Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories.

Amokrane Saibi

THÈSE DE DOCTORAT

présentée

A L'UNIVERSITÉ PARIS 6

Spécialité : Informatique

par

Amokrane SAÏBI

Sujet de la thèse :

Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories.

Soutenue le 19 Mars 1999 devant la Commission d'examen composée de :

M.	Daniel Lazard	Président
M.	Thierry Coquand	Rapporteurs
M.	Giuseppe Longo	
M.	Gérard Huet	Directeur de Thèse
Mme.	Thérèse Hardin	Examinateurs
M.	Andrzej Trybulec	

Je tiens à remercier:

- Daniel Lazard de me faire l'honneur d'accepter de présider le jury.
- mon directeur de thèse, Gérard (Huet) pour nos nombreuses discussions libres et fructeueuses et pour m'avoir fait gagner beaucoup de temps en m'indiquant toujours des pistes de recherche judicieuses. Merci aussi de m'avoir introduit auprès de chercheurs du domaine, et surtout de m'avoir toujours laissé travailler librement et en toute confiance.
- Thérèse (Hardin) de m'avoir toujours soutenu, et de m'avoir introduit à l'INRIA et à Paris VI. Le groupe de travail qu'elle anime avec Véronique (Donzeau-Gouge) à Paris VI et au CNAM est très utile.
- Thierry Coquand et Giuseppe Longo d'avoir accepté d'être rapporteurs de cette thèse. Ce sont deux chercheurs pour lesquels j'ai beaucoup d'estime (le bouquin de Longo «Categories, Types, and Structures» a longtemps été mon livre de chevet).
- Andrey Trybulec pour avoir accepté de faire partie de mon jury de thèse. Ses travaux ont toujours été une référence pour moi.
- Peter (Aczel) de m'avoir accueilli pendant 4 mois, dans son équipe à l'université de Manchester. L'ambiance y était très chaleureuse avec Anthony (Bailey) et Gilles (Barthes).
- Rod (Burstall) pour s'être intéressé à mon travail.
- mes collègues et amis: Cesar (Munoz), Cristina (Cornes), Daniel (de Rauglaudre), Valérie (Menissier), Sam (Samuel Boutin), Ben (Benjamin Werner), Henri (Laulhère), Bruno (Barras), Hugo (Herbelin), Alexandre (Miquel), Jean (Goubault) ...
- le ministère Algérien de l'éducation et de la recherche (à travers mon «université d'origine», USTHB) pour m'avoir octroyé une bourse d'études.
- l'INRIA pour l'extraordinaire cadre de travail (sur les plans matériel et humain) dont j'ai bénéficié.

Table des matières

Introduction **9**

1 Formalisation des Mathématiques **13**
1.1 Introduction 13
1.2 Logique 13
1.3 Fondements des mathématiques 15
1.4 Mathématiques constructives 16
1.5 Automatisation des preuves sur ordinateur 16
1.6 Pratique de la formalisation des mathématiques 19
1.6.1 Expériences 19
1.6.2 Difficultés et Faisabilité 19

I Théorie des types **21**

Introduction **23**

2 Systèmes de types purs **25**
2.1 λ-calcul pur 25
2.2 λ-calcul simplement typé 27
2.3 Systèmes de types purs (PTS) 29
2.3.1 Définitions générales 29
2.3.2 Cube de Barendregt 31
2.4 Pouvoir d'expression 32
2.5 Correspondance logique 34
2.6 Théorie des types extensionnelle 36

3 Extensions **39**
3.1 Constantes 39
3.1.1 Présentation 40
3.1.2 Inférence de types 42
3.1.3 Traitement des constantes locales 45
3.2 Hiérarchie d'Univers 47
3.2.1 Présentation 47
3.2.2 Inférence de types 48

3.2.3 Univers flottants 49
3.3 Types inductifs 53
3.4 Enregistrements dépendants 55
3.4.1 Représentation de structures mathématiques 55
3.4.2 Types «somme forte» 56
3.4.3 Enregistrement comme cas particulier de type inductif 57
3.5 Conclusion 61

II Outils Génériques **63**

Introduction **65**

4 Sous-termes implicites **67**
4.1 Introduction 67
4.2 Principe et Définitions 68
4.3 Unification 71
4.4 Inférence de types 76
4.5 Heuristique pour l'unification en présence d'enregistrements 78
4.6 Syntaxe concrète des applications et insertion de «?» 80
4.7 Surcharge de noms de fonctions 82

5 Coercions Implicites **85**
5.1 Introduction 85
5.2 Travaux antérieurs 87
5.2.1 Proposition de Peter Aczel 87
5.2.2 PTS avec héritage 89
5.3 Motivations des extensions introduites 91
5.4 Langage des coercions 93
5.4.1 Classes 93
5.4.2 Coercions 94
5.4.3 Coercions identité 95
5.5 Graphe d'héritage 96
5.5.1 Composition de coercions 96
5.5.2 Cohérence 97
5.5.3 Une alternative à la condition de cohérence 98
5.6 Fonction d'explicitation de termes 99
5.6.1 Insertion de coercions 99
5.6.2 Algorithme d'inférence de types 104
5.6.3 Propriétés 105
5.7 Coercions entre types inductifs paramétrés 109

6 Tactiques de Réécriture **111**
6.1 Introduction 111
6.2 Principales égalités 112
6.2.1 Égalité de Leibniz 112

6.2.2 Autres égalités . . . 113
6.3 Notions de réécriture (Rappels) . . . 114
6.4 Activité de preuve . . . 116
6.4.1 Langage ML . . . 116
6.4.2 Tactiques . . . 117
6.5 Réécriture Générique . . . 118
6.5.1 Principe Général . . . 118
6.5.2 Conversions de base . . . 118
6.5.3 Lemmes utilisés . . . 119
6.5.4 Composition de conversions . . . 120
6.6 Raisonnement Équationnel . . . 122
6.7 Réflexion . . . 122

III Application: Formalisation de la Théorie des Catégories 125

Introduction 127

7 Préliminaires 129
7.1 Syntaxe Coq . . . 129
7.1.1 Syntaxe des termes . . . 129
7.1.2 Commandes . . . 130
7.1.3 Mécanisme des sections . . . 132
7.2 Relations . . . 132
7.3 Ensembles . . . 133
7.4 Applications entre deux Setoïdes . . . 134

8 Catégories, Foncteurs et Transformations naturelles 137
8.1 Catégories . . . 137
8.2 Catégorie des Setoïdes . . . 139
8.3 Catégorie Duale . . . 140
8.4 Propriétés simples sur les Catégories . . . 140
8.5 Foncteurs . . . 142
8.6 Égalité de morphismes . . . 143
8.7 La Catégorie des Catégories . . . 145
8.8 Transformation naturelle . . . 146
8.9 Catégorie des Foncteurs . . . 147
8.10 Techniques de preuve - Complétion . . . 148
8.10.1 Description de la méthode . . . 148
8.10.2 Cas des Catégories . . . 150
8.10.3 Cas des Foncteurs . . . 152
8.10.4 Cas des Transformations Naturelles . . . 152
8.11 Exemple – La loi d'échange . . . 153
8.11.1 Énoncé . . . 153
8.11.2 Sous-termes implicites . . . 154
8.11.3 Preuve par réécriture . . . 156

8.11.4 Associativité de la composition horizontale 157

9 Constructions Universelles et Adjonctions **159**
9.1 Constructions Universelles et Limites 159
9.1.1 Morphisme Universel 159
9.1.2 Définition d'un Morphisme Co-universel 161
9.1.3 Limites et Colimites 162
9.1.4 Préservation de Limites 164
9.1.5 Exemples de limites 165
9.1.6 Existence d'Objet Initial 170
9.2 Catégorie Comma 173
9.2.1 Définition 173
9.2.2 Relation entre Catégorie Comma et Morphisme Universel 174
9.2.3 Conditions Suffisantes à la Complétude d'une Catégorie Comma 174
9.3 Adjonctions 177
9.3.1 Définitions Préliminaires 177
9.3.2 Définition d'une Adjonction 179
9.3.3 Une autre définition de l'adjonction 181
9.3.4 Adjonctions et réécriture 182
9.3.5 Construction d'un Adjoint Gauche 183
9.4 Théorème du Foncteur Adjoint de Freyd 184
9.4.1 Énoncé 184
9.4.2 Condition Nécessaire 185
9.4.3 Condition Suffisante 187
9.5 Aperçu sur le reste du développement 188

10 Analyse et Critiques **191**
10.1 Quelques chiffres 191
10.2 Sur la forme 191
10.3 Discussion - Égalité d'objets 193
10.4 Mizar 195
10.5 Autres formalisations 199

Conclusion **205**

Bibliographie **206**

Introduction

L'utilisation des ordinateurs en mathématiques ne se limite pas au calcul numérique. En effet depuis l'avènement des systèmes de calcul formel dans les années soixantes, l'ordinateur est capable de manipuler des expressions symboliques, réalisant ainsi des tâches mathématiques plus complexes telles que la factorisation de polynômes. Durant les dernières années, les systèmes de preuves ont connu un développement important. Plus universels que les systèmes de calcul formel, ils sont capables de manipuler des notions mathématiques arbitraires grâce à l'utilisation de quantificateurs et de prédicats. Ils sont aussi plus rigoureux car permettent de représenter et de manipuler des preuves formelles. Ils ont pour cela un langage précis pour écrire les propositions, et des règles précises pour définir les inférences et les preuves valides.
Parmi les systèmes de preuves, nous distinguons les «vérificateurs de preuves» (proof-checkers en anglais) dont les pionniers sont Automath et Mizar (voir le chapitre suivant pour un historique des systèmes de preuves). Les vérificateurs de preuves sont différents des systèmes de démonstration automatique: ils se contentent de vérifier la correction de la preuve fournie par l'utilisateur vis-à-vis de l'énoncé à démontrer. La raison est que la vérification automatique (donc sur ordinateur) est toujours possible, alors que la démonstration automatique est en général indécidable. Pour des domaines précis et décidables, les systèmes de démonstration automatique peuvent tout à fait convenir. Les systèmes de preuves modernes sont des systèmes interactifs intégrant les deux aspects de vérification de preuves et de démonstration automatique. En effet, il est trop pénible pour l'utilisateur d'écrire une preuve complète; les parties «faciles» de la preuve sont prouvées automatiquement par le système, sans intervention de l'utilisateur. Nous préférons dans ces cas-là parler de «système d'aide à la démonstration» (ou SAD).

Cette thèse concerne la formalisation des mathématiques dans le cadre des systèmes de preuves basés sur la théorie des types. Ces systèmes de preuves ont été à l'origine du renouveau de la méta-mathématique et de l'application de la logique à la formalisation des mathématiques; en effet nous croyons que la formalisation des mathématiques n'est réalisable qu'avec l'aide de l'ordinateur et des systèmes de preuves. Le succès récent des systèmes de preuve provient de leur utilisation pour la production de programmes informatiques certifiés (par rapport à une spécification). La problématique de la formalisation des mathématiques est toutefois liée à ce dernier point; en effet pour prouver la correction d'un programme, nous utilisons des objets et des formules mathématiques pour exprimer des propriétés sur ce programme. Dans certains cas, ces preuves de programmes nécessitent l'utilisation de théories mathématiques conséquentes.

L'adéquation des systèmes de preuves pour la formalisation des mathématiques est encore théorique et sujette à caution; en effet très peu de développements conséquents (par la taille et

par la complexité mathématique des résultats formalisés) ont été réalisés à ce jour. La raison principale en est la quantité de travail que cela représente: il faut écrire très soigneusement les énoncés mathématiques sans user d'aucun abus de notation (dont pourtant les mathématiciens raffolent) et il faut justifier chaque étape de raisonnement. De plus, le résultat final est médiocre: assimilable par l'ordinateur mais illisible par l'humain.

L'objectif de cette thèse est de contribuer à remédier à cette faiblesse en développant des outils de modélisation et de démonstration facilitant l'utilisation des systèmes de preuves basés sur la théorie des types pour la formalisation des mathématiques, notamment de l'algèbre. Tous les outils développés sont génériques au sens qu'ils ne sont pas conçus pour un développement particulier; ils peuvent être utilisés dans différents cas de figure. Nous signalons enfin que ces outils ont été implémentés et intégrés dans le système de preuve Coq[49].

- Les outils de modélisation constituent des mécanismes de simulation d'abus de notation dans lesquels certaines informations dites *implicites* peuvent être omises mais sont automatiquement reconstruites à partir du contexte. Le premier outil, qui constitue la principale contribution de cette thèse, est celui des *coercions implicites* basé sur une forme d'héritage des langages à objets. Grâce à cet outil, on peut, par exemple, appliquer une fonction définie sur une structure mathématique particulière à toute sous-structure de celle-ci. Diverses autres utilisations plus sophistiquées de l'héritage sont possibles.
  Le second outil est un outil de *sous-termes implicites* où certains paramètres polymorphiques peuvent être omis. L'exemple classique est celui de la composition des fonctions $f : A \to B$ et $g : B \to C$, qui se note *formellement* $(Comp\ A\ B\ f\ C\ g)$. Or les termes A, B et C peuvent être reconstruits à partir des types de f et g, ils sont alors omis; on obtient alors la notation usuelle $(Comp\ f\ g)$. Avec une notation infixe, nous aboutissons à la notation usuelle $f \circ g$. Notre contribution pour cet outil est plus modeste; nous avons en effet repris un mécanisme existant de Chet Murthy que nous documentons et étendons dans cette thèse.

- L'outil de démonstration consiste en une librairie de réécriture où il est possible de décrire et de combiner des stratégies de réécriture. Ces tactiques peuvent être utilisées pour démontrer des équations entre termes quelconques dans une théorie équationnelle quelconque.

Loin de nous contenter de la simple définition de ces outils, nous avons tenu à mettre en évidence leur apport en les utilisant de manière intensive dans la formalisation d'une partie conséquente de la théorie des catégories. Nous pensons que sans ces outils, un tel développement serait impossible à réaliser. C'est un problème difficile, notamment dans le contexte d'une théorie des types intensionnelle (c'est notre cas). Plusieurs formalisations de la théorie des catégories dans différents systèmes ont été réalisées dans le passé, mais la nôtre est la plus complète à ce jour. Nous prouvons ainsi de manière «constructive» qu'un tel développement est possible dans une théorie de types intensionnelle. À travers ce développement, nous avons aussi dégagé une méthodologie basée sur la formalisation des structures mathématiques comme enregistrements dépendants, et sur le langage des Setoïdes comme modélisation de la théorie des ensembles.

Le premier chapitre de cette thèse est consacré à la présentation de la problématique de la formalisation des mathématiques dans les systèmes de preuves.
Après une rapide présentation des différents systèmes de types purs au chapitre 2, le chapitre 3

décrit quelques extensions pour ces théories.
Les outils de modélisation et de démonstration sont ensuite décrits dans les chapitres 4, 5 et 6. Plus précisement, les chapitres 4 et 5 sont respectivement consacrés aux mécanismes de modélisation de sous-termes implicites et de coercions implicites. Enfin dans le chapitre 6, nous présentons une bibliothèque de tactiques de réécriture.
Les chapitres suivants sont des chapitres d'applications, et sont consacrés à la formalisation de la théorie des catégories dans Coq. Le chapitre 7 rappelle la syntaxe de Coq et décrit le langage des Setoïdes. Les notions de base (catégories, foncteurs et transformations naturelles) sont formalisées dans le chapitre 8. Dans le chapitre 9, nous abordons des concepts plus avancés (limites et adjonctions), et nous vérifions le théorème de Freyd pour l'existence d'un adjoint gauche[61]. Le dernier chapitre de cette thèse est consacré à l'analyse de ce développement, et à la comparaison avec d'autres formalisations de la théorie des catégories dans d'autres systèmes de preuves.

Chapitre 1

Formalisation des Mathématiques

1.1 Introduction

Nous pouvons diviser le projet de formalisation des mathématiques en deux parties:

- formaliser les énoncés des théorèmes, et le contexte dont ils dépendent: définitions, hypothèses, axiomes, etc.
- formaliser les preuves des résultats et les vérifier avec précision.

Ces deux parties sont d'habitude considérées ensemble, mais énoncer formellement les théorèmes sans les prouver est en soi une chose intéressante. En pratique, nous sommes aussi intéressés par la vérification de la correction des preuves.

La formalisation des mathématiques est importante pour des raisons de correction du raisonnement mathématique. Bien que les preuves mathématiques soient plusieurs fois révisées avant publication, beaucoup de preuves se sont révélées fausses, parfois bien longtemps après leur publication. Ces erreurs peuvent être évitées par l'adoption de l'approche axiomatique (Logique). Ainsi la rigueur en mathématiques est établie par la donnée d'un langage précis pour énoncer des propositions logiques et des règles précises définissant les inférences et les preuves valides. Dans l'approche axiomatique, tous les théorèmes doivent être obtenus à partir des *axiomes* par des étapes (inférences) logiques.
Ce chapitre est organisé comme suit. La première section est consacrée à un bref historique de la logique. La section 2 concerne les théories de fondement des mathématiques (théories dans lesquelles une formalisation peut être menée). La section suivante décrit sommairement les particularités des *mathématiques constructives*, une conception particulière des mathématiques plus adaptée à la formalisation sur ordinateur. Puis nous enchaînons sur les *systèmes de démonstration de théorèmes* sur ordinateur. Enfin nous discutons les principales formalisations réalisées jusqu'à maintenant et les difficultés d'une telle activité, particulièrement sur ordinateur.
Nous encourageons le lecteur à consulter la monographie de Harrison[70] sur ce sujet.

1.2 Logique

On attribue souvent l'introduction de la notion de preuve formelle aux grecs anciens. L'exemple le plus cité de formalisation dans cette période est celui de la formalisation de la géométrie par Euclide

(280 av. J.C.). Pendant longtemps, les «Éléments» d'Euclide restèrent un exemple de formalisme. En effet, les mathématiciens étaient plus concernés par l'investigation des nouvelles branches mathématiques telles que l'algèbre, que par la construction de preuves dans le style d'Euclide. Il faut attendre la fin du $19^{\text{ème}}$ siècle pour qu'une attention plus grande soit accordée à la formalisation des mathématiques. Cet intérêt fut principalement suscité par l'émergence de géométries alternatives, dites «non-Euclidiennes», rejetant le «postulat des parallèles» d'Euclide. Certains mathématiciens, appelés formalistes, ont commencé à penser que les axiomes devaient être considérés comme (en principe) des créations arbitraires, et non pas des «vérités évidentes», comme les appelaient Euclide, provenant du monde physique qui nous entoure. Le problème de relier les théorèmes déduits de ces axiomes formels, avec le monde physique est alors une question externe aux mathématiques.

À la même époque, la nature même de la déduction était à l'étude. Bien que les grecs – Aristote notamment – considéraient les «syllogismes» comme des moyens fiables de déduire des conclusions à partir des prémisses, il n'y a pas eu de tentative particulière de réduire le raisonnement mathématique aux syllogismes. Ainsi, les déductions utilisées dans les Éléments d'Euclide étaient considérées comme «évidemment vraies», sans que cette intuition ne soit formalisée. Un premier pas vers ce travail fut l'œuvre de Gottfried Leibniz (1646-1716), qui tenta de développer un calcul scientifique, et un calcul logique où on peut juger de la prouvabilité des propositions au seul vu des symboles, et en utilisant une méthode analytique systématique. La formalisation la plus importante de la logique, est cependant celle entreprise par Gottlob Frege (1848-1925). Le but de Frege était de reconstruire les mathématiques de manière logique, notamment l'arithmétique. Il a fondé pour cela le calcul des prédicats – les notations qu'il avait utilisé sont fort différentes de celle adoptées aujourd'hui. La tentative de dériver les mathématiques à partir de principes logiques de base, en évitant les axiomes arbitraires, a été poursuivie par Alfred North Whitehead (1861-1947) et Bertrand Russel (1872-1970) dans les trois volumes de leur *Principia Mathematica*[157], publiés entre 1910 et 1913. Ils commencèrent par postuler les règles basiques de la logique, comme la règle «modus ponens». De ces règles, d'autres principes logiques ont été déduits comme théorèmes. Une théorie des ensembles et des classes a été alors développée, et les nombres sont définis en utilisant les classes.

