqué

Définition 43. *: Mesure d'un Terme Marqué*

Soit Γ *un contexte marqué et* t *un terme marqué bien typé dans* Γ. *On définit par récurrence sur la structure de* t, *la mesure* $\mu(t)$ *de* t *:*

— *Si* t *est une sorte, alors* $\mu(t) = 1$,
— *Si* $t = x^T$, *alors* $\mu(t) = \mu(T)$,
— *Si* $t = (u\ v)^T$, *alors* $\mu(t) = \mu(u) + \mu(v) + \mu(T)$,
— *Si* $t = ([x : U]v)^T$, *alors* $\mu(t) = \mu(U) + \mu(v) + \mu(T)$,
— *Si* $t = ((x : U)V)^T$, *alors* $\mu(t) = \mu(U) + \mu(V) + \mu(T)$.

Définition 44. *: Forme* η*-longue d'un Terme Marqué*

Soit Γ *un contexte marqué et* t *un terme marqué* $\beta\eta$*-normal bien typé dans* Γ. *La forme* η*-longue du terme* t *est définie par récurrence sur* $\mu(t)$ *:*

— *Si* $t = ([x : U]u)^T$, *alors soit* U' *la forme* η*-longue de* U *dans* Γ, T' *la forme* η*-longue de* T *dans* Γ *et* u' *la forme* η*-longue de* u *dans* $\Gamma[x : U]$, *on définit la forme* η*-longue de* t *comme le terme* $([x : U']u')^{T'}$.
— *Si* $t = ((x : U)V)^T$, *alors soit* U' *la forme* η*-longue de* U *dans* Γ, T' *la forme* η*-longue de* T *dans* Γ *et* V' *la forme* η*-longue de* V *dans* $\Gamma[x : U]$, *on définit la forme* η*-longue de* t *comme le terme* $((x : U')V')^T$.
— *Si* $t = (((w^{T_0}\ c_1)^{T_1})\ ...\ c_p)^{T_p}$, *alors notons* $T = T_p$ *le type de* t.
Soit $T = ((x_1 : P_1)...((x_n : P_n)P)^{U_n}...)^{U_1}$ *(*P *atomique). On pose* c_i' *la forme* η*-longue de* c_i *dans* Γ, T_i' *la forme* η*-longue de* T_i *dans* Γ, P_i' *la forme* η*-longue de* P_i *dans le contexte* $\Gamma[x_1 : P_1; ...; x_{i-1} : P_{i-1}]$, U_i' *la forme* η*-longue de* U_i *dans ce même contexte et* x_i' *la forme* η*-longue de* $x_i^{P_i}$ *dans le contexte* $\Gamma[x_1 : P_1; ...; x_i : P_i]$. *On définit la forme* η*-longue de* t *comme le terme :*

$$[x_1 : P_1'][x_n : P_n'](((((w^{T_0'}\ c_1')^{T_1'}\ ...\ c_p')^{T_p'}\ x_1')^{V_1'}\ ...\ x_n')^{V_n'}$$

où $V_i' = ((x_{i+1} : P_1')...((x_{i+1} : P_n')P')^{U_n'}...)^{U_1'}$.

Proposition 25. *: La définition de la forme η-longue des termes non marqués donnée au chapitre précédent est bien fondée.*

Démonstration : Soit Γ un contexte non marqué et t un terme non marqué normal bien typé dans Γ. Soit u la forme normale de sa traduction t^*. On pose $\mu(t) = \mu(u)$. La définition de la forme η-longue des termes non marqués donnée au chapitre précédent est une récurrence sur $\mu(t)$.

2.5 Bonne Fondation de la Relation $<$

Définition 45. *: Traduction Normale d'un Terme non Marqué*

Soit t un terme bien typé dans le contexte Γ, on définit sa traduction normale t^+ comme la forme η-longue de la forme normale de sa traduction t^*.

Lemme 2. *: La relation $<$ est bien fondée.*

Démonstration : Soient t et u deux termes non marqués normaux, si t est la forme normale η-longue du type de u et le terme u n'est pas une sorte alors t^+ est la marque la plus externe de u^+, c'est donc un sous-terme strict de u^+. Si t est un sous-terme strict de u, alors par récurrence sur la structure de u, le terme t^+ est un sous-terme strict de u^+. On en déduit donc que si $t < u$, alors t^+ est un sous-terme strict de u^+.

Première partie
Démonstration Automatique

Chapitre 3

Contextes Quantifiés Contraints, Substitution

3.1 Preuves Incomplètes

Variables Universelles et Variables Existentielles

Plaçons-nous dans un système de types quelconque et considérons le contexte :

$$\Gamma = [A : Prop; B : Prop; C : Prop; u : B \to C; v : A \to B; w : A]$$

Dans ce contexte on cherche une preuve de la proposition C. Une preuve de cette proposition est $(u\ (v\ w))$. Toute l'idée de la méthode de démonstration automatique que nous présentons ici revient à pouvoir inférer cette preuve étape par étape, en considérant par exemple dans une première étape que la preuve commence par la variable u. Nous allons développer dans ce chapitre le matériel technique qui nous permettra de considérer ainsi des étapes extrêmement élémentaires dans la construction des preuves.

La première notion dont nous avons besoin est celle de preuve incomplète. Dans notre exemple, l'axiome $B \to C$ nous permet de nous ramener à la recherche d'une preuve de B. En effet, si on trouve une preuve t de B, le terme $(u\ t)$ sera une preuve de C. Cette information peut s'exprimer par le fait que $(u\ x)$ est une preuve incomplète de C. La variable x est de type B, la recherche d'une preuve de B, se traduit par la recherche d'un terme par lequel instancier cette variable. Une fois un terme trouvé, l'application de la substitution $x \leftarrow t$ à la preuve $(u\ x)$ donnera la preuve $(u\ t)$.

Cette méthode n'est qu'une reformulation de l'idée de base des méthodes algébriques. En effet lorsqu'on recherche un entier dont le carré plus six est égal au quintuple, on déclare une variable x pour cet entier, ce qui nous permet de considérer les entiers incomplets $x^2 + 6$ et $5x$ et d'exprimer la contrainte :

$$x^2 + 6 = 5x$$

Une solution de cette équation est une substitution qui lie un entier à la variable x, par exemple la substitution $x \leftarrow 2$. En considérant une expression comme x^2+6, nous avons enrichi le formalisme des entiers en ajoutant une catégorie syntaxique : les variables.

Il est tentant, pour considérer des preuves incomplètes, de s'inspirer de cette méthode c'est-à-dire d'ajouter au formalisme des termes une nouvelles sorte d'objets (les variables), définir une

notion de typage pour ces termes étendus, une notion de substitution, etc. Cette méthode est en fait excessivement lourde. En effet elle n'exploite pas une particularité essentielle du λ-calcul, qui est que ce formalisme à la différence de celui des entiers, comporte déjà une notion de variable. Notre opinion est qu'il est beaucoup plus économique d'utiliser cette notion que de la dupliquer, en considérant deux classes de variables : les variables du λ-calcul et des méta-variables dénotant des λ-termes. Ainsi, pour nous, du point de vue de la bonne formation des contextes et du typage des termes, rien ne distingue les variables x et u, en particulier le terme $(u\ x)$ n'est bien typé que dans un contexte dans lequel aussi bien les variables u que x sont déclarées. En revanche, ces variables ont des propriétés différentes vis-à-vis de la substitution : x peut être instanciée par une substitution et u ne le peut pas.

Cette information binaire est exprimée en associant un quantificateur à chaque variable d'un contexte, le quantificateur universel aux variables qui ne peuvent pas être instanciées et le quantificateur existentiel à celles qui le peuvent. Ce choix vient du fait qu'une méthode pour prouver $\forall x : T.(P\ x)$ consiste à prouver $(P\ x)$ en ajoutant au contexte une variable que l'on ne peut pas instancier $x : T$ alors qu'une méthode pour prouver $\exists x : T.(P\ x)$ consiste à prouver $(P\ x)$ en ajoutant au contexte une variable que l'on peut instancier $x : T$.

La terminologie classique de l'unification oppose variable et constante, le mot "constante" étant déjà utilisé dans les systèmes de types à d'autres fins nous lui préférerons celui de variable universelle, opposé à variable existentielle.

Pour définir la notion de substitution, le premier problème qui se pose est celui des nouvelles variables existentielles qui apparaissent dans le terme substitué. Reprenons notre exemple ci-dessus. Dans le contexte Γ on cherche une preuve de C, on introduit donc une variable existentielle de type C :

$$\Gamma[\exists p : C]$$

On veut substituer à p le terme $(u\ x)$ où x est une nouvelle variable existentielle, afin d'obtenir le contexte :

$$\Gamma[\exists x : B]$$

La déclaration de p a bien entendu disparu puisque cette variable a été instanciée, mais une nouvelle variable existentielle x est apparue dans le contexte, les informations concernant cette variable doivent figurer dans la substitution, qui doit donc s'écrire $p \leftarrow [\exists x : B], (u\ x)$. Ensuite, en appliquant les substitutions $x \leftarrow [\exists y : A], (v\ y)$ puis $y \leftarrow [\ ], w$, on obtient le contexte Γ qui ne contient plus de variables existentielles. La composition de ces trois substitutions donne la substitution $p \leftarrow [\ ], (u\ (v\ w))$ et donc la preuve $(u\ (v\ w))$ pour C.

Contraintes

Lorsqu'on applique une substitution à une variable x, il faut que le terme substitué à x ait le même type que la variable x. Considérons le contexte :

$$\Gamma = [\forall A : Prop; \exists X : Prop; \exists y : X \to A]$$

a priori on ne peut pas substituer le terme $[x : X]x$ à la variable y car le terme $[x : X]x$ est de type $X \to X$ et y est de type $X \to A$.

Mais si on applique d'abord la substitution $X \leftarrow [\ ], A$, la substitution $y \leftarrow [\ ], [x : A]x$ devient autorisée. Pour appliquer la substitution $y \leftarrow [\ ], [x : X]x$ il faut donc *d'abord* unifier le type de y et

celui de $[x : X]x$ (c'est-à-dire rechercher les substitutions qui rendent ces deux termes identiques) pour *ensuite* appliquer la substitution $y \leftarrow [\,], [x : A]x$. Il faut donc passer par une étape d'unification avant chaque substitution. L'unification dans les systèmes de types n'est pas un problème simple, notre objectif dans ce chapitre est de permettre l'expression d'étapes de construction de preuves aussi élémentaires que possible, en aucun cas la résolution d'un problème d'unification ne peut être considéré comme une étape élémentaire. Nous allons donc autoriser la substitution $y \leftarrow [\,], [x : X]x$ mais en gardant dans le contexte la contrainte $X \to X = X \to A$ pour rappeler qu'en appliquant cette substitution, nous avons pris l'engagement de résoudre, d'une façon ou d'une autre, cette équation, même si nous ne voulons pas le faire immédiatement.

3.2 Contextes Quantifiés Contraints

Définition 46. *: Contextes Quantifiés Contraints*

Une déclaration quantifiée est un triplet $< Q, x, T >$ (noté $Qx : T$) où Q est un quantificateur ($\forall$ ou $\exists$), x une variable et T un terme. Une contrainte est un couple de termes $< a, b >$ (noté $a = b$). Un contexte quantifié contraint est une liste $\Gamma = [e_1; ...; e_n]$ telle que e_i est ou bien une déclaration quantifiée ou bien une contrainte. Ces contextes quantifiés contraints sont des généralisations des préfixes mixtes de Miller [51] qui sont des listes de déclarations quantifiées.

Les contextes non quantifiés et non contraints sont identifiés avec les contextes quantifiés contraints qui ne comportent que des variables universelles.

Définition 47. *: Equivalence Modulo Contraintes*

Soit Γ un contexte quantifié contraint, on définit la relation entre termes $\equiv_\Gamma$ comme la plus petite relation d'équivalence compatible avec la structure des termes qui contiennent la $\beta\eta$-équivalence et les équations de Γ.

Elle est définie inductivement par :

— *si $t \equiv t'$, alors $t \equiv_\Gamma t'$,*
— *si $(a = b) \in \Gamma$, alors $a \equiv_\Gamma b$,*
— *si $t \equiv_\Gamma t'$, alors $t' \equiv_\Gamma t$,*
— *si $t \equiv_\Gamma t'$ et $t' \equiv_\Gamma t''$, alors $t \equiv_\Gamma t''$,*
— *si $t \equiv_\Gamma t'$, alors $[x : u]t \equiv_\Gamma [x : u]t'$,*
— *si $t \equiv_\Gamma t'$, alors $[x : t]u \equiv_\Gamma [x : t']u$,*
— *si $t \equiv_\Gamma t'$, alors $(x : u)t \equiv_\Gamma (x : u)t'$,*
— *si $t \equiv_\Gamma t'$, alors $(x : t)u \equiv_\Gamma (x : t')u$,*
— *si $t \equiv_\Gamma t'$, alors $(t\ u) \equiv_\Gamma (t'\ u)$,*
— *si $t \equiv_\Gamma t'$, alors $(u\ t) \equiv_\Gamma (u\ t')$.*

Définition 48. *: Règles de Typage*

D'abord on modifie les règles de typage pour tenir compte de la nouvelle syntaxe des contextes. On remplace la règle :

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \textit{bien formé}}\, s \in S$$

par :

$$\frac{\Gamma \vdash T : s}{\Gamma[Qx : T]\ \textit{bien formé}}\, s \in S$$

On remplace de même la règle :

$$\frac{\Gamma \text{ bien formé } \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

par :

$$\frac{\Gamma \text{ bien formé } \quad Qx : T \in \Gamma}{\Gamma \vdash x : T}$$

Et on ajoute la règle :

$$\frac{\Gamma \vdash a : T \quad \Gamma \vdash b : T}{\Gamma[a = b] \text{ bien formé}}$$

Ensuite on étend le système en remplaçant la règle :

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'} \, s \in S$$

par :

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv_\Gamma T'}{\Gamma \vdash t : T'} \, s \in S$$

On définit ainsi deux nouveaux jugements : Γ *est bien formé en utilisant les contraintes et* t *est de type* T *dans* Γ *en utilisant les contraintes.*

Remarque : Un terme peut être bien typé en utilisant les contraintes sans être normalisable. Par exemple, dans le contexte $\Gamma = [\exists T : Prop; T = T \to T]$ le terme $([x : T](x\ x)[x : T](x\ x))$ est bien typé et n'est pas normalisable.

Remarque : Dans le $\lambda\Pi$-calcul, on ne considère jamais de variables existentielles de type. Elliott [20] [21] et Pym [58] ont montré en substance que tout terme bien typé dans le $\lambda\Pi$-calcul, dans un contexte quantifié contraint sans contraintes reliant deux termes rigides (voir ci-dessous) a une forme normale de tête. L'exemple ci-dessus montre que ce théorème ne se généralise pas aux systèmes polymorphes.

Définition 49. *: Terme Bien Typé Sans Utiliser les Contraintes*

Soit Γ *un contexte et* t *et* T *deux termes. Le terme* t *est dit de type* T *dans* Γ *sans utiliser les contraintes s'il existe un sous-contexte* Δ *de* Γ *(c'est-à-dire obtenu en effaçant certains éléments de* Γ*) tel que* Δ *ne comporte pas de contraintes,* Δ *soit bien formé et* $\Delta \vdash t : T$*.*

Proposition 26. *: Un terme bien typé sans utiliser les contraintes a une forme normale unique.*

Définition 50. *: Forme Normale d'un Contexte*

Soit Γ *un contexte, la forme normale de* Γ *est obtenue en mettant en forme* β*-normale* η*-longue tous les types des variables qui sont bien typés sans utiliser les contraintes et toutes les contraintes dont les termes sont bien typés sans utiliser les contraintes.*

Définition 51. *: Terme sans Variables Existentielles*

Un terme t *est dit sans variables existentielles dans* Γ *si aucune variable existentielle n'a d'occurrence dans ce terme.*

Définition 52. *: Terme Fermé*

Un terme est dit fermé dans Γ *s'il est héréditairement sans variables existentielles c'est-à-dire si pour chaque variable* x *de type* T*, libre dans ce terme,* x *est universelle dans* Γ *et* T *est fermé dans le préfixe de* Γ *déclaré à gauche de* x*. Il est dit ouvert sinon.*

Proposition 27. *: Si* Γ *est un contexte bien formé, alors le contexte* Γ' *obtenu en effaçant les contraintes reliant deux termes identiques est également bien formé et* $\Gamma \vdash t : T$ *si et seulement si* $\Gamma' \vdash t : T$*.*

Démonstration : Par récurrence sur la longueur de la dérivation de Γ *bien formé* ou $\Gamma \vdash t : T$.

Définition 53. *: Contextes Equivalents*

Deux contextes Γ *et* Γ' *(bien formé ou non) sont dits équivalents si les contextes obtenus en effaçant dans* Γ *et* Γ' *les contraintes reliant des termes identiques sont identiques.*

Par la suite, on identifiera les contextes équivalents.

Définition 54. *: Termes Flexibles et Rigides*

Soit t *un terme bien typé sans utiliser les contraintes dans un contexte* Γ *et normal. Si* t *est une abstraction, un produit ou un terme atomique* $(w\ c_1\ ...\ c_n)$ *où* w *est une variable universelle, il est dit rigide. Si* t *est atomique* $(w\ c_1\ ...\ c_n)$ *et* w *est existentielle, il est dit flexible.*

Définition 55. *: Contexte de Succès et Contexte d'Echec*

Un contexte normal Γ *est dit contexte de succès s'il ne comporte que des variables universelles et des contraintes reliant des termes identiques, il est dit contexte d'échec s'il comporte une contrainte reliant deux termes sans variables existentielles qui sont différents.*

Proposition 28. *: Soit* Γ *un contexte normal qui n'est ni un contexte de succès ni un contexte d'échec. Il existe dans* Γ *une variable existentielle* $x : T$ *dont le type* T *est bien typé sans utiliser les contraintes et a la forme* $(x_1 : P_1)...(x_n : P_n)P$ *avec* P *atomique rigide dans* $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$*.*

Démonstration : Considérons l'objet le plus à gauche de Γ qui n'est ni une variable universelle ni une contrainte normale reliant deux termes identiques (un tel objet existe car le contexte n'est pas un contexte de succès). Cet objet n'est pas une contrainte. En effet, supposons le contraire, et posons $\Gamma = \Delta[a = b]\Delta'$. Les termes a et b sont bien typés dans Δ, donc comme il n'y a pas de variables existentielles ni de contraintes autres que triviales dans Δ, a et b sont sans variables existentielles et bien typés sans utiliser les contraintes dans Γ. Comme a et b ne sont pas identiques, Γ est alors un contexte d'échec ce qui contredit l'hypothèse.

Cet objet est donc la déclaration d'une variable existentielle $\exists x : T$, le terme T est bien typé sans utiliser les contraintes et sans variables existentielles.

3.3 Substitution

Définition 56. *: Contexte Existentiel*

Un contexte (bien formé ou non) est dit existentiel s'il ne contient que des déclarations de variables existentielles et des contraintes mais pas de déclarations de variables universelles.

Définition 57. *: Substitution*

Soit σ un ensemble fini de triplets $< x, \gamma, t >$ où x est une variable, γ un contexte existentiel et t un terme. L'ensemble σ est appelé substitution si pour toute variable x, il y a au plus un triplet de la forme $< x, \gamma, t >$ dans σ.

Dans le cas où σ est réduite à un seul triplet $< x, \gamma, t >$ on note $\sigma = x \leftarrow \gamma, t$, et $\sigma = x \leftarrow t$ quand γ est le contexte vide.

Définition 58. *: Variable Liée par une Substitution*

Soit x une variable et σ une substitution. Si la substitution σ contient un triplet $< x, \gamma, t >$, alors x est dite liée par σ. Le contexte γ est appelé le contexte associé à x dans σ.

Définition 59. *: Substitution Appliquée à un Terme*

Soit x une variable et σ une substitution. S'il existe un triplet $< x, \gamma, t >$ dans σ, alors on pose $\sigma x = t$, sinon on pose $\sigma x = x$. Cette définition s'étend aux termes de manière directe par :

- $\sigma s = s$,
- $\sigma(t\ u) = (\sigma t\ \sigma u)$,
- $\sigma[x : T]u = [x : \sigma T]\sigma u$,
- $\sigma(x : T)u = (x : \sigma T)\sigma u$.

Définition 60. *: Application d'une Substitution à un Contexte*

On définit par récurrence sur la longueur de Γ une relation de compatibilité σ est bien typée dans Γ et si σ est bien typée dans Γ un contexte bien formé $\sigma\Gamma$.

- *Dans le cas où $\Gamma = [\ ]$, σ est bien typée dans Γ et $\sigma\Gamma = [\ ]$.*
- *Dans le cas où $\Gamma = \Delta[\forall x : T]$, si σ est bien typée dans Δ et $\sigma\Delta[\forall x : \sigma T]$ est bien formé, alors σ est bien typée dans Γ et $\sigma\Gamma = \sigma\Delta[\forall x : \sigma T]$. Sinon σ n'est pas bien typée dans Γ.*
- *Dans le cas où $\Gamma = \Delta[\exists x : T]$, si x est liée par σ, alors il existe un triplet unique $< x, \gamma, t >$ dans σ, sinon on pose $t = x$ et $\gamma = [\exists x : \sigma T]$.*
  Si σ est bien typée dans Δ, $(\sigma\Delta)\gamma$ est bien formé et $(\sigma\Delta)\gamma \vdash t : \sigma T$, alors σ est bien typée dans Γ et $\sigma\Gamma = (\sigma\Delta)\gamma$. Sinon σ n'est pas bien typée dans Γ.
- *Dans le cas où $\Gamma = \Delta[a = b]$, si σ est bien typée dans Δ et $(\sigma\Delta)[\sigma a = \sigma b]$ est bien formé, alors σ est bien typée dans Γ et $\sigma\Gamma = (\sigma\Delta)[\sigma a = \sigma b]$. Sinon σ n'est pas bien typée dans Γ.*

Définition 61. *: Substitution Equivalentes*

Deux substitutions σ et σ' sont dites équivalentes si les variables liées par ces substitutions sont les mêmes, et pour toute variable x liée par ces substitutions $\sigma x = \sigma' x$ et les contextes γ et γ' associés à x par σ et σ' sont équivalents.

Par la suite, on identifiera les substitutions équivalentes.

Proposition 29. *: Si $\Delta\Delta'$ et $\Delta\gamma$ sont des contextes bien formés, alors $\Delta\gamma\Delta'$ aussi. De même si $\Delta\Delta' \vdash t : T$ et $\Delta\gamma$ est un contexte bien formé, alors $\Delta\gamma\Delta' \vdash t : T$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $\Delta\Delta'$ bien formé ou $\Delta\Delta' \vdash t : T$.

Notation : Soit $\Gamma = [e_1, ..., e_n]$ un contexte (bien formé ou non), x une variable non déclarée dans Γ et t un terme, on note $\Gamma[x \leftarrow t]$ le contexte $[e_1[x \leftarrow t], ..., e_n[x \leftarrow t]]$ où on définit $(Qy : T)[x \leftarrow t]$ comme $Qy : T[x \leftarrow t]$ et $(a = b)[x \leftarrow t]$ comme $(a[x \leftarrow t] = b[x \leftarrow t])$.

Proposition 30. *: Soit Γ un contexte, x une variable non déclarée dans Γ et T, T' et u des termes tels que $T \equiv_\Gamma T'$, alors $T[x \leftarrow u] \equiv_{\Gamma[x \leftarrow u]} T'[x \leftarrow u]$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $T \equiv_\Gamma T'$.

Proposition 31. *: Si $\Delta[Qx : U]\Delta' \vdash t : T$ et $\Delta \vdash u : U$, alors le contexte $\Delta(\Delta'[x \leftarrow u])$ est bien formé et $\Delta(\Delta'[x \leftarrow u]) \vdash t[x \leftarrow u] : T[x \leftarrow u]$.*

Démonstration : Par récurrence sur la longueur de la dérivation de $\Delta[Qx : U]\Delta' \vdash t : T$.

Proposition 32. *: Soit Γ un contexte, σ une substitution bien typée dans Γ, t et T deux termes tels que $\Gamma \vdash t : T$. On a $\sigma\Gamma \vdash \sigma t : \sigma T$.*

Démonstration : Par récurrence sur le nombre de variables de Γ liées par σ.

Si aucune des variables de Γ n'est liée par σ le résultat est immédiat.
Sinon soit $\exists x : P$ la variable existentielle liée par σ la plus à droite de Γ.
On écrit $\Gamma = \Delta[\exists x : P][e_1; ...; e_p]$.
Soit γ le contexte associé à x par σ. Soit $\tau = \sigma - \{< x, \gamma, \sigma x >\}$.

Comme x est la variable liée par σ la plus à droite de Γ, pour toute variable y déclarée dans Γ distincte de x, x n'a pas d'occurrence dans τy et donc $\sigma y = \tau y = \tau y[x \leftarrow \sigma x]$. Pour la variable x on a $x = \tau x$ donc $\sigma x = \tau x[x \leftarrow \sigma x]$. On en déduit par récurrence sur la structure de t que pour tout terme t on a $\sigma t = (\tau t)[x \leftarrow \sigma x]$.
Par hypothèse de récurrence, pour tout contexte Ξ tel que la substitution τ soit bien typée dans Ξ et tous termes u et U tel que $\Xi \vdash u : U$, on a $\tau\Xi \vdash \tau u : \tau U$.
Par une récurrence simple sur la longueur de Δ on prouve que σ et τ sont bien typées dans Γ et :

$$\tau\Gamma = (\tau\Delta)[\exists x : \tau P][\tau e_1; ...; \tau e_p] = (\sigma\Delta)[\exists x : \tau P][\tau e_1; ...; \tau e_p]$$

En utilisant la propriété de τ on a :

$$\tau\Gamma \vdash \tau t : \tau T$$

c'est-à-dire :

$$(\sigma\Delta)[\exists x : \tau P][\tau e_1; ...; \tau e_p] \vdash \tau t : \tau T$$

Comme σ est bien typée dans Γ on a :

$$(\sigma\Delta)\gamma \text{ bien formé}$$

D'après la proposition 29 :

$$(\sigma\Delta)\gamma[\exists x : \tau P][\tau e_1; ...; \tau e_p] \vdash \tau t : \tau T$$

Donc, en utilisant la proposition 31 :

$$(\sigma\Delta)\gamma[(\tau e_1)[x \leftarrow \sigma x]; ...; (\tau e_p)[x \leftarrow \sigma x]] \vdash (\tau t)[x \leftarrow \sigma x] : (\tau T)[x \leftarrow \sigma x]$$

c'est-à-dire :

$$(\sigma\Delta)\gamma[\sigma e_1; ...; \sigma e_p] \vdash \sigma t : \sigma T$$

soit :

$$\sigma\Gamma \vdash \sigma t : \sigma T$$

Définition 62. *: Composition de Substitutions*
Soit un contexte Γ *et* σ *et* τ *deux substitutions. On définit la substitution* $\tau \circ \sigma$ *comme :*

$$(\tau \circ \sigma) = \{< x, \tau\gamma, \tau t > \mid < x, \gamma, t > \in \sigma\} \cup \{< x, \gamma, t > \mid < x, \gamma, t > \in \tau \text{ et } x \text{ non liée par } \sigma\}$$

où $\tau\gamma$ *est défini par récurrence par :*
- *si* $\gamma = [\,]$*, alors* $\tau\gamma = [\,]$*,*
- *si* $\gamma = \gamma'[\exists y : U]$ *alors* $\tau\gamma = (\tau\gamma')\gamma''$ *avec* γ'' *contexte associé à* y *par* τ *si la variable* y *est liée par cette substitution et* $\gamma'' = [\exists y : \tau U]$ *sinon,*
- *si* $\gamma = \gamma'[a = b]$*, alors* $\tau\gamma = (\tau\gamma')[\tau a = \tau b]$*.*

Proposition 33. *: Soit* σ *et* τ *deux substitutions et* t *un terme,* $(\tau \circ \sigma)t = \tau\sigma t$*.*

Démonstration : Par récurrence sur la structure de t.

