Représentation d'algèbres non libres en théorie des types

Pierre Courtieu

$N°$d'ordre : 6731

L'UNIVERSITÉ PARIS XI
UFR SCIENTIFIQUE D'ORSAY

THÈSE

présentée par
PIERRE COURTIEU
pour obtenir le grade de
DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

REPRÉSENTATION D'ALGÈBRES NON LIBRES EN THÉORIE DES TYPES

Soutenue le : 14 décembre 2001
Après avis de :

MM.	HERMAN GEUVERS	Asssociate Prof.	**Rapporteurs**
	LAURENT FRIBOURG	Chargé de recherche CNRS	

Devant la commission d'examen formée de :

MM.	Joffroy BEAUQUIER	(Professeur)	**Président**
MM.	CHRISTINE PAULIN-MOHRING	(Professeur)	**Examinateurs**
MM.	LAURENCE PUEL	(Professeur)	**Examinateurs**
MM.	GILLES BARTHE	(Chargé de recherche)	**Examinateurs**

- 2001 -

Représentation d'algèbres non libres en théorie des types

Pierre Courtieu

Thèse de doctorat

Décembre 2001

Remerciements

Je veux avant tout remercier très sincèrement Laurence Puel, ma directrice de thèse, dont la rigueur scientifique restera un modèle pour moi. Ses conseils, sa gentillesse et sa disponibilité ont rendu nos nombreuses discussions non seulement intéressantes et fructueuses, mais aussi très agréables et pleines d'enseignements.

Christine Paulin-Morhing m'a apporté toute l'aide possible avec sa gentillesse et sa compétence habituelle, en particulier sur les aspects de théorie des types ; je l'en remercie vivement.

Je remercie également Herman Geuvers et Laurent Fribourg qui ont accepté tout deux d'être mes rapporteurs, leur intérêt et leurs conseils furent précieux.

Je remercie les autres membres du jury : Gilles Barthe, qui a participé pour une très large part à la relecture du manuscrit et à sa correction ; Joffroy Beauquier qui me fit l'honneur de présider ce jury.

Je suis également très reconnaissant envers Jean-Pierre Jouannaud et toute l'équipe «démons» du LRI pour son accueil très chaleureux, pour les nombreuses discussions au «coin café» et pour la bonne humeur générale et communicative. Merci à vous Xavier, Benjamin, Jean-Christophe, Bruno, mon Julien, Claude, Evelyne, Ralf, Delia, Judicaël et les autres que je ne peux pas tous citer. Merci également à David Gross, Yann Verhoeven et Kavé Salamatian, pour leur amitié.

Merci également à Hugues et Jean-Louis pour... non rien :-).

Je dois la vie à mes parents, rien d'original à cela. Les miens m'ont néanmoins apporté plus : l'un m'a donné le goût de la recherche, l'autre celui de la lecture et de l'écriture. Je les remercie du fond du cœur, ainsi que Hugues (le planeur) et Marguerite (l'artiste) pour leur bonne humeur et leurs remarques. Merci à tous.

Enfin je remercie Émilie d'exister.

Table des matières

Introduction

L'étude des fondements des mathématiques a-t-elle une autre utilité qu'occuper l'esprit de quelques mathématiciens philosophes ? Autant l'intérêt *pratique* des mathématiques est une évidence depuis des millénaires, autant celui de la méta-mathématique n'est quant à lui apparu que récemment. Quel besoin en effet l'ingénieur a-t-il de se soucier de la justification mathématique des différentes théories qu'il utilise ? La seule chose importante est que ces outils mathématiques lui permettent de construire des ponts solides. « Pourquoi n'ai-je jamais pu tirer deux déductions contradictoires dans cette théorie ? » est une question concernant plus le mathématicien logicien que l'ingénieur, même si une contradiction manifeste gênerait finalement plus le second que le premier.

Loin de nous l'idée de critiquer un tel point de vue : ce n'est pas avec de la logique méta-mathématique que l'on conçoit un pont !

Cependant il est des conceptions humaines plus récentes dont la fabrication requiert des méthodes frôlant de si près la démonstration mathématique [1] que la question méta-mathématique est à son tour devenue un enjeu économique et donc, paradoxalement, pratique. Il s'agit bien entendu des conceptions informatiques. Plus généralement nous parlons des ouvrages mettant en œuvre des procédures automatiques ; l'électronique est donc par exemple aussi concernée.

Ne minimisons pas les difficultés qu'affronte le programmeur ou l'électronicien : les processeurs et logiciels actuels sont d'une complexité qu'aucun ingénieur ne risquerait sur un chantier ! On peut même s'effrayer de la confiance placée dans ces programmes censés effectuer des tâches aussi critiques que piloter un train ou une fusée. Nombreux sont d'ailleurs les exemples où cette confiance s'est soldée par de coûteux ou tragiques accidents.

Comment alors s'assurer de manière rigoureuse de la correction d'un programme ? De nombreux éléments de réponse ont été donnés à cette question éminemment pratique, mais l'un des plus élégants est probablement l'isomorphisme de Curry-Howard [How80], qui constate en substance qu'écrire un programme et faire une démonstration procèdent fondamentalement de la même activité. Écrire un programme correct revient donc à écrire une démonstration juste. Or c'est là la préoccupation du logicien : qu'est-ce qu'une démonstration ? Comment en *démontrer* la justesse ? Quelles théories sont cohérentes et peuvent donc servir de base à des démonstrations ?

Cette préoccupation méta-mathématique s'est donc subrepticement étendue au-delà de la sphère du logicien pour pénétrer celle de l'ingénieur. Les buts de la recherche ont changé lors de ce transfert. Dans notre cas, là où il s'agissait de prouver ou de réfuter la cohérence de telle ou telle théorie, il faut maintenant démontrer que le programme de 100 000 lignes de code pilotant un métro ne provoquera pas d'accident. Un changement pour le moins radical ! Devant cette constatation, les observations suivantes s'imposent :

1. Nous verrons plus loin que ces méthodes *consistent exactement* en des démonstrations mathématiques.

i. Sans outil automatique (informatique) puissant pour l'aider à concevoir ses démonstrations, l'ingénieur-logicien ne viendra pas à bout de ce travail et commettra certainement des erreurs ;

ii. La question de cohérence des théories utilisées reste entière, elle est même exacerbée par l'utilisation d'un outil automatique ;

iii. L'outil devra être lui-même vérifié préalablement pour que le travail effectué avec son aide soit réellement convaincant.

On voit ici apparaître les grandes lignes d'une branche de la recherche se situant à la frontière de la logique et de l'informatique. Elle consiste à concevoir les outils, théoriques et pratiques, qui assisteront l'utilisateur de manière rigoureuse lors des démonstrations mathématiques. L'utilisateur peut être l'ingénieur-logicien dont nous avons parlé, mais aussi l'étudiant dans le cadre de l'enseignement des mathématiques et on peut même espérer – même s'il reste un long chemin à parcourir – qu'à terme les mathématiciens utiliseront de tels outils pour les assister dans leur travail. Les logiciels conçus dans le cadre de cette recherche s'appellent «outil d'aide à la démonstration», «assistant de preuve»... En anglais le terme «theorem prover» semble s'imposer. Citons quelques noms de tels programmes : l'atelier B, PVS, Coq, Isabelle, NUPrl... Précisons que ces outils, bien qu'encore inadaptés à des développements de très grande taille, sont utilisés aujourd'hui dans l'industrie.

Le début de cette thèse (parties I et II) s'inscrit dans l'effort actuel de conception et d'amélioration de ces outils et propose une extension du *calcul des constructions inductives* ($\mathcal{CCI}$ ou $\mathcal{CIC}$), le formalisme sous-jacent du système Coq [Coq]. Cette extension a pour but de permettre un traitement plus agréable et plus puissant de la notion de quotient, très utilisée en mathématiques. Pour comprendre ce qui se cache derrière les mots «formalisme sous-jacent», il est nécessaire d'introduire les quelques concepts fondamentaux de la logique formelle, c'est l'objet de la fin de cette introduction. Le néophyte intéressé pourra lire l'ouvrage très pédagogique de Gilles Dowek [Dow95], duquel sont tirés quelques faits historiques cités dans la suite.

La dernière partie présente le développement dans l'assistant Coq d'une preuve *d'algorithme auto-stabilisant*. Un algorithme auto-stabilisant est un algorithme s'exécutant sur un réseau[2] et ayant la propriété suivante : à partir d'une configuration initiale quelconque du réseau, l'algorithme fait revenir celui-ci dans une configuration correcte au bout d'un temps fini et ne quitte plus les configurations correctes ensuite. On entend par *configuration correcte* une configuration correspondant à un déroulement valide de l'algorithme.

La méta-mathématique

L'étude du raisonnement remonte probablement plus loin, mais les philosophes grecs sont les premiers à nous en avoir laissé trace. Aristote, face à l'évidente observation qu'il existe des raisonnements faux, s'interroge sur la nature et les mécanismes de la *déduction*. Il constate déjà que pour *déduire*, il faut en premier lieu *admettre* des faits préalables, comme par exemple dans le raisonnement suivant : "Si on admet que tous les hommes sont mortels et que les Grecs sont des hommes, alors les Grecs sont mortels". Cette pré-admission mène à la notion de *théorie*.

2. Un tel algorithme est dit *réparti*.

Théorie

Le mot théorie désigne un ensemble de faits admis[3], appelés *axiomes*. Ici les deux axiomes "tous les hommes sont mortels" et "les Grecs sont des hommes" permettent de déduire que les Grecs sont mortels.

Au risque d'enfoncer une porte ouverte, précisons qu'il n'est absolument pas correct d'en déduire que les Grecs sont mortels ! On a montré qu'ils sont mortels SI on admet les deux axiomes, rien de moins, rien de plus. On reformule cet énoncé de manière plus logicienne en : « "les Grecs sont mortels" est *démontrable* dans la théorie {"tous les hommes sont mortels", "les Grecs sont des hommes"} »[4]. On peut également exprimer ce raisonnement lui-même sous la forme d'une propriété : "(tous les hommes sont mortels ET les Grecs sont des hommes) IMPLIQUE que les Grecs sont mortels". Cette propriété est elle-même démontrable à partir d'un ensemble d'axiomes vide : c'est une tautologie. On remarque d'ailleurs que comme dans toute tautologie, les propriétés élémentaires peuvent être remplacées par n'importe quelles autres propriétés, par exemple : "(tous les éléphants sont bleus et les poissons sont des éléphants) implique que les poissons sont bleus" est en fait la même tautologie, elle est tout aussi correcte.

Déduction

Pourtant, là encore quelque chose a été admis : la notion de raisonnement correct, autrement dit le *mécanisme de déduction*. En effet, sans un mécanisme pour reconnaître un raisonnement correct, comment différencier une déduction fausse d'une déduction valide ? De la même manière qu'en cinétique la notion de vitesse n'a de sens qu'à l'intérieur d'un référentiel, une proposition en logique n'est donc démontrée qu'à l'intérieur d'une théorie (éventuellement vide pour les tautologies) et d'un système de déduction permettant de déduire des propriétés à partir des axiomes de la théorie. Si ce dernier est vide, alors rien n'est démontrable, pas même les axiomes. Un abus de langage autorise parfois à appeler théorie ce couple $\langle \textit{théorie}, \textit{système} \rangle$.

Un système est aujourd'hui généralement présenté sous la forme d'un ensemble de règles formelles. Par exemple une règle communément admise dans tout système logique est la suivante :

$$\frac{}{A} \quad \textit{(si } A \textit{ est un axiome)}$$

On lit une telle règle de la manière suivante :

— si les prémisses (les propriétés énoncées au-dessus de la barre horizontale, ici il n'y en a pas) sont démontrables,

— alors la conclusion (en dessous de la barre) l'est aussi.

On voit bien ici le mécanisme de déduction défini : la règle permet de démontrer une propriété A à condition qu'elle soit un axiome. D'autres exemples classiques de règles de déduction sont le *modus ponens* et la règle du "et" logique :

$$\frac{A \quad A \Rightarrow B}{B} \qquad\qquad \frac{A \quad B}{A \wedge B}$$

3. Le mot théorie désigne aussi parfois l'ensemble des *théorèmes* démontrables à partir des axiomes.

4. L'ensemble des propriétés démontrables dans une théorie est l'ensemble des propriétés qu'on peut *déduire* des axiomes. On verra qu'une déduction se fait à l'aide d'un *système* de règles de déduction, lui-même défini au préalable.

qui se lisent respectivement : "si A est démontrable et si A *implique* B (noté $A \Rightarrow B$) est démontrable, alors B est démontrable", et "si A et B sont chacun démontrable, alors A *et* B (notée $A \wedge B$) est démontrable". Précisons ici que nous ne sommes pas en train d'énoncer une propriété mais bien une définition. En écrivant un ensemble de règles de déduction, on donne son sens au mot "démontrable" dans le contexte que nous avons choisi. Une démonstration se présente sous la forme d'un ensemble d'instances de règles imbriquées, formant un arbre. Par exemple, voici la preuve de $a < c$ dans la théorie constituée des trois axiomes $\{a < b,\ b < c,\ (a < b \wedge b < c) \Rightarrow a < c\}$:

$$\dfrac{\dfrac{}{(a < b \wedge b < c) \Rightarrow a < c} \quad \dfrac{\dfrac{}{a < b} \quad \dfrac{}{b < c}}{a < b \wedge b < c}}{a < c}$$

On voit que : 1) Toutes les feuilles de l'arbre contiennent des axiomes et 2) Les règles sont correctement appliquées. C'est avec ces deux critères facilement vérifiables qu'on reconnaît une déduction complète et valide. Loin d'être anodin, le besoin d'un mécanisme simple pour la vérification est au contraire primordial. En effet, si la vérification de la validité d'un raisonnement nécessite elle-même un raisonnement, alors il faut aussi s'assurer de la validité de ce dernier, ce qui demandera encore un raisonnement, etc. En fait, depuis l'apparition de la notion de machine programmable (notamment la célèbre *machine théorique de Turing*) et l'avènement des ordinateurs et de leur capacité à traiter des données symboliques, on a pu à la fois relâcher et préciser ce principe : *il suffit que la vérification soit décidable*[5], c'est-à-dire qu'il soit possible d'écrire un programme (au sens *programme de machine de Turing*) capable de l'effectuer.

Le travail des logiciens

Le travail des logiciens consiste à définir des systèmes de déduction et des théories, et d'en étudier :

- *L'expressivité*, c'est-à-dire la quantité de propriétés exprimables et/ou démontrables à partir des axiomes de la théorie en utilisant les règles du système de déduction. Plus une théorie est expressive, plus elle permettra d'étudier de problèmes.
- La *cohérence*, c'est-à-dire l'impossibilité de démontrer une propriété et son contraire à partir des axiomes de la théorie en utilisant les règles du système de déduction (une telle propriété est appelée *paradoxe* de la théorie). Si c'était possible, la théorie, dite *contradictoire*, deviendrait inintéressante puisque permettant de démontrer n'importe quelle propriété. Un problème sur lequel nous reviendrons plus loin est que pour *prouver* la cohérence d'une théorie, il faut nécessairement se placer dans une autre théorie.

On s'est vite aperçu que plus une théorie est expressive, plus il est délicat de se convaincre puis de démontrer qu'elle est cohérente. L'exemple le plus célèbre de cette difficulté nous vient des théories des ensembles de Frege et Cantor, prouvées contradictoires par Cesare Burali-Forti puis Bertrand Russel en 1897 et 1902. Russel a pour cela exhibé la propriété suivante : "L'ensemble des ensembles qui ne se contiennent pas eux-mêmes se contient lui-même", qui est un paradoxe dans ces théories. Redéfinies par Ernst Zermelo en 1908 et par Alfred North Whithead et Russel en 1910 de

5. Cette notion, la *décidabilité*, est définie de manière très précise et caractérise tous les problèmes que le calcul sans raisonnement est capable de résoudre. Il est très courant de définir par inadvertance une notion de vérification indécidable, c'est-à-dire qu'aucun programme ne pourra effectuer dans tous les cas.

manière à rendre cet énoncé impossible à formuler, ces théories n'ont finalement pas vu leur avenir remis en question et servent toujours de base aux mathématiques actuelles.

"Le" siècle de la logique

De nombreuses découvertes fondamentales, et parfois surprenantes, ont été faites dans le domaine de la logique au cours du vingtième siècle. Nous en donnons ici une petite chronologie afin de présenter brièvement les évolutions majeures de la logique récente.

1900 Gottlob Frege pose les bases de la logique formelle [Fre03] ; son approche rigoureuse de la notion de *preuve* l'amène à définir les notions de *formule* et de *propriété* comme elles le sont encore aujourd'hui. Frege est à l'origine de la notion de système de preuve évoquée à la section précédente.

1900 La théorie naïve des ensembles est prouvée contradictoire au tout début du siècle, comme nous l'avons dit plus haut, ainsi que la théorie logique de Frege. C'est la première fois que le problème de la cohérence et sa difficulté se posent de manière si directe à l'ensemble des mathématiciens. Nous avons vu que ce problème n'a pas vraiment eu de conséquence et qu'il a été corrigé quelques années plus tard.

En fait plusieurs solutions ont été proposées, dont les suivantes :

— Russel propose sa *théorie des types*[6], une version de la théorie des ensembles enrichie d'une notion de *hiérarchie* permettant d'éviter le paradoxe en cloisonnant les différents niveaux du discours.

— Luitzen Brouwer propose une vision *intuitionniste* des mathématiques [Hey56], dans laquelle on ne peut prouver l'existence d'un objet qu'en exhibant, sinon l'objet lui-même, au moins un mécanisme pour le construire. L'intuitionnisme est à la base de la théorie des types intuitionniste de Martin-Löf (voir plus bas), qui est considérée aujourd'hui comme la mathématique des concepts décidables. En particulier, cette notion de mécanisme permettant de construire des objets, autrement dit *d'algorithme*, s'adapte parfaitement aux raisonnements nécessaires à la *programmation*.

— Ernst Zermelo en 1908 propose une axiomatisation de la théorie des ensembles. Le schéma de compréhension, qui permettait de construire le paradoxe de Russel, y est restreint de manière à le rendre impossible à formuler. Cette théorie, modifiée par Abraham Fraenkel, ZF, sert de base à la plupart des développements mathématiques actuels.

1931 Gödel surprend la communauté mathématique en démontrant le théorème d'incomplétude [Göd31, Göd34], qui stipule que dans toute théorie permettant d'exprimer des propriétés arithmétiques (ce qui est assez minimal), il existe des propriétés qui ne sont pas démontrables et dont la négation n'est pas démontrable non plus. Par la suite, quelques problèmes ouverts bien connus sont prouvés *indéterminés*, c'est-à-dire appartenant à cette catégorie[7] (par exemple l'hypothèse du continu est indéterminée dans la théorie des ensembles).

Ce résultat semble contrarier l'intuition mathématique : une propriété peut être soit prouvable, soit réfutable, *soit ni l'un ni l'autre*.

6. Ce n'est pas la théorie des types dont il est question dans cette thèse ; nous y parlerons de la théorie des types de Martin-Löf, laquelle est en effet une reformulation intuitionniste de celle de Russel.

7. On dit aussi qu'une propriété est *indépendante* d'une théorie.

1931 Gödel ne s'arrête pas là. Puisque toute démonstration doit se faire dans une théorie, une démonstration de cohérence d'une théorie s'appuie nécessairement sur une autre théorie. Une telle démonstration est dite relative : on prouve la cohérence *sous la condition* que la théorie utilisée pour faire cette preuve est elle-même cohérente. Il est évident que cet exercice n'est complètement convaincant que si cette cascade s'arrête sur une théorie dont on sait qu'elle est cohérente absolument. Gödel élimine tout espoir d'un tel rêve avec son second théorème d'incomplétude : la théorie utilisée est nécessairement plus expressive que la théorie dont on cherche à prouver la cohérence. Par conséquent, on ne pourra pas arrêter la cascade.

Devant cette incapacité de la mathématique à se justifier elle-même, il ne reste plus qu'à faire acte de profession de foi, comme le fait Bourbaki à la fin de son introduction [Bou66] :

> *«En résumé, nous croyons que la mathématique est destinée à survivre, et qu'on ne verra jamais les parties essentielles de ce majestueux édifice s'écrouler du fait d'une contradiction soudain manifestée ; mais nous ne prétendons pas que cette opinion repose sur autre chose que l'expérience. C'est peu diront certains. Mais voilà vingt-cinq siècles que les mathématiques ont l'habitude de corriger leurs erreurs et d'en voir leur science enrichie, non appauvrie ; cela leur donne le droit d'envisager l'avenir avec sérénité.»*

1934 Gerhard Gentzen propose le calcul des séquents [Gen69, Gen34], un système de déduction exprimant la logique du premier ordre. C'est le premier système dans lequel une propriété est démontrée non pas comme conséquence des axiomes de la théorie de base, mais comme conséquence d'un ensemble de propriétés construit au cours de la déduction. Les règles, toujours sous la forme de fractions comme plus haut, sont constituées de *séquents* au lieu de formules (propriétés). Un séquent $\Gamma \vdash A$ signifie "A est démontrable sous l'ensemble d'hypothèses Γ". Par exemple, la règle de déduction du et logique s'exprime de la manière suivante :

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

On lit cette règle de la manière suivante :

— si sous l'ensemble d'hypothèses Γ, on peut prouver A,
— et si sous l'ensemble d'hypothèses Γ, on peut prouver B,
— alors sous l'ensemble d'hypothèses Γ, on peut prouver "A et B".

On voit que ce système met l'accent sur la notion de *conséquence* plus que sur celle de *vérité*. Il permet notamment d'ajouter des hypothèses au cours du raisonnement. Il est clair aujourd'hui que cette vision est plus souple et plus conforme au raisonnement intuitif. Un autre avantage de cette approche est qu'il n'y a plus d'axiome à proprement parler, seulement une règle permettant de déduire une propriété A d'un ensemble d'hypothèses contenant A :

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

1936 L'opinion de Hilbert, selon laquelle les mécanismes de raisonnement décrits dans les systèmes logiques ne sont que des méthodes de calcul pouvant être automatisées, est réfutée à la fois par Church et Turing. Ils montrent qu'il existe des problèmes qui ne sont pas *décidables*, c'est-à-dire qu'aucun calcul n'est capable de les résoudre. C'est en particulier le cas du problème consistant à

prendre une propriété quelconque P et de répondre à la question «P est-elle démontrable dans la théorie des ensembles[8] ?».

Cette découverte décevante nous intéresse particulièrement car elle signifie que le projet d'un programme de démonstrations mathématiques complètement automatique est voué à l'échec.

1930-40 Parallèlement, Alonzo Church développe le λ-calcul [Chu40], un formalisme permettant d'exprimer les fonctions sous la forme d'algorithmes (de *termes*), accompagné d'une notion de *calcul* permettant d'évaluer les fonctions lorsqu'elles sont appliquées à leurs arguments. Par exemple, la fonction identité est représentée de la manière suivante :

$$\lambda x.x$$

qui se lit «fonction qui prend un argument auquel on donnera le nom x à l'intérieur de la définition[9], et qui rend x».

On peut appliquer cette fonction à un terme t :

$$(\lambda x.x\ t)$$

et le mécanisme de calcul stipule qu'il faut alors faire disparaître l'abstraction et remplacer l'argument abstrait par l'argument réel, ce qui redonne t dans notre cas :

$$(\lambda x.x\ t) \longrightarrow t$$

De même le terme $\lambda z.z + 1$ est intuitivement la fonction qui rend la valeur de son argument (appelé z à l'intérieur de la définition) incrémentée de un. Il faut ici préciser que 1 et + ne font pas partie du λ-calcul et doivent être *codés* de manière assez subtile sous forme de termes. Nous reviendrons sur cet aspect peu pratique pour justifier les dernières évolutions de la théorie des types à la fin de cette section. Nous étudions plus en détail le λ-calcul et ses versions typées dans la partie I de cette thèse.

Cette représentation *intentionnelle* des fonctions intéresse Church d'une part pour étudier les fonctions et leur comportement calculatoire (cet aspect a donné naissance aux langages de programmation fonctionnels) et d'autre part pour lui adjoindre des règles logiques afin de définir un nouveau système de déduction, appelé "logique d'ordre supérieur" et capable de remplacer la théorie des ensembles[10]. L'idée principale est d'utiliser comme brique de base non plus l'ensemble – comme en théorie des ensembles – mais la fonction. Plus explicitement : de même que la définition des entiers (et de tout autre concept) en théorie des ensembles passe par un codage utilisant des ensembles, de même en théorie des types les entiers (dits *de Church*) sont codés par des fonctions.

La première version de ce système s'avère contradictoire mais les suivantes, grâce à la notion de *type* associé à un terme, sont prouvées cohérentes (relativement à la théorie des ensembles). Le λ-calcul et ses avatars servent aujourd'hui de fondement à de nombreux systèmes de preuves actuels, où il est fait un usage intensif de la possibilité de représenter des fonctions intentionnellement (c'est-à-dire en donnant un algorithme pour les calculer). Cet aspect est d'ailleurs souvent présenté comme l'avantage majeur de la théorie des types sur la théorie des ensembles comme fondement des assistants de preuve.

8. Ainsi que dans toute autre théorie ayant une expressivité comparable, l'indécidabilité réside bien dans la logique exprimée par le système et non dans sa formulation.

9. x est donc un argument *abstrait*, c'est la signification de la notation λx.

10. Ce but ne sera pas atteint de la manière imaginée par Church car le λ-calcul typé n'est pas assez expressif. C'est la théorie des types de Martin-Löf qui y parviendra, voir plus bas.

1965 Dag Prawitz propose le système de *Déduction naturelle* [Pra65] (voir également section 2.6.3 figure 2.2), du même genre que celui de Gentzen. On s'aperçoit alors que les règles de ce système sont les mêmes que les règles de typage du λ-calcul. Cet isomorphisme, dit de *Curry-Howard*, fait apparaître que les preuves faites dans ce système sont isomorphes aux λ-termes. De ce fait, un λ-terme est *à la fois* un objet mathématique (une fonction de A dans B, un entier...) *et* une preuve (la preuve que $A \Rightarrow B$, la preuve que l'ensemble des entiers contient un élément...). De même les types correspondent aux propositions et une proposition (un type) est donc prouvable s'il existe un terme ayant pour type cette proposition. C'est l'idée principale de la théorie des types de Martin-Löf (ci-dessous). Les conséquences de cet isomorphisme sont étonnamment riches et continuent d'être découvertes de nos jours. C'est la pierre angulaire des démonstrateurs tels que Coq ou Lego, qui utilisent les λ-termes pour représenter les preuves et le typage de ces termes pour vérifier de quoi ils sont la preuve.

1969 Per Martin-Löf décrit la *théorie des types intuitionniste* [ML84], un système de déduction pour les mathématiques constructives (c'est-à-dire intuitionnistes). Son originalité réside dans l'utilisation de l'isomorphisme de Curry-Howard pour interpréter la notion de démonstration à l'intérieur même du formalisme. Une proposition est interprétée comme un ensemble, dont les éléments représentent les preuves possibles de cette proposition. Une proposition indémontrable est donc représentée par l'ensemble vide et une proposition démontrable par un ensemble non vide.

Remarque : *Les systèmes logiques issus de la théorie des types sont en général présentés sous la forme d'un système de typage du type λ-calcul grâce à l'isomorphisme de Curry-Howard. Nous adopterons aussi cette présentation tout au long de cette thèse.*

1970 à nos jours De nombreux systèmes ont été proposés depuis ; un des critères intéressants pour ces systèmes est la réduction des étapes de codage nécessaires pour définir les concepts. On a vu en effet que dans le λ-calcul et la théorie des types, la définition de notions aussi simples que $\mathbb{N}$ passait par des codages lourds et peu adaptés à une implantation sur machine. En plus des critères de cohérence et d'expressivité, le souci grandissant d'une implantation a donc poussé les logiciens vers de nouveaux formalismes plus riches et plus souples que les anciens systèmes jusque là très minimaux. Nous en donnons ici une très courte liste.

- Les systèmes F et F_ω par Jean-Yves Girard (années 1970, [GLT89]), dont l'expressivité est suffisante (notamment grâce au polymorphisme) pour exprimer à peu près toutes les mathématiques usuelles.
- Le *calcul des constructions* ($\mathcal{CC}$) par Thierry Coquand et Gérard Huet en 1985 ([CH88]), un système plus puissant mais surtout plus souple, grâce à la possibilité de définir des prédicats de manière directe (à l'aide de *types dépendants*). Ce calcul est présenté brièvement au chapitre 3.
- Le *calcul des constructions inductives* ($\mathcal{CCI}$ ou $\mathcal{CIC}$) par Thierry Coquand et Christine Paulin-Mohring [CPM90, PM93, PM96, Wer94], une extension de $\mathcal{CC}$ encore plus souple puisqu'elle permet de définir de nouveaux types, dits *inductifs*, semblables aux types récursifs des langages de programmation fonctionnels. Ce calcul, dont il est question dans cette thèse, est le système logique sous-jacent de l'assistant de preuve Coq ; il sera présenté au chapitre 4.

À chaque nouveau système, des commodités supplémentaires sont introduites dans le formalisme, compliquant son analyse mais facilitant sa mise en œuvre informatique et la création des outils de

démonstration.

Le but de cette thèse est

— Premièrement, de prolonger cette course à la souplesse des définitions en présentant une extension du calcul des constructions inductives dans laquelle la notion de structure quotient, qui nécessite un codage axiomatique dans $\mathcal{CCI}$, peut être définie de manière interne.
La définition de structures non libres, c'est-à-dire dans lesquelles deux termes ayant des constructeurs différents peuvent être égaux, pose en effet différents problèmes de cohérence et de simplicité d'utilisation. En particulier dans $\mathcal{CCI}$ il est bien connu que l'ajout d'égalité entre constructeurs d'un type inductif mène à une inconsistance (voir 4.1.6). Ces problèmes sont de manière générale liés aux interactions existant entre deux notions qui se combinent difficilement : le *calcul* et la *déduction*. La connexion entre ces deux activités, toutes deux indispensables au raisonnement, est faite traditionnellement par les *schémas d'élimination* (voir 4.1.1). Un schéma – ou principe – d'élimination permet de caractériser de manière constructive l'ensemble des termes d'un certain type ; ce qui permet à la fois de justifier les démonstrations par récurrence sur ce type et de permettre la définition de fonctions sur ce type (puisqu'on sait sur quels éléments il faut définir une fonction ou faire une preuve).
Notre travail consiste à proposer un calcul dans lequel on peut définir des structures non libres sans introduire d'inconsistance tout en conservant des principes d'élimination utilisables. Pour cela on permet la définition d'un type, dit *normalisé*, à partir d'un autre type T et d'une *fonction nf* de T vers T. Dans ce nouveau type, deux termes x et y seront égaux si et seulement si $(nf\ x)$ et $(nf\ y)$ sont égaux. Les propriétés de confluence, normalisation forte et stabilité du typage par rapport à la réduction sont démontrées pour ce calcul.

— Deuxièmement, de présenter un nouveau champ d'application des assistants de preuve : la preuve d'algorithmes auto-stabilisants.

Plan de la thèse

La partie I est consacrée à la présentation progressive de différents λ-calculs : les λ-calculs pur et typé, puis le calcul des constructions et enfin le calcul des constructions inductives. On introduira également quelques notions importantes comme l'isomorphisme de Curry-Howard, la sémantique de Heyting (section 2.6). Le lecteur familier de ces concepts pourra éviter de lire les premiers chapitres. Le dernier chapitre est une courte présentation de l'assistant de preuve Coq, dans lequel on illustre certains principes expliqués jusque là.

La partie II présente le calcul des constructions inductives avec types normalisés ainsi que les preuves de ses différentes propriétés.

Enfin la dernière partie décrit un développement en Coq d'une preuve d'algorithme auto-stabilisant. On y trouvera notamment quelques définitions et résultats concernant la réécriture sur les mots.

Première partie

Du λ-calcul au calcul des constructions inductives

Dans cette partie, nous donnons une description des systèmes suivants :

— le λ-calcul pur (λ),

— le λ-calcul simplement typé ($\lambda_{\rightarrow}$),

— le calcul des constructions ($\mathcal{CC}$),

— enfin le calcul des constructions inductives($\mathcal{CCI}$).

La présentation se veut progressive, le lecteur familier de ces systèmes pourra se dispenser de lire le début de ce chapitre. Nous recommandons cependant la lecture des chapitres 3 et 4. Nous y donnons une description de $\mathcal{CC}$ et $\mathcal{CCI}$, reprenant la présentation de [Wer94], qui sera reprise dans la partie II de cette thèse lors de la définition du calcul des constructions inductives avec types normalisés ($\mathcal{CCI}^{nf}$). Dans la section 3.7 nous décrivons les propriétés que nous pensons souhaitables pour qu'un calcul du type $\mathcal{CC}$ puisse servir de base à un assistant de preuve. Pour une présentation approfondie du λ-calcul typé et de ses différentes extensions, l'ouvrage de Henk Barendregt [Bar92] reste la référence en la matière.

Le dernier chapitre de cette section est une présentation de l'assistant de preuve Coq, dans laquelle nous illustrons les principes expliqués jusque là, en particulier l'utilisation de l'isomorphisme de Curry-Howard pour représenter les preuves.

Chapitre 1

Le λ-calcul pur

Proposé par Alonzo Church dans les années 1930-40 [Chu40], le λ-calcul était – comme nous l'avons dit plus haut – supposé servir à représenter les objets mathématiques de la "logique d'ordre supérieur". Ces objets étaient codés à l'aide d'une notion de base unique : la fonction, elle-même définie à partir de deux opérations élémentaires : *l'abstraction* et *l'application*. D'où la présence de deux constructions syntaxiques dans la grammaire du λ-calcul ci-dessous.

1.1 Grammaire

On se donne un ensemble infini dénombrable de *noms de variables* :

$$V ::= x, y \ldots$$

La grammaire du λ-calcul est la suivante :

$$t ::= V \mid \lambda V.t \mid (t\ t)$$

Par exemple, x, $\lambda x.x$ et $(\lambda x.\lambda y.(z\ r)\ \lambda t.w)$ sont des termes du λ-calcul. On a vu dans l'introduction que :

- La λ-abstraction $\lambda x.t$ sert à déclarer l'argument d'une fonction et à lui donner un nom temporaire. Ce nom est valable uniquement à l'intérieur de la définition de la fonction, donc le choix de ce nom ne change pas la signification de la fonction ; c'est l'α-équivalence, que nous définissons plus bas.
- L'application $(t\ t')$ sert à appliquer une fonction t à un argument réel t', ce qui mène à une étape de *calcul* consistant à remplacer le nom abstrait de l'argument par l'argument réel. Cette étape de calcul s'appelle la β-réduction, nous la définissons également plus bas.

Définition 1.1.1 On notera $\mathcal{V}$(t) l'ensemble des variables d'un terme.

On voit que dans un terme comme $\lambda x.y$ la variable y n'est pas abstraite par un λ avant d'être utilisée, elle ne pourra donc pas être utilisée dans le calcul. On dira que y est une *variable libre* dans $\lambda x.y$. Plus précisément :

Définition 1.1.2 ($\mathcal{FV}_\lambda$) On définit l'ensemble des variables ayant une occurrence libre dans un terme t, $\mathcal{FV}_\lambda(t)$, par induction :

$$\begin{array}{lcl} \mathcal{FV}_\lambda(x) & = & \{x\} \\ \mathcal{FV}_\lambda((t_1\ t_2)) & = & \mathcal{FV}_\lambda(t_1) \cup \mathcal{FV}_\lambda(t_2) \\ \mathcal{FV}_\lambda(\lambda x.t) & = & \mathcal{FV}_\lambda(t) \setminus \{x\} \end{array}$$

Les occurrences d'une variable x dans un terme t sont ainsi réparties dans deux catégories : les occurrences libres et non libres (liées) de x dans t. λ, qui transforme une variable libre en une variable liée, est parfois qualifié de *lieur*. On dira que le lieur λ *capture* la variable x dans le sous-terme t du terme $\lambda x.t$.

Remarque 1.1.1 *En pratique on supposera dans toutes la suite qu'une variable aura soit que des occurrences libres soit que des occurrences liées. Par conséquent la définition ci-dessus caractérisera les variables libres d'un terme.*

On peut définir l'α-équivalence :

Définition 1.1.3 Deux termes t_1 et t_2 sont α-équivalents s'ils ne diffèrent que par renommage de variables liées n'introduisant pas de capture. On notera alors $t_1 \equiv_\alpha t_2$.

Par exemple $\lambda x.x$ et $\lambda y.y$ sont α-équivalents. Une propriété bien connue de cette équivalence est la suivante :

Propriétés 1.1.4 L'α-équivalence est une relation d'équivalence décidable dans les différents calculs présentés dans cette thèse.

L'étude des problèmes dus à l'α-équivalence et à la capture de variables constitue un domaine de recherche actif aujourd'hui, notamment avec la définition de calculs avec *substitutions explicites*. Nous ne nous intéressons cependant pas à cet aspect des choses dans ce travail. Comme on sait par ailleurs qu'il existe des représentations des termes dans lesquels un terme n'a qu'un seul représentant modulo α-équivalence (comme par exemple dans la représentation des λ-abstractions à l'aide d'indices de De Bruijn [dB72]), on décide d'éviter ces problèmes :

Remarque 1.1.2 *Dans la suite, on ne fera plus référence à l'α-équivalence, mais toutes les définitions seront faites* modulo *α-renommage, c'est-à-dire que deux termes seront équivalents (c'est-à-dire considérés* identiques*) s'ils sont α-équivalents.*

1.2 β-réduction

La β-réduction a pour but de remplacer l'argument abstrait d'une fonction par l'argument réel auquel on applique la fonction. Pour la définir, il faut d'abord préciser la notion de remplacement d'une variable par un terme. C'est la *substitution* : étant donnés deux termes t et u et une variable x, on définit $t[x \leftarrow u]$ le terme t dans lequel toutes les occurrences libres de x ont été remplacées par u. Plus formellement :

Définition 1.2.1 (Substitution) On définit l'opération de substitution d'une variable x par un terme u dans un terme t, notée $t[x \leftarrow u]$, par induction sur t :

$$\begin{array}{lcll} x[x \leftarrow u] & = & u & \\ x[y \leftarrow u] & = & x & \text{si } x \neq y \\ \lambda x.t'[x \leftarrow u] & = & \lambda x.t' & \\ \lambda x.t'[y \leftarrow u] & = & \lambda x.(t'[y \leftarrow u]) & \text{si } x \neq y \text{ et } x \notin \mathcal{FV}_\lambda(u) \\ (t'\ t'')[x \leftarrow u] & = & (t'[x \leftarrow u]\ t''[x \leftarrow u]) & \end{array}$$

On définit la règle de réduction du λ-calcul :

Définition 1.2.2 (β-réduction λ-calcul pur) Pour tous termes t et u et toute variable x, on définit la β-réduction comme la clôture par contexte de la règle suivante :

$$(\lambda x.t\ u) \longrightarrow_\beta t[x \leftarrow u]$$

Il est bien connu que la β-réduction sur le λ-calcul non typé ne termine pas toujours, comme le montre le contre-exemple classique suivant :

Remarque 1.2.1 *Le λ-terme $(\lambda x.(x\ x)\ \lambda x.(x\ x))$ se réduit en lui-même, donc il existe des réductions infinies dans le λ-calcul non typé.*

Pour remédier à ce problème, on raffine la notion de terme avec celle de terme *bien formé*, ou terme *typable*.

Chapitre 2

Le λ-calcul simplement typé ($\lambda_\rightarrow$)

Le but du typage dans un calcul est double :

1. Assurer une certaine cohérence entre l'argument attendu d'une fonction (autrement dit les occurrences liées d'une variable dans un terme) et l'argument auquel on l'applique. Ce but est celui du typage dans tout langage de programmation typé.
2. Caractériser les termes ayant des propriétés particulières. Dans le cas du λ-calcul simplement typé, on cherche à isoler une classe de termes pour lesquels il n'y a pas de réduction infinie. Ce but est spécifique aux systèmes de preuve. Dans les langages de programmation, on n'interdit évidemment pas les programmes qui ne terminent pas.

2.1 Grammaire

La grammaire ressemble à celle du λ-calcul pur (on parlera de grammaire *à la Church*) mais on a maintenant des annotations de type sur les λ-abstractions. Comme certains objets définis par la grammaire ne seront pas bien formés (on ne pourra pas leur associer un type), on appellera ces objets les *pré-termes*, et seuls les pré-termes bien formés seront appelés termes.

On se donne un ensemble de types de base, dits *atomiques*, dont la définition importe peu du moment qu'il y a au moins un type atomique. À partir de cet ensemble, on définit la grammaire des types :

$$T ::= u \mid T \rightarrow T \quad \text{où } u \text{ est un type atomique.}$$

Remarque 2.1.1 *Tout type engendré par cette grammaire est correct. Dans le λ-calcul simplement typé, la notion de type bien ou mal formé n'existe pas, car l'ensemble des types et celui des termes sont disjoints. Il n'en sera pas de même pour les calculs plus expressifs comme le calcul des constructions décrit plus loin.*

La grammaire des pré-termes est souvent donnée sous la même forme que celle du λ-calcul pur avec une λ-abstraction de la forme $\lambda x : T.t$ ou $\lambda x^T.t$. Nous préférons néanmoins noter la λ-abstraction avec des crochets ($[x : T]t$) ; cette notation est celle utilisée traditionnellement dans le calcul des constructions et il est donc préférable de l'utiliser dans toute la thèse. Voici la grammaire :

$$t ::= V \mid [V : T]t \mid (t\ t)$$

Par exemple : x, $[x : \iota]x$ et $([x : \iota]y \ \ [z : \iota \rightarrow \iota]t)$ sont des pré-termes si ι est un type atomique.

2.2 β-réduction

Les définitions de variable libre ($\mathcal{FV}_{\lambda_\rightarrow}$), de substitution et de la β-réduction ne changent pas par rapport à λ, mis à part l'ajout de l'annotation de type. On redonne ici uniquement la définition de substitution et la β-réduction :

Définition 2.2.1 (Substitution dans $\lambda_\rightarrow$) On définit l'opération de substitution d'une variable x par un terme u dans un terme t, notée $t[x \leftarrow u]$, par induction sur t :

$$\begin{array}{lcll} x[x \leftarrow u] & = & u & \\ x[y \leftarrow u] & = & x & \text{si } x \neq y \\ [x:T]t'[x \leftarrow u] & = & [x:T]t' & \\ [x:T]t'[y \leftarrow u] & = & [x:T](t'[y \leftarrow u]) & \text{si } x \neq y \\ (t'\ t'')[x \leftarrow u] & = & (t'[x \leftarrow u]\ t''[x \leftarrow u]) & \end{array}$$

On voit que les types sont ignorés lors de la substitution. En effet, comme nous l'avons déjà dit, dans le λ-calcul simplement typé les ensembles de termes et de types sont disjoints.

Définition 2.2.2 (β-réduction λ-calcul simplement typé) Pour tous termes t et u, toute variable x et tout type T, on définit la β-réduction comme la clôture par contexte de la règle suivante :

$$([x:T]t\ u) \longrightarrow_\beta t[x \leftarrow u]$$

On voit ici que la réduction est définie sur les pré-termes puisque rien n'oblige le terme réduit à être bien typé. Nous verrons plus bas que c'est cependant sur les termes typables que la réduction a toutes les propriétés souhaitables. Signalons que la confluence sur les pré-termes est en général souhaitable.

2.3 Typage

La relation de typage s'exprime traditionnellement sous la forme d'un couple $t : T$ signifiant «t est de type T». On généralise cette notation aux *jugements de typage*, qui nécessitent la notion d'*environnement de typage*.

Définition 2.3.1 (Environnement de typage) Un environnement de typage est une suite de couples de la forme $(x : T)$, où T est un type et x une variable. De même qu'on ne s'occupe pas des problèmes de capture de variable, on supposera tout au long de cette thèse qu'il n'y a pas deux fois la même variable dans un environnement. On notera :

- $[]$ l'environnement de typage vide,
- $\Gamma :: (x : T)$ l'environnement de typage Γ auquel on a ajouté l'élément $(x : T)$,
- $\Gamma_1\Gamma_2$ l'union des deux environnements de typage Γ_1 et Γ_2,
- $(x : T) \in \Gamma$ pour signifier que le couple $(x : T)$ est dans Γ, on dira alors que x est lié dans Γ (noté $x \in \Gamma$), et on notera $\Gamma(x)$ le type auquel il est lié (ici $\Gamma(x) = T$).

Définition 2.3.2 (Jugement de typage) Un jugement de typage est un triplet de la forme :

$$\Gamma \vdash t : T$$

où Γ est un environnement de typage, t un terme et T un type. On lira : «dans l'environnement de typage Γ, t est de type T».

Remarque 2.3.1 *Dans toute la thèse, on utilisera $\vdash$ dans les jugements de typage concernant le système de typage dont il est question dans la section courante. Lorsqu'on fera référence à un autre système que celui de la section en cours, on utilisera le symbole $\vdash$ indicé par le nom du calcul. Par exemple, en dehors de la présente section, on écrira $\Gamma \vdash_{\lambda\to} t : T$ pour dire que dans le λ-calcul simplement typé, t a le type T dans Γ.*

Un jugement de typage doit être vérifié au moyen d'un *système de typage*. On donne le système de typage des pré-termes du λ-calcul simplement typé à la figure 2.1 sous la forme de règles de typage :

Définition 2.3.3 (Règle de typage) Une règle de typage est une expression de la forme :

$$\frac{J_1 \dots J_n \ ; \ C}{J}$$

où J, $J_1 \dots J_n$ sont des jugements de typage et C une condition. On lit une telle règle de la manière suivante «Si on peut dériver les jugements $J_1 \dots J_n$, alors, si la condition C est vérifiée, on peut dériver J».

On voit apparaître ici la parenté de ce genre de système avec les systèmes de déduction cités dans l'introduction (Curry-Howard) ; les jugements J_i correspondent aux prémisses d'une règle logique, et J à la conclusion. On reconnaît d'ailleurs déjà la règle du modus ponens citée à l'introduction dans la règle (App) de la figure 2.1. On s'attardera sur l'isomorphisme de Curry-Howard à la section 2.6.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash t : T \to U \quad \Gamma \vdash u : T}{\Gamma \vdash (t\ u) : U}\ (\textsc{App})$$

$$\frac{\Gamma :: (x : T) \vdash t : U}{\Gamma \vdash [x : T]t : T \to U}\ (\textsc{Lam})$$

Figure 2.1 – Typage du λ-calcul simplement typé

La règle Var permet de typer une variable à condition qu'elle appartienne à l'environnement de typage. App permet de typer l'application d'un argument à un fonction, à condition que le premier soit du type attendu par la deuxième. Enfin la règle Lam permet de donner un type flèche

à une abstraction, pourvu qu'elle soit bien formée, c'est-à-dire que le corps de la fonction doit être lui-même typable dans le contexte contenant la variable abstraite.

Pour typer un terme t il faut exhiber une *dérivation* de typage, c'est-à-dire une séquence arborescente d'application de ces trois règles aboutissant à un jugement de la forme $\Gamma \vdash t : T$. T est alors appelé type de t dans l'environnement Γ. Par exemple voici une dérivation de typage pour le terme $[x : \iota]x$:

$$\dfrac{\dfrac{}{x : \iota \vdash x : \iota}\,(\textsc{Var})}{[] \vdash [x : \iota]x \ : \ \iota \to \iota}\,(\textsc{Lam})$$

On voit ici que l'environnement de typage est vide dans la conclusion de la dérivation. Il est aussi facile de voir que les termes typables dans un environnement vide vérifient les propriétés suivantes :

— ils sont typables dans n'importe quel environnement ;

— ils n'ont pas de variables libres.