Pour les formalistes, une preuve formelle est (en principe) une séquence finie de formules où la dernière formule est le théorème à prouver, et chaque formule de la séquence est soit un axiome, soit une formule dérivée des formules précédentes par des règles d'inférence logiques. Ces règles sont formelles, dans le sens où elles opèrent syntaxiquement sur les formules considérées simplement comme des séquences de symboles; leur application ne dépend pas du sens de ces symboles. De cette manière, le formalisme tentait d'échapper aux douteux arbitrages de l'intuition humaine. Cependant, si les axiomes sont des créations arbitraires, plutôt que des «vérités» sur le monde, il n'y a a priori aucune garantie qu'ils soient mutuellement cohérents. La cohérence est une précondition indispensable pour avoir des inférences valides, sinon on peut avoir à la fois la preuve d'une proposition et de sa négation. Ainsi les preuves de cohérence des systèmes formels (ensemble d'axiomes avec des règles d'inférence) sont pour les formalistes, un résultat clé. C'est sur ce point que le «programme formel» a rencontré sa difficulté la plus connue: Kurt Gödel a démontré en 1931 que tout système assez riche pour exprimer l'arithmétique ne peut être prouvé cohérent que dans un système plus grand, pour lequel la cohérence deviendrait alors à son tour problématique. C'est un corollaire du fameux «théorème d'incomplétude» de Gödel, énonçant que tout système formel fini adéquat à la représentation de l'arithmétique est forcément incomplet, c'est-à dire il existe une proposition telle qu'il n'existe pas de preuve ni pour cette proposition ni pour sa négation.

Turing étudia le problème, d'abord abordé par Hilbert, de la décidabilité des mathématiques, i.e. de savoir s'il existait un algorithme ou une procédure mécanique qui appliquée à une proposition mathématique arbitraire, déterminait en un nombre fini d'étapes la véracité de la proposition. Turing a pour cela rendu précise la notion de «procédure mécanique», en imaginant ce qui est maintenant connu sous le nom de «machine de Turing» – un ordinateur simplifié répondant par des symboles sur une bande. Le résultat de Turing fut négatif: il n'existe pas de procédure mathématique permettant de résoudre tous les problèmes mathématiques – cependant des «procédures de décision» existent pour des systèmes formels particuliers (limités).

Les résultats négatifs de Gödel et de Turing montrèrent les limites du programme formel de Hilbert. Ces recherches ont toutefois permis l'éclosion d'une nouvelle branche des mathématiques: la théorie de la démonstration; en effet grâce à la définition formelle de la notion de preuve, il est dorénavant possible de raisonner mathématiquement sur les preuves. Les théorèmes de Gödel comptent parmi les contributions les plus importantes de cette théorie, à côté d'autres résultats comme le théorème d'Herbrand.

1.3 Fondements des mathématiques

Dans la plupart des disciplines, et en mathématiques, certaines théories sont considérées comme plus fondamentales que d'autres, dans le sens où elles servent de base pour d'autres développements de ce domaine. Certaines de ces théories, à cause notamment de leur degré élevé d'abstraction, constituent un fondement pour toutes les mathématiques.
Les théories les plus connues servant de fondement aux mathématiques sont les théories des ensembles, en particulier la théorie Zermelo-Fraenkel (ZF). Une alternative plausible est la *théorie des types* dont l'origine remonte aux travaux de Russel; diverses versions ont été depuis développées (voir le chapitre 2 de cette thèse). Ces deux théories correspondent à deux réactions différentes aux paradoxes de la théorie naïve des ensembles. Dans ZF, la formation des ensembles valides est restreinte, et est régie par des axiomes exprimés dans un langage du premier ordre. Russel, quant à lui, évite les paradoxes par l'introduction de la notion de type et par la stratification des objets mathématiques en niveaux; un objet de niveau n ne peut appartenir qu'à un objet de niveau $n+1$. La différence majeure entre les deux théories est qu'en théorie des types, la notion de fonction est primitive, alors qu'en théorie des ensembles, elle n'est qu'un cas particulier de la notion de relation. Il s'ensuit que la théorie des types est mieux adaptée aux raisonnements mettant en jeu des calculs; par exemple une preuve de $P(5)$ est aussi une preuve de $P(3^2-4)$. Dans certains cas, elle semble même plus naturelle pour représenter les mathématiques que la théorie des ensembles. En effet la notion de type est naturelle chez les mathématiciens; nous pensons tous aux nombres réels comme des objets différents des nombres complexes. Or en théorie des ensembles, tous les objets sont au même niveau, et il est alors légitime de poser des questions du genre «$0 \in 1$?». En théorie des types, ces questions ne peuvent pas être formulées.

Les deux principales tentatives de formalisations des mathématiques sont, à notre connaissance, celles de Bourbaki en théorie des ensembles et de Russel et Whitehead en théorie de types.
Les objectifs de Bourbaki[26] (un prête-nom pour un groupe de prestigieux mathématiciens français) étaient de «fournir un solide fondement pour l'ensemble des mathématiques modernes», et de «dériver l'ensemble des mathématiques connues d'une seule source, la théorie des ensembles». La théorie des ensembles utilisée est une variante de ZFC, c'est à dire ZF étendue avec l'axiome du choix. Leur travail, élégant et un peu austère comprend de très grandes parties des mathématiques.

Il est toujours la référence absolue pour les mathématiciens pures, car il a su éviter le langage pédant et ennuyeux des textes formels (voir la discussion dans la section 1.6.2).
Dans «Principia Mathematica»[157], Russel et Whitehead ont montré que d'importantes parties du raisonnement mathématique pouvaient être écrits dans une logique complètement formelle. Le résultat est encore plus formel que la formalisation de Bourbaki. Cependant cet effort n'a pas été poursuivi, et semble même avoir découragé les tentatives pratiques de formalisation des mathématiques.

1.4 Mathématiques constructives

La théorie des types est une alternative plausible pour la formalisation des mathématiques, notamment des *mathématiques constructives* (par opposition aux mathématiques *classiques*), dont le défenseur principal fut le mathématicien hollandais Brouwer (1881-1961). La principale particularité des mathématiques constructives est le rejet de l'une des règles du raisonnement classique: le tiers-exclus qui permet de démontrer «P ou non P» sans démontrer ni la propriété P ni sa négation. Les mathématiques constructives distinguent aussi entre différentes techniques de preuves, considérées («classiquement») comme équivalentes. Par exemple, pour prouver $\exists x.P(x)$, il est d'habitude suffisant de prouver l'impossibilité de la proposition $\forall x.\neg P(x)$ (le symbole $\neg$ correspond à la négation). Les constructivistes rejettent cette technique; une preuve constructive de $\exists x.P(x)$ consiste en un objet a avec une preuve de $P(a)$. Plus fondamentalement, c'est la conception même de vérité qui oppose les deux écoles, classiques et constructives. En mathématiques classiques, les propositions sont vraies ou fausses indépendemment de ce que nous en savons, alors que pour les constructivistes, ce sont les démonstrations qui donnent leur vérité aux propositions.
Au début du siècle, et après que plusieurs résultats importants aient été prouvés non-constructifs, beaucoup de mathématiciens se sont détournés des mathématiques constructives. Cependant l'intérêt pour les mathématiques constructives a été ravivé pour des considérations informatiques. En effet, elles fournissent un moyen de considérer le langage des propositions comme un langage de spécification pour les programmes informatiques; les preuves de propositions sont alors des programmes. Par exemple, une preuve de $\forall x.\exists y.P(x,y)$ est une méthode uniforme devant être exhibée, et construisant un objet y tel que $P(x,y)$ pour un x donné arbitraire. Une telle méthode est une fonction récursive, ou un programme.

Une excellente introduction aux mathématiques constructives est donnée dans Troelstra et van Dalen[145] et Dummett[58].

1.5 Automatisation des preuves sur ordinateur

Ces dernières décennies, le domaine de recherche consacré à l'automatisation des preuves sur ordinateur, a connu un développement important. Ces recherches ont donné lieu à la construction de plusieurs *systèmes de démonstration de théorèmes* (SDT en abrégé, et Theorem provers en anglais). Nous regroupons sous ce nom (ou encore celui de *système de preuve*) des systèmes fort différents comprenant les systèmes de preuves automatiques, les vérificateurs de preuves et les systèmes de développement de preuves. Presque tous ces systèmes manipulent les notions de définition, de théorème et de preuve, analogues à celles qu'emploient les mathématiciens. Ils ont aussi une sémantique précise et sont implémentés avec soin afin de s'assurer qu'ils sont conformes à leur

sémantique.

Systèmes automatiques. Les premières tentatives d'automatisation des preuves formelles coïncidèrent avec l'avènement de l'Intelligence Artificielle. Ces tentatives avaient pour but la conception de machines intelligentes, imitant le raisonnement humain.

Le premier SDT, appelé «Logic Theory Machine», fut construit par Allen Nevell et Herbert Simon en 1955. Pour la petite histoire, ils voulaient initialement construire une machine à jouer aux échecs, mais durent se «contenter» d'un SDT devant la difficulté de la tâche. Leur machine prouvait des théorèmes dans le calcul propositionnel, et était complètement automatique. Elle était programmée pour chercher une preuve en une étape du théorème à prouver. Si une telle preuve n'est pas trouvée, un ensemble de buts était engendré (formules desquelles le théorème pouvait être déduit en une étape), et les preuves pour ces buts étaient à leur tour recherchées, et ainsi de suite récursivement jusqu'à épuisement des buts. En plus des axiomes (proches de ceux de Principia), la machine avait la possibilité d'utiliser les théorèmes précédemment démontrés. Cette première machine, bien que rudimentaire, a permis de mettre en évidence un problème central des SDT: l'explosion combinatoire de l'espace de recherche.

Une autre approche, plus enracinée dans la logique, ne tarda pas à faire son apparition. Une solution aux difficultés rencontrées par les SDT dans les premières années a été d'abandonner la tentative de les faire complètement automatiques, et de rendre possible l'intervention humaine pour guider le SDT dans la recherche de preuve; ce sont les *systèmes interactifs semi-automatiques.*

Vérificateurs de preuves. Parmi ces nouveaux systèmes, nous distinguons les *Vérificateurs de Preuves* (Proof Checkers en anglais). La preuve, construite par un humain, était fournie, et ils se contentaient de vérifier sa validité. Les plus fameux de ces systèmes ont été Automath[115] et Mizar[135]. Dans ces deux systèmes, un langage pour les mathématiques formelles a été conçu, ainsi qu'un programme pour vérifier la correction des textes formels. Ils constituent ainsi une réponse aux regrets des formalistes concernant le manque de preuves mathématiques complètement formelles. Malheureusement, de tels systèmes sont encore très peu utilisés par les mathématiciens.
Automath a été développé entre 67 et 79 par N. G. de Bruijn à Technishe Hogeschool (Eindhoven) mais n'est plus utilisé aujourd'hui. Des parties significatives des mathématiques ont été vérifiées; par exemple, van Benthem Jutting[153] a formalisé en 1977 le livre de Landau (1930) sur la construction du corps des nombres réels.
Quant à Mizar, il a été développé en 1978 par A. Trybulec, et est encore très vivace. Il est basé sur une variante de ZF, la théorie des ensembles de Tarski-Grothendieck. Il a été utilisé pour vérifier de grandes parties des mathématiques, comprenant la théorie des ensembles, l'algèbre, l'analyse, la topologie, la théorie des catégories et plusieurs applications concernant l'informatique.

Systèmes de développement de preuves. Nous commençons par un SDT interactif influent, le système américain Boyer-Moore ou NQTHM (New Quantified Theorem Prover) créé en 1979. Son point fort était l'utilisation de la récurrence mathématique, pour laquelle il fournissait de nombreuses heuristiques pour le choix des paramètres sur lesquels faire la récurrence , et le choix des hypothèses de récurrence. La logique de NQTHM était très simple, plus simple que le calcul des prédicats du premier ordre, à l'exception des règles de récurrence nécessitant des quantifications sur les prédicats. NQTHM fournissait aussi la possibilité de définir des fonctions récursives. Quoique plus automatique que la plupart des autres SDT interactifs, NQTHM nécessitait une aide humaine

experte pour les preuves difficiles. Des preuves conséquentes ont néanmoins été menées à bien dans NQTHM, comme celle du théorème d'incomplétude de Gödel par Natarajan Shankar en 1980.

La vérification de preuve de bas niveau à la Automath est insatisfaisante car trop pénible et le résultat est souvent illisible. Les SDT semi-automatiques, comme NQTHM, sont difficiles à utiliser en pratique; en effet, ils sont guidés vers le résultat recherché par une série de lemmes intermédiaires que l'utilisateur doit soigneusement choisir. Ces inconvénients ont conduit Robin Milner à adopter une nouvelle approche.
Cette approche, moins automatique, est représentée par LCF(Logic for Computable Functions) développé par Robin Milner d'abord à Stanford en 1972, puis à l'université d'Edinburgh, dans le but de mécaniser les preuves dans un formalisme de description de fonctions récursives partielles appelé PPLambda. Avec Edinburgh LCF (1977), Milner introduisit le «Méta-langage» ML, dans lequel l'utilisateur pouvait écrire des procédures de preuves. Parmi les types de ML, il y avait le type `thm` des théorèmes. Les objets de ce type ne pouvaient être construits dans LCF qu'en utilisant les règles d'inférence de PPLambda aux objets existants de type `thm`, lesquels incluent les axiomes définis par l'utilisateur. La correction de LCF ne dépendait alors que de la partie codant les axiomes et les règles d'inférence. La première motivation de Milner pour la conception de ML était son utilisation pour écrire des «moyens» pour l'exécution de preuves, comprenant les *tactics* – programmes pour la réduction d'un but en une liste de sous-buts tels que si chaque sous-but était démontré, on pouvait construire une preuve pour le but original –, et les *tacticals* pour combiner les tactics.

Le style LCF a été adopté par plusieurs autres SDT interactifs, comme Nuprl[42] développé par R. L. Constable et ses collègues. Ce système n'utilise pas la logique classique de Hilbert, mais une théorie des types *constructive*. Nuprl fournit aussi une approche alternative pour la preuve de correction de programmes: plutôt que d'écrire un programme et d'essayer de prouver sa correction, Nuprl commence par construire une preuve (constructive) de l'existence de l'objet mathématique désiré, puis synthétise automatiquement un programme qui va engendrer cet objet.

Les logiques d'ordre supérieur sont souvent aussi implémentées dans un style ML. En logique d'ordre supérieur, la quantification n'est pas restreinte aux variables simples comme en logique du premier ordre, elle est de ce fait plus expressive. Les preuves par contre sont plus difficile à automatiser, ainsi les algorithmes d'unification sont souvent non-terminants. La principale motivation pour l'intérêt récent porté aux SDT d'ordre supérieur vient de la spécification du matériel, où certaines quantifications nécessitent des quantifications du deuxième ordre. Dans ce cadre, le système le plus influent a été HOL de Michael J. C. Gordon (qui a travaillé avec R. Milner sur Edinburgh LCF), implémenté sur Cambridge LCF. Les preuves en HOL nécessitent une intervention humaine intense. Dans les dernières années, la communauté des utilisateurs de HOL s'est considérablement développée, rendant HOL l'un des SDT les plus utilisés au monde. Un autre descendant de LCF est Isabelle[122], un système de preuves générique où des logiques peuvent être spécifiées puis utilisées. Il existe des systèmes de preuves implémentant des logiques d'ordre supérieure, sans adopter pour autant l'approche de LCF. C'est le cas, par exemple, du système PVS[118] qui remporte d'ailleurs un franc succès grâce à une importante collection de procédures de décision puissantes. Contrairement aux tactiques dans l'approche LCF, les procédures de preuves de PVS n'ont pas de garantie de correction.

Un traitement plus complet de la question se trouve dans [105].

1.6 Pratique de la formalisation des mathématiques

1.6.1 Expériences

La formalisation des mathématiques n'est devenue réalisable qu'avec l'avènement de l'ordinateur et des systèmes de preuves. De tels systèmes sont capables de vérifier la bonne formation des définitions et la correction des preuves, déchargeant ainsi les mathématiciens de ces activités pénibles. De plus, ils peuvent contribuer à constuire des preuves; ils peuvent démontrer automatiquement certains théorèmes.

Nous avons déjà parlé dans la section 1.3 des expériences de Bourbaki et de Russel. Nous nous concentrons dans la présente section sur les expériences réalisées sur ordinateur. L'utilisation des systèmes de preuve a été plus importante en informatique. Dans cette discipline, il y avait une motivation pratique à l'intérêt porté aux SDT interactifs: prouver la correction (relativement à une spécification) de programmes informatiques, et dans une moindre mesure du matériel informatique. Il existe toutefois un nombre considérable d'expériences de formalisation des mathématiques dans différents systèmes de preuves. Ces expériences sont malheureusement disparates et difficiles à comparer car elles mettent en jeu des systèmes, des sémantiques, et des logiques différentes. Un projet ambitieux (mais un peu utopique), QED[10], a pour objectif de regrouper toutes ces énergies pour constituer une encyclopédie des mathématiques formalisées.
Récemment, plus de notions mathématiques ont été formalisées dans Mizar que dans aucun autre système. Tout ce travail est regoupé en articles dont le nombre dépasse maintenant 300 et contenant environ 6000 théorèmes. Ces articles sont automatiquement compilés en LaTeX et publiés dans *Journal of Formalised Mathematics* (`http://math.uw.bialystok.pl/~Form.Math`). Le succès de Mizar tient à la volonté affichée de ses concepteurs de ne pas s'éloigner des pratiques usuellement acceptées en mathématiques. D'abord, le choix de la théorie des ensembles comme fondement, qui est bien plus connue dans la communauté des mathématiciens que la théorie des types. Ensuite l'utilisation de plusieurs concepts de la théorie des types, comme le sous-typage, les types paramétrés et les types de structures (classes).

Les principales autres formalisations récentes sont le développement d'algèbre de P. Jackson[88] dans Nuprl, le projet de formalisation de la théorie de Galois dans LEGO par P. Aczel[1, 13, 17] et le développement d'analyse de J. Harrison[73] dans HOL.

1.6.2 Difficultés et Faisabilité

Il est admis que l'activité de formalisation des mathématiques est difficile, non pas à cause de la longueur des preuves mais de leur manque de précision. Cette imprécision se manifeste à deux niveaux: au niveau des preuves et au niveau du texte mathématique (énoncés). Ainsi beaucoup de preuves mathématiques contiennent des sauts qui ne sont pas justifiés formellement, mais qui sont tout à fait clairs pour les spécialistes du domaine. La seconde difficulté est dûe au manque de précision des textes mathématiques. Bien que les mathématiques soient la science exacte par excellence, le langage utilisé en pratique par les mathématiciens (dans les livres, articles etc.) peut être remarquablement vague. Ainsi les textes mathématiques contiennent invariablement une partie substantielle de langage naturel, avec tout le potentiel habituel d'ambiguïté et d'imprécision. Il peut arriver que seul le lecteur averti puisse distinguer le contenu réel d'un texte, et apprécier le contexte dans lequel les résultats sont énoncés. Or ces *abus de notation* sont très importants en mathématiques, et sans eux tout texte mathématiques court le risque de devenir illisible, comme le

souligne Bourbaki[26].

> «Si la mathématique formelle était aussi simple que les jeu d'échecs, il n'y aurait plus qu'à rédiger nos démonstrations dans ce langage [...]. Mais les choses sont loin d'être aussi faciles, et point n'est besoin d'une longue pratique pour s'apercevoir qu'un tel projet est complètement irréalisable; la moindre démonstration du début de la Théorie des Ensembles exigerait déjà des centaines de signes pour être formalisée...»

Ainsi pour réduire la taille des textes formels et pour les rendre plus lisibles, Bourbaki propose d'inclure dans le langage formel certains abus de notations similaires à ceux rencontrés en mathématiques informelles.

> «Dès le livre 1 de ce traité s'impose donc la nécessité impérieuse d'abréger le texte formalisé par l'introduction de mots nouveaux [...] et des règles de syntaxe additionnelles [...]. Ce faisant, on obtient des langages beaucoup plus maniables que le langage formalisé proprement dit...»

Ces abus de notations doivent être toutefois justifiés et le passage de ce langage vers le langage complètement formel doit se faire de manière mécanique. Sinon on risque d'introduire des ambiguïtés, voire des erreurs.

> «Nous abandonnons donc très tôt la mathématique formalisée, mais non sans avoir pris soin de tracer avec précision le chemin par lequel on y pourrait revenir. Les facilités qu'apportent les premiers «abus de langage» ainsi introduits nous permettront d'écrire le reste de ce traité [...] comme le sont en pratique tous les textes mathématiques.»

Certaines conventions vont cependant plus loin que les simples abus de notations. Elles ne peuvent pas dans ce cas intégrer le langage formel. Les textes de Bourbaki n'y renoncent malheureusement pas.