Proposition 34. *: Soit* Γ *un contexte et* σ *et* τ *deux substitutions, telles que* σ *est bien typée dans* Γ *et* τ *est bien typée dans* $\sigma\Gamma$*, alors* $\tau \circ \sigma$ *est bien typée dans* Γ *et* $(\tau \circ \sigma)\Gamma = \tau\sigma\Gamma$*.*

Démonstration : Par récurrence sur la longueur de Γ.
- Si $\Gamma = \Delta[a = b]$, alors puisque σ est bien typée dans Γ, σ est bien typée dans Δ, $(\sigma\Delta)[\sigma a = \sigma b]$ est bien formé et $\sigma\Gamma = (\sigma\Delta)[\sigma a = \sigma b]$.
  Comme τ est bien typée dans $\sigma\Gamma$, τ est bien typée dans $\sigma\Delta$, $(\tau\sigma\Delta)[\tau\sigma a = \tau\sigma b]$ est bien formé et $\tau\sigma\Gamma = (\tau\sigma\Delta)[\tau\sigma a = \tau\sigma b] = (\tau\sigma\Delta)[(\tau \circ \sigma)a = (\tau \circ \sigma)b]$.
  Par hypothèse de récurrence, $(\tau \circ \sigma)$ est bien typée dans Δ et $(\tau \circ \sigma)\Delta = \tau\sigma\Delta$,
  donc $((\tau \circ \sigma)\Delta)[(\tau \circ \sigma)a = (\tau \circ \sigma)b]$ est bien formé.
  Donc $(\tau \circ \sigma)$ est bien typée dans Γ et $(\tau \circ \sigma)\Gamma = ((\tau \circ \sigma)\Delta)[(\tau \circ \sigma)a = (\tau \circ \sigma)b] = \tau\sigma\Gamma$.
- Si $\Gamma = \Delta[\forall x : T]$, alors puisque σ est bien typée dans Γ, σ est bien typée dans Δ, $(\sigma\Delta)[\forall x : \sigma T]$ est bien formé et $\sigma\Gamma = (\sigma\Delta)[\forall x : \sigma T]$.
  Comme τ est bien typée dans $\sigma\Gamma$, τ est bien typée dans $\sigma\Delta$, $(\tau\sigma\Delta)[\forall x : \tau\sigma T]$ est bien formé et $\tau\sigma\Gamma = (\tau\sigma\Delta)[\forall x : \tau\sigma T] = (\tau\sigma\Delta)[\forall x : (\tau \circ \sigma)T]$.
  Par hypothèse de récurrence, $(\tau \circ \sigma)$ est bien typée dans Δ et $(\tau \circ \sigma)\Delta = \tau\sigma\Delta$,
  donc $((\tau \circ \sigma)\Delta)[\forall x : (\tau \circ \sigma)T]$ est bien formé.
  Donc $(\tau \circ \sigma)$ est bien typée dans Γ et $(\tau \circ \sigma)\Gamma = ((\tau \circ \sigma)\Delta)[\forall x : (\tau \circ \sigma)T] = \tau\sigma\Gamma$.
- Si $\Gamma = \Delta[\exists x : T]$, alors soit γ le contexte associé à x par σ si x est liée par σ et $\gamma = [\exists x : \sigma T]$ sinon. On écrit $\gamma = [e_1; ...; e_n]$. Comme σ est bien typée dans Γ, σ est bien typée dans Δ, $(\sigma\Delta)[e_1; ...; e_n]$ est bien formé, $\sigma\Gamma = (\sigma\Delta)[e_1; ...; e_n]$ et $\sigma\Gamma \vdash \sigma x : \sigma T$.
  Si l'objet e_i est la déclaration d'une variable existentielle $y : U$ liée par τ, on pose γ'_i le contexte associé par τ à y, si c'est la déclaration d'une variable existentielle $y : U$ qui n'est pas liée par τ, on pose $\gamma'_i = [\exists y : \tau U]$ et si c'est la contrainte $(a = b)$ on pose $\gamma'_i = [(\tau a = \tau b)]$.
  Comme τ est bien typée dans $\sigma\Gamma$, il est bien typé dans $\sigma\Delta$, $(\tau\sigma\Delta)\gamma'_1...\gamma'_n = (\tau\sigma\Delta)(\tau\gamma)$ est bien formé et :
  $$\tau\sigma\Gamma = (\tau\sigma\Delta)\gamma'_1...\gamma'_n = (\tau\sigma\Delta)(\tau\gamma)$$
  Comme $\sigma\Gamma \vdash \sigma x : \sigma T$, on a $\tau\sigma\Gamma \vdash \tau\sigma x : \tau\sigma T$.
  Enfin par hypothèse de récurrence $(\tau \circ \sigma)$ est bien typée dans Δ et $(\tau \circ \sigma)\Delta = \tau\sigma\Delta$. Donc $((\tau \circ \sigma)\Delta)(\tau\gamma)$ est bien formé, $\tau\sigma\Gamma = ((\tau \circ \sigma)\Delta)(\tau\gamma)$ et $((\tau \circ \sigma)\Delta)(\tau\gamma) \vdash (\tau \circ \sigma)x : (\tau \circ \sigma)T$.
  Donc $(\tau \circ \sigma)$ est bien typée dans Γ et $(\tau \circ \sigma)\Gamma = ((\tau \circ \sigma)\Delta)(\tau\gamma) = \tau\sigma\Gamma$.

Chapitre 4

Une Méthode Complète de Synthèse de Preuves

4.1 Approches

4.1.1 Résolution et Unification

En logique du premier ordre ou d'ordre supérieur, on distingue deux catégories syntaxiques : les termes et les preuves. Les termes sont des arbres (en logique du premier ordre) ou des λ-termes simplement typés (en logique d'ordre supérieur) et les preuves sont des arbres construits avec les règles de déduction naturelle. Ces règles permettent de construire des preuves en combinant d'autres preuves et des termes. Par exemple la règle d'élimination de l'implication :

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

permet de former une preuve (de B) en combinant deux preuves (l'une de $A \to B$ et l'autre de A). La règle d'élimination du quantificateur universel :

$$\frac{\Gamma \vdash \forall x : A.B \quad \Gamma \vdash t : A}{\Gamma \vdash B[x \leftarrow t]}$$

permet de former une preuve (de $B[x \leftarrow t]$) en combinant une preuve (de $\forall x : A.B$) et un terme (de type A).

Les preuves sont donc des arbres hétérogènes. Dans les méthodes de démonstration automatique pour ces deux logiques [59] [37], les preuves sont synthétisées par l'algorithme de *résolution*. Quand au cours de la recherche d'une preuve, il est nécessaire de synthétiser un terme, on appelle un autre algorithme : l'algorithme d'*unification*. Dans les systèmes de types, termes et preuves appartiennent à une catégorie syntaxique unique. De ce fait notre méthode de synthèse de preuves est constituée d'un seul algorithme qui mêle résolution et unification.

Nous présentons maintenant les algorithmes d'unification et de résolution à l'ordre supérieur [37] [40] [41]. Nous cherchons à montrer qu'une même idée sur l'énumération des termes sous-tend ces deux algorithmes.

4.1.2 Unification à l'Ordre Supérieur

Soit Γ un contexte et a et b deux termes, l'unification consiste à chercher s'il existe une substitution σ bien typée dans Γ telle que $\sigma a = \sigma b$.

La méthode d'unification à l'ordre supérieur [40] [41] est basée sur un algorithme énumérant tous les termes normaux en forme η-longue d'un type donné dans le λ-calcul simplement typé.

Soit un type $T_1 \to ... \to T_n \to T$ (T atomique). Tous les termes normaux en forme η-longue de ce type commencent par n abstractions :

$$t = [x_1 : T_1]...[x_n : T_n]t'$$

Le terme t' a le type T, c'est donc un terme atomique. Une variable $w : U_1 \to ... \to U_p \to U$ qui est une variable du contexte ou l'un des x_i peut être tête de ce terme seulement si $U = T$. Dans ce cas, cette variable doit être appliquée à p termes afin d'obtenir un terme du bon type. Donc :

$$t = [x_1 : T_1]...[x_n : T_n](w\ c_1\ ...\ c_p)$$

Pour marquer la dépendance des c_i par rapport aux x_j on écrit :

$$t = [x_1 : T_1]...[x_n : T_n](w\ (d_1\ x_1\ ...\ x_n)\ ...\ (d_p\ x_1\ ...\ x_n))$$

Enfin pour synthétiser les d_i on utilise récursivement le même algorithme. On pose donc dans un premier temps :

$$t = [x_1 : T_1]...[x_n : T_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

puis on utilise récursivement le même algorithme pour instancier les $h_i : T_1 \to ... \to T_n \to U_i$.

Dans l'algorithme d'unification, les équations entre un terme flexible et un terme rigide sont résolue en utilisant cette méthode pour instancier la variable de tête du terme flexible. Dans ce cas, le choix de la variable w est très restreint par le terme rigide. Les équations entre termes rigides sont simplifiées et les équations entre termes flexibles ont une solution triviale.

La méthode d'énumération des termes sous-tendant l'algorithme d'unification à l'ordre supérieur peut donc être résumée ainsi :

- essayer toutes les variables de tête possibles pour la forme normale du terme,
- déclarer de nouvelles variables pour le reste du terme,
- engendrer une contrainte de type pour assurer le bon typage du terme,
- utiliser récursivement cette méthode pour instancier ces variables.

4.1.3 Résolution à l'Ordre Supérieur

Présentons maintenant la méthode de résolution à l'ordre supérieur [37]. Nous présentons en fait uniquement une restriction incomplète de cette méthode pour les clauses de Horn, c'est-à-dire les propositions de la forme $P_1 \to ... \to P_n \to Q$ où $P_1, ..., P_n$ et Q sont des propositions atomiques (on considère que les variables libres dans ces propositions sont quantifiées universellement en tête de la clause).

Quand le but B est atomique, on l'unifie avec la tête Q d'une hypothèse $P_1 \to ... \to P_n \to Q$, on engendre les buts dérivés $\sigma P_1, ..., \sigma P_n$ et on cherche récursivement à prouver ces propositions.

Quand le but est aussi une clause de Horn $A_1 \to ... \to A_n \to B$, la mise en forme clausale de la négation de cette proposition introduit les hypothèses A_1, ..., A_n et le but atomique B.

Cette restriction est appelée la méthode d'*introduction-résolution*. Elle peut être présentée par deux règles : la règle de résolution entre un but atomique et une hypothèse est appelée *résolution* et celle qui permet pour prouver $A \to B$ de prouver B après avoir introduit l'hypothèse A est appelée *introduction*. Cette méthode est incomplète. Dans [37] une autre règle (la règle de *scission* (splitting)) est ajoutée afin d'obtenir une méthode complète.

Remarquons enfin que cette méthode peut aussi être utilisée pour les *clauses de Horn héréditaires* c'est-à-dire des propositions de la forme $P_1 \to ... \to P_n \to Q$, où Q est une proposition atomique et $P_1, ..., P_n$ des clauses de Horn héréditaires.

4.1.4 Introduction-Résolution dans les Systèmes de Types

Dans un système de types, toutes les propositions peuvent être considérées comme des clauses de Horn héréditaires où l'implication est généralisée en un produit. On peut donc appliquer la méthode précédente.

Par ailleurs, dans un système de types, la preuve d'une proposition P est un terme t de type P, une méthode de synthèse de preuves ne doit donc pas se contenter d'affirmer que la proposition considérée est prouvable, mais aussi exhiber un terme-preuve de cette proposition.

L'algorithme d'introduction-résolution peut être ainsi présenté :

• *Introduction* : Pour trouver une preuve de la proposition $(x : A)B$ dans un contexte Γ, trouver une preuve de B dans le contexte $\Gamma[x : A]$.

Quand on a une preuve b de B, le terme $[x : A]b$ est une preuve de $(x : A)B$ dans Γ.

• *Résolution* : Pour trouver une preuve de B dans un contexte Γ dans lequel il y a une proposition :

$$f : (x_1 : P_1)...(x_n : P_n)Q$$

unifier B et Q (considérer $x_1, ..., x_n$ comme des variables existentielles dans Q), puis pour les x_i qui n'ont pas été instanciées par l'unification, démontrer σP_i.

Pour chaque i, on obtient (par unification ou comme preuve d'un but dérivé) un terme c_i de type $P_i[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}]$, le terme $(f\ c_1\ ...\ c_n)$ est alors une preuve de B.

Cette méthode est décrite en détail dans [35]. Donnons un exemple. On a une hypothèse :

$$f : (x : T)(y : T)(z : T)(u : (R\ x\ y))(v : (R\ y\ z))(R\ x\ z)$$

et on veut prouver $(R\ a\ c)$. Pour cela on unifie $(R\ x\ z)$ et $(R\ a\ c)$, ce qui donne la substitution $x \leftarrow a, z \leftarrow c$. Ensuite, on construit récursivement des termes $b : T$, $t : (R\ a\ b)$ et $t' : (R\ b\ c)$. On obtient le terme $(f\ a\ b\ c\ t\ t')$ qui est une preuve de $(R\ a\ c)$.

Cette méthode est en général incomplète. Un autre défaut est qu'il n'y a pas d'algorithme général d'unification connu pour les systèmes de types. Des algorithmes sont uniquement connus pour le λ-calcul simplement typé [40] [41] et pour le $\lambda\Pi$-calcul [20] [21] [58].

Une présentation alternative de cette méthode est la suivante :

• *Introduction* : On donne le terme $[x : A]h$ comme preuve du but initial, la variable h représente le but dérivé à prouver.

• *Résolution* : On donne le terme $(f\ h_1\ ...\ h_p)$ comme preuve du but initial et l'équation $Q[x_1 \leftarrow h_1, ..., x_p \leftarrow h_p] = B$ (dans notre exemple $(R\ h_1\ h_3) = (R\ a\ b)$) garantit que ce terme a le type B.

La résolution de cette équation va instancier certaines variables (ici h_1 et h_3), les autres variables (ici h_2, h_4 et h_5) représentent les buts dérivés à prouver.

Quand pour prouver une proposition on applique d'abord la règle d'introduction n fois, puis la règle de résolution, on obtient la preuve : $[x_1 : A_1]...[x_n : A_n](f\ h_1\ ...\ h_p)$. La variable f apparaît comme variable de tête de cette preuve. Remarquons que la dépendance des h_i par rapport aux x_j est implicite.

La méthode d'énumération des termes qui sous-tend la méthode d'introduction-résolution est donc :

- essayer toutes les variables de tête possibles pour la forme normale du terme,
- déclarer de nouvelles variables pour le reste du terme,
- engendrer une contrainte de type pour assurer le bon typage du terme,
- utiliser récursivement cette méthode pour instancier ces variables.

C'est donc essentiellement la même méthode que celle qui sous-tend l'algorithme d'unification.

4.1.5 Une Méthode d'Enumération des Termes d'un Type Donné

Nous allons maintenant utiliser cette méthode pour construire un algorithme de synthèse de preuves dans les systèmes de types.

Pour énumérer les termes t qui peuvent être substitués à une variable x de type T on imagine la forme normale de t :

$$t = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_p)$$

ou :

$$t = [x_1 : P_1]...[x_n : P_n](z : A)B$$

et on considère les substitutions élémentaires :

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](z : (h_1\ x_1\ ...\ x_n))(h_2\ x_1\ ...\ x_n\ z)$$

Ensuite, on énumère tous les termes qui peuvent être substitués aux variables $h_1, ..., h_p$.

Comme on considère des termes normaux en forme η-longue, le nombre d'abstraction n est le même que celui de produits dans le type T et les types $P_1, ..., P_n$ des variables liées dans ces abstractions sont ceux des variables liées dans ces produits.

Par complétude, on doit considérer :

— toutes les variables de tête w possibles,

— tous les entiers p possibles et tous les types possibles pour $h_1, ..., h_p$.

Dans une telle substitution le terme substitué à x doit avoir le même type que cette variable. Cette contrainte est une équation. Dans la méthode d'introduction-résolution, la résolution de cette équation est appelée l'étape d'unification de la résolution. Dans la méthode d'unification dans le λ-calcul simplement typé, les types sont toujours des termes sans variables existentielles, il est donc juste nécessaire de vérifier que ces deux termes sont égaux, c'est la phase de sélection des variables de tête. Dans l'algorithme d'unification dans le $\lambda\Pi$-calcul [20] [21] [58] cette contrainte est ajoutée à l'ensemble d'équations, c'est l'équation de *garantie* (accounting equation).

Dans notre algorithme, ces équations sont gardées dans le contexte. Elles forment des contraintes que les substitutions à venir devront vérifier. Puisque nous n'avons pas d'algorithme spécial pour résoudre ces équations et puisque les variables apparaissant dans ces équations sont instanciées par

l'algorithme ordinaire, notre méthode est constituée d'un seul algorithme qui mêle résolution et unification.

4.1.6 Nombre d'Applications

Dans la méthode d'introduction-résolution, quand on veut prouver une proposition atomique T et qu'on utilise la règle de résolution avec une hypothèse $w : (y_1 : Q_1)...(y_q : Q_q)Q$ (Q atomique) on ne considère que les preuves de la forme $(w\ h_1\ ...\ h_q)$ et la contrainte est que le type de ce terme doit être égal à T, c'est-à-dire $Q[y_1 \leftarrow h_1, ..., y_p \leftarrow h_p] = T$. Mais dans les systèmes de types polymorphes, le type du terme $(w\ c_1\ ...\ c_q)$ peut encore être un produit, et on doit considérer des substitutions $(w\ h_1\ ...\ h_p)$ avec p strictement supérieur à q. C'est pour cette raison que la méthode d'introduction-résolution est incomplète et qu'une règle de scission est nécessaire dans [37].

Ici nous considérons une infinité de possibilités pour ce nombre d'applications, et donc un arbre infiniment branchant. Dans une méthode incomplète mais plus efficace, nous considérerons une restriction similaire à celle de la méthode introduction-résolution.

4.1.7 Variables dans la Proposition à Prouver

Dans certains systèmes, les variables h_i peuvent avoir des occurrences dans le type de h_j pour $j > i$. On pourrait instancier d'abord la variable h_i puis n'instancier h_j que quand son type est sans variables existentielles, mais il est plus efficace de commencer par h_j quand cela est possible.

Par exemple, pour énumérer les termes de type $T = \exists x : Nat.(P\ x)$, il ne faut pas énumérer tous les termes n de type Nat puis énumérer les termes de type $(P\ n)$, mais au contraire, commencer par énumérer les termes de type $(P\ x)$.

De ce fait dans les systèmes de types polymorphes, quand une proposition à prouver P comporte une variable, il n'est pas toujours possible de connaître le nombre d'abstractions dans une preuve t de cette proposition, car une substitution peut augmenter le nombre de produits dans P et donc le nombre d'abstractions dans t. Dans ce cas, il faut retarder l'instanciation de $x : T$ jusqu'à ce que T ait été instancié en un terme $(x_1 : P_1)...(x_n : P_n)P$ où la variable de tête de P n'est pas une variable existentielle.

4.2 Une Méthode Complète

Définition 63. *: Substitutions Elémentaires*

Soit Γ un contexte bien formé dans le système de types Méta et normal tel que le type de toutes les variables universelles soit Prop ou Type mais pas Extern et qui n'est ni un contexte de succès ni un contexte d'échec. On définit un ensemble de substitutions $\Sigma(\Gamma)$.

On choisit parmi celles de Γ, une variable existentielle x dont le type est normal et de la forme $(x_1 : P_1)...(x_n : P_n)P$ avec P atomique rigide dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ (une telle variable existe parce que Γ n'est ni un contexte de succès ni un contexte d'échec).

Notation *: Quand t est un terme, $(\vec{x} : \vec{P})t$ est une abréviation pour $(x_1 : P_1)...(x_n : P_n)t$.*

— *Pour toute variable universelle w déclarée à gauche de x dans Γ ou qui est l'un des x_i :*

$$\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n] \vdash w : (y_1 : Q'_1)...(y_q : Q'_q)Q' \text{ } (Q' \text{ atomique})$$

On pose :

$$Q_1 = Q'_1$$
$$Q_2 = [y_1 : Q'_1]Q'_2$$
$$...$$
$$Q_q = [y_1 : Q'_1]...[y_{q-1} : Q'_{q-1}]Q'_q$$
$$Q = [y_1 : Q'_1]...[y_q : Q'_q]Q'$$

$$w : (y_1 : Q_1)(y_2 : (Q_2\ y_1))...(y_q : (Q_q\ y_1\ ...\ y_{q-1}))(Q\ y_1\ ...\ y_q)$$

Pour tout $r \geq 0$ on considère les substitutions de degré de scission r :
Soit s la sorte type de Q' et s' la sorte type de P, dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.
Pour tout $s_1, s'_1, ..., s_i, s'_i, ..., s_r, s'_r$ tel que $< s_1, s'_1, s >\in R$, ..., $< s_i, s'_i, s'_{i-1} >\in R$ et $s'_r = s'$, on pose le contexte existentiel γ :

$$\gamma = \varphi\chi_1...\chi_r\psi$$

où :

$$\varphi = [\exists h_1 : (\vec{x} : \vec{P})Q_1;$$
$$\exists h_2 : (\vec{x} : \vec{P})(Q_2\ (h_1\ x_1\ ...\ x_n));$$
$$...;$$
$$\exists h_q : (\vec{x} : \vec{P})(Q_q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q-1}\ x_1\ ...\ x_n))]$$
$$\chi_1 = [\exists H_1 : (\vec{x} : \vec{P})s_1;$$
$$\exists K_1 : (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))s'_1;$$
$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z);$$
$$\exists h_{q+1} : (\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)]$$

et pour tout i, $1 < i \leq r$:

$$\chi_i = [\exists H_i : (\vec{x} : \vec{P})s_i;$$
$$\exists K_i : (\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))s'_i;$$
$$(\vec{x} : \vec{P})(K_{i-1}\ x_1\ ...\ x_n\ (h_{q+i-1}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))(K_i\ x_1\ ...\ x_n\ z);$$
$$\exists h_{q+i} : (\vec{x} : \vec{P})(H_i\ x_1\ ...\ x_n)]$$

enfin, si $r = 0$:

$$\psi = [(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P]$$

et dans les autres cas :

$$\psi = [(\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P]$$

L'ensemble $\Sigma(\Gamma)$ est l'ensemble des substitutions qui associent à x le contexte γ et le terme :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n))$$

— *Si P est une sorte, alors :*
pour toute sorte s, telle que $< s, P > \in Ax$, on ajoute à $\Sigma(\Gamma)$ la substitution qui associe à la variable x le contexte vide $[\,]$ et le terme :

$$[x_1 : P_1]...[x_n : P_n]s$$

et pour toutes sortes s, s' telles que $< s, s', P > \in R$ on pose :

$$\gamma = [\exists h : (\vec{x} : \vec{P})s \ ; \ \exists k : (\vec{x} : \vec{P})(y : (h\ x_1\ ...\ x_n))s']$$

on ajoute à $\Sigma(\Gamma)$ la substitution qui associe à la variable x le contexte γ et le terme :

$$[x_1 : P_1]...[x_n : P_n](y : (h\ x_1\ ...\ x_n))(k\ x_1\ ...\ x_n\ y)$$

Définition 64. *: Dérivation*
Une dérivation d'un contexte Γ est une liste de contextes $[\Gamma_1; ...; \Gamma_m]$ telle que $\Gamma_1 = \Gamma$, Γ_m est un contexte de succès et $\Gamma_{i+1} = \sigma_i\Gamma_i$ avec $\sigma_i \in \Sigma(\Gamma_i)$.

Définition 65. *: Arbre de Recherche*
Soit un contexte Γ, on définit l'arbre de recherche de Γ. Les nœuds de cet arbre sont des contextes, la racine de l'arbre est le contexte Γ et les fils d'un nœud Δ sont tous les contextes $\sigma\Delta$ pour $\sigma \in \Sigma(\Delta)$. Les contextes de succès et d'échec sont les feuilles de cet arbre. Les nœuds de succès sont en bijection avec les dérivations de Γ. Un semi-algorithme de synthèse de preuves consiste à énumérer les nœuds de cet arbre jusqu'à trouver un nœud de succès. Comme le nombre de fils d'un nœud de cet arbre peut être infini, pour énumérer tous ses nœuds il faut retarder l'exploration des nœuds de degré de scission r de r générations.

Explication Informelle :
Dans les solutions de degré de scission nul, les q premiers éléments de γ (φ) sont les déclarations des nouvelles variables $h_1, ..., h_q$ apparaissant dans le terme substitué à x et le dernier (ψ) est une contrainte exprimant que cette substitution est bien typée.

Dans les solutions de degré de scission r $(r \geq 1)$, on déclare d'abord les variables $h_1, ..., h_q$ (φ) puis il faut que le type $(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$ du terme $(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$ soit un produit $(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$. On déclare pour cela (χ_1) deux variables existentielles H_1 et K_1 ainsi qu'une contrainte :

$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

Ensuite on peut déclarer la variable h_{q+1} avec le type $(\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)$.

Le type de $(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n))$ est alors $(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$. Si $r = 1$ on a juste besoin d'une dernière contrainte (ψ) pour assurer que la substitution est bien typée :

$$(\vec{x} : \vec{P})(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P$$

Dans le cas général on déclare ainsi quatre éléments par degré de scission et ensuite la contrainte (ψ) pour assurer que la substitution est bien typée.

Les autres substitutions sont celles où la variable x est instanciée par un terme de la forme $[x_1 : P_1]...[x_n : P_n]u$ où u est une sorte ou un produit.

Remarque : Quand on a un contexte qui contient plusieurs variables existentielles, on peut choisir la variable que l'on instancie en premier. Le choix de cette variable est déterminant pour des questions d'efficacité. Des heuristiques seront développées par la suite. Néanmoins, le choix de la variable est limité par le fait que la variable à instancier doit avoir un type bien typé sans utiliser les contraintes et de tête rigide. Cette condition impose donc une *séquentialisation* partielle de la résolution.

4.3 Exemples

4.3.1 Un Exemple au Premier Ordre

Soit $\Delta = [\forall T : Prop; \forall R : T \to T \to Prop; \forall Eq : T \to T \to Prop;$
$\forall Antisym : (x : T)(y : T)((R\ x\ y) \to (R\ y\ x) \to (Eq\ x\ y)); \forall a : T; \forall b : T; \forall u : (R\ a\ b); \forall v : (R\ b\ a)]$
et $\Gamma = \Delta[\exists x : (Eq\ a\ b)]$.
On substitue à x le terme qui a pour tête $Antisym$ et un degré de scission nul :

$$x \leftarrow (Antisym\ h_1\ h_2\ h_3\ h_4)$$

On obtient le contexte :
$\Delta[\exists h_1 : T; \exists h_2 : T; \exists h_3 : (R\ h_1\ h_2); \exists h_4 : (R\ h_2\ h_1); (Eq\ h_1\ h_2) = (Eq\ a\ b)]$
On substitue à h_1 le terme qui a pour tête a et un degré de scission nul :

$$h_1 \leftarrow a$$

et à h_2 le terme qui a pour tête b et un degré de scission nul :

$$h_2 \leftarrow b$$

On obtient le contexte $\Delta[\exists h_3 : (R\ a\ b); \exists h_4 : (R\ b\ a)]$.
(Les contraintes reliant des termes identiques sont omises.)
On substitue à h_3 le terme qui a pour tête u et un degré de scission nul :

$$h_3 \leftarrow u$$

On obtient le contexte $\Delta[\exists h_4 : (R\ b\ a)]$.
On substitue à h_4 le terme qui a pour tête v et un degré de scission nul :

$$h_4 \leftarrow v$$

Et on obtient le contexte de succès Δ.
Soit θ la composition de ces substitutions, $\theta x = (Antisym\ a\ b\ u\ v)$.