L'exemple suivant montre la structure arborescente que prendra généralement une dérivation à cause de la double prémisse de la règle (App) :

$$(\textsc{Lam})\,\dfrac{\dfrac{\dfrac{}{y : \kappa; x : \iota \vdash x : \iota}\,(\textsc{Var})}{y : \kappa \vdash ([x : \iota]x) \ : \ \iota \to \iota} \quad \dfrac{}{y : \iota \vdash y : \iota}\,(\textsc{Var})}{y : \kappa \vdash ([x : \iota]x \ \ y) \ : \ \iota}\,(\textsc{App})$$

où κ est un type quelconque.

2.4 Propriétés

On donne ici les principaux résultats sur le λ-calcul simplement typé. Premièrement, la relation de typage est décidable :

Théorème 2.4.1 (Décidabilité du typage de $\lambda_\to$) Il existe un algorithme qui, étant donnés un λ-terme t et un environnement de typage Γ :

— rend un type T tel que $\Gamma \vdash t : T$ s'il en existe un,

— échoue sinon.

La réduction est compatible avec la relation de typage :

Théorème 2.4.2 (Subject reduction) Soient deux termes t_1 et t_2 de $\lambda_\to$ tels que $t_1 \longrightarrow_\beta t_2$, si $\Gamma \vdash t_1 : T$ alors $\Gamma \vdash t_2 : T$.

Sur les termes typables, la réduction termine toujours :

Théorème 2.4.3 ($\mathcal{SN}(\lambda_\to)$) Soit t un terme typable du λ-calcul simplement typé (c'est-à-dire qu'il existe Γ et T tels que $\Gamma \vdash t : T$), alors il n'existe pas de réduction infinie partant de t. Autrement dit $\lambda_\to$ est fortement normalisant sur les termes bien formés.

La réduction est confluente sur les pré-termes :

Théorème 2.4.4 (Confluence de $(\lambda_\rightarrow)$) Soient t, t_1 et t_2 trois termes (éventuellement mal formés) du λ-calcul simplement typé tels que $t \longrightarrow_\beta^* t_1$ et $t \longrightarrow_\beta^* t_2$, alors il existe un terme t_3 tel que $t_1 \longrightarrow_\beta^* t_3$ et $t_2 \longrightarrow_\beta^* t_3$.

Une conséquence de ces deux derniers théorèmes est l'unicité de la forme normale.

Corollaire 2.4.5 Soit t un terme de $\lambda_\rightarrow$, et deux termes t_1 et t_2 en forme normale tels que $t \longrightarrow_\beta^* t_1$ et $t \longrightarrow_\beta^* t_2$, alors $t_1 = t_2$.

Nous verrons plus loin l'intérêt de ces propriétés dans l'optique d'utiliser un calcul comme fondement d'un assistant de preuve. Pour une discussion approfondie sur les systèmes de réduction par réécriture, on pourra se reporter notamment à [Klo92].

2.5 η-réduction

Il existe une autre notion de réduction, souvent considérée dans les λ-calculs, la η-réduction, qui permet d'éliminer les λ-abstractions trivialement inutiles :

Définition 2.5.1 (η-réduction) Pour tout terme t, toute variable x et tout type T, on définit la η-réduction comme la clôture par contexte de la règle suivante :

$$([x : T](t\ x)) \longrightarrow_\eta t \qquad \text{si } x \notin \mathcal{FV}(t)$$

La conséquence principale, au niveau des propriétés générales du calcul, de l'ajout de cette réduction est que la confluence n'est plus vérifiée sur tous les pré-termes comme le montre le contre-exemple suivant dû à Nederpelt [Ned73].

Remarque 2.5.1 (Contre-exemple de Nederpelt) *Soit t le terme $[x : A]([y : B]y)\ x)$ où A et B sont deux types différents, t se réduit par $\longrightarrow_\beta$ vers $[x : A]x$ et par $\longrightarrow_\eta$ vers $[y : B]y$. Ces deux termes sont en forme normale et pourtant inégaux. On en conclut que $\longrightarrow_{\beta\eta}$=$\longrightarrow_\beta \cup \longrightarrow_\eta$ n'est pas confluente sur les pré-termes.*

La confluence reste néanmoins démontrable sur les termes bien formés. Par souci de clarté et bien qu'il semble possible d'inclure la η-réduction dans notre travail, nous n'en parlerons pas au cours de cette thèse.

2.6 L'isomorphisme de Curry-Howard

2.6.1 La correspondance type/propriété

Le λ-calcul est un formalisme qui permet de décrire des *algorithmes*. L'isomorphisme de Curry-Howard nous montre que les démonstrations et les algorithmes sont deux facettes d'un même concept, et qu'ils peuvent se représenter de manière uniforme par des λ-termes. Pour illustrer cela, nous allons prendre trois règles de déduction logique (issues du système de déduction naturelle de D. Prawitz [Pra65]) et les trois règles de typage de $\lambda_\rightarrow$ correspondantes, et nous allons décortiquer l'étroite correspondance existant entre les deux formalismes.

Déduction naturelle	λ-calcul simplement typé
$\dfrac{A \in \Gamma}{\Gamma \vdash A}$	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$ (VAR)
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash u : A \quad \Gamma \vdash t : A \to B}{\Gamma \vdash (t\ u) : B}$ (APP)
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma :: (x : A) \vdash t : B}{\Gamma \vdash [x : A]t : A \to B}$ (LAM)

Il est aisé de constater la correspondance syntaxique entre les règles de la première et de la deuxième colonne : les propriétés dans la première colonne correspondent exactement aux types dans la deuxième. Cette correspondance va même plus loin, la β-réduction dans un λ-terme correspond à l'opération d'*élimination des coupures* dans une déduction. Pour une étude étendue de l'isomorphisme de Curry-Howard, on pourra consulter [SU98].

2.6.2 La sémantique de Heyting-Kolmogorov

Cette correspondance type/propriété est la base de l'isomorphisme de Curry-Howard. Pour bien comprendre sa signification, il faut en premier lieu comprendre la nature d'une propriété du point de vue intuitionniste. Comme l'écrit Martin-Löf dans *Intuitionistic Type Theory* [ML84], pour les intuitionnistes, une propriété n'est pas définie comme une valeur de vérité – soit vrai, soit faux – mais plutôt par *la description de ce que devrait être une preuve de cette propriété*, ce qui est plus précis et plus *constructif*.

C'est dans cet esprit qu'il faut aborder la sémantique de Heyting décrite ci-dessous. Elle utilise la notion de *preuve canonique*, on entend par là «preuve sans utilisation de preuve intermédiaire». Un résultat connu est qu'une preuve quelconque se réduit toujours à une preuve canonique [1]. L'un des intérêts de cette interprétation est de mettre en lumière le fait que les preuves elles-mêmes peuvent être considérées comme des objets du discours, manipulables au même titre que les entiers ou les fonctions.

1. Une preuve canonique de $A \Rightarrow B$ est une fonction qui construit, à partir d'une preuve (quelconque) de A, une preuve de B.
2. Une preuve canonique de $A \wedge B$ est une paire $< p_A, p_B >$ où p_A est une preuve de A et p_B une preuve de B.
3. Une preuve canonique $p_{A \vee B}$ de $A \vee B$ a deux formes possibles :
   (a) Soit $p_{A \vee B} =< g, p_A >$ où p_A est une preuve de A ;
   (b) Soit $p_{A \vee B} =< d, p_B >$ où p_B est une preuve de B.

   Les membres gauches des deux paires, g et d, permettent d'indiquer de quelle propriété le membre droit est une preuve. Une preuve doit en effet contenir tous les détails qui permettent

1. Du moins dans les systèmes logiques où le théorème dit d'*élimination des coupures* est démontré, ce qui est le cas dans la plupart des systèmes intéressants.

de vérifier sa validité. On voit ici que pour montrer $A \vee B$ dans ce cadre, *il faut soit savoir prouver A, soit savoir prouver B*. Ce point de vue limité sur la disjonction, qui restreint l'utilisation du tiers exclu [2], est un des traits caractéristiques de *l'intuitionnisme*.

4. Une preuve canonique de $\forall x \in A.P(x)$ est une fonction qui prend en argument un élément quelconque de A et rend une preuve de $P(x)$. On voit ici encore le paradigme "les preuves sont des objets" (*proof as object*).
5. Une preuve canonique de $\exists x \in A.P(x)$ est une paire $< a, h >$, où a est un élément particulier (le *témoin*) de A et h une preuve de $P(a)$. Comme pour la disjonction, c'est la notion intuitionniste de l'existentielle qui est exprimée ici : pour prouver une formule existentielle, *il faut exhiber un témoin*.

Pour l'instant, concentrons-nous sur le premier point, la preuve de $A \Rightarrow B$. La sémantique de Heyting stipule qu'une telle preuve, si elle est canonique, est une fonction prenant en argument une preuve de A et rendant une preuve de B. Regardons maintenant la règle de typage correspondante : (LAM). Cette règle permet de typer les termes de la forme $[x : A]t$, c'est-à-dire les fonctions. Plus précisément $[x : A]t$ est justement une fonction qui prend en argument un terme de type A et rend un terme de type B.

Il est évidemment tentant d'identifier les deux concepts "avoir pour type" et "être une preuve de". On voit alors qu'on peut prendre $[x : A]t$ elle-même comme fonction prenant en argument une preuve de A et rendant une preuve de B, autrement dit comme preuve de $A \Rightarrow B$. On s'aperçoit donc que non seulement la sémantique de Heyting met en lumière la correspondance de Curry-Howard, mais réciproquement les λ-termes *matérialisent* en quelque sorte la notion de preuve définie par Heyting.

Finalement voici une récapitulation de quelques-uns des principes de l'isomorphisme de Curry-Howard :

— Les preuves sont des objets comme les autres, sur lesquels on peut construire des fonctions rendant d'autres preuves. Il se trouve qu'il est parfaitement adéquat de les représenter par des λ-termes.

— Cette représentation des preuves par des λ-termes nous amène à identifier les types et les propriétés. Un terme de type A est une preuve de la propriété A. Par conséquent une propriété démontrable est un type habité (c'est-à-dire qu'il existe un terme ayant ce type) et une propriété indémontrable est un type non habité.

— Si on a le jugement de typage $x_1 : A_1; x_2 : A_2 \ldots; x_n : A_n \vdash t : A$, on peut l'interpréter de la manière suivante : si on a des preuves x_1,$x_2 \ldots x_n$ de A_1, $A_2 \ldots A_n$, alors on peut les combiner pour construire la preuve t de A. C'est finalement un raffinement du jugement logique $A_1; A_2 \ldots; A_n \vdash A$, puisque *t montre comment construire la preuve de A à partir des preuves des A_i dans la sémantique de Heyting*.

2.6.3 L'extension aux autres connecteurs

Une fois le cas de la flèche compris, le lecteur se demandera pourquoi les autres connecteurs n'ont pas de règles de typage correspondantes dans le λ-calcul. En fait, en exploitant la sémantique de Heyting pour les connecteurs logiques $\wedge$ et $\vee$ (cas 2 et 3 ci-dessus), on peut définir par l'isomorphisme de Curry-Howard un λ-calcul un peu plus riche que le λ-calcul simplement typé, que nous appellerons $\lambda'_{\rightarrow}$. On ajoute trois constructions syntaxiques. L'une, $< t, t >$, correspond à la paire permettant

2. En fait, on ne peut pas supposer $A \vee \neg A$ sans avoir un algorithme permettant de décider si A ou $\neg A$.

de prouver la conjonction, π_1 et π_2 représentant les projections à droite et à gauche des paires. Les deux autres, $inl_T(t)$ et $inr_T(t)$, correspondent aux deux paires permettant de prouver la disjonction, la construction Case ... of représentant l'analyse par cas permettant d'effectuer une preuve par cas sur une disjonction. Ces deux nouvelles constructions permettent de définir les deux nouvelles constructions de type $u * u$ pour la conjonction, $u + u$ pour la disjonction.

$$T ::= u \mid u \to u \mid u * u \mid u + u$$
$$t ::= V \mid [x : T]t \mid (t\ t) \mid < t, t > \mid \pi_1(t) \mid \pi_2(t) \mid inl_T(t) \mid inr_T(t) \mid \text{Case } t \text{ of } t | t \text{ end}$$

Nous donnons figure 2.2 les règles de typage de ce calcul, ainsi que les règles logiques correspondantes, qui correspondent en fait au système de déduction naturelle de Dag Prawitz.

Cette représentation des preuves par des λ-termes est non seulement élégante et concise, mais elle est relativement adaptée à une implantation sur machine. C'est d'ailleurs de cette manière que les démonstrateurs tels que Coq représentent les preuves. Un grand avantage de cette représentation est qu'en typant un terme, on vérifie de manière simple (*décidable*) la justesse de la démonstration qu'il représente. Le système Coq est décrit dans le chapitre 5.

Il est possible de poursuivre avec les connecteurs $\forall$ et $\exists$, mais cela nécessite une extension plus conséquente du système de type, dont nous décrivons un exemple dans le prochain chapitre avec le calcul des constructions.

Déduction naturelle	Typage du λ-calcul
$\dfrac{A \in \Gamma}{\Gamma \vdash A}$	$\dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{\Gamma :: (x : A) \vdash t : B}{\Gamma \vdash [x : A]t : A \to B}$
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash u : A \quad \Gamma \vdash t : A \to B}{\Gamma \vdash (t\ u) : B}$
$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$	$\dfrac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash < t, u > : A \star B}$
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$	$\dfrac{\Gamma \vdash t : A \star B}{\Gamma \vdash \pi_1(t) : A}$
$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$	$\dfrac{\Gamma \vdash t : A \star B}{\Gamma \vdash \pi_2(t) : B}$
$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$	$\dfrac{\Gamma \vdash t : A}{\Gamma \vdash inl_B(t) : A + B}$
$\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$	$\dfrac{\Gamma \vdash t : B}{\Gamma \vdash inr_A(t) : A + B}$
$\dfrac{\Gamma, A \vdash H \quad \Gamma, B \vdash H \quad \Gamma \vdash A \vee B}{\Gamma \vdash H}$	$\dfrac{\Gamma, x : A \vdash g : H \quad \Gamma, x : B \vdash d : H \quad \Gamma \vdash t : A + B}{\Gamma \vdash \text{case } t \text{ of } g \mid d \text{ end} : H}$

FIGURE 2.2 – Correspondance de Curry-Howard entre $\lambda'_{\to}$ et la déduction naturelle

Chapitre 3

Le calcul des constructions

3.1 Le type produit

Le calcul des constructions [Coq85], [CH88] est une extension du λ-calcul simplement typé dans laquelle le type flèche (appelé le type *produit*) a subi une extension. Pour justifier cette extension, rappelons la règle de typage des fonctions dans le λ-calcul simplement typé :

$$\frac{\Gamma :: (x : T) \vdash_{\lambda\rightarrow} t : U}{\Gamma \vdash_{\lambda\rightarrow} [x : T]t : T \rightarrow U}$$

Cette règle ne permet pas de définir des *types dépendants*, c'est-à-dire des types, ou plutôt des familles de types, paramétrées par des termes. On voudrait en effet définir par exemple la famille de types *tab*, telle que (tab 0) soit le type des tableaux de taille 0, (tab 1) soit le type des tableaux de taille 1, etc.

Non seulement on s'aperçoit que l'expression (tab 0) ne fait pas partie du langage des types, mais elle rend possible la définition de fonctions ayant un type dépendant de leur argument, ce qui ne peut pas être exprimé non plus avec le langage de type de $\lambda_\rightarrow$. Par exemple la fonction f prenant en argument un entier n et rendant un tableau de taille n rempli de 0 ne peut pas être considérée de type $nat \rightarrow tab$, puisque ce type ne rend pas compte du type effectivement rendu par la fonction qui est (tab n), où n est l'argument donné à la fonction.

Mieux que cela, la sémantique de Heyting précise que la démonstration d'une propriété $\forall x \in A.P(x)$ est une fonction qui prend en argument un élément de A et rend une preuve de $P(x)$. On voit qu'une telle fonction, une fois passée au crible de l'isomorphisme de Curry-Howard, devrait prendre un terme t de type A et rendre un terme de type (P t), ce qui signifie que nous avons besoin de pouvoir appliquer P à un terme t. Mais P est un type puisqu'il représente une propriété. Or dans $\lambda_\rightarrow$, le langage des termes et celui des types sont disjoints et ce dernier est restreint à une seule construction : $A \rightarrow B$.

Il nous faut donc étendre le langage des types. D'après les exemples ci-dessus, il est nécessaire d'avoir un mécanisme d'abstraction/application sur les types, comparable à celui que nous avons sur les termes. Il nous faut même un mécanisme qui nous permette d'effectuer ces abstractions/applications *entre termes et types*.

On peut constater les choses suivantes :

1. Il faut réunir les grammaires de termes et de types en une seule, avec le paradigme qui jusque là était réservé aux termes : *un terme ou un type est bien formé s'il a lui-même un type* ;

2. Ceci entraîne la nécessité d'une notion de type pour les termes, de type pour les types, mais aussi de type pour les familles de types comme *tab*. En effet, pour que *tab* soit considérée bien formée, il faut qu'elle soit elle-même typable. On voit que cette hiérarchie ne s'arrête a priori pas, et qu'il faut définir une infinité *d'univers* de types, pour les types des types de famille de types, etc. La présentation classique de $\mathcal{CC}$ ne comporte cependant que deux niveaux (en plus des types de termes), appelés *sortes*. Ces deux sortes sont en général notées Type et Kind. Il existe des formulations de $\mathcal{CC}$ ayant une hiérarchie d'univers infinie, où la sorte Type est paramétrée par un entier i indiquant son niveau dans la hiérarchie. En ce qui nous concerne, conformément à l'approche de Benjamin Werner pour le calcul des constructions inductives [Wer94], nous prendrons une hiérarchie à trois sortes : Set, Type et Extern, où Extern est une sorte non typable, ce qui est équivalent à la présentation classique.

3. Ce langage unique doit permettre d'exprimer les dépendances de types nécessaires pour :
   — être capable de typer les familles de types telles que *tab* ci-dessus ;
   — exprimer le type de la fonction f ci-dessus ou celui correspondant à la propriété $\forall x \in A.P(x)$, et ainsi étendre l'isomorphisme au quantificateur universel.

La notation du type flèche (ou type *produit*) est donc étendue. Au lieu de $nat \to tab$ on notera le type de f définie ci-dessus : $(n : nat)(tab\ n)$, ce qui rend bien compte que f prend un argument n de type *nat* et rend un tableau de taille n. La notation flèche devient donc un cas particulier de cette notation, pour le cas où le type de la fonction ne dépend pas de son argument.

On utilisera également cette notation comme traduction de la propriété $\forall x \in A.P(x)$ par l'isomorphisme de Curry-Howard, qui deviendra : $(x : A)(P\ x)$, où P est un type prenant en argument un terme de type A. On voit ici que les types dépendants permettent de représenter les *prédicats*.

3.2 Grammaire

La grammaire est donc commune aux types et aux termes, on a toujours un ensemble de variables x, $y\ldots$:

$$\begin{array}{rrl} \text{Variables :} & V ::= & x\ldots \\ \text{Sortes :} & S ::= & \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Extern} \\ \text{Termes :} & T ::= & V \mid S \mid [V : T]T \mid (V : T)T \mid TT \end{array}$$

3.3 Réduction

La réduction est toujours la β-réduction comme nous l'avons définie pour $\lambda_\to$, les définitions de variable libre et de substitution sont au préalable étendues à la nouvelle grammaire :

Définition 3.3.1 ($\mathcal{FV}_{\mathcal{CC}}$) On définit l'ensemble des variables libres d'un terme t, $\mathcal{FV}_{\mathcal{CC}}(t)$, par induction :

$$\begin{array}{lcl}
\mathcal{FV}_{\mathcal{CC}}(s) & = & \emptyset \text{ si } s \text{ est une sorte} \\
\mathcal{FV}_{\mathcal{CC}}(x) & = & \{x\} \\
\mathcal{FV}_{\mathcal{CC}}((t_1\ t_2)) & = & \mathcal{FV}_{\mathcal{CC}}(t_1) \cup \mathcal{FV}_{\mathcal{CC}}(t_2) \\
\mathcal{FV}_{\mathcal{CC}}([x:T]t) & = & \mathcal{FV}_{\mathcal{CC}}(T) \cup (\mathcal{FV}_{\mathcal{CC}}(t) \setminus \{x\}) \\
\mathcal{FV}_{\mathcal{CC}}((x:T)t) & = & \mathcal{FV}_{\mathcal{CC}}(T) \cup \mathcal{FV}_{\mathcal{CC}}(t) \setminus \{x\}
\end{array}$$

On voit que la construction $(x : T)t$ permet de *lier* la variable x dans le type t, de manière comparable à la λ-abstraction $[x : T]t$ pour le terme t. On constate également que les types peuvent maintenant contenir des variables, éventuellement libres.

La définition de substitution est étendue en conséquence :

Définition 3.3.2 (Substitution dans $\mathcal{CC}$) On définit l'opération de substitution d'une variable x par un terme u dans un terme t, notée $t[x \leftarrow u]$, par induction sur t :

$$\begin{array}{lcll}
x[x \leftarrow u] & = & u & \\
x[y \leftarrow u] & = & x & \text{si } x \neq y \\
[x:T]t'[x \leftarrow u] & = & [x:T[x \leftarrow u]]t' & \\
[x:T]t'[y \leftarrow u] & = & [x:T[x \leftarrow u]](t'[y \leftarrow u]) & \text{si } x \neq y \text{ et } x \notin \mathcal{FV}_{\mathcal{CC}}(u) \\
(x:T)t'[x \leftarrow u] & = & (x:T[x \leftarrow u])t' & \\
(x:T)t'[y \leftarrow u] & = & (x:T[x \leftarrow u])(t'[y \leftarrow u]) & \text{si } x \neq y \\
(t'\ t'')[x \leftarrow u] & = & (t'[x \leftarrow u]\ t''[x \leftarrow u]) &
\end{array}$$

La formulation de la β-réduction n'est quant à elle pas modifiée :

Définition 3.3.3 (β**-réduction dans** $\mathcal{CC}$) Pour tous termes t et u, toute variable x et tout type T, on définit la β-réduction comme la clôture par contexte de la règle suivante :

$$([x:T]t\ u) \longrightarrow_\beta t[x \leftarrow u]$$

3.4 Typage

3.4.1 Extension du typage de $\lambda_\rightarrow$

La règle de typage de l'abstraction de $\lambda_\rightarrow$ s'écrit comme suit :

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

mais dans $\mathcal{CC}$ il faut également s'assurer que les types T et U ainsi que le type construit, $(x : T)U$, sont bien formés. Comme le système de types doit être tel que si $(x : T)U$ est bien formé alors T et

U le sont aussi[1], on formule finalement la règle de la manière suivante :

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

où s est une sorte quelconque. Le jugement $\Gamma \vdash (x : T)U : s$ implique l'existence d'une règle permettant de *typer le type produit*, qui n'existait pas dans le système de types de $\lambda_{\rightarrow}$:

$$\frac{\Gamma :: (x : T) \vdash U : s}{\Gamma \vdash (x : T)U : s}$$

On voit que cette règle repose le problème de la règle naïve pour l'abstraction ci-dessus : il faut s'assurer que T est un type bien formé. De manière générale, il faut que les environnements de typage construits au cours des dérivations soient toujours bien formés. Dans le λ-calcul simplement typé, les environnements sont nécessairement bien formés puisque les types sont tous bien formés. Mais dans le calcul des constructions, il faut s'en assurer systématiquement ; par exemple dans la règle :

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

l'environnement Γ doit être bien formé.

Il existe plusieurs possibilités pour s'en assurer ; il est par exemple courant d'ajouter des règles de typage définissant la notion d'environnement bien formé :

$$\frac{}{[] \text{ bien formé}}$$

$$\frac{\Gamma \vdash T : \mathsf{Set} \text{ ou } \mathsf{Type}}{\Gamma :: (x : T) \text{ bien formé}}$$

Dans notre présentation, nous suivons la formulation équivalente de Benjamin Werner ([Wer94]) donnée à la figure 3.1. On voit que les règles (Prod-S) et (Lam-S) sont paramétrées par une sorte s.

Remarquons que le fait que s soit quelconque dans les règles (Prod-S) et (Lam-S) permet d'exprimer plus que les types dépendants expliqués plus haut. Cela permet également d'avoir un système de types incluant le *polymorphisme* et les *constructeurs de types*, que nous expliquons plus bas.

Remarque 3.4.1

- *Les systèmes de types purs (PTS) et les différents systèmes du célèbre cube de Barendregt [Bar92] correspondent à des restrictions des sortes possibles pour les sortes de t_1, t_2 et $(x : t_1)t_2$ dans la règle* (Prod-S).
- *De même bon nombre d'autres systèmes de types correspondent à des variations sur les sortes possibles de ces types, ainsi que sur le nombre de sortes et les relations de typage qui les relient.*

1. C'est une propriété évidemment souhaitable pour un système de types.

$$(\text{Ax}_1)[] \vdash \mathsf{Set} : \mathsf{Type} \qquad (\text{Ax}_2)[] \vdash \mathsf{Type} : \mathsf{Extern}$$

$$(\text{Prod-s})\frac{\Gamma :: (x : t_1) \vdash t_2 : s}{\Gamma \vdash (x : t_1)t_2 : s} \qquad (\text{Lam-s})\frac{\Gamma \vdash (x : t_1)t_2 : s \qquad \Gamma :: (x : t_1) \vdash t : t_2}{\Gamma \vdash [x : t_1]t : (x : t_1)t_2}$$

$$(\text{W-Set})\frac{\Gamma \vdash t : \mathsf{Set} \quad \Gamma \vdash A : B \quad a \notin \Gamma}{\Gamma :: (a : t) \vdash A : B} \qquad (\text{W-Type})\frac{\Gamma \vdash t : \mathsf{Type} \quad \Gamma \vdash A : B \quad \alpha \notin \Gamma}{\Gamma :: (\alpha : t) \vdash A : B}$$

$$(\text{Var})\frac{\Gamma \vdash t_1 : t_2 \qquad (x : t_1) \in \Gamma}{\Gamma \vdash x : t_1} \qquad (\text{App})\frac{\Gamma \vdash t_2 : (x : T_1)T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash (t_2\ t_1) : T_2[x \leftarrow t_1]}$$

$$(\text{Conv})\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_1 : s \quad \Gamma \vdash T_2 : s \quad T_1 \equiv_\beta T_2}{\Gamma \vdash t : T_2}$$

FIGURE 3.1 – Typage de $\mathcal{CC}$

Le *polymorphisme* et la possibilité de définir des *constructeurs de types* permettent d'utiliser les types comme paramètres de termes. L'utilité du polymorphisme dans un système de types est à peu près la même que dans les langages de programmation : on peut définir des objets paramétrés par un type, par exemple la fonction $id = [T : \mathsf{Set}][x : T]x$ de type $(T : \mathsf{Set})(T \to T)$ (de type Type), telle que (*id nat*) et (*id bool*) sont les fonctions identités sur les entiers et les booléens. Avec les constructeurs de type on peut définir des familles de types paramétrées par d'autres types. Par exemple, on peut définir la famille de types *list* de type $\mathsf{Set} \to \mathsf{Set}$ (lui-même de type Type, donc bien formé), telle que (*list nat*) soit le type des listes d'entiers, (*list bool*) soit le type des listes de booléens, etc.

La possibilité de définir de tels types dans le calcul des constructions en fait un formalisme extrêmement puissant, d'une expressivité comparable à celle de la théorie des ensembles, plus exactement l'expressivité de $\mathcal{CC}$ est la même que celle de *la logique des prédicats d'ordre supérieur* [Geu].

3.4.2 La règle de conversion

Le fait de pouvoir inclure des termes dans les types et des types dans les termes, ainsi que la possibilité de définir des fonctions sur les types et d'appliquer des types à d'autres types ont pour conséquence que la règle de réduction permet d'effectuer des étapes de calcul *à l'intérieur des types*. Un problème nouveau se pose alors, que nous illustrons par l'exemple suivant : le terme t de la forme $[y : ([x : T_1]x\ T_2)]y$, de type $([x : T_1]x\ T_2) \to ([x : T_1]x\ T_2)$, se réduit vers le terme $[y : T_2]y$ de type $T_2 \to T_2$.

Le type de t n'est pas conservé au sens strict par la réduction. Pourtant on voit bien que les deux types $[x : T_1]x\ T_2$ et T_2 sont équivalents dans un certain sens : l'un se réduit vers l'autre. Pour rendre compte de cette équivalence, les calculs tels que $\mathcal{CC}$ ont une règle de typage supplémentaire, appelée règle de *conversion* de la forme :

$$(\text{Conv})\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_1 : s \quad \Gamma \vdash T_2 : s \quad T_1 \equiv_{in} T_2}{\Gamma \vdash t : T_2}$$

où $\equiv_{in}$ est une relation congruente sur les termes que nous appellerons *égalité interne* ou *relation de conversion*, qui contiendra au moins la relation de réduction. On voit tout de suite que la décidabilité de $\equiv_{in}$ est une condition nécessaire à celle du typage.

Remarque 3.4.2 *La relation $\equiv_{in}$ contient toujours l'α-équivalence. On ne le signale pas car, comme nous l'avons dit plus tôt (remarque 1.1.2), toutes les définitions sont données modulo α-équivalence.*

3.5 L'isomorphisme de Curry-Howard

Nous allons maintenant voir ce que le passage de $\lambda_{\rightarrow}$ à $\mathcal{CC}$ signifie pour la sémantique de Heyting et pour l'isomorphisme de Curry-Howard.

3.5.1 Le produit

Nous avons déjà dit qu'un terme de type $(x : T)(P\ x)$ correspondait à une preuve de la propriété $\forall x.P(x)$, et constituait donc l'extension au quantificateur universel de l'isomorphisme de Curry-Howard. On donne ci-dessous un tableau illustrant cette extension. On voit que la nouvelle règle de typage de la λ-abstraction correspond à la fois à la quantification universelle (version dépendante) et à l'implication (version non dépendante (Lam-S)'). On a mis entre parenthèses la prémisse correspondant à la vérification du type construit, qui n'a pas de vis-à-vis en déduction naturelle.

Déduction naturelle	Calcul des constructions
$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$	$\dfrac{(\Gamma \vdash A : s) \qquad \Gamma :: (x : A) \vdash t : B}{\Gamma \vdash [x : A]t : A \rightarrow B}\ (\text{Lam-S})'$
$\dfrac{\Gamma \vdash B}{\Gamma \vdash \forall x.B}$ si $x \notin \Gamma$	$\dfrac{(\Gamma \vdash (x : A)B : s) \qquad \Gamma :: (x : A) \vdash t : B}{\Gamma \vdash [x : A]t : (x : A)B}\ (\text{Lam-S})$

On constate là encore une correspondance entre les deux règles. Le mélange termes/types fait apparaître des objets supplémentaires dans les règles de typage de $\mathcal{CC}$; par exemple la présence de $(x : A)$ dans la prémisse de la règle (Lam-S) permet d'être plus précis que la règle correspondante de la déduction naturelle, dans laquelle x n'est pas contraint.

3.5.2 La conversion

Il reste à comprendre la signification de la règle de conversion dans la sémantique de Heyting. Cette règle permet d'*identifier* des types, de manière qu'un terme de type T est aussi de type T' si $T \equiv_{in} T'$. Selon Curry-Howard, cela signifie qu'*une preuve d'une propriété P est aussi une preuve de la propriété P' si P et P' sont équivalentes suivant $\equiv_{in}$*. On a donc une notion d'équivalence sur les propriétés induite par l'équivalence sur les types. Il semble alors qu'il faille ajouter une règle supplémentaire à la sémantique de Heyting, dont nous proposons une formulation ici :

Remarque 3.5.1 *La sémantique de Heyting, donnée à la section 2.6.2, pour correspondre à la règle de conversion, peut être complétée de la manière suivante*[2] *:*

— *Une preuve canonique d'une propriété quelconque P est soit une preuve de P au sens de Heyting, soit une preuve d'une propriété équivalente à P.*

Par exemple, une preuve (canonique) que $33+3$ *est pair est aussi une preuve que* $30+6$, $18+18$ *et* 36 *sont pairs.*

3.6 Le problème de la conversion

Cette conséquence de la règle de conversion est extrêmement intéressante lorsqu'il s'agit d'implanter un assistant de preuve, puisqu'elle permet de remplacer une proposition par une autre proposition équivalente à n'importe quelle étape d'un raisonnement. En particulier, on peut remplacer un terme par un autre terme équivalent à l'intérieur d'une propriété. La méthode de raisonnement dite de *remplacement des égaux par les égaux* est donc implantée grâce à la règle de conversion.

On voit bien l'intérêt d'étendre $\equiv_{in}$ au maximum de manière à identifier le plus de propositions possible ; beaucoup considèrent que l'idéal serait de la faire correspondre avec l'égalité extensionnelle, qui stipule que deux propriétés sur un ensemble (ou un type) A sont équivalentes extensionnellement si elles sont vraies et fausses sur les mêmes éléments. Or cette notion d'égalité n'est bien entendu pas décidable, ce qui entraîne, on l'a vu, un typage indécidable si on l'inclut dans la conversion. On se trouve là devant un des dilemmes que posent les démonstrateurs basés sur l'isomorphisme de Curry-Howard :

— Soit on étend la conversion à l'égalité extensionnelle, comme dans le système Nuprl. Mais la vérification des preuves n'est plus décidable car l'égalité extensionnelle ne l'est pas. Une solution retenue par exemple dans PVS est de permettre au vérificateur de type d'engendrer des obligations de preuve lorsqu'il échoue sur un test de convertibilité (*type checking condition*). L'utilisateur doit alors fournir une preuve d'égalité au système.

— Soit on garde une conversion décidable, comme dans Coq, mais elle est fatalement très limitée. Un domaine de recherche consiste à définir des calculs dans lesquels l'égalité interne est la plus large possible tout en restant décidable [Dow99], [DHK98].

Remarque 3.6.1 *Dans le cadre du deuxième choix ci-dessus, le calcul proposé dans la deuxième partie de cette thèse a une égalité interne enrichie afin d'identifier les éléments équivalents des quotients.*

3.7 Les propriétés souhaitables d'un calcul

Nous énumérons ici les propriétés que nous considérerons souhaitables pour qu'un calcul puisse servir de base à un assistant de preuve du type Coq. Les deux propriétés primordiales à nos yeux sont :

— la cohérence du calcul,

— la décidabilité du typage (c'est-à-dire de la vérification des preuves).

2. À notre connaissance, cette constatation n'a jamais été faite et semble pourtant rendre plus explicite l'utilité de la règle de conversion. Signalons que pour être conforme au point de vue intuitionniste, il faudrait que l'équivalence entre les deux propriétés soit vérifiée par un algorithme.

3.7.1 Cohérence

Avant d'exprimer la première, il est nécessaire de préciser la notion de négation. Premièrement une propriété est fausse si une preuve de cette propriété entraîne l'absurde. On exprime en général cela de la manière suivante :

Définition 3.7.1 Une propriété est dite *fausse* si elle a pour conséquence que toute autre propriété est vraie. Autrement dit P est fausse si $P \to (Q : Type)Q$.

Remarque 3.7.1 *La propriété $(Q : Type)Q$ tient lieu de propriété absurde, on la notera par le symbole $\bot$ (appelé* bottom *ou* absurd*). On utilisera pour le calcul des constructions inductives une autre formulation de $\bot$: la propriété inductive à zéro constructeur* `False` *(section 5.4.)*

Par définition la propriété $\neg P$ négation d'une propriété P est vraie si P est fausse, ce qui s'exprime comme suit :

Définition 3.7.2 La négation $\neg P$ d'une propriété P s'exprime de la manière suivante :

$$P \to \bot$$

La cohérence d'un système (ou d'un calcul) logique s'énonce de la manière suivante :

Définition 3.7.3 On dira qu'un calcul est consistant si pour toute propriété P, il n'est pas possible de prouver à la fois (dans un environnement vide) P et $\neg P$.

L'autre formulation de la cohérence est énoncée par le lemme suivant :

Lemme 3.7.4 Un calcul admettant la règle du modus ponens est incohérent si et seulement si il admet une preuve de $\bot$ (dans un environnement vide).

Preuve : Trivial dans les deux sens :
- S'il existe une preuve H de P et une preuve H' de $\neg P = P \to \bot$, alors $(H'\ H)$ est la preuve de $\bot$ par le modus ponens.
- S'il existe une preuve $H_\bot$ de $\bot = (P : Type)P$, alors $(H_\bot\ P)$ est une preuve de P et $(H_\bot\ \neg P)$ est une preuve de $\neg P$.

___

3.7.2 Décidabilité du typage

La décidabilité de la vérification des preuve correspond à la décidabilité du typage ; nous avons vu qu'un typage décidable n'était pas absolument indispensable, mais nous considérerons cela comme une nécessité dans la suite, conformément à la philosophie du système Coq.

Définition 3.7.5 On dira qu'un calcul $\mathcal{C}$ a une inférence de type décidable s'il existe un algorithme qui, étant donnés un terme t et un environnement de typage Γ :
- rend un type T tel que $\Gamma \vdash_{\mathcal{C}} t : T$ si un tel type existe,
- échoue sinon.

Définition 3.7.6 On dira qu'un calcul $\mathcal{C}$ a une vérification de type décidable s'il existe un algorithme qui, étant donnés deux termes t et T et un environnement de typage Γ, rend
— true si $\Gamma \vdash_{\mathcal{C}} t : T$,
— false sinon.

Nous avons vu également que la décidabilité du typage nécessitait celle de la conversion. Bien entendu, la décidabilité de l'inférence de type et la décidabilité de la conversion entraînent la décidabilité de la vérification de type. Signalons que la plupart du temps, la décidabilité de la conversion est prouvée comme conséquence de la *confluence* et de la *normalisation forte* de la réduction, toutes deux définies plus bas. En effet, la conversion est en général définie comme la clôture congruente de la réduction, et ces deux propriétés garantissent *l'unicité de la forme normale*. La décision de la conversion se fait donc par simple comparaison des formes normales.

On précise maintenant les propriétés qui permettent généralement d'obtenir les deux précédentes. Dans les définitions suivantes, lorsqu'on parlera d'un système $\mathcal{C}$ quelconque, on notera $\vdash_{\mathcal{C}}$ sa relation de typage, $\longrightarrow_{\mathcal{C}}$ sa relation de réduction et $\equiv_{\mathcal{C}}$ sa relation de conversion.

3.7.3 Préservation du typage par réduction

La préservation de type (*subject reduction*) est la première propriété qu'on demande à un système de types : un terme contenant un calcul inachevé doit être du même type après chaque étape de calcul. Sans cette propriété basique, il n'y a a priori pas d'étude possible du calcul.

Définition 3.7.7 On dira qu'un calcul $\mathcal{C}$ vérifie la préservation de typage par réduction si, pour tous termes t_1 et t_2 de $\mathcal{C}$ tels que $t_1 \longrightarrow_{\mathcal{C}} t_2$, si $\Gamma \vdash_{\mathcal{C}} t_1 : T$ alors $\Gamma \vdash_{\mathcal{C}} t_2 : T$.

3.7.4 Compatibilité entre réduction et conversion - Confluence

La réduction doit être *incluse* dans l'égalité interne. En revanche, il ne semble pas nécessaire que la conversion se limite à la clôture réflexive, transitive, symétrique et par contexte de la réduction. Signalons à ce sujet qu'en toute rigueur l'α-équivalence, qui est comprise dans la conversion, ne fait pas partie de cette clôture. Il est possible, même si c'est rare, d'avoir une conversion plus riche que la clôture de la réduction (sans parler de l'α-conversion). C'est le cas en particulier dans le calcul des constructions inductives avec types normalisés ($\mathcal{CCI}^{nf}$), que nous définissons dans la deuxième partie de ce travail. Dans $\mathcal{CCI}^{nf}$, par exemple, les deux éléments 0 et 3 du quotient $\mathbb{Z}/3\mathbb{Z}$ sont convertibles, mais il ne se réduisent pas l'un vers l'autre par la réduction.

Définition 3.7.8 On dira qu'un calcul $\mathcal{C}$ a une notion de réduction $\longrightarrow_{\mathcal{C}}$ compatible avec sa conversion $\equiv_{\mathcal{C}}$ si pour tous termes t_1 et t_2, $t_1 \equiv_{\longrightarrow_{\mathcal{C}}} t_2 \Rightarrow t_1 \equiv_{\mathcal{C}} t_2$, où $\equiv_{\longrightarrow_{\mathcal{C}}}$ est la clôture réflexive, transitive, symétrique et par contexte de $\longrightarrow_{\mathcal{C}}$.
Remarquons que comme les deux relations sont des relations d'équivalence, il suffit que : $t_1 \longrightarrow_{\mathcal{C}} t_2 \Rightarrow t_1 \equiv_{\mathcal{C}} t_2$.

En général, les calculs ont une propriété plus forte que la compatibilité entre conversion et réduction : la conversion *est* la clôture congruente de la réduction par définition. Auquel cas la propriété ci-dessus est vérifiée par définition.

Afin de démontrer entre autres la décidabilité de la conversion, on demande en général que la réduction soit confluente :

Définition 3.7.9 On dira qu'une relation de réduction $\longrightarrow$ est confluente (respectivement confluente sur les pré-termes) si pour tous termes (respectivement pré-termes) t t_1 et t_2 tels que $t \longrightarrow t_1$ et $t \longrightarrow t_2$, alors il existe un terme t_3 tel que $t_1 \longrightarrow^* t_3$ et $t_2 \longrightarrow^* t_3$.
On dira d'un calcul qu'il est confluent (respectivement sur les termes bien formés) si sa réduction est confluente (respectivement sur les termes bien formés).

Une propriété un peu plus forte est la propriété de Church-Rosser :

Définition 3.7.10 On dira que la relation de réduction $\longrightarrow_{\mathcal{C}}$ d'un calcul $\mathcal{C}$ vérifie la propriété de Church-Rosser pour la conversion $\equiv_{\mathcal{C}}$ si pour tous termes t_1 et t_2 tels que $t_1 \equiv_{\mathcal{C}} t_2$, il existe un terme t_3 tel que $t_1 \longrightarrow^*_{\mathcal{C}} t_3$ et $t_2 \longrightarrow^*_{\mathcal{C}} t_3$.
On dira d'un calcul qu'il est Church-Rosser (respectivement sur les termes bien formés) si sa réduction est Church-Rosser (respectivement sur les termes bien formés).

Il est bien connu que la propriété de Church-Rosser est équivalente à celle de confluence si la réduction vérifie la propriété de normalisation forte, que nous définissons ci-dessous.

Lorsque la conversion d'un calcul n'est pas la clôture de la réduction (mais on a vu qu'elle la contient nécessairement), la confluence de la réduction n'est pas nécessairement vérifiée, et ne suffit de toute façon pas pour montrer la décidabilité de la conversion. Dans la partie II de cette thèse, nous montrerons à cet effet la confluence d'une autre réduction que celle du calcul, qui n'est pas confluente.

En toute généralité, la confluence n'est pas une propriété nécessaire pour qu'un calcul soit acceptable, il suffit que la conversion soit décidable. Cependant, en pratique, nous dirons qu'une certaine forme de confluence est nécessaire.

3.7.5 Normalisation forte et cohérence

Outre les propriétés de cohérence que la normalisation forte permet d'obtenir (voir plus bas), nous pensons que dans le cadre d'un assistant de preuve implanté sur machine, la réduction du système doit vérifier au moins une certaine forme de terminaison. En général, on demande la normalisation forte :

Définition 3.7.11 (Normalisation forte d'un terme) Un terme t est dit fortement normalisant si toute suite de réductions issue de t est finie. On notera $\mathcal{SN}$(t).

Définition 3.7.12 (Normalisation forte d'un calcul) On dit qu'un calcul $\mathcal{C}$ vérifie la propriété de normalisation forte si tous ses termes sont fortement normalisants ou, s'il s'agit d'un calcul typé, si ses termes bien formés (typables) sont fortement normalisants. On notera $\mathcal{SN}(\mathcal{C})$.

Nous donnons maintenant une idée de la preuve que la normalisation forte implique la cohérence du calcul. On trouvera une preuve détaillée dans [Bar92](Vol.2, section 5.2), qui s'applique entre autres à tous les systèmes du cube de Barendregt.

La démonstration se fait par l'absurde. Supposons qu'il existe une preuve H de $\bot$, donc telle que $\emptyset \vdash H : (P : Type)P$. Par la propriété de normalisation forte, il existe une preuve $H_{\downarrow}$ *en forme*

normale de $\bot$. On montre la contradiction en prouvant les deux lemmes suivants (à faire pour chaque système considéré) :

1. Premier lemme : étant donné son type, $H_{\downarrow}$ ne peut avoir que la forme $[x : Type]t$ et (par conséquent) le jugement $x : Type \vdash t : x$ est dérivable.
2. Deuxième lemme : on montre qu'un jugement de la forme $x : Type \vdash t : x$ n'est pas dérivable.

Ces deux résultats étant contradictoires, on a montré qu'il n'existe pas de preuve de $\bot$ dans le système considéré. Nous verrons dans la section 4.6 que cette méthode de preuve est encore valide pour le calcul des constructions inductives.

3.7.6 Les propriétés du calcul des constructions

Théorème 3.7.13 $\mathcal{CC}$ vérifie les propriétés 3.7.3 (cohérence), 3.7.7 (préservation du typage), 3.7.5 (décidabilité du typage), 3.7.8 (compatibilité réduction/conversion), 3.7.9 (confluence), 3.7.10 (Church-Rosser) 3.7.12 (normalisation forte des termes bien formés).

Les preuves de ces propriétés peuvent être trouvées dans [Coq85]. La preuve de normalisation forte se fait par la méthode des *candidats de réductibilité*.

Chapitre 4

Le calcul des constructions inductives ($\mathcal{CCI}$)

Dans cette partie nous décrivons le calcul des constructions inductives. Notre présentation est celle de Benjamin Werner [Wer94], elle-même inspirée de celle de Christine Paulin ([PM93]).

Le calcul des constructions inductives est une extension du calcul des constructions conçue pour permettre la définition de *types inductifs* et de fonctions par récurrence d'une manière comparable, mais plus restreinte puisqu'on n'autorise pas les réductions infinies, à ce qu'on trouve en programmation fonctionnelle. Dans $\mathcal{CCI}$ la notion de *type inductif* est primitive. Nous allons d'abord décrire cette notion puis donner la définition formelle du système. En fait, la puissance du système original, $\mathcal{CC}$, rend le système de types beaucoup plus puissant que celui des langages de programmation, notamment grâce aux types dépendants. Nous verrons en particulier que la possibilité de définir des types inductifs dépendants permet une définition très agréable des *propriétés inductives*.

4.1 Les types inductifs

La notion de type de données récursif est un des paradigmes centraux de la programmation moderne. Citons comme exemples de structures récursives les listes, les arbres, les graphes, etc. La possibilité de définir de telles structures de manière primitive est un des traits caractéristiques des langages de programmation fonctionnels. On définit un nouveau type en énumérant les *constructeurs* de ce type, chacun muni de son propre type. Par exemple, dans le langage CAML on peut définir les booléens de la manière suivante (notons que la syntaxe exacte serait true au lieu de true : bool.) :

```
type bool =
  |  true : bool
  |  false : bool
```

où true et false sont les deux constructeurs du nouveau type bool. On peut aussi définir le type (vraiment récursif cette fois) des entiers :

```
type nat =
  |  0 : nat
  |  S : nat → nat
```

Dernier exemple, le type des listes d'entiers :

$$\begin{array}{l}\mathsf{type\ nat\_list} = \\ \quad | \quad \mathsf{nat\_nil} : \mathsf{list} \\ \quad | \quad \mathsf{nat\_cons} : \mathsf{nat} \to \mathsf{nat\_list} \to \mathsf{nat\_list}\end{array}$$

Dans le calcul des constructions inductives, on autorise de telles définitions de types. Nous énumérons les principaux intérêts de cet ajout, on trouvera dans [PM93] une discussion plus approfondie sur cette question :

- La souplesse et la facilité de définition de nombreux concepts. Par exemple, on constate que le type nat ci-dessus ne nécessite plus de codage comme dans le λ-calcul. De plus ces définitions inductives, évitant le codage, s'accompagnent d'un meilleur comportement calculatoire.
- La génération automatique de *principes d'induction* à partir de la définition d'un type inductif, qui permettront de faire des démonstrations par récurrence. L'isomorphisme de Curry-Howard s'étend aux types inductifs et permet de représenter une preuve par induction par une fonction définie par récurrence ; nous explicitons ceci dans la section suivante, ainsi qu'au chapitre 5.
- La définition des *propriétés inductives* grâce à la dépendance de type à la $\mathcal{CC}$. Nous verrons plus loin des exemples montrant l'intérêt de ces définitions en matière de simplicité. Combinée au point précédent, cette possibilité rend même possible le *raisonnement par récurrence sur les preuves de propriétés inductives*, qui se révèle en pratique extrêmement utile (voir toujours le chapitre 5).
- Un gain en expressivité : en effet l'ajout des types inductifs n'est en fait pas uniquement une facilité d'écriture, mais aussi un moyen d'avoir une forme *d'élimination forte*, permettant de prouver que deux termes distincts clos et en forme normale d'un type inductif sont différents. Nous décrirons cette caractéristique plus loin dans cette section.