> «Souvent même on se servira même du langage courant d'une manière bien plus libre encore, par des abus de langages volontaires, par omission pure et simple des passages qu'on présume pouvoir être restitués par un lecteur tant soit peu exercé...»

Malgré l'utilisation d'abus de notations, les textes formels restent encore souvent illisibles. Ceci est souvent dû au fait que les preuves formelles évacuent l'intuition. Or celle-ci, ainsi que le processus de découverte, sont parfois essentiels à la compréhension d'une preuve. C'est pourquoi une preuve formelle doit toujours être accompagnée d'une discussion explicative informelle (comme les programmes doivent contenir des commentaires).

Première partie

Théorie des types

Introduction

Cette partie est consacrée à la description d'une famille de théories appelée *théorie des types* (ou λ-calculs typés), que nous avons l'intention d'utiliser comme fondements pour les mathématiques (constructives). Dans le premier chapitre, nous nous efforçons de mettre en évidence l'apport de chacun des systèmes de types introduits. Cet apport est matérialisé par l'ensemble des fonctions représentables dans une théorie et la capacité de représentation de proposition logiques.
Le second chapitre concerne les aspects algorithmiques de ces théories, indispensables pour leur utilisation dans un cadre informatique. Nous verrons ainsi comment vérifier qu'une preuve est une preuve correcte pour une proposition; cet algorithme dépend étroitement de celui d'inférence de types, problème bien connu pour les langages de programmation. Ce chapitre introduit aussi quelques extensions (constantes, hiérarchies d'univers, types inductifs et enregistrements), adoptées par la quasi-totalité des systèmes de preuves.

Chapitre 2

Systèmes de types purs

La théorie des types (ou λ-calcul typé) est actuellement une discipline connaissant un développement important. L'introduction des *types* en mathématiques remonte au début du siècle, lorsque B.A.W. Russel et A.N. Whitehead les ont incorporés dans leur fameux Principia Mathematica (1910-1913) pour éviter les paradoxes. Le développement de la théorie des types a été depuis stimulé par l'informatique où les types jouent un rôle important, notamment dans les langages de programmation. Il y a une variété de *systèmes de types*, chacun dédié à une utilisation particulière.

Plusieurs systèmes d'aide à la démonstration sont basés sur des théories de types. Parmi ces systèmes, nous citons Coq et LEGO implémentant des variantes du Calcul des Constructions (plus précisément, CCI pour Coq [49] et ECC pour LEGO [104]), et NuPRL et ALF implémentant des théories dites de Martin-Löf. Les difficultés à raisonner à propos de calculs en théorie des ensembles semblent avoir favorisé le développement des théories de types où la fonction a un statut particulier, contrairement à la théorie des ensembles où elle est un cas particulier des relations. Ces théories sont des extensions du *λ-calcul typé* où les preuves sont représentées à travers l'isomorphisme de Curry-Howard.

2.1 λ-calcul pur

Le λ-calcul (originellement non typé) fut inventé par A. Church dans les années 30. Il l'utilisa pour formaliser la notion de fonction effectivement calculable. Il est maintenant établi que ce formalisme est équivalent aux machines de Turing et aux systèmes d'équations de Kleene-Gödel. Le λ-calcul peut être vu comme un langage de programmation permettant la définition de fonctions par la donnée d'algorithmes, et contenant une méthode d'évaluation (ou calcul) de ces algorithmes. Il s'agit là d'une notion intensionnelle de fonction, s'opposant à la notion classique de graphe de fonction comme ensemble de paires (argument,résultat).

Les termes du λ-calcul sont inductivement définis. Il y a d'abord un ensemble de variables V. Chaque variable est elle-même un λ-terme. Puis, nous avons deux principes de base de constructions de termes qui sont l'*abstraction* et l'*application*.

L'abstraction combine un terme a et une variable x en un nouveau terme $[x]a$ (la notation originale est $\lambda x.a$). La variable x est dite *muette* car elle ne sert qu'à indiquer des positions liées dans a. Cette notation correspond à la notation usuelle de fonction $x \mapsto a$. Ainsi par exemple, $[x]x$ dénote la fonction identité.

L'application est une construction plus simple, correspondant à l'application de fonctions. Le terme $(a\ b)$ exprime l'application de la fonction a à l'argument b. La notation dérive de celle usuelle $a(b)$. Remarquez que le λ-calcul est très libéral, $(x\ y)$ est un λ-terme légal bien que x n'ait pas l'apparence d'une fonction. De plus, les abstractions peuvent apparaître à l'intérieur d'un terme.

Définition 2.1 (Termes du λ-calcul pur)

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

avec $x \in V$.

Notation 2.1

- *Nous notons par* $\equiv$, *l'égalité syntaxique entre différentes entités (termes...).*
- *Pour plus de commodité, nous noterons l'application* $(...((a\ b_1)\ b_2)...b_n)$ *par* $(a\ b_1\ b_2...b_n)$.

Lorsque a a la forme d'une abstraction, i.e. est de la forme $[x]a'$ pour une certaine variable x et un certain terme a', $(a\ b)$ donne alors plus d'information sur le résultat attendu. Ainsi $([x](x\ y)\ z)$ doit avoir $(z\ y)$ comme résultat. Il y a alors une relation entre $([x](x\ y)\ z)$ et $(z\ y)$, appelée β-réduction.

Définition 2.2 (Variable libre et variable liée)

- *Toute occurrence de la variable* x *dans le terme* $[x]a$ *est liée. Une variable est liée dans un terme si elle a au moins une occurrence liée dans ce terme.*
- *L'ensemble des variables libres d'un terme est défini par récurrence:*

$$\begin{aligned} VL(x) &= \{x\} \\ VL([x]a) &= VL(a) \setminus \{x\} \\ VL((a\ b)) &= VL(a) \cup VL(b) \end{aligned}$$

Définition 2.3 (Substitution) *Soient* a *et* c *deux termes et* x *une variable, le terme* $a[x \leftarrow c]$ *est obtenu en substituant le terme* c *à toutes les occurrences libres de* x *dans* a.

$$\begin{aligned} x[x \leftarrow c] &= c \\ y[x \leftarrow c] &= y\ si\ y \not\equiv x \\ ([x]a)[x \leftarrow c] &= [x]a \\ ([y]a)[x \leftarrow c] &= [y](a[x \leftarrow c])\ si\ y \not\equiv x\ et\ y \notin VL(c) \\ ([y]a)[x \leftarrow c] &= [z](a[y \leftarrow z][x \leftarrow c])\ si\ y \not\equiv x, y \in VL(c), z \notin VL(a)\ et\ z \notin VL(c) \\ (a\ b)[x \leftarrow c] &= (a[x \leftarrow c]\ b[x \leftarrow c]) \end{aligned}$$

Remarque 2.1 (α-conversion) La difficulté de la définition de la substitution provient du fait qu'on doit éviter les captures de variables. Ainsi par exemple, pour substituer une variable x à y dans $[x](x\ y)$, on doit d'abord renommer la variable liée x en une nouvelle, disons z, pour obtenir enfin $(z\ x)$. Les termes $[x](x\ y)$ et $[z](z\ y)$ sont considérés comme identiques. Plus généralement, $[x]a$ et $[z]a'$ sont dits identiques si a et a' ne diffèrent que par le point suivant: à chaque occurrence libre de x dans a, nous avons une occurrence libre de z dans a' à la même position, et vice versa. Cette relation est appelée α-*conversion*. La formulation dite de de Bruijn permet d'éviter ces tracas. Dans cette formulation, les variables liées sont remplacées par des *indices de de Bruijn*[115]; l'idée est que toute variable liée peut-être codée par sa hauteur de liaison (le nombre de λ que l'on rencontre avant d'atteindre le λ lieur). Par exemple, $[x][y](x\ y)$ devient $\lambda\lambda(1\ 0)$.

Définition 2.4 (Réduction)

- *La relation de β-réduction en une étape $\rightarrow_\beta$ est la plus petite relation telle que:*
  - $([x]a\ b) \rightarrow_\beta a[x \leftarrow b]$,
  - *si* $M \rightarrow_\beta N$ *alors* $(M\ b) \rightarrow_\beta (N\ b)$, $(a\ M) \rightarrow_\beta (a\ N)$ *et* $[x]M \rightarrow_\beta [x]N$.
- *La relation de β-réduction $\rightarrow_\beta^*$ est la plus petite relation réflexive et transitive contenant la relation $\rightarrow_\beta$.*
- *La relation de β-équivalence $=_\beta$ est la plus petite relation d'équivalence contenant $\rightarrow_\beta$.*
- *Un terme M est dit normal s'il n'existe pas de terme N tel que $M \rightarrow_\beta N$.*

Il existe une autre règle de réduction qui est la η-réduction.

Définition 2.5 (η-réduction)

$$[x](a\ x) \rightarrow_\eta a \ \textit{si}\ x \notin VL(a)$$

2.2 λ-calcul simplement typé

Dans le λ-calcul pur, rien n'est dit concernant le domaine d'appartenance des variables apparaissant dans un terme. Ceci a des effets indésirables: les calculs (formalisés par la β-réduction) peuvent ne pas terminer. C'est notamment le cas de $([x](x\ x)\ [x](x\ x))$. Ceci est dû au fait que les règles de formation des λ-termes sont trop laxistes. En effet la règle d'application permet d'appliquer n'importe quel terme t à n'importe quel terme u, sans s'assurer que t est bien une fonction et u un objet du domaine de cette fonction. Une façon d'y remédier est d'associer à chaque objet un type.

Il existe deux manières d'ajouter des types au λ-calcul pur. Étant donné un λ-terme pur, on cherche à lui associer un type, s'il existe; ou mieux: on essaie de trouver son type le plus général, ou *schéma de type principal*[98]. Ce typage est appelé *typage implicite* ou *typage à la Curry*.

Une autre approche, appelée *typage explicite* ou *typage à la Church*, consiste à insérer des informations de typage dans le terme lui-même, lors de sa construction. Dans un tel système, à chaque variable liée (suivant un λ) est associé un type. Nous adoptons cette approche car les systèmes à la Church sont plus adaptés à l'implémentation sur ordinateur.

Les types sont formés grâce à deux règles:

- les constantes de types (*nat*, *bool*, etc.) sont des types,
- si A et B sont des types, alors $A \rightarrow B$ est un type. Il correspond au type des fonctions de A dans B.

Définition 2.6 (Termes et types du λ-calcul simplement typé)

$$\begin{array}{lll} T & ::= & x \mid [x:U]T \mid (T\ T) \\ U & ::= & c \mid U \rightarrow U \end{array}$$

où $c \in C$, l'ensemble des constantes de types.

Notation 2.2 (Séquences (ou listes) d'objets) *Nous notons par* $[a_1; ...; a_n]$ *les séquences ordonnées (ou listes) d'objets. L'objet d'ordre* i *est* a_i*. La séquence vide est notée par* $[]$*.*

Les opérations suivantes sont définies pour les séquences:

- *Le prédicat d'appartenance, noté par* $\in$*, est défini comme pour les ensembles:* $a \in \Gamma$ *si et seulement si* a *est un objet de la séquence* Γ*.*
- *L'ajout d'objets à une séquence peut se faire par son début comme par sa fin. Ainsi:* $a; [a_1; ...; a_n] = [a; a_1; ...; a_n]$ *et* $[a_1; ...; a_n]; a = [a_1; ...; a_n; a]$*.*
- *La séquence* $\Gamma \setminus a$ *est la séquence* Γ *sans la première occurrence de l'objet* a *(si elle existe).*
- *Soit* P *un prédicat. L'expression* $Prem_{P(.)}(\Gamma)$ *désigne le premier objet de* Γ *vérifiant* P *s'il existe (donc* $P(Prem_{P(.)}(\Gamma))$ *est vrai).*
- *La concaténation de séquences, notée* $\cdot$*, est définie par:*

$$[a_1; ...; a_n] \cdot [b_1; ...; b_m] = [a_1; ...; a_n; b_1; ...; b_m]$$

Comme un terme peut contenir des variables libres, on ne peut le typer que si l'on précise d'abord le type de ses variables. Un contexte, généralement désigné par une lettre majuscule grecque (Γ ou Δ) est donc une liste de paires $x : A$ composées chacune d'une variable et d'un type. On désigne par $[]$ le contexte vide, et par $\Gamma; x : A$ le contexte Γ auquel on a ajouté la paire $x : A$. Un jugement de typage est donc décrit par un triplet (contexte, terme, type). On écrit $\Gamma \vdash M : A$ pour dire que M est de type A dans Γ. Nous désignons par $Dom(\Gamma)$ l'ensemble des variables déclarées dans Γ. Enfin, nous désignons par Γ_x le type associé à x dans Γ, il est indéfini si x n'est pas déclaré dans Γ.

Définition 2.7 (Règles du λ-calcul simplement typé)

Soit C *un ensemble de constantes de types.*

- *Prédicat Type pour les expressions de types bien formées.*

$$(\text{TConst})\ \frac{c \in C}{c\ Type}$$

$$(\text{TFun})\ \frac{A\ Type \qquad B\ Type}{A \to B\ Type}$$

- *Jugement* $\Gamma \vdash$ *pour les contextes bien formés.*

$$(\text{CVide})\ \frac{}{[] \vdash}$$

$$(\text{CVar})\ \frac{\Gamma \vdash A\ Type \qquad x \notin Dom(\Gamma)}{\Gamma; x : A \vdash}$$

- *Jugement de typage.*

$$(\textsc{Nom})\ \frac{x \in Dom(\Gamma)}{\Gamma \vdash x : \Gamma_x}$$

$$(\textsc{Lam})\ \frac{\Gamma; x : A \vdash b : B}{\Gamma \vdash [x : A]b : A \to B}$$

$$(\textsc{App})\ \frac{\Gamma \vdash a : B \to A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a\ b) : A}$$

Remarque 2.2 Dans la relation de typage que nous venons de présenter, les problèmes de nommage ont été quelque peu négligés. En effet la règle (CVar) impose que les noms de variables d'abstractions imbriquées soient tous différents. Une opération de α-conversion (remarque 2.1) peut alors être nécessaire avant typage, comme dans le terme $[x : A][x : B]x$, qui doit être renommé par exemple en $[y : A][x : B]x$.

Définition 2.8 (Terme bien typé) *Un terme M est dit bien typé dans un contexte Γ s'il existe un type A tel que $\Gamma \vdash M : A$.*

Nous énonçons les principales propriétés de ce calcul. La normalisation forte signifie que si M est un terme bien typé dans un contexte Γ, alors toute réduction issue de ce terme termine sur une forme normale; et la confluence nous assure que deux réductions issues d'un même terme terminent sur la même forme normale.

Lemme 2.1 (Confluence) *La β-réduction est confluente, i.e. $\forall M, N, P.\ M \to_\beta^* N \ \wedge\ M \to_\beta^* P \Rightarrow \exists Q.\ N \to_\beta^* Q \ \wedge\ P \to_\beta^* Q$.*

Lemme 2.2 (Normalisation forte) *La β-réduction est fortement normalisable sur les types bien typés, i.e. que si M est bien typé, alors il n'existe pas de séquence infinie de réductions de la forme $M \to_\beta N_1 \to_\beta N_2 \to_\beta ....$*

Lemme 2.3 (Préservation du type) *Si $\Gamma \vdash M : A$ et $M \to_\beta^* N$, alors $\Gamma \vdash N : A$.*

2.3 Systèmes de types purs (PTS)

2.3.1 Définitions générales

Le λ-calcul simplement typé et plusieurs de ses extensions peuvent être obtenus comme instances de systèmes de types purs[19] (en anglais, PTS: Pure Type System).

Définition 2.9 (PTS) *Un PTSest défini par la donnée d'un triplet $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$:*

- *un ensemble $\mathcal{S}$ de constantes appelées* sortes,
- *un ensemble $\mathcal{A}$ d'*axiomes *de la forme $\langle s_1, s_2 \rangle$ avec $s_1, s_2 : \mathcal{S}$,*
- *un ensemble $\mathcal{R}$ de* règles *de la forme $\langle s_1, s_2, s_3 \rangle$ avec $s_1, s_2, s_3 : \mathcal{S}$.*

Nous commençons par généraliser le type des fonctions en introduisant les *types dépendants*. Considérons le type $A \to B$; dans un système avec types dépendants, le type du résultat doit pouvoir dépendre de la valeur de l'argument $x : A$. Pour rendre compte de cette dépendance, nous adoptons la notation $(x : A)B$ (ou $\Pi x : A.B$) où x est lié, et peut donc apparaître dans B. La notation $A \to B$ devient une abréviation de $(x : A)B$ lorsque x n'apparaît pas dans B.

Nous allons aussi cesser de distinguer entre les termes et les types. Ils sont tous des termes de la théorie; certains termes vont agir comme type de certains autres.

Définition 2.10 (Termes des PTS)

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

Définition 2.11 (Règles des PTS) *Les jugements* $\Gamma \vdash$ *et* $\Gamma \vdash M : A$ *sont définis de manière mutuellement inductive.*

$$(\text{CVide})\ \frac{}{\emptyset \vdash}$$

$$(\text{CVar})\ \frac{\Gamma \vdash A : s \qquad s \in \mathcal{S} \qquad x \notin Dom(\Gamma)}{\Gamma; x : A \vdash}$$

$$(\text{Axiom})\ \frac{\langle s_1, s_2 \rangle \in \mathcal{A}}{\Gamma \vdash s_1 : s_2}$$

$$(\text{Name})\ \frac{x \in \Gamma}{\Gamma \vdash x : \Gamma_x}$$

$$(\text{Lam})\ \frac{\Gamma; x : A \vdash b : B \qquad \Gamma \vdash (x : A)B : s}{\Gamma \vdash [x : A]b : (x : A)B}$$

$$(\text{Prod})\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma; x : A \vdash B : s_2 \qquad \langle s_1, s_2, s_3 \rangle \in \mathcal{R}}{\Gamma \vdash (x : A)B : s_3}$$

$$(\text{App})\ \frac{\Gamma \vdash a : (x : B)A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a\ B) : A[x \leftarrow b]}$$

$$(\text{Conv})\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash M : B \qquad A =_\beta B}{\Gamma \vdash M : A}$$

Cette définition nécessite quelques commentaires. Les sortes peuvent être vues comme des «super-types», par exemple les produits dépendants sont eux-mêmes typés par des sortes. Les produits possibles sont indiqués dans $\mathcal{R}$. L'ensemble $\mathcal{A}$ est quant à lui donné pour typer les sortes (non nécessairement toutes). Du coté des règles de typage, les règles (Lam) et (App) sont similaires à celles des types fonctionnels usuels, mais font apparaître la dépendance lors de l'application. Enfin la règle (Conv) est rajoutée pour nous garantir que deux termes β-équivalents ont les mêmes propriétés vis-à-vis du typage.

Les PTS les plus étudiés sont ceux qui sont dits *fonctionnels*, car ils garantissent l'unicité du type (modulo conversion).

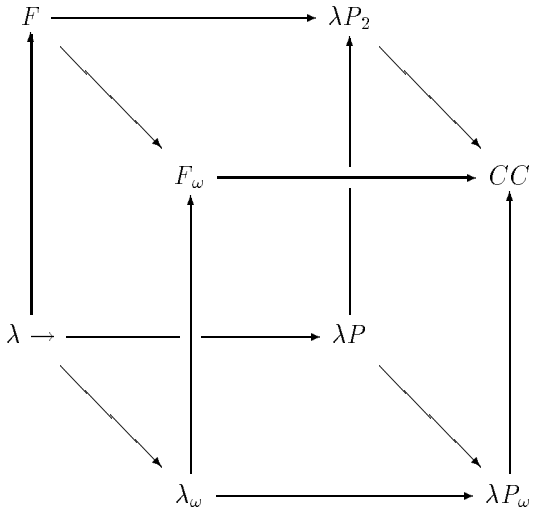
Définition 2.12 (PTS fonctionnels) *Un PTS est dit* fonctionnel *si pour tout $s \in \mathcal{S}$, il existe au plus un $s' \in \mathcal{S}$ tel que $\langle s, s' \rangle \in \mathcal{A}$, et pour tous $s_1, s_2 \in \mathcal{S}$, il existe au plus un $s_3 : \mathcal{S}$ tel que $\langle s_1, s_2, s_3 \rangle \in \mathcal{R}$.*

Lemme 2.4 (Unicité du type pour les PTS fonctionnels)
Si $\Gamma \vdash M : A$ et $\Gamma \vdash M : B$, alors $A =_\beta B$.