Comme nous le verrons par la suite, dans cet exemple, à chaque étape, toutes les substitutions sauf celle considérée mènent de façon évidente à des contextes voués à l'échec.

4.3.2 Un Exemple avec Scission

Soit :
$\Delta = [\forall A : Prop; \forall B : Prop; \forall I : Prop \to Prop; \forall u : (P : Prop)((I\ P) \to P); \forall v : (I(A \to B));$
$\forall w : A]$

et $\Gamma = \Delta[\exists p : B]$.
On substitue à p le terme qui a pour tête u et 1 pour degré de scission :

$$p \leftarrow (u\ h_1\ h_2\ h_3)$$

avec les nouvelles variables : $h_1 : Prop$, $h_2 : (I\ h_1)$, $H : Prop$, $K : H \to Prop$ et $h_3 : H$.
On obtient le contexte :
$\Delta[\exists h_1 : Prop; \exists h_2 : (I\ h_1); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H; h_1 = (x : H)(K\ x); (K\ h_3) = B]$.
On substitue à h_1 un produit :

$$h_1 \leftarrow (x : h')(k'\ x)$$

On obtient le contexte :
$\Delta[\exists h' : Prop; \exists k' : h' \to Prop; \exists h_2 : (I\ (x : h')(k'\ x)); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H;$
$(x : H)(K\ x) = (x : h')(k'\ x); (K\ h_3) = B]$.
On substitue à h_2 le terme qui a pour tête v et un degré de scission nul :

$$h_2 \leftarrow v$$

On obtient le contexte :
$\Delta[\exists h' : Prop; \exists k' : h' \to Prop; (I\ (x : h')(k'\ x)) = (I\ (A \to B)); \exists H : Prop; \exists K : H \to Prop;$
$\exists h_3 : H; (x : H)(K\ x) = (x : h')(k'\ x); (K\ h_3) = B]$.
On substitue à h' le terme qui a pour tête A et un degré de scission nul :

$$h' \leftarrow A$$

$\Delta[\exists k' : A \to Prop; (I\ (x : A)(k'\ x)) = (I\ (A \to B)); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H;$
$(x : H)(K\ x) = (x : A)(k'\ x); (K\ h_3) = B]$.
On substitue à k' le terme qui a pour tête B et un degré de scission nul :

$$k' \leftarrow [x : A]B$$

On obtient le contexte :
$\Delta[\exists H : Prop; \exists K : H \to Prop; \exists h_3 : H; (x : H)(K\ x) = (A \to B); (K\ h_3) = B]$.
On substitue à H le terme qui a pour tête A et un degré de scission nul :

$$H \leftarrow A$$

On obtient le contexte :
$\Delta[\exists K : A \to Prop; \exists h_3 : A; (x : A)(K\ x) = (A \to B); (K\ h_3) = B]$.
On substitue à K le terme qui a pour tête B et un degré de scission nul :

$$K \leftarrow [x : A]B$$

On obtient le contexte :
$\Delta[\exists h_3 : A]$
On substitue à h_3 le terme qui a pour tête w et un degré de scission nul :

$$h_3 \leftarrow w$$

Et on obtient le contexte de succès Δ.
Soit θ la composition de ces substitutions, $\theta p = (u\ (A \to B)\ v\ w)$.

4.4 Propriétés

4.4.1 Typage

Soit Γ un contexte et x une variable existentielle de Γ. Ecrivons $\Gamma = \Delta[\exists x : T]\Delta'$. Soit $\sigma \in \Sigma(\Gamma)$, $\sigma = x \leftarrow \gamma, t$. Nous montrons à présent que σ est bien typé dans Γ. Pour cela nous procédons en deux étapes : d'abord nous montrons que $\Delta\gamma$ est bien formé, c'est-à-dire que les types des nouvelles variables existentielles et les nouvelles contraintes sont bien typés, ensuite nous montrons que dans $\Delta\gamma$, le terme t est bien typé et a même type que x.

Proposition 35. *: Les types des nouvelles variables existentielles et les nouvelles contraintes sont bien typés dans le système de types Méta.*

Démonstration : Comme $(x_1 : P_1)...(x_n : P_n)P$ est bien typé dans le système de types $Méta$, les termes P_i sont bien typés dans le système de types $Méta$ du type $Prop$ ou $Type$ (mais pas $Extern$).

Le terme $(y_1 : Q'_1)...(y_q : Q'_q)Q'$ est ou bien le type d'une variable universelle ou bien l'un des P_i. C'est donc un terme de type $Prop$ ou $Type$ mais pas $Extern$. Le type des Q'_i et de Q' est donc $Prop$ ou $Type$ mais pas $Extern$. Les termes Q_i et Q sont donc bien typés dans le système de types $Méta$.

Les termes $(H_i\ x_1\ ...\ x_n)$ et $(h\ x_1\ ...\ x_n)$ sont bien typés dans le système de types $Méta$ et ont le type s avec $< s, s', s'' > \in R$ pour des sortes s' et s''. Donc s est égal à $Prop$ ou $Type$ mais pas $Extern$.

Les termes $(Q_i\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{i-1}\ x_1\ ...\ x_n))$, s_i, s'_i, $(H_i\ x_1\ ...\ x_n)$, $(K_i\ x_1\ ...\ x_n\ z)$, $(K_i\ x_1\ ...\ x_n\ (h_i\ x_1\ ...\ x_n))$, P, s, s', $(h\ x_1\ ...\ x_n)$ et $(k\ x_1\ ...\ x_n\ y)$ sont bien typés dans le système de types $Méta$ et ont le type $Prop$, $Type$ ou $Extern$.

Donc le type des variables existentielles et les contraintes sont bien typés dans le système de types $Méta$.

Proposition 36. *: Le terme t est bien typé dans $\Delta\gamma$ et a le même type que x dans le système Méta.*

Démonstration : Pour les substitutions de degré de scission nul, on a :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$$

donc en utilisant la contrainte ψ :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})P$$

Pour les substitutions de degré de scission r $(r \geq 1)$, on prouve par récurrence sur i que pour tout i, $i \geq 1$, on a dans le contexte $\Delta\gamma$:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_i\ x_1\ ...\ x_n\ (h_{q+i}\ x_1\ ...\ x_n))$$

Pour $i = 1$ on a :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$$

donc en utilisant la contrainte de χ_1 :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

dans le contexte $\Delta\gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ on a :

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

et comme :

$$h_{q+1} : (\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)$$

on déduit :

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n)) : (K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$$

donc dans le contexte $\Delta\gamma$:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$$

Puis, si on suppose cette proposition vérifiée pour i :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_i\ x_1\ ...\ x_n\ (h_{q+i}\ x_1\ ...\ x_n))$$

En utilisant la contrainte de χ_i :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(z : (H_{i+1}\ x_1\ ...\ x_n))(K_{i+1}\ x_1\ ...\ x_n\ z)$$

on a donc, dans le contexte $\Delta\gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$:

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (z : (H_{i+1}\ x_1\ ...\ x_n))(K_{i+1}\ x_1\ ...\ x_n\ z)$$

et comme :

$$h_{q+i+1} : (\vec{x} : \vec{P})(H_{i+1}\ x_1\ ...\ x_n)$$

on en déduit :

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i+1}\ x_1\ ...\ x_n)) : (K_{i+1}\ x_1\ ...\ x_n\ (h_{q+i+1}\ x_1\ ...\ x_n))$$

donc dans le contexte $\Delta\gamma$:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i+1}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_{i+1}\ x_1\ ...\ x_n\ (h_{q+i+1}\ x_1\ ...\ x_n))$$

Donc on déduit :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n))$$

et en utilisant la dernière contrainte (ψ) :

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})P$$

Si P est une sorte et σ une substitution supplémentaire, σt est bien typé de façon évidente.

Proposition 37. *: Soit Γ un contexte et σ une substitution, si $\sigma \in \Sigma(\Gamma)$, alors σ est bien typée dans Γ.*

Démonstration : Par récurrence sur la longueur de Γ et en utilisant les deux propositions ci-dessus et la proposition de préservation des jugements de typage par application d'une substitution bien typée.

4.4.2 Correction

Lemme 3. *: Soit un contexte Γ, s'il existe une dérivation $[\Gamma_1, ..., \Gamma_m]$ de Γ, alors il existe une substitution θ telle que $\theta\Gamma$ soit un contexte de succès. La substitution θ est appelée substitution dénotée par cette dérivation.*

Démonstration : Soit $\theta = \sigma_{m-1} \circ ... \circ \sigma_1$, $\theta\Gamma = \Gamma_m$ est un contexte de succès.

Prouvons maintenant que la substitution θ qui est bien typée dans le système *Méta* est aussi bien typée dans le système originel $\mathcal{T}$.

Proposition 38. *: Soit un contexte Γ bien formé dans le système de types $\mathcal{T}$ et T un type bien typé dans Γ dans le système de types $\mathcal{T}$.*

Soit t un terme tel que $\Gamma \vdash t : T$ dans le système de types Méta et tel que pour tout sous-terme de t qui est un produit $(x : U)U'$, si on pose s le type de U, s' le type de U' et s'' le type de $(x : U)U'$, on a $< s, s', s'' > \in R$ (l'ensemble de règles du système $\mathcal{T}$) et que pour tout sous-terme de t qui est une sorte s, on a $s = Prop$ (et non $Type$). On a $\Gamma \vdash t : T$ dans le système $\mathcal{T}$.

Démonstration : Par récurrence sur la structure de t :

- Si $t = [x : U]u'$, alors $T = (x : U)U'$ est bien typé dans le système $\mathcal{T}$ donc U' est bien typé dans le contexte $\Gamma[\forall x : U]$ dans ce système $\mathcal{T}$ et par hypothèse de récurrence $\Gamma[\forall x : U] \vdash u' : U'$ dans le système $\mathcal{T}$, donc $\Gamma \vdash t : T$ dans le système $\mathcal{T}$.
- Si $t = (x : U)U'$, alors par hypothèse de récurrence U et U' sont bien typés dans le système $\mathcal{T}$ et comme la règle $< s, s', s'' >$ est une règle de ce système, on a $\Gamma \vdash t : T$ dans le système $\mathcal{T}$.
- Si $t = (x\ c_1\ ...\ c_n)$ et x est une variable déclarée dans Γ, alors x est bien typé dans Γ dans le système $\mathcal{T}$ et on prouve par récurrence sur i que le type de c_i est bien typé dans le système $\mathcal{T}$ et donc que c_i est bien typé dans le système $\mathcal{T}$, on en conclut que $\Gamma \vdash t : T$ dans le système $\mathcal{T}$.
- Si t est une sorte, alors par hypothèse $t = Prop$, ce terme est bien typé dans le système $\mathcal{T}$.

Proposition 39. *: Soit un contexte Γ bien formé dans le système $\mathcal{T}$, $[\Gamma_1; ...; \Gamma_m]$ une dérivation de Γ et θ la substitution dénotée par cette dérivation. Soit x une variable existentielle de Γ et $t = \theta x$. Pour tout sous-terme de t qui est un produit $(x : U)U'$, si on pose s le type de U, s' le type de U' et s'' le type de $(x : U)U'$, on a $< s, s', s'' > \in R$ (l'ensemble de règles du système $\mathcal{T}$) et pour tout sous-terme de t qui est une sorte s, on a $s = Prop$ (et non $Type$).*

Démonstration : Par récurrence sur m.

Proposition 40. *: Soit un contexte Γ bien formé dans le système $\mathcal{T}$, $[\Gamma_1; ...; \Gamma_m]$ une dérivation de Γ et θ la substitution dénotée par cette dérivation. La substitution θ est bien typée dans Γ dans le système $\mathcal{T}$.*

Démonstration : Par récurrence sur la longueur de Γ.

Théorème 3. *: Correction*

Soit un contexte (non quantifié et non contraint) et P un type bien typé dans Γ. Soit le contexte $\Gamma_1 = \Gamma[\exists x : P]$. S'il existe une dérivation de Γ_1, alors il existe une preuve t de P dans Γ dans $\mathcal{T}$. La preuve t est appelée la preuve dénotée par la dérivation.

Démonstration : Soit θ la substitution dénotée par la dérivation de Γ_1. La substitution θ est bien typée dans Γ_1 dans le système $\mathcal{T}$ et $\Gamma_m = \theta\Gamma_1$ est un contexte de succès. Comme le terme P est sans variables existentielles, $\theta P = P$.

Soit $t = \theta x$ et γ le contexte associé à x par θ, $\theta\Gamma = \Gamma$, donc $\theta\Gamma_1 = \Gamma_m = \Gamma\gamma \vdash t : P$. Le contexte Γ_m est un contexte de succès et γ est un contexte existentiel, donc ce contexte est une liste de contraintes reliant des termes identiques, donc $\Gamma \vdash t : P$.

4.4.3 Complétude

Définition 66. *: Taille d'un Terme*

Soit un contexte Γ *et* t *un terme bien typé dans* Γ*. Soit* T *la forme normale du type de* t *dans* Γ*. On définit par récurrence sur l'ordre* $<$*, la taille de* t_Γ *(*$|t_\Gamma|$*) :*

- *Si* t *est une sorte, alors* $|t_\Gamma| = 1$*,*
- *Si* t *est une variable, alors* $|t_\Gamma| = |T_\Gamma|$*,*
- *Si* $t = (u\ v)$*, alors* $|t_\Gamma| = |u_\Gamma| + |v_\Gamma| + |T_\Gamma|$*,*
- *Si* $t = [x : U]u$*, alors* $|t_\Gamma| = |u_{\Gamma[x:U]}|$*,*
- *Si* $t = (x : U)V$*, alors* $|t_\Gamma| = |U_\Gamma| + |V_{\Gamma[x:U]}| + |T_\Gamma|$*.*

Définition 67. *: Taille d'une Substitution*

La taille d'une substitution $\theta = \{< x_i, \gamma_i, t_i >\}$ *où les* t_i *sont normaux est la somme de la taille des* t_i*.*

Proposition 41. *: Si* Γ *est un contexte d'échec, alors pour toute substitution* σ *bien typée dans* Γ*,* $\sigma\Gamma$ *est aussi un contexte d'échec et de ce fait n'est pas un contexte de succès.*

Démonstration : Soit $a = b$ une contrainte de Γ reliant deux termes sans variables existentielles et différents, la contrainte $(\sigma a = \sigma b)$ est la contrainte $(a = b)$ qui est une contrainte reliant deux termes sans variables existentielles et différents.

Lemme 4. *: Soit* Γ *un contexte et* θ *une substitution telle que* $\theta\Gamma$ *soit un contexte de succès. Il existe une dérivation de* Γ *qui dénote la substitution* θ*.*

Démonstration : Par récurrence sur la taille de θ.

Le contexte $\theta\Gamma$ étant un contexte de succès, tous les contextes associés aux variables existentielles de Γ sont des listes de contraintes reliant des termes sans variables existentielles et égaux. On peut donc considérer ces contextes comme vides.

Le contexte Γ n'est pas un contexte d'échec car il existe une substitution θ telle que $\theta\Gamma$ est un contexte de succès. Si Γ est un contexte de succès, alors $[\Gamma]$ est une dérivation de Γ.

Sinon choisissons parmi celles de Γ, une variable existentielle x dont le type est normal et de la forme $(x_1 : P_1)...(x_n : P_n)P$ avec P est atomique rigide et posons $t = \theta x$.

Le type de t est $(x_1 : \theta P_1)...(x_n : \theta P_n)\theta P$. Comme P est atomique rigide, il en est de même de θP. Donc $t = [x_1 : \theta P_1]...[x_n : \theta P_n]u$ avec u terme atomique ou produit (forme η-longue).

- Si $u = (w\ u_1\ ...\ u_p)$, alors soit q le nombre de produits dans le type de w et $r = p - q$. Comme le type de u est atomique $p \geq q$, donc $r \geq 0$. Posons :

$$\sigma = \{< x, \gamma, [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n)) >\}$$

  avec γ défini dans l'algorithme.

Construisons à présent les termes devant être substitués aux variables h_i $(1 \leq i \leq p)$ et H_i et K_i $(1 \leq i \leq r)$. Posons :

$$u'_i = [x_1 : P_1]...[x_n : P_n]u_i$$

Et pour tout i, $1 \leq i \leq r$, $(w\ u_1\ ...\ u_{q+i-1})$ a un type qui est un produit. Soit $(y : U_i)V_i$ ce produit. Posons :

$$U'_i = [x_1 : P_1]...[x_n : P_n]U_i$$

$$V'_i = [x_1 : P_1]...[x_n : P_n][y : U_i]V_i$$

$$\theta' = \theta - \{< x, [\,], \theta x >\} \cup \{< h_i, [\,], u'_i >\} \cup \{< H_i, [\,], U'_i >, < K_i, [\,], V'_i >\}$$

Nous avons $\theta = \theta' \circ \sigma$.
Prouvons à présent que la substitution θ' a une plus petite taille que θ :
$\theta x = [x_1 : \theta P_1]...[x_n : \theta P_n](w\ u_1\ ...\ u_p)$. Soit T_i le type de $(w\ u_1\ ...\ u_i)$.
$|\theta x| = |(w\ u_1\ ...\ u_p)| = |T_0| + |u_1| + |T_1| + ... + |u_p| + |T_p|$
$> |T_q| + ... + |T_{p-1}| + |u_1| + ... + |u_p| = |(y : U_1)V_1| + ... + |(y : U_r)V_r| + |u_1| + ... + |u_p|$
$> |U_1| + ... + |U_r| + ... + |V_1| + ... + |V_r| + |u_1| + ... + |u_p|$
$= |U'_1| + ... + |U'_r| + |V'_1| + ... + |V'_r| + |u'_1| + ... + |u'_p|$.
Donc $|\theta| > |\theta'|$.

— Si maintenant $u = (x : U)V$, alors posons :

$$\sigma = \{< x, \gamma, [x_1 : P_1]...[x_n : P_n](y : (h\ x_1\ ...\ x_n))(k\ x_1\ ...\ x_n\ y) >\}$$

avec γ défini dans l'algorithme.
Construisons les termes qui doivent être substitués aux variables h et k. Posons :

$$U' = [x_1 : P_1]...[x_n : P_n]U$$

$$V' = [x_1 : P_1]...[x_n : P_n][y : U]V$$

$$\theta' = \theta - \{< x, [\,], \theta x >\} \cup \{< h, [\,], U' >, < k, [\,], V' >\}$$

Nous avons $\theta = \theta' \circ \sigma$.
Prouvons à présent que la substitution θ' a une plus petite taille que θ :
$\theta x = [x_1 : \theta P_1]...[x_n : \theta P_n](y : U)V$.
$|\theta x| = |(y : U)V| > |U| + |V| = |U'| + |V'|$.
Donc $|\theta| > |\theta'|$.

Dans les deux cas, par hypothèse de récurrence, il existe une dérivation D de $\sigma\Gamma$ qui dénote la substitution θ' et $[\Gamma]D$ est une dérivation de Γ qui dénote θ.

Théorème 4. *: Complétude*

Soit Γ un contexte (non quantifié et non contraint) et P un type bien typé dans Γ tel qu'il existe un terme t tel que $\Gamma \vdash t : P$. Alors il existe une dérivation de $\Gamma_1 = \Gamma[\exists x : P]$ qui dénote la preuve t.

Démonstration : Soit $\theta = x \leftarrow [\,], t$. On a $\theta\Gamma_1 = \Gamma$, donc il existe une dérivation de Γ_1 qui dénote la substitution θ.

4.5 Améliorations de l'Efficacité

Pour améliorer l'efficacité de la méthode, il faut reconnaître le plus tôt possible les nœuds qui ne peuvent en aucun cas conduire à un nœud de succès pour élaguer l'arbre de recherche.

4.5.1 Vérification Incrémentale des Contraintes

Tout d'abord nous devons vérifier incrémentalement les contraintes. Pour ce faire, nous définissons une fonction de simplification des contraintes très similaire à la fonction SIMPL de [40] [41].

Pour pouvoir définir cette fonction nous modifions une peu la syntaxe des contraintes : une contrainte est à présent un triplet $< \delta, a, b >$ où δ est un contexte qui ne contient que des variables universelles. Si $\Gamma = \Delta[< \delta, a, b >]\Delta'$, alors les termes a et b doivent être bien typés et du même type dans $\Delta\delta$.

Définition 68. *: Simplification*

Tant qu'il y a, dans un contexte Γ*, une contrainte bien typée sans utiliser les contraintes et reliant deux termes rigides et que la simplification n'a pas échoué, on répète le processus consistant à remplacer ce contexte* Γ *par le contexte* Γ' *défini ci-dessous.*

Soit $\delta, a = b$ *une contrainte de* Γ *bien typée sans utiliser les contraintes et reliant deux termes rigides. On écrit* $\Gamma = \Delta[\delta, a = b]\Delta'$*.*

- *Si* a *et* b *sont deux abstractions* $a = [x : T]a'$*,* $b = [x : U]b'$*, comme les termes* a *et* b *ont le même type dans* $\Delta\delta$*, on a* $T = U$*, on pose* $\Gamma' = \Delta[\delta[\forall x : T], a' = b']\Delta'$*.*
- *Si* a *et* b *sont deux produits* $a = (x : T)a'$*,* $b = (x : U)b'$ *et* T *et* U *ont le même type, on pose* $\Gamma' = \Delta[\delta, T = U][\delta[\forall x : T], a' = b']\Delta'$*.*
- *Si* a *et* b *sont deux termes atomiques et ont la même variable de tête appliquée au même nombre de termes,* $a = (w\ u_1\ ...\ u_p)$*,* $b = (w\ t_1\ ...\ t_p)$*, on pose* $\Gamma' = \Delta[\delta, u_1 = t_1]...[\delta, u_p = t_p]\Delta'$*.*
- *Dans les autres cas la simplification échoue.*

Quand la simplification échoue, le nœud ne peut pas mener à une nœud de succès et cette branche de l'arbre peut être élaguée.

4.5.2 Utilisation des Contraintes Flexible-Rigide

Quand on a dans un contexte une contrainte bien typée reliant un terme flexible à un terme rigide, et que la tête du terme flexible est instanciable (c'est-à-dire quand la tête de son type est rigide) il faut instancier cette contrainte en premier, les seuls candidats pour la variable de tête du terme substitué à cette variable sont la variable de tête du terme rigide et les variables liées dans une abstraction [40] [41].

Les autres substitutions mènent à des contraintes reliant deux termes rigides dont la simplification échoue.

4.5.3 Utilisation des Contraintes Flexible-Flexible

Quand on a une contrainte bien typée reliant deux termes flexibles et que l'on peut instancier la variable de tête de l'un des termes, il faut l'instancier en premier. Cette contrainte n'aidera pas à restreindre les substitutions pour cette variable, mais on peut, en la substituant ainsi, la transformer

en une contrainte reliant un terme flexible à un terme rigide. De plus, résoudre une contrainte permet de rendre d'autres contraintes bien typés sans utiliser les contraintes et donc exploitables.

4.5.4 Economie de la Scission

Le terme $(Q\ (h_1\ x_1\ ...\ x_n)...(h_q\ x_1\ ...\ x_n))$ est atomique, s'il est rigide, il est inutile de considérer les solutions avec un degré de scission non nul, car elles mènent à des contraintes reliant des termes rigides dont la simplification échoue.

En particulier dans le $\lambda\Pi$-calcul, on ne considère jamais de variables existentielles de type ou de prédicat, et donc la scission peut toujours être évitée.

4.5.5 Sous-Problèmes Décidables de l'Unification

Quand une contrainte bien typée sans utiliser les contraintes est un problème d'unification décidable (unification du premier ordre, filtrage du deuxième ordre, etc.), on peut la résoudre et appliquer les substitutions ainsi obtenues au contexte Γ. La seconde partie de cette thèse est consacrée à l'étude de ces contraintes.

Dans un système possédant une telle heuristique, l'élimination d'un quantificateur universel du premier ordre est une opération élémentaire alors que l'élimination d'un quantificateur universel à l'ordre supérieur est une opération plus complexe qui peut nécessiter une exploration non déterministe.

4.5.6 Priorité à la Variable la Plus à Droite

Quand on a deux variables existentielles x et y telles que y est déclarée à droite de x et x a une occurrence dans le type de y, si ces deux variables peuvent être instanciées, il faut commencer par le variable la plus à droite. Considérons par exemple le cas où on a un axiome $u : (P\ n)$ et où on cherche un entier x tel que $(P\ x)$. On a deux variables existentielles : $x : Nat$ et $y : (P\ x)$, instancier y par u donnera la contrainte $x = n$ et x sera automatiquement instancié par n alors que si on commence par x on va l'instancier par $0, 1, 2, etc.$ et échouer dans la recherche d'une preuve de $(P\ 0)$, $(P\ 1)$, $(P\ 2)$, etc. avant d'arriver à $x = n$.

4.5.7 Normalisation de Certains Termes Mal Typés

Une amélioration importante de cette méthode consisterait à reconnaître certains termes mal typés mais néanmoins normalisables, ce qui permettrait d'utiliser les contraintes plus efficacement.

Par exemple, on peut montrer que si t est un terme normal et σ une substitution élémentaire de la forme $x \leftarrow [x_1 : P_1]...[x_n : P_n]u$ où u est un produit ou un terme atomique dont la variable de tête est une variable universelle ou une sorte (mais pas l'un des x_i), alors σt est normalisable.

4.6 Extension au Calcul des Constructions avec Univers

Dans une extension de cette méthode au Calcul des Constructions avec Univers, il n'est plus nécessaire de considérer un système de types *Méta* car tous les termes considérés dans cet algorithme sont bien typés dans le Calcul des Constructions avec Univers. Mais un nouveau problème survient : quand nous voulons instancier $x : Prop$ par un produit, il y a une infinité de possibilités $(y : h)(k\ y)$

avec $h : Type_i$ et $k : (y : h)Prop$. Ce problème peut être résolu en utilisant la méthode des univers flottants [43] [34].

Quand on a aussi une règle de cumulativité, un terme bien typé n'a plus nécessairement un type unique (ce problème est également résolu en utilisant la méthode des univers flottants) et les contraintes ne sont plus des égalités mais deviennent des inégalités pour la relation de cumulativité.

4.7 Application à la Synthèse et à l'Evaluation de Programmes

4.7.1 Synthèse de Programmes

Une application possible de la synthèse de preuves est la synthèse de programmes. En effet si $S : T \to U \to Prop$ est une spécification d'un programme prenant en entrée un objet de type T et retournant un objet de type U, d'une preuve constructive de la proposition $\forall x : T.\exists y : U.(S\ x\ y)$ on peut extraire un terme (programme) f de type $T \to U$ tel que $\forall x : T.(S\ x\ (f\ x))$ [53] [54]. En composant un algorithme de synthèse de preuves avec un algorithme d'extraction de programmes on obtient un algorithme de synthèse de programmes à partir de spécifications.

Pour pouvoir extraire des programmes non triviaux, il faut se placer dans un système logique dont le langage des termes est suffisamment riche. Il est donc exclu de se placer au premier ordre ou à l'ordre supérieur, les systèmes des types sont en revanche de bons candidats.