Deux principes régissent l'utilisation des types inductifs. Nous les expliquons tous les deux dans la suite en donnant également leur signification dans $\mathcal{CCI}$.

4.1.1 Le principe d'élimination

Les éléments d'un type inductif sont, par définition, les objets construits par application bien typée des constructeurs et uniquement ceux-là. Autrement dit un type inductif est le plus petit ensemble clos par application de ses constructeurs. On exprime cela mathématiquement par un *schéma de récurrence* pour chaque type inductif. Par exemple celui des booléens est le suivant :

$$\mathsf{bool\_rec} : \forall P : \mathsf{bool} \to \mathsf{Set}.(P\ \mathsf{true}) \Rightarrow ((P\ \mathsf{false}) \Rightarrow \forall x : \mathsf{bool}.(P\ x)).$$

On exprime ce schéma dans le formalisme du calcul des constructions à l'aide de l'isomorphisme de Curry-Howard en disant que bool_rec est un terme[1] typé de la manière suivante :

$$\mathsf{bool\_rec} : (P : \mathsf{bool} \to \mathsf{Set})(P\ \mathsf{true}) \to (P\ \mathsf{false}) \to (x : \mathsf{bool})(P\ x).$$

On interprète ces principes de deux manières en fonction de la facette de l'isomorphisme de Curry-Howard considérée, c'est-à-dire suivant qu'on se place du côté termes/types ou du côté preuve/propriété :

1. En fait, bool_rec est construit à partir des nouvelles constructions de $\mathcal{CCI}$, qui n'existaient pas dans $\mathcal{CC}$. Voir la définition du système.

1. Du côté termes/types, on interprète les deux principes de la manière suivante : toute fonction sur le type bool (de type bool $\rightarrow P$ ou plus généralement $\forall x : \mathsf{bool}.(P\ x)$) définie pour true et false est définie sur tout le type bool, à la manière de l'expression *if then else* des langages de programmation. Autrement dit true et false sont les deux seuls éléments *canoniques* du type bool. Cette notion est à rapprocher de celle d'éléments canoniques d'un ensemble de la théorie constructive des types de Martin-Löf (voir section 6.2.1).
2. Du côté preuve/propriété, on voit que cela signifie que toute propriété vérifiée par true et false est vérifiée par tout élément du type bool. On reconnaît ici aussi la canonicité de true et false. On reconnaît même en fait le principe d'induction du type bool (celui-ci est dégénéré puisque bool n'est pas un type récursif). Cet aspect sera plus clair ci-dessous avec le schéma des entiers.

Le schéma de récurrence, ou *principe d'élimination* des entiers, illustre le même principe :

$$\mathsf{Nat\_rec} : \forall P : \mathsf{nat} \rightarrow \mathsf{Set}.(P\ 0) \Rightarrow (\forall n : \mathsf{nat}.(P\ n) \Rightarrow (P\ (S\ n))) \Rightarrow \forall x : \mathsf{nat}.(P\ x).$$

On reconnaît ici l'axiome de Peano permettant la récurrence sur les naturels. Sa traduction à la Curry-Howard donne les deux versions, dépendante et non dépendante :

$$\mathsf{Nat\_rec} : (P : \mathsf{nat} \rightarrow \mathsf{Set})(P\ 0) \rightarrow ((n : \mathsf{nat})(P\ n) \rightarrow (P\ (S\ n))) \rightarrow (x : \mathsf{nat})(P\ x).$$
$$\mathsf{Nat\_rec'} : (P : \mathsf{Set})P \rightarrow (P \rightarrow P) \rightarrow (\mathsf{nat} \rightarrow P).$$

On interprète encore une fois de deux manières :

1. Une fonction sur les entiers est complètement définie par la donnée d'une valeur pour 0 et d'une fonction permettant de calculer la valeur pour $(S\ n)$ à partir de n et de la valeur de la fonction pour n. C'est le schéma de définition de fonction *par récurrence*, très utilisé en programmation fonctionnelle.
2. Par Curry-Howard, on reconnaît donc l'axiome de Peano de preuve par récurrence.

Rappelons ici la règle de la sémantique de Heyting traitant du quantificateur universel :

> Une preuve canonique de $\forall x \in A.P(x)$ est une fonction qui prend en argument un élément x de A et rend une preuve de $P(x)$.

On voit que si cette fonction est définie par récurrence, alors la preuve est par récurrence [2].

4.1.2 Le filtrage

Une fois qu'une définition de fonction est donnée à l'aide d'un schéma d'élimination, il reste à définir sa sémantique, autrement dit savoir quelle est l'image par cette fonction d'un terme donné. Cette sémantique est déterminée par la notion de calcul (ou réduction) associée au principe d'élimination, qui correspond à celle des définitions à la fois par *filtrage* et par récurrence, que nous expliquons ci-dessous, et qui correspond encore une fois au paradigme de la programmation fonctionnelle. Ce calcul permet d'évaluer une fonction définie par cas (et éventuellement par récurrence) appliquée à un argument. L'opération définie pour cela est le filtrage. Par exemple la fonction ML suivante effectue la somme des éléments d'une liste d'entiers (on suppose l'opérateur + défini) :

2. Le mot récurrence est en principe réservé à l'utilisation de l'axiome de récurrence de Peano sur les entiers. On utilisera de préférence le mot *induction* lorsqu'il s'agira d'un autre type.

```
let list_add l =
  match l with
  |  nat_nil              ⇒ 0
  |  (nat_cons elt suite) ⇒ elt + (list_add suite)
```

Le principe d'élimination des listes d'entiers est le suivant :

$$\begin{aligned}\mathsf{list\_nat\_rec}: \;& (P : \mathsf{list\_nat} \to \mathsf{Set})\\ & (P\ \mathsf{nat\_nil})\\ & \to ((n : \mathsf{nat})(l : \mathsf{list\_nat})(P\ l) \to (P\ (\mathsf{nat\_cons}\ n\ l)))\\ & \to ((l : \mathsf{list\_nat})(P\ l)).\end{aligned}$$

Et sa version non dépendante :

$$\mathsf{list\_nat\_rec'} : (P : \mathsf{Set})P \to (\mathsf{nat} \to \mathsf{list\_nat} \to P \to P) \to (\mathsf{list\_nat} \to P).$$

On voit que pour définir une fonction des listes vers un type (non dépendant) P, il suffit de donner :

— un terme de type P (correspondant au cas de base nil) ;

— et une fonction de type ($\mathsf{nat} \to \mathsf{list\_nat} \to P \to P$) correspondant à l'étape de calcul par récurrence.

Or la définition de la fonction *list_ add* ci-dessus, dans laquelle le type P est nat, fournit exactement cela :

— un entier : 0

— une fonction qui prend en argument un nat *elt*, une list_nat *suite* et, sous l'hypothèse de connaître le résultat de l'appel récursif (c'est le troisième argument, de type $P = \mathsf{nat}$, exprimé dans `list_add` par l'expression `(list_add suite)`), rend un résultat du type attendu nat : elt + (list_add suite). Cette fonction est donc bien de type ($\mathsf{nat} \to \mathsf{list\_nat} \to P \to P$).

La définition de *list_ add* est donc complète. Reste donc à définir sa sémantique. On va pour cela montrer deux exemples d'application de cette fonction, et suivre la réduction pas à pas.

1. On considère pour commencer l'application de la fonction list_add à l'argument nil. La première étape consiste à remplacer l'argument formel par l'argument réel. On reconnaît une β-réduction classique :

```
(list_add nil)  ⇝β  match nil with
                     | nil ⇒ 0
                     | cons(elt,suite) ⇒ elt + (list_add suite)
```

   Puis on utilise le constructeur de tête de l'argument réel nil (situé maintenant entre le match et le with) pour *choisir* la branche de la définition de la fonction qui va être utilisée, ici la première. Le résultat est donc 0 :

```
match nil with
  | nil ⇒ 0                                  ⇝  0
  | cons(elt,suite) ⇒ elt + (list_add suite)
```

C'est l'opération de *filtrage* qui effectue ce choix. Le travail du filtrage est en fait légèrement plus compliqué que ce simple choix, comme nous le montrons dans le second exemple.

2. On considère maintenant l'application de la fonction list_add à l'argument cons(2,cons(3,nil)) :

$$\begin{array}{l}\mathsf{(list\_add\ cons(2,cons(3,nil)))} \quad \leadsto_\beta \quad \mathsf{match\ cons(2,cons(3,nil))\ with} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{|\ nil \Rightarrow 0} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{|\ cons(elt,suite) \Rightarrow elt + (list\_add\ suite)}\end{array}$$

Cette fois, c'est la deuxième branche du filtrage qui doit être choisie puisque le constructeur de tête est cons. De plus les deux variables elt et suite du motif cons(elt,suite) sont remplacées dans le résultat respectivement par 2 et cons(3,nil). Dernière opération : le troisième argument attendu par cette branche (qui rappelons-le est de type nat → list_nat → P → P) est remplacé par un appel récursif à la fonction list_add :

$$\begin{array}{l}\mathsf{match\ cons(2,cons(3,nil))\ with} \\ \quad \mathsf{|\ nil \Rightarrow 0} \qquad\qquad\qquad\qquad\qquad \leadsto \quad \mathsf{2 + list\_add\ cons(3,nil)} \\ \quad \mathsf{|\ cons(elt,suite) \Rightarrow elt + (list\_add\ suite)}\end{array}$$

On voit donc que le mécanisme de filtrage instancie les variables du motif choisi, de manière à rendre égaux l'argument réel et le motif. On voit également comment un *appel récursif* est calculé, comme ici où on doit à nouveau calculer list_add cons(3,nil), pour obtenir le résultat final 5.

Par ces deux exemples, on a illustré le *comportement calculatoire* des fonctions définies à l'aide des principes d'élimination des types inductifs. Ce calcul s'ajoute à celui qu'on connaît déjà depuis le λ-calcul : la β-réduction. Cette nouvelle réduction de $\mathcal{CCI}$ par rapport à $\mathcal{CC}$ est appelée ι-réduction dans la suite, et est définie formellement plus loin dans ce chapitre.

4.1.3 Types polymorphes + types inductifs

Il est assez naturel de vouloir paramétrer les définitions inductives par des types, afin que par exemple le type des listes présentées plus haut ne soit pas limité aux entiers. Pour cela on a recours, comme dans les langages de programmation, au polymorphisme, c'est-à-dire à la possibilité de définir des fonctions des types vers les types. Le polymorphisme, déjà abordé dans la présentation de $\mathcal{CC}$, se combine de manière agréable avec les types inductifs. On définit par exemple le type list paramétré par le type des éléments (c'est donc une famille de types) :

$$\begin{array}{l}\mathsf{type\ list\ [A : Set]\ :\ Set=} \\ \quad \mathsf{|\ \ nil : (list\ A)} \\ \quad \mathsf{|\ \ cons : nat \to (list\ A) \to (list\ A)}\end{array}$$

Ainsi, les types (list nat) et (list bool) caractérisent respectivement le type des listes d'entiers et le type des listes de booléens, et des termes comme (nil nat) ou (cons bool true (nil bool)) sont leurs habitants.

4.1.4 Types inductifs + types dépendants

Nous avons vu que le système de types de $\mathcal{CCI}$, hérité de $\mathcal{CC}$, permet d'autres formes de types produits (c'est-à-dire de la forme (x : T)U), en particulier les types dépendants. On peut donc définir des *types inductifs dépendants*. On définit par exemple ci-dessous la famille listbn des types de listes de booléens paramétrée par la taille de ces listes. (listbn 0) est le type de la liste de booléens vide, (listbn 1) est le type des listes de booléens de taille 1, etc :

```
type listbn : nat → Set=
  |  nil0 : (listbn 0)
  |  consn : (n :nat) bool → (listbn n) → (listbn (S n))
```

Ce qui permet par exemple de donner la fonction de concaténation des listes de type : (n,m :nat)(listbn n)→ (listbn m) → (listbn (m+n))

Par l'isomorphisme de Curry-Howard appliqué aux types dépendants, on sait déjà qu'on peut définir des prédicats. Avec les types inductifs dépendants, on a la possibilité de définir des *prédicats inductifs*, comme par exemple le prédicat pair suivant :

```
type pair : nat → Set=
  |  pair0 : (pair 0)
  |  pairSS : (x :nat) (pair x) → (pair (S (S x)))
```

On voit qu'il correspond bien à la propriété de parité :
— pair0 est de type (pair 0),
— (pairSS 0 pair0) est de type (pair (S(S 0))) autrement dit (pair 2),
— (pairSS (S(S 0)) (pairSS 0 pair0)) est de type (pair 4),
— ...
— Il ne peut pas exister de terme de type (pair (S 0)).

On sait même que si on a un terme `H` clos bien typé de type `(pair t)`, alors `t` est nécessairement convertible à un terme de la forme `0`, `(S (S 0))`... Ceci illustre comment les types inductifs dépendants permettent de définir les propriétés ou prédicats inductifs.

On peut également définir des prédicats à plusieurs arguments, comme par exemple la relation le, représentant la relation $\leq$:

```
type le : nat → nat → Set=
  |  lexx : (x :nat)(le x x)
  |  leS : (x,y :nat) (le x y) → (le x (S y)).
```

qui est telle que :
— (lexx 0) et (lexx (S 0)) sont de types (le 0 0) et (le (S 0) (S 0)),
— (leS 0 0 (lexx 0)) est de type (le 0 (S 0)),
— ...
— Il n'y a pas de terme de type (le (S 0) 0).

Nous verrons dans le chapitre 5 que cette possibilité est très pratique dans Coq, car elle permet la définition sans réel codage d'un grand nombre de concepts.

4.1.5 L'élimination forte

L'intérêt des types inductifs est à plusieurs niveaux. On a vu qu'ils permettent une simplicité de définition très appréciable pour de nombreux concepts, qu'ils permettent d'avoir un comportement calculatoire intéressant pour ces concepts ainsi qu'une notion primitive de récurrence. Tous ces avantages ne sont pourtant que des *facilités*, pas de réelles nouvelles *possibilités* puisqu'il est possible de coder ces concepts dans $\mathcal{CC}$. Autrement dit, nous n'avons pour l'instant pas vu de gain d'expressivité par rapport au calcul des constructions pur.

En fait le gain d'expression existe, il est dû à la possibilité de définir des principes d'élimination *sur les types* et non plus sur les termes. C'est ce qu'on appelle l'élimination forte. Par exemple on peut dans $\mathcal{CCI}$ définir les termes nat_recs, nat_recs' et nat_recs" typés comme suit :

$$\begin{array}{ll}
\mathsf{nat_recs} : & (Q : \mathsf{nat} \to \mathsf{Type})(Q\ 0) \to ((n : \mathsf{nat})(Q\ n) \to (Q\ (S\ n))) \to (n : \mathsf{nat})(Q\ n). \\
\mathsf{nat_recs'} : & (Q : \mathsf{Type})(Q\ 0) \to (\mathsf{nat} \to Q \to Q) \to \mathsf{nat} \to Q. \\
\mathsf{nat_recs''} : & \mathsf{Set} \to (\mathsf{nat} \to \mathsf{Set} \to \mathsf{Set}) \to \mathsf{nat} \to \mathsf{Set}.
\end{array}$$

On verra dans le chapitre 5 que nat_recs" permet de démontrer que 0 et (S 0)[3] sont différents en définissant la fonction de type nat → Set suivante[4] :

$$(\mathsf{nat\_recs''}\ \mathit{True}[n : \mathsf{nat}][Q : \mathsf{Set}]\mathit{False})$$

qui se réduit en *True* quand elle est appliquée à 0 et en *False* quand elle est appliquée à (S 0).

On voit que l'élimination forte permet d'utiliser le filtrage sur les termes pour calculer des types. Autrement dit (Curry-Howard oblige), il permet de définir une propriété par filtrage sur des termes d'un type inductif. Dans $\mathcal{CC}$, on peut définir le type nat, mais sans cette possibilité, il n'est pas possible de montrer $0 \neq (S\ 0)$.

4.1.6 Des types inductifs problématiques

Un type inductif est complètement défini par la liste des types de ses constructeurs. Il se pose néanmoins le problème de la forme possible de ces types de constructeurs. On s'est aperçu que deux classes de types de constructeurs posaient problèmes.

Ceux qui mènent à des réductions infinies Certains types inductifs permettent de définir des calculs infinis dans les langage de programmation, comme le montre l'exemple, écrit dans le langage CAML, tiré de [Wer94] :

On définit le type à un constructeur suivant :

```
type absurd = C: (absurd -> absurd) -> absurd;;
```

puis on définit la fonction suivante par filtrage :

```
let f = function (C g) -> g (C g);;
```

qui mène à une réduction infinie si on cherche à calculer `f (C f)`.

Pour interdire de tels constructeurs, on définit une condition syntaxique sur le type des constructeurs, appelée *condition de positivité*, que nous énonçons plus loin dans la définition du calcul.

3. Plus généralement deux termes clos commençant par des constructeurs différents

4. Où *True* est le type inductif à un constructeur sans argument caractérisant la propriété vraie ($\top$) (toujours prouvable) et *False* le type inductif sans constructeur caractérisant la propriété fausse $\bot$

Ceux qui mènent au paradoxe T. Coquand a montré qu'il est possible de reproduire le paradoxe de Burali-Forti dans le calcul des constructions avec sommes fortes [Coq86]. Les types sommes sont une généralisation des types paires $(A * B)$ introduits succinctement à la section 2.6.3. La généralisation consiste à permettre au deuxième élément de la paire d'avoir un type dépendant du premier élément. Une telle *paire dépendante* est souvent notée (t, u) et son type $(\Sigma x : A)B$. La règle de typage est de la forme :

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B[x \leftarrow t]}{\Gamma \vdash (t, u) : (\Sigma x : A)B}$$

On obtient en fait ici l'interprétation par Curry-Howard de la règle de typage du quantificateur existentiel de la déduction naturelle :

$$\frac{\Gamma \vdash B[x \leftarrow t]}{\Gamma \vdash \exists x.B}$$

Comme dans la section 2.6.3, on peut définir les deux projections π_1 et π_2. Si cet ajout est sans restriction, alors il est possible de construire des termes de la forme $\Sigma x : A.B$ tels que le type de A est plus "grand" que celui de $\Sigma x : A.B$, par exemple avec $A : Type$ et $B : Set$. De telles sommes sont dites *fortes*. Le résultat de Coquand nous assure que le calcul des constructions devient inconsistant si on permet la projection π_2 sur de tels termes.

Or les types inductifs sont eux-mêmes une généralisation des types sommes. On verra dans le chapitre 5 comment le quantificateur existentiel (intuitionniste) peut-être représenté par un type inductif dépendant. Par conséquent si on autorise l'élimination forte sur tous les types inductifs, on peut reproduire le paradoxe.

On limite donc l'élimination forte à une classe de types inductifs, dont les constructeurs sont dits *petits*, dont la définition est donnée au moment de la définition du typage de $\mathcal{CCI}$, en section 4.5.

4.1.7 Des fonctions récursives problématiques

Il est bien connu que les fonctions définies par récurrence ne terminent pas nécessairement, par exemple on peut définir la fonction suivante en CAML :

```
let rec fboucle x = fboucle x
```

qui ne termine évidemment pour aucun argument. Certaines fonctions peuvent ne terminer que pour un ensemble restreint de valeurs, comme la suivante :

```
let rec fpartielle x=
  match x with
  |  0 → 0
  |  S y → fpartielle (S y)
```

qui ne termine que pour 0.

Pour préserver la normalisation forte du système, il faut que de telles fonctions soient interdites dans $\mathcal{CCI}$. C'est pourquoi la définition des fonctions récursives est soumise à un critère de terminaison garantissant que toute fonction définie dans $\mathcal{CCI}$ termine pour toute valeur.

Comme il n'est pas possible de déterminer automatiquement dans tous les cas si une fonction termine, le critère de terminaison utilisé dans $\mathcal{CCI}$ est une vérification syntaxique suffisante (mais pas nécessaire) relativement simple sur la définition d'une fonction : la définition d'une fonction f doit être telle que tous les appels récursifs contenus dans la définition de f ne se font que sur les sous-termes stricts de cet argument. Comme l'ordre "sous-terme" est bien fondé, les fonctions respectant ce critère terminent nécessairement.

Il existe des techniques pour définir une classe importante de fonctions en respectant ce critère. Dans la définition formelle du système ci-dessous, cet argument de décroissance sur les appels récursifs est exprimé dans la définition même des fonctions récursives (construction Elim et ι-réduction).

Signalons que dans Coq la condition syntaxique sur les appels récursifs est plus forte : la décroissance par l'ordre sous-terme doit être sur *un* argument particulier de la fonction.

4.2 Syntaxe

4.2.1 Notations

On notera les séquences de termes à l'aide de la notation suivante :

Définition 4.2.1 On signalera qu'une lettre désigne une séquence de termes grâce à la notation vectorielle : $\vec{u}$. u_i désignera le $i^{\text{ème}}$ élément de la séquence $\vec{u}$. On utilisera également une notation comparable à celle des listes :

$\{\}$	désigne la séquence de termes vide;
$t :: \vec{u}$	designe la séquence de termes $\vec{u}$ à laquelle on a ajouté le terme t en tête.

4.2.2 Grammaire

On a deux ensembles dénombrables distincts de variables : Vt et VT. Le premier sera l'ensemble des *variables de termes* et l'autre celui des *variables de prédicats*. Cette distinction sert dans les règles de typage pour définir la notion de *grand ou petit constructeur* définie plus bas. On notera les premières a, b, etc et les deuxièmes α, β, etc ; lorsque la distinction ne sera pas nécessaire, on notera x, y, X, etc. On dira que Set est *la sorte associée* à une variable de terme, et Type est la sorte associée à une variable de prédicats.

On se donne la même hiérarchie de sortes que pour $\mathcal{CC}$: Set : Type : Extern. L'ajout d'une hiérarchie d'univers semble possible. La syntaxe est donnée figure 4.1.

$$
\begin{array}{rrl}
\text{Variables :} & V ::= & \alpha \mid a \ldots \\
\text{Sortes :} & S ::= & \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Extern} \\
\text{Termes :} & T ::= & V \mid S \mid [V:T]T \mid (V:T)T \mid TT \\
 & & \mid \mathsf{Ind}(\alpha:T)\{\vec{T}\} \mid \mathsf{Constr}(n,T)\ n \in N \mid \mathsf{Elim}(T,T,\vec{T},T)\{\vec{T}\}
\end{array}
$$

FIGURE 4.1 – Syntaxe des pré-termes de $\mathcal{CC}$

On voit que les constructions syntaxiques de $\mathcal{CC}$ sont conservées, et qu'il y en a trois nouvelles :

Ind, Constr et Elim sont les constructions spécifiques de $\mathcal{CCI}$ qui permettent de définir et utiliser les types inductifs :

- $\mathsf{Ind}(\alpha : T)\{\vec{T}\}$ est le constructeur de type inductif, il prend en argument la liste $\vec{T}$ des types de chacun des constructeurs du type inductif défini. La variable typée $(\alpha : T)$ sert à nommer le type construit à l'intérieur de sa définition afin de permettre la définition de constructeurs récursifs.
- $\mathsf{Constr}(i, I)$ permet de désigner le $i^{\text{ème}}$ constructeur du type inductif I.
- Enfin, Elim est le *destructeur* de type inductif, c'est-à-dire la construction permettant de faire de l'élimination sur les types inductifs. Elim permettra de définir des fonctions, éventuellement récursives, par *filtrage* à la ML sur les constructeurs (voir la section précédente).
  $\mathsf{Elim}(I, Q, \vec{a}, t)\{\vec{f_i}\}$ se lit :
  filtrage du terme t (de type inductif $(I\ \vec{a})$, $\vec{a}$ est une séquence de *paramètres*),
  rendant un objet de type $(Q\ \vec{a}t)$,
  de branches $\vec{f_i}$, la branche i correspondant au $i^{\text{ème}}$ constructeur de I.

La notion de variable ayant une occurrence libre dans un terme est étendue aux nouvelles constructions ci-dessous. Comme précédemment, on supposera qu'une variable n'a soit que des occurrences libres, soit que des occurrences liées, et dons $\mathcal{FV}_{\mathcal{CCI}}$ caractérise les variables libres d'un terme.

Définition 4.2.2 (Variable libre dans $\mathcal{CCI}$, $\mathcal{FV}_{\mathcal{CCI}}$)

- $\mathcal{FV}_{\mathcal{CCI}}(s) = \emptyset$ si s est une sorte
- $\mathcal{FV}_{\mathcal{CCI}}(x) = \{x\}$
- $\mathcal{FV}_{\mathcal{CCI}}((t_1\ t_2)) = \mathcal{FV}_{\mathcal{CCI}}(t_1) \cup \mathcal{FV}_{\mathcal{CCI}}(t_2)$
- $\mathcal{FV}_{\mathcal{CCI}}([x : t_1]t_2) = \mathcal{FV}_{\mathcal{CCI}}(t_1) \cup (\mathcal{FV}_{\mathcal{CCI}}(t_2) \setminus \{x\})$
- $\mathcal{FV}_{\mathcal{CCI}}((x : t_1)t_2) = \mathcal{FV}_{\mathcal{CCI}}(t_1) \cup (\mathcal{FV}_{\mathcal{CCI}}(t_2) \setminus \{x\})$
- $\mathcal{FV}_{\mathcal{CCI}}(\mathsf{Ind}(\alpha : t)\{\vec{u}\}) = \mathcal{FV}_{\mathcal{CCI}}(t) \cup (\mathcal{FV}_{\mathcal{CCI}}(\vec{u}) \setminus \{\alpha\})$
- $\mathcal{FV}_{\mathcal{CCI}}(\mathsf{Constr}(n, t)) = \mathcal{FV}_{\mathcal{CCI}}(t)$
- $\mathcal{FV}_{\mathcal{CCI}}(\mathsf{Elim}(t_1, t_2, \vec{v}, t)\{\vec{u}\})$
  $= \mathcal{FV}_{\mathcal{CCI}}(t_1) \cup \mathcal{FV}_{\mathcal{CCI}}(t_2) \cup \mathcal{FV}_{\mathcal{CCI}}(\vec{v}) \cup \mathcal{FV}_{\mathcal{CCI}}(t) \cup \mathcal{FV}_{\mathcal{CCI}}(\vec{u})$

On étend $\mathcal{FV}_{\mathcal{CCI}}$ aux séquences de termes :

- $\mathcal{FV}_{\mathcal{CCI}}(\{\}) = \emptyset$
- $\mathcal{FV}_{\mathcal{CCI}}(t :: \vec{u}) = \mathcal{FV}_{\mathcal{CCI}}(t) \cup \mathcal{FV}_{\mathcal{CCI}}(\vec{u})$

On voit d'après cette définition que la construction Ind est un lieur, au même titre que la λ-abstraction et le produit.

4.2.3 Substitutions

On notera les substitutions de manière usuelle : $t[x \leftarrow u]$ représente le terme t dans lequel on a remplacé toutes les occurrences libres de x par u.

Définition 4.2.3 (Substitution dans $\mathcal{CCI}$) On définit l'opération de substitution d'une variable x par un terme u dans un terme t, notée $t[x \leftarrow u]$, par induction sur t :

$$
\begin{array}{lcll}
x[x \leftarrow u] & = & u & \\
x[y \leftarrow u] & = & x & \text{si } x \neq y \\
[x:T]t'[x \leftarrow u] & = & [x:T[x \leftarrow u]]t' & \\
[x:T]t'[y \leftarrow u] & = & [x:T[x \leftarrow u]](t'[y \leftarrow u]) & \text{si } x \neq y \text{ et } x \notin \mathcal{FV}_{\mathcal{CCI}}(u) \\
(x:T)t'[x \leftarrow u] & = & (x:T[x \leftarrow u])t' & \\
(x:T)t'[y \leftarrow u] & = & (x:T[x \leftarrow u])(t'[y \leftarrow u]) & \text{si } x \neq y \\
(t'\ t'')[x \leftarrow u] & = & (t'[x \leftarrow u]\ t''[x \leftarrow u]) & \\
\mathsf{Ind}(\alpha:t)\{\vec{u}\}[\alpha \leftarrow u] & = & \mathsf{Ind}(\alpha:t[\alpha \leftarrow u])(\vec{u}) & \\
\mathsf{Ind}(\alpha:t)\{\vec{u}\}[x \leftarrow u] & = & \mathsf{Ind}(\alpha:t[x \leftarrow u])\{\overrightarrow{u[x \leftarrow u]}\} & \text{si } x \neq \alpha \\
\mathsf{Constr}(n,t)[x \leftarrow u] & = & \mathsf{Constr}(n,t[x \leftarrow u]) & \\
\mathsf{Elim}(t_1,t_2,\vec{v},t)\{\vec{u}\}[x \leftarrow u] & = & \mathsf{Elim}(t_1[x \leftarrow u],t_2[x \leftarrow u],\overrightarrow{v[x \leftarrow u]},t[x \leftarrow u])\{\overrightarrow{u[x \leftarrow u]}\} &
\end{array}
$$

On définit également une notation pour les substitutions séquentielles :

Définition 4.2.4 Soit t un terme, $\vec{x}$ et $\vec{u}$, respectivement des séquences de variables et de termes de même longueur. On définit le terme $t[\vec{x} \leftarrow \vec{u}]$ de la manière suivante :

$$
\begin{array}{lcl}
t[[] \leftarrow []] & = & t \\
t[x :: \vec{x} \leftarrow u :: \vec{u}] & = & t[x \leftarrow u][\vec{x} \leftarrow \vec{u}]
\end{array}
$$

4.3 Réduction

La réduction interne de $\mathcal{CCI}$ est l'union de deux réductions : la β-réduction et la ι-réduction, définies ci-dessous. La β-réduction correspond comme toujours au remplacement de l'argument formel d'une fonction par l'argument réel. La ι-réduction correspond à l'opération de filtrage sur les types inductifs.

4.3.1 β-réduction

La β-réduction est définie comme dans $\mathcal{CC}$:

Définition 4.3.1 ($\longrightarrow_\beta$) La β-réduction dans $\mathcal{CCI}$ est la clôture par contexte de la règle suivante :

$$([x:t]t_1\ t_2) \rightarrow_\beta t_1[x \leftarrow t_2]$$

4.3.2 ι-réduction

La ι-réduction correspond à la réduction des fonctions récursives expliquée à la section 4.1. Sa définition nécessite quelques précautions comme nous l'avons expliqué plus haut. On commence donc par définir la *condition de positivité* des types des constructeurs donnés dans la construction

Ind. Cette condition syntaxique permet de restreindre les types de constructeur autorisés et ainsi d'éviter une classe de types inductifs sur lesquels la réduction ne termine pas (voir section 4.1.6).

Définition 4.3.2 (Occurrence positive) Soit X une variable de prédicat. Un terme t est dit *strictement positif* en X si :
- $t \equiv (\vec{x} : \vec{t})(X\ \vec{t'})$,
- X n'a pas d'occurrence libre dans $\vec{t}$ et $\vec{t'}$,
- et aucun élément de $\vec{x}$ n'est égal à X.

On notera alors $Pos(X, t)$.

On peut maintenant définir les types de constructeurs autorisés :

Définition 4.3.3 (Type de constructeur) Un terme C est un type de constructeur en X (on dira aussi que C(X) est un type de constructeur) si l'une des conditions ci-dessous est vérifiée :
- $C = (X\ \vec{t})$ si X n'a pas d'occurrence libre dans $\vec{t}$
- $C = (x : t)D$ si D est un type de constructeur en X, X n'a pas d'occurrence libre dans t, et $X \neq x$
- $C = P \to D$ si D est un type de constructeur en X et *Pos (X,P)*.

On notera alors $constr(C(X))$.

On a vu que le filtrage ne se réduit pas à un simple choix de branche, il effectue en même temps une substitution. Les deux définitions suivantes permettent de formaliser cette opération.

Définition 4.3.4 Soit $C(X)$ un type de constructeur en X et deux termes f et F. On définit le terme $\Delta[C(X), f, F]$ par récurrence sur la preuve que C(X) est un type de constructeur :

$$\begin{array}{lcl} \Delta[(X\ \vec{t}), f, F] & = & f \\ \Delta[(x : t)D(X), f, F] & = & [x : t]\Delta[D(X), (f\ x), F] \\ \Delta[((\vec{x} : \vec{t})(X\ \vec{t'})) \to D(X), f, F] & = & [p : (\vec{x} : \vec{t})(X\ \vec{t'})]\Delta[D(X), (f\ p\ [\vec{x} : \vec{t}](F\ \vec{t'}\ (p\ \vec{x}))), F] \end{array}$$

Définition 4.3.5 (Arité) Soit s une sorte, les termes définis par

$$Ar_s := s \mid (x : t)Ar_s,$$

où x est une variable et t un terme, sont appelés *arités* de sorte s. $u \in Ar_s$ sera noté $Ar(u, s)$.

On définit la notation *Fun_Elim* utilisée dans la définition de la ι-réduction ci-dessous.

Définition 4.3.6 Soit A une arité, I le terme $\mathsf{Ind}(X : A)\{C(\vec{X})\}$, Q un terme et $\vec{f}$ et $\vec{a}$ deux séquences de termes. On définit le terme *Fun_Elim* comme suit :

$$Fun\_Elim(I, Q, \vec{a}, \vec{f}) = [\vec{x} : \vec{a}][c : (I\ \vec{x})]\mathsf{Elim}(I, Q, \vec{x}, c)\{\vec{f}\}$$

On peut maintenant formuler la ι-réduction de manière concise :

Définition 4.3.7 (ι-réduction) On définit la ι-réduction comme la clôture par contexte de la règle ci-dessous :

$$\begin{array}{rcl} \mathsf{Elim}(I, Q, \vec{a}, \mathsf{Constr}(k, I')\vec{m})\{\vec{f}\} & \rightarrow_\iota & (\Delta[C_k(I), f_k, \mathit{Fun\_Elim}(I, Q, \vec{a}, \vec{f})]\vec{m}) \\ & \text{où} & I = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \end{array}$$

Intuitivement :

- La ι-réduction prend la k$^{\text{ème}}$ branche de la définition de la fonction, où k est le numéro du constructeur de tête de l'argument. Ceci correspond à l'étape de filtrage expliquée à la section 4.1.
- Le terme $(\Delta[C_k(I), f_k, \mathit{Fun\_Elim}(I, Q, \vec{a}, \vec{f})])$ permet d'appliquer la branche choisie directement aux arguments $(\vec{m})$ du constructeur de l'argument. Ceci correspond à l'instanciation des variables du motif.

Rappelons que chaque f_k est une fonction prenant en argument un élément du type inductif et le résultat de l'appel récursif sur cet élément. Dans le chapitre suivant consacré au système Coq, on donne un exemple de terme ι-réductible.

4.3.3 $\mathcal{CCI}$-réduction

Définition 4.3.8 (Réduction $\mathcal{CCI}$) La réduction $\longrightarrow_{\mathcal{CCI}}$ du calcul des constructions inductives est l'union des β et ι-réductions.

$$\longrightarrow_{\mathcal{CCI}} \;=\; \longrightarrow_\iota \cup \longrightarrow_\beta$$

4.4 Conversion

La conversion $\equiv_{\mathcal{CCI}}$ de $\mathcal{CCI}$ est définie de manière classique comme la clôture de la réduction :

Définition 4.4.1 (Conversion $\mathcal{CCI}$) La conversion $\equiv_{\mathcal{CCI}}$ du calcul des constructions inductives est la clôture réflexive, symétrique et transitive de la réduction $\longrightarrow_{\mathcal{CCI}}$.

4.5 Typage

On définit les environnements de typage comme des séquences de paires de la forme $(x : T)$, comme dans $\mathcal{CC}$.

Les règles de typage de $\mathcal{CCI}$ (figure 4.2) sont présentées en deux groupes. Le premier reprend exactement les règles de $\mathcal{CC}$, sauf la conversion qui est modifiée, et le deuxième permet de typer les trois nouvelles constructions Ind, Constr et Elim. Ce second ensemble de règles nécessite deux précautions :

- L'élimination forte n'est autorisée que sur les types inductifs dont les constructeurs sont *petits* (voir section 4.1.6), dont on donne ici la définition :

Définition 4.5.1 (Petits et grands constructeurs) Soit $C(X)$ un constructeur de type. On a $C(X) \equiv (\vec{x} : \vec{t})(X\ x')$. Si $\vec{x}$ est une séquence de *variables de termes* (autrement dit $\vec{x} = \vec{a}$), alors on dira que $C(X)$ est un *petit constructeur*. On écrira $Small(C(X))$. Dans le cas contraire, il sera dit *grand*, noté $Big(C(X))$.

— Les types inductifs ne sont également pas tous autorisés (section 4.1.6). C'est la notion de type de constructeur qui permet cette restriction. Par ailleurs la notation $\Delta\{C_i(I), Q, \mathsf{Constr}(i, I)\}$ définie ci-dessous permet de définir le type attendu des branches $\vec{f_i}$ du filtrage par Elim :

Définition 4.5.2 Soit $C(X)$ un type de constructeur et deux termes Q et c. On définit le terme $\Delta\{C(X), Q, c\}$ par récurrence sur la preuve que $C(X)$ est un type de constructeur :

$$\begin{array}{lcl} \Delta\{(X\ \vec{t}), Q, c\} & = & (Q\ \vec{t}\ c) \\ \Delta\{(x : t)D(X), Q, c\} & = & (x : t)\Delta\{D(X), Q, (c\ x)\} \\ \Delta\{((\vec{x} : \vec{t})(X\ \vec{t'})) \to D(X), Q, c\} & & \\ & = & (p : (\vec{x} : \vec{t})(X\ \vec{t'}))((\vec{x} : \vec{t})((Q\ \vec{t'})(p\ \vec{x}))) \to \Delta\{D(X), Q, (c\ p)\} \end{array}$$

4.6 Propriétés du calcul des constructions inductives

Théorème 4.6.1 $\mathcal{CCI}$ vérifie toutes les propriétés citées à la section 3.7.

Les preuves des propriétés de $\mathcal{CCI}$ peuvent être trouvées dans la thèse de B. Werner [Wer94], dans laquelle la η-réduction est également considérée.

4.6.1 Cohérence

Nous donnons maintenant une idée de la preuve que la normalisation forte implique la cohérence du calcul des constructions inductives. Il s'agit de la même méthode que celle décrite dans la section 3.7.5, simplifiée par la présence du type inductif `False` dont on sait qu'il n'a pas de preuve (dans un environnement vide)[5].

La démonstration se fait par l'absurde. Supposons qu'il existe une preuve H de $\bot = \texttt{False}$ donc telle que $\emptyset \vdash H : \texttt{False}$. Par la propriété de normalisation forte (et de la subject reduction), il existe une preuve $H_\downarrow$ *en forme normale* de `False`. Or il est possible de prouver (nous ne le faisons pas ici) qu'aucune règle de typage ne peut avoir comme conclusion un jugement de la forme $\emptyset \vdash H : \texttt{False}$ avec H en forme normale. On a donc immédiatement une contradiction.

5. Rappelons que `False` est par définition le type inductif à zéro constructeur suivant :$\mathsf{Ind}(\texttt{False} : \mathsf{Type})\{\}$.

$$(\text{Ax}_1)[] \vdash \mathsf{Set} : \mathsf{Type} \qquad (\text{Ax}_2)[] \vdash \mathsf{Type} : \mathsf{Extern}$$

$$(\text{Prod-s})\frac{\Gamma :: (x : t_1) \vdash t_2 : s}{\Gamma \vdash (x : t_1)t_2 : s} \qquad (\text{Lam-s})\frac{\Gamma \vdash (x : t_1)t_2 : s \qquad \Gamma :: (x : t_1) \vdash t : t_2}{\Gamma \vdash [x : t_1]t : (x : t_1)t_2}$$

$$(\text{W-Set})\frac{\Gamma \vdash t : \mathsf{Set} \quad \Gamma \vdash A : B \quad a \notin \Gamma}{\Gamma :: (a : t) \vdash A : B} \qquad (\text{W-Type})\frac{\Gamma \vdash t : \mathsf{Type} \quad \Gamma \vdash A : B \quad \alpha \notin \Gamma}{\Gamma :: (\alpha : t) \vdash A : B}$$

$$(\text{Var})\frac{\Gamma \vdash t_1 : t_2 \qquad (x : t_1) \in \Gamma}{\Gamma \vdash x : t_1} \qquad (\text{App})\frac{\Gamma \vdash t_2 : (x : T_1)T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash (t_2\ t_1) : T_2[x \leftarrow t_1]}$$

$$(\text{Conv})\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_1 : s \quad \Gamma \vdash T_2 : s \quad T_1 \equiv_{\mathcal{CCI}} T_2}{\Gamma \vdash t : T_2}$$

$$(\text{Ind})\frac{Ar(A, \mathsf{Set}) \quad \Gamma \vdash A : \mathsf{Type} \quad \forall i.(\Gamma :: (X : A) \vdash C_i(X) : \mathsf{Set}) \quad \forall i.constr(C_i(X))}{\Gamma \vdash \mathsf{Ind}(X : A)\{C_i(\vec{X})\} : A}$$

$$(\text{Intro})\frac{\Gamma \vdash \mathsf{Ind}(X : A)\{C_i(\vec{X})\} : T}{\forall n.(\Gamma \vdash \mathsf{Constr}(n, \mathsf{Ind}(X : A)\{C_i(\vec{X})\}) : C_n(\mathsf{Ind}(X : A)\{C_i(\vec{X})\}))}$$

$$(\text{W-Elim})\frac{\begin{array}{c}A \equiv (\vec{x} : \vec{A})Set \quad I = \mathsf{Ind}(X : A)\{C_i(\vec{X})\} \quad \Gamma \vdash \vec{u} : \vec{A} \quad \Gamma \vdash t : (I\ \vec{u}) \\ \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to \mathsf{Set} \quad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i, I)\})\end{array}}{\Gamma \vdash \mathsf{Elim}(I, Q, \vec{u}, t)\{\vec{f_i}\} : (Q\ \vec{u}\ t)}$$

$$(\text{S-Elim})\frac{\begin{array}{c}A \equiv (\vec{x} : \vec{A})Set \quad I = \mathsf{Ind}(X : A)\{C_i(\vec{X})\} \quad \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to \mathsf{Type} \\ \Gamma \vdash \vec{u} : \vec{A} \quad \Gamma \vdash t : (I\ \vec{u}) \quad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i, I)\}) \quad \forall i.Small(C_i(X))\end{array}}{\Gamma \vdash \mathsf{Elim}(I, Q, \vec{u}, t)\{\vec{f_i}\} : (Q\ \vec{u}\ t)}$$

FIGURE 4.2 – Typage de $\mathcal{CCI}$

Chapitre 5

Le système Coq

Ce chapitre présente l'outil d'aide à la démonstration Coq à l'aide d'un certain nombre d'exemples destinés à illustrer les principes expliqués jusqu'ici, notamment concernant les types inductifs et/ou dépendants. Le chapitre se termine (section 5.5) sur une première introduction au problème des structures non libres, que nous traitons dans la deuxième partie de ce travail. Plus précisément nous montrons quels problèmes surviennent lorsqu'on introduit des équations entre constructeurs.

En première approximation Coq est un programme interactif permettant simplement à l'utilisateur de définir et de typer des termes du calcul des constructions inductives [1]. D'après ce que nous avons vu, cela lui permet (par l'isomorphisme de Curry-Howard) de :

1. Définir des termes objets mathématiques, c'est-à-dire
   — les objets du discours, comme les entiers, les listes, etc, représentés par des termes ;
   — des propriétés sur ces objets, représentées aussi par des termes (puisque les propriétés sont des types et que types et termes ne sont pas différenciés dans $\mathcal{CCI}$).
2. Construire des démonstrations de ces propriétés, qui sont également des termes. Pour construire ces termes, on dispose d'un langage de *tactiques* permettant de fabriquer interactivement un terme d'un type donné ;
3. Vérifier les preuves (c'est-à-dire les termes) ainsi construites, en demandant au programme de les *typer*.

On voit tout de suite que le mécanisme de construction des preuves n'a pas besoin d'être certifié, puisque le *noyau* du programme Coq, c'est-à-dire la fonction de typage, permet de vérifier les preuves. Le noyau, un programme relativement "petit", fait en revanche l'objet de certifications et de vérifications. Ajoutons que la représentation concrète des preuves a d'autres avantages : elle rend possible l'écriture d'un vérificateur de type indépendant de Coq, et elle permet également d'utiliser toutes sortes d'outils pour engendrer des preuves que Coq pourra vérifier a posteriori .

5.1 Opérations de base

Nous allons maintenant donner une suite d'exemples de définitions tout en introduisant la syntaxe nécessaire. Tous les exemples donnés dans la syntaxe Coq seront dans une police `verbatim`, et seront en général suivis de la réponse du système. À l'exception du type `nat` ci-dessous, les différents

1. En fait, il est plus puissant puisqu'il permet entre autres la compilation de fichiers, l'extraction de code à partir des preuves, etc.

exemples de cette section peuvent être tapés directement dans Coq. On désignera dans la suite Coq indifféremment par "le programme" ou "le système".

Montrons tout d'abord comment demander à Coq de typer un terme. Cela se fait par la commande `Check`, et la réponse s'affiche à la ligne suivante :

```
Check Set.
Set
     : Type

Check [A:Set]A.
[A:Set]A
      : Set->Set


Check [A:Set][x:A]x.
[A:Set; x:A]x
           : (A:Set)A->A
```

Comme nous l'avons dit, Coq permet de construire interactivement un terme d'un type donné ; pour cela on utilise la commande `Goal` :

```
Goal (A:Set)A->A.
1 subgoal

  ============================
   (A:Set)A->A
```

Le système passe alors en mode *preuve* ; comme on connaît déjà un terme ayant le type demandé (au-dessus), l'utilisateur peut le donner directement :

```
Exact [A:Set][x:A]x.
Subtree proved!
```

Le système signale que le processus de construction du terme est terminé. C'est lorsque l'utilisateur veut sauvegarder l'objet construit en lui donnant un nom, grâce à la commande `Save`, que le noyau du système vérifie son type.

```
Save id.
Exact [A:Set; x:A]x.

id is defined
```

Le système répond ici positivement et associe dans l'environnement le nom `id` au terme `[A:Set; x:A]x`, ce qu'on peut vérifier immédiatement avec la commande `Print` :

```
Print id.
id = [A:Set; x:A]x
     : (A:Set)A->A
```

5.2 Set et Prop

Signalons que le terme `id` est l'identité polymorphe, dont nous avons déjà parlé, mais aussi la preuve de la propriété $\forall A.A \Rightarrow A$. Il s'avère parfois pratique de considérer un terme uniquement comme une preuve, en ignorant son interprétation calculatoire ; c'est pourquoi il existe dans Coq une sorte supplémentaire `Prop`. Comme `Set`, `Prop` est de type `Type` (pour être exact, ces deux sortes sont de type `Type(i)`, pour tout `i`$\in \mathbb{N}$).