PREUVE [19] ■

2.3.2 Cube de Barendregt

Le cube de Barendregt[19] consiste en huit PTS fonctionnels particuliers disposés sur les sommets d'un cube et où les flèches dénotent des inclusions.



Tous ces PTS comportent deux sortes $\mathcal{S} = \{*, \Box\}$, un seul axiome $\mathcal{A} = \{* : \Box\}$, et leur règles sont de la forme $\langle s, s', s' \rangle$ qu'on abrège en $\langle s, s' \rangle$. Nous détaillons quatre de ces systèmes. Les autres systèmes sont λP, λP_ω, λ_ω et λP_2.

- **λ-calcul simplement typé ($\lambda \rightarrow$)** $\mathcal{R} = \{\langle *, * \rangle\}$
  C'est une formulation légèrement différente de celle qu'on a vu précédemment; le terme $*$ est le type de tous les types, mais il n'est pas lui-même de type $*$. Ce qui nous permet d'énoncer la proposition $A\ Type$ de manière plus uniforme par $A : *$. Un autre avantage est qu'on n'a plus à fixer une fois pour toutes l'ensemble des types de base; ils sont introduits au fur et à mesure de nos besoins par $c : *$ en utilisant la règle (CVAR).

- **Système F** $\mathcal{R} = \{\langle *, * \rangle, \langle \Box, * \rangle\}$
  Ce système a été inventé par Girard en 1972 et redécouvert par Reynolds en 74 sous le nom de λ-calcul polymorphe. Il étend le λ-calcul simplement typé par le *polymorphisme*. Ceci permet la formation de termes dépendants de types, comme pour la fonction identité $[A : *][x : A]x$ dont le type est $(A : *)A \rightarrow A$; sans polymorphisme, il aurait fallu définir une fonction identité pour chaque type.

- **Système F_ω** $\mathcal{R} = \{\langle *, *\rangle, \langle \Box, *\rangle, \langle \Box, \Box\rangle\}$
  Le système F_ω généralise le système F en autorisant les fonctionnelles de types, comme dans $list : * \to *$ ou encore la définition du produit de types $[A : *][B : *](C : *)(A \to B \to C) \to C : * \to * \to *$. Un PTS possèdant la règle $\langle \Box, \Box\rangle$ est dit admettre des *constructeurs de types.*

- **Calcul des Constructions CC** $\mathcal{R} = \{\langle *, *\rangle, \langle *, \Box\rangle, \langle \Box, *\rangle, \langle \Box, \Box\rangle\}$
  Enfin, CC rajoute une nouvelle règle $\langle *, \Box\rangle$ permettant de faire dépendre les types des termes. On peut ainsi définir le type des vecteurs de booléen $vectb : nat \to *$ où $(vectb\ n)$ est le type des vecteurs de booléens de longueur n. En utilisant le polymorphisme, on peut définir le type polymorphe $vect : * \to nat \to *$. Le type de la concaténation de vecteurs sera alors $(A : *)(n : nat)(vect\ A\ n) \to (m : nat)(vect\ A\ m) \to (vect\ A\ m + n)$. C'est toute la puissance des types dépendants. Ce système a été inventé par Coquand et Huet en 1985.

Les propriétés simples ci-dessous peuvent être trouvées dans [19].

Lemme 2.5 (Lemmes de Génération)

- $\Gamma \vdash s : X \Longrightarrow \exists s'.\ s' =_\beta X \ \wedge\ \langle s, s'\rangle \in \mathcal{A}$.
- $\Gamma \vdash x : X \Longrightarrow \exists A.\ A =_\beta X \ \wedge\ (x : A) \in \Gamma$.
- $\Gamma \vdash (x : A)B : X \Longrightarrow \exists s_1, s_2, s_3.\ s_3 =_\beta X \ \wedge\ \langle s_1, s_2, s_3\rangle \in \mathcal{R} \ \wedge\ \Gamma \vdash A : s_1 \ \wedge\ \Gamma; x : A \vdash B : s_2$.
- $\Gamma \vdash [x : A]b : X \Longrightarrow \exists s_1, s_2, s_3, B.\ (x : A)B =_\beta X \ \wedge\ \langle s_1, s_2, s_3\rangle \in \mathcal{R} \ \wedge\ \Gamma \vdash A : s_1 \ \wedge\ \Gamma; x : A \vdash b : B \ \wedge\ \Gamma; x : A \vdash B : s_2$.
- $\Gamma \vdash (a\ b) : X \Longrightarrow \exists x, A, B.\ A[x \leftarrow b] =_\beta X \ \wedge\ \Gamma \vdash a : (x : B)A \ \wedge\ \Gamma \vdash b : B$.

Tous les calculs du cube ont les propriétés de normalisation forte, confluence et préservation du typage. La preuve de ces propriétés se trouve dans [19].

Lemme 2.6 (Confluence) *La β-réduction est confluente.*

Lemme 2.7 (Normalisation forte) *La β-réduction est fortement normalisable sur tous les termes bien typés.*

Lemme 2.8 (Préservation du typage) *Si $\Gamma \vdash M : A$ et $M \to_\beta^* N$, alors $\Gamma \vdash N : A$.*

2.4 Pouvoir d'expression

Une motivation pour l'introduction de ces différentes extensions du λ-calcul simplement typé est d'accroître sa puissance d'expression, i.e. l'ensemble des fonctions représentables.

Les entiers naturels peuvent être représentés par les *entiers de Church*. L'entier n est représenté par le terme $\overline{n}$ ci-dessous où A est un type (fixé une fois pour toutes) et f est itéré n fois.

$$\overline{n} = [f : A \to A][x : A](f ... (f\ x) ...)$$

Son type est $(A \to A) \to A \to A$. On dit qu'une fonction $f : \mathbb{N}^k \to \mathbb{N}$ est représentée par un terme M si pour tout k-uplet $\langle n_1, ..., n_k \rangle$, on a:

$$(M\ \overline{n_1}...\overline{n_k}) =_\beta \overline{f(n_1, ..., n_k)}$$

Rappelons que le λ-calcul simplement typé ne permet de représenter que les polynômes étendus par le test à zéro.

Dans le système F, il est possible d'abstraire la définition des entiers par rapport au type A, on obtient alors:

$$\overline{n} = [A : *][f : A \to A][x : A](f...(f\ x)...)$$

de type $nat = (A : *)(A \to A) \to A \to A$. Cette définition permet d'itérer des fonctions de type $A \to A$ pour n'importe quel type A. Le système F permet de représenter toutes les fonctions prouvablement totales en arithmétique du second ordre. Comme exemple, nous donnons la représentation de la fonction puissance:

$$\overline{n}^{\overline{m}} = [A : *](\overline{m}\ (A \to A)\ (\overline{n}\ A))$$

Quant au système F_ω, il permet de représenter toutes les fonctions arithmétiques prouvablement totales d'ordre supérieur.

Une autre alternative est de représenter les entiers par une variable $0 : nat$ et une variable successeur $\mathcal{S} : nat \to nat$ avec $nat : *$ une variable. Malheureusement, l'expressivité est encore plus faible qu'auparavant; seules les fonctions constantes et les fonctions ajoutant une constante à l'un de leurs arguments sont représentables dans le λ-calcul simplement typé. Pour obtenir plus de fonctions représentables, il faut se donner un opérateur de récursion. On obtient alors le système T de Gödel. Les règles de calcul pour l'opérateur de récursion R sont:

$$\begin{array}{lcl} (R\ 0\ f\ x) & \to & x \\ (R\ (S\ n)\ f\ x) & \to & (f\ n\ (R\ n\ f\ x)) \end{array}$$

Beaucoup de fonctions arithmétiques sont représentables dans ce calcul: les fonctions primitives récursives, mais également les fonctions primitives récursives avec paramètres fonctionnels comme la fonction d'Ackermann.

Il faut toutefois tempérer ces résultats qui ne concernent que la définition de fonctions extensionnelles. Des fonctions extensionnellement équivalentes peuvent différer par leur *complexité*. Cette analyse a poussé L. Colson[41] à s'intéresser à la définissabilité d'*algorithmes* (fonctions intensionnelles). Il démontre alors que la récursion primitive n'est pas capable d'exprimer l'algorithme de calcul du minimum de deux entiers n et m en faisant un nombre d'opérations $min(n, m)$. Le système F, extentionnellement beaucoup plus puissant que le système T, peine pourtant à simuler efficacement les opérateurs de récursion primitive de ce dernier; l'algorithme min_F implémente dans le système F le calcul de min de deux entiers n et m en un temps équivalent à $\frac{9}{2}min(n, m)^2$.

2.5 Correspondance logique

Ces systèmes de types admettent une interprétation logique; c'est la correspondance de Curry-Howard (on dit aussi en anglais, *propositions-as-types*). Il convient aussi de rajouter le nom de de Bruijn qui a découvert indépendemment la même notion.

L'idée est simple: les types sont interprétés comme des propositions, et la preuve d'une proposition est un terme du type correspondant. Prouver une proposition revient alors à exhiber un terme du type correspondant. Lorsqu'une proposition n'a pas de preuve, il n'existe pas de terme de ce type. Il est d'ailleurs d'usage de noter la sorte $*$ par $Prop$ (type des propositions) et $\Box$ par $Type$. Il est aussi d'usage d'appeler *habitants* les preuves d'une proposition; une proposition est *habitée* si elle admet au moins une preuve.

Considérons maintenant le λ-calcul simplement typé. Le terme $[x : A]B$ de type $A \to B$ peut être considéré comme une fonction associant à tout terme de type A (preuve de A), un terme de type B (preuve de B). Le terme $[x : A]B$ peut ainsi être vu comme une preuve de $A \Rightarrow B$, d'après la sémantique des preuves élaborée par Heyting: une preuve de $A \Rightarrow B$ est une construction transformant toute preuve de A en une preuve de B. De cette observation, nous déduisons que $\to$ interprète l'implication $\Rightarrow$.

Les règles de typage deviennent ainsi des systèmes d'inférences logiques présentées sous forme de déduction naturelle; les règles (Prod) et (App), restreintes au cas du λ-calcul simplement typé, sont respectivement les règles d'introduction et d'élimination de $\Rightarrow$. Les contextes de typage se sont donc tout naturellement transformés en contextes d'axiomes (ou hypothèses). Par exemple, si on se donne deux axiomes $A \Rightarrow B \Rightarrow C$ et B et que l'on veut prouver $A \Rightarrow C$, ceci revient à se donner $f : A \to B \to C$ et $b : B$. Le terme $[a : A]((f\ a)\ b)$ de type $A \to C$ est une preuve de $A \Rightarrow C$. En résumé, le λ-calcul simplement typé représente la logique minimale. Il permet aussi de représenter la logique propositionnelle intuitionniste en ajoutant des règles de typage (inférence) pour les autres connecteurs.

De même, le système F correspond à une logique propositionnelle du second ordre et F_ω à une logique d'ordre supérieur. En effet il est possible dans le système F de quantifier sur les propositions grâce aux types dépendants; la proposition $Q = \forall A : Prop.A \Rightarrow A$ est traduite par $(A : Prop)A \to A$, et sa preuve est le terme $[A : Prop][x : A]x$. Dans le contexte de la logique, le polymorphisme est appelé *imprédicativité* car il permet d'appliquer des propositions à elles-mêmes. La négation et la proposition absurde sont définissables dans F, cependant la conjonction et la disjonction ne le sont pas. Toutefois, pour toutes propositions A et B, on peut définir $A \wedge B$ et $A \vee B$.

• absurde:

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

$$\bot\text{-}elim \quad = \quad [C : Prop][p : \bot](p\ C) : (C : Prop)\bot \to C$$

• négation:

$$\neg \quad = \quad [C : Prop]C \to \bot$$

Les règles $\neg$-$intro$ et $\neg$-$elim$ sont des cas particuliers de (Prod) ($\Rightarrow$ -$intro$) et (App) ($\Rightarrow$ -$elim$).

Le système F_ω étend cette quantification aux implications de propositions. Les connecteurs logiques sont ainsi définissables dans F_ω:

• conjonction:

$$\begin{array}{rcl} \wedge & = & [A:Prop][B:Prop](C:Prop)(A \to B \to C) \to C \\ & : & Prop \to Prop \to Prop \end{array}$$

$$\begin{array}{rcl} \wedge\text{-}intro & = & [A:Prop][B:Prop][a:A][b:A][C:Prop][f:A \to B \to C](f\ a\ b) \\ & : & (A:Prop)(B:Prop)A \to B \to A \wedge B \end{array}$$

$$\begin{array}{rcl} \wedge\text{-}elim1 & = & [A:Prop][B:Prop][p:A \wedge B](p\ A\ [a:A][b:B]a) \\ & : & (A:Prop)(B:Prop)A \wedge B \to A \end{array}$$

$$\begin{array}{rcl} \wedge\text{-}elim2 & = & [A:Prop][B:Prop][p:A \wedge B](p\ B\ [a:A][b:B]b) \\ & : & (A:Prop)(B:Prop)A \wedge B \to B \end{array}$$

• disjonction:

$$\begin{array}{rcl} \vee & = & [A:Prop][B:Prop](C:Prop)(A \to C) \to (B \to C) \to C \\ & : & Prop \to Prop \to Prop \end{array}$$

$$\begin{array}{rcl} \vee\text{-}intro1 & = & [A:Prop][B:Prop][a:A][C:Prop][f:A \to C][q:B \to C](f\ a) \\ & : & (A:Prop)(B:Prop)A \to A \vee B \end{array}$$

$$\begin{array}{rcl} \vee\text{-}intro2 & = & [A:Prop][B:Prop][b:B][C:Prop][f:A \to C][g:B \to C](g\ b) \\ & : & (A:Prop)(B:Prop)B \to A \vee B \end{array}$$

$$\begin{array}{rcl} \vee\text{-}elim & = & [A:Prop][B:Prop][C:Prop][p:A \vee B][f:A \to C][g:B \to C](p\ C\ f\ g) \\ & : & (A:Prop)(B:Prop)(C:Prop)(A \vee B) \to (A \to C) \to (B \to C) \to C \end{array}$$

Les termes canoniques de type $A \wedge B$ sont de la forme $(\wedge\text{-}intro\ A\ B\ a\ b)$ où a et b sont respectivement de type A et B. Alors que les termes canoniques de type $A \vee B$ sont de la forme $(\vee\text{-}intro1\ A\ B\ a)$ et $(\vee\text{-}intro2\ A\ B\ b)$ où a et b sont respectivement de type A et B. La correspondance logique est encore une fois confirmée par la sémantique de Heyting:

- une preuve de $A \wedge B$ est une paire formée d'une preuve de A et d'une preuve de B.
- une preuve de $A \vee B$ est une paire dont la seconde composante est soit une preuve de A soit une preuve de B, la première composante étant un booléen indiquant s'il s'agit d'une preuve de A ou de B.

Nous allons voir maintenant comment représenter le calcul des prédicats. Il nous faut pour cela nous placer dans le calcul des constructions. Un prédicat P sur un type A peut être considéré comme une fonction associant à tout objet $a : A$, une proposition $(P\ a)$; le prédicat P est vérifié en a si et seulement si $(P\ a)$ est habité. Un prédicat sur les entiers est alors tout terme de type $nat \to Prop$, comme $EstPair : nat \to Prop$; étant donné un terme $n : nat$, on pourra construire la proposition $(EstPair\ n) : Prop$. Les types dépendants sont alors utilisés pour interpréter la quantification universelle; $(n : nat)(EstPair\ n + 2)$ se lit $\forall n : nat.(EstPair\ n + 2)$. En effet, on retrouve bien la sémantique de Heyting du quantificateur universel: une preuve de $\forall x : A.P$ est une fonction qui à tout a de type A associe une preuve de $B[x \leftarrow a]$. Quant au quantificateur existentiel $\exists$, il est définissable comme suit:

• quantificateur existentiel:

$$\begin{array}{rcl} \exists & = & [A:Prop][P:A\to Prop](C:Prop)((x:A)(P\ x)\to C)\to C \\ & : & (A:Prop)(A\to Prop)\to Prop \end{array}$$

$$\begin{array}{rcl} \exists\text{-}intro & = & [A:Prop][P:A\to Prop][a:A][p:(P\ a)][C:Prop][q:(x:A)(P\ x)\to C](q\ a\ p) \\ & : & (A:Prop)(A\to Prop)(a:A)(P\ a)\to \exists A.P \end{array}$$

$$\begin{array}{rcl} \exists\text{-}elim & = & [A:Prop][P:A\to Prop][C:Prop][p:\exists A.P][q:(x:A)(P\ x)\to C](p\ C\ q) \\ & : & (A:Prop)(A\to Prop)(C:Prop)\exists A.P\to((x:A)(P\ x)\to C)\to C \end{array}$$

Un terme (preuve) de type $\exists A.P$ est de la forme ($\exists$-*intro* $A\ P\ a\ p$) où a est une preuve de A et p une preuve de $(P\ a)$. Ce qui correspond, encore une fois, à la sémantique de Heyting pour $\exists$.

2.6 Théorie des types extensionnelle

L'égalité est l'une des plus importantes constructions dans l'étude de la logique et des systèmes de preuves. En particulier, le problème concernant la nature intensionnelle ou extensionnelle de l'égalité a été longtemps au centre de ces recherches. L'égalité dans notre langage est l'égalité définitionnelle; elle est intensionnelle: deux objets sont égaux si et seulement si ils ont la même valeur. La notion d'extensionnalité de l'égalité concerne l'égalité de fonctions (termes dont le type est fonctionnel $(x:A)B$). En général, une égalité est extensionnelle si pour toutes fonctions f et g de même type, pour avoir f et g égaux, il suffit que $(f\ a)$ et $(g\ a)$ soient égaux pour tout a dans leur domaine commun. Notre égalité n'est pas extensionnelle dans ce sens: $[x:nat](+\ x\ 1)$ et $[x:nat](+\ 1\ x)$ retournent la même valeur pour tout entier n, mais ne sont pas définitionnellement égaux car leurs valeurs sont différentes. Notre égalité échoue aussi à être faiblement extensionnelle: les η-égalités $[x:A](f\ x)=f$ avec $x\notin VL(f)$ ne sont pas vérifiées en général.

D'autre part l'égalité est considérée comme un opérateur de base pour former des propositions logiques. Comme l'égalité définitionnelle n'est pas propositionnelle, il est important de décrire et de raisonner sur la notion d'égalité définitionnelle à l'intérieur même du langage. Dans les théories intensionnelles, la logique d'ordre supérieur permet la définition d'une égalité propositionnelle sur le principe de Leibniz que deux objets sont égaux si et seulement s'ils ne peuvent être distingués par aucune propriété (exprimée comme prédicat sur le type des objets considérés). Cette égalité sera étudiée en 6.2.1.

En théorie extensionnelle des types, l'égalité définitionnelle est appelée «égalité de jugement» et est notée $a=b:A$, énonçant que a et b sont deux objets de type A et sont définitionnellement égaux. Quant à l'égalité propositionnelle $Id(A,a,b)$, ses objets sont des preuves pour l'assertion «a et b sont des objets égaux de type A». La seule règle distinguant les théories intensionnelles des théories extensionnelles est la *règle de réflexion*:

$$\frac{\Gamma\vdash e:Id(A,a,b)}{\Gamma\vdash a=b:A}$$

C'est la seule règle qui déduit un jugement d'égalité à partir d'un jugement de typage. Grâce à elle, on peut substituer un égal à un égal non seulement pour l'égalité définitionnelle mais aussi pour l'égalité définitionnelle. L'ajout de cette règle ne met pas en péril la cohérence de la théorie, mais

modifie profondément son comportement calculatoire. Dans les théories intensionnelles, le calcul est essentiellement la β-réduction, alors que dans les théories extensionnelles, une étape de calcul élémentaire peut nécessiter la preuve que des objets de complexité arbitraire sont égaux. Aussi l'égalité définitionnelle n'est plus décidable, ainsi que le jugement de typage, i.e. on ne peut pas décider $\Gamma \vdash M : A$ pour Γ, M et A donnés. Leur implémentation sur ordinateur devenant ainsi très compliquée. Le système Nuprl[42] est un exemple d'implémentation de théorie extensionnelle des types (une version extensionnelle de la théorie de Martin-Löf).

Chapitre 3

Extensions

Le calcul des constructions a été introduit comme une base pour la formalisation des mathématiques constructives[44]. C'est un puissant formalisme pour exprimer des notions mathématiques. Toutefois un nombre important d'extensions ont été considérées. Ces extensions ont plusieurs motivations, non seulement le désir d'explorer le champ des extensions cohérentes du calcul, mais aussi pour satisfaire des besoins pratiques de preuves formelles et de développements de programmes.