4.7.2 Programmation Logique

Dès le premier ordre il est néanmoins possible d'utiliser la synthèse de preuves pour programmer. Une spécification $S : T \to U \to Prop$ et une valeur d'entrée $t : T$ données, il est possible de synthétiser une preuve constructive de $\exists y : U.(S\ t\ y)$ et d'extraire de cette preuve un terme u tel que $(S\ t\ u)$. Cette méthode est celle utilisée par l'interpréteur Prolog. Le fait qu'un programme Prolog soit en fait une spécification et que l'évaluation de ce programme soit en quelque sorte la synthèse d'un programme d'arité nulle u, explique le décalage entre programmation fonctionnelle et logique [47]. En effet en programmation fonctionnelle, une spécification est une proposition, un programme est une preuve de cette spécification et l'évaluation de ce programme est la normalisation de cette preuve, alors qu'en programmation logique c'est le programme qui est une proposition et son évaluation est la recherche d'une preuve de cette proposition.

4.7.3 Interprétation et Compilation

Le fait qu'au premier ordre, la recherche de preuve ne puisse se faire qu'une fois la valeur d'entrée connue et que à l'ordre supérieur (dans les systèmes de types) cette recherche puisse se faire avant, met en lumière une analogie entre les oppositions premier ordre / ordre supérieur (systèmes de types) et interprétation / compilation. Une méthode de synthèse de programmes fonctionnels à partir de spécifications peut être vue comme une méthode de compilation des programmes logiques en programmes fonctionnels. Cette compilation élimine toute recherche non déterministe à l'exécution, en quelque sorte le programme logique est suffisamment analysé à la compilation pour qu'on soit capable de distinguer en fonction de la valeur d'entrée quelles sont les branches de l'arbre d'évaluation qui sont vouées à l'échec et se diriger directement vers une branche vouée au succès.

La synthèse de preuves dans les systèmes de types semble donc ouvrir des voies prometteuses en programmation logique. Naturellement ces voies sont encore trop floues pour qu'on puisse dire

aujourd'hui si elles sont réalistes ou si la compilation d'un programme logique en un programme fonctionnel déterministe demande la synthèse d'une preuve beaucoup trop complexe. Néanmoins la question mérite d'être posée.

Chapitre 5

Une Restriction Incomplète

5.1 Une Restriction

Revenons à l'exemple que nous avons considéré au chapitre précédent :
$\Delta = [\forall A : Prop; \forall B : Prop; \forall I : Prop \to Prop; \forall u : (P : Prop)((I\ P) \to P); \forall v : (I\ (A \to B));$
$\forall w : A]$
et $\Gamma = \Delta[\exists p : B]$.

Pour obtenir la preuve $(u\ (A \to B)\ v\ w) : B$, nous avons dû appliquer une substitution avec un degré de scission égal à 1 :

$$p \leftarrow (u\ h_1\ h_2\ h_3)$$

où le terme substitué à p a trois applications alors que le type de u ne commence que par deux produits. Comme nous considérons tous les degrés de scission possibles, notre méthode est assez inefficace. On la rend beaucoup plus réaliste en restreignant à des substitutions de degré de scission nul et un nombre d'abstractions compris entre zéro et le nombre de produits dans le type de la variable existentielle substituée. Appelons cet algorithme algorithme avec *scission faible* (par opposition, l'algorithme précédent sera appelé algorithme avec *scission forte*).

L'arbre de recherche de l'algorithme avec scission faible est finiment branchant.

Cette méthode avec scission faible ne peut pas synthétiser la preuve de B ci-dessus. En revanche elle peut synthétiser une preuve de $A \to B$. En effet, soit p' une variable existentielle $p' : A \to B$. Appliquons la substitution :

$$p' \leftarrow (u\ h_1\ h_2)$$

avec $h_1 : Prop$, $h_2 : (I\ h_1)$ (remarquons que bien que le type de p' commence par un produit, le nombre d'abstractions dans $(u\ h_1\ h_2)$ est nul). La contrainte $h_1 = A \to B$ suggère la substitution :

$$h_1 \leftarrow (A \to B)$$

puis pour la variable existentielle $h_2 : (I\ (A \to B))$, on applique la substitution :

$$h_2 \leftarrow v$$

La preuve synthétisée est $(u\ (A \to B)\ v) : (A \to B)$. Remarquons que ce terme n'est pas en forme η-longue, sa forme η-longue est $[w : A](u\ (A \to B)\ v\ w) : (A \to B)$.

5.2 Généralisation de l'Hypothèse de Récurrence

Dans les deux preuves que nous avons considérées (celles de B et $A \to B$), le problème était de remarquer que le terme $u : (P : Prop)((I\ P) \to P)$ devait être appliqué à la proposition $(A \to B)$.

L'algorithme avec scission forte trouve cette proposition dans les deux preuves, le prix payé est l'inefficacité de cet algorithme. L'algorithme avec scission faible ne trouve cette proposition que dans le cas de la preuve de $A \to B$.

Cela signifie que quand on utilise une hypothèse avec un quantificateur sur une proposition ou un prédicat (par exemple un principe de récurrence), l'algorithme avec scission faible ne trouve la proposition à laquelle appliquer cette hypothèse que quand cette proposition *peut être lue* dans le but à prouver et non quand la preuve demande une *généralisation de l'hypothèse de récurrence* (induction loading) [35] (ici de B à $A \to B$).

Cette méthode avec scission faible est similaire à la méthode d'introduction-résolution pour un certain algorithme d'unification. En fait certaines preuves qui ne sont pas synthétisées par introduction-résolution le sont avec scission faible : par exemple dans un contexte qui contient les variables universelles $u : (P : Prop)(P \to A)$ et $v : (B \to C)$, la preuve $(u\ (B \to C)\ [x : C](v\ x)) : A$ est synthétisée par l'algorithme avec scission faible mais pas par la méthode d'introduction-résolution.

Pour obtenir exactement la méthode d'introduction-résolution, il faudrait supprimer totalement la règle de produit.

5.3 Complétude Transitive

Nous allons maintenant montrer que cette méthode avec scission faible bien qu'incomplète est transitivement complète, c'est-à-dire que pour toute preuve t d'une proposition P dans un contexte Γ il existe une suite de lemmes, tels que chacun d'eux peut être démontré par l'algorithme avec scission faible si on admet ceux qui le précèdent et dont le dernier est la proposition P. De plus, en composant les preuves de ces lemmes, on obtient la preuve t. Cette propriété tire son importance du fait qu'elle montre que, sous réserve d'être guidée par un utilisateur qui lui indique les lemmes intermédiaires, la méthode avec scission faible peut construire toutes les preuves. Cette méthode peut donc servir de base à un système d'aide à la mise au point interactive de preuves et à un langage de preuve de haut niveau que nous détaillerons par la suite.

Il nous faut d'abord préciser la notion de lemme. Pour cela nous allons étendre la notion de contextes en lui ajoutant une notion de constante.

Définition 69. *: Constante*

Une constante est un couple de termes $< c, T >$ (noté $c : T$). Un contexte quantifié contraint avec constantes est une liste $\Gamma = [e_1; ...; e_n]$ telle que e_i est une déclaration quantifiée, une contrainte ou une constante.

Définition 70. *: Règles de Typage*

On ajoute les deux règles de typage suivantes qui permettent de manipuler ces constantes :

$$\frac{\Gamma \vdash c : T}{\Gamma[c : T]\ \textit{bien formé}}$$

$$\frac{c : T \in \Gamma}{\Gamma \vdash c : T}$$

Remarque : Dans la définition que nous avons donnée, les constantes n'ont pas de nom. Nous évitons ainsi les problèmes liés à l'expansion des noms de constantes dans les propositions à prouver. En quelque sorte nos constantes sont toujours expansées.

Les règles que nous rajoutons n'apportent rien quant à la syntaxe des termes ou à la possibilité de typer ces termes. Il est en effet très facile de vérifier que si $\Gamma \vdash t : T$ et Γ' est obtenu en effaçant toutes les constantes de Γ, alors Γ' est bien formé et $\Gamma' \vdash t : T$.

Définition 71. *: Algorithme de Synthèse de Preuves avec Lemmes*

Nous étendons les algorithmes de synthèse de preuves développés ci-dessus en permettant d'utiliser les constantes comme les variables dans les substitutions élémentaires.

Par exemple considérons le contexte :
$\Delta = [\forall A : Prop; \forall B : Prop; \forall I : Prop \to Prop; \forall u : (P : Prop)((I\ P) \to P); \forall v : I(A \to B);$
$\forall w : A; (u\ (A \to B)\ v) : A \to B]$
et $\Gamma = \Delta[\exists p : B]$.
L'algorithme avec scission faible engendre une dérivation de Γ en effet en appliquant une substitution élémentaire avec la constante $(u\ (A \to B)\ v) : A \to B$ comme tête :

$$p \leftarrow (u\ (A \to B)\ v\ h)$$

avec $h : A$. On obtient le contexte $\Delta[\exists h : A]$ puis en appliquant une substitution élémentaire avec la variable w comme tête :

$$h \leftarrow w$$

On obtient le contexte de succès Δ.

La preuve dénotée par cette dérivation est $(u\ (A \to B)\ v\ w)$.

Définition 72. *: Clôture Transitive d'un Algorithme de Synthèse de Preuves*

Soit un algorithme de synthèse de preuves, on note $\Gamma \rightsquigarrow t : T$ le jugement indiquant qu'il existe une dérivation de $\Gamma[\exists x : T]$ engendrée par l'algorithme avec scission faible qui dénote la preuve t. On définit inductivement le jugement $\Gamma \hookrightarrow t : T$.

$$\frac{\Gamma \hookrightarrow c_1 : Q_1\ ...\ \Gamma \hookrightarrow c_n : Q_n\ \Gamma[c_1 : Q_1; ...; c_n : Q_n] \rightsquigarrow t : P}{\Gamma \hookrightarrow t : P}$$

Définition 73. *: Complétude Transitive*

Une méthode de synthèse de preuves est dite transitivement complète si $\Gamma \vdash t : T$ (t normal) implique $\Gamma \hookrightarrow t : T$.

Proposition 42. *: Soit Γ un contexte contenant une variable existentielle $y : (x_1 : P_1)...(x_n : P_n)P$. Soit $x : P$ une variable universelle ou une constante déclarées à gauche de y ou l'un des x_i.*

Il existe une suite de contextes engendrée par l'algorithme avec scission faible dont le premier élément est Γ, qui dénote la substitution :

$$y \leftarrow [x_1 : P_1]...[x_n : P_n]x'$$

où x' est la forme η-longue de x.

Démonstration : Par récurrence sur la structure de x'.

Ecrivons $P = (y_1 : Q_1)...(y_p : Q_p)Q$ (Q atomique). On a :

$$y : (x_1 : P_1)...(x_n : P_n)(y_1 : Q_1)...(y_p : Q_p)Q \text{ (Q atomique)}$$

On considère la substitution σ :

$$y \leftarrow [x_1 : P_1]...[x_n : P_n][y_1 : Q_1]...[y_p : Q_p](x\ (h_1\ x_1\ ...\ x_n\ y_1\ ...\ y_p)\ ...\ (h_p\ x_1\ ...\ x_n\ y_1\ ...\ y_p))$$

On obtient un contexte qui contient les variables :

$$\exists h_1 : (x_1 : P_1)...(x_n : P_n)(y_1 : Q_1)...(y_p : Q_p)Q_1$$

$$...$$

$$\exists h_p : (x_1 : P_1)...(x_n : P_n)(y_1 : Q_1)...(y_p : Q_p)Q_p$$

Par hypothèse de récurrence, il existe une suite de contextes dont le premier contexte est $\sigma\Gamma$, engendrée par l'algorithme avec scission faible que dénote la substitution σ_1 :

$$h_1 \leftarrow [x_1 : P_1]...[x_n : P_n][y_1 : Q_1]...[y_p : Q_p]y'_1$$

Puis une dont le premier contexte est $\sigma_1\sigma\Gamma$ qui dénote la substitution σ_2 :

$$h_2 \leftarrow [x_1 : P_1]...[x_n : P_n][y_1 : Q_1]...[y_p : Q_p]y'_2$$

etc., enfin une dont le premier contexte est $\sigma_{p-1}...\sigma_1\sigma\Gamma$ qui dénote la substitution σ_p :

$$h_p \leftarrow [x_1 : P_1]...[x_n : P_n][y_1 : Q_1]...[y_p : Q_p]y'_p$$

La composée de ces substitutions est la substitution :

$$y \leftarrow [x_1 : P_1]...[x_n : P_n][y_1 : Q_1]...[y_p : Q_p](x\ y'_1\ ...\ y'_p)$$

c'est-à-dire :

$$y \leftarrow [x_1 : P_1]...[x_n : P_n]x'$$

Proposition 43. *: La méthode de synthèse de preuves avec scission faible est transitivement complète.*

Démonstration : Soit Γ, t et T tels que $\Gamma \vdash t : T$, montrons par récurrence sur la taille de t que $\Gamma \hookrightarrow t : T$.

• Si $t = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_p)$.

Le terme t est en forme η-longue, donc $(w\ c_1\ ...\ c_p)$ a un type atomique.

Pour tout i posons $d_i = [x_1 : P_1]...[x_n : P_n]c_i$. et D_i le type de d_i dans Γ.

On définit la suite finie $t_1, ..., t_k = t$.

Soit q_1 le nombre de produits dans le type de w, on pose $t_1 = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_{q_1})$. Si $t_1 = t$, on pose $k = 1$. Sinon, soit q_2 le nombre de produits dans le type de $(w\ c_1\ ...\ c_{q_1}))$, on pose $t_2 = [x_1 : P_1]...[x_n : P_n](t_1\ x_1\ ...\ x_n\ c_{q_1+1}\ ...\ c_{q_1+q_2}) = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_{q_1+q_2}))$, etc.

On pose T_i le type de t_i dans Γ. On a $T_k = T$ et d'après la proposition précédente :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_k : T_k] \rightsquigarrow t_k : T_k$$

c'est-à-dire :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_k : T_k] \rightsquigarrow t : T$$

Montrons maintenant par récurrence descendante sur i que pour tout i, $0 \leq i \leq k$

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_i : T_i] \rightsquigarrow t : T$$

Nous avons déjà prouvé cette proposition pour $i = k$. Admettons-la pour i :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_i : T_i] \hookrightarrow t : T$$

On a :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_{i-1} : T_{i-1}] \rightsquigarrow t_i : T_i$$

En effet, considérons le contexte :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_{i-1} : T_{i-1}; \exists h : T_i]$$

On applique à h la substitution élémentaire de tête t_{i-1} et avec n abstractions :

$$h \leftarrow [x_1 : P_1]...[x_n : P_n](t_{i-1}\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_n\ x_1\ ...\ x_n)\ (k_1\ x_1\ ...\ x_n)\ ...\ (k_{q_i}\ x_1\ ...\ x_n))$$

Puis d'après la proposition précédente, il existe une suite de contextes engendrée par l'algorithme avec scission faible telle que la substitution dénotée par cette suite instancie les h_j par le terme $[x_1 : P_1]...[x_n : P_n]x'_j$ et les k_j par le terme $d'_{q_1+...+q_{i-1}+j}$. La substitution dénotée par cette suite est donc :

$$h \leftarrow [x_1 : P_1]...[x_n : P_n](t_{i-1}\ x'_1\ ...\ x'_n\ c_{q_1+...+q_{i-1}+1}\ ...\ c_{q_1+...+q_{i-1}+q_i})$$

c'est-à-dire :

$$h \leftarrow t_i$$

Donc :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_{i-1} : T_{i-1}] \rightsquigarrow t_i : T_i$$

Comme par ailleurs on sait que :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_i : T_i] \hookrightarrow t : T$$

On en déduit :

$$\Gamma[d_1 : D_1; ...; d_p : D_p; t_1 : T_1; ...; t_{i-1} : T_{i-1}] \hookrightarrow t : T$$

On a donc :

$$\Gamma[d_1 : D_1; ...; d_p : D_p] \hookrightarrow t : T$$

Par hypothèse de récurrence $\Gamma \hookrightarrow d_i : D_i$, donc :

$$\Gamma \hookrightarrow t : T$$

- Si $t = [x_1 : P_1]...[x_n : P_n](z : U)V$, alors on pose :

$$u = [x_1 : P_1]...[x_n : P_n]U$$

$$v = [x_1 : P_1]...[x_n : P_n][z : U]V$$

Par hypothèse de récurrence :

$$\Gamma \hookrightarrow u : (x_1 : P_1)...(x_n : P_n)s$$

et :

$$\Gamma \hookrightarrow v : (x_1 : P_1)...(x_n : P_n)(z : U)s'$$

en appliquant la règle de produit et la proposition précédente :

$$\Gamma[u : (x_1 : P_1)...(x_n : P_n)s; v : (x_1 : P_1)...(x_n : P_n)(z : U)s'] \rightsquigarrow t : T$$

donc :

$$\Gamma \hookrightarrow t : T$$

- Si $t = [x_1 : P_1]...[x_n : P_n]s$ (où s est une sorte), on a :

$$\Gamma \rightsquigarrow t : T$$

donc :

$$\Gamma \hookrightarrow t : T$$

5.4 Accès aux Axiomes

Dans la section précédente nous avons vu que quand on a un axiome ou un lemme $x : T$ dans le contexte Γ, prouver cette proposition peut demander plusieurs étapes en fonction de la forme η-longue de x. Dans certaines applications cela est ennuyeux. On peut donc ajouter une règle à notre méthode : la règle *d'accès aux axiomes* . Quand on instancie une variable existentielle de type $(x_1 : P_1)...(x_n : P_n)P$ (P atomique), on considère toutes les substitutions $[x_1 : P_1]...[x_k : P_k]x'$ où x est une variable ou une constante du contexte de type $(x_{k+1} : P_{k+1})...(x_n : P_n)P$.

Définition 74. *: Algorithme avec Scission Faible, Accès aux Axiomes et Nombre d'Itérations Borné par Deux*

On considère l'algorithme avec scission faible et accès aux axiomes et on note $\Gamma \rightsquigarrow_2 t : P$ *le jugement indiquant qu'il existe une dérivation de* $\Gamma[\exists x : P]$ *qui dénote la preuve* t*, dans laquelle on instancie* x *par un terme introduisant éventuellement des variables existentielles* $h_1, ..., h_p$*, puis on instancie les variables* $h_1, ..., h_p$ *par des termes n'introduisant pas de nouvelles variables existentielles. (On dit dans ce cas que le nombre d'itérations est borné par deux.)*

Proposition 44. *: L'algorithme avec scission faible, accès aux axiomes et nombre d'itérations borné par deux termine toujours et est transitivement complet.*

Démonstration : Pour la terminaison, il suffit de remarquer que l'arbre de recherche est finiment branchant et ses branches finies car en associant au nœud initial l'ordinal ω et aux autres leur nombre de variables existentielles, on obtient une suite d'ordinaux strictement décroissante le long d'une branche.

Pour la complétude transitive, on reprend la démonstration ci-dessus, en remarquant que grâce à la règle d'accès aux axiomes toutes les dérivations considérées ont un nombre d'itérations borné par deux.

5.5 Synthèse Interactive de Preuves

5.5.1 Tactiques

Nous allons maintenant utiliser l'algorithme avec scission faible pour mettre en œuvre un système d'aide à la mise au point de preuves par tactiques, dans l'esprit du système LCF [32].

Dans ce système, l'utilisateur déclare d'abord un contexte et l'énoncé d'un théorème. Cet énoncé est compris par la machine comme le type d'une nouvelle variable existentielle. Puis il donne des instructions à la machine. Ces instructions sont de deux sortes : appliquer une substitution élémentaire de l'algorithme avec scission faible et accès aux axiomes ou déclarer l'énoncé d'un lemme pour ce théorème. A chaque étape, la machine simplifie les contraintes, résout les contraintes décidables qui ont une solution unique (par exemple en s'appuyant sur [57]) et affiche la liste des types des variables existentielles du contexte.

Dans ce système, quand on veut prouver une proposition $A \to B$, il n'est pas possible d'ajouter au contexte une proposition A et de chercher à prouver B. En effet cette transformation n'est pas l'application d'une substitution au contexte courant. Il est néanmoins beaucoup plus pratique de procéder ainsi que d'aller jusqu'à la forme normale de tête de la preuve et de marquer explicitement la dépendance des but intermédiaires par rapport aux hypothèses comme nous l'avons fait.

Une façon de procéder est d'introduire un mécanisme de sections et de portées des identificateurs dans le contexte [4] [5]. Grossièrement, on peut dire que quand on a une variable existentielle de type $y : A \to B$ et que l'on introduit l'hypothèse A, localement l'hypothèse A est visible et y est de type B, mais du point de vue des autres objets du contexte, l'hypothèse A n'est pas visible et y a toujours le type $A \to B$.

Du fait de la présence de cette règle d'introduction, la règle de substitution peut se limiter à des termes sans abstraction.

On obtient alors un système très proche de ceux qui comportent les tactiques introduction, résolution, produit et accès aux axiomes. Un progrès qui apparaît dans ce système par rapport à ces systèmes est qu'il est à tout moment possible à l'utilisateur d'interrompre sa preuve, pour introduire un lemme ou une hypothèse. Egalement, la tactique de résolution n'est pas limitée par un algorithme d'unification (du premier ordre par exemple) mais elle engendre une contrainte arbitrairement complexe. Deux cas peuvent alors se produire : si la contrainte engendrée appartient à une classe que l'on sait résoudre, alors elle est résolue, sinon elle reste dans le contexte et l'utilisateur peut, en quelque sorte, guider la machine dans l'arbre d'unification de la même façon qu'il la guide dans l'arbre de synthèse de preuves.

5.5.2 Un Exemple Simple

Voici un exemple des instructions données à la machine pour montrer que tout booléen est égal à $True$ ou à $False$ dans un prototype d'un tel système. Ce prototype comporte également la possibilité de définir des constantes nommées, c'est-à-dire qu'après une définition $a := t$ où a est un symbole et t un terme bien typé dans le contexte courant, on peut utiliser le symbole a et tout se passe comme si ce symbole était remplacé par le terme t à chacune de ses occurrences.

Déclarons tout d'abord le type des booléens et les deux variables $True$ et $False$:

```
Parameter Bool:Prop. (* Prop = Set *)

Parameter True:Bool.

Parameter False:Bool.
```

Pour exprimer le fait que le type $Bool$ ne contient rien d'autre que ces deux éléments, déclarons un axiome de récurrence sur les booléens :

```
Axiom RecB.
Assumes (P:Bool -> Prop)((P True) -> (P False) -> (x:Bool)(P x)).
```

Déclarons ensuite une relation (d'égalité) réflexive sur ce type :

```
Parameter Eq: (Bool -> Bool -> Prop).

Axiom Refl.
Assumes (x:Bool)(Eq x x).
```

Nous pouvons maintenant démontrer notre théorème :

```
Theorem True_or_False.
   Variable b:Bool.
Statement ((Eq b True)\/(Eq b False)).
Resolution RecB.
Go x1.
Local P = [x:Bool]((Eq x True)\/(Eq x False)).
Assumption P.
Resolution or_intro2.
Resolution Refl.
Resolution or_intro1.
Resolution Refl.
```

Lors de l'application de la tactique `Resolution RecB.` la contrainte engendrée n'est pas une contrainte triviale. La machine affiche donc quatre buts `x1`, `x2`, `x3` et `x4`. Ainsi que la contrainte :

```
(x1 x4) == (Eq b True)\/(Eq b False).
```

en fait, dans cette contrainte, la définition de la disjonction est développée (voir chapitre 1) et elle s'écrit donc :

```
            (x1 x4) == (C:Prop)((Eq b True)->C)->((Eq b False)->C)->C.

 <<< Begin Section True_or_False
*** [b :Bool]
 <<< Begin Section x1
*** [?x1 :Bool->Prop]
 >>> End Section x1
 <<< Begin Section x2
*** [?x2 :(x1 True)]
 >>> End Section x2
 <<< Begin Section x3
*** [?x3 :(x1 False)]
 >>> End Section x3
 <<< Begin Section x4
============================================================
*** [?x4 :Bool]
 >>> End Section x4
[[  ]](x1 x4) == (C:Prop)((Eq b True)->C)->((Eq b False)->C)->C
True_or_False : (Eq b True)\/(Eq b False)
 >>> End Section True_or_False
```

La double ligne indique que le but considéré par défaut est `x4`. On change de but par l'instruction `Go x1`. Puis on définit une constante locale `P` et on instancie `x1` par `P`. Par cette instanciation la contrainte devient, après simplification, `x4 = b`. Elle est maintenant triviale est donc `x4` est automatiquement instanciée. Les buts restant à prouver sont donc :

```
x2:((Eq True True)\/(Eq True False))
```

et :

```
x3:((Eq False True)\/(Eq False False)).
```

Pour `x3` par exemple, on applique la tactique `Resolution or_intro2` qui nous ramène à la proposition `(Eq False False)` qui est prouvée par `Resolution Refl`. Remarquons que dans ces deux cas la contrainte engendrée est triviale et est donc résolue automatiquement par la machine.

5.5.3 Un Autre Exemple : Un Lemme du Théorème de Tarski

Déclarons tout d'abord un type T, une relation d'égalité Eq et une relation R antisymétrique et transitive.

```
Parameter T:Prop. (* Prop = Set *)

Parameter Eq:T->T->Prop.

Parameter R:T->T->Prop.
```

```
Axiom Antisym.
Assumes (x:T)(y:T)(R x y) -> (R y x) -> (Eq x y).

Axiom Trans.
Assumes (x:T)(y:T)(z:T)(R x y) -> (R y z) -> (R x z).
```

Déclarons ensuite une fonction croissante de T dans T et définissons l'ensemble des pré-points-fixes.

```
Parameter f:T->T.

Axiom Incr.
Assumes (x:T)(y:T)(R x y) -> (R (f x) (f y)).

Definition Pre = [x:T](R x (f x)).
```

Déclarons ensuite un élément de T et supposons que c'est la borne supérieure de l'ensemble des pré-points-fixes.

```
Parameter M:T.

Axiom Up.
Assumes (x:T) (Pre x) -> (R x M).

Axiom Least.
Assumes (y:T)((x:T)(Pre x)->(R x y))->(R M y).
```

Démontrons maintenant que M est un point fixe de f.

```
Theorem Tarski.

Statement (Eq M (f M)).

Resolution Antisym.
Resolution Up.
Resolution Incr.
Resolution Least.
Intro.
Intro.  (* x13 *)
Resolution Trans.
Resolution Incr.
Resolution Up.
Resolution x13.
Resolution x13.
Resolution Least.
Intro.
```

```
Intro. (* x27 *)
Resolution Trans.
Resolution Incr.
Resolution Up.
Resolution x27.
Resolution x27.
```

5.6 Prolégomènes à un Vernaculaire Mathématique

Dans l'exemple ci-dessus, on peut introduire des lemmes (c'est-à-dire des constantes) et rendre ainsi la preuve de chaque proposition beaucoup plus courte.

```
Theorem Tarski.

Statement (Eq M (f M)).

      Remark One.

         Variable y:T.

         Hypothesis v.
         Assumes (Pre y).

      Statement (R y (f M)).

         Remark Rem.
         Statement (R y M).
         Resolution Up. Resolution v.

         Remark Rem'.
         Statement (R (f y) (f M)).
         Resolution Incr. Resolution Rem.

      Resolution Trans. Resolution Rem'. Resolution v.

      Remark Two.
      Statement (R M (f M)).
      Resolution Least. Assumption One.

      Remark Three.
      Statement (R (f M) (f (f M))).
      Resolution Incr. Resolution Two.

      Remark Four.
      Statement (R (f M) M).
```

```
    Resolution Up. Resolution Three.

Resolution Antisym. Resolution Four. Resolution Two.
```

Cette façon d'écrire une preuve, contrairement à la précédente, au terme-preuve ou à la preuve en déduction naturelle, commence à ressembler à une preuve du théorème de Tarski telle qu'on peut la lire dans un livre de théorie des treillis.