On peut donc définir le terme suivant :

```
Definition idProp:=(A:Prop)A->A.
idProp is defined

Check idProp.
idProp
     : Prop
```

Même si `idProp` et `id` correspondent en fait au même concept, on utilisera `id` pour désigner l'algorithme et `idProp` pour désigner la propriété.

5.3 Les types inductifs

Pour définir un type inductif, on dispose d'une syntaxe comparable à celle des langages de programmation. Par exemple le type des naturels `nat` peut être défini comme le type inductif suivant :

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

Les quatre messages engendrés ci-dessus par le système signifient que le type inductif `nat` est défini, ainsi que trois principes d'élimination : un sur `Prop` (`nat_ind`), un sur `Set` (`nat_rec`) et un sur `Type` (`nat_rect`) qui correspond à l'élimination forte.

```
Check nat_ind.
  : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

Signalons que les restrictions, présentées au chapitre 4, quant à la forme et au type des constructeurs dans les types inductifs, sont respectées. On illustre cela par les deux exemples suivants, tirés de [Wer94] :

— Si un des constructeurs n'est pas *petit*, le schéma d'élimination forte n'est pas engendré, conformément à la définition donnée dans le chapitre 4 :

```
Inductive U: Set :=
  encap: Set -> U.
U is defined
U_ind is defined
U_rec is defined
```

— Toujours conformément au chapitre 4, si la condition de positivité n'est pas respectée dans la définition d'un type inductif, celle-ci est refusée par le système :

```
Inductive absurd : Set :=
 C: (absurd -> absurd) -> absurd.
Error: Non strictly positive occurrence of absurd in
 (absurd->absurd)->absurd
```

On montre maintenant un exemple de définition de fonction sur les entiers :

```
Fixpoint plus: nat ->  nat ->  nat :=
[n,m:nat]
 Cases n of
     O      => m
   | (S n') => (S (plus n' m))
 end.
plus is defined
```

Remarquons que cette définition correspond à la construction suivante de $\mathcal{CCI}$:

[n,m :nat]Elim(nat,nat,{},n){ m , [x :nat][y :nat](S y)}

Dans l'expression [x :nat][y :nat](S y), x correspond à l'argument `n` de la fonction, et y à l'appel récursif `(plus n' m)`.

Remarque 5.3.1 *Dans toute la suite, on écrira souvent une expression du type*

```
Cases n of pat1 => t1 | pat2 => t2 ...
```

à la place de l'expression correspondante de la forme Elim(...). La première notation, qui correspond à l'usage dans les langages de programmation fonctionnels, est en effet plus lisible.

On peut appliquer cette fonction à des arguments :

```
Check (plus O O).
(plus O O)
     : nat
```

Pour calculer le résultat, il faut faire appel à la réduction du système, ce qui se fait par la suite de commandes suivante :

```
Eval Compute in (plus O O).
     = O
     : nat


Eval Compute in (plus (S (S O)) (S O)).
     = (S (S (S O)))
     : nat
```

Un bon exercice pour comprendre la définition formelle de la ι-réduction (donnée en section 4.3.2) consiste à effectuer manuellement les différentes réductions du terme `(plus (S (S O)) (S O))` qui correspond au $\mathcal{CCI}$-terme suivant :

((([n,m :nat]Elim(nat,nat,{},n){ m , [x :nat][y :nat](S y)}) (S (S O))) (S O))

5.4 Les propriétés inductives

5.4.1 `True` et `False`

Le type inductif le plus simple est celui qui n'a pas de constructeur. Comme les seuls termes d'un type inductif sont ceux construits à partir des constructeurs, ce type inductif *n'a pas d'habitant*. Il s'agit donc d'un propriété indémontrable, qui est appelée `False`.

```
Inductive False  : Prop :=  .
```

En conséquence, toute preuve de `False` constitue un paradoxe. C'est d'ailleurs ce qu'exprime le principe d'élimination de `False` :

```
Check False_ind.
False_ind
     : (P:Prop)False->P
```

On utilisera `False` pour exprimer la négation logique. En effet on sait que $\neg P \iff (P \Rightarrow \bot)$, donc la négation de `P`, notée `~P`, sera définie comme `P -> False`.

Le deuxième type inductif qu'on peut construire est celui qui a un unique constructeur, sans argument. Il n'y a qu'un seul habitant pour ce type, quel que soit l'environnement de typage. Il s'agit donc d'une propriété toujours démontrable. On l'appelle `True` :

```
Inductive True  : Prop :=  I : True
```

Comme `True` est toujours démontrable, le principe d'élimination de `True` ne sert à rien :

```
Check True_ind.
True_ind
     : (P:Prop)P->True->P
```

5.4.2 L'égalité

L'égalité propositionnelle, c'est-à-dire celle qu'on peut prouver dans le système, est définie comme la propriété inductive –à un constructeur– suivante :

```
Inductive eq [A : Set]  : A -> A -> Prop :=
   refl_equal : (x : A)(eq A x x).
```

L'expression `[A:Set]` après le nom du type inductif signifie que `A` est un *paramètre* de la définition, c'est-à-dire que le type et les constructeurs seront tous abstraits par rapport à `A`[2] :

2. Dans la librairie standard de Coq, `x` est aussi abstrait.

```
Check eq.
   : (A:Set) A -> A -> Prop.
Check refl_equal.
   : (A:Set) (x:A)(eq A x x).
```

On notera `x=x` dans la suite le type `(eq A x x)`. On voit que l'égalité ne peut être prouvée qu'entre un terme et lui-même. En fait, grâce à la règle de conversion, elle peut être prouvée entre deux termes *convertibles*, c'est-à-dire équivalents pour $\equiv_{CCI}$.

Le principe d'élimination exprime la possibilité, évoquée à la section 3.6, de remplacer dans une propriété un terme par un terme qui lui est convertible :

```
Check eq_ind.
eq_ind
     : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)x=y->(P y)
```

Par exemple, la preuve que `O` est égal à `O` est aussi la preuve que `(plus O O)` est égal à `O`, comme le montrent les commandes suivantes :

```
Check (refl_equal nat O).
(refl_equal nat O)
     : O=O (* ce terme est de type O=O*)

Goal (plus O O)=O.
Exact (refl_equal nat O).
--> Subtree proved! (* Mais il est aussi de type (plus O O)=O *)
Save plusOOeqO. (* preuve finie, on donne un nom au lemme *)

(* On peut le vérifier aisément: *)
Print plusOOeqO.
 --> (refl_equal nat O)
     :(plus O O)=O.
```

Voir les sections 3.4.2 et 3.5.2 pour une explication plus détaillée de la règle de conversion.

5.4.3 Quantification existentielle

Un autre exemple intéressant de type inductif est le suivant :

```
Inductive ex [A : Set; P : A->Prop]  : Prop :=
      ex_intro : (x:A)(P x)->(ex A P).
```

Les termes de ce type sont de la forme `(ex_intro A P t H)`, où :
- `A` est un type,
- `P` un prédicat sur `A`,
- `t` est un terme de type `A`,
- `H` est une preuve de `(P t)`, c'est-à-dire que `t` vérifie `P`.

On reconnaît ici les composantes d'une preuve canonique de $\exists x : A.(P\ x)$ au sens de Heyting (section 2.6.2) : `t` est bien le *témoin*. On représente par ce type inductif (dépendant) le quantificateur existentiel intuitionniste. En fait les types inductifs sont une généralisation des types sommes, dont la règle de typage correspond par l'isomorphisme de Curry-Howard à celle du quantificateur $\exists$ dans la déduction naturelle.

Dans la suite, on notera parfois `(EX x:A|(P x))` au lieu de `(ex A P)`, conformément à la syntaxe de Coq.

5.4.4 Autres exemples

On reprend maintenant la définition de la propriété `le` introduite à la section 4.1.4, en changeant la sorte d'arrivée en `Prop` puisqu'il s'agit d'une propriété.

```
Inductive le: nat -> nat -> Prop :=
   | lexx: (x:nat)(le x x)
   | leS : (x,y:nat)(le x y) -> (le x (S y)).
le_ind is defined
```

On démontre facilement que 0 est inférieur à 1 par la suite de commandes suivante :

```
Goal (le O (S O)).
Apply leS.
(* reste à prouver (le O O)*)
Apply lexx.
Save LeO1.
```

On obtient la preuve suivante :

```
Print LeO1.
LeO1 = (leS O O (lexx O))
     : (le O (S O))
```

Signalons ici aussi que cette preuve est aussi la preuve que `(plus O O)` est inférieur à `(plus O (S O))`, comme on le montre ici :

```
Goal (le (plus O O) (plus O (S O))).
Exact LeO1.
--> Subtree proved!
```

Dernier exemple de propriété inductive : la propriété d'*accessibilité*, équivalente à la propriété d'ordre *bien fondé*. Soit `R` une relation d'ordre sur `A`, `(Acc A R x)` sera prouvable s'il n'y a pas de suite infinie décroissante à partir de x par `R` :

```
Inductive Acc [A : Set; R : A->A->Prop]  : A->Prop :=
      Acc_intro : (x:A)((y:A)(R y x)->(Acc A R y))->(Acc A R x)
```

Nous verrons dans la partie III que la notion de *convergence* des algorithmes auto-stabilisants peut s'exprimer comme une variante de la notion d'accessibilité.

5.5 Les égalités entre constructeurs : incohérence

Nous illustrons maintenant par un exemple le problème des équations entre constructeurs de types inductifs. Supposons qu'on veuille définir les entiers relatifs, une solution naïve consiste à définir le type inductif `int` suivant :

```
Inductive int : Set :=
    OI : int
  | SI : int -> int
  | PI : int -> int.
```

et de poser les équations suivantes, via la commande `Axiom` qui permet d'ajouter une constante d'un type donné dans l'environnement [3] :

```
Axiom SPOeqO: (x:int)(SI (PI x))=x.
Axiom PSOeqO: (x:int)(PI (SI x))=x.
```

Malheureusement, l'élimination (forte) sur les types inductifs nous permet de prouver que deux termes commençant par des constructeurs différents sont différents. Par exemple on peut démontrer `(SI (PI OI)) = OI -> False`. On a donc introduit une preuve de `False`, c'est-à-dire une incohérence. Plus directement, on peut définir une fonction qui rendra `True` pour `OI` et `False` pour `(SI(PI OI))` :

```
Definition discriminate_head :=
[x:int]
 Cases x of
   OI => True
 | (SI _) => False
 | (PI _) => False
 end.
```

On peut alors d'une part prouver à l'aide de l'axiome `SPOeqO` que le terme `(discriminate_head (SI(PI OI)))` est égal à `(discriminate_head OI)` qui se réduit vers (donc est égal à) `True`, et d'autre part `(discriminate_head (SI(PI OI)))` se réduit vers `False`. On a donc prouvé que `True = False` et donc `False`. On donne le script de la preuve de `False` à la section 6.2.3.

La définition d'algèbres non libres, c'est-à-dire dont certains constructeurs ne sont pas libres, n'est donc pas possible directement sur les types inductifs.

Plusieurs solutions existent pour définir les quotients, la plus simple consistant à utiliser un *constructeur de type* qui bloque l'élimination. Nous verrons dans la suite les solutions déjà proposées (section 6.3.1 et 6.3.2).

Signalons que la solution naïve présentée ci-dessus comporte d'autres faiblesses :

— Les deux axiomes `SPOeqO` et `PSOeqO` étendent l'égalité mais n'étendent pas en conséquence la conversion, ce qui est pourtant souhaitable d'un point de vue pratique. Nous verrons que parmi les solutions déjà proposées, certaines ne corrigent pas ce problème.

— Le principe d'élimination du type `int` n'est plus valable, ni son comportement calculatoire.

L'objet de la deuxième partie de ce travail est de proposer une nouvelle solution à la représentation des algèbres non libres dans une extension du calcul des constructions inductives.

3. Avec tous les risques que cela comporte évidemment.

5.6 Le mécanisme de section

Coq permet de déclarer des objets et des hypothèses de manière temporaire afin d'alléger les définitions. Pour cela l'utilisateur :

1. Commence une nouvelle *section*,

   ```
   Section nelle_section.
   ```

2. Déclare les objets et hypothèses temporaires,

   ```
   Variable A:Set.
   Variable R:A -> A -> Prop.
   Hypothesis R_refl: (x:A)(R x x).
   Hypothesis R_commut: (x,y:A)(R x y) -> (R y x).
   Hypothesis R_trans: (x,y,z:A)(R x y) -> (R y z) -> (R x z).
   ```

3. Utilise ces objets dans des définitions et des preuves,

   ```
   Definition Rstar: A -> A -> Prop := ...

   Lemma Rstar_equiv :(x,y:A) (Rstar x y) <->  (R x y).
   ```

4. Sort de la section,

   ```
   End nelle_section
   ```

   Les objets et hypothèses temporaires du point 2 sont alors *mis en paramètre* – lorsque c'est nécessaire – dans les définitions et les preuves du point 3. La section est donc oubliée dans la suite.

   ```
   Check Rstar.
     : (A:Set)(R:A -> A -> Prop) A -> A -> Prop

   Check Rstar_equiv.
     : (A:Set; R:(A->A->Prop))
           ((x:A)(R x x))
           ->((x,y,z:A)(R x y)->(R y z)->(R x z))
           ->(x,y:A)(Rstar A R x y)<->(R x y)
   ```

Nous utiliserons cette fonctionnalité au cours de cette thèse, notamment dans la partie III sur les algorithmes auto-stabilisants.

Deuxième partie

Des types normalisés

Dans cette partie, on s'intéresse à la représentation des quotients dans le calcul des constructions inductives. Un quotient est défini à partir d'un ensemble et d'une relation d'équivalence sur ses éléments. Dans le cas de la théorie des types et du calcul des constructions inductives en particulier, cela signifie que ces structures (dites non libres) sont telles que deux termes clos, en forme normale et différents, peuvent être égaux.

On a déjà vu à la fin du chapitre précédent que cette caractéristique rend impossible l'utilisation des types inductifs pour représenter les quotients. Nous donnons dans le prochain chapitre d'autres raisons pour lesquelles les types inductifs – conçus pour représenter des structures libres – sont mal adaptés pour la représentation des quotients.

Bien entendu les quotients sont exprimables dans le calcul des constructions inductives, comme ils l'étaient dans le calcul des constructions. Nous verrons également dans le premier chapitre de cette partie quelles ont été les représentations proposées jusqu'à présent. Nous y verrons également pourquoi ces représentations ne sont pas satisfaisantes.

Nous nous trouvons alors dans une situation comparable à celle de T. Coquand et C. Paulin-Morhing devant le calcul des constructions. Non satisfaits d'un formalisme ($\mathcal{CC}$ pour eux, $\mathcal{CCI}$ pour nous) permettant pourtant d'exprimer tous les concepts voulus, nous décidons de le "polluer" avec de nouvelles constructions afin de représenter de manière plus agréable une certaine catégorie de structures.

On sacrifie ainsi un peu plus la sobriété minimaliste du calcul des constructions original au profit d'une plus grande simplicité d'utilisation. Les deux chapitres qui suivent sont consacrés à la description de cette extension du calcul des constructions inductives, appelée *calcul des constructions inductives avec types normalisés* (en abrégé : $\mathcal{CCI}^{nf}$).

Chapitre 6

Les quotients

6.1 Dans la théorie des ensembles

Dans le développement de la théorie des ensembles donné dans [Bou66], la notion de quotient est définie assez tôt. On associe en effet naturellement à une relation d'équivalence R, définie sur un ensemble E, l'ensemble des classes d'équivalence associées à R, appelé ensemble quotient de E par R. Les points important sont les suivants :

1. Définition d'une relation d'équivalence et d'un quotient ;
2. Caractérisation des relations et fonctions compatibles avec une relation d'équivalence ;
3. Définition de la méthode de définition de relations et de fonctions sur un quotient à partir de relations et de fonctions (compatibles) sur l'ensemble support : le *passage au quotient.*

On présente maintenant ces trois étapes :

6.1.1 Relation d'équivalence, ensemble quotient

On définit une relation d'équivalence, puis un quotient :

Définition 6.1.1 (Relation d'équivalence) *Une relation R sur un ensemble E est une relation d'équivalence ssi :*
— $\forall x.R(x,x)$ *(R est réflexive),*
— $\forall x \forall y.R(x,y) \Rightarrow R(y,x)$ *(R est symétrique)*
— $\forall x \forall y \forall z.R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$*($R$ est transitive)*

Définition 6.1.2 (Graphe d'une relation) *Soit R une relation sur un ensemble E. On appelle* graphe de R *l'unique sous-ensemble G_R de $E * E$ tel que :*

$$\forall x, y.R(x,y) \Longleftrightarrow (x,y) \in G_R$$

Définition 6.1.3 (Ensemble quotient) *Soit E un ensemble, et R une relation d'équivalence sur E, de graphe G_R. On appelle quotient de E par R, noté E/R, l'ensemble des classes d'équivalence de E par R, c'est-à-dire l'ensemble $G_R(x)$ des y tels que $R(x,y)$. On dira que E est le* support *de E/R, et que R est la relation de E/R.*

On peut définir ensuite l'application canonique d'un quotient :

Définition 6.1.4 *Soit E/R le quotient de l'ensemble E par la relation d'équivalence R, on appelle* application canonique *de E sur E/R l'unique application p telle que :*

$$\begin{array}{rl} p: & E \to E/R \\ & x \mapsto G_R(x) \end{array}$$

Dont on énonce la propriété principale :

Propriété 6.1.5 *Pour tout x, y de E, $R(x,y) \Leftrightarrow p(x) = p(y)$.*

Remarquons que la définition de l'application canonique n'est pas constructive, on ne donne pas d'algorithme permettant de calculer la classe d'équivalence à partir d'un élément. En fait, la plupart du temps on parlera de "la classe d'équivalence de x", notée $G_R(x)$ ou $[x]_R$, sans chercher à la caractériser davantage. Il est également courant d'appeler une classe d'équivalence par l'un de ses éléments lorsque cela ne prête pas à confusion.

6.1.2 Propriétés et fonctions compatibles avec une relation d'équivalence

Vient ensuite la notion de propriétés et fonctions compatibles avec une relation d'équivalence.

Définition 6.1.6 *Soit R une relation d'équivalence sur un ensemble E. On dira qu'une propriété P sur E est compatible avec R si pour tout élément x de E tel que $P(x)$, $\forall y.R(x,y) \Rightarrow P(y)$.*

Définition 6.1.7 *Soit R une relation d'équivalence sur un ensemble E. On dira qu'une fonction f de E vers F (un ensemble quelconque) est compatible avec R si (sous-entendu : pour tout y) la propriété $P(x) \Longleftrightarrow f(x) = y$ est compatible avec R.*

6.1.3 Passage au quotient

On donne maintenant la méthode standard pour définir des propriétés sur un quotient :

Définition 6.1.8 *Soit E/R un quotient et P une propriété compatible avec R. On appellera propriété déduite de P par passage au quotient la propriété P^{lift} sur E/R définie par : $P^{lift}(t) \Longleftrightarrow \exists x \in t.P(x)$.*

Une remarque de [Bou66] établit qu'une fonction f de E dans F compatible avec une relation R peut toujours s'exprimer comme la composée de l'application canonique de E/R et d'une fonction de E/R dans F. Cette dernière est définie de manière unique par f. C'est de cette façon qu'on définit des fonctions sur les quotients : *par passage au quotient*. On désignera la composition de fonction de manière classique à l'aide du symbole $\circ$: $f \circ g$ représente la fonction qui à x associe $f(g(x))$.

Propriété 6.1.9 *Soit R une relation d'équivalence dans un ensemble E, et g l'application canonique de E dans E/R. Pour qu'une application f de E dans F soit compatible avec R, il faut et il suffit que f puisse se mettre sous la forme $h \circ g$, h étant une application de E/R dans F. h est (uniquement) déterminée par f (remarque : mais pas constructivement).*

Si de plus on a une application s (pour section*) de E/R dans E telle que $\forall x \in E/R.g \circ s(x) = x$, alors $h = f \circ s$ par associativité de la composition $\circ$.*

On appelle h l'application déduite de f par passage au quotient.

Pour définir constructivement une fonction f de E/R dans F, il faut donc :
— définir une fonction f' sur le support E telle que $f'(x) = f(G_R(x))$,
— prouver que f' est compatible par rapport à la relation d'équivalence R du quotient,
— enfin en déduire la fonction par passage au quotient.

On peut déjà voir que pour donner un comportement calculatoire à une fonction passée au quotient [1], il est nécessaire d'avoir une fonction (s) permettant de retrouver un représentant à partir d'une classe d'équivalence de E/R.

6.2 Dans la théorie des types

6.2.1 La théorie de Martin-Löf

Per Martin Löf dans son ouvrage sur la théorie des types intuitionniste [ML84] énonce les principes de définition d'un ensemble. Cette définition est basée sur la notion d'*éléments canoniques* et *non canoniques*, par exemple pour $\mathbb{N}$ l'élément 10^{10} n'est pas en forme canonique. Voici les quatre principes de Martin Löf :

(1) Un ensemble A est défini par la description des processus :
— de construction de ces *éléments canoniques* et
— de construction de deux éléments canoniques *égaux*.

(2) Deux ensembles A et B sont égaux si
— ils ont les mêmes éléments et
— deux éléments égaux dans A sont égaux dans B, et vice versa.

(3) Un élément a d'un ensemble A est une méthode (un programme) qui, quand elle est exécutée, donne un élément canonique de A (10^{10} représente en fait un calcul).

(4) Deux éléments a et b de l'ensemble A sont égaux si ils donnent deux éléments canoniques égaux de A lorsqu'ils sont exécutés au sens de (3).

Suivant ce principe, connaître un ensemble, selon Martin Löf, c'est connaître ses éléments et savoir quand deux de ses éléments sont égaux. Il semble effectivement que si on possède ces deux clés, réunies dans le couple (ensemble,égalité) appelé *setoïde*, un ensemble est parfaitement défini.

Les règles de la théorie des types de Martin Löf font que l'égalité définie sur un ensemble est nécessairement une relation d'équivalence. On voit que la notion de quotient est en fait au cœur de la théorie des types, mais sous une forme différente de l'approche ensembliste :
— En théorie des ensembles, on définit un ensemble E, une relation d'équivalence sur E, puis on définit un *autre* ensemble E/R dont les éléments sont les classes d'équivalence de E par R.
— En théorie des types, on procède de manière plus directe : on définit un ensemble E *muni* d'une relation d'égalité R. L'ensemble ainsi défini représente la même structure que E/R, mais avec une notion d'éléments canoniques et non canoniques.
Plus précisément, l'ensemble des éléments canoniques est isomorphe au quotient E/R, et les éléments non canoniques ne sont que d'autres notations (d'autres représentants) pour les éléments canoniques, avec une notion de calcul permettant de retrouver l'élément canonique à partir d'un représentant. Cependant, même si ce principe permet une grande souplesse

1. C'est-à-dire donner un algorithme permettant de calculer cette fonction; on suppose ici qu'on a un algorithme pour calculer la fonction initiale.

dans les définitions, on va voir qu'il n'existe que sous une forme limitée dans le calcul des constructions inductives.

6.2.2 Le problème de l'égalité et de la conversion

Comme nous l'avons remarqué dans la première partie (section 3.6) à propos du calcul des constructions, la relation de conversion d'un calcul ne correspond pas nécessairement à l'égalité extensionnelle. Dans le système Coq par exemple, le choix a été fait de se restreindre à la clôture congruente de la réduction qui a l'avantage d'être décidable.

En conséquence, la définition des quotients pose le problème suivant : faut-il inclure la relation d'équivalence d'un quotient dans l'égalité extensionnelle ou dans l'égalité définitionnelle ? Il y a bien entendu deux réponses à cette question :

— Si notre théorie est extensionnelle (égalité extensionnelle = conversion), alors on peut mettre la relation du quotient dans la conversion sans restriction, puisque de toute façon la conversion n'est pas décidable.

— En revanche si on est dans un cadre intentionnel (égalité extensionnelle $\supsetneq$ conversion), alors il n'est possible d'inclure cette relation dans la conversion que si elle ne remet pas en cause sa décidabilité. Ce n'est évidemment pas le cas en général.

Dans le deuxième cas, une solution consiste à restreindre les quotients aux relations décidables. Cette solution semble intéressante et a déjà été étudiée ailleurs [Pot00], mais elle ne résout pas le problème décrit dans la section suivante.

6.2.3 Schémas d'élimination

On a vu qu'un principe d'élimination a pour but d'exprimer la forme des éléments canoniques d'un type. Cela permet d'une part de définir des fonctions et d'autre part de faire des preuves par récurrence : en définissant ses valeurs pour les éléments canoniques, on sait qu'une fonction est bien définie partout ; en prouvant une propriété pour tous les éléments canoniques, on sait qu'on l'a prouvée pour tout le type.

Dans le cas des quotients, comme le seul moyen de désigner une classe d'équivalence est de donner un représentant, la notion d'élément canonique prend une signification particulière. Par exemple, dans le quotient $\mathbb{Z}/3\mathbb{Z}$, on a trois classes d'équivalence : celle de 0 notée par exemple [0], celle de 1 ([1]) et celle de 2 ([2]). Cependant [3] est aussi un élément du quotient. Mais est-il canonique ? La réponse dépend de l'approche envisagée :

— [3] ne mène pas à un calcul et peut donc être considéré comme canonique ; c'est la vision dictée par les principes de la théorie des types.

— [3] possède un représentant [0], donc il n'est pas nécessaire de le considérer comme canonique ; cette vision est plus précise que la première mais moins générale puisqu'il n'est pas évident que tout quotient possède une telle notion de représentant canonique unique.

Ces deux réponses mènent à deux schémas d'élimination différents :

— Dans la première approche, le schéma d'élimination correspondant est le suivant.

$$\forall P.(\forall x \in \mathbb{Z}.P([x])) \Rightarrow \forall x \in \mathbb{Z}/3\mathbb{Z}.P(x)$$

qu'on traduit dans le calcul des constructions :

$$(P : \mathbb{Z}/3\mathbb{Z} \to s) \quad ((x : \mathbb{Z})(P\ [x])) \to (x : \mathbb{Z}/3\mathbb{Z})(P\ x)$$

où s est une sorte (`Prop`, `Set`...).

— Dans la deuxième approche, on voit qu'on peut déduire un schéma d'élimination assez intéressant puisqu'il permet de ne montrer une propriété que pour un représentant de chaque classe d'équivalence afin d'en déduire la propriété sur tout le quotient :

$$\forall P.(\forall x \in \{0,1,2\}.P([x])) \Rightarrow \forall x \in \mathbb{Z}/3\mathbb{Z}.P(x)$$

qu'on traduit également dans le calcul des constructions :

$$(P : \mathbb{Z}/3\mathbb{Z} \to s)\ \ ((x : \mathbb{Z})(x = O \vee x = 1 \vee x = 2) \to P([x])) \to (x : \mathbb{Z}/3\mathbb{Z})P(x)$$

Nous précisons dans la suite pourquoi ces deux principes ne sont pas acceptables sans modification, soit de leur définition, soit de la réduction associée. En un mot : ces deux principes ne prennent pas en compte la nécessité d'avoir une fonction ou une propriété compatible avec la relation du quotient.

Nous avons vu à la fin du chapitre 5 que la définition d'équations entre constructeurs de types inductifs n'était pas directement possible dans le calcul des constructions inductives, parce qu'elle engendrait une inconsistance. On rappelle l'exemple utilisé pour illustrer cela : une fois qu'on a posé des équations sur les constructeurs du type `int`, on est en mesure de construire une fonction rendant `True` pour `OI` et `False` pour `(SI (PI OI))`, qui sont pourtant égaux. Il s'ensuit que la preuve `I` de `True` est aussi une preuve de `False` par la règle de conversion. On donne ici le script Coq effectuant cela :

```
Inductive int : Set :=
    OI : int
  | SI : int -> int
  | PI : int -> int.

Axiom SPOeqO: (x:int)(SI (PI x))=x.
Axiom PSOeqO: (x:int)(PI (SI x))=x.


Definition discriminate_head :=
[x:int]
 Cases x of
   OI => True
 | (SI _) => False
 | (PI _) => False
 end.

Goal False.
Replace False with True.
 Exact I. (* True est facile à prouver *)
          (* Reste à prouver que True == False *)
 Replace True with (discriminate_head OI); Auto.
 Replace False with (discriminate_head (SI (PI OI))); Auto.
          (* Reste à prouver que *)
```

```
          (* (discriminate_head (SI(PI OI)))=(discriminate_head OI) *)
 Rewrite SPOeqO.
 Auto.
Save FalseisTrue.
```

En fait, pour définir la fonction `discriminate_head`, nous avons défini une fonction sur le type inductif `int`, puis nous l'avons *passée au quotient*, au sens de la section 6.1, sans prendre la précaution de vérifier que la fonction initiale était *compatible* avec la relation d'équivalence induite par les deux axiomes `SPOeqO` et `PSOeqO`.

On a vu que cette inconsistance était due à la réduction, qui calculait `(discriminate_head (SI(PI OI)))` en `False`. La réduction fautive est la ι-réduction, qui considère uniquement le constructeur de tête pour effectuer le choix de la branche à suivre dans la définition de la fonction `discriminate_head`. Il est clair que ce n'est pas un comportement correct en présence d'un type non libre. Il y a plusieurs comportements corrects possibles :

1ère solution : Conformément à la définition de fonction par passage au quotient dans la théorie des ensembles, le schéma d'élimination d'un type non libre (on l'appellera *type quotient*) exige une preuve de la compatibilité de la fonction initiale. Le premier principe d'élimination pour $\mathbb{Z}$ cité plus haut est donc modifié :

$$\forall P.(\forall x, y.R(x,y) \Rightarrow (P(x) \Leftrightarrow P(y))) \Rightarrow ((\forall x \in \mathbb{Z}.P([x])) \Rightarrow \forall x \in \mathbb{Z}/3\mathbb{Z}.P(x))$$

où R désigne la relation d'égalité du type quotient (telle que $R([0],[3])$).
Cette solution a été étudié sous différentes formes, nous en expliquons deux dans la suite : les *types quotients* et les *types congruences*. On remarque que la condition de compatibilité $(\forall x, y.(R\ x\ y) \Rightarrow ((P\ x) \Leftrightarrow (P\ y)))$ n'est intéressante que si on utilise ce principe pour définir une fonction. Elle ne sert à rien en ce qui concerne la démonstration par induction puisque le principe oblige à démontrer la propriété pour tous les éléments du type quotient. On voit en fait que cette condition n'est nécessaire que pour l'élimination forte. On verra avec les types quotients et les types congruences qu'on a alors un principe fort avec condition de compatibilité et un principe faible sans condition de compatibilité.
En revanche, si on considère le deuxième schéma d'élimination de $\mathbb{Z}$, on voit que cette condition de compatibilité est nécessaire, même pour le schéma faible, car c'est elle qui permet de restreindre la propriété à montrer aux cas : [0], [1] et [2].

$$\forall P.(\forall x, y.R(x,y) \Rightarrow (P(x) \Leftrightarrow P(y))) \Rightarrow ((\forall x \in \{0,1,2\}.P([x])) \Rightarrow \forall x \in \mathbb{Z}/3\mathbb{Z}.P(x))$$

2ème solution : La réduction ne réduit pas `(discriminate_head (SI(PI OI)))` en `False` mais d'abord en `(discriminate_head OI)` puis applique la ι-réduction. Pour que cette solution soit correcte, il faut donc être capable de remplacer dans un terme un sous-terme par sa *forme canonique*. Ici la forme canonique de `(SI(PI OI))` est clairement `OI`, mais on voit bien que l'existence d'une forme canonique pour chaque classe d'équivalence n'est pas une évidence pour un quotient quelconque. Nous avons néanmoins retenu cette solution car elle évite de faire dépendre la réduction d'une preuve.

6.3 Dans le calcul des constructions

6.3.1 Les types quotients

Les types quotients, que nous présentons plus bas, ont été définis en particulier dans le système Nuprl [CAB+86] dans le cadre d'une théorie extensionnelle, c'est-à-dire dans laquelle les égalités définitionnelles et propositionnelles sont identifiées.

Les types quotients ont été reformulés dans un cadre intentionnel par Hofmann [Hof95a, Hof95b], Barthe [Bar95], et enfin Boutin [Bou97]. La principale difficulté de cette reformulation réside dans la nature restreinte de l'égalité dans le cadre intentionnel. Ces travaux font le choix de mettre l'égalité des types quotients dans l'égalité de Leibniz ($A =_L B$ ssi $\forall P.P(A) \Leftrightarrow P(B)$), puisqu'elle permet d'effectuer des substitutions lors des raisonnements. Comme cette dernière n'est pas décidable en général, le statut de l'égalité des quotients reste un problème non résolu, même si Hofmann propose une interprétation des quotients vers les *setoïdes*, dans laquelle deux termes égaux s'interprètent vers des termes convertibles dans le modèle des setoïdes.

Samuel Boutin a axiomatisé les types quotients dans le système Coq, donc dans un cadre intentionnel. C'est cette axiomatisation que nous allons présenter maintenant. Nous verrons que l'égalité des quotients est définie dans l'égalité propositionnelle (de Leibniz), mais pas dans la conversion.

Axiomatisation dans Coq des types quotients Un type quotient est défini par l'association d'un type (le type *support*) A et d'une relation d'équivalence R de manière très similaire aux quotients de la théorie des ensembles.

On obtient alors un nouveau type, A/R, dont les éléments sont de la forme $[x]_R$ où x est de type A. On voit ici encore une fois qu'on désigne une classe d'équivalence par un de ses éléments.

Les schémas d'élimination suivants permettent de poser que les éléments de ce type seront tous de cette forme :

$$Lift : \forall B : \mathsf{Set}.\forall f : A \to B.\forall t : A/R.(lift\ f\ t) : B$$
$$Lift' : \forall P : A \to \mathsf{Set}.(\forall x : A.P([x]_R)) \Rightarrow \forall x : A/R.P(x)$$

Le premier permet de définir des fonctions sur le type quotient : l'opération *lift* permet de *faire passer au quotient* une fonction sur le type support.

Le deuxième permet d'effectuer des raisonnements par induction sur le quotient de la manière habituelle : pour prouver une propriété sur le type quotient, il faut la prouver pour tous les représentants possibles de chaque classe d'équivalence.

On rappelle maintenant la définition des types quotients en Coq donnée par Samuel Boutin dans la contribution au système Coq nommée `RATIONAL`. Elle suit à peu près l'enchaînement de [Bou66].

On commence par définir une nouvelle constante (la commande `Parameter` est équivalente à `Axiom`), qui sera le constructeur de type quotient, prenant en argument un type et une relation sur ce type. On verra dans la suite que c'est la clôture congruente de cette relation qui sera considérée.

```
Parameter MK_QUO : (A:Set)(R:A->A->Prop)Set.
```

On voit qu'un type quotient n'est pas un type inductif. Les principes d'élimination que nous lui donnerons plus loin ne correspondront pas à une réduction, mais seulement à l'égalité propositionnelle.

On définit ensuite la notion de compatibilité pour une fonction et une propriété (dans Set et Prop) :

```
Definition Compat := [A:Set][R:A->A->Prop][B:Set][f:A->B]
                     (x,y:A)(R x y) -> (f x) = (f y).

Definition Compat_set := [A:Set][R:A->A->Prop][f:A->Set]
                     (x,y:A)(R x y) -> (f x) == (f y).

Definition Compat_prop := [A:Set][R:A->A->Prop][P:A->Prop]
                      (x,y:A)(R x y) -> (P x) <-> (P y).
```

On pose ensuite les axiomes permettant de faire passer au quotient une fonction ou une propriété. `(lift ... f)` correspondra à la fonction `f` (prouvée compatible) passée au quotient. Notons encore une fois que même si on donnera plus loin d'autres axiomes permettant d'interpréter les fonctions et relations passées au quotient, il s'agit bien d'axiomes ne menant à aucun calcul :

```
Axiom lift : (A:Set)(R:A->A->Prop)(B:Set)(f:A->B)
                    (Compat A R B f) -> (MK_QUO A R) -> B.

Axiom lift_prop :  (A:Set)(R:A->A->Prop)(P:A->Prop)
                    (Compat_prop A R P) -> (MK_QUO A R) -> Prop.

Axiom lift_set :  (A:Set)(R:A->A->Prop)(P:A-> Set)
                    (Compat_set A R P) -> (MK_QUO A R) -> Set.
```

On définit les éléments du quotient (`MK_QUO`) grâce au constructeur `In`, qui correspond à l'application canonique du quotient.

```
Parameter In : (A:Set)(R:A->A->Prop)A->(MK_QUO A R).

Axiom From_R_to_L : (A:Set)(R:A->A->Prop)(x,y:A)
                  (R x y) -> (In A R x) = (In A R y).

Axiom From_L_to_R : (A:Set)(R:A->A->Prop)(x,y:A)
                   (In A R x) = (In A R y) -> (R x y).
```

On pose ensuite les schémas d'élimination établissant que les seuls éléments canoniques du quotient sont les éléments construits avec `In`. Autrement dit, pour prouver une propriété ou définir une fonction sur un quotient, il est suffisant de prouver ou définir sur les éléments de la forme `(In ... x)` :

```
Axiom Closure: (A:Set)(R:A->A->Prop)(X:(MK_QUO A R))(P:(MK_QUO A R)->Set)
                    ((x:A)(P (In A R x))) -> (P X).

Axiom Closure_prop :  (A:Set)(R:A->A->Prop)(X:(MK_QUO A R))
                (P:(MK_QUO A R)->Prop) ((x:A)(P (In A R x))) -> (P X).
```

Enfin on définit (toujours par des axiomes) comment réduire une fonction ou une relation passée au quotient. On voit qu'on procède un peu comme pour un type inductif à un constructeur, sauf que

— les principes d'élimination comportent une condition de compatibilité ;
— il n'y a pas de réduction interne (c'est-à-dire considérée dans la conversion) associée à ces principes, seulement un axiome décrivant cette réduction.

```
Axiom Reduce : (A:Set)(R:A->A->Prop)(B:Set)(f:A->B)
      (c:(Compat A R B f))(a: A)
     (lift A R B f c (In A R a)) = (f a).

Axiom Reduce_prop : (A:Set)(R:A->A->Prop)(f:A->Prop)
      (c:(Compat_prop A R f))(a: A)
     (lift_prop A R f c (In A R a)) == (f a).

Axiom Reduce_type : (A:Set)(R:A->A->Prop)(f:A->Set)
      (c:(Compat_set A R f))(a: A)
     (lift_set A R f c (In A R a)) == (f a).
```

Cette définition est très conforme à la définition des quotients en théorie des ensembles. Elle n'est cependant pas tout à fait satisfaisante car ni la réduction ni la conversion ne prennent en compte l'égalité du quotient. Même si on peut prouver `(In...x) = (In...y)`, ces deux termes ne seront pas convertibles. Donc, par exemple dans le quotient $\mathbb{Z}/3\mathbb{Z}$, si on a une preuve de $(P\ [0])$, ce n'est pas une preuve de $(P\ [3])$.

On illustre cela dans l'exemple de type quotient qui suit, toujours tiré du développement de Samuel Boutin : la définition traditionnelle de $\mathbb{Z}$ comme quotient de $\mathbb{N} * \mathbb{N}$. On utilise dans la suite une facilité d'écriture permettant d'écrire `|t|` à la place de `(In Z_typ Z_rel t)`. On utilise également l'accès aux deux éléments d'une paire par les notations `x.1` et `x.2`, ainsi que la notation `x+y` pour l'addition. Ces deux dernières notations sont délimitées par `[|` et `|]`.

On définit le type support et la relation, puis on déclare le quotient.

```
Definition Z_typ := nat * nat.

Definition Z_rel := [x,y:nat*nat] [| x.1 + y.2 = y.1 + x.2 |].

Hints Unfold  Z_rel.

Definition Z := (MK_QUO Z_typ Z_rel).
```

On définit ensuite le moins unaire comme le passage au quotient de l'opération `swap` échangeant les deux éléments d'une paire. Il faut évidemment prouver au préalable que `swap` est compatible avec la relation du quotient.

```
Definition swap : nat*nat -> Z
                := [x:nat*nat] |[| (x.2,x.1) |]|.

Lemma Compat_swap : (Compat Z_typ Z_rel Z swap).
```

```
...
Save.

Definition unary_minus : Z -> Z
                       := (lift Z_typ Z_rel  Z swap Compat_swap).
```

On remarque que la fonction `unary_minus` ne se *calcule* pas :

```
Eval Compute in (unary_minus |(0,1)|).
--> (unary_minus |(0,1)|)
```

En revanche on peut prouver l'égalité avec le résultat attendu :