Nous décrivons dans ce chapitre quelques unes de ces extensions. La théorie des types obtenue est alors très proche de la théorie du système de preuve Coq[49], que nous utiliserons dans les prochains chapitres pour la formalisation de la théorie des catégories.

Nous nous intéressons particulièrement à l'aspect algorithmique, dans le but de répondre aux questions d'implémentation sur ordinateur. Les deux principaux problèmes abordés sont:

- Le problème de vérification de type pour une relation d'inférence $\vdash$, consiste à déterminer si $\Gamma \vdash M : A$ (à lire «M admet le type A dans le contexte de déclarations Γ») pour un contexte Γ et des termes M et A donnés.

- Le problème du typage, consiste à décider, étant donnés un contexte Γ et un terme M, s'il existe un type B tel que $\Gamma \vdash M : B$.

Ces problèmes sont résolus par la construction d'un algorithme d'*inférence de type* qui, étant donnés un contexte Γ et un terme M, calcule si possible, un type B tel que $\Gamma \vdash M : B$, et échoue sinon. Pour les PTS fonctionnels (donc possédant la propriété d'unicité du type, voir lemme 2.4), pour vérifier si $\Gamma \vdash M : A$, il suffit alors de vérifier que A et le type inféré de M sont convertibles.

Chaque section de ce chapitre est consacrée à une extension. Les extensions abordées sont dans l'ordre: les constantes, les univers et les univers flottants, les types inductifs et les enregistrements dépendants.

3.1 Constantes

Les langages de programmation fournissent d'habitude un mécanisme pour introduire des définitions, c'est à dire une abréviation ou nom pour un terme pouvant être utilisé plusieurs fois dans

d'autres termes. En mathématiques aussi, au cours d'un énoncé ou d'une démonstration, il n'est pas rare d'utiliser une abréviation pour manipuler une expression compliquée souvent répétée. Dans les deux cas, l'introduction d'abréviations n'affecte ni la signification des objets manipulés, ni leurs propriétés.

Dans un système de preuve, un mécanisme de définitions est nécessaire pour faciliter la manipulation des objets mathématiques par l'humain. Il évite aussi à l'ordinateur les dépassements de capacité (overflow) causés par la taille des termes.

Les définitions sont d'habitude considérées au niveau du méta-langage et non comme faisant partie du système logique. Il est néanmoins important d'inclure les définitions dans la syntaxe du λ-calcul et d'étudier ses propriétés afin que les systèmes de preuve soient fidèles aux systèmes logiques qu'ils implémentent. C'est d'autant plus vrai avec les définitions récursives dont le dépliage peut s'itérer.

3.1.1 Présentation

Les PTS avec constantes ou définitions ont été notamment étudiés par E. Poll[130]l et P. Severi[143]. Pour étendre un PTS avec des constantes, il faut ajouter les constructions suivantes (A est le type de la constante):

$c := a : A$	constante globale (dans le contexte)
$[c := a : A]b$	constante locale (dans un terme)

Nous offrons aussi dans notre langage deux nouvelles possibilités:

- Déclarer le type d'un terme; ainsi $a :: A$ est un terme dont le type est A, A est appelé *cast*[1]. Bien entendu le type déclaré de a doit être convertible avec son type inféré. C'est par exemple utile pour imposer un type plus concis pour un terme.

- La déclaration du type d'une constante est facultative, puisqu'il peut être inféré à partir de sa valeur. On peut d'ailleurs toujours l'imposer en utilisant des casts; $c := a : A$ est équivalent à $c := a :: A$.

Nous étendons les définitions vues précédemment pour prendre en compte les constantes.

Définition 3.1

- *Les contextes sont définis par:*

$$\Gamma ::= [] \mid \Gamma; x : T \mid \Gamma; c := T : T$$

- *La grammaire des termes est maintenant:*

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

- *Nous définissons deux fonctions $Decl$ et Def calculant respectivement l'ensemble des variables déclarées et l'ensemble des constantes définies dans un contexte. La fonction Dom est alors redéfinie par $Dom(\Gamma) = Decl(\Gamma) \cup Def(\Gamma)$.*

[1] le terme le plus approprié est *coercion*; ce terme est toutefois utilisé pour une autre notion (chapitre 5).

- *On étend la notation Γ_x au cas des constantes pour désigner leur type. Nous désignons par Γ_c^{val} la valeur de la constante c dans Γ.*

Les définitions des notions de variable libre et substitution sont modifiées.

Définition 3.2 (Variables libres et substitution)

- *L'ensemble des variables libres est défini comme en 2.2, en ajoutant les cas suivants:*

$$\begin{aligned} VL([x := a : A]b) &= VL(a) \cup VL(A) \cup (VL(b) - \{x\}) \\ VL(a :: A) &= VL(a) \cup VL(A) \end{aligned}$$

- *L'opération de substitution est définie comme en 2.3, en rajoutant les cas suivants:*

$$\begin{aligned} ([x := a : A]b)[x \leftarrow c] &= [x := a : A]b \\ ([y := a : A]b)[x \leftarrow c] &= [y := a[x \leftarrow c] : A[x \leftarrow c]]b[x \leftarrow c] \quad si\ y \not\equiv x\ et\ y \notin VL(c) \\ ([y := a : A]b)[x \leftarrow c] &= [z := a[x \leftarrow c] : A[x \leftarrow c]]b[y \leftarrow z][x \leftarrow c] \\ & \quad si\ y \neq x,\ y \in VL(c)\ et\ z \notin VL(c) \\ (a :: A)[x \leftarrow c] &= a[x \leftarrow c] :: A[x \leftarrow c] \end{aligned}$$

De nouvelles règles de calcul sont aussi introduites pour prendre en compte l'opération d'expansion des constantes. Le principe est de remplacer un nom de constante par sa valeur. Pour celà, la réduction dépend du contexte contenant toutes les définitions de constantes.

Définition 3.3 (δ-réduction) *La relation de δ-réduction en une étape $\rightarrow_\delta$ est la plus petite relation telle que:*

- $\Gamma \vdash c \rightarrow_\delta \Gamma_c^{val}$ *avec* $c \in Def(\Gamma)$
- $\Gamma \vdash [x := a : A]b \rightarrow_\delta b$ *si* $x \notin VL(b)$
- *si* $\Gamma \vdash M \rightarrow_\delta N$ *alors* $\Gamma \vdash (M\ b) \rightarrow_\delta (N\ b)$, $\Gamma \vdash (a\ M) \rightarrow_\delta (a\ N)$, $\Gamma \vdash [x : M]a \rightarrow_\delta [x : N]a$, $\Gamma \vdash (x : M)a \rightarrow_\delta (x : N)a$, $\Gamma \vdash [x := a : M]b \rightarrow_\delta [x := a : N]b$, $\Gamma \vdash M :: A \rightarrow_\delta N :: A$, $\Gamma \vdash a :: M \rightarrow_\delta a :: N$ *et* $\Gamma \vdash [x := M : A]b \rightarrow_\delta [x := N : A]b$
- *si* $\Gamma; x : A \vdash M \rightarrow_\delta N$ *alors* $\Gamma \vdash [x : A]M \rightarrow_\delta [x : A]N$ *et* $\Gamma \vdash (x : A)M \rightarrow_\delta (x : A)N$.
- *si* $\Gamma; x := A \vdash M \rightarrow_\delta N$ *alors* $\Gamma \vdash [x := a : A]M \rightarrow_\delta [x := a : A]N$.

La relation de $\beta\delta$-équivalence $=_{\beta\delta}$ est la plus petite relation d'équivalence contenant $\rightarrow_\beta$ et $\rightarrow_\delta$. Nous écrirons $M =_{\beta\delta} N$ au lieu de $\Gamma \vdash M =_{\beta\delta} N$ lorsque le contexte Γ peut facilement être déduit.

Nous introduisons aussi de nouvelles règles d'inférence régissant la déclaration et le typage des constantes.

Définition 3.4 *Nous étendons le système d'inférence des PTS (2.11).*

- *On étend la relation de validité des contextes par:*

$$(\text{CConst})\ \frac{\Gamma \vdash a : A \qquad c \notin Dom(\Gamma)}{\Gamma; c := a : A \vdash}$$

- *On rajoute les trois règles suivantes au jugement de typage:*

$$(\text{CConstL})\ \frac{\Gamma; c := a : A \vdash B : s}{\Gamma \vdash [c := a : A]B : s}$$

$$(\text{CConstL-intro})\ \frac{\Gamma; c := a : A \vdash b : B \qquad \Gamma; c := a : A \vdash B : s}{\Gamma \vdash [c := a : A]b : [c := a : A]B}$$

$$(\text{Cast})\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash a : A}{\Gamma \vdash a :: A : A}$$

et on modifie la règle (Conv) *comme suit:*

$$(\text{Conv})\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash M : B \qquad \Gamma \vdash A =_{\beta\delta} B}{\Gamma \vdash M : A}$$

3.1.2 Inférence de types

L'inférence de type dans un PTS quelconque avec constantes est étudiée dans [143]. Pour notre part, dans le but d'avoir un algorithme d'inférence très simple, nous ne considérons que les PTS avec constantes globales. Les définitions locales seront traitées dans la section suivante. Nous suivons l'approche décrite dans [152]. Le PTS $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ que nous considérons vérifie les conditions suivantes:

- Il est fortement normalisable et confluent. On en déduit que la relation $=_{\beta\delta}$ est décidable.
- Les ensembles suivants sont décidables :

$$\mathcal{A}_0 = \{s_1 \in \mathcal{S} \mid \exists s_2 \in \mathcal{S}.\langle s_1, s_2 \rangle \in \mathcal{A}\}$$

$$\mathcal{R}_0 = \{s_1 \in \mathcal{S} \mid \exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3 \rangle \in \mathcal{R}\}$$

$$\mathcal{R}_1 = \{\langle s_1, s_2 \rangle \in \mathcal{S} \times \mathcal{S} \mid \exists s_3 \in \mathcal{S}.\langle s_1, s_2, s_3 \rangle \in \mathcal{R}\}$$

  Ces différentes conditions de décidabilité nous garantissent la décidabilité des différentes conditions annexes présentes dans les règles de typage.

- le PTS est semi-plein.

  Définition 3.5 (PTS semi-pleins) *Un PTS est* semi-plein *si et seulement si:*

$$\forall s_1 \in \mathcal{S}.(\exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3 \rangle \in \mathcal{R} \Rightarrow \forall s_2 \in \mathcal{S}.\exists s_3 \in \mathcal{S}.\langle s_1, s_2, s_3 \rangle \in \mathcal{R})$$

- le PTS est fonctionnel, afin de nous assurer l'unicité du type.

Ces restrictions paraissent au premier abord contraignantes mais sont en fait vérifiées par tous les PTS intéressants qui sont utilisés en pratique dans des implémentations.

La condition la plus importante concerne les PTS semi-pleins. Elle nous permet d'éviter la prémisse $\Gamma \vdash (x : A)B : s$ de la règle (Lam).

$$(\text{Lam})\ \frac{\Gamma; x : A \vdash b : B \qquad \Gamma \vdash (x : A)B : s}{\Gamma \vdash [x : A]b : (x : A)B}$$

Celle-ci est problématique pour la preuve de complétude de l'algorithme d'inférence (une discussion à ce sujet se trouve dans [152] et [131]).

Définition 3.6 (Règles d'inférence pour un PTS semi-plein) *La règle* (LAM) *est simplifiée pour les PTS semi-pleins comme suit:*

$$(\text{LAM})\ \frac{\Gamma \vdash A : s_1 \qquad \Gamma; x : A \vdash b : B \qquad (B \in \mathcal{A}_o \vee B \notin \mathcal{S}) \wedge \langle s_1, s_2, s_3 \rangle \in \mathcal{R}}{\Gamma \vdash [x : A]b : (x : A)B}$$

L'ensemble de règles des PTS n'est pas *dirigé par la syntaxe*, c'est à dire que pour certains contextes Γ et certains termes M, il existe plus d'une règle avec une conclusion de la forme $\Gamma \vdash M : A$. La règle fautive est la règle (CONV) qui peut être utilisée en n'importe quel point de la dérivation sans changer le sujet M. Chaque forme du terme M détermine alors une règle à appliquer, modulo l'utilisation de la règle (CONV). L'idée est de restreindre l'utilisation de la règle (CONV); elle ne sera utilisée que pour «modifier» le type d'un terme si nécessaire. Dans le cas des règles (NAME), (PROD) et (LAM), certains types doivent être des sortes, alors que dans (APP), le type attendu du terme en position fonctionnelle doit être un produit. Dans ces cas-là, il suffit de normaliser en utilisant une réduction de tête faible, évitant ainsi les réductions inutiles.

Définition 3.7 (réduction de tête faible) *La relation de réduction de tête faible en une étape, notée* $\rightarrow^{wh}_{\beta\delta}$*, est définie par les règles suivantes:*

$$\Gamma \vdash ([x : A]b)a \rightarrow^{wh}_{\beta\delta} b[x \leftarrow a] \qquad \Gamma \vdash c \rightarrow^{wh}_{\beta\delta} \Gamma^{val}_c \ \textit{avec}\ c \in Def(\Gamma)$$

$$\frac{\Gamma \vdash a \rightarrow^{wh}_{\beta\delta} a'}{\Gamma \vdash (a\ b) \rightarrow^{wh}_{\beta\delta} (a'\ b)}$$

Un terme en forme normale de tête a l'une des formes x, s, $(x : A)B$, $[x : A]b$ ou $(x\ a_1 ... a_n)$.

Définition 3.8 (Mise sous forme normale de tête faible) *La fonction de normalisation en forme normale de tête faible est:*

$$\begin{aligned}
\mathsf{Whnf}_\Gamma(c) &= \mathsf{Whnf}_\Gamma(\Gamma^{val}_c)\ si\ c \in Def(\Gamma)\\
\mathsf{Whnf}_\Gamma((c\ a_1 ... a_n)) &= \mathsf{Whnf}_\Gamma((\Gamma^{val}_c\ a_1 ... a_n))\ si\ c \in Def(\Gamma)\\
\mathsf{Whnf}_\Gamma((([x : A]b)a\ a_1 ... a_n)) &= \mathsf{Whnf}_\Gamma(b[x \leftarrow a]\ a_1 ... a_n)\\
\mathsf{Whnf}_\Gamma(a :: A) &= \mathsf{Whnf}_\Gamma(a)\\
\mathsf{Whnf}_\Gamma(M) &= M\ \textit{pour tous les autres cas}
\end{aligned}$$

Il est aussi nécessaire de vérifier, dans la règle (APP), la convertibilité du domaine du type du terme en position fonctionnelle avec le type de son argument. Pour ce faire, nous définissons un algorithme de test de convertibilité; M et N sont convertibles, noté $M \downarrow_\Gamma N$, si et seulement M et N se réduisent par une séquence standard de réduction vers un même terme. La conception d'un algorithme de conversion performant est primordiale pour tout système de preuve basé sur la théorie des types. Nous présenterons un deuxième algorithme, ayant comme avantage l'expansion «paresseuse» (selon le besoin) des constantes. Des algorithmes beaucoup plus efficaces sont en cours de développement dans l'équipe Coq (notamment par B. Barras et H. Laulhère).

Définition 3.9 (Algorithme de conversion (1))

$$M \downarrow_\Gamma N = \begin{cases} s_1 = s_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv s_1\ et\ \mathsf{Whnf}_\Gamma(N) \equiv s_2 \\ x = y & si\ \mathsf{Whnf}_\Gamma(M) \equiv x\ et\ \mathsf{Whnf}_\Gamma(N) \equiv y \\ A_1 \downarrow_\Gamma A_2 \wedge B_1 \downarrow_\Gamma B_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv (x : A_1)B_1\ et\ \mathsf{Whnf}_\Gamma(N) \equiv (x : A_2)B_2) \\ A_1 \downarrow_\Gamma A_2 \wedge b_1 \downarrow_\Gamma b_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv [x : A_1]b_1\ et\ \mathsf{Whnf}_\Gamma(N) \equiv [x : A_2]b_2) \\ a_1 \downarrow_\Gamma a_2 \wedge b_1 \downarrow_\Gamma b_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv (a_1\ b_1)\ et\ \mathsf{Whnf}_\Gamma(N) \equiv (a_2\ b_2) \\ \bot & pour\ tous\ les\ autres\ cas \end{cases}$$

Le prédicat $\downarrow_\Gamma$ coïncide avec $=_{\beta\delta}$ sur les termes bien typés. Les lemmes ci-dessous sont prouvés dans [143].

Lemme 3.1 (Correction) *Si $M \downarrow_\Gamma N$ alors $M =_{\beta\delta} N$.*

Lemme 3.2 (Complétude) *Si $M =_{\beta\delta} N$ et M et N sont fortement normalisables, alors $M \downarrow_\Gamma N$.*

Lemme 3.3 (Décidabilité) *$M \downarrow_\Gamma N$ est décidable pour M et N fortement normalisables.*

Le deuxième algorithme de conversion est basé sur une fonction de normalisation en forme normale de tête faible sans expansions de constantes.

Définition 3.10 (Mise sous forme normale de tête faible sans expansions de constantes) *La fonction de normalisation en forme normale de tête faible sans expansions de constantes est:*

$$\begin{aligned} \mathsf{Whnf}_\Gamma^{-\delta}((([x : A]b)a\ a_1...a_n)) &= \mathsf{Whnf}_\Gamma^{-\delta}(b[x \leftarrow a]\ a_1...a_n) \\ \mathsf{Whnf}_\Gamma^{-\delta}(a :: A) &= \mathsf{Whnf}_\Gamma^{-\delta}(a) \\ \mathsf{Whnf}_\Gamma^{-\delta}(M) &= M\ pour\ tous\ les\ autres\ cas \end{aligned}$$

Définition 3.11 (Algorithme de conversion (2))

$$M \downarrow_\Gamma N = \begin{cases} s_1 = s_2 & si\ \mathsf{Whnf}_\Gamma^{-\delta}(M) \equiv s_1\ et\ \mathsf{Whnf}_\Gamma^{-\delta}(N) \equiv s_2 \\ x = y & si\ \mathsf{Whnf}_\Gamma^{-\delta}(M) \equiv x\ et\ \mathsf{Whnf}_\Gamma^{-\delta}(N) \equiv y \\ A_1 \downarrow_\Gamma A_2 \wedge B_1 \downarrow_\Gamma B_2 & si\ \mathsf{Whnf}_\Gamma^{-\delta}(M) \equiv (x : A_1)B_1\ et \\ & \quad \mathsf{Whnf}_\Gamma^{-\delta}(N) \equiv (x : A_2)B_2) \\ A_1 \downarrow_\Gamma A_2 \wedge b_1 \downarrow_\Gamma b_2 & si\ \mathsf{Whnf}_\Gamma^{-\delta}(M) \equiv [x : A_1]b_1 et \\ & \quad \mathsf{Whnf}_\Gamma^{-\delta}(N) \equiv [x : A_2]b_2) \\ (c \equiv d\ \wedge\ n \equiv m\ \wedge\ (a_i \downarrow_\Gamma b_i)_{1 \le i \le n}) & si\ \mathsf{Whnf}_\Gamma^{-\delta}(M) \equiv (c\ a_1...a_n)(n \ge 0)\ et \\ \vee\ \mathsf{Whnf}_\Gamma^{-\delta}((\Gamma_c^{val}\ a_1...a_n)) \downarrow_\Gamma (d\ b_1...b_m) & \quad \mathsf{Whnf}_\Gamma^{-\delta}(N) \equiv (d\ b_1...b_m)(m \ge 0) \\ a_1 \downarrow_\Gamma a_2 \wedge b_1 \downarrow_\Gamma b_2 & si\ \mathsf{Whnf}_\Gamma^{-\delta}(M) \equiv (a_1\ b_1)\ et \\ & \quad \mathsf{Whnf}_\Gamma^{-\delta}(N) \equiv (a_2\ b_2) \\ \bot & pour\ tous\ les\ autres\ cas \end{cases}$$

Nous utilisons aussi l'optimisation dite des *contextes valides*, toujours utilisée en pratique, et définie dans [152, 131]. Elle consiste à supposer, lors du processus de typage, le contexte de départ bien formé, puis à vérifier que toutes les règles étendant ce contexte le maintiennent valide. Nous évitons ainsi la vérification de bonne formation des contextes pour chaque prémisse de règle.

Nous donnons enfin l'algorithme d'inférence.