Après de Bruijn, on appelle *vernaculaire mathématique* [4] [5] le langage dans lequel les démonstrations mathématiques sont écrites de manière naturelle par les mathématiciens. Une application de la démonstration automatique est la conception de langages formels aussi proches que possible du vernaculaire mathématique. En effet, en vernaculaire, une démonstration se présente souvent comme une liste de propositions telle que chaque proposition est *évidente* quand les précédentes sont connues. Le lecteur doit donc pour vérifier la correction de l'argumentation retrouver les petites preuves qui relient les propositions les unes aux autres. Pour qu'une machine également puisse vérifier la correction d'une telle argumentation, elle doit disposer d'un algorithme de synthèse de preuves. On écrira dans un tel formalisme la démonstration comme ci-dessus sauf que l'on remplacera les indications `Intro`, `Resolution` et `Assumption` par l'appel à un tel algorithme de synthèse de preuves auquel on indiquera, au plus, celles parmi les propositions précédentes qui peuvent être utilisées.

Pour que ce formalisme soit complet et que la correction d'une argumentation soit décidable, l'algorithme de synthèse de preuves doit être transitivement complet et toujours terminer. L'algorithme avec scission faible, accès aux axiomes et nombre d'itérations borné a ces propriétés.

Seconde partie
Résolution d'Equations

Chapitre 6

Unification

6.1 Unification Ouverte et Fermée

Définition 75. *: Problème d'Unification*
Un problème d'unification est un triplet $< \Gamma, a, b >$ *tel que* Γ *est un contexte quantifié (mais non contraint) bien formé et* a *et* b *deux termes normaux bien typés dans* Γ*.*

Définition 76. *: Solution (ouverte) d'un Problème d'Unification*
Soit $< \Gamma, a, b >$ *un problème d'unification, une solution (ouverte) de* $< \Gamma, a, b >$ *est une substitution* θ *bien typée dans* Γ*, telle que pour toute variable existentielle* x *de* Γ *le contexte associé à* x *par* θ *ne comporte pas de contraintes (autres que triviales) et telle que les formes normales de* θa *et* θb *soient identiques. L'ensemble des solutions de* $< \Gamma, a, b >$ *est noté* $U < \Gamma, a, b >$*.*

Au premier ordre et à l'ordre supérieur, la substitution θ n'est pas supposée instancier toutes les variables existentielles de Γ, elle peut même en introduire de nouvelles. Cette tolérance est due au fait qu'aussi bien au premier ordre qu'à l'ordre supérieur, tous les types sont supposés habités, donc l'existence d'un unificateur θ et d'un unificateur θ' tel que $\theta'\Gamma$ soit un contexte de succès est équivalente. Ce n'est plus le cas dans les systèmes de types où un type peut être habité ou vide. Ce qui motive la définition :

Définition 77. *: Solution Fermée d'un Problème d'Unification*
Soit $< \Gamma, a, b >$ *un problème d'unification, une solution fermée de* $< \Gamma, a, b >$ *est une solution ouverte de* $< \Gamma, a, b >$ *telle que* $\theta\Gamma$ *soit un contexte de succès.*

Exemple : Dans le contexte $\Gamma = [\exists x : (P : Prop)P]$ l'équation $x = x$ a une solution ouverte (l'identité) mais pas de solution fermée.

Remarque : Dans le problème de l'unification dans un contexte quantifié dans le λ-calcul simplement typé [51], les types ne sont pas tous habités, les notions d'unifiabilité ouverte et fermée ne sont donc plus équivalentes. La notion d'unifiabilité considérée est celle d'unifiabilité fermée.

6.2 Une Méthode d'Unification Fermée

Proposition 45. *: Soit* $< \Gamma, a, b >$ *un problème d'unification et* θ *une substitution,* θ *est un unificateur fermé de* $< \Gamma, a, b >$ *si et seulement si* $\theta(\Gamma[a = b])$ *est un contexte de succès.*

Démonstration : De façon évidente si θ est un unificateur de $< \Gamma, a, b >$, $\theta(\Gamma[a = b])$ est un contexte de succès. Réciproquement si $\theta(\Gamma[a = b])$ est un contexte de succès, alors θ est bien typée dans Γ et pour toute variable existentielle x de Γ le contexte associé à x par θ est un sous-contexte de $\theta(\Gamma[a = b])$ et donc ne comporte pas de contraintes autres que triviales, $\theta a = \theta b$ et $\theta\Gamma$ est un contexte de succès.

Définition 78. *: Méthode d'Unification Fermée*

Une méthode de semi-décision d'existence d'un unificateur fermé pour un problème $< \Gamma, a, b >$ consiste à appliquer la méthode développée au chapitre 4 au contexte $\Gamma[a = b]$. De plus, cet algorithme énumère tous les unificateurs fermés.

Remarque : Le lemme qui permet dans le cas de l'unification dans le λ-calcul simplement typé [40] [41] de résoudre trivialement les équations entre termes flexibles ne se généralise pas aux systèmes de types et une équation entre termes flexibles peut ne pas avoir de solution (fermée) puisque les variables de tête des équations peuvent avoir des types vides. De plus, il est montré dans [51] que dans le λ-calcul simplement typé, le problème de l'unification entre termes flexibles dans un contexte quantifié est indécidable. Cette preuve se généralise aisément aux systèmes de types.

6.3 Vers une Méthode d'Unification Ouverte

L'intérêt d'un algorithme d'unification ouverte n'est pas immédiat, car dans les algorithmes de synthèse de preuves (ou plus généralement dans les algorithmes qui utilisent l'unification) une variable existentielle est introduite pour être instanciée et non pour demeurer indéfiniment.

Si on cherche malgré tout un tel algorithme, on doit remarquer que le lemme qui permet dans le cas de l'unification dans le λ-calcul simplement typé [40] [41] de résoudre de façon immédiate les équations entre termes flexibles ne se généralise pas aux systèmes de types polymorphes.

Proposition 46. *: Dans un système polymorphe, une équation entre termes flexibles peut ne pas avoir de solution ouverte.*

Démonstration : Considérons le problème :

$$\Gamma = [\forall A : Prop; \forall B : Prop; \forall u : A; \exists x : (P : Prop)P]$$

$$a = (x\ (A \to B)\ u) \qquad b = (x\ B)$$

Les termes a et b sont flexibles. Pourtant le problème $< \Gamma, a, b >$ n'a pas de solution ouverte. En effet, supposons qu'il en ait une θ et posons $t = \theta x$. On a $t : (P : Prop)P$, donc $t = [P : Prop]t'$ où t' est de type P. Le terme t' n'est donc ni une abstraction ni un produit. C'est un terme atomique $(f\ c_1\ ...\ c_n)$. Si on avait $f = P$, on aurait $n = 0$ et $t' = P$, ce qui est impossible pour des raisons de type, donc $f \neq P$. Comme $(t'[P \leftarrow (A \to B)]\ u) = t'[P \leftarrow B]$, on a :

$$(f\ c_1[P \leftarrow (A \to B)]\ ...\ c_n[P \leftarrow (A \to B)]\ u) = (f\ c_1[P \leftarrow B]\ ...\ c_n[P \leftarrow B])$$

Donc $(f\ c_1[P \leftarrow (A \to B)]) = f$, ce qui est impossible puisqu'une variable ne peut être une application.

Il semble difficile de décrire toutes les solutions des équations entre termes flexibles car la variable de tête d'une substitution élémentaire peut, a priori, être non seulement n'importe quelle variable du

contexte ou n'importe quelle variable liée, mais aussi n'importe quelle nouvelle variable. Il faudrait donc utiliser un algorithme de synthèse de preuves similaire à celui développé précédemment mais qui énumère toutes les preuves d'une proposition *sous tous les ensembles d'axiomes possibles*. Un premier problème est donc de reconnaître les équations entre termes flexibles qui ont des solutions et celles qui n'en n'ont pas et de garder les premières comme contraintes comme on le fait dans l'unification dans le λ-calcul simplement typé. Dans ce cas, il n'est plus immédiat que les termes considérés ont toujours une forme normale de tête. Il faut donc considérer la réduction d'un radical de tête comme une étape de l'algorithme. Egalement des substitutions-produits doivent être appliquées quand les variables de têtes des termes flexibles des équations (entre termes flexibles et entre un terme flexible et un terme rigide) ont une variable de tête existentielle.

Dans l'état actuel de nos connaissances, l'unification ouverte nous apparaît donc être un problème plutôt ardu et qui présente peu d'intérêt.

Chapitre 7

Equations Décidables

Dans le λ-calcul simplement typé, deux classes de problèmes sont connus comme décidables. L'unification du premier ordre [59] et le filtrage du deuxième ordre [41] [45].

L'unification du premier ordre a été récemment généralisée par Miller à l'*unification entre termes à arguments restreints* [51] [52], puis par Pfenning aux systèmes du cube [57]. Nous généralisons ici le filtrage du deuxième ordre aux systèmes du cube.

Un exemple d'application de ces algorithmes est la résolution automatique de certaines des contraintes dans un système d'aide à la mise au point de preuves par tactiques, tel que celui présenté au chapitre 5. L'unification du premier ordre (et entre termes à arguments restreints) est celle qui est effectivement utilisée dans ce prototype. Elle consiste après simplification à reconnaître les contraintes du type $x = t$ où x est une variable existentielle et à instancier x par t. (Dans le cas de l'unification entre termes à arguments restreints, on reconnaît aussi les contraintes $(x\ t_1\ ...\ t_n) = t$ où x est une variable existentielle et les t_i sont des variables universelles distinctes déclarées à droite de x dans Γ.)

Un exemple d'application du filtrage du second ordre est le cas de l'utilisation d'un axiome de récurrence avec un but fermé comme dans le cas considéré au paragraphe 5.5.2. Néanmoins cette possibilité d'application doit être modérée par le fait que la contrainte engendrée est un problème de filtrage du second ordre uniquement quand les types de données sont déclarés comme des variables de types (comme *Bool* dans l'exemple du chapitre 5) et leurs propriétés axiomatisées. Dès que les types de données sont définis, comme dans le cas des entiers de Church pour les entiers (où $Nat = (P : Prop)(P \to (P \to P) \to P)$) ou des projections pour les booléens (où $Bool = (P : Prop)(P \to P \to P)$), les contraintes obtenues ne sont plus du second ordre. En fait nous verrons au chapitre suivant que l'impératif de décidabilté du filtrage est incompatible avec celui de la richesse calculatoire du système considéré, ce qui suggère la nécessité d'un compromis.

7.1 Ordre d'un Type

Dans le λ-calcul simplement typé, on classe les types selon leur ordre de fonctionnalité. Les types du premier ordre sont ceux qui ne sont pas fonctionnels, c'est-à-dire les types atomiques. Les types du deuxième ordre sont les types des fonctions qui s'appliquent aux termes dont le type est du premier ordre, les types du troisième ordre sont les types des fonctionnelles qui s'appliquent à des termes dont le type est du deuxième ordre, etc. L'ordre d'un type est défini par $o(T) = 1$ si T

est atomique et $o(T \to U) = max\{1 + o(T), o(U)\}$.

Dans les systèmes polymorphes, se pose le problème de l'ordre du type $(P : Prop)(P \to P)$. En effet, en appliquant un terme de ce type on peut obtenir un terme dont le type a un ordre arbitrairement élevé. On prendra donc par convention l'ordre de ce type égal à l'infini. De même si X est une variable existentielle de type $Prop$, X peut être instanciée par un type d'ordre arbitraire, on prendra l'ordre de X égal à l'infini.

Définition 79. *: Ordre d'un Type*

Soit Γ un contexte et T un type normal dans Γ. L'ordre de T dans Γ est défini comme l'élément suivant de $N \cup \{\infty\}$:

- *si $T = (y : U)V$, alors $o(T) = max\{1 + u, v\}$ où u est l'ordre de U dans le contexte Γ et v l'ordre de V dans le contexte $\Gamma[\exists y : U]$,*
- *si $T = (x\ t_1\ ...\ t_n)$, alors si x est une variable universelle de Γ, alors $o(T) = 1$, si x est une variable existentielle de Γ, alors $o(T) = \infty$ et si x est une sorte, alors $o(T) = 2$.*

Remarque : On prend la convention habituelle $n + \infty = \infty$ et $max\{n, \infty\} = \infty$.

Proposition 47. *: Soit Γ un contexte, T un type normal dans Γ et σ une substitution bien typée dans Γ, alors σT a un ordre dans $\sigma\Gamma$ inférieur ou égal à $o(T)$.*

Démonstration : Par récurrence sur la structure de T.

- Si $T = (y : U)V$, alors soit u l'ordre de σU dans le contexte $\sigma\Gamma$ et v l'ordre de σV dans le contexte $(\sigma\Gamma)[\exists y : \sigma U] = \sigma(\Gamma[\exists y : U])$, par hypothèse de récurrence $u \leq o(U)$ et $v \leq o(V)$ et donc $o(\sigma T) \leq o(T)$,
- Si $T = (w\ t_1\ ...\ t_n)$, alors :
  - si w est universelle dans Γ, alors elle n'est pas instanciée par σ, donc $\sigma T = (w\ \sigma t_1\ ...\ \sigma t_n)$ et $o(\sigma T) = o(T) = 1$,
  - si w est une sorte, alors $o(\sigma T) = o(T) = 2$,
  - si w est existentielle dans Γ, alors $o(T) = \infty$ donc $o(\sigma T) \leq o(T)$.

Définition 80. *: Ordre d'un Problème d'Unification*

On dit qu'un problème d'unification $< \Gamma, a, b >$ est d'ordre n si le type de toutes les variables existentielles de Γ est au plus d'ordre n.

7.2 Unification du Premier Ordre et Entre Termes à Arguments Restreints

L'unification du premier ordre [59] a été récemment généralisée par Miller à l'*unification entre termes à arguments restreints* [51] [52]. L'idée de cette généralisation est la suivante : considérons un problème d'unification du premier ordre sous un contexte quantifié $< \Gamma, a, b >$, imaginons que nous ayons dans Γ une variable existentielle $x : T$ et juste à gauche de cette variable existentielle une variable universelle $c : U$. La variable c peut apparaître dans un terme substitué à x. On peut marquer explicitement cette dépendance en antiskolémisant cette variable, c'est-à-dire en écrivant $x = (f\ c)$ où $f : U \to T$ est une variable déclarée juste à droite de c, et donc qui ne dépend pas de c. En remplaçant x par $(f\ c)$ dans $a = b$ on obtient une équation $a' = b'$ qui n'est plus un problème du premier ordre car $f : U \to T$ peut avoir un ordre arbitraire. Malgré tout, résoudre ce problème

ne devrait pas être plus difficile que résoudre le problème initial puisque les solutions de celui-là sont les antiskolémisées des solutions de celui-ci.

Remarquons que dans le problème antiskolémisé la variable f ne peut apparaître que dans le terme $(f\ c)$, c'est-à-dire que ses arguments sont toujours des variables universelles déclarées à sa droite. Cela motive la définition :

Définition 81. *: Terme à Arguments Restreints*

Soit Γ un contexte quantifié et t un terme bien typé dans Γ. Le terme t est dit à arguments restreints si et seulement si tout sous-terme u de t qui a la forme $(x\ t_1\ ...\ t_n)$ où x est une variable existentielle est tel que les t_i sont des termes η-équivalents à des variables universelles distinctes et déclarées à droite de x dans Γ.

Remarque : La définition originale est un peu plus générale, elle impose uniquement aux t_i d'être η-équivalents à des termes atomiques dont les variables de tête sont universelles, distinctes et déclarées à droite de x

Théorème 5. *: Décidabilité de l'Unification entre Termes à Arguments Restreints*

L'unification entre termes à arguments restreints est décidable et quand un problème a une solution, il a une solution principale.

Démonstration : Voir [51] [52].

Ce théorème a été généralisé par Pfenning aux systèmes du cube [57].

7.3 Filtrage du Deuxième Ordre dans le λ-calcul Simplement Typé

Le second problème décidable dans le λ-calcul simplement typé est le filtrage du deuxième ordre [41] [45].

Définition 82. *: Problème de Filtrage*

On dit qu'un problème d'unification $< \Gamma, a, b >$ est un problème de filtrage si le terme b est fermé dans Γ.

Dans [41] [45] les variables existentielles sont du deuxième ordre au plus et les variables universelles du troisième ordre au plus. On généralise ici ce problème de trois manières : (1) on considère un calcul quelconque du cube, (2) l'ordre des variables universelles est arbitraire, (3) le terme ouvert peut être non seulement du deuxième ordre mais aussi un terme *à arguments restreints du deuxième ordre* dans l'esprit de Miller et Pfenning. Outre son intérêt propre, la troisième généralisation est nécessaire pour atteindre les deux premières.

7.3.1 Avec Variables Universelles du Troisième Ordre au Plus

Commençons par nous placer dans le λ-calcul simplement typé. La méthode donnée dans [41] [45] consiste à appliquer l'algorithme d'unification à l'ordre supérieur de [40] [41] qui termine si le problème considéré est un problème de filtrage du deuxième ordre.

Proposition 48. *: L'algorithme d'unification à l'ordre supérieur [40] [41] termine si le problème considéré est un problème de filtrage du deuxième ordre.*

Démonstration : Le couple $< \lambda', \lambda >$ où λ est la somme des tailles des termes de gauche et λ' la somme des tailles des termes de droite décroît strictement quand on applique une substitution élémentaire. Considérons une équation $(f\ t_1\ ...\ t_n) = (F\ u_1\ ...\ u_p)$ avec f existentielle et F universelle. Dans une substitution élémentaire, on instancie f par un terme de la forme $[x_1 : T_1]...[x_n : T_n](w\ (h_1\ x_1\ ...\ x_n)...(h_q\ x_1\ ...\ x_n))$. La variable w ne peut être que F ou l'un des x_i.

Si $w = F$ (imitation), le terme de gauche substitué commence par la variable F, l'équation se simplifie en des équations dont la somme des tailles des membres de droite est plus petite que la taille de $(F\ u_1\ ...\ u_p)$. Si $w = x_i$ (projection), comme f a un type au plus du deuxième ordre, x_i a un type du premier ordre. On a donc $q = 0$ et la substitution est $f \leftarrow [x_1 : T_1]...[x_n : T_n]x_i$. Appliquer cette substitution fait diminuer la taille du terme de gauche en laissant le terme de droite inchangé.

Remarque : Il est aussi possible de prendre pour λ le nombre de variables existentielles du problème.

7.3.2 Avec Variables Universelles Arbitraires

Dans [41] [45], les variables universelles sont au plus du troisième ordre car si on autorisait des variables universelles d'ordre plus grand, une substitution élémentaire d'imitation pourrait introduire des variables existentielles d'ordre strictement supérieur à deux. Par exemple dans le contexte :

$$[\forall T : Prop; \forall F : ((T \to T) \to T) \to T; \forall a : T; \exists f : T \to T]$$

on considère le problème :

$$(f\ a) = (F\ ([x : T \to T](x\ a)))$$

la substitution d'imitation est :

$$f \leftarrow [x : T](F\ (h\ x))$$

avec $h : T \to (T \to T) \to T$ qui est une variable du troisième ordre.

Malgré tout, quand on applique cette substitution, on obtient le problème :

$$(F\ (h\ a)) = (F\ ([x : T \to T](x\ a)))$$

La forme η-longue de ce problème est :

$$(F\ ([x : T \to T](h\ a\ [y : T](x\ y)))) = (F\ ([x : T \to T](x\ a)))$$

Et sa forme simplifiée :

$$(h\ a\ [y : T](x\ y)) = (x\ a)$$

Dans ce problème, la variable h est appliquée à deux termes : a qui était déjà un argument de f et qui de ce fait est du premier ordre et $[y : T](x\ y)$ introduit lors de la η-expansion qui est donc η-équivalent à une variable universelle.

On généralise donc ainsi la notion de termes à arguments restreints :

Définition 83. *: Terme à Arguments Restreints du Deuxième Ordre*

Un terme t est dit à arguments restreints du deuxième ordre si tout sous-terme de t de la forme $u = (x\ t_1\ ...\ t_n)$ avec x existentielle, est tel que pour tout i, t_i est un terme du premier ordre ou bien un terme η-équivalent à une variable universelle.

De même appelons problème de filtrage *à arguments restreints du deuxième ordre* un problème dont l'un des termes est à arguments restreints du deuxième ordre et l'autre fermé.

Toute substitution élémentaire appliquée à un problème à arguments restreints du deuxième ordre donne un problème à arguments restreints du deuxième ordre.

L'argument de terminaison de [41] [45] se généralise aux problèmes à arguments restreints du deuxième ordre. En effet, les imitations et les projections sur les termes η-équivalents à une variable universelle font décroître λ' et les projections sur les terme du premier ordre font décroître λ.

7.4 Filtrage du Deuxième Ordre dans les Systèmes de Types

7.4.1 Termes à Arguments Restreints du Deuxième Ordre

Dans les systèmes de types, une autre situation où les termes à arguments restreints du deuxième ordre apparaissent est l'application d'une substitution-produit. Par exemple dans le contexte :

$$[\forall T : Prop; \forall a : T; \forall Q : Prop; \exists f : T \to Prop]$$

on considère le problème :

$$(f\ a) = (P : Prop)Q$$

On lui applique la substitution élémentaire :

$$f \leftarrow [x : T](P : (h\ x))(k\ x\ P)$$

avec $h : T \to s$ et $k : (x : T)((h\ x) \to Prop)$. La variable k a un ordre infini, mais ici encore dans le problème :

$$(P : (h\ a))(k\ a\ P) = (P : Prop)Q$$

elle n'est appliquée qu'à des termes du premier ordre et des termes η-équivalents à une variable universelle.

Définition 84. *: Terme du Deuxième Ordre*

Soit Γ un contexte et t un terme bien typé dans Γ. Le terme t est dit du deuxième ordre dans Γ si pour toute variable x de type T libre dans t,

- *si x est existentielle dans Γ, alors T est d'ordre au plus 2,*
- *et T est un terme du deuxième ordre dans le préfixe de Γ déclaré à gauche de x.*

Définition 85. *: Terme à Arguments Restreints du Deuxième Ordre*

Soit Γ un contexte et t un terme normal bien typé dans Γ. Le terme t est dit à arguments restreints du deuxième ordre dans les cas suivants :

- *$t = [x : T]u$ où T est un terme à arguments restreints du deuxième ordre dans Γ et u un terme à arguments restreints du deuxième ordre dans $\Gamma[\forall x : T]$,*
- *$t = (x : T)u$ où T est un terme à arguments restreints du deuxième ordre dans Γ et u a un terme à arguments restreints du deuxième ordre dans $\Gamma[\forall x : T]$,*
- *$t = (x\ c_1\ ...\ c_n)$ où x est une variable universelle dans Γ et $c_1, ..., c_n$ des termes à arguments restreints du deuxième ordre dans Γ et le type de x un terme à arguments restreints du deuxième ordre dans le préfixe de Γ déclaré à gauche de x,*

— $t = (x\ c_1\ ...\ c_n)$ *où* x *est une variable existentielle et pour tout* i, c_i *est un terme dont le type est du premier ordre dans* Γ *ou un terme* η*-équivalent à une variable universelle de* Γ, c_i *est un terme à arguments restreints du deuxième ordre dans* Γ *et le type de* x *est un terme à arguments restreints du deuxième ordre dans le préfixe de* Γ *déclaré à gauche de* x,
— t *est une sorte.*

Proposition 49. *: Un terme du deuxième ordre est à arguments restreints du deuxième ordre.*

Démonstration : Par récurrence sur la structure de t, tous les cas sont triviaux sauf celui où $t = (x\ c_1\ ...\ c_n)$ avec x existentielle. Dans ce cas, le type de x est d'ordre au plus 2, donc les types des c_i sont du premier ordre.

Proposition 50. *: Soit* t *un terme à arguments restreints du deuxième ordre, et* t' *la forme* η*-longue de* t*. Le terme* t' *est un terme à arguments restreints du deuxième ordre.*

Démonstration : Par récurrence sur $\mu(t)$:
— Si $t = [x : T]u$, alors soit T' la forme η-longue de T et u' la forme η-longue de u, par hypothèse de récurrence T' et u' sont des termes à arguments restreints du deuxième ordre. Le terme $t' = [x : T']u'$ est donc à arguments restreints du deuxième ordre.
— Si $t = (x : T)u$, alors soit T' la forme η-longue de T et u' la forme η-longue de u, par hypothèse de récurrence T' et u' sont des termes à arguments restreints du deuxième ordre. Le terme $t' = (x : T')u'$ est donc à arguments restreints du deuxième ordre.
— Si $t = (x\ c_1\ ...\ c_n)$ et x est une variable universelle ou une sorte, alors par hypothèse de récurrence les c'_i, les T'_j et les x'_k sont à arguments restreints du deuxième ordre. Donc le terme $t' = [x_1 : T'_1]...[x_p : T'_p](x\ c'_1\ ...\ c'_n\ x'_1\ ...\ x'_p)$ est à arguments restreints du deuxième ordre.
— Si $t = (x\ c_1\ ...\ c_n)$ et x est une variable existentielle, alors par hypothèse de récurrence les c'_i, les T'_j et les x'_k sont des termes à arguments restreints du deuxième ordre, si c_i a un type du premier ordre, alors c'_i aussi et si c_i est η-équivalent à une variable universelle de Γ, alors c'_i aussi. Les termes x'_j sont η-équivalents à une variable liée et le type de x est à arguments restreints du deuxième ordre. Le terme $t' = [x_1 : T_1]...[x_p : T_p](x\ c'_1\ ...\ c'_n\ x'_1\ ...\ x'_p)$ est donc à arguments restreints du deuxième ordre.

7.4.2 Equations de Garantie

Equations de Garantie

Dans le λ-calcul simplement typé, avant d'appliquer une substitution élémentaire, il faut vérifier que le terme substitué et la variable à laquelle on le substitue ont même type. Comme l'ont remarqué Elliott [20] [21] et Pym [58], dans les systèmes de types, avant d'appliquer une substitution élémentaire, $x \leftarrow \gamma, t$ il faut unifier les types de x et t. En général, cette *équation de garantie* est un problème d'unification qui peut avoir des variables existentielles dans les deux membres. Mais nous allons montrer que si le problème initial est un problème de filtrage à arguments restreints du deuxième ordre, alors cette équation de garantie est aussi un problème de filtrage à arguments restreints du deuxième ordre et peut donc être résolue.

Le lemme clé montre que si Γ est un contexte et $t = (u\ c_1\ ...\ c_n)$ un terme à arguments restreints du deuxième ordre dont le type est fermé dans Γ et que le type de la variable u est $(x_1 : P_1)...(x_n : P_n)P$ (P atomique), alors P est fermé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.

L'intuition est que le type de $(u\ c_1\ ...\ c_n)$ est $P[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$. Ce terme est fermé, donc P doit également être fermé car les c_i sont des termes atomiques ou des termes η-équivalents à une variable universelle et leur substitution ne peut donc pas effacer de variables existentielles dans P. Prouvons ce lemme :

Proposition 51. *: Soit Γ un contexte, c et T deux termes bien typés dans Γ tel que le type de c soit T et P un terme normal, en forme η-longue, bien typé dans le contexte $\Gamma[\forall x : T]$.*

Si le terme c a un type du premier ordre ou est η-équivalent à une variable universelle et P est atomique, alors $P[x \leftarrow c]$ est atomique.