```
Goal (unary_minus |(0,1)|) = |(1,0)|.
Rewrite Reduce;Assumption.
Save minus_01.
```

6.3.2 Les types congruences

La notion de type congruence est définie pour la première fois dans [RBS89] dans le cadre de la théorie des types extensionnelle (et pas d'un calcul). C'est une généralisation des types inductifs dans laquelle on déclare un type par la liste de ses constructeurs et les équations qui les lient. Les schémas d'élimination incluent les conditions de compatibilité adéquates.

On illustre ce principe par l'exemple des *bags*, un type non libre isomorphe aux multi-ensembles. On introduit le type des bags par la règle suivante :

$$\frac{A : \mathsf{Type}}{bags(A) : \mathsf{Type}}$$

On déclare ensuite les deux constructeurs $\emptyset$ et $\bullet$ des bags, comparables à ceux des listes :

$$\frac{}{\emptyset \in bags(A) : \mathsf{Type}}$$

$$\frac{a \in A \quad s \in bags(A) : \mathsf{Type}}{a \bullet s \in bags(A) : \mathsf{Type}}$$

On voit ici que l'appartenance à un type congruence est un objet sur lequel porte le typage. On a même une notion d'égalité de jugement, comme le montre la règle suivante, qui définit l'égalité sur le type des bags :

$$\frac{a \in A \quad b \in A \quad s \in bags(A) : \mathsf{Type}}{a \bullet b \bullet s = b \bullet a \bullet s \in bags(A) : \mathsf{Type}}$$

On voit comment cette définition s'inscrit dans la vision des ensembles de Martin-Löf : une fois qu'on a défini les éléments canoniques d'un ensemble, on donne le moyen de savoir quand deux éléments canoniques sont égaux.

On donne ensuite un schéma d'élimination aux bags :

$$\frac{\begin{array}{c} w \in bags(A) \to C(w) : \mathsf{Type} \\ t \in bags(A) \\ c \in C(\Phi) \\ a \in A; s \in bags(A); h \in C(s) \to d(a,s,h) \in C(a \bullet s) \\ a \in A; b \in A; s \in bags(A); h \in C(s) \to d(a, b \bullet s, d(b,s,h)) = d(b, a \bullet s, d(a,s,h)) \in C(a \bullet b \bullet s) \end{array}}{bags(A)\text{-}elim(t,c,d) \in C(t)}$$

La dernière prémisse – sans qui la règle donnerait le principe d'élimination des listes – exige que la fonction, définie dans un premier temps sur le type support par les deux objets c et d, soit démontrée compatible par rapport aux équations déclarées sur les constructeurs. Ici aussi, on voit apparaître un jugement d'égalité, typique de la théorie des types de Martin-Löf mais non autorisé dans le calcul des constructions.

Plusieurs travaux proposent une reformulation des types congruences dans un cadre intentionnel. Notamment Samuel Boutin dans sa thèse [Bou97], et Barthe et Geuvers [BG95]. Ces derniers définissent une notion de type congruence sensiblement différente afin de pallier le problème du comportement calculatoire des types congruences. En effet, comme pour les types quotients, il n'est pas question d'inclure les égalités entre constructeurs dans la conversion sans précaution puisque l'égalité ainsi engendrée ne serait pas décidable.

Barthe et Geuvers présentent donc une nouvelle notion de type congruence. On définit un type congruence en associant à la liste de ses constructeurs un ensemble de *règles de réécriture* sur ces constructeurs (à la place d'équations). Ce système de réécriture doit être *canonique*, c'est-à-dire confluant et terminant, pour être accepté. Cette définition est moins générale que les types congruences de Backhouse, mais elle permet d'inclure l'égalité des types congruences dans la conversion, puisque le système de réécriture du type congruence est ajouté à la réduction interne du système. On obtient donc avec cette méthode un comportement calculatoire satisfaisant pour une certaine classe de quotients : celle des quotients dont la relation d'équivalence correspond à un calcul. Plus précisément, les quotients dans lesquels il existe un représentant unique pour chaque classe d'équivalence, *calculable* à partir d'un représentant quelconque.

Sous cette hypothèse on peut se demander si l'approche traditionnelle des quotients reste pertinente. En particulier, puisqu'il est possible de calculer un représentant unique pour chaque classe d'équivalence, n'est-il pas possible de se débarrasser de la condition de compatibilité ? Il semble également que cette hypothèse sur les quotients rend possible l'amélioration des schémas d'élimination : l'ensemble des éléments canoniques d'un quotient constitue une *sorte recouvrante* au sens de [Com89] et [BJar].

Enfin, même si la réécriture est l'outil le plus adapté au raisonnement équationnel, son utilisation dans la représentation des structures dans le calcul des constructions s'avère difficile en pratique.

Les types normalisés sont une tentative d'exploitation et de généralisation de l'hypothèse du représentant canonique afin d'obtenir une représentation des quotients plus souple que les types congruences de Barthe et Geuvers.

6.3.3 Ensembles en théorie des types

Cette dernière approche nous donne l'occasion de nous pencher sur la remarque de Per Martin Löf, citée plus haut, selon laquelle un ensemble est défini par la forme de ses éléments et par le moyen de reconnaître lorsque deux éléments (canoniques) sont égaux. Certes, connaître les éléments

et la notion d'égalité sur un ensemble permet de définir complètement cet ensemble d'un point de vue classique, mais d'un point de vue constructif ?

En particulier lorsqu'il s'agit de *calculer* à partir d'un élément : après qu'il soit prouvé que le calcul était indépendant du choix d'un représentant canonique particulier, il faut en *choisir* un afin de calculer effectivement, ce qui n'est pas constructif au sens strict.

Lorsqu'on désigne un élément d'un quotient E/R par un élément x de E, on désigne en fait la classe d'équivalence de x ou plus précisément «l'élément de E/R correspondant à la classe d'équivalence de x». Mais n'est-ce pas l'axiome de description que nous utilisons ici ? On peut opter pour une reformulation de la phrase de Martin Löf : «pour connaître constructivement un ensemble il faut connaître les objets qui lui appartiennent et être capable de connaître la classe d'équivalence de chaque élément» c'est-à-dire connaître constructivement *l'application canonique* du quotient. Cet énoncé est plus restrictif, mais également plus conforme à l'approche intuitionniste.

Nous allons voir comment cette restriction permet de définir une nouvelle construction mathématique en théorie des types : les types normalisés. Pour résumer, un type normalisé est un type T muni d'une fonction de normalisation nf. Cette structure sera suffisante pour représenter le quotient T/R où R est la relation d'équivalence induite par nf. On se limitera bien entendu aux quotients dont la relation d'équivalence peut s'exprimer de cette manière.

Chapitre 7

Le calcul des constructions inductives avec types normalisés ($\mathcal{CCI}^{nf}$)

7.1 Introduction

Le but de ce chapitre est d'étendre le calcul des constructions inductives afin de permettre une définition des quotients plus agréable et plus puissante que la méthode axiomatique. On a vu en effet que cette dernière n'était pas satisfaisante à plusieurs égards :

— Elle ne permet pas d'inclure la relation d'équivalence du quotient dans la conversion du système.
— Il n'y a pas de mécanisme général pour construire le schéma d'élimination du quotient, mis à part le schéma trivial.
— Le comportement calculatoire des fonctions définies sur le quotient pose des problèmes.

7.1.1 Des quotients calculatoires

On cherche donc à définir une extension de $\mathcal{CCI}$ qui remédie, au moins en partie, à ces différents inconvénients ; c'est-à-dire que dans le nouveau système, on doit pouvoir prendre en compte la relation d'équivalence d'un quotient dans l'égalité définitionnelle, faciliter la génération de schémas d'élimination non triviaux, et enfin *calculer* sur les quotients de manière satisfaisante.

La piste que nous avons suivie est la suivante : certains quotients sont tels qu'il est possible de calculer un représentant particulier pour chacune de ses classes d'équivalence. Or, si pour un quotient A/R il existe une fonction nf qui calcule le représentant canonique d'une classe d'équivalence à partir d'un représentant quelconque x, alors A/R est isomorphe à l'image $nf(A)$ de A par nf.

Cette remarque a déjà été faite par Samuel Boutin dans [Bou97] mais n'a pas été exploitée. Elle apparaît également dans [BG95] sous une autre forme : la fonction nf y est définie par un système de réécriture extérieur au calcul des constructions. Cette approche a été présentée et discutée à la section 6.3.

Il semble en tout cas intéressant de se pencher sur la représentation dans le calcul des constructions inductives de ces quotients particuliers, que nous qualifierons de "calculatoires".

Définition 7.1.1 Soit A un ensemble et R une relation sur A, on dira que le quotient A/R est *calculatoire* si il existe une fonction nf de A vers A telle que :

$$\forall x, y \in A \quad R(x,y) \Leftrightarrow nf(x) = nf(y)$$

On appellera *nf fonction de normalisation de* A/R.

Remarque 7.1.1 *Un quotient calculatoire A/R de fonction de normalisation nf n'est pas nécessairement tel que : $\forall x \in A, R(x, nf(x))$. Dans ce chapitre nous ne supposerons pas que $\forall x \in A, R(x, nf(x))$, sauf lorsque ce sera précisé.*

Comme $R(x,y)$ est équivalent à $nf(x) = nf(y)$, le quotient A/R est complètement défini par la donnée de A et nf. On a donc une nouvelle façon de définir une structure non libre, qui consiste à donner un ensemble A et une fonction de A vers A.

Définition 7.1.2 Soit A un ensemble et nf une fonction de A vers A, on définit *l'ensemble normalisé* A/nf comme le quotient A/R où R est la relation définie par : $R(x,y) \Leftrightarrow nf(x) = nf(y)$.

7.1.2 Comment représenter ces quotients dans $\mathcal{CCI}$

Comme on veut exploiter les caractéristiques calculatoires de ces quotients, on rejette d'emblée la solution axiomatique ; on s'intéresse plutôt à une représentation plus interne permettant de définir un principe d'élimination général pour les types normalisés. La solution que nous proposons est comparable à celle consistant à ajouter les types inductifs au calcul des constructions : on ajoute les *types normalisés* au calcul des constructions inductives.

Le type normalisé $\mathsf{Norm}(A, nf)$ est le type A associé à la fonction de normalisation nf de type $A \to A$, $\mathsf{Class}(A, nf, x)$ désigne le terme de type $\mathsf{Norm}(A, nf)$ représentant la classe d'équivalence de x.

Bien entendu une classe d'équivalence aura plus d'un représentant et ces derniers devront être convertibles les uns avec les autres, donc Norm ne peut pas être considéré comme un type inductif pour les raisons expliquées à la section 6.2.3. Il nous reste donc à définir le schéma d'élimination des types normalisés.

On a vu à ce propos (section 6.2.3) que le schéma d'élimination consistant à faire passer au quotient une fonction ou une preuve définie sur le type support était assez désagréable d'un point de vue calculatoire, et peu pertinent lorsqu'il s'agit de faire des démonstrations par induction. C'est là que l'existence de la fonction nf prend son intérêt. On sait en effet qu'un quotient calculatoire a la propriété d'être isomorphe à l'ensemble de ses formes canoniques, ce qui nous amène à faire les remarques suivantes :

1. Une fonction f définie sur les éléments canoniques du type support définit trivialement une fonction f' sur le quotient par

   $$(f'\ \mathsf{Class}(A, nf, x)) =_{def} (f(nf\ x))$$

   On voit que la preuve de compatibilité n'est pas nécessaire.
2. Une propriété vraie sur les représentants canoniques de chacune des classes d'équivalence d'un quotient est vraie pour le quotient tout entier.

Une première possibilité consiste à inclure dans la réduction du système une règle qui remplace systématiquement tout terme $\mathsf{Class}(A, nf, x)$ par son représentant canonique $\mathsf{Class}(A, nf, (nf\ x))$. Nous voyons plusieurs raisons pour ne pas procéder ainsi :

— Premièrement, l'égalité entre les éléments d'une même classe d'équivalence ne correspond pas à une réduction mais bien à un *jugement*. Par exemple, dans le quotient $Z/3Z$, il ne semble pas pertinent de réduire $\mathsf{Class}(..., S(S(S\ 0)))$ en $\mathsf{Class}(..., 0)$; en revanche il semble très souhaitable que ces deux termes soient convertibles, ce qui peut sembler paradoxal puisque la conversion est traditionnellement définie comme la clôture transitive réflexive et symétrique de la relation de réduction. En fait nous ne voyons pas ce qui permet de justifier une telle restriction : il est seulement nécessaire que cette clôture soit *incluse* dans la conversion mais rien n'empêche la conversion d'être plus riche. Disons simplement que dans notre exemple 3 et 0 sont convertibles et pourtant tous deux en forme normale.

— Deuxièmement, il faudrait alors prévoir un mécanisme pour éviter d'appliquer à nouveau cette réduction et obtenir $\mathsf{Class}(..., nf(nf\ x))$.

— Enfin, cette méthode suppose que $\mathsf{Class}(..., x)$ et son représentant $\mathsf{Class}(..., nf\ x)$ appartiennent à la même classe d'équivalence. En toute généralité ceci n'est pas nécessaire.

La solution que nous avons adoptée consiste à effectuer cette transformation uniquement lorsque il est vraiment nécessaire de considérer le représentant canonique, c'est-à-dire lorsqu'on cherche à faire une élimination sur un terme d'un type normalisé. La conversion en revanche identifiera systématiquement les différents représentants d'une même classe d'équivalence.

7.2 Définition du calcul

On définit le calcul en donnant tout d'abord la syntaxe, puis les règles de réduction et la définition de la convertibilité, et enfin les règles de typage.

Ce système étant une extension de $\mathcal{CCI}$, lui-même défini dans la partie précédente, on se permettra parfois d'omettre certains détails spécifiques de $\mathcal{CCI}$, d'une part parce qu'il s'agirait d'une répétition, et d'autre part parce que l'extension est indépendante d'un certain nombre de détails de définition de $\mathcal{CCI}$.

7.2.1 Syntaxe

La syntaxe est basée sur les notations de la thèse de B. Werner [Wer94], rappelées en section 4.2.

On prend dans cette définition (mais il s'agit justement d'un de ces "détails" qui ne sont pas critiques pour l'extension) la hiérarchie de sortes $\mathsf{Set} : \mathsf{Type} : \mathsf{Extern}$.

Variables :

$$V ::= x, y, z \ldots$$

Sortes :

$$S ::= \mathsf{Set} \mid \mathsf{Type} \mid \mathsf{Extern}$$

Termes (les termes n'appartenant pas à $\mathcal{CCI}$ sont à la dernière ligne) :

$$\begin{aligned} T ::= \ & V \mid S \mid [V:T]T \mid (V:T)T \mid TT \\ & \mid \mathsf{Ind}(V:T)\{\vec{T}\} \mid \mathsf{Constr}(n,T)\ n \in N \mid \mathsf{Elim}(T,T,\vec{T},T)\{\vec{T}\} \\ & \mid \mathsf{Norm}(T,T) \mid \mathsf{Class}(T,T,T) \mid \mathsf{Elimnorm}(T,T,T,T) \end{aligned}$$

$\vec{T}$ dénote une séquence de termes. On reconnaît les constructions inductives habituelles Ind, Constr et Elim. Les trois nouvelles constructions Norm, Class et Elimnorm sont respectivement le constructeur de type normalisé, le constructeur de terme de type normalisé et le destructeur de type normalisé. Intuitivement, on peut comparer le type Norm à un type inductif à un constructeur Class, avec un schéma d'élimination spécifique : Elimnorm.

On utilisera les mêmes conventions et abus de notation que pour $\mathcal{CCI}$.

Remarque 7.2.1 *Tous les termes de $\mathcal{CCI}$ sont aussi des termes de $\mathcal{CCI}^{nf}$.*

Les notions d'occurrence positive (Pos), d'arité (Ar) et de type de constructeur ($constr$) sont les mêmes que pour $\mathcal{CCI}$.

Remarque 7.2.2 *Comme aucune des trois nouvelles constructions de $\mathcal{CCI}^{nf}$ n'est un lieur, la notion de variable libre de $\mathcal{CCI}$ s'étend de manière conservative à $\mathcal{CCI}^{nf}$. On notera donc cette notion $\mathcal{FV}$ indifféremment pour les termes de $\mathcal{CCI}$ et $\mathcal{CCI}^{nf}$. On donne ici les cas supplémentaires de la définition récursive de $\mathcal{FV}$ de la section 4.2 :*

Définition 7.2.1 On étend la notion de variable libre de $\mathcal{CCI}$ (section 4.2) à $\mathcal{CCI}^{nf}$ de la manière suivante :

$$\mathcal{FV}(\mathsf{Norm}(A, nf)) = \mathcal{FV}(A) \cup \mathcal{FV}(nf))$$
$$\mathcal{FV}(\mathsf{Class}(A, nf, t)) = \mathcal{FV}(A) \cup \mathcal{FV}(nf) \cup \mathcal{FV}(t)$$
$$\mathcal{FV}(\mathsf{Elimnorm}(A, nf, t, t')) = \mathcal{FV}(A) \cup \mathcal{FV}(nf) \cup \mathcal{FV}(t) \cup \mathcal{FV}(t')$$

La notion de substitution de $\mathcal{CCI}$ (section 4.2) est par conséquent étendue à $\mathcal{CCI}^{nf}$ de la manière suivante :

Définition 7.2.2 Pour tous termes A, nf, t, t', u, et toute variable x :

$$\mathsf{Norm}(A, nf)[x \leftarrow u] = \mathsf{Norm}(A[x \leftarrow u], nf[x \leftarrow u])$$
$$\mathsf{Class}(A, nf, t)[x \leftarrow u] = \mathsf{Class}(A[x \leftarrow u], nf[x \leftarrow u], t[x \leftarrow u])$$
$$\mathsf{Elimnorm}(A, nf, t, t')[x \leftarrow u] = \mathsf{Elimnorm}(A[x \leftarrow u], nf[x \leftarrow u], t[x \leftarrow u], t'[x \leftarrow u])$$

On remarque que, dans $\mathcal{CCI}$ comme dans $\mathcal{CCI}^{nf}$, mis à part les problèmes de renommage de variables, x (respectivement α) ne pourra pas apparaître dans T dans les constructions $[x : T]t$ et $(x : T)t$ (respectivement $\mathsf{Ind}(\alpha : T)\{\vec{u}\}$). On s'autorisera donc dans la suite à utiliser la définition allégée des substitutions suivante :

$$
\begin{array}{lcll}
x[x \leftarrow u] & = & u & \\
x[y \leftarrow u] & = & x & \text{si } x \neq y \\
[x:T]t'[x \leftarrow u] & = & [x:T]t' & \\
[x:T]t'[y \leftarrow u] & = & [x:T[y \leftarrow u]](t'[y \leftarrow u]) & \text{si } x \neq y \\
(x:T)t'[x \leftarrow u] & = & (x:T)t' & \\
(x:T)t'[y \leftarrow u] & = & (x:T[y \leftarrow u])(t'[y \leftarrow u]) & \text{si } x \neq y \\
(t'\ t'')[x \leftarrow u] & = & (t'[x \leftarrow u]\ t''[x \leftarrow u]) & \\
\mathsf{Ind}(\alpha:t)\{\vec{u}\}[\alpha \leftarrow u] & = & \mathsf{Ind}(\alpha:t)\{\vec{u}\} & \\
\mathsf{Ind}(\alpha:t)\{\vec{u}\}[x \leftarrow u] & = & \mathsf{Ind}(\alpha:t[x \leftarrow u])\{\overrightarrow{u[x \leftarrow u]}\} & \text{si } x \neq \alpha \\
\mathsf{Constr}(n,t)[x \leftarrow u] & = & \mathsf{Constr}(n,t[x \leftarrow u]) & \\
\mathsf{Elim}(t_1,t_2,\{\vec{v}\},t)\{\vec{u}\}[x \leftarrow u] & = & \mathsf{Elim}(t_1[x \leftarrow u],t_2[x \leftarrow u],\{\overrightarrow{v[x \leftarrow u]}\},t[x \leftarrow u])\{\overrightarrow{u[x \leftarrow u]}\} & \\
\mathsf{Norm}(A,nf)[x \leftarrow u] & = & \mathsf{Norm}(A[x \leftarrow u],nf[x \leftarrow u]) & \\
\mathsf{Class}(A,nf,t)[x \leftarrow u] & = & \mathsf{Class}(A[x \leftarrow u],nf[x \leftarrow u],t[x \leftarrow u]) & \\
\mathsf{Elimnorm}(A,nf,t,t')[x \leftarrow u] & = & \mathsf{Elimnorm}(A[x \leftarrow u],nf[x \leftarrow u],t[x \leftarrow u],t'[x \leftarrow u]) &
\end{array}
$$

7.2.2 Réduction

On définit la nouvelle réduction $\longrightarrow_{nf}$ de $\mathcal{CCI}^{nf}$.

Définition 7.2.3 ($\longrightarrow_{nf}$) On définit la nf-réduction comme la clôture par contexte de la règle suivante :

$$\mathsf{Elimnorm}(u_1,u_2,f,\mathsf{Class}(A,nf,t)) \longrightarrow_{nf} (f\ (nf\ t))$$

On voit ici comment les problèmes d'incohérence cités à la section 6.2.3 sont évités : l'élimination sur un terme de type normalisé consiste d'abord à appliquer la fonction de normalisation, puis à appliquer la fonction. Ainsi il n'est pas possible d'avoir deux réduits différents pour deux représentants de la même classe d'équivalence.

On peut maintenant définir la réduction du système :

Définition 7.2.4 ($\longrightarrow_{\mathcal{CCI}^{nf}}$) La réduction du calcul des constructions avec types normalisés est définie comme l'union de la nf-réduction et de la réduction de $\mathcal{CCI}$.

$$\longrightarrow_{\mathcal{CCI}^{nf}} \quad = \quad \longrightarrow_{\mathcal{CCI}} \cup \longrightarrow_{nf}$$

Définition 7.2.5 ($\equiv_{nf+\mathcal{CCI}}$) On notera $\equiv_{nf+\mathcal{CCI}}$ la congruence définie comme la clôture transitive, réflexive, symétrique et par contexte de $\longrightarrow_{\mathcal{CCI}^{nf}}$.

On voit que cette relation ne relie pas deux termes de la forme $\mathsf{Class}(A,nf,t)$ et $\mathsf{Class}(A,nf,t')$, même si $(nf\ t)\equiv_{nf+\mathcal{CCI}}(nf\ t')$. On voit donc que $\equiv_{nf+\mathcal{CCI}}$ n'est pas assez riche pour être la conversion de notre système. C'est pourquoi la définition qui suit sort un peu des schémas habituels.

7.2.3 Conversion

Une interprétation des termes de $\mathcal{CCI}^{nf}$ dans $\mathcal{CCI}^{nf}$

Intuitivement, on veut que l'égalité interne du calcul – l'égalité définitionnelle – qui sera utilisée dans la règle de conversion, soit la plus petite congruence contenant $\equiv_{nf+\mathcal{CCI}}$ et telle que si $(nf\ t)$ et $(nf\ t')$ sont égaux, alors $\mathsf{Class}(A, nf, t)$ et $\mathsf{Class}(A, nf, t')$ sont égaux. On choisit pour la définir d'utiliser une fonction φ de l'ensemble des termes de $\mathcal{CCI}^{nf}$ vers lui-même. Cette fonction, définie par induction sur les termes à la figure 7.2.3, appliquée à un terme, rend son représentant canonique.

$\varphi(\mathsf{Class}(A, nf, t))$	$=$	$\mathsf{Class}(\varphi(A), \varphi(nf), \varphi((nf\ t)))$
Dans les autres cas φ est le morphisme trivial :		
$\varphi(\mathsf{Extern})$	$=$	Extern
$\varphi(\mathsf{Set})$	$=$	Set
$\varphi(\mathsf{Type})$	$=$	Type
$\varphi(x)$	$=$	x
$\varphi((x : t_1)t_2)$	$=$	$(x : \varphi(t_1))\varphi(t_2)$
$\varphi([x : t_1]t_2)$	$=$	$[x : \varphi(t_1)]\varphi(t_2)$
$\varphi((t_1\ t_2))$	$=$	$(\varphi(t_1)\ \varphi(t_2))$
$\varphi(\mathsf{Constr}(i, t))$	$=$	$\mathsf{Constr}(i, \varphi(t))$
$\varphi(\mathsf{Ind}(x : t)\{\vec{t_i}\})$	$=$	$\mathsf{Ind}(x : \varphi(t))\{\varphi(\vec{t_i})\}$
$\varphi(\mathsf{Elim}(t_1, t_2, \vec{t_i}, t_3))$	$=$	$\mathsf{Elim}(\varphi(t_1), \varphi(t_2), \varphi(\vec{t_i}), \varphi(t_3))$
$\varphi(\mathsf{Norm}(A, nf))$	$=$	$\mathsf{Norm}(\varphi(A), \varphi(nf))$
$\varphi(\mathsf{Elimnorm}(A, nf, f, t))$	$=$	$\mathsf{Elimnorm}(\varphi(A), \varphi(nf), \varphi(f), \varphi(t))$
On étend $\varphi()$ aux environnements :		
$\varphi([])$	$=$	$[]$
$\varphi(\Gamma :: x : T)$	$=$	$\varphi(\Gamma) :: x : \varphi(T)$

FIGURE 7.1 – L'interprétation des termes de $\mathcal{CCI}^{nf}$ dans lui-même

Afin de définir une équivalence sur les termes obtenus par cette traduction, on définit également une nouvelle règle de réduction $\longrightarrow_{nf'}$, telle que si $t_1 \longrightarrow_{nf} t_2$ alors $\varphi(t_1) \longrightarrow_{nf'} \varphi(t_2)$:

Définition 7.2.6 $(\longrightarrow_{nf'})$ On définit la nf'-réduction comme la clôture par contexte de la règle :

$$\mathsf{Elimnorm}(u_1, u_2, f, \mathsf{Class}(A, nf, t)) \longrightarrow_{nf'} (f\ t)$$

Définition 7.2.7 $(\longrightarrow_{nf'+\mathcal{CCI}})$ On définit $\longrightarrow_{nf'+\mathcal{CCI}}$ de la manière suivante :

$$\longrightarrow_{nf'+\mathcal{CCI}} \ = \ \longrightarrow_{\mathcal{CCI}} \cup \longrightarrow_{nf'}$$

Définition 7.2.8 ($\equiv_{nf'+\mathcal{CCI}}$) On notera $\equiv_{nf'+\mathcal{CCI}}$ la congruence définie comme la clôture transitive, réflexive, symétrique et par contexte de $\longrightarrow_{nf'+\mathcal{CCI}}$.

La conversion

La fonction φ peut être vue comme une *interprétation à la Hofmann* [Hof95a] permettant de définir l'égalité définitionnelle. Celle-ci sera en effet définie de la manière suivante :

Définition 7.2.9 ($\equiv_{\mathcal{CCI}^{nf}}$) On définit l'égalité définitionnelle $\equiv_{\mathcal{CCI}^{nf}}$ de $\mathcal{CCI}^{nf}$ comme suit : Pour tous $\mathcal{CCI}^{nf}$-termes t_1 et t_2,

$$t_1 \equiv_{\mathcal{CCI}^{nf}} t_2 \text{ ssi } \varphi(t_1) \equiv_{nf'+\mathcal{CCI}} \varphi(t_2)$$

Remarque 7.2.3
— *Comme on le voit ci-dessus, la conversion de notre calcul* n'est pas la clôture congruente de sa relation de réduction ($\equiv_{nf+\mathcal{CCI}}$).
— *On précise que $\varphi(t)$ n'est pas convertible à t a priori.*

Comme on l'a vu dans la première partie (section 3.7), les deux propriétés que doit impérativement respecter la relation de conversion sont :
— La décidabilité, garante de la décidabilité du typage.
— La compatibilité avec la réduction, c'est-à-dire que les réduits de deux termes convertibles restent convertibles. Autrement dit $\equiv_{nf+\mathcal{CCI}} \subset \equiv_{\mathcal{CCI}^{nf}}$.

Cette dernière propriété est triviale lorsque la conversion est définie comme la clôture congruente de la réduction. Elle nécessite en revanche une démonstration dans notre cas. On remarque la conséquence suivante : si $(nf\ t) \equiv_{\mathcal{CCI}^{nf}} (nf\ u)$, alors $\mathsf{Class}(A, nf, t) \equiv_{\mathcal{CCI}^{nf}} \mathsf{Class}(A, nf, u)$. On voit ici comment les différents représentants d'une même classe sont bien convertibles. Cette propriété est énoncée et démontrée à la fin de ce travail (section 7.8) comme conséquence de la préservation des réductions par φ.

7.2.4 Typage

Pour le typage, on reprend exactement les règles – sauf la conversion que l'on redéfinit – de $\mathcal{CCI}$, et on ajoute quatre nouvelles règles correspondant aux nouvelles constructions de notre syntaxe. Les cinq règles sont données à la figure 7.2. Le système de typage complet de $\mathcal{CCI}^{nf}$ étant donné dans le tableau récapitulatif de la figure 7.3.

Les règles (FormN), (IntroN) et (Conv) n'appellent pas de commentaire particulier. En revanche les règles (ElimN) nécessitent une explication. Par souci de clarté nous allons d'abord expliquer sa version non dépendante (ElimN_{ND}). Elle se lit de la manière suivante : étant donnée une fonction H de type $A \to P$, le terme $\mathsf{Elimnorm}(A, nf, H, t)$ représente l'application à t de la fonction définie par passage au quotient (ou plutôt au type normalisé) de H. Autrement dit le terme

$$[t : \mathsf{Norm}(A, nf)]\mathsf{Elimnorm}(A, nf, H, t)$$

est la fonction de type $\mathsf{Norm}(A, nf) \to P$ correspondant à H.

$$(\textsc{FormN}) : \frac{\Gamma \vdash A : \mathsf{Set} \qquad \Gamma \vdash nf : A \to A}{\Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}}$$

$$(\textsc{IntroN}) : \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}}{\Gamma \vdash \mathsf{Class}(A, nf, t) : \mathsf{Norm}(A, nf)}$$

$$(\textsc{ElimN}_{\textsc{ND}}) \frac{\begin{array}{c}\Gamma \vdash P : s \quad \Gamma \vdash t : \mathsf{Norm}(A, nf) \\ \Gamma \vdash H : A \to P\end{array}}{\Gamma \vdash \mathsf{Elimnorm}(A, nf, H, t) : P}$$

$$(\textsc{ElimN}) : \frac{\begin{array}{c}\Gamma \vdash P : \mathsf{Norm}(A, nf) \to s \quad \Gamma \vdash t : \mathsf{Norm}(A, nf) \\ \Gamma \vdash H : (a : A)(P\ \mathsf{Class}(A, nf, a))\end{array}}{\Gamma \vdash \mathsf{Elimnorm}(A, nf, H, t) : (P\ (\mathit{IdNorm}(A, nf, t)))}$$

avec IdNorm(A,nf,t) $=_{def}$ $\mathsf{Elimnorm}(A, nf, ([x : A]\mathsf{Class}(A, nf, x)), t)$

$$(\textsc{Conv}) : \frac{\Gamma \vdash t : T_1 \qquad \Gamma \vdash T_1, T_2 : s \qquad T_1 \equiv_{\mathcal{CCI}^{nf}} T_2}{\Gamma \vdash t : T_2}$$

FIGURE 7.2 – La conversion et les règles de typage des type normalisés

Finalement, pour définir une fonction sur un type normalisé $\mathsf{Norm}(A, nf)$, il faut en premier lieu définir une fonction sur son type support A, puis appliquer $\mathsf{Elimnorm}$ pour la faire passer au type normalisé. Cette méthode est comparable à celle qui consiste à faire passer une fonction au quotient, sauf qu'ici il n'y a pas de condition de compatibilité à satisfaire. En effet, comme on l'a remarqué plus haut, la nf-réduction se charge de choisir le représentant canonique de l'argument avant d'appliquer la fonction.

La règle (ElimN) est la même que ($\textsc{ElimN}_{\textsc{ND}}$) pour le cas où H a un type dépendant de son argument. H n'est donc plus de type $A \to P$ mais $(s : A)(P\ \mathsf{Class}(A, nf, s))$, et $\mathsf{Elimnorm}(A, nf, H, t)$ n'a plus le type P mais $(P\ (\mathit{IdNorm}\ t))$. On pouvait s'attendre plutôt au type $P(t)$, mais la préservation du typage par nf-réduction ne serait alors plus vérifiée. IdNorm est la fonction qui, à un terme de la forme $\mathsf{Class}(A, nf, t)$, associe le terme $\mathsf{Class}(A, nf, (nf\ t))$, c'est-à-dire son représentant canonique. C'est la contrepartie au niveau du type de l'ajout implicite d'une normalisation dans la nf-réduction.

Règles de typage communes à $\mathcal{CCI}$ et $\mathcal{CCI}^{nf}$

$$(\text{Ax}_1)[] \vdash \mathsf{Set} : \mathsf{Type} \qquad (\text{Ax}_2)[] \vdash \mathsf{Type} : \mathsf{Extern}$$

$$(\text{Prod-S})\frac{\Gamma :: (x : t_1) \vdash t_2 : s}{\Gamma \vdash (x : t_1)t_2 : s} \qquad (\text{Lam-S})\frac{\Gamma \vdash (x : t_1)t_2 : s \qquad \Gamma :: (x : t_1) \vdash t : t_2}{\Gamma \vdash [x : t_1]t : (x : t_1)t_2}$$

$$(\text{W-Set})\frac{\Gamma \vdash t : \mathsf{Set} \quad \Gamma \vdash A : B \quad a \notin \Gamma}{\Gamma :: (a : t) \vdash A : B} \qquad (\text{W-Type})\frac{\Gamma \vdash t : \mathsf{Type} \quad \Gamma \vdash A : B \quad \alpha \notin \Gamma}{\Gamma :: (\alpha : t) \vdash A : B}$$

$$(\text{Var})\frac{\Gamma \vdash t_1 : t_2 \qquad (x : t_1) \in \Gamma}{\Gamma \vdash x : t_1} \qquad (\text{App})\frac{\Gamma \vdash t_2 : (x : T_1)T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash (t_2\ t_1) : T_2[x \leftarrow t_1]}$$

$$(\text{Ind})\frac{Ar(A, \mathsf{Set}) \quad \Gamma \vdash A : \mathsf{Type} \quad \forall i.(\Gamma :: (X : A) \vdash C_i(X) : \mathsf{Set}) \quad \forall i.constr(C_i(X))}{\Gamma \vdash \mathsf{Ind}(X : A)\{C_i(\vec{X})\} : A}$$

$$(\text{Intro})\frac{\Gamma \vdash \mathsf{Ind}(X : A)\{C_i(\vec{X})\} : T}{\forall n.(\Gamma \vdash \mathsf{Constr}(n, \mathsf{Ind}(X : A)\{C_i(\vec{X})\}) : C_n(\mathsf{Ind}(X : A)\{C_i(\vec{X})\}))}$$

$$(\text{W-Elim})\frac{\begin{array}{c}A \equiv (\vec{x} : \vec{A})\mathsf{Set} \quad I = \mathsf{Ind}(X : A)\{C_i(\vec{X})\} \quad \Gamma \vdash \vec{u} : \vec{A} \quad \Gamma \vdash t : (I\ \vec{u}) \\ \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to \mathsf{Set} \quad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i, I)\})\end{array}}{\Gamma \vdash \mathsf{Elim}(I, Q, \vec{u}, t)\{\vec{f_i}\} : (Q\ \vec{u}\ t)}$$

$$(\text{S-Elim})\frac{\begin{array}{c}A \equiv (\vec{x} : \vec{A})\mathsf{Set} \quad I = \mathsf{Ind}(X : A)\{C_i(\vec{X})\} \quad \Gamma \vdash \vec{u} : \vec{A} \quad \Gamma \vdash t : (I\ \vec{u}) \\ \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to \mathsf{Type} \quad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i, I)\}) \quad \forall i.Small(C_i(X))\end{array}}{\Gamma \vdash \mathsf{Elim}(I, Q, \vec{u}, t)\{\vec{f_i}\} : (Q\ \vec{u}\ t)}$$

Règles de typage spécifiques de $\mathcal{CCI}^{nf}$

$$(\text{FormN}) : \frac{\Gamma \vdash A : \mathsf{Set} \qquad \Gamma \vdash nf : A \to A}{\Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}} \qquad (\text{IntroN}) : \frac{\Gamma \vdash t : A \qquad \Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}}{\Gamma \vdash \mathsf{Class}(A, nf, t) : \mathsf{Norm}(A, nf)}$$

$$(\text{ElimN}_{\text{ND}})\frac{\begin{array}{c}\Gamma \vdash P : s \quad \Gamma \vdash t : \mathsf{Norm}(A, nf) \\ \Gamma \vdash H : A \to P\end{array}}{\Gamma \vdash \mathsf{Elimnorm}(A, nf, H, t) : P} \qquad (\text{Conv}) : \frac{\begin{array}{c}T_1 \equiv_{\mathcal{CCI}^{nf}} T_2 \\ \Gamma \vdash t : T_1 \quad \Gamma \vdash T_1, T_2 : s\end{array}}{\Gamma \vdash t : T_2}$$

$$(\text{ElimN}) : \frac{\begin{array}{c}\Gamma \vdash P : \mathsf{Norm}(A, nf) \to s \quad \Gamma \vdash t : \mathsf{Norm}(A, nf) \\ \Gamma \vdash H : (a : A)(P\ \mathsf{Class}(A, nf, a))\end{array}}{\Gamma \vdash \mathsf{Elimnorm}(A, nf, H, t) : (P\ (IdNorm\ t))}$$

avec $IdNorm(\text{t}) =_{def} \mathsf{Elimnorm}(A, nf, ([x : A]\mathsf{Class}(A, nf, x)), t)$

Égalité définitionnelle

$$t_1 \equiv_{\mathcal{CCI}^{nf}} t_2 \quad ssi \quad \varphi(t_1) \equiv_{nf' + \mathcal{CCI}} \varphi(t_2)$$

FIGURE 7.3 – Récapitulatif $\mathcal{CCI}^{nf}$

7.3 Propriétés préliminaires

On prouve dans cette section un ensemble de propriétés nécessaires pour la suite ; elle est largement inspirée de la thèse de B. Werner [Wer94].

Pour commencer, signalons que les systèmes de types de $\mathcal{CCI}$ et $\mathcal{CCI}^{nf}$ coïncident sur $\mathcal{CCI}$.

Lemme 7.3.1 Pour tout environnement Γ, et tous termes t et T de $\mathcal{CCI}$, $\Gamma \vdash_{\mathcal{CCI}} t : T$ si et seulement si $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$. Et les deux dérivations sont identiques.

Preuve : Dans les deux sens : par récurrence sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$(si) et $\Gamma \vdash_{\mathcal{CCI}} t : T$(seulement si), puis par cas sur la dernière règle utilisée, qui ne peut être qu'une règle commune aux deux calculs ou bien (Conv). Comme les règles de $\mathcal{CCI}$ sont toutes (sauf (Conv) qui diffère) dans $\mathcal{CCI}^{nf}$, pour chaque cas la même règle s'applique dans l'autre calcul par hypothèse de récurrence.

La seule règle de $\mathcal{CCI}$ qui n'est pas exactement dans $\mathcal{CCI}^{nf}$est la règle (Conv), mais comme pour tout terme u de $\mathcal{CCI}$, $\varphi(u) = u$, on applique (Conv) de manière exactement identique.

On énonce quelques propriétés mineures sur le typage dans $\mathcal{CCI}^{nf}$.

Lemme 7.3.2 (Variables libres) Si le jugement $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_1 : t_2$ est dérivable, alors $\mathcal{FV}(t_1) \cup \mathcal{FV}(t_2) \subset \mathcal{V}(\Gamma)$.

Preuve : Par induction sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_1 : t_2$, puis par cas sur la dernière règle utilisée. Tous les cas sont immédiats.

Lemme 7.3.3 (Sous-termes) Tout sous-terme d'un terme bien formé est bien formé.

Preuve : Par induction sur la preuve que le terme est bien formé.

Lemme 7.3.4 (Substitution) Si les jugements suivants sont dérivables :

$$\Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} u : B \qquad \Gamma_1 \vdash_{\mathcal{CCI}^{nf}} v : T$$

alors on peut dériver :

$$\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} u[x \leftarrow v] : B[x \leftarrow v]$$

Preuve : Par induction sur la preuve de $\Gamma_1 :: (x : A)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} u : B$. Puis par cas sur la dernière règle utilisée.

– (FormN)

$$\frac{\Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} A : \mathsf{Set} \qquad \Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} nf : A \to A}{\Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} u = \mathsf{Norm}(A, nf) : \mathsf{Set} = B}\ (\text{FormN})$$

Par hypothèse d'induction, pour tout v tel que $\Gamma_1 \vdash_{\mathcal{CCI}^{nf}} v : T$, on a :

$$\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} A[x \leftarrow v] : \mathsf{Set}[x \leftarrow v] = \mathsf{Set}$$
$$\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} nf[x \leftarrow v] : (A \to A)[x \leftarrow v]$$

Or $\mathsf{Norm}(A, nf)[x \leftarrow v] = \mathsf{Norm}(A[x \leftarrow v], nf[x \leftarrow v])$, donc on peut appliquer la règle (FormN) de la manière suivante :

$$\frac{\begin{array}{c}\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} A[x \leftarrow v] : \mathsf{Set} \\ \Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} nf[x \leftarrow v] : (A[x \leftarrow v] \to A[x \leftarrow v])\end{array}}{\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} \mathsf{Norm}(A[x \leftarrow v], nf[x \leftarrow v]) : \mathsf{Set} = \mathsf{Set}[x \leftarrow v]}\ (\text{FormN})$$

Ce qui conclut ce cas.

– (IntroN),(ElimN) Même raisonnement.

– (Conv)

$$\frac{T_1 \equiv_{\mathcal{CCI}^{nf}} T_2 \quad \Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} t : T_1 \quad \Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} T_1, T_2 : s}{\Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} t : T_2}\ (\text{Conv})$$

Par hypothèse d'induction, pour tout v tel que $\Gamma_1 \vdash_{\mathcal{CCI}^{nf}} v : T$, on a :

$$\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} t[x \leftarrow v] : T_1[x \leftarrow v]$$
$$\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} T_1[x \leftarrow v], T_2[x \leftarrow v] : s[x \leftarrow v] = s$$

D'autre part, on sait que $T_1 \equiv_{\mathcal{CCI}^{nf}} T_2 \Rightarrow T_1[x \leftarrow v] \equiv_{\mathcal{CCI}^{nf}} T_2[x \leftarrow v]$, donc on peut appliquer la règle (Conv), comme suit :

$$\frac{\begin{array}{c}\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} t[x \leftarrow v] : T_1[x \leftarrow v] \qquad T_1[x \leftarrow v] \equiv_{\mathcal{CCI}^{nf}} T_2[x \leftarrow v] \\ \Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} T_1[x \leftarrow v], T_2[x \leftarrow v] : s\end{array}}{\Gamma_1\ \Gamma_2[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} t[x \leftarrow v] : T_2[x \leftarrow v]}\ (\text{Conv})$$

Ce qui conclut ce cas.

– (Prod-S)

$$\frac{\Gamma_1 :: (x : T)\ \Gamma_2 :: (x : t_1) \vdash_{\mathcal{CCI}^{nf}} t_2 : B}{\Gamma_1 :: (x : T)\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} (x : t_1)t_2 : B}\ (\text{Prod-S})$$

Par hypothèse de récurrence, pour tout v tel que $\Gamma_1 \vdash_{\mathcal{CCI}^{nf}} v : T$, on a :

$$\Gamma_1\ (\Gamma_2 :: (x : t_1))[x \leftarrow v] \vdash_{\mathcal{CCI}^{nf}} t_2[x \leftarrow v] : B[x \leftarrow v]$$

On peut appliquer la règle (Prod-S) comme suit :

$$\frac{\Gamma_1\ \Gamma_2[x \leftarrow v] :: (x : t_1[x \leftarrow v]) \vdash_{\mathcal{CCI}^{nf}} t_2[x \leftarrow v] : B[x \leftarrow v]}{\Gamma_1\ \Gamma_2 \vdash_{\mathcal{CCI}^{nf}} (x : t_1[x \leftarrow v])t_2[x \leftarrow v] : B[x \leftarrow v]}\ (\text{Prod-S})$$

Ce qui conclut ce cas.

– Les autres cas sont également des conséquences directes des hypothèses d'induction.

Lemme 7.3.5 (affaiblissement) Si les jugements suivants sont dérivables :

$$\Gamma \vdash_{\mathcal{CCI}^{nf}} u : A$$
$$\Gamma' \vdash_{\mathcal{CCI}^{nf}} v : B \quad \text{(c'est-à-dire } \Gamma' \text{ bien formé)}$$

avec $\Gamma \subset \Gamma'$, alors on peut dériver :

$$\Gamma' \vdash_{\mathcal{CCI}^{nf}} u : A$$

Preuve : Par induction sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : A$.

Lemme 7.3.6 Si le jugement $\Gamma_1 :: (x : T)\Gamma_2 \vdash_{\mathcal{CCI}^{nf}} A : B$ est dérivable, alors $\Gamma_1 \vdash_{\mathcal{CCI}^{nf}} T : S$ est dérivable où S est la sorte associée à x, et de plus (par le lemme ci-dessus) $\Gamma_1 :: (x : T)\Gamma_2 \vdash_{\mathcal{CCI}^{nf}} T : S$.

Preuve : Par induction sur la dérivation de $\Gamma_1 :: (x : T)\Gamma_2 \vdash_{\mathcal{CCI}^{nf}} A : B$.

Lemme 7.3.7 Si : $\Gamma \vdash_{\mathcal{CCI}^{nf}} (x : T_1)T_2 : S$ alors $\Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} T_2 : S$.

Preuve : Par induction sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} (x : T_1)T_2 : S$. Celle-ci se termine par une application de (Prod-S), (W-Set), (W-Type)ou (Conv). Les trois premiers cas sont immédiats soit par la forme de la règle soit par hypothèse d'induction. Pour la règle (Conv), on sait par hypothèse d'induction que $\Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} T_2 : S'$ pour une certaine sorte S' telle que $S \equiv_{\mathcal{CCI}^{nf}} S'$, mais on sait alors que $S = S'$ car la conversion ne permet pas d'identifier les sortes.

Lemme 7.3.8 Si $\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} : T$ alors :

$$T \equiv_{\mathcal{CCI}^{nf}} A,$$
$$\Gamma \vdash_{\mathcal{CCI}^{nf}} A, T : \mathsf{Type},$$

et pour tous les i :

$$\Gamma :: (X : A) \vdash_{\mathcal{CCI}^{nf}} C_i(X) : \mathsf{Set}$$
$$\mathit{constr}(C_i(X))$$

Preuve : Par induction sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} : T$. La dernière règle utilisée est alors (Ind), (W-Set), (W-Type)ou (Conv). Les propriétés sont immédiates à chaque fois pour les trois premières. Pour la règle (Conv) :

- les deux propriétés $\Gamma \vdash_{\mathcal{CCI}^{nf}} A : \mathsf{Type},$; $\forall i.\mathit{constr}(C_i(X))$ et $\forall i.\Gamma :: (X : A) \vdash_{\mathcal{CCI}^{nf}} C_i(X) : \mathsf{Set}$ découlent immédiatement de l'hypothèse d'induction.
- $T \equiv_{\mathcal{CCI}^{nf}} A$ par hypothèse d'induction et par transitivité de $\equiv_{\mathcal{CCI}^{nf}}$.
- par la prémisse de la règle (Conv) on sait que $\Gamma \vdash_{\mathcal{CCI}^{nf}} T, T' : s$; d'autre part $\Gamma \vdash_{\mathcal{CCI}^{nf}} T' :$ Type par hypothèse d'induction. Donc par la règle (Conv) on obtient $\Gamma \vdash_{\mathcal{CCI}^{nf}} T : \mathsf{Type}$

Lemme 7.3.9 Si le jugement $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$ est dérivable, alors soit $T = \mathsf{Extern}$, soit $\Gamma \vdash_{\mathcal{CCI}^{nf}} T : S$ pour une certaine sorte S.

Preuve : Par induction sur la dérivable de $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$, puis par cas sur la dernière règle utilisée.

— Les cas des règles (Ax), (Prod-S), (Lam-S), (W-Set), (W-Type), (Ind), (W-Elim), (S-Elim), (Conv), (FormN), (IntroN) et (ElimN) sont immédiats.

– (App) $t = (t_1\ t_2)$ et $T = T_2[x \leftarrow t_1]$. Par hypothèse d'induction :
  — $\Gamma \vdash_{\mathcal{CCI}^{nf}} (x : T_1)T_2 : S$, or d'après le lemme 7.3.7 on a $\Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} T_2 : S$.
  — $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_1 : T_1$
  Et le lemme des substitutions (7.3.4) nous permet de conclure.

– (Intro) $t = \mathsf{Constr}(i, \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\})$ et $T = C_i(\mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\})$.
  La prémisse de la règle est de la forme :

$$\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} : U$$

  donc grâce au lemme 7.3.8 on en déduit :

$$\Gamma :: (X : A) \vdash_{\mathcal{CCI}^{nf}} C_i(X) : \mathsf{Set}$$
$$\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} : A$$

___ Et par le lemme des substitutions 7.3.4 on a bien $\Gamma \vdash_{\mathcal{CCI}^{nf}} T = C_i(\mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\}) : \mathsf{Set}$.

Lemme 7.3.10 (Inversion) Si le jugement $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$ est dérivable, alors :

(i) $t = a$
$$\Rightarrow \begin{cases} a \in \Gamma \\ T \equiv_{\mathcal{CCI}^{nf}} \Gamma(A) \end{cases}$$

(ii) $t = \alpha$
$$\Rightarrow \begin{cases} \alpha \in \Gamma \\ T \equiv_{\mathcal{CCI}^{nf}} \Gamma(\alpha) \end{cases}$$

(iii) $t = \mathsf{Set} \quad \Rightarrow \quad T \equiv_{\mathcal{CCI}^{nf}} \mathsf{Type}$

(iv) $t = \mathsf{Type} \quad \Rightarrow \quad T \equiv_{\mathcal{CCI}^{nf}} \mathsf{Extern}$

(v) $t = \mathsf{Extern} \quad \Rightarrow$ impossible

(vi) $t = (x : T_1)T_2$
$$\Rightarrow \begin{cases} \Gamma \vdash_{\mathcal{CCI}^{nf}} T_1 : s_1 \text{ avec } s_1 \in \{\mathsf{Set}, \mathsf{Type}\} \\ \Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} T_2 : s_2 \\ T \equiv_{\mathcal{CCI}^{nf}} s_2 \\ \text{avec } s_2 \text{ une sorte quelconque.} \end{cases}$$

(vii) $t = [x : T_1]t_2$

$$\Rightarrow \begin{cases} \Gamma \vdash_{\mathcal{CCI}^{nf}} T_1 : s_1 \text{ où } s_1 \text{ est la sorte de } x \\ \Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} t_2 : B \\ \Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} B : s_2 \\ T \equiv_{\mathcal{CCI}^{nf}} (x : T_1)B \\ \text{où } s_2 \in \{\mathsf{Set}, \mathsf{Type}\} \end{cases}$$

(viii) $t = (t_1\ t_2)$

$$\Rightarrow \begin{cases} \Gamma \vdash_{\mathcal{CCI}^{nf}} t_1 : (x : T_2)T_1 \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} t_2 : T_2 \\ T \equiv_{\mathcal{CCI}^{nf}} T_1[x \leftarrow t_2] \end{cases}$$

(ix) $t = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\}$

$$\Rightarrow \begin{cases} A \equiv_{\mathcal{CCI}^{nf}} (\vec{x} : \vec{A})\mathsf{Set} \\ \forall i. constr(C_i(X)) \\ \forall i. \Gamma :: (X : A) \vdash_{\mathcal{CCI}^{nf}} C_i(X) : \mathsf{Set} \end{cases}$$

(x) $t = \mathsf{Constr}(i, I)$

$$\Rightarrow \begin{cases} I = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \\ \text{avec les mêmes conditions sur } I \text{ que ci-dessus} \end{cases}$$

(xi) $t = \mathsf{Elim}(I, Q, \vec{a}, u)\{\overrightarrow{f_i}\}$

$$\Rightarrow \begin{cases} I = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \\ \text{avec les mêmes conditions sur } I \text{ que ci-dessus} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to s \text{ avec } s \in \{\mathsf{Set}, \mathsf{Type}\} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} \vec{a} : \vec{A} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} u : (I\ \vec{a}) \\ \forall i. \Gamma \vdash_{\mathcal{CCI}^{nf}} f_i : \Delta\{C_i(X, Q, \mathsf{Constr}(i, I))\} \\ T \equiv_{\mathcal{CCI}^{nf}} (Q\ \vec{a}\ u) \end{cases}$$

(xii) $t = \mathsf{Norm}(A, nf)$

$$\Rightarrow \begin{cases} T \equiv_{\mathcal{CCI}^{nf}} \mathsf{Set} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} A : \mathsf{Set} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} nf : A \to A \end{cases}$$

(xiii) $t = \mathsf{Class}(A, nf, u)$

$$\Rightarrow \begin{cases} T \equiv_{\mathcal{CCI}^{nf}} \mathsf{Norm}(A, nf) \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} A : \mathsf{Set} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} nf : A \to A \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} u : A \end{cases}$$

(xiv) $t = \mathsf{Elimnorm}(A, nf, f, u)$

$$\Rightarrow \begin{cases} \Gamma \vdash_{\mathcal{CCI}^{nf}} A : \mathsf{Set} \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} nf : A \to A \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} u : \mathsf{Norm}(A, nf) \\ \Gamma \vdash_{\mathcal{CCI}^{nf}} f : (s : A)(P\ \mathsf{Class}(A, nf, s)) \\ T \equiv_{\mathcal{CCI}^{nf}} (P\ (IdNorm\ u)) \end{cases}$$

Preuve : Par induction sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$, puis par cas sur la dernière règle utilisée. Dans chaque cas, seules certaines formes sont possibles pour t. On ne traite que les cas non immédiats.

– (Prod-S) $t = (x : T_1)T_2$. La conclusion et la prémisse de la règle donnent :

$$\Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} T_2 : s_2 \tag{7.1}$$
$$T = s_2$$

Le lemme 7.3.6 appliqué à (7.1) nous assure que $\Gamma \vdash_{\mathcal{CCI}^{nf}} T_1 : s$. La forme des règles (W-Set) et (W-Type) nous assure que $s \neq \mathsf{Extern}$, donc $s \in \{\mathsf{Set}, \mathsf{Type}\}$.

– (Lam-S) $t = [x : T_1]t_2$. La conclusion et les prémisses de la règle nous donnent :

$$T = (x : T_1)T_2$$
$$\Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} t_2 : T_2 \tag{7.2}$$
$$\Gamma \vdash_{\mathcal{CCI}^{nf}} (x : T_1)T_2 : s \tag{7.3}$$

Par (7.3) et le lemme 7.3.7, on a $\Gamma \vdash_{\mathcal{CCI}^{nf}} T_2 : s$. Par hypothèse d'induction appliquée à 7.3, on a : $\Gamma \vdash_{\mathcal{CCI}^{nf}} s : s_2$, donc $s \in \{\mathsf{Set}, \mathsf{Type}\}$. On en conclut que $B = T_2$ convient. De plus le lemme 7.3.6 appliqué au jugement 7.2 nous assure que $\Gamma \vdash_{\mathcal{CCI}^{nf}} T_1 : s_1$, où s_1 est la sorte de x.

– (W-Elim) $t = \mathsf{Elim}(I, Q, \vec{a}, u)\{\overrightarrow{f_i}\}$. Par les prémisse de la règle (W-Elim), on a immédiatement :

$$I = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \tag{7.4}$$
$$\Gamma \vdash_{\mathcal{CCI}^{nf}} \vec{a} : \vec{A} \tag{7.5}$$
$$\Gamma \vdash_{\mathcal{CCI}^{nf}} Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to \mathsf{Set} \tag{7.6}$$
$$T = (Q\ \vec{a}\ u) \tag{7.7}$$
$$\forall i. \Gamma \vdash_{\mathcal{CCI}^{nf}} f_i : \Delta\{C_i(X, Q, \mathsf{Constr}(i, I))\} \tag{7.8}$$
$$\Gamma \vdash_{\mathcal{CCI}^{nf}} u : (I\ \vec{a}) \tag{7.9}$$

D'après le lemme des sous-termes et (7.6), $I = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\}$ est bien typé dans l'environnement Γ. Le lemme 7.3.8 nous assure alors que les conditions sur I sont vérifiées.

– (Conv)

$$\frac{T' \equiv_{\mathcal{CCI}^{nf}} T \quad \Gamma \vdash t : T' \quad \Gamma \vdash T', T : s}{\Gamma \vdash t : T}\ (\text{Conv})$$

t est quelconque. Pour tous les cas, l'hypothèse de récurrence permet de conclure pour les jugements de typage concernant t et ses sous-termes, et la transitivité de la $\equiv_{\mathcal{CCI}^{nf}}$ permet de conclure pour les propriétés de la forme $T \equiv_{\mathcal{CCI}^{nf}} U$. On donne ci-dessous la preuve des autres propriétés.

(i) Par hypothèse d'induction $a \in \Gamma$.

(ii) Même raisonnement.

(ix) L'hypothèse d'induction nous assure que les conditions sur I sont vérifiées.

(x) Même argument.

(xi) Même argument.

- (FORMN) $t = \mathsf{Norm}(A, nf)$, la propriété est immédiate.
- (INTRON) $t = \mathsf{Class}(A, nf, u)$, on a immédiatement $T \equiv_{\mathcal{CCI}^{nf}} \mathsf{Norm}(A, nf)$ et $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : A$. l'hypothèse d'induction appliquée à la prémisse $\Gamma \vdash \mathsf{Norm}(A, nf) : \mathsf{Set}$ permet de conclure pour les deux autres propriétés.
- (ELIMN) $t = \mathsf{Elimnorm}(A, nf, f, u)$, on a immédiatement $T \equiv_{\mathcal{CCI}^{nf}} (P\ (\mathit{IdNorm}\ u))$, $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : \mathsf{Norm}(A, nf)$ et $\Gamma \vdash_{\mathcal{CCI}^{nf}} f : (s : A)(P\ \mathsf{Class}(A, nf, s))$, l'hypothèse d'induction nous permet ici aussi de conclure pour les deux dernières propriétés.

7.4 Confluence

Dans notre calcul, deux termes en forme normale peuvent être convertibles sans pour autant être égaux, la propriété de Church Rosser n'est donc évidemment pas vérifiée. Comme il nous faut néanmoins nous assurer de la décidabilité de la conversion, on va démontrer une version adaptée de cette propriété. Plus précisément, on va s'assurer que les interprétations par φ de deux termes convertibles ont un réduit commun par $\longrightarrow_{nf'+\mathcal{CCI}}$.

La preuve se fait de la manière suivante :

i. On prouve la confluence forte de la *réduction parallèle* correspondant à $\longrightarrow_{nf'+\mathcal{CCI}}$ sur tous les termes (y compris les mal typés).

ii. Puis on en déduit Church-Rosser sans avoir recours à la normalisation forte.

Définition 7.4.1 (Réduction parallèle $\twoheadrightarrow$) On définit la réduction parallèle de $\longrightarrow_{nf'+\mathcal{CCI}}$, notée $\twoheadrightarrow$, comme la plus petite relation vérifiant les clauses suivantes, dans lesquelles $u \twoheadrightarrow u'$, $v \twoheadrightarrow v'$ ... :

$$
\begin{array}{rcl}
u & \twoheadrightarrow & u \\
(u\ v) & \twoheadrightarrow & (u'\ v') \\
[x : T]u & \twoheadrightarrow & [x : T']u' \\
(x : T)u & \twoheadrightarrow & (x : T')u' \\
\mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} & \twoheadrightarrow & \mathsf{Ind}(X : A')\{\overrightarrow{C_i(X)'}\} \\
\mathsf{Elim}(I, Q, \vec{u}, t)\{\overrightarrow{f_i}\} & \twoheadrightarrow & \mathsf{Elim}(I', Q', \vec{u'}, t')\{\overrightarrow{f_i'}\} \\
([x : T]u)\ v & \twoheadrightarrow & u'[x \leftarrow v'] \\
\mathsf{Elim}(I, Q, \vec{u}, \mathsf{Constr}(j, I)\vec{m}) & \twoheadrightarrow & (\Delta[C_j(X)', f_j', \mathit{Fun\_\ Elim}(I', Q, \vec{u}, \vec{f_i'})]\vec{m}) \\
 & & \text{Avec } I \equiv \mathsf{Ind}(X : A)\{\overrightarrow{(}\ C_i(X))\} \\
\mathsf{Norm}(A, nf) & \twoheadrightarrow & \mathsf{Norm}(A', nf') \\
\mathsf{Class}(A, nf, t) & \twoheadrightarrow & \mathsf{Class}(A,', nf', t') \\
\mathsf{Elimnorm}(A, nf, f, t) & \twoheadrightarrow & \mathsf{Elimnorm}(A', nf', f', t') \\
\mathsf{Elimnorm}(A, nf, f, \mathsf{Class}(B, C, t)) & \twoheadrightarrow & (f'\ t')
\end{array}
$$

Lemme 7.4.2 Si $u \twoheadrightarrow u'$ et $v \twoheadrightarrow v'$ alors $u[x \leftarrow v] \twoheadrightarrow u'[x \leftarrow v']$.

Preuve : Par induction sur la preuve de $u \twoheadrightarrow u'$.

Lemme 7.4.3 (Confluence forte de $\twoheadrightarrow$) Pour tout terme t, u et v,

$$(t \twoheadrightarrow u \wedge t \twoheadrightarrow v) \Rightarrow \exists w.u \twoheadrightarrow w \wedge v \twoheadrightarrow w$$

Preuve : Par induction sur t

— $t = x$, Extern, Typeou Set, alors t est irréductible.

— $t = [x : T]t_1$ la réduction par $\twoheadrightarrow$ se fait sur les sous-termes, donc l'hypothèse d'induction permet de conclure immédiatement.

— $t = \mathsf{Norm}(A, nf)$, $t = \mathsf{Class}(A, nf, t_1)$, $t = (x : T)t_1$, même remarque.

— $t = (t'\ t_2)$, si t' n'est pas de la forme $[x : T]t_1$, alors l'hypothèse d'induction permet de conclure immédiatement, sinon $t = ([x : T]t_1\ t_2)$ On a plusieurs cas possibles :

(i) $u = u_1[x \leftarrow u_2]$ et $v = v_1[x \leftarrow v_2]$ avec

$$t_1 \twoheadrightarrow u_1 \text{ et } t_1 \twoheadrightarrow v_1$$
$$t_2 \twoheadrightarrow u_2 \text{ et } t_2 \twoheadrightarrow v_2$$

D'après l'hypothèse d'induction, il existe $w1$ et $w2$ tels que

$$u_1 \twoheadrightarrow w_1 \text{ et } v_1 \twoheadrightarrow w_1$$
$$u_2 \twoheadrightarrow w_2 \text{ et } v_2 \twoheadrightarrow w_2$$

Et le lemme 7.4.2 nous permet de conclure avec $t = w_1[x \leftarrow w_2]$.

(ii) $u = ([x : T_u]u_1\ u_2)$ et $v = v_1[x \leftarrow v_2]$ avec

$$T \twoheadrightarrow T_u$$
$$t_1 \twoheadrightarrow u_1 \text{ et } t_1 \twoheadrightarrow v_1$$
$$t_2 \twoheadrightarrow u_2 \text{ et } t_2 \twoheadrightarrow v_2$$

Il suffit de prendre le même w qu'au cas précédent.

(iii) $u = ([x : T_u]u_1\ u_2)$ et $v = ([x : T_v]v_1\ v_2)$, avec

$$T \twoheadrightarrow T_u \text{ et } T \twoheadrightarrow T_v$$
$$t_1 \twoheadrightarrow u_1 \text{ et } t_1 \twoheadrightarrow v_1$$
$$t_2 \twoheadrightarrow u_2 \text{ et } t_2 \twoheadrightarrow v_2$$

Et on prend encore le même w.

— $t = \mathsf{Elim}(I, Q, \vec{n}, (\mathsf{Constr}(j, I_0)\vec{m_k}))$, alors on a plusieurs cas possibles :

(i)

$$\begin{array}{rrcl} & u & = & (\Delta[C_j(X)', f'_j, Fun\_\,Elim(I', Q', \vec{n'}, \vec{f'_i})]\vec{m'_k}) \\ \text{et :} & v & = & (\Delta[C_j(X)'', f''_j, Fun\_\,Elim(I'', Q'', \vec{n''}, \vec{f''_i})]\vec{m''_k}) \\ \text{avec :} & & & \\ & I' & \equiv & \mathsf{Ind}(X : A')\{\overrightarrow{C_i(X)'}\} \\ & I'' & \equiv & \mathsf{Ind}(X : A'')\{\overrightarrow{C_i(X)''}\} \\ & A & \twoheadrightarrow & A''etA \twoheadrightarrow A'' \\ & C_i(X) & \twoheadrightarrow & C_i(X)'etC_i(X) \twoheadrightarrow C_i(X)'' \\ \text{etc.} & & & \end{array}$$

Soient I''', Q''' $C_i(X)'''$ etc. les réduits communs (hypothèse d'induction) de I' et I'', Q' et Q'' etc. On vérifie facilement que le w suivant satisfait la propriété :

$$w = (\Delta[C_j(X)''', f_j''', \mathit{Fun\_}\ \mathit{Elim}(I''', Q''', \vec{n'''}, \vec{f_i'''})]\vec{m_k'''})$$

(ii) $u = (\Delta[C_j(X)', f_j', \mathit{Fun\_}\ \mathit{Elim}(I', Q', \vec{n'}, \vec{f_i'})]\vec{m_k'})$ et $v = \mathsf{Elim}(I'', Q'', \vec{n''}, (\mathsf{Constr}(j, I_0'')\vec{m_k''}))$, on définit A''', Q''' etc comme ci-dessus, et on prend :

$$w = (\Delta[C_j(X)''', f_j''', \mathit{Fun\_}\ \mathit{Elim}(I''', Q''', \vec{f_i'''})]\vec{m_k'''})$$

(iii) $u = \mathsf{Elim}(I', Q', \vec{n'}, (\mathsf{Constr}(j, I_0')\vec{m_k'}))$ et $v = \mathsf{Elim}(I'', Q'', \vec{n''}, (\mathsf{Constr}(j, I_0'')\vec{m_k''}))$, les réductions s'effectuent dans les sous-termes de t, donc l'hypothèse d'induction permet de conclure immédiatement.

— $t = \mathsf{Elimnorm}(A, nf, f, t_1)$, si t_1 n'est pas de la forme $\mathsf{Class}(A_1, nf_1, t_2)$, alors les réductions vers u et v se font dans les sous-termes de t et l'hypothèse d'induction permet de conclure immédiatement, sinon :

$$t = \mathsf{Elimnorm}(A, nf, f, \mathsf{Class}(A_1, nf_1, t_2))$$

On a plusieurs cas :

(i) $u = (f'\ t_2')$ et $v = (f''\ t_2'')$, auquel cas l'hypothèse d'induction nous assure qu'il existe f''' et t_2''' tels que $f' \twoheadrightarrow f'''$, $f' \twoheadrightarrow f'''$, $t_2' \twoheadrightarrow f'''$ et $t_2' \twoheadrightarrow f'''$, et on prend $w = (f'''\ t_2''')$.

(ii) $u = \mathsf{Elimnorm}(A', nf', f', t_1')$ et $v = (f''\ t_2'')$, avec $A \twoheadrightarrow A'$ etc, et on prend le même w qu'au cas précédent.

(iii) $u = \mathsf{Elimnorm}(A', nf', f', t_1')$ et $v = \mathsf{Elimnorm}(A'', nf'', f'', t_1'')$, on prend :

$$w = \mathsf{Elimnorm}(A''', nf''', f''', t_1''')$$

A partir de cette propriété, il est bien connu que la confluence et Church-Rosser se déduisent facilement. Le principal argument étant que $\longrightarrow_{nf'+\mathcal{CCI}} \subset \twoheadrightarrow \subset \longrightarrow^*_{nf'+\mathcal{CCI}}$:

Propriétés 7.4.4 Pour tous termes t et t' :

$$t \longrightarrow_{nf'+\mathcal{CCI}} t' \Rightarrow t \twoheadrightarrow t' \Rightarrow t \longrightarrow^*_{nf'+\mathcal{CCI}} t'$$

Preuve : La première inclusion est triviale par cas sur la preuve de $t \longrightarrow_{nf'+\mathcal{CCI}} t'$. La deuxième se fait par induction sur la preuve de $t \twoheadrightarrow t'$.

Corollaire 7.4.5 (Confluence de $\longrightarrow_{nf'+\mathcal{CCI}}$) Soient t, t_1 et t_2 trois termes de $\mathcal{CCI}^{nf}$ tels que $t \longrightarrow^*_{nf'+\mathcal{CCI}} t_1$ et $t \longrightarrow^*_{nf'+\mathcal{CCI}} t_2$, alors il existe un terme t_3 tel que $t_1 \longrightarrow^*_{nf'+\mathcal{CCI}} t_3$ et $t_2 \longrightarrow^*_{nf'+\mathcal{CCI}} t_3$.

Preuve : Conséquence bien connue du lemme 7.4.3 et de la propriété 7.4.4.

Théorème 7.4.6 (Church-Rosser) Soient t_1 et t_2 deux termes de $\mathcal{CCI}^{nf}$ tels que $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_1, t_2 : T$ et $t_1 \equiv_{\mathcal{CCI}^{nf}} t_2$, alors il existe un troisième terme t_3 tel que $\varphi(t_1) \longrightarrow^*_{nf'+\mathcal{CCI}} \varphi(t_3)$ et $\varphi(t_2) \longrightarrow^*_{nf'+\mathcal{CCI}} \varphi(t_3)$

Preuve : Par définition de $\equiv_{\mathcal{CCI}^{nf}}$, on a $\varphi(t_1) \equiv_{nf'+\mathcal{CCI}} \varphi(t_2)$. On prouve alors la propriété par récurrence sur la preuve de $\varphi(t_1) \equiv_{nf'+\mathcal{CCI}} \varphi(t_2)$, en utilisant la confluence de $\longrightarrow_{nf'+\mathcal{CCI}}$ ci-dessus.

Corollaire 7.4.7 (Unicité de la forme normale pour $\longrightarrow_{nf'+\mathcal{CCI}}$) Soient deux termes t_1 et t_2 en forme $\longrightarrow_{nf'+\mathcal{CCI}}$-normale et tels que et $t_1 \equiv_{nf'+\mathcal{CCI}} t_2$, alors $t_1 = t_2$.

Preuve : C'est encore classique, on procède par l'absurde ; supposons qu'il existe deux tels termes inégaux, alors la propriété de Church-Rosser nous assure de l'existence d'un terme t_3 tel que $t_1 \longrightarrow^*_{nf'+\mathcal{CCI}} t_3$ et $t_1 \longrightarrow^*_{nf'+\mathcal{CCI}} t_3$, ce qui est contradictoire puisque t_1 et t_2 ne sont ni réductibles ni égaux.

7.5 Correction de la réduction vis-à-vis du typage

On va maintenant démontrer une des propriétés essentielles d'un λ-calcul : la préservation du typage par réduction (Subject reduction). Pour cela on commence par trois lemmes préliminaires. Le premier permet de déduire la convertibilité des sous-termes de deux termes convertibles de la forme $\lambda x : u.t$ ou $(x : u)t$, il est nécessaire à la preuve de la subject reduction.

Lemme 7.5.1 (Lemme clé) Pour tous termes t_1, t_2, t_3 et t_4 si

$$\begin{array}{rl} & (x : t_1)t_2 \equiv_{\mathcal{CCI}^{nf}} (x : t_3)t_4 \\ \text{ou bien :} & [x : t_1]t_2 \equiv_{\mathcal{CCI}^{nf}} [x : t_3]t_4 \end{array}$$

alors

$$\begin{array}{l} t_1 \equiv_{\mathcal{CCI}^{nf}} t_3 \\ t_2 \equiv_{\mathcal{CCI}^{nf}} t_4 \end{array}$$

Preuve : On utilise la propriété de confluence démontrée ci-dessus. La preuve est identique pour les deux cas, on ne montre que celui du produit.

Par définition de $\equiv_{\mathcal{CCI}^{nf}}$, $\varphi((x : t_1)t_2) \equiv_{nf'+\mathcal{CCI}} \varphi((x : t_3)t_4)$, soit :

$$(x : \varphi(t_1))\varphi(t_2) \equiv_{nf'+\mathcal{CCI}} (x : \varphi(t_3))\varphi(t_4)$$

Comme $\longrightarrow_{nf'+\mathcal{CCI}}$ vérifie la propriété de Church-Rosser (lemme 7.4.6 ci-dessus), on sait qu'il existe un terme u tel que :

$$\begin{array}{l} (x : \varphi(t_1))\varphi(t_2) \longrightarrow_{nf'+\mathcal{CCI}} u \\ (x : \varphi(t_3))\varphi(t_4) \longrightarrow_{nf'+\mathcal{CCI}} u \end{array}$$

Or par définition de $\longrightarrow_{nf'+\mathcal{CCI}}$, u est nécessairement de la forme $(x : \varphi(u_1))\varphi(u_2)$, avec

$$\varphi(t_1) \longrightarrow_{nf'+\mathcal{CCI}} u_1$$
$$\varphi(t_3) \longrightarrow_{nf'+\mathcal{CCI}} u_1$$
$$\varphi(t_2) \longrightarrow_{nf'+\mathcal{CCI}} u_2$$
$$\varphi(t_4) \longrightarrow_{nf'+\mathcal{CCI}} u_2$$

Donc $t_1 \equiv_{\mathcal{CCI}^{nf}} t_3$ et $t_2 \equiv_{\mathcal{CCI}^{nf}} t_4$.

Les deux lemmes mineurs suivants permettent de traiter certains cas de l'induction utilisée dans la preuve de subject reduction.

Lemme 7.5.2 Soient A et A' deux termes de $\mathcal{CCI}^{nf}$ tels que $A \longrightarrow_{nf'+\mathcal{CCI}} A'$. Si $Ar(A, s)$ alors $Ar(A', s)$.

Preuve : Par induction sur la preuve de $Ar(A, s)$.
- $A = s$, A n'est pas réductible, donc la propriété est vérifiée.
- $A = (x : t)B$, avec $Ar(B, s)$. Le cas où la réduction se fait dans B est immédiat par hypothèse d'induction. Si la réduction se fait dans t, alors $A' = (x : t')B$, avec $t \longrightarrow_{nf'+\mathcal{CCI}} t'$, et A' est donc bien une arité.

Lemme 7.5.3 Soit C et C' deux termes de $\mathcal{CCI}^{nf}$ tels que $C \longrightarrow_{nf'+\mathcal{CCI}} C'$. Si pour une variable X de C, $constr(C(X))$ alors $constr(C'(X))$.

Preuve : Par induction sur la preuve de $constr(C(X))$.
- $C = (X\vec{t_i})$, avec X non libre dans $\vec{t_i}$. La réduction s'effectue dans t_k, appartenant aux t_i, et donc $C' = (X\vec{u_i})$, où $u_i = t_i$ pour $i \neq k$, et $t_k \longrightarrow_{nf'+\mathcal{CCI}} u_k$, on voit que X n'est toujours pas libre dans $\vec{u_i}$, donc $constr(C')$.
- $C = (x : t)D$, avec $constr(D(X))$, X non libre dans t et $X \neq x$. Si la réduction se fait dans D, la propriété est immédiate par hypothèse d'induction. Si elle se fait dans t, alors $C' = (x : t')D$, comme X n'est pas libre dans t', on a immédiatement $constr(C'(X))$
- $C = P \to D$, avec $constr(D(X))$ et $Pos(X, P)$. Si la réduction se fait dans D, c'est encore une fois facile. Si elle se fait dans P, alors $C' = P' \to D$ avec $P \longrightarrow_{nf'+\mathcal{CCI}} P'$. Comme $Pos(X, P')$, on conclut que $constr(C'(X))$.

Lemme 7.5.4 (Correction de $\longrightarrow_{\mathcal{CCI}^{nf}}$) Si le jugement $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : A$ est dérivable, et si $t \longrightarrow_{\mathcal{CCI}^{nf}} t'$ et $\Gamma \longrightarrow_{\mathcal{CCI}^{nf}} \Gamma'$, alors les jugements suivants sont dérivables :

$$\Gamma \vdash_{\mathcal{CCI}^{nf}} t' : A$$
$$\Gamma' \vdash_{\mathcal{CCI}^{nf}} t : A$$

Preuve : Par induction sur la dérivation de $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : A$.
- (Prod-S), (Lam-S), (W-Set), (W-Type), (Intro), (IntroN), (FormN), (Conv) Les deux propriétés sont des conséquences directes de l'hypothèse d'induction appliquée aux prémisses de la règle. On montre le cas (Prod-S) :

$$(\text{Prod-S})\frac{\Gamma :: (x : t_1) \vdash t_2 : s}{\Gamma \vdash t = (x : t_1)t_2 : s = A}$$

(b) est immédiat par hypothèse d'induction. Pour (a) il nous faut distinguer deux cas possibles pour t' :

i. $t_1 \longrightarrow_{\mathcal{CCI}^{nf}} t'_1$ et $t' = (x : t'_1)t_2$. Par hypothèse d'induction, $\Gamma :: (x : t'_1) \vdash t_2 : s$ et on conclut par la règle (Prod-S).

ii. $t_2 \longrightarrow_{\mathcal{CCI}^{nf}} t'_2$ et $t' = (x : t_1)t'_2$. Par hypothèse d'induction, $\Gamma :: (x : t_1) \vdash t'_2 : s$ et on conclut aussi par la règle (Prod-S).

– (Var)

$$(\text{Var})\frac{\Gamma \vdash t_1 : t_2 \qquad (x : t_1) \in \Gamma}{\Gamma \vdash t = x : t_1 = A}$$

La réduction se fait nécessairement dans Γ ; si ce n'est pas dans le type associé à x dans Γ, la règle (Var) s'applique de la même manière. Sinon la prémisse de la règle nous assure que $\Gamma \vdash_{\mathcal{CCI}^{nf}} A : B$ avec $\Gamma = \Gamma_1 :: (x : t)\Gamma_2$ et $\Gamma' = \Gamma_1 :: (x : t')\Gamma_2$. Par le lemme 7.3.6 on en déduit que $\Gamma_1 \vdash_{\mathcal{CCI}^{nf}} t : S$, et donc on construit facilement une dérivation de $\Gamma' \vdash_{\mathcal{CCI}^{nf}} t : S$. Comme $\Gamma' \vdash_{\mathcal{CCI}^{nf}} x : T'$ et $t \equiv_{\mathcal{CCI}^{nf}} t'$, on a par (Conv)$\Gamma' \vdash_{\mathcal{CCI}^{nf}} x : t$.

– (Ind)

$$(\text{Ind})\frac{Ar(A, \mathsf{Set}) \quad \Gamma \vdash A : \mathsf{Type} \quad \forall i.(\Gamma :: (X : A) \vdash C_i(X) : \mathsf{Set}) \quad \forall i.constr(C_i(X))}{\Gamma \vdash t = \mathsf{Ind}(X : A)\{C_i(\vec{X})\} : A}$$

Par hypothèse d'induction,

$$\begin{aligned} \Gamma \vdash A' : \mathsf{Type} \\ \Gamma' \vdash A : \mathsf{Type} \\ \forall i.(\Gamma' :: (X : A) \vdash C_i(X) : \mathsf{Set}) \\ \text{pour tout } C'_i \text{ tel que } C_i \longrightarrow_{\mathcal{CCI}^{nf}} C'_i \quad \forall i.(\Gamma :: (X : A) \vdash C'_i(X) : \mathsf{Set}) \end{aligned}$$

D'autre part, les lemmes 7.5.2 et 7.5.3 nous assurent que $Ar(A', \mathsf{Set})$ et $constr(C'_i(X))$. On conclut donc immédiatement par la règle Ind.

– (App)

$$(\text{App})\frac{\Gamma \vdash t_2 : (x : T_1)T_2 \qquad \Gamma \vdash t_1 : T_1}{\Gamma \vdash t = (t_2\ t_1) : T_2[x \leftarrow t_1] = A}$$

Si la réduction s'effectue dans Γ, t_2 u t_1, les hypothèses de récurrence appliquées aux prémisses de la règles nous permettent de conclure facilement. Le cas intéressant ici est la réduction en tête, qui ne peut être qu'une β-réduction, donc :

$$\begin{aligned} t_2 = [x : T_3]t_4 \ \text{ et : } \ \Gamma \vdash_{\mathcal{CCI}^{nf}} [x : T_3]t_4 : (x : T_1)T_2 \\ t = ([x : T_3]t_4)t_1 \\ t' = t_4[x \leftarrow t_1] \end{aligned} \tag{7.10}$$

D'après le lemme d'inversion appliqué à 7.10, on a :

$$\begin{aligned} \Gamma :: (x : T_3) \vdash_{\mathcal{CCI}^{nf}} t_4 : B \\ \text{avec } (x : T_1)T_2 \equiv_{\mathcal{CCI}^{nf}} (x : T_3)B \end{aligned}$$

On en déduit par le lemme clé (7.5.1) que $T_2 \equiv_{\mathcal{CCI}^{nf}} B$ et $T_1 \equiv_{\mathcal{CCI}^{nf}} T_3$, donc $\Gamma :: (x : T_1) \vdash_{\mathcal{CCI}^{nf}} t_4 : T_2$, et par substitution, $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_4[x \leftarrow t_1] : T_2[x \leftarrow t_1]$. Ce qui conclut ce cas.

– (W-Elim), (S-Elim)

$$(\text{W-Elim})\frac{\begin{array}{c}A \equiv (\vec{x} : \vec{A})\mathsf{Set} \quad I = \mathsf{Ind}(X : A)\{\overrightarrow{C_i(X)}\} \quad \Gamma \vdash \vec{u} : \vec{A} \quad \Gamma \vdash v : (I\ \vec{u}) \\ \Gamma \vdash Q : (\vec{x} : \vec{A})(I\ \vec{x}) \to \mathsf{Set} \quad \forall i.(\Gamma \vdash f_i : \Delta\{C_i(I), Q, \mathsf{Constr}(i, I)\})\end{array}}{\Gamma \vdash t = \mathsf{Elim}(I, Q, \vec{u}, v)\{\vec{f_i}\} : (Q\ \vec{u}\ v) = A}$$

Il nous faut traiter ici de la ι-réduction en tête. On prend donc :

$$\begin{aligned} v &= (\mathsf{Constr}(j, I)\vec{v}) \\ t' &= (\Delta[C_j(I), f_j, \mathit{Fun\_\ Elim}(I, Q, \vec{u}, \vec{f_i})]\ \vec{v}) \end{aligned}$$

Par définition, $\mathit{Fun\_\ Elim}(I, Q, \vec{u}, \vec{f_i}) = [\vec{x} : \vec{A}][c : (I\ \vec{x})]\mathsf{Elim}(I, Q, \vec{x}, c)\{\vec{f_i}\}$, et grâce au lemme d'inversion appliqué à la conclusion de la règle ci-dessus, on construit la dérivation de :

$$\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathit{Fun\_\ Elim}(I, Q, \vec{u}, \vec{f_i}) : (x : \vec{A})(c : (I\ \vec{x}))(Q\ \vec{x}\ c) \tag{7.11}$$

Le lemme d'inversion appliqué à la conclusion de la règle ci-dessus nous donne : $\Gamma \vdash_{\mathcal{CCI}^{nf}} v : (I\ \vec{u})$. Comme tout $C_j(X)$ est de la forme $(\overrightarrow{y_k} : \overrightarrow{U_k})(X\ \overrightarrow{w})$ (forme générale de tout type de constructeur), on sait également que $\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Constr}(j, I) : C_j(I) = (\overrightarrow{y_k} : \overrightarrow{U_k})(I\ \vec{w})$. En appliquant de nouveau le lemme d'inversion, on obtient $\Gamma \vdash_{\mathcal{CCI}^{nf}} \vec{v} : \overrightarrow{U_k}$, et $\vec{w}[y_k \leftarrow \vec{v}] \equiv_{\mathcal{CCI}^{nf}} \vec{u}$. D'autre part, sachant 7.11, on peut prouver par induction sur $C_j(X)$ que si $C_j(X) = (\overrightarrow{y_k} : \overrightarrow{U_k})(X\ \overrightarrow{w})$ alors :

$$\Gamma \vdash_{\mathcal{CCI}^{nf}} \Delta[C_j(I), f_j, \mathit{Fun\_\ Elim}(I, Q, \vec{u}, \vec{f_i})] : (\overrightarrow{y_k} : \overrightarrow{U_k})(Q\ \overrightarrow{w}\ (\mathsf{Constr}(j, I)\ \overrightarrow{y_k}))$$

On en déduit :

$$\Gamma \vdash_{\mathcal{CCI}^{nf}} (\Delta[C_j(I), f_j, \mathit{Fun\_\ Elim}(I, Q, \vec{u}, \vec{f_i})]\ \vec{v}) : (Q\ \vec{u}\ (\mathsf{Constr}(j, I)\ \vec{v})) = (Q\ \vec{u}\ v)$$

Ce qui conclut ce cas.

– (ElimN)

$$(\text{ElimN}) : \frac{\begin{array}{c}\Gamma \vdash P : \mathsf{Norm}(A, nf) \to \mathit{Sort} \quad \Gamma \vdash u : \mathsf{Norm}(A, nf) \\ \Gamma \vdash H : (s : A)(P\ \mathsf{Class}(A, nf, s))\end{array}}{\Gamma \vdash t = \mathsf{Elimnorm}(A, nf, H, u) : (P\ \mathsf{Elimnorm}(A, nf, ([x : A]\mathsf{Class}(A, nf, x)), u)) = T}$$

Si la réduction se fait dans un sous-terme de t, la propriété est immédiate par hypothèse d'induction. Sinon c'est une nf-réduction :

$$\begin{aligned} u &= \mathsf{Class}(A, nf, u') \\ t' &= (H\ (nf\ u')) \end{aligned}$$

On en déduit facilement que $\Gamma \vdash_{\mathcal{CCI}^{nf}} t' : (P\ \mathsf{Class}(A, nf, (nf\ u')))$.
Or on voit que $T \equiv_{\mathcal{CCI}^{nf}} (P\ \mathsf{Class}(A, nf, (nf\ u')))$ en réduisant T de la manière suivante :

$$\begin{aligned} T = (P\ \mathsf{Elimnorm}(A, nf, ([x : A]\mathsf{Class}(A, nf, x)), \mathsf{Class}(A, nf, u'))) \\ \longrightarrow_{\mathcal{CCI}^{nf}} (P\ (([x : A]\mathsf{Class}(A, nf, x))(nf\ u'))) \\ \longrightarrow_{\mathcal{CCI}^{nf}} (P\ \mathsf{Class}(A, nf, (nf\ u')) \end{aligned}$$

Ce qui conclut ce cas.

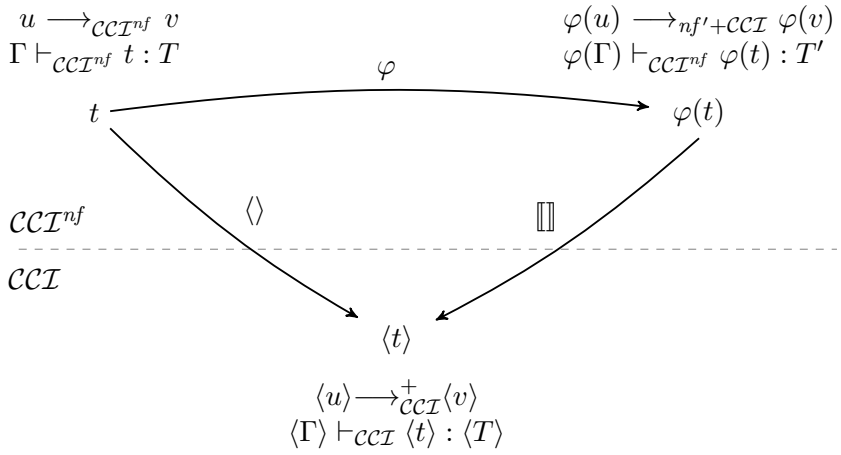


FIGURE 7.4 – Les trois traductions $\langle\rangle$, φ et $[\![]\!]$

Théorème 7.5.5 (Subject reduction) Soient t_1 et t_2 deux termes de $\mathcal{CCI}^{nf}$ tels que $t_1 \longrightarrow_{\mathcal{CCI}^{nf}} t_2$, si $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_1 : T$ alors $\Gamma \vdash_{\mathcal{CCI}^{nf}} t_2 : T$.

Preuve : Lemme 7.5.4.

7.6 Normalisation forte

Il faut prouver la normalisation forte des deux réductions $\longrightarrow_{\mathcal{CCI}^{nf}}$ et $\longrightarrow_{nf'+\mathcal{CCI}}$. La première nous assurera que les réductions terminent, la deuxième nous permettra, avec la propriété de Church-Rosser, de prouver la décidabilité de la conversion et donc du typage. Les deux propriétés se prouvent de la même manière : on ramène les terminaisons de $\longrightarrow_{\mathcal{CCI}^{nf}}$ et $\longrightarrow_{nf'+\mathcal{CCI}}$ à celle de $\longrightarrow_{\mathcal{CCI}}$ à l'aide des deux traductions $\langle\rangle$ et $[\![]\!]$ de $\mathcal{CCI}^{nf}$ vers $\mathcal{CCI}$. Ces deux nouvelles traductions sont définies ci-dessous et correspondent, avec φ, aux trois flèches du schéma figure 7.4 :

7.6.1 Deux traductions des termes de $\mathcal{CCI}^{nf}$ vers $\mathcal{CCI}$

Comme on l'a remarqué précédemment, un type normalisé peut être vu comme un type inductif à un argument, avec un schéma d'élimination spécifique. Nous allons préciser cette correspondance en définissant une traduction complète des termes de $\mathcal{CCI}^{nf}$ vers $\mathcal{CCI}$, notée $[\![]\!]$.

Cette traduction ressemble un peu à l'interprétation φ qui sert dans la définition de la conversion de notre système. $[\![]\!]$ quant à elle sera utilisée dans les sections suivantes pour démontrer la normalisation forte de $\mathcal{CCI}^{nf}$ comme conséquence de la propriété analogue de $\mathcal{CCI}$.

La traduction $[\![]\!]$ interprète les constructions communes aux deux calculs vers elles-mêmes, et les termes spécifiques de $\mathcal{CCI}^{nf}$ (Norm(...),Class(...) et Elimnorm(...)) vers un type inductif (*Indnorm* défini plus bas) de $\mathcal{CCI}$.

$\mathcal{T}(\mathcal{CCI}^{nf})$		$\mathcal{T}(\mathcal{CCI})$
$[\![\mathsf{Norm}(A, nf)]\!]$	$=$	$(Indnorm\ [\![A]\!]\ [\![nf]\!])$
$[\![\mathsf{Class}(A, nf, t)]\!]$	$=$	$(indclass\ [\![A]\!]\ [\![nf]\!]\ [\![t]\!])$
$[\![\mathsf{Elimnorm}(A, nf, f, t)]\!]$	$=$	$([\![f]\!]\ (rep\ [\![A]\!]\ [\![nf]\!]\ [\![t]\!]))$

Dans les autres cas (qu'on ne détaille pas) $[\![]\!]$ est le morphisme trivial.

On étend $[\![]\!]$ aux environnements :

$[\![[]]\!]$	$=$	$[]$
$[\![\Gamma :: x : T]\!]$	$=$	$[\![\Gamma]\!] :: x : [\![T]\!]$

FIGURE 7.5 – Définition de $[\![]\!]$

Définition 7.6.1 On définit la fonction $\langle\rangle$ comme la composée de φ et $[\![]\!]$:

$$\langle t\rangle = [\![\varphi(t)]\!]$$

On donne figure 7.6.1 la définition récursive de $\langle\rangle$. On voit qu'elle ne diffère de $[\![]\!]$ qu'à la ligne traitant la construction Class.

$\mathcal{T}(\mathcal{CCI}^{nf})$		$\mathcal{T}(\mathcal{CCI})$
$\langle\mathsf{Norm}(A, nf)\rangle$	$=$	$(Indnorm\ \langle A\rangle\ \langle nf\rangle)$
$\langle\mathsf{Class}(A, nf, t)\rangle$	$=$	$(indclass\ \langle A\rangle\ \langle nf\rangle\ \langle(nf\ t)\rangle)$
$\langle\mathsf{Elimnorm}(A, nf, f, t)\rangle$	$=$	$(\langle f\rangle\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle))$

Dans les autres cas (qu'on ne détaille pas) $\langle\rangle$ est le morphisme trivial.

On étend $\langle\rangle$ aux environnements :

$\langle[]\rangle$	$=$	$[]$
$\langle\Gamma :: x : T\rangle$	$=$	$\langle\Gamma\rangle :: x : \langle T\rangle$

FIGURE 7.6 – Propriétés de $\langle\rangle$

Les termes de $\mathcal{CCI}$ utilisés par $[\![]\!]$ et $\langle\rangle$ La définition de $[\![]\!]$ fait appel au type inductif *Indnorm*, à son constructeur *indclass*, et à son destructeur *rep*. Ces trois termes sont définis et typés de la manière suivante :

— *Indnorm* est le type correspondant à Norm de $\mathcal{CCI}^{nf}$, il est paramétré par A et nf (X est le nom du type inductif sans paramètre, on n'y fera pas référence dans la suite) :

$$Indnorm := [A : \mathsf{Set}]\ [nf : A \to A]\ \mathsf{Ind}(X : \mathsf{Set})\ \{A \to X\}$$

On peut en déduire le type de *Indnorm* (dans un environnement quelconque) :

Indnorm : (A :Set) (nf :A → A) Set.

— L'unique constructeur de *Indnorm* correspond à la construction Class, et on définit *indclass* comme ce constructeur paramétré lui aussi par A et *nf* :

indclass := [A :Set] [nf :A → A] Constr(1,(Indnorm A nf))

On en déduit :

indclass : (A :Set) (nf :A → A) A → (Indnorm A nf)

Pour clarifier ces deux définitions, on peut dire que ce sont celles que Coq engendre automatiquement lorsqu'on écrit la définition suivante :