Définition 3.12 (Algorithme d'inférence)

$$
\begin{array}{lcll}
\mathsf{Infer}(\Gamma, s_1) & = & s_2 & \textit{avec } \langle s_1, s_2\rangle \in \mathcal{A} \\
\mathsf{Infer}(\Gamma, x) & = & \Gamma_x & \\
\mathsf{Infer}(\Gamma, c) & = & \Gamma_c & \\
\mathsf{Infer}(\Gamma, (x:A)B) & = & \mathcal{R}_{\langle s_1,s_2\rangle} & \textit{avec } \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma, A)) \equiv s_1, \\
 & & & x \notin \mathit{Dom}(\Gamma) \textit{ et} \\
 & & & \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma; x:A, B)) \equiv s_2 \\
\mathsf{Infer}(\Gamma, [x:A]b) & = & (x:A)B & \textit{avec } \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma, A)) \equiv s_1, \\
 & & & x \notin \mathit{Dom}(\Gamma), \\
 & & & \mathsf{Infer}(\Gamma; x:A, b) \equiv B \textit{ et} \\
 & & & (\mathsf{Whnf}_\Gamma(B) \in \mathcal{A}_o \vee \mathsf{Whnf}_\Gamma(B) \notin \mathcal{S}) \wedge \exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3\rangle \in \mathcal{R} \\
\mathsf{Infer}(\Gamma, (a\ b)) & = & A[x \leftarrow b] & \textit{avec } \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma, a)) \equiv (x:B)A \textit{ et} \\
 & & & B \downarrow_\Gamma \mathsf{Infer}(\Gamma, b) \\
\mathsf{Infer}(\Gamma, a :: A) & = & A & \textit{avec } A \downarrow_\Gamma \mathsf{Infer}(\Gamma, a)
\end{array}
$$

Les preuves de correction et de complétude de cet algorithme sont similaires à celles présentées dans [152, 131]; l'introduction des constantes globales ne complique en rien ces résultats.

Théorème 1 (Correction) *Si* $\mathsf{Infer}(\Gamma, M) \not\equiv \bot$, *alors* $\Gamma \vdash M : \mathsf{Infer}(\Gamma, M)$.

Théorème 2 (Complétude) *Si* $\Gamma \vdash M : A$, *alors* $\mathsf{Infer}(\Gamma, M) =_{\beta\delta} A$.

3.1.3 Traitement des constantes locales

Alors que l'introduction des constantes globales ne pose aucune difficulté majeure, les choses semblent plus compliquées pour les constantes locales. Le problème de savoir si l'ajout de constantes locales à un PTS quelconque préserve la normalisation forte est encore un problème ouvert. Des théories de types particulières (dont le calcul des constructions) ont toutefois été étudiées en présence de constantes locales. Une étude plus générale a aussi été menée dans [143].

Dans notre calcul, nous avons choisi de considérer les définitions locales comme des abréviations. Une première alternative (adoptée par exemple par Automath) est de considérer $[x := a : A]b$ comme une autre écriture de $([x : A]b\ a)$. Cependant ces deux termes ne sont pas strictement équivalents, le premier peut être typable sans que le second ne le soit. Considérons l'exemple donné dans [143]:

$$[A : *][B := A : *][b : B][f : A \to A](f\ b) : (A : *)[B := A : *]B \to (A \to A) \to A$$

Sa traduction donnée ci-dessous, est mal typée car dans $(f\ b)$, le type B de b est incompatible avec le type $A \to A$ de f.

$$[A : *]([B : *][b : B][f : A \to A](f\ b)\ A)$$

L'alternative que nous adoptons est de traduire $[x := a : A]b$ par $b[x \leftarrow a]$; nous perdons toutefois la liberté de remplacer seulement quelques occurrences de x dans b. Cette transformation est réalisée lors du processus d'inférence de type car nous voulons nous assurer que a est bien typé. Nous considérons ainsi comme mal typés les termes $[x := a : A]b$ avec $x \notin VL(b)$ si $\Gamma \not\vdash a : A$.

Nous présentons maintenant un algorithme d'inférence prenant en compte toutes les notions qu'on vient de voir. La fonction Infer prend un contexte et un terme et rend une paire formée du terme original sans constantes locales et de son type. Nous détaillons aussi les opérations de délaration de « variables globales » et de définition de constantes.

Définition 3.13 (Variables globales ou Axiomes) *Les* variables globales *sont des variables du contexte ne provenant pas de lieurs* $[x : A]$ *ou* $(x : A)$*; ils correspondent à des constantes sans corps de définition, c'est à dire des* axiomes.

Définition 3.14 (Algorithme d'inférence)

Déclaration de variables (Axiomes).

$$\mathsf{Déclaration}(\Gamma, x, A) \quad = \quad \Gamma; x : A' \quad avec\ x \notin Dom(\Gamma),\ \mathsf{Infer}(\Gamma, A) \equiv A', X\ et\ \mathsf{Whnf}_\Gamma(X) \equiv s$$

Définition de constantes.

$$\mathsf{Définition}(\Gamma, c, a) \quad = \quad \Gamma; c := a' : A \quad avec\ c \notin Dom(\Gamma)\ et\ \mathsf{Infer}(\Gamma, a) \equiv a', A$$

Inférence de types.

$\mathsf{Infer}(\Gamma, s_1)$	$=$	s_1, s_2	$avec\ \langle s_1, s_2\rangle \in \mathcal{A}$
$\mathsf{Infer}(\Gamma, x)$	$=$	x, Γ_x	
$\mathsf{Infer}(\Gamma, c)$	$=$	c, Γ_c	
$\mathsf{Infer}(\Gamma, [x := a]b)$	$=$	$\mathsf{Infer}(\Gamma, b[x \leftarrow a'])$	$avec\ \mathsf{Infer}(\Gamma, a) \equiv a', A$
$\mathsf{Infer}(\Gamma, (x : A)B)$	$=$	$(x : A')B', \mathcal{R}_{\langle s_1, s_2\rangle}$	$avec\ \mathsf{Infer}(\Gamma, A) \equiv A', X,$ $\mathsf{Whnf}_\Gamma(X) \equiv s_1,$ $\mathsf{Infer}(\Gamma; x : A', B) \equiv B', Y\ et$ $\mathsf{Whnf}_\Gamma(Y) \equiv s_2$
$\mathsf{Infer}(\Gamma, [x : A]b)$	$=$	$[x : A']b', (x : A')B'$	$avec\ \mathsf{Infer}(\Gamma, A) \equiv A', X,$ $\mathsf{Whnf}_\Gamma(X) \equiv s_1,$ $\mathsf{Infer}(\Gamma; x : A', b) \equiv b', B'\ et$ $(\mathsf{Whnf}_\Gamma(B') \in \mathcal{A}_o \vee \mathsf{Whnf}_\Gamma(B') \notin \mathcal{S}) \wedge$ $\exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3\rangle \in \mathcal{R}$
$\mathsf{Infer}(\Gamma, (a\ b))$	$=$	$(a'\ b'), A[x \leftarrow b']$	$avec\ \mathsf{Infer}(\Gamma, a) \equiv a', X,$ $\mathsf{Whnf}_\Gamma(X) \equiv (x : B)A,$ $\mathsf{Infer}(\Gamma, b) \equiv b', B'\ et$ $B \downarrow_\Gamma B'$

$$\mathsf{Infer}(\Gamma, a :: A) \quad = \quad a' :: A', A' \quad \begin{array}{l} avec\ \mathsf{Infer}(\Gamma, a) \equiv a', X, \\ \mathsf{Infer}(\Gamma, A) \equiv A', Y\ \ et \\ A' \downarrow_\Gamma X \end{array}$$

3.2 Hiérarchie d'Univers

3.2.1 Présentation

Dans certaines extensions du calcul des constructions (CC$^\omega$et ECC[103]), la sorte $\Box$ (ou $Type$) est elle-même bien typée. On introduit pour cela une nouvelle sorte pour la typer, et on itère le processus; on obtient ainsi une *hiérarchie d'univers* $Type_i$ avec $i \in \mathbb{N}$.

$$Prop : Type_0 : Type_1 : ...$$

Ces univers sont prédicatifs, dans le sens où chaque univers ne contient que des objets appartenant à des univers de niveau inférieur.

$$Prop \subset Type_0 \subset Type_1 \subset ...$$

En particulier, il ne se contient pas lui-même, et on ne peut pas quantifier sur un univers pour former un objet de cet univers. Notre référence principale pour cette section est [103].

Définition 3.15 (Termes) *Le langage des termes est:*

$$T ::= Prop \mid Type_i \mid x \mid [x : T]T \mid (T\ T) \mid (x : T)T \quad (i \in \mathbb{N})$$

En plus de la relation de conversion, il existe une relation de *cumulativité* définie sur les termes. Cette relation reflète l'inclusion des univers: tout objet de type $Prop$ est de type $Type_0$, et tout objet de type $Type_i$ est un objet de type $Type_{i+1}$. On y inclut aussi la conversion et la contravariance à droite.

Définition 3.16 (Relation de cumulativité) $\preceq$ *est la plus petite relation d'ordre vérifiant:*

- $Prop \preceq Type_0$
- $Type_i \preceq Type_{i+1}$ *pour* $i \geq 1$
- $=_{\beta\delta} \subset \preceq$
- $A_1 =_{\beta\delta} A_2$ *et* $B_1 \preceq B_2 \Rightarrow (x : A_1)B_1 \preceq (x : A_2)B_2$

Notre présentation est celle de ECC[103]; elle se distingue de celle de CC$^\omega$ par le fait que la règle (Cum) est remplacée par deux règles:

$$\frac{\Gamma \vdash A : Type_i \qquad \Gamma \vdash M : B \qquad \Gamma \vdash A =_{\beta\delta} B}{\Gamma \vdash M : A} \qquad \frac{\Gamma \vdash M : Type_i}{\Gamma \vdash M : Type_{i+1}}$$

De plus la contravariance de la cumulativité n'est pas prise en compte.

Définition 3.17 (Règles d'inférence) *Les règles de typage de ECC sont:*

$$(\textsc{Axiom1})\ \frac{}{\Gamma \vdash Prop : Type_0}$$

$$(\textsc{Axiom2})\ \frac{}{\Gamma \vdash Type_i : Type_{i+1}}$$

$$(\textsc{Name})\ \frac{x \in Dom(\Gamma)}{\Gamma \vdash x : \Gamma_x}$$

$$(\textsc{Lam})\ \frac{\Gamma; x : A \vdash b : B}{\Gamma \vdash [x : A]b : (x : A)B}$$

$$(\textsc{Prod1})\ \frac{\Gamma; x : A \vdash B : Prop}{\Gamma \vdash (x : A)B : Prop}$$

$$(\textsc{Prod2})\ \frac{\Gamma \vdash A : Type_i \qquad \Gamma; x : A \vdash B : Type_i}{\Gamma \vdash (x : A)B : Type_i}$$

$$(\textsc{App})\ \frac{\Gamma \vdash a : (x : B)A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a\ b) : A[x \leftarrow b]}$$

$$(\textsc{Cum})\ \frac{\Gamma \vdash A : Type_i \qquad \Gamma \vdash M : B \qquad B \preceq A}{\Gamma \vdash M : A}$$

3.2.2 Inférence de types

À cause de l'inclusion des univers, la propriété d'unicité de type (modulo conversion) n'est plus vérifiée. ECC admet toutefois une notion simple de *type principal* qui caractérise l'ensemble des types d'un terme bien typé.

Les définitions précédentes concernant la forme normale de tête restent valables, mais l'algorithme de conversion est modifié pour prendre en compte la cumulativité.

Définition 3.18 (Conversion)

$$M \downarrow_\Gamma^\preceq N = \begin{cases} vrai & si\ \mathsf{Whnf}_\Gamma(M) \equiv Prop\ et\ \mathsf{Whnf}_\Gamma(N) \equiv Type_i \\ i \leq j & si\ \mathsf{Whnf}_\Gamma(M) \equiv Type_i\ et\ \mathsf{Whnf}_\Gamma(N) \equiv Type_j \\ A_1 \downarrow_\Gamma A_2 \wedge B_1 \downarrow_\Gamma^\preceq B_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv (x : A_1)B_1\ et\ \mathsf{Whnf}_\Gamma(N) \equiv (x : A_2)B_2 \\ M \downarrow_\Gamma N & pour\ tous\ les\ autres\ cas \end{cases}$$

Nous donnons maintenant un algorithme d'inférence qui étant donné un terme dans un contexte, calcule son type principal s'il est bien typé, et échoue (retourne $\perp$) sinon.

Définition 3.19 (Type principal) *A est appelé type principal pour M dans Γ si et seulement si $\Gamma \vdash M : A$, et pour tout A', $\Gamma \vdash M : A'$ si et seulement si $A \preceq A'$.*

Lemme 3.4 (Existence d'un type principal) *Tour terme M bien typé dans un contexte Γ possède un type principal; c'est le type minimal de M par rapport à $\preceq$. Nous noterons ce type par $Min_\Gamma(M)$.*

PREUVE voir [103]. ■

L'algorithme d'inférence utilise une fonction de calcul de la sorte maximale par rapport à la cumulativité entre deux sortes.

Définition 3.20 $s_1 \Uparrow s_2 = \begin{cases} s_1 & si\ s_2 \equiv Prop \vee (s_1 \equiv Type_i \wedge s_2 \equiv Type_j \wedge i > j) \\ s_2 & sinon \end{cases}$

Définition 3.21 (Algorithme d'inférence)

$$
\begin{array}{lcll}
\mathsf{Infer}(\Gamma, Prop) & = & Type_0 & \\
\mathsf{Infer}(\Gamma, Type_i) & = & Type_{i+1} & \\
\mathsf{Infer}(\Gamma, x) & = & \Gamma_x & \\
\mathsf{Infer}(\Gamma, c) & = & \Gamma_c & \\
\mathsf{Infer}(\Gamma, (x:A)B) & = & \begin{cases} Prop & si\ s_1 \equiv Prop \\ s_1 \Uparrow s_2 & sinon \end{cases} & avec \begin{cases} \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma, A)) \equiv s_1, \\ x \notin Dom(\Gamma)\ et \\ \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma; x:A, B)) \equiv s_2 \end{cases} \\
\mathsf{Infer}(\Gamma, [x:A]b) & = & (x:A)B & avec\ \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma, A)) \equiv s_1, \\
 & & & x \notin Dom(\Gamma)\ et \\
 & & & \mathsf{Infer}(\Gamma; x:A, b) \equiv B \\
\mathsf{Infer}(\Gamma, (a\ b)) & = & A[x \leftarrow b] & avec\ \mathsf{Whnf}_\Gamma(\mathsf{Infer}(\Gamma, a)) \equiv (x:B)A\ et \\
 & & & \mathsf{Infer}(\Gamma, b) \downarrow_\Gamma^\preceq B \\
\mathsf{Infer}(\Gamma, a :: A) & = & A & avec\ \mathsf{Infer}(\Gamma, a) \downarrow_\Gamma^\preceq A
\end{array}
$$

Théorème 3 (Correction et complétude de l'algorithme d'inférence)

$$\mathsf{Infer}(\Gamma, a) = \begin{cases} Min_\Gamma(M) & si\ et\ seulement\ si\ M\ est\ bien\ typé\ dans\ \Gamma \\ \bot & sinon \end{cases}$$

PREUVE voir [103]. ■

3.2.3 Univers flottants

Bien qu'essentiels pour la formalisation des mathématiques, les univers sont d'un maniement pénible en pratique. En effet, on doit faire des choix précis d'univers et s'assurer de leur cohérence. Une approche alternative a été introduite dans Principia Mathematica[157]. Elle consiste à adopter une convention appelée «ambiguïté de type», dans laquelle les niveaux d'univers ne sont pas explicitement mentionnés, mais où il est convenu qu'il existe une «affectation de niveaux» tel que le terme résultant est bien typé vis-à-vis des règles du calcul. De plus, le choix exact des niveaux d'univers est sans importance; seule la relation entre les choix de niveaux à des points différents de la preuve

importe. Il devient alors inutile de définir plusieurs fois une même notion à des niveaux différents. L'ambiguïté de type est donc un moyen d'atteindre la flexibilité d'avoir un type de tous les types, sans sacrifier la cohérence logique du système. C'est cette notion que nous appelons «univers flottants». Notre référence principale pour cette section est [68], mais les idées remontent aux travaux de [83]. L'algorithme d'inférence présenté dans [68] est pour CC^ω, nous l'étendons au cas de ECC.

Nous distinguons entre deux syntaxes. La *syntaxe concrète* ne comprenant qu'une seule sorte *anonyme Type* à la place de la hiérarchie $\{Type_i \mid i \in \mathbb{N}\}$. Dans la *syntaxe abstraite*, on dispose d'une famille d'univers indicés par des *variables de niveau*. Ces variables de niveau sont utilisées pour décrire l'ensembles des «lectures» possibles pour un terme. De chaque affectation de niveau résulte un terme valide; ce ne sont pas les valeurs absolues de ces indices qui importent mais les relations entre eux. Ces relations forment les contraintes sous lesquelles le terme est valide.

Définition 3.22 (Termes)

- *Syntaxe concrète (ou termes ambigus):*

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

- *Syntaxe abstraite:*

$$U ::= Prop \mid Type_\alpha \mid x \mid [x:U]U \mid (U\ U) \mid (x:U)U \quad (\alpha \in V)$$

avec V un ensemble de variable de niveaux. L'ensemble $VN(M)$ dénote l'ensemble des variables de niveaux apparaissant dans M.

Nous définissons maintenant les notions d'ensemble de contraintes et d'affectation de niveaux.

Définition 3.23 (Ensemble de contraintes)

- *Un ensemble de contraintes est un ensemble d'inégalités de la forme $\alpha > \lambda$ ou $\alpha \geq \lambda$.*
- *La notation $\alpha = \lambda$ est une abréviation pour la paire de contraintes $\lambda \geq \alpha$ et $\alpha \geq \lambda$.*
- *Nous notons par $VN(C)$ l'ensemble des variables de niveaux apparaissant dans un ensemble de contraintes C.*

Définition 3.24 (Affectation de niveau)

- *Une* affectation de niveau *est toute fonction partielle $\sigma : V \to \mathbb{N}$ avec un support fini; nous notons par $Dom(\sigma)$ son domaine.*
- *Les affectations de niveau sont étendues aux termes de la façon usuelle, c'est à dire chaque $Type_\alpha$ est remplacé par $Type_{\sigma(\alpha)}$ si $\alpha \in Dom(\sigma)$.*
- *On l'étend aussi de la même manière aux ensembles de contraintes et aux contextes.*
- *Une affectation de niveaux σ* satisfait *un ensemble de contraintes C, noté $\sigma \models C$, si et seulement si $Dom(\sigma) \supseteq VN(C)$ et toutes les inégalités de $\sigma(C)$ sont vérifiées.*

- *Un ensemble de contraintes C est dit* cohérent *s'il existe une affectation de niveaux σ telle que $\sigma \models C$. Nous parlons dans le cas contraire d'incohérence d'univers.*

Nous donnons un nouvel algorithme de conversion, qui étant deux termes M et N, construit le plus faible ensemble de contraintes C tel que si $\sigma \models C$, alors $\sigma(M) \downarrow_\Gamma \sigma(N)$ (sans univers flottants).

Définition 3.25 (Conversion)

$$M \downarrow_\Gamma N = \begin{cases} \emptyset & si\ \mathsf{Whnf}_\Gamma(M) \equiv Prop\ et\ \mathsf{Whnf}_\Gamma(N) \equiv Prop \\ \{\alpha = \beta\} & si\ \mathsf{Whnf}_\Gamma(M) \equiv Type_\alpha\ et\ \mathsf{Whnf}_\Gamma(N) \equiv Type_\beta \\ \emptyset & si\ \mathsf{Whnf}_\Gamma(M) \equiv x, \mathsf{Whnf}_\Gamma(N) \equiv y\ et\ x = y \\ A_1 \downarrow_\Gamma A_2 \cup B_1 \downarrow_\Gamma B_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv (x : A_1)B_1\ et\ \mathsf{Whnf}_\Gamma(N) \equiv (x : A_2)B_2 \\ A_1 \downarrow_\Gamma A_2 \cup b_1 \downarrow_\Gamma b_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv [x : A_1]b_1\ et\ \mathsf{Whnf}_\Gamma(N) \equiv [x : A_2]b_2 \\ a_1 \downarrow_\Gamma a_2 \cup b_1 \downarrow_\Gamma b_2 & si\ \mathsf{Whnf}_\Gamma(M) \equiv (a_1\ b_1)\ et\ \mathsf{Whnf}_\Gamma(N) \equiv (a_2\ b_2) \\ \bot & pour\ tous\ les\ autres\ cas \end{cases}$$

Cet algorithme vérifie toutes les bonnes propriétés ci-dessous, prouvées dans [68].