Démonstration : Si $P = (y\ c_1\ ...\ c_n)$ où $y \neq x$, alors $P[x \leftarrow c] = (y\ c_1[x \leftarrow c]\ ...\ c_n[x \leftarrow c])$ qui est atomique. Si $P = (x\ c_1\ ...\ c_n)$, alors si c a un type du premier ordre, x a aussi un type du premier ordre, donc $n = 0$, $P = x$, $P[x \leftarrow c] = c$ est un terme atomique, si c est η-équivalent à une variable universelle z, alors $P[x \leftarrow c] = (z\ c_1[x \leftarrow c]\ ...\ c_n[x \leftarrow c])$ qui est atomique.

Proposition 52. *: Soit Γ un contexte et $t = (x\ c_1\ ...\ c_n)$ un terme bien typé, à arguments restreints du deuxième ordre, en forme η-longue et $x : (x_1 : P_1)...(x_p : P_p)P$ (P atomique) une variable existentielle. Alors $p = n$.*

Démonstration : Le terme $(x\ c_1\ ...\ c_n)$ n'est pas une abstraction et est en forme η-longue, donc son type n'est pas un produit. Donc $p \leq n$, en effet si on avait $n < p$ le terme $(x\ c_1\ ...\ c_n)$ aurait le type :

$$(x_{n+1} : P_{n+1}[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n])...(x_p : P_p[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n])P[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$$

qui est un produit.

Le type du terme $(x\ c_1\ ...\ c_p)$ est $P[x_1 \leftarrow c_1, ..., x_p \leftarrow c_p]$. Comme le terme P est atomique et tous les c_i sont des termes dont le type est du premier ordre ou qui sont η-équivalents à une variable universelle, le terme $P[x_1 \leftarrow c_1, ..., x_p \leftarrow c_p]$ est aussi atomique, alors si on avait $n > p$, le terme $(x\ c_1\ ...\ c_n)$ serait mal typé. Donc $n = p$.

Proposition 53. *: Soit Γ un contexte, c et T deux termes bien typés dans Γ tels que le type de c soit T et P un terme bien typé dans le contexte $\Gamma[\forall x : T]$. Si c a un type du premier ordre ou est un terme η-équivalent à une variable universelle, alors toute variable y, $y \neq x$ qui a une occurrence dans P a aussi une occurrence dans la forme normale de $P[x \leftarrow c]$.*

Démonstration : Par récurrence sur la structure de P. Le résultat est immédiat si P est une abstraction, un produit ou un terme atomique avec une tête différente de x. Si $P = (x\ c_1\ ...\ c_n)$, alors si c a un type du premier ordre, x aussi, donc $n = 0$, $P = x$ et il n'y a pas de variable $y \neq x$ qui ait une occurrence dans P, si c est η-équivalent à une variable universelle z, alors la forme normale de $P[x \leftarrow c]$ est la forme normale de $(z\ c_1[x \leftarrow c]\ ...\ c_n[x \leftarrow c])$. La variable y a une occurrence dans l'un des c_i, par hypothèse de récurrence, elle a aussi une occurrence dans la forme normale de $c_i[x \leftarrow c]$, elle a donc une occurrence dans la forme normale de $P[x \leftarrow c]$.

Proposition 54. *: Soit Γ un contexte, c et T deux termes bien typés dans Γ tels que le type de c soit T et P un terme bien typé dans le contexte $\Gamma[\forall x : T]$.*

Si la variable x a une occurrence dans P et c a un type du premier ordre ou est un terme η-équivalent à une variable universelle, alors toute variable y qui a une occurrence dans c en a aussi une dans la forme normale de $P[x \leftarrow c]$.

Démonstration : Par récurrence sur la structure de P. Le résultat est immédiat si P est une abstraction, un produit ou un terme atomique avec une tête différente de x. Si $P = (x\ c_1\ ...\ c_n)$, alors si c a un type du premier ordre, x aussi, $n = 0$, $P = x$, $P[x \leftarrow c] = c$, donc y a une occurrence dans la forme normale de $P[x \leftarrow c]$, si c est η-équivalent à une variable universelle z, alors la forme normale de $P[x \leftarrow c]$ est la forme normale de $(z\ c_1[x \leftarrow c]\ ...\ c_n[x \leftarrow c])$. La variable y a une occurrence dans c, donc $y = z$ a une occurrence dans la forme normale de $P[x \leftarrow c]$.

Proposition 55. *: Soit Γ un contexte et P_1, ..., P_n des termes tels que $\Gamma' = \Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ est un contexte bien formé. Soit P_{n+1} un terme et s une sorte tels que $\Gamma' \vdash P_{n+1} : s$. Soit c un terme tel que $\Gamma \vdash c : P_1$. On a $\Gamma[\forall x_2 : P_2[x_1 \leftarrow c]; ...; \forall x_n : P_n[x_1 \leftarrow c]] \vdash P_{n+1}[x_1 \leftarrow c] : s$.*

Si le terme $P_{n+1}[x_1 \leftarrow c]$ est fermé dans $\Gamma[\forall x_2 : P_2[x_1 \leftarrow c]; ...; \forall x_n : P_n[x_1 \leftarrow c]]$, alors P_{n+1} est fermé dans Γ'.

Démonstration : On considère le plus petit ensemble I inclus dans $\{2, ..., n+1\}$ tel que $n+1 \in I$ et si $j \in I$ et x_i est une variable libre de la forme normale de $P_j[x_1 \leftarrow c]$, alors $i \in I$.

On prouve par récurrence descendante que pour tout $i \in I$, la forme normale du terme $P_i[x_1 \leftarrow c]$ est fermée dans le contexte $\Gamma[\forall x_2 : P_2[x_1 \leftarrow c]; ...; \forall x_n : P_n[x_1 \leftarrow c]]$. Pour $i = n+1$, c'est vrai par hypothèse. Pour $i \neq n+1$, x_i a une occurrence dans la forme normale de $P_j[x_1 \leftarrow c]$ pour $j > i$, par hypothèse de récurrence, ce terme est fermé, donc le type de x_i dans Γ' est fermé, c'est-à-dire que la forme normale de $P_i[x_1 \leftarrow c]$ est un terme fermé.

Ensuite on prouve que si x_1 a une occurrence dans l'un des P_i ($i \in I$), alors P_1 est fermé. En effet si x_1 a une occurrence dans P_i, alors toutes les variables qui ont un occurrence dans c ont aussi une occurrence dans la forme normale de $P_i[x_1 \leftarrow c]$. Le terme c est de ce fait fermé, et donc son type P_1 également.

On montre ensuite par récurrence que pour tout $i \in I$, P_i est fermé. Supposons que cela est vrai pour tous les $j < i$ et soit x une variable libre arbitraire de P_i, si $x = x_1$, alors x est universelle et son type est fermé, sinon x est aussi une variable libre de la forme normale de $P_i[x_1 \leftarrow c]$. C'est donc une variable universelle. Si c'est l'une des x_j, alors $j \in I$ et par hypothèse de récurrence son type P_j est fermé. Si c'est une variable déclarée dans Γ, alors son type est le même dans $\Gamma[\forall x_1 : P_1; \forall x_2 : P_2; ...; \forall x_{n+1} : P_{n+1}]$ que dans $\Gamma[\forall x_2 : P_2[x_1 \leftarrow c]; ...; \forall x_{n+1} : P_{n+1}[x_1 \leftarrow c]]$ et donc ce type est fermé.

Enfin on conclut que comme $n+1 \in I$, P_{n+1} est fermé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.

Proposition 56. *: Soit Γ un contexte, P_1, ..., P_n des termes tels que $\Gamma' = \Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ est un contexte bien formé. Soit P_{n+1} un terme et s une sorte tels que $\Gamma' \vdash P_{n+1} : s$. Soit c_i des termes tels que $\Gamma \vdash c_i : P_i[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}]$. On a $\Gamma \vdash P_{n+1}[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n] : s$.*

Si le terme $P_{n+1}[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$ est fermé dans Γ, alors P est fermé dans Γ'.

Démonstration : Par récurrence sur n.

Lemme 5. *: Soit Γ un contexte et $(u\ c_1\ ...\ c_n)$ un terme à arguments restreints du deuxième ordre bien typé dans Γ. On a $\Gamma \vdash u : (x_1 : P_1)...(x_n : P_n)P$ (P atomique).*

Si le terme $(u\ c_1\ ...\ c_n)$ a un type fermé dans Γ, alors P est fermé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.

Démonstration : Le type de $(u\ c_1\ ...\ c_n)$ est $P[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$. Ce type est fermé dans Γ et donc P est fermé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.

On démontre aussi la proposition suivante :

Proposition 57. *: Soit Γ un contexte et $(u\ c_1\ ...\ c_n)$ un terme à arguments restreints du deuxième ordre bien typé dans Γ. On a $\Gamma \vdash u : (x_1 : P_1)...(x_n : P_n)P$.*

Si c_i est η-équivalent à une variable universelle v qui a un type fermé dans Γ, alors P_i est fermé dans le contexte $\Gamma[\forall x_1 : P_1; ...; \forall x_{i-1} : P_{i-1}]$.

Démonstration : Le type de $(u\ c_1\ ...\ c_{i-1})$ est :

$$(x_i : P_i[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}])...(x_n : P_n[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}])P[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}]$$

Le terme $P_i[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}]$ est le type de c_i, c'est-à-dire le type de v, c'est donc un terme fermé dans Γ. Donc P_i est fermé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_{i-1} : P_{i-1}]$.

Terminaison

Nous devons à présent montrer que le fait d'avoir à résoudre cette équation de garantie avant chaque substitution ne met pas en péril la propriété de terminaison de l'algorithme. C'est-à-dire que cette équation de garantie est plus petite pour une certaine mesure que l'équation initiale. Pour cela, on utilise la propriété de bonne fondation de la relation $<$.

Définition 86. *: Complexité d'un Terme*

Soit Γ un contexte et t un terme bien typé dans Γ. Soit T la forme normale du type de t dans Γ. On définit par récurrence sur $<$, la complexité $\kappa(t_\Gamma)$ de t_Γ :

- *Si t est une sorte, alors $\kappa(t_\Gamma) = 1$,*
- *si t est une variable, alors $\kappa(t_\Gamma) = 1 + \kappa(T_\Gamma)$,*
- *si $t = (u\ v)$, alors $\kappa(t_\Gamma) = \kappa(u_\Gamma) + \kappa(v_\Gamma) + \kappa(T_\Gamma)$,*
- *si $t = [x : U]u$, alors $\kappa(t_\Gamma) = \kappa(U_\Gamma) + \kappa(u_{\Gamma[x:U]}) + \kappa(T_\Gamma)$,*
- *si $t = (x : U)V$, alors $\kappa(t_\Gamma) = \kappa(U_\Gamma) + \kappa(V_{\Gamma[x:U]}) + \kappa(T_\Gamma)$.*

Définition 87. *: Complexité d'un Problème*

La complexité d'un problème $< \Gamma, a, b >$ est le couple $\kappa(< \Gamma, a, b >) =< \kappa(b), ex(\Gamma) >$ où $ex(\Gamma)$ est la nombre de variables existentielles déclarées dans Γ.

Proposition 58. *: Soit t_Γ un sous-terme strict de u_Δ. On a $\kappa(t) < \kappa(u)$.*

Démonstration : Par récurrence sur la structure de t.

Proposition 59. *: Soit Γ un contexte, c et T deux termes bien typés dans Γ et P un terme bien typé dans $\Gamma[\forall x : T]$. Si le terme c a un type du premier ordre ou est η-équivalent à une variable universelle et qu'on pose P' la forme normale de $P[x \leftarrow c]$, alors on a $\kappa(P) \leq \kappa(P')$.*

Démonstration : Par récurrence sur la structure de P.

Proposition 60. *: Soit Γ un contexte bien formé et a et b deux termes du même type, tels que $a = (u\ c_1\ ...\ c_n)$ soit un terme à arguments restreints du deuxième ordre et $u : (x_1 : P_1)...(x_n : P_n)P$ est une variable existentielle de Γ. Si b n'est pas une sorte, alors $\kappa(P) < \kappa(b)$.*

Démonstration : On remarque d'abord que comme b n'est pas une sorte, on a $\kappa(Q) < \kappa(b)$. Ensuite le type de a et b est Q la forme normale de $P[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$ et on a $\kappa(P) \leq \kappa(Q) < \kappa(b)$.

7.4.3 Type des Nouvelles Variables

Nous avons vu que dans les calculs polymorphes une variable v dont le type commence par p produits pouvait être appliquée plus de p fois. Donc au moment où on applique une substitution élémentaire on n'a pas toujours toute l'information nécessaire pour calculer le type des nouvelles variables. Par exemple dans le contexte :

$$[\forall T : Prop; \forall U : Prop; \forall v : (P : Prop)P; \forall a : T; \forall G : (Prop \to T) \to Prop; \exists f : Prop \to T]$$

on considère le problème :

$$(f\ U) = (v\ (T \to T)\ a)$$

On veut appliquer la substitution :

$$f \leftarrow [p : Prop](v\ (h_1\ p)\ (h_2\ p))$$

La variable h_1 a le type $Prop \to Prop$ mais comme le terme $(v\ (h_1\ p))$ a le type $(h_1\ p)$ qui n'est pas encore un produit, on ne peut pas donner de type à la variable h_2.

Si on appliquait cette substitution on obtiendrait le problème :

$$(v\ (h_1\ U)\ (h_2\ U)) = (v\ (T \to T)\ a)$$

qui se simplifie en le système :

$$(h_1\ U) = (T \to T)$$

$$(h_2\ U) = a$$

L'idée est de résoudre d'abord la première de ces équations dérivées et de donner ensuite un type à la variable h_2. Cette équation a une solution :

$$h_1 \leftarrow [p : Prop](T \to T)$$

Le terme $(v\ (T \to T))$ a le type $T \to T$, on donne donc le type $Prop \to T$ à h_2 et on peut résoudre la deuxième équation :

$$(h_2\ U) = a$$

Montrons maintenant que dans le cas général comme dans cet exemple, quand on a résolu les k premières équations dérivées, le type de $(v\ ((\sigma h_1)\ x_1\ ...\ x_n)...((\sigma h_k)\ x_1\ ...\ x_n))$ est un produit.

Proposition 61. *: Soit Γ un contexte. Soit $u : (x_1 : P_1)...(x_n : P_n)P$ une variable existentielle de ce contexte. Soient $(u\ c_1\ ...\ c_n)$ un terme bien typé dans Γ à arguments restreints du deuxième ordre, $(v\ d_1\ ...\ d_p)$ un terme bien typé dans Γ et w une variable telle que $w[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n] = v$. Soit k un entier $0 \leq k \leq p-1$. Soient $\alpha_1, ..., \alpha_k$ des termes bien typés dans Γ tels que pour tout i, $1 \leq i \leq k$, $(\alpha_i\ c_1\ ...\ c_n) = d_i$ et tels que pour tout i strictement inférieur à k, le terme $(w\ (\alpha_1\ x_1\ ...\ x_n)...(\alpha_i\ x_1\ ...\ x_n))$ est bien typé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$, son type est un produit $(x : T)T'$ et α_{i+1} a le type $(x_1 : P_1)...(x_n : P_n)T$.*

Alors le terme $(w\ (\alpha_1\ x_1\ ...\ x_n)...(\alpha_k\ x_1\ ...\ x_n))$ est bien typé dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ et son type est un produit.

Démonstration : On a :

$$(v\ d_1\ ...\ d_k) = (v\ (\alpha_1\ c_1\ ...\ c_n)\ ...\ (\alpha_k\ c_1\ ...\ c_n))$$

$$= (w\ (\alpha_1\ x_1\ ...\ x_n)\ ...\ (\alpha_k\ x_1\ ...\ x_n))[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$$

Le terme $(v\ d_1\ ...\ d_k)$ est bien typé dans Γ et son type est un produit, il en est donc de même du type de $(w\ (\alpha_1\ x_1\ ...\ x_n)\ ...\ (\alpha_k\ x_1\ ...\ x_n))[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$.

Soit U le type du terme $(w\ (\alpha_1\ x_1\ ...\ x_n)...(\alpha_k\ x_1\ ...\ x_n))$ dans $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$. Le terme $U[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$ est un produit.

Si le terme U était atomique, il en serait de même de $U[x_1 \leftarrow c_1, ..., x_n \leftarrow c_n]$ puisque les c_i ont un type de premier ordre ou sont η-équivalents à une variable universelle. Le terme U est donc un produit.

Non Linéarité

Dans l'exemple précédent, la variable f n'a qu'une occurrence dans le problème et de ce fait h_2 n'avait pas d'occurrence dans la première équation dérivée. Si on considère à présent le problème :

$$(f\ (G\ f)) = (v\ (T \to T)\ a)$$

en considérant à nouveau la substitution :

$$f \leftarrow [p : Prop](v\ (h_1\ p)\ (h_2\ p))$$

on peut toujours donner un type à h_1 mais pas à h_2. Et h_2 a une occurrence dans la première équation :

$$(h_1\ (G\ [p : Prop](v\ (h_1\ p)\ (h_2\ p)))) = (T \to T)$$

L'idée est de garder la variable f dans l'équation :

$$(h_1\ (G\ f)) = (T \to T)$$

Pour chaque substitution σ solution des équations dérivées, on veut instancier f par le terme $t_1 = [p : Prop](v\ (\sigma h_1\ p)\ (\sigma h_2\ p))$, mais f peut être déjà instancié par $t_2 = \sigma f$. A priori il faut unifier t_1 et t_2.

En fait nous allons voir que dans une substitution solution d'un problème de filtrage, une variable est ou bien invariante, ou bien instanciée par un terme sans variables existentielles et que, de plus, la variable tête du terme flexible est toujours instanciée par un terme sans variables existentielles. Donc le terme t_1 est sans variables existentielles et le terme t_2 est ou bien la variable f ou bien un terme sans variables existentielles. Trois cas peuvent donc se produire : si t_2 est la variable f, alors $\sigma \cup \{< f, [\ ], t_1 >\}$ est une solution du problème initial, si t_1 est sans variables existentielles et différent de t_2, alors σ ne mène pas à une solution du problème initial, si $t_1 = t_2$, alors σ est une solution du problème initial.

7.4.4 Un Algorithme de Filtrage

On considère maintenant des problèmes de filtrage $< \Gamma, a, b >$ tels que Γ est un contexte quantifié bien formé dans le système de types *Méta*, a et b sont des termes normaux bien typés dans le système de types *Méta* dans le contexte Γ et leur type est également bien typé dans Γ (c'est-à-dire est différent de *Extern*). Le terme a est à arguments restreints du deuxième ordre et le terme b est fermé.

Par rapport à [41] [45], nous devons renforcer le contrôle sur l'algorithme. Nous définissons par récurrence sur $\kappa(< \Gamma, a, b >)$ une fonction *Sol* qui prend en argument un problème $< \Gamma, a, b >$ et retourne un ensemble de substitutions solutions du problème $< \Gamma, a, b >$.

Nous utilisons les symboles $\Sigma, \Upsilon, \Phi, \Psi, \Omega$ pour des ensembles de substitutions, étapes intermédiaires dans la construction de $Sol < \Gamma, a, b >$.

Définition 88. *: Insertion*
Soit Γ un contexte, x une variable de Γ. Soit Δ et Δ' tels que $\Gamma = \Delta[Qx : T]\Delta'$. Soit d une déclaration. Le contexte $\Delta\ [d]\ [Qx : T]\ \Delta'$ est appelé l'insertion de d à gauche de x dans Γ.

Notation : Soit Σ un ensemble de substitutions et σ une substitution, on note $\Sigma \circ \sigma$ l'ensemble :

$$\Sigma \circ \sigma = \{\tau \circ \sigma \mid \tau \in \Sigma\}$$

Définition 89. *: Algorithme de Filtrage*

— *Si les termes a et b sont tous les deux des abstractions, modulo α-conversion, on peut considérer que la variable liée dans ces abstractions est la même et est différente de toutes les variables de Γ, $a = [x : U]c$ et $b = [x : U]d$. On pose :*

$$Sol < \Gamma, a, b >= Sol < \Gamma[\forall x : U], c, d >$$

— *Si les termes a et b sont tous les deux des produits, modulo α-conversion, on peut considérer que la variable liée dans ces produits est la même et est différente de toutes les variables de Γ, $a = (x : U)U'$ et $b = (x : V)V'$. Si U et V ont des types différents, alors $Sol < \Gamma, a, b >= \emptyset$, sinon on pose :*

$$\Sigma = Sol < \Gamma, U, V >$$

$$Sol < \Gamma, a, b >= \cup_{\sigma \in \Sigma}(Sol < \sigma\Gamma[\forall x : V], \sigma U', V' > \circ \sigma)$$

— *Si les termes a et b sont tous les deux atomiques et rigides avec la même tête et le même nombre d'arguments : $a = (v\ c_1\ ...c_n)$ et $b = (v\ d_1\ ...d_n)$, on pose :*

$$\Sigma_0 = \{\emptyset\}$$

$$\Sigma_{i+1} = \cup_{\sigma \in \Sigma_i}(Sol < \sigma\Gamma, \sigma c_{i+1}, d_{i+1} > \circ \sigma)$$

$$Sol < \Gamma, a, b >= \Sigma_n$$

— *Dans tous les autres cas où a et b sont tous les deux rigides, on pose : $Sol < \Gamma, a, b >= \emptyset$.*

Considérons maintenant le cas où le terme a est flexible. Comme a est atomique et en forme η-longue, le type commun de a et b n'est pas un produit, donc b n'est pas une abstraction, c'est donc un produit ou un terme atomique.

— *Si a est atomique flexible $a = (u\ c_1\ ...\ c_n)$ et b est un produit $b = (y : V)V'$, alors posons $(x_1 : P_1)...(x_n : P_n)P$ (P atomique) le type de u. Soit s le type de V dans Γ et s' le type de V' dans $\Gamma[\forall y : V]$, s et s' sont des sortes.*
On pose Γ_1 l'insertion à gauche de u dans Γ des déclarations $\exists h : (x_1 : P_1)...(x_n : P_n)s$ et $\exists k : (x_1 : P_1)...(x_n : P_n)(y : (h\ x_1\ ...\ x_n))s'$. On pose :

$$\Sigma = Sol < \Gamma_1, (h\ c_1\ ...\ c_n), V >$$

$$\Upsilon = \cup_{\sigma\in\Sigma}(Sol < \sigma(\Gamma_1[\forall y : V]), (k\ \sigma c_1\ ...\ \sigma c_n\ y), V' > \ \circ\ \sigma)$$

Pour tout $\sigma \in \Upsilon$ on pose :

$$t_\sigma = [x_1 : \sigma P_1]...[x_n : \sigma P_n](y : ((\sigma h)\ x_1\ ...\ x_n))((\sigma k)\ x_1\ ...\ x_n\ y)$$

On pose :

$$Sol < \Gamma, a, b >= \{\sigma \cup \{< u, [\ ], t_\sigma >\} \mid \sigma \in \Upsilon \text{ et } \sigma u = u\} \cup \{\sigma \mid \sigma \in \Upsilon \text{ et } \sigma u = t_\sigma\}$$

— *Si a est atomique flexible $a = (u\ c_1\ ...\ c_n)$ et b est une sorte, alors posons $(x_1 : P_1)...(x_n : P_n)P$ (P atomique) le type de u. On pose :*

$$Sol < \Gamma, a, b >= \{< u, [\ ], [x_1 : P_1]...[x_n : P_n]b >\}$$

— *Si a est atomique flexible $a = (u\ c_1\ ...\ c_n)$ et b atomique $b = (v\ d_1\ ...d_p)$ avec v universelle, on a $u : (x_1 : P_1)...(x_n : P_n)P$ (P atomique).*
On pose $\Gamma' = \Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$. Le terme P est fermé dans Γ'.
Soit $Heads$ l'ensemble contenant tous les x_i tel que c_i a un type du premier ordre ou est η-équivalent à v. Egalement si la variable u est déclarée à droite de v, la variable v est dans $Heads$.
— *Pour toute variable x_i de $Heads$ telle que c_i a un type du premier ordre, les termes P et P_i sont bien typés dans Γ' et leurs types sont des sortes. Si ces sortes sont différentes, alors $\Omega_{x_i} = \emptyset$, sinon on pose :*

$$\Sigma_{x_i} = \{\{< u, [\ ], [x_1 : \sigma P_1]...[x_n : \sigma P_n]x_i >\}\ \circ\ \sigma \mid \sigma \in Sol < \Gamma', P_i, P >\}$$

$$\Omega_{x_i} = \cup_{\sigma\in\Sigma_{x_i}}(Sol < \sigma\Gamma, \sigma a, b > \ \circ\ \sigma)$$

— *Pour toute variable w de $Heads$ telle que $w = v$ ou $w = x_i$ où c_i est η-équivalent à v, par récurrence sur i on construit $\Sigma_{w,i}$ l'ensemble des substitutions qui lient des variables de Γ et aussi des variables supplémentaires $h_1, ..., h_i$.*

$$\Sigma_{w,0} = \{\emptyset\}$$

Pour tout $\sigma \in \Sigma_{w,i}$, le terme $(w\ ((\sigma h_1)\ x_1\ ...\ x_n)\ ...\ ((\sigma h_i)\ x_1\ ...\ x_n))$ est bien typé dans $\sigma\Gamma'$ et son type est un produit $(y : T)T'$. Soit Γ_{i+1} l'insertion à gauche de u dans Γ' de la déclaration $\exists h_{i+1} : (x_1 : P_1)...(x_n : P_n)T$. Soit $H_{i+1} = T[x_1 \leftarrow \sigma c_1, ..., x_n \leftarrow \sigma c_n]$ qui est le type de $(h_{i+1}\ \sigma c_1\ ...\ \sigma c_n)$ dans $\sigma\Gamma_{i+1}$, et D_{i+1} le type de d_{i+1}.
Les termes H_{i+1} et D_{i+1} sont bien typés dans $\sigma\Gamma_{i+1}$ et leurs types sont des sortes. Si elles sont différentes, alors on pose $\Phi_{w,\sigma,i+1} = \emptyset$, sinon on pose :

$$\Upsilon_{w,\sigma,i+1} = Sol < \sigma\Gamma_{i+1}, H_{i+1}, D_{i+1} > \ \circ\ \sigma$$

$$\Phi_{w,\sigma,i+1} = \cup_{\tau \in \Upsilon_{w,\sigma,i+1}} Sol < \tau\Gamma_{i+1}, (h_{i+1}\ \tau c_1\ ...\ \tau c_n), d_{i+1} >\ \circ\ \tau$$

et :

$$\Sigma_{w,i+1} = \cup_{\sigma \in \Sigma_{w,i}} \Phi_{w,\sigma,i+1}$$

Pour tout $\sigma \in \Sigma_{w,p}$, soit $T_{w,\sigma}$ le type du terme $(w\ ((\sigma h_1)\ x_1\ ...\ x_n)\ ...\ ((\sigma h_p)\ x_1\ ...\ x_n))$ dans le contexte $\sigma\Gamma'$. On pose :

$$\Psi_w = \{\sigma \in \Sigma_{w,p} \mid T_{w,\sigma} = P\}$$

Pour tout σ de Ψ_w, on pose :

$$t_{w,\sigma} = [x_1 : \sigma P_1]...[x_n : \sigma P_n](w\ ((\sigma h_1)\ x_1\ ...\ x_n)\ ...\ ((\sigma h_p)\ x_1\ ...\ x_n))$$

et :

$$\Omega_w = \{\sigma \cup \{< u, [\,], t_{w,\sigma} >\} \mid \sigma \in \Psi_w\ et\ \sigma u = u\} \cup \{\sigma \mid \sigma \in \Psi_w\ et\ \sigma u = t_{w,\sigma}\}$$

Enfin :

$$Sol < \Gamma, a, b >= \cup_{w \in Heads} \Omega_w$$

7.4.5 Propriétés

Proposition 62. *: La définition de la fonction Sol est bien fondée.*

Démonstration : La fonction *Sol* est utilisée plusieurs fois dans la définition de $Sol < \Gamma, a, b >$. Dans l'un des cas le membre de droite est b et le contexte a moins de variables existentielles que Γ et dans tous les autres cas le membre de droite est de complexité strictement inférieure à $\kappa(b)$.