```
Inductive Indnorm [A:Set, nf:A -> A] : Set :=
              indclass : A -> (Indnorm A nf).
```

où A et *nf* sont des *paramètres* du type inductif *Indnorm*.

— le destructeur *rep* du type *Indnorm* sera utilisé pour la traduction de Elimnorm :

rep :=[A :Set] [nf :A → A] [t :(Indnorm A nf)]
Cases t of
(indclass A nf x) ⇒ x
end.

Encore une fois on en déduit le type de *rep* :

rep :(A :Set) (nf :A → A) (Indnorm A nf) → A.

On énonce ici un résultat concernant *Indnorm* :

Lemme 7.6.2 Pour tous termes X, Y et T, et pour tout environnement Γ, si $\Gamma \vdash_{CCI} (Indnorm\ X\ Y) : T$ ou $\Gamma \vdash_{CCI} T : (Indnorm\ X\ Y)$, alors $\Gamma \vdash_{CCI} X : \mathsf{Set}$ et $\Gamma \vdash_{CCI} Y : X \to X$.

Preuve : Pour la première propriété, par induction sur la preuve de $\Gamma \vdash_{CCI} (Indnorm\ X\ Y) : T$, les seules règles possibles pour la racine de l'arbre sont (App), (Conv), (W-Set)et (W-Type), et la conclusion est immédiate à chaque fois.

Pour la deuxième propriété, on commence par prouver qu'il existe U tel que : $\Gamma \vdash_{CCI} (Indnorm\ X\ Y) : U$ (propriété de $\mathcal{CCI}$), puis on applique la première propriété.

7.6.2 Préservation des substitutions par les traductions

Lemme 7.6.3 Si $t = u[x \leftarrow v]$ est un terme de $\mathcal{CCI}^{nf}$, alors $[\![t]\!] = [\![u]\!][x \leftarrow [\![v]\!]]$.

Preuve : Par induction sur u, puis par cas. On suppose sans perte de généralité que toutes les occurrences de x dans u sont libres.

— Cas de base : Si $u =$ Extern, Set, Type ou $y \neq x$, alors le résultat est trivial. Si $u = x$ alors $[\![t]\!] = [\![x[x \leftarrow v]]\!] = [\![v]\!] = x[x \leftarrow [\![v]\!]] \overset{\langle\rangle}{=} [\![x]\!][x \leftarrow [\![v]\!]]$.

— $u = (y : T)w$. Comme on suppose que x est libre dans u, on sait que $y \neq x$, donc :

$[\![t]\!] = [\![u[x \leftarrow v]]\!]$
$= [\![(y : T)w[x \leftarrow v]]\!]$
$= [\![(y : T[x \leftarrow v])(w[x \leftarrow v])]\!]$
$= (y : [\![T[x \leftarrow v]]\!]))[\![w[x \leftarrow v]]\!]$
qui est par hypothèse d'induction égal à :
$= (y : [\![T]\!][x \leftarrow [\![v]\!]])([\![w]\!][x \leftarrow [\![v]\!]])$.
$= ((y : [\![T]\!])[\![w]\!])[x \leftarrow [\![v]\!]]$
$= [\![u]\!][x \leftarrow [\![v]\!]]$

— Les cas pour les autres constructions de $\mathcal{CCI}$ sont similaires.

— $u = \mathsf{Class}(A, nf, w)$.
$[\![t]\!] = [\![u[x \leftarrow v]]\!]$
$= [\![\mathsf{Class}(A, nf, w)[x \leftarrow v]]\!]$
$= [\![\mathsf{Class}(A[x \leftarrow v], nf[x \leftarrow v], w[x \leftarrow v]]\!])$
$= (\mathit{indclass}\ [\![A[x \leftarrow v]]\!]\ [\![nf[x \leftarrow v]]\!]\ [\![w[x \leftarrow v]]\!])$
qui est par hypothèse d'induction égal à :
$= (\mathit{indclass}\ [\![A]\!][x \leftarrow [\![v]\!]]\ [\![nf]\!][x \leftarrow [\![v]\!]]\ [\![w]\!][x \leftarrow [\![v]\!]])$
qui est par définition égal à :
$= [\![u]\!][x \leftarrow [\![v]\!]]$.

— Les cas $u = \mathsf{Elimnorm}(A, nf, f, t')$ et $u = \mathsf{Norm}(A, nf)$ sont similaires.

Lemme 7.6.4 Si $t = u[x \leftarrow v]$ est un terme de $\mathcal{CCI}^{nf}$, alors $\varphi(t) = \varphi(u)[x \leftarrow \varphi(v)]$.

Preuve : Même induction que ci-dessus.

Lemme 7.6.5 Si $t = u[x \leftarrow v]$ est un terme de $\mathcal{CCI}^{nf}$, alors $[\![t]\!] = [\![u]\!][x \leftarrow [\![v]\!]]$.

Preuve : Conséquence des deux lemmes 7.6.4 et 7.6.3 précédents.

7.6.3 Préservation de la réduction par les traductions

Lemme 7.6.6 Pour tous termes t_1 et t_2 de $\mathcal{CCI}^{nf}$, si $t_1 \longrightarrow_{\mathcal{CCI}^{nf}} t_2$ alors $\varphi(t_1) \longrightarrow_{nf'+\mathcal{CCI}} \varphi(t_2)$.

Preuve : Par induction sur t_1, puis par cas.

— Cas de base : $t_1 = x$, $t_1 = \mathsf{Extern}, \mathsf{Set}, \mathsf{Type}$, alors t_1 n'est pas réductible par $\longrightarrow_{\mathcal{CCI}^{nf}}$.

— $t_1 = (x : u)v$, alors $\varphi(t_1) = (x : \varphi(u))\varphi(v)$ et la réduction est faite nécessairement sur un sous-terme strict de t_1. On a deux cas possibles :

i. ou bien $t_2 = (x : u')v$ avec $u \longrightarrow_{\mathcal{CCI}^{nf}} u'$, alors $\varphi(t_2) = (x : \varphi(u'))\varphi(v)$. Par hypothèse d'induction, $\varphi(u) \longrightarrow_{nf'+\mathcal{CCI}} \varphi(u')$, par conséquent $\varphi(t_1) = (x : \varphi(u))\varphi(v) \longrightarrow_{nf'+\mathcal{CCI}} (x : \varphi(u'))\varphi(v) = \varphi(t_2)$.

ii. ou bien $t_2 = (x : u)v'$ avec $v \longrightarrow_{\mathcal{CCI}^{nf}} v'$, et on conclut par le même argument.

— $t_1 = [x : u]v$, similaire au cas précédent, c'est-à-dire que la réduction est faite sur un sous-terme strict de t_1, et l'hypothèse d'induction s'applique directement.

— $t_1 = \mathsf{Ind}(x : u)\{\vec{v}\}$, idem.

— $t_1 = \mathsf{Constr}(i, u)$, idem.

— $t_1 = \mathsf{Norm}(u, v, w)$, idem.

— $t_1 = \mathsf{Class}(t, u, v, w)$, idem.
— $t_1 = (u\ v)$, alors $\varphi(t_1) = (\varphi(u)\ \varphi(v))$, on distingue deux cas :
  i. Si la réduction se fait sur un sous-terme strict de t_1, alors on conclut avec le même argument que ci-dessus.
  ii. Si la réduction s'effectue en tête de t_1, alors c'est une β-réduction, $t_1 = ([x : T]u'\ v)$, et $t_2 = u'[x \leftarrow v]$. Donc $\varphi(t_1) = ([x : \varphi(T)]\varphi(u')\ \varphi(v))$. Par le lemme 7.6.4 $\varphi(t_2) = \varphi(u')[x \leftarrow \varphi(v)]$ et finalement $\varphi(t_1) \rightarrow_\beta \varphi(t_2)$.
— $t_1 = \mathsf{Elim}(u_1, u_2, \vec{v_i}, u_3)$, alors $\varphi(t_1) = \mathsf{Elim}(\varphi(u_1), \varphi(u_2), \varphi(\vec{v_i}), \varphi(u_3))$, et on distingue encore deux cas :
  i. Si la réduction se fait sur un sous-terme strict de t_1, alors on conclut avec le même argument que ci-dessus.
  ii. Si la réduction s'effectue en tête de t_1, alors c'est une ι-réduction et
  $t_1 = \mathsf{Elim}(u_1, u_2, \vec{v_i}, \mathsf{Constr}(k, T)\vec{m})$ et $t_2 = (v_k\ \vec{m})$.
  On a $\varphi(t_1) = \mathsf{Elim}(\varphi(u_1), \varphi(u_2), \varphi(\vec{v_i}), \mathsf{Constr}(k, \varphi(T))\varphi(\vec{m}))$ qui se réduit par ι à :
  $\varphi(v_k)\varphi(\vec{m}) = \varphi(t_2)$.
— $t_1 = \mathsf{Elimnorm}(A, nf, f, t)$, alors $\varphi(t_1) = (\varphi(f)\ (rep\ \varphi(A)\ \varphi(nf)\ \varphi(t)))$, il y a deux cas :
  i. Si la réduction se fait sur un sous-terme strict de t_1, alors on conclut avec le même argument que ci-dessus.
  ii. Si la réduction s'effectue en tête de t_1, alors c'est une nf'-réduction, et nous savons que :
  — $t = \mathsf{Class}(A, nf, u)$ et par conséquent :
  $\varphi(t_1) = \mathsf{Elimnorm}(\varphi(A), \varphi(nf), \varphi(f), \varphi(\mathsf{Class}(A, nf, u))))$
  $\overset{\langle\rangle}{=} \mathsf{Elimnorm}(\varphi(A), \varphi(nf), \varphi(f), \mathsf{Class}(\varphi(A), \varphi(nf), (\varphi(nf)\ \varphi(u))))$
  qui peut être réduit par $\longrightarrow_{nf'}$:
  $\longrightarrow_{nf'} (\varphi(f)\ (\varphi(nf)\ \varphi(u)))$
  — $t_2 = (f\ (nf\ u))$. Et donc $\varphi(t_2) = (\varphi(f)\ (\varphi(nf)\ \varphi(u)))$.
  Nous avons donc prouvé que $\varphi(t_1) \longrightarrow_{nf'+\mathcal{CCI}} \varphi(t_2)$.

—

Lemme 7.6.7 Pour tous termes t_1 et t_2 de $\mathcal{CCI}^{nf}$, si $t_1 \longrightarrow_{nf'+\mathcal{CCI}} t_2$ alors $[\![t_1]\!] \longrightarrow^+_{\mathcal{CCI}} [\![t_2]\!]$.

Preuve : Par induction sur t_1, puis par cas. La plupart des cas sont les mêmes que pour la preuve précédente, on ne traite que les cas qui diffèrent.
— $t_1 = \mathsf{Norm}(u, v)$, $t_2 = (Indnorm\ [\![u]\!]\ [\![v]\!])$. t_1 n'est pas réductible en tête, donc la réduction se fait sur un sous-terme strict de t_1, c'est-à-dire u ou v. Par hypothèse de récurrence elle se fait également sur un sous-terme de t_2.
— $t_1 = \mathsf{Class}(t, u, v)$, $t_2 = (indclass\ [\![t]\!]\ [\![u]\!]\ [\![v]\!])$ même déduction.
— $t_1 = (u\ v)$, alors $[\![t_1]\!] = ([\![u]\!]\ [\![v]\!])$, on distingue deux cas :
  i. Si la réduction se fait sur un sous-terme strict de t_1, alors on conclut avec le même argument que ci-dessus.
  ii. Si la réduction s'effectue en tête de t_1, alors c'est une β-réduction, $t_1 = ([x : T]u'\ v)$, et $t_2 = u'[x \leftarrow v]$. Donc $[\![t_1]\!] = ([x : [\![T]\!]][\![u']\!]\ [\![v]\!])$. Par le lemme 7.6.3 $[\![t_2]\!] = [\![u']\!][x \leftarrow [\![v]\!]]$ et finalement $[\![t_1]\!] \rightarrow_\beta [\![t_2]\!]$.
— $t_1 = \mathsf{Elimnorm}(A, nf, f, t)$, alors $[\![t_1]\!] = ([\![f]\!]\ (rep\ [\![A]\!]\ [\![nf]\!]\ [\![t]\!]))$, il y a deux cas :

i. Si la réduction se fait sur un sous-terme strict de t_1, alors on conclut avec le même argument que ci-dessus.

ii. Si la réduction s'effectue en tête de t_1, alors c'est une nf-réduction, et nous savons que :
— $t = \mathsf{Class}(A, nf, u)$ et par conséquent :
$[\![t_1]\!] = ([\![f]\!]\ (rep\ [\![A]\!]\ [\![nf]\!]\ [\![\mathsf{Class}(A, nf, u)]\!]))$
$\overset{\langle\rangle}{=} ([\![f]\!]\ (rep\ [\![A]\!]\ [\![nf]\!]\ (indclass\ [\![A]\!]\ [\![nf]\!]\ [\![u]\!])))$.
qui peut être réduit par β et ι à :
$\rightarrow^{+}_{\beta i}\ ([\![f]\!]\ [\![u]\!])$
— $t_2 = (f\ u)$). Et donc $[\![t_2]\!] = ([\![f]\!]\ [\![u]\!])$.
Nous avons donc prouvé que $[\![t_1]\!] \longrightarrow^{+}_{\mathcal{CCI}} [\![t_2]\!]$.

On en déduit la propriété de préservation de la relation de réduction par $\langle\rangle$:

Lemme 7.6.8 Pour tous termes t_1 et t_2 de $\mathcal{CCI}^{nf}$, si $t_1 \longrightarrow_{\mathcal{CCI}^{nf}} t_2$ alors $\langle t_1\rangle \longrightarrow^{+}_{\mathcal{CCI}} \langle t_2\rangle$.

Preuve : Conséquence immédiate des lemmes 7.6.6 et 7.6.7.

7.6.4 Préservation de la convertibilité par $\langle\rangle$

Lemme 7.6.9 Si $u \equiv_{\mathcal{CCI}^{nf}} v$ alors $\langle u\rangle \equiv_{\mathcal{CCI}} \langle v\rangle$.

Preuve : Par définition, $\varphi(u) \equiv_{nf'+\mathcal{CCI}} \varphi(v)$, par le lemme 7.6.7 on en conclut que $[\![\varphi(u)]\!] \equiv_{\mathcal{CCI}} [\![\varphi(v)]\!]$, c'est-à-dire $\langle u\rangle \equiv_{\mathcal{CCI}} \langle v\rangle$.

7.6.5 Préservation du typage par $\langle\rangle$ and φ

Lemme 7.6.10 Si $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : T$ alors $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle u\rangle : \langle T\rangle$.

Preuve : Par induction sur la preuve de l'hypothèse $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : T$, puis par cas selon la dernière règle utilisée pour prouver ce jugement.
– (Conv) La dernière règle utilisée est (Conv). Il existe donc T_1 et s tels que la prémisse de la règle contient les trois jugements :

i. $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : T_1$,

ii. $\Gamma \vdash_{\mathcal{CCI}^{nf}} T, T_1 : s$ et

iii. $T \equiv_{\mathcal{CCI}^{nf}} T_1$.

Par hypothèse d'induction, les deux premiers jugements sont préservés par $\langle\rangle$, c'est-à-dire :

$$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle u\rangle : \langle T_1\rangle$$
$$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle T\rangle, \langle T_1\rangle : \langle s\rangle$$

D'autre part, le lemme 7.6.9 nous assure que $\langle T\rangle \equiv_{\mathcal{CCI}} \langle T_1\rangle$. Par application de la règle de conversion de $\mathcal{CCI}$, on a donc :

$$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle u\rangle : \langle T\rangle$$

– (FormN) La dernière règle est (FormN), donc $u = \mathsf{Norm}(A, nf)$, $\langle u\rangle \overset{\langle\rangle}{=} \mathit{Indnorm}(\langle A\rangle\ \langle nf\rangle)$ et $\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Norm}(A, nf) : \mathsf{Set}$.
Par la règle (FormN), on sait que :

i. $\Gamma \vdash_{\mathcal{CCI}^{nf}} A : \mathsf{Set}$

ii. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle A\rangle : \langle\mathsf{Set}\rangle$

iii. $\Gamma \vdash_{\mathcal{CCI}^{nf}} nf : A \to A$.

iv. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle nf\rangle : \langle A\rangle \to \langle A\rangle$

Comme $\langle\mathsf{Norm}(A, nf)\rangle \overset{\langle\rangle}{=} (\mathit{Indnorm}\ \langle A\rangle\ \langle nf\rangle)$, et *Indnorm* a le type *(A :Set) (nf :A → A) Set*, par la règle (App), on peut conclure que : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle\mathsf{Norm}(A, nf)\rangle : \mathsf{Set} = \langle\mathsf{Set}\rangle$.

– (IntroN) La dernière règle est (IntroN), donc $u = \mathsf{Class}(A, nf, t)$, et :

$$\langle u\rangle \overset{\langle\rangle}{=} (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ \langle(nf\ t)\rangle)$$

et enfin $\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Class}(A, nf, t) : (\mathsf{Norm}\ A\ nf)$. Par (IntroN), nous savons que les jugements suivants sont valides :

i. $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : A$

ii. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle t\rangle : \langle A\rangle$

iii. $\Gamma \vdash_{\mathcal{CCI}^{nf}} (\mathsf{Norm}\ A\ nf) : \mathsf{Set}$

iv. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} (\mathit{Indnorm}\ \langle A\rangle\ \langle nf\rangle) : \langle\mathsf{Set}\rangle$

Par le lemme 7.6.2 et iv, on a :
$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle A\rangle : \mathsf{Set}$, et $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle nf\rangle : \langle A\rangle \to \langle A\rangle$.
Comme le type de *indclass* est *(A :Set) (nf :A → A) A → (Indnorm A nf)*, on peut conclure que : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ \langle(nf\ t)\rangle) : (\mathit{Indnorm}\ \langle A\rangle\ \langle nf\rangle)$.

– (ElimN) La dernière règle est (ElimN). Ce cas est un peu plus compliqué. On sait que :
— $\Gamma \vdash_{\mathcal{CCI}^{nf}} \mathsf{Elimnorm}(A, nf, f, t) : (P\ (\mathit{IdNorm}\ t))$
— $u = \mathsf{Elimnorm}(A, nf, f, t)$, et donc $\langle u\rangle \overset{\langle\rangle}{=} (\langle f\rangle\ (\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle))$
Par (ElimN) on sait également que ;

i. $\Gamma \vdash_{\mathcal{CCI}^{nf}} f : (s : A)(P\ \mathsf{Class}(A, nf, s))$

ii. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle f\rangle : (s : \langle A\rangle)(\langle P\rangle\ (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ s)))$

iii. $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : \mathsf{Norm}(A, nf)$

iv. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle t\rangle : (\mathit{Indnorm}\ \langle A\rangle\ \langle nf\rangle)$

v. $\Gamma \vdash_{\mathcal{CCI}^{nf}} P : \mathsf{Norm}(A, nf) \to \mathit{Sort}$.

Par le lemme 7.6.2 et iv, on a :
$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle A\rangle : \mathsf{Set}$, et $\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle nf\rangle : \langle A\rangle \to \langle A\rangle$
Et donc par le type de *rep* on obtient :
$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} (\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle) : \langle A\rangle$
Notons v le terme $(\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)$. On applique à présent la règle (App) :
$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} (\langle f\rangle\ v) : (\langle P\rangle\ (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ s)))[s \leftarrow v]$.
Finalement :

$$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle u\rangle : (\langle P\rangle\ (\mathit{indclass}\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ (\mathit{rep}\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)))) \tag{7.12}$$

D'autre part, on effectue la traduction du type de t :
$\langle P\ (IdNorm(A, nf, t))\rangle \stackrel{\langle\rangle}{=} (\langle P\rangle\langle IdNorm(A, nf, t)\rangle)$
$\langle IdNorm(A, nf, t)\rangle =_{def} (\langle \mathsf{Elimnorm}(A, nf, ([x : A]\mathsf{Class}(A, nf, x)), t)\rangle)$
Par définition de $\langle\rangle$ et *IdNorm* :
$\langle IdNorm(A, nf, t)\rangle \stackrel{\langle\rangle}{=} (\langle [x : A]\mathsf{Class}(A, nf, x)\rangle\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle))$
$\langle IdNorm(A, nf, t)\rangle \stackrel{\langle\rangle}{=} [x : \langle A\rangle](indclass\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ x))\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)$
Qui se réduit par β à :
$(indclass\ \langle A\rangle\ \langle nf\rangle\ (\langle nf\rangle\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)))$
Par conséquent :
$\langle P\ (IdNorm(A, nf, t))\rangle \equiv_{\mathcal{CCI}} (\langle P\rangle\ (indclass\ \langle A\rangle\ \langle nf\rangle\ (rep\ \langle A\rangle\ \langle nf\rangle\ \langle t\rangle)))$ et on retrouve le type de $\langle u\rangle$ dans l'environnement $\langle\Gamma\rangle$(7.12). Donc par la règle de conversion on peut conclure que :

$$\langle\Gamma\rangle \vdash_{\mathcal{CCI}} \langle u\rangle : \langle P\ (IdNorm(A, nf, t))\rangle$$

– (Lam-S) La dernière règle est (Lam-S). Donc $u = [x : t_1]t$, et $T = (x : t_1)t_2$, donc :
— $\langle u\rangle = [x : \langle t_1\rangle]\langle t\rangle$
— $\langle T\rangle = (x : \langle t_1\rangle)\langle t_2\rangle$.
Par les prémisses de la règle on a :

i. $\Gamma \vdash_{\mathcal{CCI}^{nf}} (x : t_1)t_2 : s$
ii. Donc par hypothèse d'induction : $\langle\Gamma\rangle \vdash_{\mathcal{CCI}^{nf}} (x : \langle t_1\rangle)\langle t_2\rangle : s$
iii. $\Gamma :: (x : t_1) \vdash_{\mathcal{CCI}^{nf}} t : t_2$
iv. Donc par hypothèse d'induction : $\langle\Gamma\rangle :: (x : \langle t_1\rangle) \vdash_{\mathcal{CCI}^{nf}} \langle t\rangle : \langle t_2\rangle$.

On peut donc appliquer la règle (Lam-S) et obtenir :

$$\frac{\langle\Gamma\rangle \vdash_{\mathcal{CCI}^{nf}} (x : \langle t_1\rangle)\langle t_2\rangle : s \qquad \langle\Gamma\rangle :: (x : \langle t_1\rangle) \vdash_{\mathcal{CCI}^{nf}} \langle t\rangle : \langle t_2\rangle}{\langle\Gamma\rangle \vdash_{\mathcal{CCI}^{nf}} [x : \langle t_1\rangle]\langle t\rangle : (x : \langle t_1\rangle)\langle t_2\rangle}\ (\text{Lam-S})$$

— Les autres règles de $\mathcal{CCI}$ se traitent de la même manière, c'est-à-dire par simple application de l'hypothèse de récurrence.

Lemme 7.6.11 Si $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : T$ alors il existe T' tel que : $\varphi(\Gamma) \vdash_{\mathcal{CCI}^{nf}} \varphi(u) : T'$.

Preuve : Par induction sur la preuve de l'hypothèse $\Gamma \vdash_{\mathcal{CCI}^{nf}} u : T$, puis par cas selon la dernière règle utilisée pour prouver ce jugement.

7.6.6 Normalisation forte de $\longrightarrow_{\mathcal{CCI}^{nf}}$ et $\longrightarrow_{nf'+\mathcal{CCI}}$

En utilisant les propriétés de $\langle\rangle$ démontrées plus haut, on déduit la normalisation forte de notre calcul à partir des propriétés connues de $\mathcal{CCI}$.

Lemme 7.6.12 S'il existe une réduction infinie Δ partant d'un $\mathcal{CCI}^{nf}$-terme t bien typé par $\longrightarrow_{\mathcal{CCI}^{nf}}$, alors il existe une réduction infinie Δ' partant du $\mathcal{CCI}$-terme bien typé $\langle t\rangle$ par $\longrightarrow_{\mathcal{CCI}}$.

Preuve : La réduction existe par itération du lemme 7.6.8, et $\langle t\rangle$ est bien typé par le lemme 7.6.10.

Théorème 7.6.13 $\mathcal{CCI}^{nf}$ est fortement normalisant sur les termes biens typés.

Preuve : C'est un corollaire du lemme précédent, comme $\mathcal{CCI}$ est fortement normalisant ([Wer94]).

Lemme 7.6.14 S'il existe une réduction infinie Δ par $\longrightarrow_{nf'+\mathcal{CCI}}$ partant de l'interprétation $\varphi(t)$ d'un $\mathcal{CCI}^{nf}$-terme t bien typé, alors il existe une réduction infinie Δ' partant du $\mathcal{CCI}$-terme bien typé $\langle t\rangle$ par $\longrightarrow_{\mathcal{CCI}}$.

Preuve : La réduction existe par itération du lemme 7.6.7, et $\langle t\rangle$ est bien typé par le lemme 7.6.10.

Théorème 7.6.15 $\longrightarrow_{nf'+\mathcal{CCI}}$ est fortement normalisant sur l'interprétation par φ des termes bien typés.

Preuve : C'est un corollaire du lemme précédent.

7.7 Cohérence

On montre que `False` n'admet pas de preuve dans l'environnement vide. On utilise la traduction $\langle\rangle$ vers $\mathcal{CCI}$ et le fait qu'elle préserve le typage pour dériver trivialement une contradiction de l'existence d'une telle preuve, sachant que $\mathcal{CCI}$ est cohérent.

Lemme 7.7.1 Il n'existe pas de terme $H_\perp$ en forme normale tel que $\emptyset \vdash_{\mathcal{CCI}^{nf}} H_\perp : \texttt{False}$.

Preuve : Par l'absurde, supposons qu'une telle preuve $H_\perp$ existe, alors $\langle H_\perp\rangle$ est telle que $\emptyset \vdash_{\mathcal{CCI}} \langle H_\perp\rangle : \texttt{False}$. Ce qui est une contradiction puisque $\mathcal{CCI}$ est cohérent. Une telle preuve n'existe pas.

7.8 Compatibilité de la conversion par rapport à la réduction

Théorème 7.8.1 Pour tous termes t_1 et t_2 de $\mathcal{CCI}^{nf}$, si $t_1 \equiv_{nf+\mathcal{CCI}} t_2$ alors $t_1 \equiv_{\mathcal{CCI}^{nf}} t_2$.

Preuve : Conséquence directe du lemme 7.6.6.

7.9 Unicité du typage

Lemme 7.9.1 (Unicité du type) Si les deux jugements $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$ et $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T'$ sont dérivables, alors $T \equiv_{\mathcal{CCI}^{nf}} T'$.

Preuve : Par induction sur t. À chaque fois on utilise le cas correspondant du lemme d'inversion.

- $t = \mathsf{Elim}(I, Q, \vec{a}, u)\{\overrightarrow{f_i}\}$ Par le cas xi du lemme d'inversion, on obtient que $T \equiv_{\mathcal{CCI}^{nf}} (Q\ \vec{a}\ u)$ et $T' \equiv_{\mathcal{CCI}^{nf}} (Q\ \vec{a}\ u)$ donc $T \equiv_{\mathcal{CCI}^{nf}} T'$ par transitivité de $\equiv_{\mathcal{CCI}^{nf}}$.

– $t = \mathsf{Norm}(A, nf)$ même raisonnement avec le cas xii du lemme d'inversion :
$$T \equiv_{\mathcal{CCI}^{nf}} T' \equiv_{\mathcal{CCI}^{nf}} \mathsf{Set}$$
– $t = \mathsf{Class}(A, nf, u)$ même raisonnement avec le cas xiii du lemme d'inversion :
$$T \equiv_{\mathcal{CCI}^{nf}} T' \equiv_{\mathcal{CCI}^{nf}} \mathsf{Norm}(A, nf, u)$$
– t=Elimnorm(A,nf,f,u) Même raisonnement avec les cas xiv :
$$T \equiv_{\mathcal{CCI}^{nf}} T' \equiv_{\mathcal{CCI}^{nf}} (P\ u)$$
où P est déterminé par le type de f, donné au même cas du lemme d'inversion.

7.10 Décidabilité du typage

Théorème 7.10.1 (décidabilité de la conversion) Il existe un algorithme qui, si les deux jugements $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : T$ et $\Gamma \vdash_{\mathcal{CCI}^{nf}} t' : T$ sont dérivables,
— retourne Vrai si $t \equiv_{\mathcal{CCI}^{nf}} t'$
— retourne Faux sinon

Preuve : Par définition, l'algorithme doit décider si $\varphi(t) \equiv_{nf'+\mathcal{CCI}} \varphi(t')$. Or d'après l'unicité des formes normales, $\varphi(t) \equiv_{nf'+\mathcal{CCI}} \varphi(t')$ si et seulement si les formes normales de $\varphi(t)$ et $\varphi(t')$ sont égales.

Il suffit donc de réduire $\varphi(t)$ et $\varphi(t')$, et de comparer les formes normales. L'algorithme termine car la réduction termine sur les termes bien typés (théorème 7.6.13).

La décidabilité du typage de $\mathcal{CCI}^{nf}$ est une conséquence de la décidabilité du typage de $\mathcal{CCI}$, et de la décidabilité de la conversion.

Théorème 7.10.2 (décidabilité typage) Il existe un algorithme $\mathcal{A}_{\mathcal{CCI}^{nf}}$ qui, étant donnés un terme t et un environnement de typage Γ,
— échoue s'il n'y a pas de terme U tel que $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : U$,
— retourne un tel terme sinon.

Preuve : On raisonne par récurrence sur la somme des tailles de Γ et U. Dans chaque cas on utilise la clause correspondante du lemme d'inversion 7.3.10.

Corollaire 7.10.3 Il existe un algorithme qui étant donnés deux termes t et U, et un environnement de typage Γ
— retourne vrai si $\Gamma \vdash_{\mathcal{CCI}^{nf}} t : U$
— retourne faux sinon

Preuve : Par l'unicité du typage, la décidabilité du typage et la décidabilité de la conversion. Il suffit d'appliquer l'algorithme de typage $\mathcal{A}_{\mathcal{CCI}^{nf}}$ sur l'entrée (Γ, t) et
— si $\mathcal{A}_{\mathcal{CCI}^{nf}}$ échoue retourner faux,
— sinon
  — si le terme U' retourné par $\mathcal{A}_{\mathcal{CCI}^{nf}}$ est tel que $U \equiv_{\mathcal{CCI}^{nf}} U'$, retourner vrai
  — sinon retourner faux

7.11 Conclusion

En conclusion, on constate que l'extension consistant à ajouter les types normalisés au calcul des constructions inductives ne pose pas de problèmes, les propriétés importantes sont préservées : normalisation forte, décidabilité du typage et subject reduction. En fait cette extension semble être possible dans n'importe quel calcul possédant des types inductifs. Il semble également que l'ajout de la η-réduction ne pose pas de problème majeur :

— la preuve de normalisation ne change pas beaucoup,

— la preuve de confluence est un peu plus compliquée puisque la réduction n'est confluente que sur les termes typables. Cependant la méthode donnée dans [Geu92] ou [Wer94], consistant à prouver la confluence *après* la normalisation, devrait marcher sans problème.

7.12 Exemples de types normalisés

Nous allons maintenant donner quelques exemples de types normalisés. Nous les donnerons dans une syntaxe Coq [1].

7.12.1 Les entiers relatifs

On commence par définir les entiers relatifs comme quotient des paires d'entiers naturels.

Pour faire un type normalisé, il faut une fonction de normalisation. On choisit de normaliser les nombres positifs vers les couples de la forme $(0, n)$ et les nombres négatifs vers $(n, 0)$.

Type support