Lemme 3.5 (Correction) *Si $M \downarrow_\Gamma N = C$ et $\sigma \models C$, alors $\sigma(M) \downarrow_\Gamma \sigma(N)$*

Lemme 3.6 (Compléude) *Si $\sigma(M) \downarrow_\Gamma \sigma(N)$, alors il existe C tel que $M \downarrow_\Gamma N = C$ et $\sigma \models C$.*

Lemme 3.7 (Décidabilité) *Étant donnés deux termes fortement normalisables, il est décidable si $M \downarrow_\Gamma N = \bot$ ou non.*

Nous définissons maintenant trois fonctions intermédiaires, utilisées pour l'algorithme d'inférence.

- $\mathsf{Whnf}^*_\Gamma(M)$ pour décomposer un terme en un produit.
- $\mathsf{Cum}_\Gamma(A, C)$ applique, si possible, la cumulativité à A.
- $s_1 \Uparrow_C s_2$ calcule le «maximum» des deux sortes s_1 et s_2.

Dans les deux dernières fonctions, les nouvelles contraintes engendrées sont enregistrées dans C.

Définition 3.26

$$\begin{aligned} \mathsf{Whnf}^*_\Gamma((c\ a_1 ... a_n)) &= \mathsf{Whnf}^*_\Gamma((\Gamma_c^{val}\ a_1 ... a_n))\ si\ c \in Def(\Gamma) \\ \mathsf{Whnf}^*_\Gamma((([x : A]b)a\ a_1 ... a_n)) &= \mathsf{Whnf}^*_\Gamma(b[x \leftarrow a]\ a_1 ... a_n) \\ \mathsf{Whnf}^*_\Gamma((x : A)B) &= (x : A)\mathsf{Whnf}^*_\Gamma(B) \\ \mathsf{Whnf}^*_\Gamma(a :: A) &= \mathsf{Whnf}^*_\Gamma(a) \\ \mathsf{Whnf}^*_\Gamma(M) &= M\ pour\ tous\ les\ autres\ cas \end{aligned}$$

$$\mathsf{Cum}_\Gamma(A,C) = \begin{cases} (x_1:A_1)...(x_n:A_n)Type_\alpha, C \cup \{\alpha \geq \lambda\} & si\ \mathsf{Whnf}^*_\Gamma(A) \equiv \\ & \quad (x_1:A_1)...(x_n:A_n)Type_\lambda \\ A, C & sinon \end{cases}$$

$$s_1 \Uparrow_C s_2 = \begin{cases} Prop, C & si\ s_2 = Prop \\ Type_\alpha, C \cup \{\alpha \geq \lambda\} & si\ s_1 = Prop\ et\ s_2 = Type_\lambda \\ Type_\alpha, C \cup \{\alpha \geq \lambda, \alpha \geq \mu\} & si\ s_1 = Type_\lambda\ et\ s_2 = Type_\mu \\ \bot & sinon \end{cases}$$

Il est important de remarquer que les contextes ne peuvent par contenir des termes ambigus; le type (resp. type et valeur) d'une variable (resp. constante) est fixé dès sa déclaration (resp. définition).
L'algorithme d'inférence prend en paramètre un contexte Γ, un ensemble de contraintes C et un terme M en syntaxe concrète à typer. L'ensemble C est tel que $VN(\Gamma) \subseteq VN(C)$. Si $\sigma \models C$, alors $\sigma(\Gamma)$ est un «contexte ordinaire». Le résultat de l'algorithme est un terme M' (version en syntaxe abstraite de M), son type (aussi en syntaxe abstraite) et un nouvel ensemble de contraintes.

Définition 3.27 (Algorithme d'inférence)

$$\begin{array}{lcl}
\mathsf{Infer}(\Gamma, C, Prop) & = & Prop, Type_\alpha, \{\alpha \geq 0\} \\
\mathsf{Infer}(\Gamma, C, Type) & = & Type_\beta, Type_\alpha, \{\alpha > \beta \geq 0\} \\
\mathsf{Infer}(\Gamma, C, x) & = & x, \mathsf{Cum}(\Gamma_x, \emptyset) \\
\mathsf{Infer}(\Gamma, C, c) & = & c, \mathsf{Cum}(\Gamma_c, \emptyset) \\
\mathsf{Infer}(\Gamma, C, (x:A)B) & = & (x:A')B', \mathsf{Whnf}_\Gamma(X) \Uparrow_{D \cup E} \mathsf{Whnf}_\Gamma(Y) \\
 & & avec\ \mathsf{Infer}(\Gamma, C, A) \equiv A', X, D, \\
 & & C \cup D\ coh\acute{e}rent, \\
 & & x \notin Dom(\Gamma)\ et \\
 & & \mathsf{Infer}((\Gamma; x:A'), C \cup D, B) \equiv B', Y, \mathcal{E} \\
\mathsf{Infer}(\Gamma, C, [x:A]b) & = & [x:A']b', \mathsf{Cum}((x:A')B, D \cup \mathcal{E}) \\
 & & avec\ \mathsf{Infer}(\Gamma, C, A) \equiv A', X, D, \\
 & & \mathsf{Whnf}_\Gamma(X) \equiv s, \\
 & & C \cup D\ coh\acute{e}rent, \\
 & & x \notin Dom(\Gamma)\ et \\
 & & \mathsf{Infer}((\Gamma; x:A'), C \cup D, b) \equiv b', B, \mathcal{E} \\
\mathsf{Infer}(\Gamma, C, (a\ b)) & = & (a'\ b'), \mathsf{Cum}(A[x \leftarrow b'], D \cup \mathcal{E} \cup (B \downarrow_\Gamma B')) \\
 & & avec\ \mathsf{Infer}(\Gamma, C, a) \equiv a', X, D, \\
 & & \mathsf{Whnf}_\Gamma(X) \equiv (x:B)A\ et \\
 & & \mathsf{Infer}(\Gamma, C, b) \equiv b', B', \mathcal{E}
\end{array}$$

Cet algorithme jouit des bonnes propriétés de correction et de complétude. Pour exprimer ces propriétés, nous avons besoin de la fonction ci-dessous.

Définition 3.28 *$\widehat{M}$ est obtenu à partir du terme M en remplaçant toute occurrence de $Type_j$ ($j \in \mathbb{N}$) par $Type$.*

Théorème 4 (Correction) *Si* $\mathsf{Infer}(\Gamma, C, M) \equiv M', A, D$, *alors*

1. $VN(M') \subseteq VN(D)$, $VN(A) \subseteq VN(C \cup D)$ *et* $VN(D) - VN(C)$ *est un ensemble de nouvelles variables.*
2. $M \equiv \widehat{M'}$
3. *Si* $\sigma \models C \cup D$, *alors* $\sigma(\Gamma) \vdash \sigma(M) : \sigma(A)$.

PREUVE Adaptation de [68]. ■

Théorème 5 (Complétude) *Si* $\sigma \models C$ *avec* $Dom(\sigma) = VN(C)$, *et* $\sigma(\Gamma) \vdash M : A$, *alors il existe* M', A' *et* τ *tels que:*

1. $\tau \models D$ *et* $Dom(\tau) = VN(C \cup D)$
2. $\tau(M') \equiv M$ *et* $\tau(A') = A$.
3. $\mathsf{Infer}(\Gamma, C, \widehat{M}) \equiv M', A', D$,

PREUVE [68] ■

L'algorithme gérant les univers flottants décrit dans [68] et implémenté dans LEGO, est plus complexe. En effet, il incorpore à la fois des univers anonymes et des univers constants $Type_j$ ($j \in \mathbb{N}$). Plus important encore, il supporte le *polymorphisme d'univers*. Dans notre cas, soit l'identité polymorphe définie par:

$$Id := [A : Type][x : A]x$$

Son analyse produit le contexte contraint suivant:

$$Id := [A : Type_\alpha][x : A]x : (A : Type_\beta)A \to A \ , \{\beta > \alpha\}$$

Le terme $(Id\ ((A : Type)A \to A)\ Id)$ est alors mal typé. Avec le polymorphisme d'univers, ce dernier terme est bien typé car il est équivalent à $(Id\ ((A : Type)A \to A)\ ([A : Type][x : A]x))$. En effet, chaque constante est «relue» à chacune de ses apparitions. C'est cette relecture qui rend les implémentations de ce mécanisme de polymorphisme d'univers extrêmement coûteuses en temps. Le mécanisme de polymorphisme d'univers s'avère toutefois utile pour la formalisation de certaines notions mathématiques (un exemple est donné dans la section 8.7).

3.3 Types inductifs

L'introduction des types inductifs dans le CC permet de définir les types de données et les prédicats inductifs de manière plus naturelle. Le calcul résultant est appelé Calcul des Constructions Inductives (CCI). Cette extension, définie par T. Coquand[47] et C. Paulin-Mohring[119], consiste à étendre le langage des termes par de nouvelles constructions inductives, et par les règles de calcul correspondantes. Une présentation détaillée de cette extension demanderait un trop long développement; nous préferons nous en tenir à une compréhension intuitive et renvoyons le lecteur intéressé aux travaux déjà cités.

Un type défini inductivement est le plus petit type vérifiant un certain nombre de prédicats. Ainsi le type des entiers naturels est un type contenant une constante 0 et est clos par la fonction successeur S. Les constantes 0 et S sont les *constructeurs* du type inductif *nat*.

$Inductive\ nat\ :\ *\ :=\ 0 : nat \mid S : nat \rightarrow nat.$

La condition de minimalité de *nat* est exprimée par le principe de récurrence suivant:

$$nat\_ind : (P : nat \rightarrow Prop)(P\ 0) \rightarrow ((n : nat)(P\ n) \rightarrow (P\ (S\ n))) \rightarrow (n : nat)(P\ n)$$

Définition 3.29 (Types inductifs) *La syntaxe des types inductifs est :*

$$Inductive\ I\ [p_1 : P_1; ...; p_n : P_n]\ :\ (x : M_1)...(x : M_r)s\ :=\ c_1 : C_1\ |...|\ c_m : C_m$$

Les p_i sont les paramètres *du type inductif, n le nombre de paramètres, et les c_i sont les* constructeurs *du type inductif.*

Nous donnons quelques exemples représentatifs de types inductifs:

- Le type *bool* des booléens est représenté par:

  $Inductive\ bool\ :\ *\ :=\ true : bool \mid false : bool.$

- Le type *list* des listes est un exemple de type inductif polymorphe:

  $Inductive\ list\ [A : *]\ : * := nil : (list\ A)$
  $\mid cons : A \rightarrow (list\ A) \rightarrow (list\ A).$

  Le type des vecteurs est, quant à lui, un exemple de type inductif dépendant d'un paramètre entier:

  $Inductive\ vect\ [A : s]\ : nat \rightarrow * := nilv : (vect\ A\ O)$
  $\mid consv : A \rightarrow (n : nat)(vect\ A\ n) \rightarrow (vect\ A\ (S\ n)).$

- Le dernier exemple est celui du prédicat inductif *EstPair*:

  $Inductive\ EstPair\ : nat \rightarrow Prop := e0 : (EstPair\ 0)$
  $\mid eS : (n : nat)(EstPair\ n) \rightarrow (EstPair\ (S(S\ n))).$

Pour préserver la cohérence de la théorie, les types inductifs doivent vérifier une condition de validité portant sur le type de leurs constructeurs (voir [49] pour plus de détails). Ainsi par exemple, le type ci-dessous est non valide:

$Inductive\ I\ :*\ :=\ C :\ (I \rightarrow I) \rightarrow I.$

À chaque type inductif est associé un schéma d'élimination correspondant à la combinaison de deux opérateurs : un opérateur de filtrage et un opérateur de point fixe.
L'idée de base de l'opérateur de filtrage $Case...of...end$ est que pour définir (resp. démontrer) une fonction (resp. propriété) sur les objets d'un type inductif I, il suffit de la définir (démontrer) sur chacun de ses constructeurs. Ainsi dans le terme ci-dessous, une fonction f_i est associée à chaque constructeur c_i.

$$\langle P\rangle Case\ m\ of\ f_1...f_m\ end$$

Quant au prédicat P, il spécifie le type de la fonction construite; ce type peut dépendre de l'argument. Ainsi si $M : (I\ p_1 ... p_n\ t_1 ... t_r)$, alors $\langle P \rangle Case\ M\ of\ f_1 ... f_m\ end$ a pour type $(P\ t_1 ... t_r\ M)$. Le comportement de cet opérateur est régi par la règle de calcul suivante :

$$\langle P \rangle Case\ (c_i\ p_1 ... p_r\ a_1 ... a_s)\ of\ f_1 ... f_m\ end \rightarrow_\iota (f_i\ a_1 ... a_s)$$

Pour préserver la cohérence de la théorie, il y a des restrictions concernant la définition par filtrage suivant le type P et le type inductif éliminé. Nous analyserons avec plus de détails ces restrictions dans le cas des enregistrements, dans la section prochaine.

Le second opérateur est l'opérateur *Fixpoint* de point fixe permettant la définition de fonctions (et fonctionnelles) récursives, suivant le schéma primitif récursif. Nous allons considérer maintenant quelques exemples montrant l'utilisation de ces opérateurs. Nous donnons deux exemples, *EstNul* qui teste si un entier est nul, et *longueur* qui calcule la longueur d'une liste. Les constantes $True$ et $False$ sont de type $*$ (ou $Prop$), à ne pas confondre avec *true* et *false* de type *bool*. Elles sont définies de manière inductive; $True$ a un seul constructeur (ou preuve), et $False$ n'a aucun constructeur (donc aucune preuve close).

$Inductive\ True : * := I : True.$

$Inductive\ False : * := .$

$$Fixpoint\ EstNul := [n : nat]\langle Prop \rangle Case\ n\ of\ True \mid [p : nat] False\ end.$$

$$Fixpoint\ Longueur := [A : s][l : (list\ A)]\langle nat \rangle Case\ l\ of\ (nil\ A) \mid [a : A][l1 : (list\ A)](S\ (Longueur\ A\ l1))\ end.$$

Plus de détails concernant les types inductifs dans Coq se trouvent dans [49, 47, 119, 155].

Les noms des types inductifs et des constructeurs ne peuvent pas être expansés; ils sont en forme normale. Nous ignorons, par souci de simplicité, la règle $\rightarrow_\iota$.

Définition 3.30 (Mise sous forme normale de tête faible (Suite 3.8)) $\mathsf{Whnf}_\Gamma(M) \equiv M$ *si* M *est un nom de type inductif ou de constructeur.*

3.4 Enregistrements dépendants

3.4.1 Représentation de structures mathématiques

Dans l'état présent de notre théorie, il nous est impossible de définir le type de structures mathématiques telles que les groupes; on doit plutôt travailler dans un contexte de déclarations correspondant au type des groupes. Le fait de pouvoir représenter ces structures comme des types est primordial car cela permet de les abstraire et de définir des fonctions entre elles. Ainsi le *raisonnement abstrait* devient possible. En effet nous pouvons prouver des théorèmes abstraits sur des structures abstraites, par exemple une propriété P quelconque sur les groupes:

$$Thm : (G : Group)(P\ G)$$

puis instancier ce théorème avec des instances concrètes, par exemple le groupe $\langle \mathbb{Z}, + \rangle$. Cette preuve est alors, grâce à la puissance des types dépendants, tout simplement $(Thm\ \langle \mathbb{Z}, + \rangle)$.

Nous entendons par le terme de *structure mathématique*, une variété de constructions fort diverses:

- les structures comprenant un support et un ensemble d'opérations sur ce support; exemple, groupes, anneaux etc.
- les automates, graphes, catégories, topologies et ordres.
- les *types de données abstraits* (TDA) comme les piles. Un type correspond alors à une spécification d'un ADT, et les objets de ce type à ses implémentations.

En général, un type structure peut être vu comme contenant deux parties:

1. une *signature*, comprenant les types des composantes des objets: ensembles, fonctions, relations, etc.
2. un *prédicat* sur cette signature spécifiant les propriétés que doivent vérifier les objets de cette structure.

3.4.2 Types «somme forte»

Plusieurs théories de types (notamment ECC[103] et Martin-Löf[108]) généralisent le produit de type «$*$» au cas dépendant en introduisant les sommes fortes (appelées aussi Σ-types). Un Σ-type est de la forme $\Sigma x : A.B$; ses habitants sont des paires $\langle a, b \rangle$ avec $a : A$ et $b : B[x \leftarrow a]$. Intuitivement, il représente l'ensemble des paires dépendantes des éléments de A et B, $\{\langle a, b \rangle \mid a \in A \text{ et } b \in B[x \leftarrow a]\}$. Dans ECC[103] (calcul des constructions avec univers et somme fortes), les règles de formation sont données ci-dessous. Notons que les objets $\langle a, b \rangle_{\Sigma x:A.B}$ sont fortement typés pour éviter les ambiguïtés lors de l'inférence de type, dûes à la deuxième prémisse de (Pair).

$$(\Sigma)\ \frac{\Gamma \vdash A : Type_i \qquad \Gamma; x : A \vdash B : Type_i}{\Sigma x : A.B : Type_i}$$

$$(\text{Pair})\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[x \leftarrow a] \qquad \Gamma; x : A \vdash B : Type_i}{\langle a, b \rangle_{\Sigma x:A.B}}$$

Les Σ-types sont toutefois logiquement incohérents avec l'imprédicativité. Ajouter arbitrairement les Σ-types au niveau imprédicatif $Prop$ produirait un système incohérent où le paradoxe de Girard[43] peut être dérivé.

$$(\Sigma\bot)\ \frac{\Gamma; x : A \vdash B : Prop}{\Sigma x : A.B : Prop}$$

Il faut alors se restreindre au cas des petits Σ-types, similaires aux types existentiels.

$$(\Sigma')\ \frac{\Gamma \vdash A : Prop \qquad \Gamma; x : A \vdash B : Prop}{\Sigma x : A.B : Prop}$$

Les objets de type $\Sigma x : A.B$ peuvent être destructurés en utilisant les deux opérateurs de projection π_1 et π_2 obéissant aux deux règles suivantes:

$$(\pi_1)\ \frac{\Gamma \vdash p : \Sigma x : A.B}{\Gamma \vdash (\pi_1\ p) : A} \qquad (\pi_2)\ \frac{\Gamma \vdash p : \Sigma x : A.B}{\Gamma \vdash (\pi_2\ p) : B[x \leftarrow (\pi_1\ p)]}$$

Nous disposons aussi des deux réductions suivantes:

$$(\pi_1\ \langle a, b\rangle_{\Sigma x:A.B}) \rightarrow a \qquad (\pi_2\ \langle a, b\rangle_{\Sigma x:A.B}) \rightarrow b$$

L'imbrication des Σ-types nous permet de représenter des structures mathématiques complexes. En effet une structure comprenant n composantes (types, opérations et propriétés) peut être représentée par $\Sigma x_1 : A_1 .... \Sigma x_{n-1} : A_{n-1}.B$ (Le symbole Σ associe à droite). Les différentes composantes sont alors accessibles en composant les projections. La $i^{\text{ème}}$ $(0 < i < n)$ composante d'un objet $p : \Sigma x_1 : A_1 .... \Sigma x_{n-1} : A_{n-1}.B$ est fournie par:

$$(\pi_1\ (\underbrace{\pi_2\ (...(\pi_2}_{i-1 \text{ fois}}\ p))))$$

alors que la dernière composante est donnée par:

$$(\underbrace{\pi_2(...(\pi_2}_{n-1 \text{ fois}}\ p)))$$

À titre d'exemple, le type des groupes peut être défini par:

$$\begin{aligned} Group\ :=& \Sigma A : Type_j.\Sigma o : A \rightarrow A \rightarrow A.\Sigma i : A \rightarrow A.\Sigma e : A. \\ & (x, y, z : A)(o\ x\ (o\ y\ z)) = (o\ (o\ x\ y)\ z) \wedge \\ & (x : A)(o\ (i\ x)\ x) = x \wedge \\ & (x : A)(o\ x\ (i\ x)) = x \wedge \\ & (x : A)(o\ x\ e) = x \wedge \\ & (x : A)(o\ e\ x) = x \end{aligned}$$

On peut alors accéder aux composantes d'un groupe en utilisant les projections:

$$\begin{aligned} CarGrp\ &:=\ [m : Group](\pi_1\ m) \\ OpGrp\ &:=\ [m : Group](\pi_1\ (\pi_2\ m)) \\ InvGrp\ &:=\ [m : Group](\pi_1\ (\pi_2\ (\pi_2\ m))) \\ IdGrp\ &:=\ [m : Group](\pi_1\ (\pi_2\ (\pi_2\ (\pi_2\ m)))) \\ PropGrp\ &:=\ [m : Group](\pi_2\ (\pi_2\ (\pi_2\ (\pi_2\ m)))) \end{aligned}$$

Nous sommes restés vagues sur la nature exacte de l'égalité $=$; nous en dirons plus dans le chapitre 6 et lors de la définition des Setoïdes en 7.3.