Proposition 63. *: Dans le cas où le terme a est flexible, le type des nouvelles variables est bien typé dans le système de type Méta dans le préfixe de Γ déclaré à gauche de la variable u.*

Démonstration : L'unique moment où de nouvelles variables sont introduites est quand a est flexible. Quand b est un produit, on introduit deux variables de types $(x_1 : P_1)...(x_n : P_n)s$ et $(x_1 : P_1)...(x_n : P_n)(y : (h\ x_1\ ...\ x_n))s'$ et quand b est atomique on introduit n variables de types $(x_1 : P_1)...(x_n : P_n)T$. Dans les deux cas, les termes P_i sont de types $Prop$ ou $Type$.

Dans le premier, on a $s = Prop$ ou $s = Type$ donc $(x_1 : P_1)...(x_n : P_n)s$ est un terme bien typé. Comme le type de b est $Prop$ ou $Type$ mais pas $Extern$, on a $s' = Prop$ ou $s' = Type$ donc $(x_1 : P_1)...(x_n : P_n)(y : (h\ x_1\ ...\ x_n))s'$ est un terme bien typé.

Dans le second cas, le type du terme $(w\ ((\sigma h_1)\ x_1\ ...\ x_n)\ ...\ ((\sigma h_n)\ x_1\ ...\ x_n))$ n'est pas une sorte donc il est bien typé dans le contexte $\sigma\Gamma$ et on a $T : Prop$ ou $T : Type$. Donc $(x_1 : P_1)...(x_n : P_n)T$ est un terme bien typé.

Prouvons maintenant que quand le problème $< \Gamma, a, b >$ est bien formé dans un système du cube $\mathcal{T}$, alors toute substitution $\theta \in Sol < \Gamma, a, b >$ est bien typée dans Γ dans ce même système $\mathcal{T}$.

Proposition 64. *: Si le problème $< \Gamma, a, b >$ est tel que tout sous-terme de b qui est un produit $(x : U)U'$ est tel que $< s, s', s'' > \in R$ (l'ensemble des règles du système $\mathcal{T}$) où s est le type de U, s' le type de U' et s'' le type de $(x : U)U'$ et tout sous-terme de b qui est une sorte est Prop mais pas Type, alors il en est de même des substitutions de $Sol < \Gamma, a, b >$.*

Démonstration : Par récurrence sur la complexité du problème $< \Gamma, a, b >$.

Proposition 65. *: Si le problème $< \Gamma, a, b >$ est bien formé dans le système $\mathcal{T}$, alors toute substitution $\theta \in Sol < \Gamma, a, b >$ est bien typée dans Γ dans le système $\mathcal{T}$.*

Démonstration : Par récurrence sur la longueur de Γ, en utilisant la proposition précédente et la proposition 38.

Théorème 6. *: Correction*

Soit $< \Gamma, a, b >$ un problème, toutes les substitutions de $Sol < \Gamma, a, b >$ sont dans $U < \Gamma, a, b >$ (l'ensemble des substitutions solutions de $< \Gamma, a, b >$).

Démonstration : Par récurrence sur la complexité du problème $< \Gamma, a, b >$.

Proposition 66. *: Pour toute substitution $\theta \in Sol < \Gamma, a, b >$ et toute variable u de Γ, $\theta u = u$ ou θu est un terme sans variables existentielles dans $\theta\Gamma$. Et si u est la tête de a, alors θu est un terme sans variables existentielles dans $\theta\Gamma$.*

Démonstration : Par récurrence sur la complexité du problème $< \Gamma, a, b >$.

Théorème 7. *: Complétude*

Soit $< \Gamma, a, b >$ un problème et θ une substitution de $U < \Gamma, a, b >$, alors il existe une substitution $\sigma \in Sol < \Gamma, a, b >$ et une substitution ρ bien typée dans $\sigma\Gamma$ telle que pour toute variable x de Γ, $\theta x = (\rho \circ \sigma)x$.

Démonstration : Par récurrence sur la complexité du problème $< \Gamma, a, b >$.

Dans le cas où le terme a est rigide, a et b sont de façon évidente ou bien deux abstractions, ou bien deux produits ou bien deux termes atomiques avec la même tête et le même nombre d'arguments et les sous-termes se correspondent deux à deux par θ.

Considérons maintenant le cas où le terme a est flexible, $a = (u\ c_1\ ...\ c_n)$.

Soit $(x_1 : P_1)...(x_n : P_n)P$ (P atomique) le type de u, le terme P est fermé dans le contexte $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$. On considère le terme θu :

$$\theta u = [x_1 : \theta P_1]...[x_n : \theta P_n]t$$

Comme $\theta a = b$, on a :

$$t[x_1 \leftarrow \theta c_1, ..., x_n \leftarrow \theta c_n] = b$$

Le terme t est fermé car b est fermé et les θc_i sont ou bien des termes qui ont un type du premier ordre ou bien des termes η-équivalents à une variable universelle.

— Si b est un produit $b = (y : V)V'$, alors le terme t n'est pas une abstraction, il n'est pas atomique parce que $t[x_1 \leftarrow \theta c_1, ..., x_n \leftarrow \theta c_n] = (y : V)V'$ et tous les θc_i sont des termes qui ont un type du premier ordre ou des termes η-équivalents à une variable universelle, c'est donc un produit $t = (y : (\beta_1\ x_1\ ...\ x_n))(\beta_2\ x_1\ ...\ x_n\ y)$ et on a $(\beta_1\ \theta c_1\ ...\ \theta c_n) = V$ et $(\beta_2\ \theta c_1\ ...\ \theta c_n\ y) = V'$.
On pose $\theta_1 = \theta \cup \{< h, [\ ], \beta_1 >, < k, [\ ], \beta_2 >\}$ et Γ_1 l'insertion des déclarations de h et k à gauche de u dans Γ. On a $\theta_1(h\ c_1\ ...\ c_n) = V$ donc, par hypothèse de récurrence, il existe des substitutions $\sigma_1 \in Sol < \Gamma_1, (h\ c_1\ ...\ c_n), V >$ et θ_2 telles que pour chaque variable x de Γ, $\theta_1 x = (\theta_2 \circ \sigma_1)x$.

On a également $\theta_2(k\ \sigma_1 c_1\ ...\ \sigma_1 c_n\ y) = V'$ donc, par hypothèse de récurrence, il existe des substitutions $\sigma_2 \in Sol < \Gamma_2, (k\ \sigma_1 c_1\ ...\ \sigma_1 c_n\ y), V' >$ et θ_3 telles que pour toute variable x de Γ_2, $\theta_2 x = (\theta_3\ \circ\ \sigma_2)x$.
Soit $\tau = \sigma_2\ \circ\ \sigma_1$. On a $\tau \in \Upsilon$ et pour chaque variable x de Γ, $(\theta_3\ \circ\ \tau)x = \theta x$.
Soit $\alpha_1 = \tau h$ et $\alpha_2 = \tau k$. On a $\theta_3\alpha_1 = \beta_1$ et $\theta_3\alpha_2 = \beta_2$.
On considère les termes $t_1 = (y : (\alpha_1\ x_1\ ...\ x_n))(\alpha_2\ x_1\ ...\ x_n\ y)$ et $t_2 = (\tau u\ x_1\ ...\ x_n)$. Ces termes sont tous les deux bien typés et ont le type P dans le contexte $\tau\Gamma[\forall x_1 : \tau P_1; ...; \forall x_n : \tau P_n]$. On a $\theta_3 t_1 = \theta_3 t_2 = (\theta u\ x_1\ ...\ x_n)$. Le terme t_1 est sans variables existentielles dans ce contexte.
Le terme τu est ou bien la variable u ou bien un terme sans variables existentielles. S'il est sans variables existentielles, alors le terme t_2 est aussi sans variables existentielles et $t_1 = t_2$ donc $\tau u = [x_1 : \tau P_1]...[x_p : \tau P_n](y : (\alpha_1\ x_1\ ...\ x_n))(\alpha_2\ x_1\ ...\ x_n\ y)$. On pose $\sigma = \tau$ et $\rho = \theta_3$. On a $\sigma \in Sol < \Gamma, a, b >$ et pour toute variable x de Γ, $\theta x = (\rho\ \circ\ \sigma)x$.
Si $\tau u = u$, on pose $\sigma = \tau \cup \{< u, [\,], [x_1 : \tau P_1]...[x_p : \tau P_n](y : (\alpha_1\ x_1\ ...\ x_n))(\alpha_2\ x_1\ ...\ x_n\ y) >\}$ et $\rho = \theta_3 - \{< u, \gamma, \theta_3 u >\}$ où γ est le contexte associé à u par θ_3.
On a $\sigma \in Sol < \Gamma, a, b >$ et pour toute variable x de Γ, $(\rho\ \circ\ \sigma)x = \theta x$.

— Si b est une sorte, alors comme on a $t[x_1 \leftarrow \theta c_1, ..., x_n \leftarrow \theta c_n] = b$, le terme t est atomique et sa tête est ou bien b ou bien un x_i. Ce n'est pas un x_i parce que les c_i sont ou bien des termes dont le type est du premier ordre ou bien des termes η-équivalents à une variable universelle. La tête de t est donc la sorte b et comme une sorte ne peut être appliquée, on a $t = b$. Donc $\theta u = [x_1 : P_1]...[x_n : P_n]b$. On pose $\sigma = \{< u, [\,], [x_1 : P_1]...[x_n : P_n]b >\}$ et $\rho = \theta - \{< u, \gamma, \theta u >\}$ où γ est le contexte associé à u par θ. On a $\sigma \in Sol < \Gamma, a, b >$ et pour toute variable x de Γ, $\theta x = (\rho\ \circ\ \sigma)x$.
— Si b est atomique $b = (v\ d_1\ ...\ d_p)$ où v est une variable universelle, alors comme on a $t[x_1 \leftarrow \theta c_1, ..., x_n \leftarrow \theta c_n] = b$, le terme t est atomique et sa tête est ou bien v ou bien l'un des x_i. Si c'est l'un des x_i tel que c_i est η-équivalent à une variable universelle, alors cette variable universelle est v.
  — Si la tête de t est un x_i tel que le type de c_i est du premier ordre, alors cette variable ne peut être appliquée et $\theta u = [x_1 : \theta P_1]...[x_n : \theta P_n]x_i$.
  Comme θ est bien typé dans Γ on a $\theta P_i = \theta P$, c'est-à-dire $\theta P_i = P$. Donc, par hypothèse de récurrence, il existe une substitution $\sigma_1 \in Sol < \Gamma', P_i, P >$ et une substitution θ_1 telle que pour toute variable x de Γ, $\theta x = (\theta_1\ \circ\ \sigma_1)x$.
  La variable u n'est pas une variable de P_i, donc elle n'est pas liée par σ_1, $\sigma_1 u = u$ et $\theta_1 u = [x_1 : \theta_1\sigma_1 P_1]...[x_n : \theta_1\sigma_1 P_n]x_i$.
  On pose $\sigma_2 = \{< u, [\,], [x_1 : \sigma_1 P_1]...[x_n : \sigma_1 P_n]x_i >\}$ et $\theta_2 = \theta_1 - \{< u, \gamma, \theta_1 u >\}$ où γ est le contexte associé à u par θ_1.
  On a $\theta_1 = \theta_2\ \circ\ \sigma_2$, donc pour toute variable x de Γ, $\theta x = (\theta_2\ \circ\ \sigma_2\ \circ\ \sigma_1)x$.
  Par hypothèse de récurrence, il existe une substitution $\sigma_3 \in Sol < \sigma_2\sigma_1\Gamma, \sigma_2\sigma_1 a, b >$ et une substitution ρ telle que pour toute variable x de $\sigma_2\sigma_1\Gamma$, $\theta_2 x = (\rho\ \circ\ \sigma_3)x$.
  On pose $\sigma = \sigma_3\ \circ\ \sigma_2\ \circ\ \sigma_1$. On a $\sigma \in Sol < \Gamma, a, b >$ et pour toute variable x de Γ, $\theta x = (\rho\ \circ\ \sigma)x$.
  — Si la tête de t est la variable v ou l'un des x_i tel que c_i est η-équivalent à v, soit w cette tête.
  Le type de w est le type de v ou le type de l'un des x_i tel que c_i est η-équivalent à v, c'est donc un terme fermé.

Dans θu cette variable est appliquée à p termes :

$$\theta u = [x_1 : \theta P_1]...[x_n : \theta P_n](w\ (\beta_1\ x_1\ ...\ x_n)\ ...\ (\beta_p\ x_1\ ...\ x_n))$$

On a $\theta a = b$, donc :

$$(v\ (\beta_1\ \theta c_1\ ...\ \theta c_n)\ ...\ (\beta_p\ \theta c_1\ ...\ \theta c_n)) = (v\ d_1\ ...\ d_p)$$

Donc pour tout i, on a $(\beta_i\ \theta c_1\ ...\ \theta c_n) = d_i$.
Par récurrence sur i on construit des substitutions ρ_i et σ_i telles que pour toute variable de Γ, $(\rho_i\ \circ\ \sigma_i)x = \theta x$ et si on pose $\alpha_j = \sigma_i h_j$, on a pour tout $j < i$, $(\alpha_j\ x_1\ ...\ x_n)$ est sans variables existentielles dans $\sigma_i\Gamma[\forall x_1 : \sigma_i P_1; ...; \forall x_n : \sigma_i P_n]$ et égal à $(\beta_j\ x_1\ ...\ x_n)$.
Pour $i = 0$, on pose $\rho_0 = \theta$ et $\sigma_0 = \emptyset$.
Soit Q_i le type de $(w\ (\alpha_1\ x_1\ ...\ x_n)\ ...\ (\alpha_i\ x_1\ ...\ x_n))$ dans $\sigma_i\Gamma$. D'après la proposition 61, le terme Q_i est un produit $Q_i = (y : T)T'$.
On pose $h_{i+1} : (x_1 : \sigma_i P_1)...(x_n : \sigma_i P_n)T$ et $\theta_{i+1} = \rho_i \cup \{< h_i, [\ ], \beta_i >\}$.
On a $\theta_{i+1}(h_{i+1}\ \sigma_i c_1\ ...\ \sigma_i c_n) = d_i$. Donc, par hypothèse de récurrence, il existe une substitution $\tau_{i+1} \in Sol < \sigma_i\Gamma, (h_{i+1}\ \sigma_i c_1\ ...\ \sigma_i c_n), d_{i+1} >$ et une substitution ρ_{i+1} telles que pour toute variable x de $\sigma_i\Gamma$, $\theta_{i+1}x = (\rho_{i+1}\ \circ\ \tau_{i+1})x$. On pose $\sigma_{i+1} = \tau_{i+1}\ \circ\ \sigma_i$.
Pour toute variable x de Γ on a $\rho_{i+1}\sigma_{i+1}x = \rho_{i+1}\tau_{i+1}\sigma_i x = \theta_{i+1}\sigma_i x = \rho_i\sigma_i x = \theta x$.
On a alors deux substitutions ρ_p et σ_p telles que pour tout x de Γ, $(\rho_p\ \circ\ \sigma_p)x = \theta x$.
On pose pour tout i, $\alpha_i = \sigma_p h_i$.
Soit $t_1 = (w\ (\alpha_1\ x_1\ ...\ x_n)\ ...\ (\alpha_p\ x_1\ ...\ x_n))$ et T le type de t_1 dans le contexte $\sigma_p\Gamma[\forall x_1 : \sigma_p P_1; ...; \forall x_n : \sigma_p P_n]$.
Le terme T est sans variables existentielles car le type de w est sans variables existentielles et les $(\alpha_i\ x_1\ ...\ x_n)$ sont sans variables existentielles.
On a $\rho T = P$ et le terme T est sans variables existentielles donc $T = P$ et $\sigma_p \in \Psi_w$.
Soit $t_2 = \sigma_p u$. On a $\rho_p t_1 = \rho_p t_2$, t_1 est sans variables existentielles et t_2 est ou bien sans variables existentielles ou bien la variable u.
Si t_2 est sans variables existentielles, alors $t_1 = t_2$, on pose $\sigma = \sigma_p$ et $\rho = \rho_p$. Si $t_2 = u$, alors on pose $\sigma = \sigma_p \cup \{< u, [\ ], [x_1 : \sigma_p P_1]...[x_p : \sigma_p P_n]t >\}$ et $\rho = \rho_p - \{< u, \gamma, \rho_p u >\}$ où γ est le contexte associé à u par ρ_p.
Dans les deux cas $\sigma \in Sol < \Gamma, a, b >$ et pour toute variable x de Γ, $(\rho\ \circ\ \sigma) = \theta x$.

Remarque : Le filtrage considéré ici est ouvert au sens du chapitre précédent, pour obtenir un algorithme de filtrage fermé, il faut décider si les types des variables existentielles qui restent dans $\sigma\Gamma$ sont habités ou vides. Bien que tous ces types soient du deuxième ordre au plus, le problème est indécidable dans le plupart des systèmes.

Chapitre 8

Equations Indécidables

8.1 Unification du Deuxième Ordre

Dans le λ-calcul simplement typé, l'unification du deuxième ordre est indécidable (Goldfarb [31]). L'indécidabilité de ce problème dans ce système n'implique pas nécessairement son indécidabilité dans les extensions de ce système. En effet il n'est pas a priori équivalent pour un problème formulé dans le λ-calcul simplement typé d'avoir des solutions dans le λ-calcul simplement typé et dans une de ses extensions.

Mais la preuve d'indécidabilité de l'unification du deuxième ordre dans le λ-calcul simplement typé peut être formulée dans tous les calculs du cube. En effet, en reprenant les notations de [31], si H est un système d'équations arithmétiques et S est le problème d'unification codant ce système, comme dans le λ-calcul simplement typé, pour toute solution θ de S, les variables F_i sont instanciées par des termes de la forme :

$$\theta F_i = [w_1 : U](\overline{n_i}\ w_1)$$

et les variables G_l pour $l = 2^i 3^j 5^k$ par des termes de la forme :

$$\theta G_l = [w_1 : U][w_2 : U][w_3 : U](g\ (t_0^l\ w_1\ w_2)\ (g\ (t_1^l\ w_1\ w_2)\ ...\ (g\ (t_{n_j-1}^l\ w_1\ w_2)\ w_3)))$$

avec :

$$t_p^l = [w_1 : U][w_2 : U](g\ (\overline{n_i.p}\ w_1)\ (\overline{p}\ w_2))$$

$$\overline{n} = [w_1 : U](g\ a\ ...\ (g\ a\ w_1))\ (n \text{ fois})$$

de tout unificateur de S on peut donc déduire une solution de H.

Proposition 67. *: (Goldfarb) Dans un calcul quelconque du cube il n'existe pas de méthode effective permettant de décider si un problème d'unification du deuxième ordre a une solution.*

Remarque : L'unification considérée ici est ouverte au sens du chapitre 6. L'unification fermée est tout aussi indécidable car si le problème $< \Gamma, a, b >$ est le codage d'une équation arithmétique, alors dans toute substitution solution de ce problème les variables de Γ sont instanciées par des termes fermés.

De plus :

Définition 90. *: Problèmes d'Unification Elémentaires*

Un problème d'unification $< \Gamma, a, b >$ *est dit élémentaire au niveau des termes si le contexte* Γ *est de la forme* $\Gamma = [\forall U : Prop]\Gamma'$*, pour toute variable* $Qx : T$ *déclarée dans* Γ'*, le terme* T *est l'un des termes* U*,* $U \to U$*,* $U \to U \to U$ *ou* $U \to U \to U \to U$ *et le type commun de* a *et* b *est* U*.*

Dans un calcul avec constructeurs de types, un problème d'unification est dit élémentaire au niveau des types si pour toute variable $Qx : T$ *déclarée dans* Γ*, le terme* T *est l'un des termes* $Prop$*,* $Prop \to Prop$*,* $Prop \to Prop \to Prop$ *ou* $Prop \to Prop \to Prop \to Prop$ *et le type commun de* a *et* b *est* $Prop$*.*

Proposition 68. *: Dans un calcul quelconque du cube, il n'existe pas de méthode effective permettant de décider si un problème d'unification élémentaire au niveau des termes a une solution.*

Dans un calcul avec constructeurs de types, il n'existe pas de méthode effective permettant de décider si un problème d'unification élémentaire au niveau des types a une solution.

8.2 Filtrage du Troisième Ordre dans les Systèmes avec Types Dépendants et Constructeurs de Types

Nous avons vu précédemment que dans les algorithmes de synthèse de preuves, d'unification et de filtrage, avant d'appliquer une substitution élémentaire, il faut considérer une équation garantissant que cette substitution est bien typée. Dans le chapitre précédent, nous avons vu que dans le cas du filtrage du deuxième ordre, cette équation était également un problème de filtrage. Dès que l'on autorise des variables du troisième ordre, des variables peuvent apparaître dans les deux membres de cette équation.

Cela suggère qu'un problème d'unification du deuxième ordre peut se coder comme l'équation de garantie d'une substitution dans un problème de filtrage du troisième ordre. C'est effectivement le cas dans les systèmes de types avec types dépendants ou constructeurs de types. Le filtrage du troisième ordre est donc indécidable dans ces systèmes.

Théorème 8. *: Indécidabilité du Filtrage du Troisième Ordre dans les Calculs avec Types Dépendants*

Dans un calcul avec types dépendants, il n'existe pas de méthode effective permettant de décider si un problème de filtrage dont toutes les variables existentielles sont du troisième ordre au plus a une solution.

Démonstration : Pour tout problème d'unification élémentaire au niveau des termes $< \gamma, u_1, u_2 >$ on construit un problème de filtrage $< \Gamma, t_1, t_2 >$ tel que tous les types des variables existentielles de Γ soient du troisième ordre au plus et $< \gamma, u_1, u_2 >$ a une solution si et seulement si $< \Gamma, t_1, t_2 >$ en a aussi une. On pose :

$$\Gamma = \gamma[\forall z : U; \forall P : U \to Prop; \forall c : (P\ z); \forall d : (P\ z); \forall G : (P\ z) \to (P\ z) \to (P\ z);$$

$$\exists f : (h : U \to U)(P\ (h\ u_1)) \to (P\ (h\ u_2))]$$

$$t_1 = (G\ (f\ [x : U]z\ c)\ (f\ [x : U]z\ d)) \qquad t_2 = (G\ c\ d)$$

S'il existe une substitution τ telle que $\tau u_1 = \tau u_2$, alors on pose :

$$\sigma = \tau \cup \{< f, [\,], [x_1 : U \to U][x_2 : (P\ (x_1\ (\tau u_1)))]x_2 >\}$$

La substitution σ est bien typée dans Γ et est solution du problème de filtrage $\sigma t_1 = t_2$.

Réciproquement, s'il existe une substitution σ bien typée dans Γ telle que $\sigma t_1 = t_2$, alors σ est bien typée dans γ et on montre que $\sigma u_1 = \sigma u_2$. Soit :

$$\Delta = \Gamma[\forall x_1 : U \to U; \forall x_2 : (P\ (x_1\ (\sigma u_1)))]$$

$$v = ((\sigma f)\ x_1\ x_2) : (P\ (x_1\ (\sigma u_2)))$$

L'équation $\sigma t_1 = t_2$ est équivalente au système :

$$v[x_1 \leftarrow [x : U]z, x_2 \leftarrow c] = c \quad v[x_1 \leftarrow [x : U]z, x_2 \leftarrow d] = d$$

Le terme v n'est pas une abstraction ni un produit parce que son type est $(P\ (x_1\ (\sigma u_2)))$. C'est donc un terme atomique $v = (x\ r_1\ ...\ r_p)$ où x est une variable ou l'un des symboles $Prop$ et $Type$. Le symbole x est l'une des variables x_1, x_2, c parce que $v[x_1 \leftarrow [x : U]z, x_2 \leftarrow c] = c$. Il est différent de c parce que $v[x_1 \leftarrow [x : U]z, x_2 \leftarrow d] = d$. Il est différent de x_1 parce que le type de v est différent de $U \to U$ et U. Donc $x = x_2$. Comme le type de x_2 est atomique, $p = 0$ et $v = x_2$. Les termes v et x_2 ont donc même type et on en déduit :

$$\sigma u_1 = \sigma u_2$$

Théorème 9. *: Indécidabilité du Filtrage du Troisième Ordre dans les Calculs avec Constructeurs de Types*

Dans un calcul avec constructeurs de types, il n'existe pas de méthode effective permettant de décider si un problème de filtrage dont toutes les variables existentielles sont du troisième ordre au plus a une solution.

Démonstration : Pour tout problème d'unification élémentaire au niveau des types $< \gamma, u_1, u_2 >$ on construit un problème de filtrage $< \Gamma, t_1, t_2 >$ tel que tous les types des variables existentielles de Γ soient du troisième ordre au plus et $< \gamma, u_1, u_2 >$ a une solution si et seulement si $< \Gamma, t_1, t_2 >$ en a aussi une. On pose :

$$\Gamma = \gamma[\forall Z : Prop; \forall c : Z; \forall d : Z; \forall G : Z \to Z \to Z; \exists f : (h : Prop \to Prop)(h\ u_1) \to (h\ u_2)]$$

$$t_1 = (G\ (f\ [X : Prop]Z\ c)\ (f\ [X : Prop]Z\ d)) \qquad t_2 = (G\ c\ d)$$

S'il existe une substitution τ telle que $\tau u_1 = \tau u_2$, alors on pose :

$$\sigma = \tau \cup \{< f, [\ ], [x_1 : Prop \to Prop][x_2 : (x_1\ (\tau u_1))]x_2 >\}$$

La substitution σ est bien typée dans Γ et est solution du problème de filtrage $\sigma t_1 = t_2$.

Réciproquement, s'il existe une substitution σ bien typée dans Γ telle que $\sigma t_1 = t_2$, alors σ est bien typée dans γ et on montre que $\sigma u_1 = \sigma u_2$. Soit :

$$\Delta = \Gamma[\forall x_1 : Prop \to Prop; \forall x_2 : (x_1\ (\sigma u_1))]$$

$$v = ((\sigma f)\ x_1\ x_2) : (x_1\ (\sigma u_2))$$

L'équation $\sigma t_1 = t_2$ est équivalente au système :

$$v[x_1 \leftarrow [X : Prop]Z, x_2 \leftarrow c] = c \quad v[x_1 \leftarrow [X : Prop]Z, x_2 \leftarrow d] = d$$

Le terme v n'est pas une abstraction ni un produit parce que son type est $(x_1\ (\sigma u_2))$. C'est donc un terme atomique $v = (x\ r_1\ ...\ r_p)$ où x est une variable ou l'un des symboles $Prop$ et $Type$. Le symbole x est l'une des variables x_1, x_2, c parce que $v[x_1 \leftarrow [X : Prop]Z, x_2 \leftarrow c] = c$. Il est différent de c parce que $v[x_1 \leftarrow [X : Prop]Z, x_2 \leftarrow d] = d$. Il est différent de x_1 parce que le type de v est différent de $Prop \rightarrow Prop$ et $Prop$. Donc $x = x_2$. Comme le type de x_2 est atomique, $p = 0$ et $v = x_2$. Les termes v et x_2 ont donc même type et on en déduit :

$$\sigma u_1 = \sigma u_2$$

Remarque : Le filtrage considéré ici est ouvert au sens du chapitre 6. Le filtrage fermé est tout aussi indécidable car si le problème $< \gamma, u_1, u_2 >$ est le codage d'une équation arithmétique, alors dans toute substitution solution du problème $< \Gamma, t_1, t_2 >$ aussi bien les variables de γ que la variable f sont instanciées par des termes fermé.