```
Inductive nat  : Set :=
   O : nat
 | S : nat->nat.
```

Fonction de normalisation

```
Definition N2:= nat*nat.

Fixpoint nfN2_aux [n,m:nat]: N2 :=
  Cases n of
  | O => (n,m)
  | (S n') =>
      Cases m of
       | O => (n,m)
       | (S m') => (nfZ_aux n' m')
      end
  end.


Definition nfN2 :=[x:N2]
```

1. même si l'implantation des types normalisés n'est pas encore faite.

```
 let (n,m)=x in (nfN2_aux n m).


Eval Compute in (nfN2 ((S (S (S O))), (S (S O)))).
-->= ((S O),O)
Eval Compute in (nfN2 ((S (S (S O))), (S (S (S (S O)))))).
-->= (O,(S O))
```

Déclaration du type normalisé

À partir d'ici on suppose que les types normalisés ont été implantés dans Coq.

```
Definition Z1:Set := Norm(N2 , nfN2).

Check class(N2, nfN2, (O,O)).
--> Norm(N2 , nfN2)
```

Les principes d'élimination du type normalisé peuvent être générés automatiquement :

```
Print Z1_rec.
[P:Z1 -> Prop] [H:(x:N2)(P class(N2,nfN2,x))][t:Z1](elimnorm (N2, nfN2, H, t))
 :(P:Z1 -> Prop) (H:(x:N2)(P class(N2,nfN2,x)))(t:Z1) (P (Idnorm t))
```

Une preuve sur les types normalisés

On illustre ci-dessous l'utilisation de la conversion sur les types normalisés.

```
Goal class(N2, nfN2, ((S (S O)), (S (S O)))) = class(N2, nfN2, (O, O)).
Apply refl_equal.
Save.
```

Définition de fonctions sur un type normalisé

Pour définir une fonction sur un type normalisé, il faut d'abord en définir une sur son type support, puis la faire passer au type normalisé :

```
Definition SN2: N2 -> Z1 :=
 [x:N2] let (a,b)=x in class(N2, nfN2, ((S a),b)).

Definition SZ: Z1 -> Z1 :=
 [x:Z1] elimnorm(N2, nfN2, SN2, x).
```

L'évaluation d'une telle fonction utilise la *nf*-réduction :

```
Eval Compute in class(N2,nfN2,(SZ ((S O), (S (S O))))).
 -->class(N2,nfN2,((S O), (S O)))
```

Précisons que pour donner ce résultat, deux calculs ont été exécutés, celui de la forme normale de l'argument, puis l'évaluation de la fonction initiale :

```
(SZ class(N2,nfN2,((S O), (S (S O))))) ⟶* (SN2 (nfN2 ((S O), (S (S O)))))
                                       ⟶* (SN2 (O, (S O)))
                                       ⟶* ((S O), (S O))
```

Définition de relation sur un type normalisé

```
Inductive leZ: Z1 -> Z1 -> Prop:=
  | leOZ: (leZ class(N2,nfN2, (O, O)) class(N2,nfN2, (O, O)))
  | leSZ: (x,y:Z1) (leZ x y)  -> (le x (SZ y)).
```

7.12.2 Les entiers relatifs, une autre formalisation

On peut également définir les entiers relatifs comme la normalisation du type inductif à trois constructeurs : `O`, `S` et `P`. La fonction de normalisation éliminant les combinaisons `(S (P ...))`.

Type support

```
Inductive int  : Set :=
   OI : int
 | SI : int->int
 | PI : int->int.
```

Fonction de normalisation

Une première version de la fonction de normalisation est la suivante :

```
Fixpoint nfZ1 [n:int]: int :=
  Cases n of
   OI => OI
  |(SI x) => Cases (nfZ1 x) of
               OI     => (SI OI)
             |(SI x') => (SI (SI x'))
             |(PI x') => x'
            end
  |(PI x) => Cases (nfZ1 x) of
               OI     => (PI OI)
             |(SI x') => x'
             |(PI x') => (PI (PI x'))
            end
  end
```

Mais cette fonction utilise une stratégie "en profondeur d'abord", ce qui n'est pas adapté aux arguments non clos, comme on le voit dans l'exemple suivant, où on aimerait obtenir le résultat `(nfZ1 x)` :

```
Eval Compute in (nfZ1 (SI (PI x))).
 --> Cases (Cases (nfZ1 x) of ...)
```

On prend donc une définition moins simple pour la fonction de normalisation, qui utilise une stratégie "en tête d'abord" :

```
Fixpoint nfZ [n:int]: int :=
  Cases n of
      OI    => OI
     | (SI a) => Cases a of
                  OI  => (SI OI)
                 |(SI y) => Cases (nfZ a) of
                              OI  => (SI OI)
                            |(SI x) => (SI (SI x))
                            |(PI x) => x
                            end
                 |(PI y) => (nfZ y)
                 end
     | (PI a) => Cases a of
                    OI  => (PI OI)
                  |(SI y) => (nfZ y)
                  |(PI y) => (Cases (nfZ a) of
                                  OI  => (PI OI)
                                |(SI x) => x
                                |(PI x) => (PI (PI x))
                                end)
                   end
  end.


Eval Compute in (nfZ (SI (PI (SI OI)))).
-->= (SI OI)
```

Déclaration du type normalisé

À partir d'ici on suppose à nouveau que les types normalisés ont été implantés dans Coq.

```
Definition Z:Set := Norm(int , nfZ).

Check class(int, nfZ, OI).
--> Norm(int , nfZ)

Print Z_rec.
[P:Z -> Prop] [H:(x:int)(P class(int,nfZ,x))][t:Z](elimnorm (int, nfZ, H, t))
 :(P:Z -> Prop) (H:(x:int)(P class(int,nfZ,x)))(t:Z) (P (Idnorm t))
```

Une preuve sur les types normalisés

On peut toujours utiliser la conversion sur les types normalisés :

```
Goal class(int, nfZ, (SI (PI O))) = class(N2, nfZ, O).
Apply refl_equal.
Save.
```

Définition de fonctions sur un type normalisé

On veut définir la fonction de `Z` dans `nat` correspondant à la valeur absolue. On commence par définir une fonction sur le type support :

```
Definition absint: int -> nat :=
 [x:int]
  Cases x of
     OI => O
   | (SI x) => (S (abs x))
   | (PI x) => (S (abs x))
  end.

Definition absZ: Z -> nat :=
 [x:Z] elimnorm(int, nfZ, absint, x).


Eval Compute in (absZ class(int,nfZ,(SI (PI OI)))).
 --> O
```

Précisons que pour donner ce résultat, deux calculs ont été exécutés, celui de la forme normale de l'argument, puis l'évaluation de la fonction initiale :

$$\begin{array}{lll} \texttt{(absZ class(int,nfZ,(SI (PI OI))))} & \longrightarrow_{nf} & \texttt{(absZ (nfZ (SI (PI OI))))} \\ & \longrightarrow^{*} & \texttt{(absZ OI)} \\ & \longrightarrow^{*} & \texttt{O} \end{array}$$

On voit que la fonction `absint` n'est pas compatible avec la relation engendrée par `nfZ`. On peut néanmoins la faire passer au type normalisé puisque la *nf*-réduction sera appliquée à l'argument avant que celui-ci ne soit appliqué à `absint`. On illustre ce principe par un dernier exemple d'application de la fonction `absZ` :

```
Eval Compute in (absZ class(int,nfZ,(SI (PI (PI OI))))).
 --> (S O)

Fixpoint plusint'[n:int]:int -> int :=
[m:int]
Cases n of
 | OI => m
  | (SI n') => (SI (plusint' n' m))
 | (PI n') => (PI (plusint' n' m))
end.

Definition plusint : int  -> int  -> Z :=
 [n,m:int] class(int, nfZ, (plusint n m)).
```

```
Definition plusZ : Z  -> Z -> Z :=
[n,m:Z]
 let plusZ1= [p:int] elimnorm(int, nfZ, (plusint p), m) in
 elimnorm(int, nfZ, plusZ1, n).


Eval Compute in (plusZ class(..., class(...OI)
                       class(..., class((SI OI)))).
 --> (SI OI)
Eval Compute in (plusZ class(..., class((SI (PI OI))))
                       class(..., class((SI OI)))).
 --> (SI OI)
Eval Compute in (plusZ class(..., class((PI OI)))
                       class(..., class((SI OI)))).
 --> (PI (SI OI))

Fixpoint lebool [n:int]:int -> bool :=
[m:int]
Cases n m of
 | (SI n') (SI m') => (lebool n' m')
 | (PI n') (PI m') => (lebool n' m')
 | OI OI => true
 | OI (SI m') => true
 | (PI n') OI => true
 | _ _ => false
end.


Definition leboolZ : Z  -> Z -> bool :=
[n,m:Z]
 let leboolZ1= [p:int] elimnorm(int, nfZ, (lebool p), m) in
 elimnorm(int, nfZ, leboolZ1, n).
```

Définition de relation sur un type normalisé

```
Inductive leZ: Z -> Z -> Prop:=
  | leOZ: (leZ class(int,nfZ, OI) class(int,nfZ, OI))
  | leSZ: (x,y:int) (leZ class(int,nfZ,x) class(int,nfZ,y))
  -> (le class(int,nfZ,x) class(int,nfZ,(SI y))).
```

7.12.3 $\mathbb{Z}/3\mathbb{Z}$

On définit maintenant les quotients du type $\mathbb{Z}/3\mathbb{Z}$, à partir de la définition de $\mathbb{Z}$ ci-dessus. Le type support est donc `int`, et la fonction de normalisation est la fonction `mod3` définie ci-dessous :

```
Fixpoint mod3int [i:int] : int:=
```

```
 Cases i of
  | (PI OI) => (SI (SI OI))
  | (PI (PI OI)) => (SI OI)
  | (PI (PI (PI i'))) => (mod3int i')
  | (SI (SI (SI i'))) => (mod3int i')
  | i' => i'
 end.

Definition mod32 : int -> Z :=
 [i:int]class(int, nfZ, (mod3int i)).

Definition mod3 : Z -> Z :=
 [i:Z]elimnorm(int, nfZ, mod3Z, i).

Eval Compute in (mod3 class(...,(SI (PI OI)))) --> class(...OI)
Eval Compute in (mod3 class(..;(PI (SI (PI OI))))) --> class(SI (SI OI))
```

On définit le type normalisé `Z3Z` :

```
Definition Z3Z: Set := Norm(Z , mod3).

Check class(Z,mod3,class(N2,nfZ,OI))
  : Norm(Z , mod3)
```

Un exemple intéressant est le suivant : on veut une fonction qui rend `true` si son argument est 0 modulo 3, et `false` sinon. Pour cela on écrit la fonction suivante, sur `int` :

```
Fixpoint isOint: int -> bool :=
[n:int]Cases n of
|  OI => true
| _ => false
end.
```

Il est clair que cette fonction n'est pas compatible avec la relation du quotient $\mathbb{Z}/3\mathbb{Z}$. On la passe cependant à $\mathbb{Z}$, puis à $\mathbb{Z}/3\mathbb{Z}$ grâce à `Elimnorm` :

```
Definition isOZ: Z -> bool := [n:Z] Elimnorm(int,nfZ,isOint,n).

Definition isOZ3Z: Z3Z -> Z3Z -> bool :=
[n:Z3Z]Elimnorm(Z,mod3int,isOZ,n).
```

La fonction ainsi définie se comporte comme on l'attend, c'est-à-dire que le choix du représentant ne change pas le résultat :

```
(* Cas 0 *)
Eval Compute in (isOZ3Z Class(Z,mod3int,(Class(int,nfZ,OI)))).
-> true
```

```
Eval Compute in
 (isOZ3Z Class(Z,mod3int,(Class(int,nfZ,(SI (SI (SI OI)))))))).
-> true

Eval Compute in
 (isOZ3Z Class(Z,mod3int,(Class(int,nfZ,(PI (PI (PI OI)))))))).
-> true

(* Cas 1 *)
Eval Compute in (isOZ3Z Class(Z,mod3int,(Class(int,nfZ,(SI OI))))).
-> false

Eval Compute in (isOZ3Z Class(Z,mod3int,(Class(int,nfZ,(PI (PI OI)))))).
-> false
```

Troisième partie

Démonstration d'auto-stabilisation dans le système Coq

Chapitre 8

Les algorithmes auto-stabilisants

Les algorithmes auto-stabilisants [Dij74, Shn93] sont des algorithmes répartis, c'est-à-dire s'exécutant sur un réseau d'ordinateurs. Nous allons d'abord préciser cette première notion avant de caractériser plus particulièrement les algorithmes auto-stabilisants.

8.1 Les algorithmes répartis

Un réseau d'ordinateurs est un graphe (une machine = un nœud du graphe). Chaque machine exécute un programme, qui envoie et reçoit des messages à travers le réseau, et réagit en fonction du contenu des messages reçus. On fait les remarques suivantes :

- L'état interne du programme s'exécutant sur un nœud détermine l'*état* de ce nœud.
- L'ensemble des états des nœuds du réseau représente la *configuration du réseau*.
- Un nœud n'est en général pas relié à toutes les autres machines du réseau. Au niveau de détail auquel nous nous plaçons, on considère qu'un nœud ne communique qu'avec ses voisins immédiats dans le graphe. Dans le modèle le plus couramment utilisé, cette communication ne se fait qu'à travers une seule sorte de message : l'envoi par un nœud à ses voisins immédiats de son propre état.
- Avec ce modèle – que nous utilisons dans la suite – on peut représenter le déroulement d'un algorithme réparti en considérant que le réseau passe d'une configuration à une autre grâce à un système de transition non déterministe. Une transition correspondant au changement d'état d'un nœud du réseau en fonction de celui de ses voisins immédiats.

8.2 Auto-stabilisation

Les algorithmes auto-stabilisants sont des algorithmes répartis ayant la particularité suivante : en partant de n'importe quelle configuration initiale, l'algorithme fait revenir le réseau dans une configuration correcte (*légitime*) au bout d'un temps fini. L'ensemble des configurations légitimes, déterminé par ailleurs, est l'ensemble des configurations correspondant à un déroulement "autorisé" de l'algorithme. Par exemple pour un algorithme de passage de jeton dans un anneau, les configurations légitimes sont celles dans lesquelles il y a un et un seul jeton sur le réseau.

En toute généralité, on définit l'auto-stabilisation de la manière suivante :

Définition 8.2.1 (Auto-stabilisation 1) Soit A un algorithme réparti, s'exécutant sur un réseau Δ, dont $\mathcal{L}$ est l'ensemble des configurations légitimes. On dira que A est auto-stabilisant pour $\mathcal{L}$ si :

Toute exécution de A à partir de n'importe quelle configuration initiale aboutit au bout d'un temps fini dans $\mathcal{L}$, et n'en sort plus.

En général on utilise une définition moins générale mais plus précise de l'auto-stabilisation :

Définition 8.2.2 (Auto-stabilisation 2) Soit A un algorithme réparti, s'exécutant sur un réseau Δ, dont $\mathcal{L}$ est l'ensemble des configurations légitimes. On dira que A est auto-stabilisant pour $\mathcal{L}$ si :

1. Toute exécution de A à partir de n'importe quelle configuration initiale aboutit au bout d'un temps fini dans $\mathcal{L}$ (*convergence*),
2. Toute transition de A partant d'un élément de $\mathcal{L}$ aboutit dans $\mathcal{L}$ (*stabilité*).

D'un point de vue pratique, cette définition est en fait très semblable à la précédente car un algorithme auto-stabilisant pour $\mathcal{L}$ au sens de la première définition est auto-stabilisant pour un sous-ensemble de $\mathcal{L}$ au sens de la deuxième.

L'intérêt de l'auto-stabilisation réside bien entendu dans la *tolérance aux pannes* et aux *fautes* : si une machine du réseau, pour une raison quelconque, a (ponctuellement) un comportement imprévu qui brise le déroulement correct de l'algorithme, alors au bout d'un temps fini le déroulement correct pourra reprendre, sans intervention extérieure.

Signalons que le point 1 de la définition ci-dessus implique qu'une exécution de l'algorithme ne peut pas s'arrêter en dehors de $\mathcal{L}$(*no-deadlock*).

8.3 En Coq

On représente en Coq l'algorithme réparti par une relation sur les configurations, représentant les transitions effectuées par l'algorithme. Celui-ci étant non déterministe, il n'est en effet pas possible de définir une fonction correspondante.

On commence donc par quelques définitions sur les relations, permettant de les considérer comme des systèmes de transitions.

8.3.1 Sur les relations

On commence par déclarer le type `A` des configurations. Grâce au mécanisme des sections, ce type sera un paramètre de toutes nos définitions et sera instancié pour chaque exemple traité. Un algorithme réparti sera donc de type `A -> A -> Prop`. Puis on déclare un certain nombre de notions utiles sur les relations ainsi que des facilités d'écriture pour ces notions. Cette première partie (jusqu'à la section 8.3.2) ne présente pas de nouveauté, en fait la plupart de ces notions sont déjà présentes dans la bibliothèque standard de Coq.

```
Section extension_rels.

Variable A: Set.
```

On définit l'union de deux relations (\/ représente le *ou* sur Prop) :

```
Definition explicit_union :=
        [Q,R:A -> A -> Prop][x,y:A] (Q x y)\/(R x y).
```

La clôture transitive d'une relation R est une relation inductive :

```
Inductive explicit_rel_plus [R:A->A->Prop]:A->A->Prop:=
   relplus_1step: (x,y:A)(R x y)->(explicit_rel_plus R x y)
 | relplus_trans1: (x,y,z:A)(R x y)->(explicit_rel_plus R y z)
                                ->(explicit_rel_plus R x z).
```

On définit ensuite les notions de réductibilité et irréductibilité d'un terme t par une relation R (considérée comme un système de réduction). (explicit_irr R t) sera prouvable s'il n'existe pas de successeur par R de t. (explicit_red R t) sera prouvable s'il en existe un.

```
Definition explicit_irr :=
         [R: A -> A -> Prop][x:A]~(EX y|(R x y)).

Definition explicit_red :=
         [R: A -> A -> Prop][x:A](EX y|(R x y)).
```

Puis on clôt cette section :

```
End extension_rels.
```

On définit ensuite quelques facilités syntaxiques permettant d'utiliser les notations de manière intuitive. Par exemple pour exprimer explicit_union A R1 R2, on notera (union R1 R2). De même on utilisera les notations (rel_plus R), (star R), (irr R t) et (red R t). On ne donne pas ici les détails de ces déclarations.

```
Syntactic Definition union := (explicit_union ?).
...
```

Puis on démontre quelques propriétés sur les relations. Voici par exemple la démonstration de transitivité de la clôture transitive d'une relation R sur un type T :

```
Lemma star_star: (T:Set)(R:T -> T -> Prop)(x,y,z:T)
 (star R x y) -> (star R y z)  -> (star R x z).
Intros T R x y z starxy.
Elim starxy.
Intros.
Assumption.
Intros.
EApply star_trans1 with y0.
Assumption.
Apply (H1 H2).
Save.
```

8.3.2 Définition de l'auto-stabilisation

On ouvre une nouvelle section en déclarant (comme variables temporaires) à nouveau le type des configurations, puis la relation `R` représentant l'algorithme, puis le prédicat `L` représentant les configurations légitimes. Tous seront donc des paramètres des prochaines définitions :

```
Section selfstab.

(* Type des configurations *)
Variable str:Set.

(* L'algorithme *)
Variable R: str -> str -> Prop.

(* Configurations légitimes *)
Variable L: str  -> Prop.
```

On définit ensuite les différentes propriétés nécessaires à l'auto-stabilisation : la stabilité et la convergence. La stabilité ne pose pas de problème :

```
(* Clôture de L par R: on ne sort plus des états légitimes une fois
qu'on est dedans *)
Definition closure:= (u,v:str) (R u v) -> (L u) -> (L v).
```

Pour définir la convergence, on doit définir une variante de la notion d'*accessibilité* (voir section 5.4.4) ; nous donnons tout d'abord une reformulation de cette dernière [1] :

```
Inductive finite [R : A->A->Prop] : A->Prop :=
  finite_intro : (x:A)((y:A)(R x y)->(finite R y))->(finite R x).
```

On appellera la propriété de convergence l'*inévitabilité*. On dira que `L` est inévitable pour un terme `t` si toutes les réductions partant de `t` par `R` atteignent `L`. On remarque que cette notion correspond à la propriété d'un ensemble d'être le *bar* d'un élément pour une réduction.

Pour cela on utilise la propriété inductive suivante :

```
Inductive inevitable : str -> Prop :=
   (* Si x est dans L, alors L est inévitable à partir de x *)
 | L_inevitable:(x:str) (L x) -> (inevitable x)
   (* Si L est inévitable pour tous les successeurs de x, alors
      L est inévitable pour x*)
 | all_inevitable:(x:str) ((y:str) (R x y) -> (inevitable y))
                          -> (inevitable x).
```

On peut maintenant définir l'auto-stabilisation de `R` :

```
Definition self_stab:= closure /\ (w:str) (inevitable w).
```

1. on remarque que l'ordre des arguments de la relation est inversé par rapport à la section 5.4.4. Ceci afin de correspondre davantage à l'écriture habituelle : `(R a b)` représente `a`$\longrightarrow$`b`

On peut donner une autre formulation dans le cas où le nombre de configurations est fini : `L` est inévitable si et seulement si il n'y a pas de cycle en dehors de `L`.

On commence par définir la propriété correspondant à l'absence de cycle en dehors de `L` :

```
(* Pas de cycle commençant par une configuration non légitime *)
Definition no_cycle:= (w:str) (Rplus w w) -> (L w).

(*trivialement équivalent a: *)
Lemma no_cycle2: (w:str) no_cycle -> ~(L w) -> ~(Rplus w w).
Auto.
Save.
```

Et l'autre formulation de l'auto-stabilisation, équivalente à la première seulement si le nombre de configurations est fini (en toute rigueur, il faut faire la preuve dans Coq de cette équivalence) est donc :

```
Definition self_stab2:=   closure /\ no_cycle.
```

Signalons une dernière formulation de la convergence : une relation R converge vers L si sa restriction aux éléments en dehors de L termine.

On commence par définir la restriction :

```
Definition restrict_R:=[x,y:str] ~(L y) /\ (R x y).
```

Puis on donne la définition de l'auto-stabilisation :

```
Definition self_stab3:=   closure /\ (w:str)(finite restrict_R x).

End selfstab.
```

Chapitre 9

Une technique de preuve d'auto-stabilisation sur les réseaux linéaires

Nous allons maintenant implanter dans Coq la méthode de preuve d'auto-stabilisation de [JBM01]. Nous donnerons plus loin les hypothèses précises d'application de cette méthode, mais précisons déjà qu'elle s'applique sur des réseaux linéaires, c'est-à-dire dont chaque nœud a au plus deux voisins ; on obtient donc une topologie en ligne ou en anneau.

9.1 Les mots : une modélisation intéressante

Les intérêts de cette topologie particulière sont les suivants :

— Les configurations peuvent être représentées par des mots sur l'alphabet des états des nœuds du réseau. Dans le cas d'un anneau, on distinguera un nœud particulier pour déterminer le début du mot.

— L'algorithme peut être modélisé par un système de réécriture sur les mots représentant les configurations.

— Comme un nœud change d'état en fonction de ses deux voisins, les membres gauches des règles du système de réécriture seront de taille inférieure ou égale à 3. Comme d'autre part la topologie du réseau ne change pas au cours de l'exécution de l'algorithme, la taille des mots n'est pas modifiée par le système de réécriture. On aura donc des règles de la forme :

$$\begin{array}{rl} & abc \longrightarrow ab'c \\ ou : & ab \longrightarrow ab' \\ ou : & ab \longrightarrow a'b \end{array}$$

9.2 Exemple : l'algorithme de Ghosh

L'algorithme [Gho93] de Ghosh est un algorithme d'exclusion mutuelle à 4 états. Le système considéré est un réseau linéaire à N machines, numérotées de 0 à $N-1$. Chaque nœud du réseau a quatre états possibles : $\{0,1,2,3\}$, sauf le nœud 0 (la machine *bottom*) qui ne peut avoir que les deux états $\{1,3\}$ et le nœud $N-1$ (la machine *top*) qui ne peut avoir que les deux états $\{0,2\}$.

Remarque 9.2.1 *L'algorithme de Ghosh, ainsi que les autres algorithmes que nous considérons dans la suite, distingue l'un des nœuds du réseau afin de lui appliquer des règles spécifiques. La lettre correspondant à ce nœud particulier sera en général placée à la fin du mot représentant les configurations.*

On représente donc les configurations du réseau par des mots de taille $N+2$ car on signale le début et la fin des mots par la lettre #. Les mots #000#, #101010# représentent donc respectivement des configurations de réseaux à 3 et 6 machines.

L'ensemble des configurations légitimes $\mathcal{L}$ est défini par l'expression régulière suivante :

$$\mathcal{L} = \#\{1,3\}^+\{0,2\}^+\#$$

L'algorithme est représenté par un système de réécriture sur les configurations :

$$\begin{array}{lrcll}
M_1: & \#X(q+1)qY\# & \longrightarrow & \#X(q+1)(q+1)Y\# & \text{si } Y \neq \varepsilon \\
M_2: & \#Xq(q+1)Y\# & \longrightarrow & \#X(q+1)(q+1)Y\# & \text{si } X \neq \varepsilon \\
T_1: & \#X32\# & \longrightarrow & \#X30\# & \\
T_1: & \#X10\# & \longrightarrow & \#X12\# & \\
B_1: & \#12Y\# & \longrightarrow & \#32Y\# & \\
B_1: & \#30Y\# & \longrightarrow & \#10Y\# &
\end{array}$$

où $+$ est l'addition modulo 4 et $q \in \{0,1,2,3\}$.

On constate qu'il y a trois types de règle :

— les règles *Middle* (M_i) qui modifient les lettres du mot qui ne sont pas voisines d'un #,

— les règles *Bottom* (B_i) qui modifient la première lettre du mot,

— les règles *Top* (T_i) qui modifient la dernière lettre du mot.

Nous nous intéressons dans la suite aux algorithmes ayant la même présentation – trois ensembles de règles – et nous allons utiliser les techniques de preuve d'auto-stabilisation pour ces algorithmes présentées par Beauquier, Bérard, Fribourg et Magniette dans [JBM01]. Nous préciserons au fur et à mesure du développement les différentes hypothèses nécessaires à ces techniques.

9.3 Représentation en Coq d'un algorithme réparti

9.3.1 Les configurations

Dans le chapitre précédent, on avait laissé en paramètre le type des configurations pour définir l'auto-stabilisation de manière générique. Nous allons maintenant définir une instance particulière du type des configurations : une configuration sera représentée par un mot sur un alphabet.

On représente les mots par des listes sur un alphabet en paramètre, en utilisant le module `PolyList`, une modélisation des listes en Coq. On définit ensuite les ensembles de mots sous forme de prédicats sur les mots ; cette partie est inspirée du développement en Coq de la théorie des langages par J. Courant et J.C. Filliâtre, mais pour un fragment très simple de cette théorie (l'utilisation directe de ce développement s'avère inutilement lourde). On peut ainsi définir les opérations de base sur les expressions régulières : puissance, concaténation, clôture transitive, etc.