8.3 Filtrage dans les Systèmes Polymorphes et Systèmes avec Types Inductifs

Rappelons d'abord quelques résultats classiques sur les algorithmes récursifs primitifs :

Proposition 69. *: Il existe des algorithmes récursifs primitifs qui calculent les fonctions suivantes :*
- *l'addition et la multiplication,*
- *la fonction Equal telle que $(Equal\ x\ y) = 0$ si $x = y$ et $(Equal\ x\ y) = 1$ sinon,*
- *la fonction α telle que $(\alpha\ x\ n)$ est l'exposant du $n^{ème}$ entier premier dans la décomposition de x.*

Démonstration : Voir [65].

Proposition 70. *: Pour toute suite finie d'entiers $a_1, ..., a_n$ il existe un entier x tel que pour tout i, $1 \leq i \leq n$, $a_i = (\alpha\ x\ i)$.*

Démonstration : On prend $x = \Pi_{i=1}^{n} p_n^{a_n}$ où p_n est le $n^{\text{ème}}$ entier premier.

Proposition 71. *: Indécidabilité des Equations Primitives Récursives*

Il n'existe pas de méthode effective qui décide si un algorithme récursif primitif f donné, l'équation $(f\ x_1\ ...\ x_n) = 0$ a une solution.

Démonstration : On réduit le dixième problème de Hilbert [14] à ce problème. Soient $(P\ x_1\ ...\ x_n)$ et $(Q\ x_1\ ...\ x_n)$ deux polynômes. Soit f l'algorithme :

$$(f\ x_1\ ...\ x_n) = (Equal\ (P\ x_1\ ...\ x_n)\ (Q\ x_1\ ...\ x_n))$$

L'équation $(f\ x_1\ ...\ x_n) = 0$ a une solution si et seulement si $(P\ x_1\ ...\ x_n) = (Q\ x_1\ ...\ x_n)$ en a aussi une.

Remarque : Dans la proposition précédente, on peut se restreindre à des équations à une variable en considérant l'algorithme :

$$(f\ x) = (Equal\ (P\ (\alpha\ x\ 1)\ ...\ (\alpha\ x\ n))\ (Q\ (\alpha\ x\ 1)\ ...\ (\alpha\ x\ n)))$$

Plaçons-nous ensuite dans un quelconque calcul polymorphe du cube.

Proposition 72. *: Nous avons vu au chapitre 1 que si on pose :*

$$Nat = (P : Prop)P \to (P \to P) \to P$$

$$n = [P : Prop][x : P][f : P \to P](f\ ...\ (f\ x)\ ...\ )\ \textit{(n fois)}$$

pour tout algorithme primitif récursif f d'arité n, il existe un terme $t : Nat \to ... \to Nat \to Nat$ tel que si $a_1, ..., a_n$ sont des entiers, alors $(t\ a_1\ ...\ a_n)$ se réduit sur l'entier $(f\ a_1\ ...\ a_n)$.

De plus, le terme t peut être effectivement construit à partir de l'algorithme f. On dit que t est un représentant de f.

Proposition 73. *: Soit Γ un contexte et t un terme de type Nat dans Γ. Si la forme normale de $(t\ Nat\ 0\ [y : Nat]y)$ est 0, alors le terme t est un entier de Church.*

Démonstration : Ecrivons $t = [P : Prop][x : P][f : P \to P]u$ où u est de type P dans le contexte $\Gamma[P : Prop; x : P; f : P \to P]$.

Le forme normale du terme $(t\ Nat\ 0\ [y : Nat]y)$ est 0, donc il en est de même de celle du terme $u[P \leftarrow Nat, x \leftarrow 0, f \leftarrow [y : Nat]y]$. On prouve par récurrence sur la structure de u que $u = (f\ ...\ (f\ x)\ ...\ )$.

Le terme u a le type P, donc ce n'est ni une abstraction ni un produit. C'est un terme atomique $(w\ c_1\ ...\ c_p)$. Si w est différent de P, f et x, la forme normale de $u[P \leftarrow Nat, x \leftarrow 0, f \leftarrow [y : Nat]y]$ est également atomique de tête w, ce n'est donc pas le terme 0. La variable w est donc parmi P, f et x. Ce n'est pas la variable P car dans ce cas on aurait $p = 0$ et donc la forme normale de $u[P \leftarrow Nat, x \leftarrow 0, f \leftarrow [y : Nat]y]$ serait le terme Nat qui est différent de 0. La variable w est donc x ou f.

Si $w = x$, alors $p = 0$ et le terme u a la forme requise. Si $w = f$, alors $p = 1$, $u = (f\ u')$. Le terme $u[P \leftarrow Nat, x \leftarrow 0, f \leftarrow [y : Nat]y]$ se réduit sur $u'[P \leftarrow Nat, x \leftarrow 0, f \leftarrow [y : Nat]y]$, donc la forme normale de ce terme est 0. Par hypothèse de récurrence $u' = (f\ ...\ (f\ x)\ ...\ )$, donc $u = (f\ (f\ ...\ (f\ x)\ ...\ ))$ a la forme requise.

Théorème 10. *: Indécidabilité du Filtrage dans les Calculs Polymorphes*

Il n'y a pas de méthode effective permettant de décider si un problème de filtrage dans un système de types polymorphe a une solution.

Démonstration : Soit f un algorithme primitif récursif unaire, on construit un problème de filtrage qui a une solution si et seulement si f prend la valeur 0.

Soit t un terme représentant l'algorithme f. Posons :

$$Pair = [x : Nat][y : Nat][g : Nat \to Nat \to Nat](g\ x\ y)$$

$$\Gamma = [\exists x : Nat]$$

$$a = (Pair\ (x\ Nat\ 0\ [y : Nat]y)\ (t\ x))$$

$$b = (Pair\ 0\ 0)$$

Soit n un entier tel que $(f\ n) = 0$, la substitution $\{< x, [\ ], n >$ est solution du problème $< \Gamma, a, b >$.

Réciproquement, soit θ une solution de $< \Gamma, a, b >$, et $u = \theta x$. On a

$$(u\ Nat\ 0\ [y : Nat]y) = 0$$

$$(t\ u) = 0$$

Par la première équation u est un entier de Church n et par la seconde $(f\ n) = 0$.

Remarque : Le filtrage considéré ici est ouvert au sens du chapitre 6. Le filtrage fermé est tout aussi indécidable car dans toute substitution solution du problème $< \Gamma, a, b >$, la variable x est instanciée par un terme sans variables existentielles.

Remarque : Le problème de filtrage considéré ici est d'ordre ∞. Comme nous avons montré que le filtrage du deuxième ordre est décidable, le problème de la décidabilité du filtrage dans le système F avec uniquement des variables d'ordre fini est laissé ouvert. Ce problème semble lié à celui du filtrage dans le λ-calcul simplement typé.

Plaçons-nous maintenant dans un calcul avec types inductifs.

Proposition 74. *: Nous avons vu au chapitre 1 que pour tout algorithme primitif récursif f d'arité n, il existe un terme $t : Nat \to ... \to Nat \to Nat$ tel que si $a_1, ..., a_n$ sont des entiers, alors le terme $(t\ a_1\ ...\ a_n)$ se réduit sur l'entier $(f\ a_1\ ...\ a_n)$. De plus, le terme t peut être effectivement construit à partir de l'algorithme f. On dit que t est un représentant de f.*

Proposition 75. *: Soit t un terme normal de type Nat tel que le terme $(R\ 0\ \lambda y : Nat.\lambda z : Nat.z\ t)$ ait 0 pour forme normale, alors t a la forme $(S\ ...\ (S\ 0)\ ...\ )$.*

Démonstration : Par récurrence sur la structure de t. Le terme t est de type Nat, ce n'est donc ni une abstraction ni un produit, comme il est normal, c'est un terme atomique $(w\ c_1\ ...\ c_p)$. Si w était différent de 0 et S, alors le terme $(R\ 0\ \lambda y : Nat.\lambda z : Nat.z\ t)$ serait normal et différent de 0. Donc la variable w est 0 ou S.

Si $w = 0$, alors $p = 0$, donc $t = 0$ a la forme requise. Si $w = S$, alors $p = 1$, $t = (S\ t')$. Le terme $(R\ 0\ \lambda y : Nat.\lambda z : Nat.z\ t)$ se réduit sur $(R\ 0\ \lambda y : Nat.\lambda z : Nat.z\ t')$, donc la forme normale de ce terme est 0. Donc par hypothèse de récurrence $t' = (S\ ...\ (S\ 0)\ ...\ )$ et $t = (S\ (S\ ...\ (S\ 0)\ ...\ ))$ a la forme requise.

Théorème 11. *: Le filtrage est indécidable dans les systèmes avec types inductifs. De plus, comme le type Nat est primitif dans ces systèmes, le filtrage est indécidable dès le premier ordre.*

Démonstration : La preuve est la même que celle pour le système F, sauf qu'on remplace le terme $(x\ Nat\ 0\ \lambda y : Nat.y)$ par $(R\ 0\ \lambda y : Nat.\lambda z : Nat.z\ x)$.

Remarque : Plus généralement, si on dit que les algorithmes récursifs primitifs sont *fidèlement* représentables dans un λ-calcul typé quand ces algorithmes sont représentables dans ce système et qu'il existe un terme t de type $Nat \to Nat$ tel que $(t\ u)$ se réduise sur 0 si et seulement si u

représente un entier, alors le filtrage est indécidable dans les systèmes où les fonctions récursives primitives sont fidèlement représentables.

Remarque : Le filtrage considéré ici est ouvert au sens du chapitre 6. Le filtrage fermé est tout aussi indécidable car dans toute substitution solution du problème $< \Gamma, a, b >$ la variable x est instanciée par un terme sans variables existentielles.

Conclusion de la Seconde Partie

Considérons un ensemble E d'équations tel qu'il existe une méthode permettant de décider s'il existe une solution ou non à une équation de E. Soit e une équation qui n'appartient pas à E, l'ensemble $E \cup \{e\}$ admet aussi une méthode de décision et est un sur-ensemble strict de E. Il n'y a donc pas de problème décidable *maximal*. Quand on a prouvé qu'un problème est décidable, il est donc toujours possible de raffiner ce résultat en trouvant un problème décidable plus général. De même, quand on a prouvé qu'un problème est indécidable on peut toujours trouver un problème indécidable moins général. C'est pour cela que nous avons *a priori* limité l'espace des problèmes auxquels nous nous intéressons. Trois critères les déterminent :

- l'ordre des variables,
- la fermeture éventuelle de l'un des termes,
- la clôture de ces problèmes par anti-skolémisation.

Pour les problèmes d'unification où on limite l'ordre des variables, mais où on n'impose pas à l'un des termes d'être fermé, le problème est résolu dans tous les systèmes du cube puisque l'unification est décidable au premier ordre et indécidable au-delà. Il est aussi résolu dans les systèmes avec types inductifs puisque l'unification est indécidable dès le premier ordre.

Pour les problèmes de filtrage où on limite l'ordre des variables et on impose à l'un des termes d'être fermé, le problème est résolu dans les six systèmes comportant des types dépendants et des constructeurs de types puisque le filtrage est décidable jusqu'au deuxième ordre et indécidable au-delà. Pour le λ-calcul simplement typé, le filtrage est décidable au deuxième ordre et la question est ouverte au-delà. Pour le système F le filtrage est décidable au deuxième ordre et indécidable à l'ordre infini. La question est ouverte entre les deux. Pour les systèmes avec types inductifs, le problème est résolu puisque le filtrage est indécidable dès le premier ordre.

Enfin si on prend la clôture des ces problèmes par anti-skolémisation, les problèmes indécidables restent indécidables (puisque ces problèmes sont plus généraux que ceux dont ils sont la clôture) et les problèmes décidables restent décidables puisque l'unification entre termes à arguments restreints est décidable et le filtrage avec un terme ouvert à arguments restreints du deuxième ordre également.

Conclusion

Dans cette thèse, nous avons décrit une méthode de semi-décision pour les systèmes de types. La vérification de preuves étant décidable dans ces systèmes, l'existence d'une telle méthode est en fait immédiate : il suffit d'énumérer toutes les chaînes de caractères jusqu'à tomber sur une preuve de la proposition à démontrer. Cette méthode (qui est souvent utilisée pour prouver la semi-décidabilité du calcul des prédicats) n'a naturellement aucune valeur pratique.

On peut la rendre moins irréaliste en énumérant tous les λ-termes typés (arbres de preuve) normaux jusqu'à tomber sur une preuve de la proposition en question. Cette méthode est comparable aux méthodes de recherche systématique d'un contre-modèle de Herbrand du début des années soixante.

La résolution permet d'exclure a priori davantage de tentatives vouées à l'échec en remarquant que dans une prémisse $\forall x.(P\ x)$, il n'est utile d'instancier x par t que si $(P\ t)$ est une instance de la proposition à prouver ou d'une hypothèse d'une autre prémisse. Cette méthode formulée par Robinson pour la logique du premier ordre est généralisée par Huet à la logique d'ordre supérieur et ici aux systèmes de types. La notion de terme-preuve qui apparaît dans les systèmes de types permet de rendre plus explicite l'idée d'énumération aveugle des preuves régulée par un mécanisme d'anticipation de l'échec par exploitation des contraintes.

Le problème de l'instanciation des variables de prédicat, qui est très brutale en résolution à l'ordre supérieur, apparaît encore ici comme une source majeure d'inefficacité. Ces variables apparaissent dans trois cas : le codage des connecteurs à l'ordre supérieur, l'égalité et les principes de récurrence.

• *Le codage des connecteurs à l'ordre supérieur* : La méthode consistant à coder les connecteurs à l'ordre supérieur et à utiliser la méthode standard est vraisemblablement beaucoup trop naïve. Par exemple, on garde une méthode complète en se restreignant à appliquer une preuve de $A \wedge B$ aux seules propositions A et B.

• *L'égalité* : Remarquons d'abord que l'expressivité du langage des termes dans les systèmes de types permet de ramener de nombreuses preuves d'égalité à des calculs de formes normales. Cette idée de réduction des termes opposée à l'exploration arbitraire de substitutions est la base des méthodes de réécriture. L'intégration de ces méthodes à des méthodes générales comme celle développée ici reste à comprendre.

• *Les principes de récurrence* : L'investigation des méthodes de démonstration automatique confirme une remarque que chacun a faite en cherchant une preuve "à la main" : le choix de l'hypothèse de récurrence est un problème difficile dans le recherche de démonstrations. Cela suggère une utilisation de la synthèse de preuves où un utilisateur indique à la machine les propositions à prouver par récurrence et laisse la machine faire le travail de bas niveau, comme dans les systèmes développés au chapitre 5.

Ce problème du choix de l'hypothèse de récurrence sera sans doutes central dans les travaux à

venir. La question est de savoir s'il est possible de ne retenir a priori qu'un nombre fini d'hypothèses de récurrences (de la même façon que la résolution ne retient a priori qu'un nombre fini de termes pour instancier les variables du premier ordre) ou si l'exploration aveugle d'un nombre infini de propositions est nécessaire à la complétude. Autrement dit, de savoir s'il n'y a que des preuves longues, ou également des preuves difficiles. Cette question peut vraisemblablement se formuler comme un problème de décidabilité de k-prouvabilité. Dans le cas, probable, d'une réponse négative à cette question, il reste à comprendre quelles heuristiques mèneront à un système acceptable, à défaut de complet.

Bibliographie

[1] H. Barendregt, Introduction to Generalized Type Systems, *Journal of Functional Programming*, à parraître.

[2] N.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, pp. 381-392.

[3] N.G. de Bruijn, A Survey of the Project Automath, *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.R. Hindley, J.P. Seldin (Eds.), Academic Press, 1980.

[4] N.G. de Bruijn, The Mathematical Vernacular, A Language For Mathematics With Typed Sets, *Proceedings of the Workshop on Programming Logic*, Marstrand, Sweden, 1987.

[5] N.G. de Bruijn, The Mathematical Vernacular : Examples, Non publié.

[6] A. Church, A Formulation of the Simple Theory of Types, *Journal of Symbolic Logic*, 5,1, 1940, pp. 56-68.

[7] A. Church, The Calculi of Lambda-Conversion, *Princeton University Press*, 1941.

[8] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université Paris VII, 1985.

[9] Th. Coquand, An Analysis of Girard's Paradox, *Proceedings of Logic in Computer Science*, 1986, pp. 227-236.

[10] Th. Coquand, Metamathematical Investigations of a Calculus of Constructions, *Logic and Computer Science*, P. Odifreddi (Ed.), Academic Press, London, 1990, pp. 91-122.

[11] Th. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[12] Th. Coquand, Ch. Paulin, Inductively Defined Types, *Proceedings of the Internatioanl Conference on Computer Logic*, P. Martin-Löf, G. Mints (Eds.) Lecture Notes in Computer Science 417, 1988, pp. 50-66.

[13] H.B. Curry, R.Feys, Combinatory Logic, Vol. 1, *North Holland*, Amsterdam, 1968.

[14] M. Davis, Hilbert's Tenth Problem is Unsolvable, *The American Mathematician Monthly*, 80, 3, 1973, pp. 233-269.

[15] G. Dowek, Naming and Scoping in a Mathematical Vernacular, *Rapport de Recherche 1283*, INRIA, 1990.

[16] G. Dowek, A Complete Proof Synthesis Method for Type Systems of the Cube.

[17] G. Dowek, L'Indécidabilité du Filtrage du Troisième Ordre dans les Calculs avec Types Dépendants ou Constructeurs de Types (The Undecidability of Third Order Pattern Matching in Calculi with Dependent Types or Type Constructors), *Compte Rendu à l'Académie des Sciences*, I, 312, 12, 1991, pp. 951-956.

[18] G. Dowek, A Second Order Pattern Matching Algorithm in the Cube of Typed λ-Calculi, *Proceedings of Mathematical Foundation of Computer Science*, Lecture Notes in Computer Science 520, 1991, pp. 151-160. Rapport de Recherche, INRIA, 1991.

[19] G. Dowek, The Undecidability of Pattern Matching in Calculi where Primitive Recursive Functions are Representable.

[20] C. M. Elliott, Higher-order Unification with Dependent Function Types, *Proceedings of the 3^{rd} International Conference on Rewriting Techniques and Applications*, N. Dershowitz (Ed.), Lecture Notes in Computer Science, 355, Springer-Verlag, 1989, pp. 121-136.

[21] C. M. Elliott, Extensions and Applications of Higher-order Unification, *PhD Thesis*, Carnegie Mellon University, Pittsburgh, 1990.

[22] G. Frege, Begriffsschrift, A Formula Language, Modeled upon that of Arithmetic, for Pure Thought, 1879, reproduit dans [64].

[23] G. Gentzen, The Collected Work of Gehrard Gentzen, M.E. Szabo (Ed.), *North Holland*, 1969.

[24] H. Geuvers, Type Systems for Higher Order Logic, *Catholic University Nijmegen.*

[25] H. Geuvers, The Church-Rosser Property for $\beta\eta$-reduction in Typed Lambda Calculi, *Catholic University Nijmegen*, 1991.

[26] H. Geuvers, M.J. Nederhof, A Modular Proof of Strong Normalization for the Calculus of Constructions, *Catholic University Nijmegen.*

[27] J. Gallier, On Girard's Candidats de Réductibilité, *Logic and Computer Science*, P. Odifreddi (Ed.), Academic Press, London, 1990, pp. 123-203.

[28] J.Y. Girard, Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur, *Thèse de Doctorat d'État*, Université de Paris VII, 1972.

[29] J.Y. Girard, Types and Proofs, translated and with appendices by P. Taylor and Y. Lafont, *Cambrige University Press*, 1989.

[30] K. Gödel, Über eine Bisher Noch Nicht Benützte Erweiterung des Finiten Standpunktes, *Dialectica*, 12, 1958.

[31] W.D. Goldfarb, The Undecidability of the Second-Order Unification Problem, *Theoretical Computer Science*, 13, 1981., pp. 225-230.

[32] M.J. Gordon, A.J. Milner, C.P. Wadsworth, Edinburgh LCF, *Lecture Notes in Computer Science* 78, Springer-Verlag, 1979.

[33] R. Harper, F. Honsell, G. Plotkin, A Framework for Defining Logics, *Proceedings of Logic in Computer Science*, 1987, pp. 194-204.

[34] R. Harper, R. Pollack, Type Checking, Universe Polymorphism and Typical Ambiguity in the Calculus of Constructions, *CCIPL, TAPSOFT'89*, Barcelona, 1989.

[35] L. Helmink, Resolution and Type Theory, *Proceedings of the ESOP Conference*, Copenhagen, Lecture Notes in Computer Science, 432, Springer-Verlag, 1990.

[36] W.A. Howard, The Formulæ-as-type Notion of Construction, 1969, reproduit dans *To H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.R. Hindley, J.P. Seldin (Eds.), Academic Press, 1980.

[37] G. Huet, Constrained Resolution A Complete Method for Higher Order Logic, *PhD Thesis*, Case Western Reserve University, 1972.

[38] G. Huet, A Mechanisation of Type Theory, *Proceedings of the 3rd Intern. Joint Conf. on Artificial Inteligenge*, 1973, pp. 139-146.

[39] G. Huet, The Undecidability of Unification in Third Order Logic, *Information and Control*, 22, 1973, pp. 257-267.

[40] G. Huet, A Unification Algorithm for Typed λ-calculus, *Theoretical Computer Science*, 1, 1975, pp. 27-57.

[41] G. Huet, Résolution d'Équations dans les Langages d'Ordre 1,2, ..., ω, *Thèse de Doctorat d'État*, Université de Paris VII, 1976.

[42] G. Huet, A Uniform Approach to Type Theory, *Rapport de recherche 795*, INRIA, 1988. *Logical Foundation of Functional Programming*, G. Huet (Ed.), Addison-Wesley, 1990, pp. 337-397.

[43] G. Huet, Adding Type :Type to the Calculus of Constructions, Non publié, 1988.

[44] G. Huet, The Constructive Engine, *A Perspective in Theoretical Computer Science*, Commemorative Volume for Gift Siromoney, R. Narasimhan (Ed.), World Scientific Publishing, 1989.

[45] G. Huet, B. Lang, Proving and Applying Program Transformations Expressed with Second Order Patterns, *Acta Informatica*, 11, 1978, pp. 31-55.

[46] D.C. Jensen T. Pietrzykowski, Mecanizing ω-order type theory through unification, *Theoretical Computer Science*, 3, 1976, pp. 123-171.

[47] Y. Lafont, Communication Personelle.

[48] C. L. Lucchesi, The Undecidability of the Unification Problem for Third Order Languages, *Report CSRR 2060*, Department of Applied Analysis and Computer Science, University of Waterloo, 1972.

[49] Z. Luo, An Extended Calculus of Constructions, *PhD Thesis*, University of Edinburgh, 1990.

[50] P. Martin-Löf, Intuitionistic Type Theory, *Bibliopolis*, Napoli, 1984.

[51] D. A. Miller, Unification Under a Mixed Prefix, *Journal of Symbolic Computation*, à parraître.

[52] D. A. Miller, A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification *Extension of Logic Programming*, P. Schroeder-Heister (Ed.), Lecture Notes in Computer Science 475, Springer-Verlag, 1991, pp. 253-281. Report ECS-LFCS-91-159, University of Edinburgh, 1991. Journal of Logic and Computation, à parraître.

[53] Ch. Paulin-Mohring, Extraction de Programmes dans le Calcul des Constructions, *Thèse de Doctorat*, Université Paris VII, 1989.

[54] Ch. Paulin-Mohring, Extracting Fω Programs from Proofs in the Calculus of Constructions, *Proceedings of Principles of Programming Languages*, 1989.

[55] T. Pietrzykowski, D.C. Jensen, A Complete Mechanization of ω-order Type Theory, *Assoc. Comp. Mach. Nat. Conf.*, 1972 Vol 1, pp. 82-92.

[56] F. Pfenning, Logic Programming in the LF Logical Framework, *Logical Frameworks I*, G. Huet et G. Plotkin (Eds.), Cambridge University Press, 1991.

[57] F. Pfenning, Unification and anti-Unification in the Calculus of Constructions, *Proceedings of Logic in Computer Science*, 1991.

[58] D. Pym, Proof, Search and Computation in General Logic, *PhD thesis*, University of Edinburgh, 1990.

[59] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the Association for Computing Machinery*, 12, 1, 1965, pp. 23-41.

[60] A. Salvesen, The Church-Rosser Theorem for LF with β/η-reduction, Manuscrit, University of Edinburgh, 1989.

[61] L.E. Sanchis, Functionals Defined by Recursion, *Notre Dame Journal of Formal Logic*, VIII, 3, 1967, pp. 161-174.

[62] H. Schwichtenberg Definierbare Funktionen im Lambda-Calcul mit Typen, *Archiv Logik Grundlagenforsch*, 17, 1976, pp. 113-114.

[63] W.W. Tait, Intensional Interpretation of Functionals of Finite Type I, *Journal of Symbolic Logic*, 32, 2, 1967, pp. 198-212.

[64] J. van Heijenoort, From Frege to Gödel, *Harvard University Press*, Cambridge, Massachusetts, 1967.

[65] A. Yasuhara, Recursive Function Theory and Logic, *Academic Press*, New York, 1971.

Index

Résumé

Le Calcul des Constructions est une extension de la logique d'ordre supérieur dans laquelle le langage des termes est un λ-calcul typé comprenant des types dépendants, des types polymorphes et des constructeurs de types. La richesse de ce système de types permet, en se basant sur la sémantique de Heyting et l'isomorphisme de Curry-Howard, de représenter les preuves par des termes. Sa puissance calculatoire en fait également un outil bien adapté au raisonnement sur des programmes et à la preuve de leur correction.

La démonstration automatique dans un système logique est l'étude des méthodes qui, partant d'une proposition, en construisent automatiquement une preuve. Dans la première partie de cette thèse nous développons une telle méthode pour le Calcul des Constructions. Cette méthode s'inscrit dans la lignée de celles développées par Robinson pour la logique du premier ordre et Huet pour la logique d'ordre supérieur sous le nom de résolution. En ce qui concerne la démonstration automatique, la nouveauté qu'apportent les systèmes de types, tels que le Calcul des Constructions, par rapport aux systèmes logiques précédents concerne le rôle respectif de la résolution et de l'unification. Au premier ordre et à l'ordre supérieur résolution et unification sont séparées. Dans les systèmes de types, où termes et preuves sont de même nature, résolution et unification se mêlent en un algorithme unique.

Dans la seconde partie de cette thèse, nous étudions les problèmes de résolution d'équations dans le Calcul des Constructions et certaines de ses restrictions. Tout d'abord nous montrons que l'algorithme de synthèse de preuves de la première partie peut être utilisé comme algorithme d'unification, ou plus exactement comme algorithme d'unification fermée. Ensuite, nous étudions les cas particuliers décidables et indécidables de l'unification. Nous montrons que dans le Calcul des Constructions, comme dans le λ-calcul simplement typé, le filtrage du deuxième ordre est décidable. Nous montrons également que le filtrage d'ordre supérieur est indécidable dans tous les calculs qui comportent des types dépendants, des types polymorphes, des constructeurs de types ou des types inductifs.

Mots-Clés

Démonstration Automatique, Résolution, Unification, Filtrage, λ-calcul, Logique d'Ordre Supérieur, Théorie des Types, Calcul des Constructions.