```
Require Export PolyList.
(* Rappel:
Inductive list [A : Set]  : Set :=
      nil : (list A) | cons : A->(list A)->(list A)
*)
```

On commence une nouvelle section, dans laquelle on déclare (temporairement, voir section 5.6) le type de l'alphabet :

```
Section Ensf_sec.

Variable ens_symb:Set.
```

Ensuite on définit les mots comme les listes sur l'alphabet :

```
Definition Word := (list ens_symb).
Syntactic Definition epsilon := (nil ens_symb).
```

On définit ensuite le type des ensembles de mots :

```
Definition wordset := Word -> Prop.
```

La première opération sur les ensembles est la construction de l'ensemble contenant un mot donné :

```
Definition lword : Word->wordset :=   [w, w1 : Word] w = w1.
```

Signalons pour faciliter la lecture que pour un mot `w`, `(lword w)` est de type `Word -> Prop`, c'est-à-dire de type `wordset`.

Puis la définition de l'ensemble des mots construits :

— par concaténation de mots de deux ensembles : `lconc` [1],
— par mise à la puissance de mots d'un ensemble : `lpuiss`,
— par clôture transitive et réflexive : `lstar`,
— par clôture transitive : `lplus`.

```
Definition lconc : wordset->wordset->wordset :=
  [l1,l2 : wordset] [w : Word]
       ((EX w1:Word |
         ((EX w2:Word|
              (l1 w1)
           /\ (l2 w2)
           /\  w=(app w1 w2) )))).

Fixpoint lpuiss [n:nat] : wordset->wordset :=
  [l:wordset]Cases n of
  | O => (lword epsilon)
  | (S p) => (lconc l (lpuiss p l))
  end.

Definition lstar : wordset->wordset :=
  [l:wordset][w:Word]
    ((EX n:nat |(lpuiss n l w))).

Definition lplus : wordset->wordset :=
  [l:wordset][w:Word]
    ((EX n:nat |(lpuiss (S n) l w))).
```

1. La fonction app effectue la concaténation de deux mots.

On démontre ensuite quelques résultats sur ces opérations ; à titre d'exemple, on montre ici la preuve d'un principe d'induction sur les ensembles de la forme `(lstar l)`.

```
Lemma induction_star : (P:Word->Prop)(l:wordset)
 ((n:nat)(w:Word) (lpuiss n l w)->(P w) ) -> ( (w:Word)(lstar l w)->(P w) ).
Intros P l hyp1 w hyp2.
Elim hyp2; Clear hyp2.
Intros x hyp2.
Apply (hyp1 x w); Auto.
Save.
```

Puis on ferme la section.

```
End Ensf_sec.
```

9.3.2 Les réductions

On a vu (8.3.1) que les algorithmes répartis seraient représentés par des relations sur les mots, correspondant à des systèmes de réécriture sur les mots.

Un système de réécriture sur les mots se présente sous la forme d'un ensemble de règles de la forme $l_i \longrightarrow r_i$, où l_i et r_i sont des mots. On dira qu'un mot w se réécrit en w' en une étape si $w = w_1 l_k w_2$ et $w' = w_1 r_k w_2$ pour un k donné. On va donc représenter un système de réécriture comme la clôture par contexte d'une relation représentant les règles $l_i \longrightarrow r_i$.

On a vu que les règles des algorithmes que nous considérons sont réparties en trois ensembles de règles T, M et B, correspondant chacun à une extension au contexte différente :

— M est étendu à droite et à gauche,

— T est étendu seulement à gauche car ses règles ne s'appliquent qu'à la fin du mot,

— B est étendu seulement à droite car ses règles ne s'appliquent qu'au début du mot.

On formalise cela de la manière suivante :

```
Section reduct.
Variable A:Set.
Syntactic Definition epsilon := (nil A).
```

On déclare les trois relations correspondant aux trois ensembles de règles :

```
Variables T,M,B: A -> A -> Prop.
```

Puis on définit la relation de réécriture associée, clôture à gauche pour `T`, à droite pour `B`, et à gauche et à droite pour `M`.

```
Inductive TBM_R: A -> A -> Prop :=
 | T_T: (u,v,w:A) (T u v) -> (TBM_R (w^u) (w^v))
 | B_B: (u,v,w:A) (B u v) -> (TBM_R (u^w) (v^w))
 | M_M: (u,v,w,w':A) ~(w=epsilon) -> ~(w'=epsilon)
                   -> (M u v) -> (TBM_R (w^(u^w')) (w^(v^w'))).
End reduct.
```

On voit ici que pour M, les contextes droite et gauche ne doivent pas être vides. On peut toujours respecter cette contrainte, même si elle n'apparaît pas dans l'algorithme. Il suffit pour cela d'énumérer les trois cas, contexte vide à gauche, contexte vide à droite, contexte non vide à droite et à gauche.

Une des hypothèses admises sur le système de réécriture représentant un algorithme réparti est qu'il ne change pas la topologie du réseau. `T`, `B` et `M` doivent donc préserver la taille des mots. On définit donc la longueur d'un mot ainsi que la propriété `length_preserve`. On définit également la propriété permettant de caractériser deux relations s'appliquant sur des termes de même taille.

```
Fixpoint norm [v:A]: nat :=
  Cases v of
   | nil => O
   | (cons a s) => (S (norm s))
  end.

Inductive same_length: A -> A -> Prop:=
   | empty_empty: (same_length epsilon epsilon)
   | cons_length: (x,y:lettre) (u,v:A) (same_length u v)
              -> (same_length (cons x u) (cons y v)).

Definition length_preserve :=
   [R:A -> A -> Prop] (u,v:A) (R u v)
         -> (same_length u v).

Definition apply_same_length :=
   [R,Q:A -> A -> Prop] (u,v,w,x:A)
        (R u v) -> (Q w x) -> (same_length u w).
```

9.4 Les hypothèses

Nous énonçons maintenant les hypothèses sur l'algorithme A permettant d'appliquer notre méthode de preuve d'auto-stabilisation ; nous verrons plus loin leur traduction en Coq :

1. A se présente sous la forme des trois ensembles de règles expliqués plus haut, notamment ces trois systèmes préservent la longueur des mots ;
2. Comme pour l'algorithme de Ghosh, le nombre d'états possibles pour chaque nœud du réseau est fini, donc le nombre de configurations possibles est fini ;
3. A est stable pour $\mathcal{L}$;
4. A ne s'arrête pas en dehors de $\mathcal{L}$;
5. *A privé des règles de T termine.*

La méthode de preuve décrite dans [JBM01] consiste en deux étapes. L'une est faite une fois pour toutes (c'est la partie générique de la preuve) et l'autre est à appliquer pour chaque nouvelle preuve d'algorithme (c'est la partie spécifique).

9.5 La partie générique de la preuve

Dans cette partie, il est prouvé que la convergence d'une certaine classe de réductions suffit à démontrer la convergence de toutes les réductions. Nous allons préciser maintenant cette classe.

9.5.1 Les mots activés

On introduit tout d'abord la notion de partie *active* et *inactive* d'un mot. On signalera une lettre active en la soulignant, les lettre non soulignées étant inactives. Par exemple dans le mot $ab\underline{cd}ef$, les positions 1,2,5 et 6 sont inactives, et les positions 3 et 4 sont actives. Plus formellement :

Définition 9.5.1 (Alphabet activé) Soit E un alphabet, on définit l'alphabet $\underline{E}$ contenant les mêmes lettres que E mais soulignées.

Définition 9.5.2 (Mot activé) Soit E un alphabet et $\underline{E}$ sa version activée, on appellera *mots activés* les mots définis sur $E \cup \underline{E}$.

Définition 9.5.3 (Mot actif) Un mot contenant au moins une lettre active est appelé actif.

Définition 9.5.4 (Mot inactif) Un mot ne contenant aucune lettre active est appelé inactif.

On définit également l'opération de *désactivation* retournant la version inactive d'un mot.

9.5.2 Représentation des mots activés dans Coq

On commence par définir les lettres actives et inactives à l'aide d'un type inductif à deux constructeurs (un pour les lettres actives, un pour les lettres inactives). On met ici le type de l'alphabet en paramètre, mais on instanciera ce paramètre par `nat` à la fin de cette section, afin d'avoir une représentation concrète des lettres.

```
Section act_inact.
Variable A:Set.

Inductive lettre: Set :=
 |act  : A -> lettre
 |inact: A -> lettre.
```

On définit ensuite quelques notations :
- `|+x+|` signifie `(act x)`,
- `|-x-|` signifie `(inact x)`,
- `x::l` correspond à `(cons x l)`,
- `l1^l2` correspond à la concaténation de mots.

On définit maintenant le type des mots activés : `stria`, l'opération de désactivation d'une lettre `destr_lettre`, ainsi que d'autres opérations de manipulation de lettres actives et inactives. Notamment `active_last` permet d'activer la dernière lettre d'un mot, ce qui permettra de construire les réductions actives.

```
Syntactic Definition stria :=(Word lettre).

Definition destr_lettre :lettre ->  A :=
 [l:lettre]Cases l of
 |(act x) => x
 |(inact x) => x
 end.

Definition inactive_lettre [n:lettre]:lettre := (inact (destr_lettre n)).

Definition active_lettre [n:lettre]:lettre := (act (destr_lettre n)).

Definition inactive_all_Elt  :=
  [l:lettre]
 Cases l of
 | (inact e) => l
 | (act e) => (inact e)
 end.

Definition active_all_Elt  :=
  [l:lettre]
 Cases l of
 | (inact e) => (act e)
 | (act f) => l
 end.

Fixpoint inactive_all_Word [w:(Word lettre)]: (Word lettre) :=
 Cases w of
   | nil => (nil lettre)
   | (cons x u) => (cons (inactive_all_Elt x) (inactive_all_Word u))
 end.


Fixpoint active_last [x:stria]: stria:=
 Cases x of
 |  nil => epsilon
 |  (cons  (inact n) epsilon) => (act n)::epsilon
 |  (cons  (act n) nil) => (act n)::epsilon
 |  (cons x w) => x::(active_last w)
 end.
```

On définit quelques prédicats sur les mots activés : être actif, inactif, etc.

```
Inductive active_word_strict: stria -> Prop:=
 | strac_act: (u:stria) (c:A) (active_word_strict (|+c+| :: u))
 | strac_ina: (u:stria) (c:lettre) (active_word_strict u)
```

```
                                    -> (active_word_strict (c::u)).

Inductive active_word: stria -> Prop:=
 | empty_act: (active_word epsilon)
 | strict_act:(w:stria) (active_word_strict w) -> (active_word w).

Inductive inact_word:stria -> Prop :=
 | empty_ina:(inact_word epsilon)
 | strin_ina:(u:stria) (c:A) (inact_word u) -> (inact_word (|-c-| :: u)).

Inductive pure_active:stria -> Prop :=
        | empty_pact:(pure_active epsilon)
        | strac_pact:(u:stria) (c:A) (pure_active u)
                            -> (pure_active (|+c|+ :: u)).
...
```

9.5.3 Les réductions actives

On définit l'extension aux mots activés d'un système de réécriture sur les mots.

Définition 9.5.5 Soit R un système de réécriture sur les mots formés sur l'alphabet A, on étend R aux mots activés de la manière suivante :
Soit u et u' deux mots sur A tels que u se réécrit en u' et P l'ensemble des positions du sous-mot de u réécrit. Soit w un mot activé tel que $u = desactive(w)$. Alors $w \longrightarrow_R w'$, où w' est défini comme suit :

- en ce qui concerne les lettre de w' : $desactive(w') = u'$;
- en ce qui concerne l'activation des lettre de w' :
  - si $active\_pos(w) \cap P = \emptyset$, alors $active\_pos(w') = active\_pos(w)$
  - sinon $active\_pos(w') = active\_pos(w) \cup P$. On dira alors que la partie active s'est *propagée*.

Par exemple, pour l'exemple de Ghosh, on a les réductions suivantes (le sous-mot réécrit est 30, qui se réécrit en 10) :

$$\begin{array}{lcl}
\#300\# & \longrightarrow & \#100\# \\
\#30\underline{0}\# & \longrightarrow & \#10\underline{0}\# \\
\#3\underline{0}0\# & \longrightarrow & \#\underline{1}0\underline{0}\# \\
\#3\underline{00}\# & \longrightarrow & \#\underline{100}\# \\
\#\underline{3}00\# & \longrightarrow & \#\underline{1}00\# \\
\#\underline{30}0\# & \longrightarrow & \#\underline{10}0\# \\
\#\underline{300}\# & \longrightarrow & \#\underline{100}\#
\end{array}$$

Cette notion permet d'introduire celle de *réduction active*, dont nous donnons maintenant la définition.

Définition 9.5.6 (Étape de réduction active) Soit R un système de réécriture. Une étape de réduction active de R est une étape de réduction de R dans laquelle le sous-mot réécrit est actif.

Définition 9.5.7 (Réduction active) Soit R un système de réécriture. Une réduction active de R est une réduction :
— commençant par un mot w dont la dernière lettre est la seule lettre active ;
— dont chaque étape est active.

Par exemple, la réduction suivante par l'algorithme de Ghosh est active :

$$\#312\underline{2}\# \longrightarrow_{M_2} \#32\underline{2}\# \longrightarrow_{M_1} \#\underline{33}2\# \longrightarrow_{T1} \#\underline{33}0\# \longrightarrow_{M_2} \#\underline{30}0\#$$

On voit que la partie active se propage jusqu'à contenir complètement le mot. Nous verrons plus loin que les réductions actives sont cette classe de réductions qu'il suffit de prouver convergentes pour prouver l'auto-stabilisation de l'algorithme. Intuitivement, tant qu'une partie d'un mot n'est pas active, il n'est pas pertinent de considérer les réductions qui s'y appliquent, donc il suffit de considérer les réductions actives.

9.5.4 Représentation des réductions actives

On a vu que les réductions actives d'un algorithme A étaient une reformulation de A sur les mots activés, telle que la partie active est propagée si la réécriture s'applique sur une partie active. On définit donc une nouvelle relation, prenant en paramètre une relation (un système de réécriture T, B ou M), et correspondant à la définition 9.5.7. `(actify T)` est donc la restriction de la relation `T` qui ne s'applique que sur des mots au moins partiellement actifs, et qui rend actifs les redex réécrits (c'est-à-dire qui *propage* la partie active du mot).

```
Inductive actify [A:Set; Ra: (A -> A -> Prop); u,v:A]:
  A -> A -> Prop :=
 | actify_ai:  (active_word u) -> (pure_active v)
  -> (Ra (inactive_all_Word u) (inactive_all_Word v))
  -> (actify Ra u v).
```

Nous aurons également besoin dans la suite de considérer les relations ne s'appliquant qu'aux mots inactifs ; c'est pourquoi nous définissons la propriété suivante, qui caractérise de telles relations :

```
Definition Rel_inactive:=[R:stria -> stria  -> Prop] (u,v: stria)
        (R u v) -> (inact_word u) /\ (inact_word v).
End act_inact.
```

On peut donc maintenant définir la réécriture active. Pour cela on commence par déclarer les trois ensembles de règles. Ces derniers sont supposés avoir le type `stria -> stria -> Prop` mais ne s'appliquent qu'aux mots inactifs :

```
Section active_red.
Variable A:Set.
Syntactic Definition stria :=(Word (lettre A)).
Variables T,B,M: stria -> stria -> Prop.

Hypothesis T_in: (Rel_inactive ? T).
Hypothesis B_in: (Rel_inactive ? B).
Hypothesis M_in: (Rel_inactive ? M).
```

Le mot-clé `Hypothesis` permet d'introduire une variable temporaire, de la même manière que `Variable`. Toutes ces hypothèses seront donc à démontrer à chaque cas particulier.

On définit ensuite la relation de réécriture sur les mots inactifs en utilisant `TBM_R` sur `T`, `B` et `M`.

```
(* La réécriture sur les mots inactifs *)
Definition R := (TBM_R A T B M).

(* Restriction de R aux mots n'appartenant pas à L *)
Definition R' := (restrict_R A R L).
```

On définit également les restrictions de `R` à {`T`} et à {`M`,`B`}. Cette dernière correspond à l'hypothèse 5 sur les algorithmes considérés (section 9.4).

```
Inductive Top: stria -> stria -> Prop :=
 | TopT: (u,v,w:stria) (T u v) -> (Top (w^u) (w^v)).

Inductive RmoinsTop:stria -> stria -> Prop :=
 | B_B': (u,v,w:stria) (B u v) -> (RmoinsTop (u^w) (v^w))
 | M_M': (u,v,w,w':stria) ~(w=stre) -> ~(w'=stre)
  -> (M u v) -> (RmoinsTop (w^(u^w')) (w^(v^w'))).
```

On définit ensuite la réécriture qui propage la partie active en utilisant toujours `TBM_R` sur les versions "activées" de `T`, `B` et `M` :

```
(* Les versions activées de T, B, M *)
Definition Tact:=(actify A T).
Definition Bact:=(actify A B).
Definition Mact:=(actify A M).

(* La réécriture sur les mots qui propage la partie active *)
Definition Ract := (TBM_R A Tact Bact Mact).

(* Restriction: *)
Definition Ract' := (restrict_R A Ract L).

Inductive Topact: stria -> stria -> Prop :=
 | TopTa: (u,v,w:stria) (Tact u v) -> (Topact (w^u) (w^v)).
```

Remarque 9.5.1 *On remarque que si `R'`∪`Ract'` termine, alors `R` est auto-stabilisant pour `L`. En effet, l'union des deux relations permet de simuler `R` sur les mots actifs et inactifs. On verra plus loin que, sous certaines hypothèses, il suffit même que `R'`∪`Ract'` termine pour les mots dont seule la dernière lettre est active.*

9.5.5 Les hypothèses sur le réseau

Pour correspondre aux conditions décrites à la section 9.4, il faut que l'algorithme vérifie les propriétés suivantes :

— T, B et M préservent la longueur des mots ;
— A ne s'arrête pas en dehors de $\mathcal{L}$;
— l'algorithme privé des règles de T termine ;
— l'algorithme est stable pour $\mathcal{L}$.

```
(* Les 3 systèmes préservent la longueur *)
Hypothesis T_length_p:(length_preserve ? T).
Hypothesis M_length_p:(length_preserve ? M).
Hypothesis B_length_p:(length_preserve ? B).

Hypothesis RmoinsTop_termine: (x:stria) (finite stria RmoinsTop x).
Hypothesis R_notL_red: (x:stria) ~(L x) -> (EX y|(R x y)).

Hypothesis R_closure:(closure ? R L).
Hypothesis Ract_closure:(closure ? Ract L).
```

On va maintenant démontrer que si les réductions actives convergent vers $\mathcal{L}$, alors toutes les réductions convergent vers $\mathcal{L}$. La démonstration donnée dans [JBM01] utilise un lemme intermédiaire.

9.5.6 La démonstration du lemme principal

Le lemme est un résultat bien connu de réécriture, dont on donne ici la preuve habituelle (et non constructive) :

Lemme 9.5.8 Soient deux relations R et Q vérifiant la propriété de semi-commutation suivante :

$$\forall x, y, z.(R\ x\ y) \wedge (Q\ y\ z) \Rightarrow \exists y'.(Q\ x\ y') \wedge (R^*\ y'\ z)$$

Si R et Q termine, alors $R \cup Q$ termine aussi.

Preuve : On procède par l'absurde ; supposons qu'il existe une réduction infinie Δ par $R \cup Q$, commençant par un terme t. On distingue deux cas possibles (notons qu'on utilise ici le tiers exclu sur une propriété non décidable) :
— soit il y a un nombre fini d'étapes correspondant à Q, auquel cas il y a un élément de la réduction à partir duquel on a une réduction infinie par R, ce qui constitue une contradiction par rapport aux hypothèses ;
— soit il y a une infinité d'étapes correspondant à Q, auquel cas on peut construire une réduction infinie par Q à partir de t en utilisant la semi-commutation pour "faire remonter" les étapes de Q en tête de la réduction. On obtient donc ici aussi une contradiction.
Par l'absurde, il n'y a donc pas de réduction infinie par $R \cup Q$.

On voit que cette démonstration – utilisant le tiers exclu sur la propriété indécidable «contenir un nombre infini d'étapes de Q» – n'est pas constructive. Il est donc nécessaire d'en donner une différente dans Coq. En fait, on peut démontrer cette propriété par différentes inductions imbriquées, comme nous le montrons maintenant.

Comme d'habitude, on commence par la déclaration des hypothèses :

```
Section unionacc.
```

```
Variable T:Set.
Variables R,Q:T -> T -> Prop.

(* Q termine, l'hypothèse "R termine" n'est pas nécessaire au début *)
Hypothesis Qacc:(x:T) (finite Q x).
Hypothesis commut_R:(x,y,z:T) (R x y) -> (Q y z)
  -> (EX y'| (Q x y') & (star R y' z)).

(* On déduit, par itération de l'hypothèse commut_R, la propriété suivante: *)
Lemma commut_R_rec:(x,y,z:T) (star R x y) -> (Q y z)
  -> (EX y'| (Q x y') & (star R y' z)).
...
Save.
```

On veut maintenant montrer que si R et Q terminent et semi-commutent, alors $R \cup Q$ termine aussi. En fait, pour un x donné, on montre que si Q termine et R termine pour tous les successeurs de x par Q, alors $R \cup Q$ termine. On procède par lemmes successifs, chaque lemme utilise le précédent.

On commence par montrer que si $R \cup Q$ termine pour les successeurs de x par R^* o Q, alors $R \cup Q$ termine aussi pour les successeurs de x par R ou Q (trivial).

```
Lemma acc_union1:(x:T) (finite R x)
  -> ((y:T) (EX z|(star R x z)&(Q z y))
      -> (finite (union Q R) y))
  -> (finite (union Q R) x).
Intros x accRx.
Elim accRx.
Intros. Constructor. Intros y hyp. Inversion hyp.
Apply H1. Exists x0;Auto with reldb.
Apply H0. Assumption. Intros.
Apply H1. Elim H3. Intros.
Exists x1; Try Assumption.
EApply star_trans1 with y;Auto.
Save.
```

On restreint l'hypothèse : il suffit que seuls les successeurs par Q soient accessibles pour $R \cup Q$ (on utilise ici la commutation), pourvu que R termine :

```
Lemma acc_union2:(x:T) (finite R x)
         -> ((y:T) (Q x y) -> (finite (union Q R) y))
         -> (finite (union Q R) x).
Intros.
Apply acc_union1.
Assumption. Intros. Elim H1;Intros.
Elim (commut_R_rec x x0 y);Try Assumption. Intros.
Cut (finite (union Q R) x1).
Elim H5.
```

```
Intros. Assumption. Intros. Apply H8.
Inversion H9. Apply H10. Right. Assumption.
Auto.
Save.
```

On en déduit que si R termine pour tous les successeurs de x par Q^*, alors $R \cup Q$ termine aussi pour x (on rappelle que Q termine par l'hypothèse `Qacc`).

```
Lemma acc_union: (x:T) ((e:T) (star Q x e) -> (finite R e))
  -> (finite (union Q R) x).
Intros x hyp1.
Apply acc_union2.
Apply hyp1. Apply star_refl.
Generalize hyp1.
Elim (Qacc x).
Intros. Apply acc_union2.
Apply hyp0;EAuto with reldb.
Intros. Apply (H0 y);Try Assumption.
Intros. Apply hyp0.
EApply star_trans1 with y;Auto.
Save.
```

On en déduit finalement le lemme :

```
Lemma acc_union_forall:
 ((x:T) (finite R x))
  -> ((x:T) (finite (union Q R) x)).
Intros. Apply acc_union. Auto.
Save.
```

9.5.7 La restriction aux réductions actives

On peut maintenant démontrer le principal résultat de la partie générique. On sait déjà qu'il suffit de montrer que `R' ∪Ract'` termine pour montrer que `R` converge vers `L`. Le théorème suivant permet de se restreindre aux réductions actives (c'est-à-dire par `Ract'` et commençant par un mot dont la dernière lettre est active).

Théorème 9.5.9 Si la réduction `Ract'` termine pour tout mot ayant pour seule lettre active la dernière, alors la réduction `R'∪Ract'` termine pour tout mot (non vide).

On exprime ce théorème de la manière suivante :

```
Theorem topc:((x:stria) (last_active x) -> (finite stria Ract'  x))
  -> (x:stria) ~x=epsilon -> (finite stria (union R' Ract') x).
Intros hyp x hyp2.
Apply lastact_only. (* lemme *)
Apply Ract_only.    (* lemme *)
...
Save.
```

On explique ici les deux lemmes principaux permettant de prouver ce résultat :

— Premièrement, pour prouver la convergence de `R'∪Ract'`, il suffit de considérer les réductions commençant par un mot dont seule la dernière lettre est active. En effet, l'utilisation de lettres actives et inactives est redondante et ce sous-ensemble de mots de départ suffit pour prouver la convergence sur tous les mots de la relation `R` de départ.

```
Lemma lastact_only: (x:stria)
 ((finite  (union R' Ract') (active_last ? x)))
   -> (finite  (union R' Ract') x).
```

La preuve de ce lemme intermédiaire nécessite un travail assez lourd de raisonnement sur les mots, dont nous ne donnons pas les détails ici.

— Deuxièmement, si pour un `x` donné tous les successeurs `e` de `x` par `R'` sont tels que `Ract'` termine à partir de `e`, alors `R'∪Ract'` termine à partir de `x`.

```
Lemma Ract_only:(x:stria)
 (last_active x)
  ->((e:stria)(star R' x e)->(finite Ract' e))
  ->(finite (union R' Ract') x).
Intros x accR'x accRacttrR'.
Apply (acc_union ? R' Ract').
...
```

La preuve de ce lemme utilise le lemme `acc_union` décrit plus haut. En effet `R'` et `Ract'` semi-commutent et `R'` termine, donc il suffit que `Ract'` termine pour que `R'∪Ract'` termine.

9.6 La partie spécifique de la preuve

9.6.1 L'alphabet $\mathbb{N}$

Remarque 9.6.1 *Afin d'avoir une représentation concrète des lettres de l'alphabet, on décide de se placer désormais sur l'alphabet* $\mathbb{N}$. *On redéfinit donc le type* `stria` *comme les mots activés sur l'alphabet des entiers (*`nat`*) :*

```
Require Arith.(* Pour avoir la syntaxe (1) pour l'entier 1. *)

Syntactic Definition stria :=(Word (lettre nat)).
```

On effectue quelques définitions spécifiques aux mots activés sur `nat`.

```
Definition w_one := |+(1)+| :: epsilon.
Definition set_one:= (lword (lettre nat) w_one).
Definition w_zero := |+(0)+| :: epsilon.
Definition set_zero:= (lword (lettre nat) w_zero).
```

On définit également les ensembles correspondant aux expressions régulières $\underline{1}^*, \underline{1}^+, \underline{0}\,\underline{1}^+, \underline{1}\,\underline{1}^+$ qui serviront dans la suite.

```
Definition Unstar := (lstar (lettre nat) set_one).
Definition Unplus := (lplus (lettre nat) set_one).
Inductive zero_unplus:stria -> Prop:=
```

```
        zunpl:(u:stria) (Unplus u) -> (zero_unplus (|+(0)+| :: u)).
Inductive Un_unplus:stria -> Prop:=
        uunpl:(u:stria) (Unplus u) -> (Un_unplus (|+(1)+| :: u)).
...
```

9.6.2 Un exemple d'algorithme

On utilise pour illustrer la méthode un exemple très simple d'algorithme à deux états : 0 ou 1. Les trois ensembles de règles sont les suivants :

— T :

$$00 \longrightarrow 01$$
$$10 \longrightarrow 11$$

— B :

$$00 \longrightarrow 01$$
$$11 \longrightarrow 01$$
$$10 \longrightarrow 11$$

— M :

$$00 \longrightarrow 01$$
$$10 \longrightarrow 11$$

L'ensemble des configurations légitimes est le suivant :

$$L = \{01^+\}$$

On déclare ces données de la manière suivante :

```
Inductive ex1T: stria -> stria -> Prop :=
        | ex1T1:(ex1T (|-(0)-| :: (|-(0)-| :: epsilon))
                      (|-(0)-| :: (|-(1)-| :: epsilon)))
        | ex1T3:(ex1T (|-(1)-| :: (|-(0)-| :: epsilon))
                      (|-(1)-| :: (|-(1)-| :: epsilon))).

Inductive ex1B: stria -> stria -> Prop :=
        | ex1B1:(ex1B (|-(0)-| :: (|-(0)-| :: epsilon))
                      (|-(0)-| :: (|-(1)-| :: epsilon)))
        | ex1B2:(ex1B (|-(1)-| :: (|-(1)-| :: epsilon))
                      (|-(0)-| :: (|-(1)-| :: epsilon)))
        | ex1B3:(ex1B (|-(1)-| :: (|-(0)-| :: epsilon))
                      (|-(1)-| :: (|-(1)-| :: epsilon))).

Inductive ex1M: stria -> stria -> Prop :=
        | ex1M1:(ex1M (|-(0)-| :: (|-(0)-| :: epsilon))
                      (|-(0)-| :: (|-(1)-| :: epsilon)))
        | ex1M2:(ex1M (|-(1)-| :: (|-(0)-| :: epsilon))
                      (|-(1)-| :: (|-(1)-| :: epsilon))).

Hints Resolve ex1T1 ex1T3 ex1B1 ex1B2 ex1B3 ex1M1 ex1M2 : acthint.
```

```
Definition ex1L:=zero_unplus.

Syntactic Definition ex1R :=(R ex1T ex1M ex1B).
Syntactic Definition ex1Ract :=(Ract ex1T ex1M ex1B).
Syntactic Definition ex1R' :=(R' ex1T ex1M ex1B ex1L).
Syntactic Definition ex1Ract' :=(Ract' ex1T ex1M ex1B ex1L).
Syntactic Definition ex1Tact :=(Tact ex1T).
Syntactic Definition ex1Bact :=(Bact ex1B).
Syntactic Definition ex1Mact :=(Mact ex1M).
```

On démontre ensuite que les hypothèses sont vérifiées :

```
Lemma ex1L_stable:(u:stria)
 (ex1L u) -> (v:stria) (ex1Ract u v) -> (ex1L v).
...
Save.
...
```

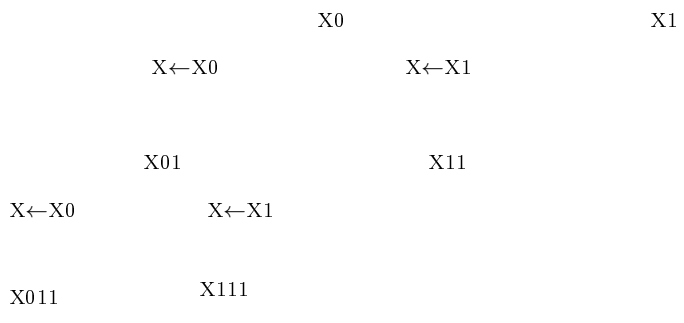
9.6.3 Démonstration de la terminaison de `Ract'`

Pour démontrer la terminaison de `Ract'`, on utilise un méthode issue de la réécriture : le *narrowing*. Cette technique, décrite dans [JBM01], consiste à décrire toutes les réductions possibles sous la forme d'un graphe où les nœuds correspondent à des ensembles de mots et les arêtes à des étapes de réduction (actives). On utilise des expressions régulières pour décrire ces ensembles. On verra qu'en fait la construction du graphe correspond à un raisonnement par cas sur les lettres inactives des mots.

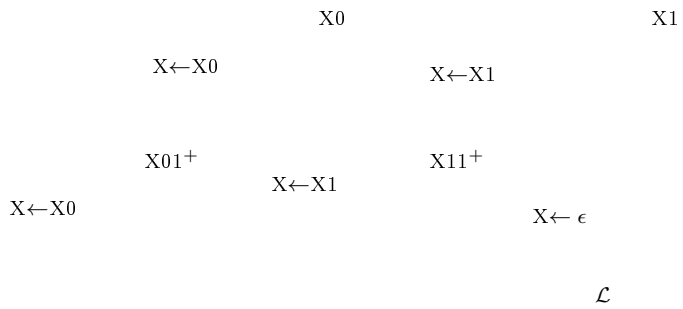
Dans notre exemple, une réduction active ne peut commencer que par une étape de l'ensemble de règles T, puisque seule la dernière lettre est active. Donc d'après les règles de T :

- le mot de départ est de la forme $w0$,
- le second mot d'une réduction active est soit de la forme $w01$ soit $w11$, et seules les deux dernières lettres sont actives,
- etc.

On obtient le graphe (incomplet) suivant :



Pour que le processus termine, on doit étiqueter certains nœuds du graphe par des expressions régulières. Le choix des expressions régulières est assez subtil et n'est a priori pas automatisable. On obtient un graphe fini cette fois, mais contenant des cycles :



Tous les chemins du graphe conduisent à $\mathcal{L}$, sauf le chemin qui passe indéfiniment par le cycle. Cependant on montre que ce cycle ne pourra être emprunté qu'un nombre fini de fois par un mot, dont on rappelle que la longueur est finie et ne change pas, car la partie active du mot augmente à chaque cycle. Cet argument est en fait un raisonnement par récurrence sur la taille du mot considéré. On montre ci-dessous les énoncés des lemmes correspondant aux flèches du graphe. À chaque fois, le lemme est de la forme «Si les réductions terminent à partir de tels mots (chacun contenant une variable), alors toutes les réductions terminent».

Le premier lemme correspond aux deux premières flèches du graphe, c'est-à-dire celles qui partent du sommet X0. Pour l'instant on n'utilise pas d'expression régulière.

```
Lemma fleche1:
 ((u:stria)
   (EX v|(inact_word v) /\ (u=v^(|+(0)+| :: (|+(1)+| :: stre))
  \/ u=v^(|+(1)+| :: (|+(1)+| :: stre))))
  -> (finite ex1Ract' u))
  -> (u:stria) ~(u=omega) -> (finite (union ex1R' ex1Ract') u).
```

Pour prouver ce lemme, on utilise le résultat de la partie générique de la preuve : il suffit de prouver que `Ract'` termine pour prouver que `R'`$\cup$`Ract'` termine.

Le deuxième lemme correspond à un affaiblissement du premier, on introduit des expressions régulières : le mot v01 est remplacé par v01$^+$ et v11 par v11$^+$.

```
(* On generalise fleche1 avec les regexp *)
Lemma fleche1':
 ((u:stria) (EX v|(inact_word v)
  /\ (EX t| u=v^t /\ ((zero_unplus t) \/ (Un_unplus t))))
  -> (finite ex1Ract' u))
  -> (u:stria)~(u=omega) -> (finite (union ex1R' ex1Ract') u).
```

Il s'agit donc maintenant de montrer `(finite ex1Ract' u)` avec `u`=XO1$^+$ ou X11$^+$. Les deux lemmes suivants correspondent aux flèches du graphe qui partent des deux sommets X11$^+$ et X01$^+$.

De X11$^+$ on ne peut réduire que vers $\mathcal{L}$, donc `Ract'` termine :

```
Lemma fleche_X11:(u,v:stria)(inact_word v)
  -> (t:stria) (pure_active t) -> u=v^t
  -> (Un_unplus t) -> (finite ex1Ract' u).
```

De X01$^+$, on peut réduire
— vers X11$^+$ et le lemme précédent permet de conclure,
— vers X01$^+$ lui-même, et on prouve par récurrence qu'au bout d'un nombre fini de cycles, la réduction ira vers X11$^+$.

Donc `Ract'` termine :

```
Lemma fleche_X01:(v:stria)(inact_word v)
  -> (u,t:stria) (pure_active t) -> u=v^t
  -> (zero_unplus t) -> (finite ex1Ract' u).
```

Voici le théorème final établissant que `R'` ∪ `Ract'` termine :

```
Theorem Ract'_finite:
         (u:stria) ~(u=epsilon) -> (finite (union ex1R' ex1Ract') u).
```

Pour la preuve, on applique les trois lemmes successivement, ce qui revient à énumérer les chemins du graphe, et à prouver pour chacun qu'il aboutit nécessairement dans $\mathcal{L}$.

```
Intros u unotomega.
```

On applique d'abord le premier lemme, selon lequel il suffit de montrer que `ex1Ract'` termine pour XO1$^+$ ou X11$^+$.

```
Apply fleche1';Try Assumption. Clear unotomega u.
```

Puis on traite les deux cas séparément en utilisant les deux lemmes `fleche_XO1` et `fleche_X11`, ce qui permet de conclure immédiatement :

```
Intros u hyp.
Decompose [ex or and] hyp. (* Séparation des deux cas *)
EApply fleche_XO1 with v:=x t:=x0;Auto with acthint.
EApply fleche_X11 with v:=x t:=x0;Auto with acthint.
Save.
```

9.6.4 Conclusion

On voit qu'une fois que les lemmes correspondant aux flèches sont démontrés, la démonstration est relativement simple. Le théorème générique démontré plus haut (9.5.7) nous a servi pour prouver le premier lemme.

Dans le cadre d'un environnement de preuve pour les algorithmes auto-stabilisants, on aura besoin d'un outil particulier pour trouver et prouver les lemmes. Plusieurs pistes sont possibles, l'écriture d'une tactique permettant d'engendrer et d'aider à la preuve des lemmes en développant le graphe comme ci-dessus est envisagée. Par ailleurs un programme interactif [Mag] dédié pour engendrer les lemmes intermédiaires est en cours d'implantation par Frédéric Magniette [JBM01].

Quel qu'il soit, l'outil devra en tout cas laisser une place à l'utilisateur pour intervenir, puisque l'étape, critique pour la réussite de la preuve, de choix des expressions régulières étiquetant le graphe ne semble pas facilement automatisable.

Chapitre 10

Conclusion, perspectives

Dans cette thèse, nous avons présenté d'une part un travail théorique avec l'extension du calcul des constructions inductives permettant un meilleur traitement des quotients, et d'autre part un travail plus appliqué de formalisation dans l'assistant de preuve Coq d'une preuve d'algorithme auto-stabilisant.

Concernant l'extension de $\mathcal{CCI}$, nous avons défini une nouvelle représentation des quotients, à l'aide d'une fonction d'interprétation des termes du type support. Le fait que cette fonction soit exprimée *dans* le calcul est un des avantages principaux de cette approche, car elle permet de garantir un certain nombre de propriétés, comme la terminaison et la confluence, sans faire appel à des notions extérieures comme la réécriture.

En fait, exprimer la fonction d'interprétation dans le calcul est un avantage d'un point de vue théorique, mais présente des inconvénients au niveau pratique. Il s'agit en effet ici de définir une égalité entre les différentes représentations de la même classe d'équivalence, et donc de permettre un raisonnement équationnel ; or c'est la réécriture qui est l'outil privilégié du raisonnement équationnel, notamment grâce à la possibilité de remplacer *n'importe quel* sous-terme par un sous-terme équivalent. Il semble envisageable de combiner les avantages de la réécriture et de notre méthode en se basant sur les travaux récents de calculs permettant la définition de fonctions par réécriture [Bla01, BJO01]. L'idéal semble alors de définir notre fonction d'interprétation par un système de réécriture.

On récapitule maintenant brièvement les fonctionnalités des types normalisés :

— l'égalité du quotient est inclue dans la conversion du calcul,
— les principes d'élimination faible et fort sont définis sur les types normalisés, avec un comportement calculatoire acceptable et sans preuve de compatibilité à fournir,
— il est possible de prouver des principes d'induction plus fidèles à la structure du quotient.

Outre l'implantation dans une version dérivée de Coq du calcul des constructions inductives avec types normalisés, quelques perspectives nous semblent intéressantes.

Concernant le dernier point ci-dessus notamment, il semble intéressant d'étudier le moyen d'engendrer les principes d'élimination automatiquement, notamment en adaptant les résultats d'induction automatique connus sur la réécriture [Com94, Com89, BJar].

Il semble aussi intéressant d'étudier les rapport existant entre notre approche et celle des quotient décidable de L. Pottier [Pot00], qui semble assez proche.

Enfin on peut supposer que notre méthode d'extension du calcul des constructions inductives, consistant à associer une fonction et un type, s'applique à d'autres extensions. Par exemple, il est peut-être possible de définir un calcul des constructions inductives avec sous-types en utilisant une fonction de passage au sur-type, à la manière d'une coercion.

Plus directement, il semble intéressant d'étudier l'utilité d'un calcul des constructions inductives avec types normalisés dans lequel la fonction d'interprétation *nf* pour un type `Norm(A,`*nf*`)` n'est pas de type `A -> A` mais de type `A -> B` où `B` est un type quelconque. On obtient alors deux représentations pour une même structure avec un moyen simple de passer de l'une à l'autre. Un exemple fourni par G. Barthe est celui des λ-termes et leurs deux représentations : la représentation classique et celle par les indices de De Bruijn ; l'une a l'avantage d'être intuitive et facile à manipuler, l'autre a l'avantage d'être une structure libre.

En ce qui concerne l'auto-stabilisation, si la formalisation de la partie générique est faite une fois pour toutes, la preuve spécifique nécessaire à chaque nouveau cas semble pouvoir s'automatiser en grande partie. Il nous semble intéressant de :

— concevoir des outils d'aide pour les parties (semi-)automatisables, en particulier l'étape de construction des expressions régulières ;

— formaliser d'autres preuves utilisant des techniques d'auto-stabilisation, et permettre ainsi la conception d'un véritable environnement de preuve spécialisé.

Index

Table des figures

Bibliographie

[Bar92] Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background : Mathematical Structures) and 2 (Background : Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

[Bar95] G. Barthe. Extensions of pure type systems. In *proc TLCA'95*, volume 902 of *lncs*. Springer-Verlag, 1995.

[BG95] G. Barthe and H. Geuvers. *Congruence types.* In H. Kleine Buening, editor, *Proc. Conf. Computer Science Logic*, volume 1092 of *Lecture Notes in Computer Science*, pages 36–51, 1995.

[BJO01] F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-Data-Type Systems. *Theoretical Computer Science*, 277, 2001.

[BJar] Adel Bouhoula and Jean-Pierre Jouannaud. Automata-driven automated induction. *Information and Computation*, 2000, to appear.

[Bla01] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. In *Proceedings of LICS2001*, 2001. to appear.

[Bou66] N. Bourbaki. Théorie des ensembles. In *Éléments de mathématiques.* Hermann, 1966.

[Bou97] Samuel Boutin. *Réflexions sur les quotients.* Thèse d'université, Paris 7, April 1997.

[CAB+86] Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, 1986.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76 :95–120, February 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940.

[Com89] Hubert Comon. Inductive proofs by specifications transformation. In *Proc. 3rd Rewriting Techniques and Applications, Chapel Hill, LNCS 355*, pages 76–91. Springer-Verlag, April 1989.

[Com94] Hubert Comon. Inductionless induction. In René David, editor, *2nd International Conference in Logic For Computer Science : Automated Deduction. Lecture notes.*, Chambéry, July 1994. Univ. de Savoie.

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

[Coq85] Thierry Coquand. Une théorie des constructions. Thèse de Doctorat, Univ. Paris 7, 1985.

[Coq86] Thierry Coquand. An analysis of girard's paradox. In *Proc. 1st IEEE Symp. Logic in Computer Science, Cambridge, Mass.*, 1986.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[dB72] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. of the Koninklijke Nederlands Akademie*, 75(5) :380–392, 1972.

[DHK98] G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo, 1998. Rapport de recherche INRIA 3400.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11) :643–644, 1974.

[Dow95] G. Dowek. *La logique*. Flammarion, 1995.

[Dow99] Gilles Dowek. *La part du calcul*. Thèse d'habilitation, Université Paris 7, 1999.

[Fre03] G. Frege. *Grundgesetze der Arithmetik, Begriffsschriftlich abgeleitet*. Verlag Hermann Pohle, Jena, 1893–1903.

[Gen34] G. Gentzen. Untersuchungen über das logische schliessen, 1934.

[Gen69] G. Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*. North-Holland, 1969.

[Geu] J. Geuvers. The calculus of constructions and higher order logic.

[Geu92] H. Geuvers. The church rosser property for $\beta\eta$-reduction in typed λ-calculi. In *Proc. 7th IEEE Symp. Logic in Computer Science, Santa Cruz*, pages 453–460, 1992.

[Gho93] S. Ghosh. An alternative solution to a problem of system stabilization. In *Proceedings of TOPLAS*, pages 735–742. ACM Press, 1993.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[Göd31] K. Gödel. Uber formal unentscheidbare satze der principia mathematica und verwandter systeme, 1931.

[Göd34] K. Gödel. On undecidable propositions of formal mathematical systems ; in : Collected works, 1934.

[Hey56] A. Heyting. *Intuitionism : An Introduction*. North-Holland, Amsterdam, 1956.

[Hof95a] M. Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, Edinburgh university, 1995.

[Hof95b] M. Hofmann. A simple model for quotient types. In *proc TLCA'95*, volume 902 of *lncs*. Springer-Verlag, 1995.

[How80] William A. Howard. The formulae-as-types notion of construction. In J. Roger Hindley Jonathan P. Seldin, editor, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[JBM01] L. Fribourg J. Beauquier, B. Bérard and F. Magniette. Proving convergence of self-stabilizing systems using first-order rewriting and regular languages. *Distributed Computing*, 14(2) :83–95, 2001.

[Klo92] Jan Willem Klop. Term Rewriting Systems. In S. Abramsky, Dov.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Clarendon Press, 1992.

[Mag] Frédéric Magniette. Poulet. `http://www.lri.fr/~magniett/poulet/index.html`.

[ML84] Per Martin-Löf. *Intuitionistic type theory.* Bibliopolis, 1984.

[Ned73] R. P. Nederpelt. *Strong normalization in a typed lambda calculus with lambda structured types.* PhD thesis, Eindhoven The Netherlands, 1973.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, pages 328–345. Springer-Verlag, 1993. LNCS 664.

[PM96] Christine Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur.* thèse d'habilitation, Université Claude Bernard Lyon I, Décembre 1996.

[Pot00] Loïc Pottier. Quotients dans le cci, November 2000. Rapport de recherche INRIA 4053.

[Pra65] Dag Prawitz. *Natural Deduction, A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

[RBS89] G. Malcolm R. Backhouse, P. Chisholm and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1989.

[Shn93] M. Shneider. Self-stabilization. *ACM Computing Surveys*, 25(1), 1993.

[SU98] Morten Heine. Sørensen and P. Urzyczyn. Lectures on the curry-howard isomorphism. Available as DIKU Rapport 98/14, 1998.

[Wer94] Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives.* Thèse de doctorat, Univ. Paris VII, 1994.