3.4.3 Enregistrement comme cas particulier de type inductif

Il est possible de définir directement une notion primitive d'enregistrement dépendant avec étiquettes. Une telle extension a été considérée dans le système de preuves ALF par G. Betarte[23].

La théorie a aussi été étendue par une notion «naturelle» de sous-typage entre enregistrements; nous développerons aussi une relation de sous-typage analogue en utilisant des coercions implicites (chapitre 5).

Dans une théorie de types avec types inductifs, il est possible de définir les enregistrements comme cas particuliers des types inductifs. Ils correspondent aux types inductifs non-récursifs à un seul constructeur. Vue leur importance, nous disposons d'une syntaxe particulière pour leur définition. Remarquez que contrairement aux Σ-types, nos types enregistrements ne peuvent pas être créés dynamiquement; un objet enregistrement ne peut être utilisé dans un terme que si le type enregistrement correspondant a été déclaré au préalable.

Définition 3.31 (Définition d'un enregistrement)
• *La déclaration d'un enregistrement R avec m paramètres et n champs dans un contexte Γ est réalisée par la règle ci-dessous:*

$$(\text{CSTRUCT})\ \frac{\begin{array}{c} R \notin Dom(\Gamma) \qquad (L_i \notin Dom(\Gamma) \cup \{R, L_1, ..., L_{i-1}\})_{1\leq i\leq n} \\ (\Gamma; p_1 : P_1; ...; p_{i-1} : P_{i-1} \vdash P_i : s_i)_{1\leq i\leq m} \\ (\Gamma; p_1 : P_1; ...; p_m : P_m; L_1 : T_1; ...; L_{i-1} : T_{i-1} \vdash T_i : s_i)_{1\leq i\leq n} \end{array}}{\begin{array}{l} \Gamma;\ (Structure\ R\ [p_1 : P_1; ...; p_m : P_m] : s\ :=\ R\_Cons\ \{L_1 : T_1; ...; L_n : T_n\}); \\ R\_Cons : (p_1 : P_1)...(p_m : P_m)(L_1 : T_1)...(L_n : T_n)(R\ p_1...p_m) \vdash \end{array}}$$

• *La règle de typage des enregistrements est:*

$$(\text{STRUCT})\ \frac{(Structure\ R\ [p_1 : P_1; ...; p_m : P_m] : s\ :=\ R\_Cons\ \{L_1 : T_1; ...; L_n : T_n\}) \in \Gamma}{\Gamma \vdash R : (p_1 : P_1)...(p_m : P_m)s}$$

Chaque champ a une étiquette L_i et un type T_i. Évidemment toutes les étiquettes doivent être distinctes. La principale différence avec les enregistrements usuels des langages de programmation est que le type T_i d'une étiquette L_i peut dépendre des étiquettes précédentes (L_j avec $j < i$). Cet enregistrement est alors traduit en un type inductif de même nom:

$$Inductive\ R\ [p_1 : P_1; ....; p_m : P_m] : s := R\_Cons : (L_1 : T_1)...(L_n : T_n)(R\ p_1...p_m)$$

Le nom du constructeur R_Cons est optionnel; s'il n'est pas fourni, $Build\_R$ est utilisé. Le terme canonique $(R\_Cons\ p_1...p_m\ a_1...a_n)$ est l'objet de type $(R\ p_1...p_m)$ construit à partir des composantes $a_1...a_n$.

Les fonctions de projection sont automatiquement engendrées en utilisant l'opérateur de filtrage associé au type inductif R. L'opérateur de filtrage dans le cas de l'enregistrement R a la forme ci-dessous avec P de type $(R\ p_1...p_m) \to s'$.

$$\langle P\rangle Case\ m\ of\ [x_1 : T_1]...[x_n : T_n]f\ end$$

Cette opération de filtrage n'est permise que pour des valeurs particulières du couple $\langle s, s'\rangle$, qui sont $(Prop, Prop)$, $(Type_i, Prop)$ et $(Type_i, Type_j)$ avec $j \leq i$. En résumé, on ne peut pas utiliser l'opérateur $Case$ pour construire un terme dont la sorte est plus grande (au sens de $\preceq$) que celle de l'objet destructuré. Dans le cas des enregistrements, une projection L_i ne peut pas être définie

si on a $T_i : s_i$ et $s_i \succ s$. Cette approche est différente de celle des Σ-types; en effet, dans notre cas, on peut construire tous les enregistrements qu'on veut (même $(\Sigma\bot)$), mais on ne pourra pas toujours construire toutes ses projections; certains champs sont donc cachés. Toutefois, ils peuvent être utilisés pour la construction de fonctions par filtrage vers une sorte plus petite que s (la sorte de l'enregistrement). Exemple : soit un contexte contenant la définition du type A et une variable $f : Prop \to A$. Définissons maintenant un type enregistrement R; le champ X ne peut pas être défini car $X : Prop : Type_0 \succ Prop$.

$$Structure\ R : Prop := \{X : Prop; Y : A\}$$

Par contre ce champ peut être utilisé pour la construction d'un entier car $A : Prop \preceq Prop$. Le champ X n'est donc pas tout à fait inaccesible.

$$c := [m : R]\langle [m : R]A\rangle Case\ m\ of\ [x : Prop][y : A](f\ x)\ end\ :\ R \to A$$

Définition 3.32 (Définition d'une projection)

$$\frac{\begin{array}{c}(Structure\ R\ [p_1 : P_1; ...; p_m : P_m] : s\ :=\ R\_Cons\ \{L_1 : T_1; ...; L_n : T_n\}) \in \Gamma \\ (L_j \in VL(T_i) \Rightarrow L_j \in Def(\Gamma))_{j<i} \\ \Gamma; p_1 : P_1; ....; p_m : P_m; x : (R\ p_1 ... p_m) \vdash \\ (T_i[L_1 \leftarrow (L_1\ p_1 ... p_m\ x)] ... [L_{i-1} \leftarrow (L_{i-1}\ p_1 ... p_m\ x)]) : s_i \qquad s_i \preceq s\end{array}}{\Gamma; \left(\begin{array}{lcl} L_i & := & [p_1 : P_1] .... [p_m : P_m][x : (R\ p_1 ... p_m)] \\ & & \langle [y : (R\ p_1 ... p_m)](T_i[L_1 \leftarrow (L_1\ p_1 ... p_m\ y)] ... [L_{i-1} \leftarrow (L_{i-1}\ p_1 ... p_m\ y)])\rangle \\ & & Case\ x\ of\ [x_1 : T_1] ... [x_n : T_n] x_i\ end \\ & : & (p_1 : P_1) .... (p_m : P_m)(x : (R\ p_1 ... p_m)) \\ & & (T_i[L_1 \leftarrow (L_1\ p_1 ... p_m\ x)] ... [L_{i-1} \leftarrow (L_{i-1}\ p_1 ... p_m\ x)]) \end{array}\right) \vdash}$$

La projection L_i a alors le comportement attendu sur les termes canoniques de type $(R\ p_1 ... p_m)$, grâce à la combinaison des réductions $\to_\beta$, $\to_\delta$ et $\to_\iota$.

$$(L_i\ p_1 ... p_m\ (R\_Cons\ p_1 ... p_m\ a_1 ... a_n))$$

$$\downarrow_\delta$$

$$\begin{array}{l}((([p_1 : P_1] .... [p_m : P_m][x : (R\ p_1 ... p_m)] \\ \langle [y : (R\ p_1 ... p_m)](T_i[L_1 \leftarrow (L_1\ p_1 ... p_m\ y)] ... [L_{i-1} \leftarrow (L_{i-1}\ p_1 ... p_m\ y)])\rangle \\ Case\ x\ of\ [x_1 : T_1] ... [x_n : T_n] x_i\ end)\ p_1 ... p_m\ (R\_Cons\ p_1 ... p_m\ a_1 ... a_n))\end{array}$$

$$\downarrow^*_\beta$$

$$\begin{array}{l}(\langle [y : (R\ p_1 ... p_m)](T_i[L_1 \leftarrow (L_1\ p_1 ... p_m\ y)] ... [L_{i-1} \leftarrow (L_{i-1}\ p_1 ... p_m\ y)])\rangle \\ Case\ (R\_Cons\ p_1 ... p_m\ a_1 ... a_n)\ of\ [x_1 : T_1] ... [x_n : T_n] x_i\ end)\end{array}$$

$$\downarrow_\iota$$

$$([x_1 : T_1] ... [x_n : T_n] x_i\ a_1 ... a_n)$$

$$\downarrow^*_\beta$$

$$a_i$$

Nous avons plusieurs remarques concernant la règle de construction des projections.

- Le nom de la $i^{\text{ème}}$ projection est identique à celui du $i^{\text{ème}}$ champ.
- Il existe une deuxième raison (en plus de celle citée plus haut) pour laquelle une projection n'est pas définissable. Si le type de L_i dépend d'un champ L_j non définissable, alors L_i est aussi non définissable (voir la deuxième prémisse de la règle ci-dessous).
- Le type du résultat des projections tient compte de l'éventuelle dépendance de L_i vis-à-vis des étiquettes précédentes.

Les noms des enregistrements, des projections et des constructeurs de projections ne peuvent pas être expansés; ils sont dejà en forme normale. Nous modifions la fonction de mise en forme normale de tête pour prendre en compte la règle de réduction des projections.

Définition 3.33 (Mise sous forme normale de tête faible (Suite 3.30))
Nous étendons l'algorithme 3.30 par:

- $\mathsf{Whnf}_\Gamma(M) = M$ *si* M *est un nom d'enregistrement, de projection ou de constructeur (*R_Cons*) de d'enregistrement.*
- $$\mathsf{Whnf}_\Gamma((L_i\ p_1...p_m\ a\ b_1...b_k)) = \begin{cases} \mathsf{Whnf}_\Gamma((a_i\ b_1...b_k)) & si\ \mathsf{Whnf}_\Gamma(a) \equiv \\ & (R\_Cons\ q_1...q_m\ a_1...a_n) \\ (L_i\ p_1...p_n\ a\ b_1...b_k) & sinon \end{cases}$$

Lemme 3.8 (Termes en forme normale de tête) *Les termes en forme normale de tête ont finalement l'une des formes:*

- x, s, $(x : A)B$,$[x : A]b$, $(x\ a_1...a_n)$, c *où* c *est un nom d'un type inductif, d'un constucteur, d'un enregistrement, d'une projection ou d'un constructeur d'un enregistrement.*
- $(c\ a_1...a_n)$ *où* c *est un nom d'un type inductif, d'un constucteur, d'un enregistrement ou d'un constructeur d'un enregistrement.*
- $(L_i\ p_1...p_m\ a\ b_1...b_k)$ *où* L_i *est une projection d'un enregistrement* R *(possèdant* m *paramètres), et tel que la forme normale de tête de* a *n'est pas de la forme* $(R\_Cons\ q_1...q_m\ a_1...a_n)$.

PREUVE Par simple analyse de l'algorithme 3.8 étendu par les définitions 3.30 et 3.33. ■

Nous donnons maintenant la définition des groupes en utilisant nos enregistrements:

$$\begin{array}{l} Structure\ Group\ : Type\ := \\ \{CarGrp : Type; \\ \ OpGrp : CarGrp \to CarGrp \to CarGrp; \\ \ InvGrp : CarGrp \to CarGrp; \\ \ IdGrp : CarGrp; \\ \ PropGrp : (x, y, z : A)(OpGrp\ x\ (OpGrp\ y\ z)) = (OpGrp\ (OpGrp\ x\ y)\ z) \wedge \\ \qquad\qquad (x : A)(OpGrp\ (InvGrp\ x)\ x) = x \wedge \\ \qquad\qquad (x : A)(OpGrp\ x\ (InvGrp\ x)) = x \wedge \\ \qquad\qquad (x : A)(OpGrp\ x\ IdGrp) = x \wedge \\ \qquad\qquad (x : A)(OpGrp\ IdGrp\ x) = x\} \end{array}$$

Nous verrons dans les chapitres suivants la nature exacte de l'égalité utilisée (=).

Notation 3.1 (égalité définitionnelle) *Nous appellerons dorénavant* égalité définitionnelle, *la congruence associée à toute combinaison des réductions β, δ, η et ι contenant la β-réduction. Nous la noterons généralement par $\cong$, sans précisions supplémentaires.*

3.5 Conclusion

À l'issue de ce chapitre, nous disposons d'un langage logique pouvant servir de cadre formel pour la formalisation de théories mathématiques. Les principales caractéristiques de ce langage sont:

- C'est une théorie des types comprenant une sorte imprédicative *Prop* et une hiérarchie d'univers prédicatifs $Type_i$ (avec $i \in \mathbb{N}$).
- Un mécanisme de définition permet de nommer les constantes.
- Un mécanisme de gestion d'univers flottants nous permet de confondre la hiérarchie d'univers $Type_i$ ($i \in \mathbb{N}$) avec une seule sorte *Type*, nous évitant ainsi la duplication de constantes à différents niveaux d'univers.
- Un mécanisme de définitions inductives.
- Un mécanisme de définition d'enregistrements dépendants, adéquat notamment pour la représentation de structures algébriques.

Seconde partie

Outils Génériques

Introduction

Cette partie est consacrée à la description d'outils génériques sensés faciliter l'activité de formalisation dans les systèmes de preuves basés sur des théories de types. Nous décrivons trois outils que nous avons conçus et implémentés dans Coq[2]. Ces outils se divisent en deux familles:

Outils de modélisation. Ils tendent à faciliter la représentation de notions mathématiques en théorie des types. Leur objectif est d'améliorer l'expressivité des théories des types en la rendant plus intuitive et plus succincte, sans pour autant augmenter son pouvoir d'expression. Pour cela, les outils de sous-termes implicites et de coercions implicites simulent des mécanismes généraux d'abus de notation, et définissent une syntaxe implicite pour une théorie de types donnée. Dans cette syntaxe implicite, certaines informations (termes) peuvent être omises car peuvent être reconstruites à partir du contexte. Cette reconstruction de termes s'effectue lors du processus de typage et nécessite ainsi la modification de l'algorithme d'inférence vu dans le chapitre précédent. Ces outils tentent ainsi de combler la distance qui sépare les textes formels, compréhensibles par la machine, et les textes mathématiques usuels, plus lisibles et plus compréhensibles par l'humain.
Nous voulons faire une mise au point. Le langage implicite n'est qu'une interface pour le langage explicite; en aucun cas, il ne se substitue à lui. Ainsi il ne s'agit pas de construire une nouvelle théorie des preuves à base du langage implicite. En effet, la «théorie officielle» est toujours dans notre cas les PTS présentés dans la première partie de cette thèse; il serait dommage d'abandonner ces théories bien établies et dont la méta-théorie est bien connue aujourd'hui. Ces outils ne sont pourtant pas de simples mécanismes de «macro-génération» permettant des effets de «sucre syntaxique». Ils se rapprochent plutôt des techniques de compilation des langages de programmation où on compile un langage évolué (que nous appelons dans notre cas, «langage implicite») vers un langage de plus bas niveau («langage explicite»).

Outils de démonstration. Le niveau de détail requis dans une preuve formelle est beaucoup plus grand que celui dans une preuve informelle. Le raisonnement dans un système de preuve nécessite que chaque étape de raisonnement soit décomposée en des règles primitives de très bas niveau. Beaucoup de travaux ont étés réalisés pour remédier à cet handicap des systèmes de preuves; la communauté de la démonstration (semi-)automatique a développé de nombreux outils: tactiques, procédures de décision, techniques d'automatisation de la récurrence etc. Tous ces outils visent à automatiser la génération de certaines preuves. Nous avons développé, pour notre part, une bibliothèque de tactiques de réécriture pour la simplification d'expressions ou la démonstration de buts équationnels.

[2]Chet Murthy est le principal concepteur et implémenteur de l'outil des «sous-termes implicites»; notre contribution se limite à quelques modifications et extensions (voir le chapitre 4).

Nos trois outils sont dits génériques car ils s'appliquent dans diverses situations. Pour l'outil des sous-termes implicites, peu importe l'application, il suffit d'indiquer l'emplacement des sous-termes implicites dans un terme. De même pour le mécanisme des coercions implicites où l'utilisateur peut déclarer à sa guise ses propres coercions suivant son domaine d'application. Quant aux tactiques de réécriture, l'utilisateur a le choix de la relation de réécriture, des règles à utiliser ainsi que de la stratégie (décrite dans un langage spécifique).

En outre, ces outils sont décrits dans le cadre de la famille des théories de types PTS. Nous décrivons ci-dessous avec exactitude la famille de théories de types dans laquelle nous définissons nos trois outils. Pour cela nous reprenons les conditions des PTS semi-pleins (voir 3.1.2):

- PTS $\langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ avec une réduction fortement normalisable et confluente, nous garantissant la décidabilité de l'égalité définitionnelle.
- le PTS doit être semi-plein (définition 3.5) et fonctionnel (définition 2.12).
- les différentes conditions annexes présentes dans les règles de typage des PTS doivent être décidables.

De plus, nous prenons en compte les constantes globales, et les enregistrements dépendants. Nous abandonnons par contre les types inductifs dont la présentation classique est trop technique et alourdit considérablement la description des algorithmes.
Le traitement de la hiérarchie d'univers $\{Type_i \mid i \in \mathbb{N}\}$ est tout à fait particulier. Il s'efféctue en deux étapes:

- le typage est réalisé par nos nouveaux algorithmes en considérant une seule sorte anonyme $Type$ à la place de la hiérarchie $\{Type_i \mid i \in \mathbb{N}\}$. Le résultat obtenu est un terme «partiellement explicite» dans lequel nous avons inséré les sous-termes implicites et les coercions implicites appropriées.
- le terme obtenu est alors typé par un algorithme «standard» gérant les univers flottants (voir l'algorithme 3.27).

Chapitre 4

Sous-termes implicites

4.1 Introduction

Dans la formulation de Church du λ-calcul typé, toute l'information de type doit être fournie dans les termes sous forme d'abstractions et d'applications typées. Cette approche s'oppose à celle de Curry où les types ne sont qu'une propriété des termes du λ-calcul non typé. C'est cette formulation, appelée aussi *polymorphisme implicite* (par opposition au *polymorphisme explicite* des calculs polymorphiques à la Church), qui est adoptée dans le langage ML. Le problème d'inférence de type pour ML est décidable. On ne dispose malheureusement que de très peu de résultats de décidabilité pour des systèmes plus expressifs comme le système F (voir [19]). L'approche de Church a l'avantage de disposer d'algorithmes simples pour l'inférence et le contrôle de type pour tous les PTS du λ-cube, mais au prix d'une redondance de l'information de type dans les termes.

Lorsque nous écrivons un terme dans un λ-calcul typé, nous voulons souvent omettre certains arguments d'une fonction pouvant être inférés à partir des arguments suivants de la fonction. Par exemple, considérons l'opérateur de composition de fonctions ci-dessous:

$$comp \;=\; [A:*][B:*][f:A \to B][C:*][g:B \to C][x:A](g\ (f\ x))$$

Nous pouvons l'utiliser, par exemple, pour définir la composition des deux fonctions $longueur$: $(list\ bool) \to nat$ et $premier : nat \to bool$ avec la construction «explicite»:

$$(comp\ (list\ bool)\ nat\ longueur\ bool\ premier)$$

Cette écriture est toutefois redondante; lier f à $longueur$ et g à $premier$ devrait lier respectivement les variables A, B et C à $(list\ bool)$, nat et $bool$. On aimerait donc écrire $(comp\ longueur\ premier)$, comme dans l'écriture usuelle; c'est le terme «implicite». Il est clair que les arguments A, B et C peuvent être synthétisés comme étant des composantes du type de f et g.

Le calcul implicite est considéré comme un langage informel, permettant de se rapprocher des notations usuelles en mathématiques (et dans les langages de programmation), nous évitant ainsi d'écrire de gros termes dans le calcul explicite.

L'idée est de combiner les deux approches de typage à la Curry et à la Church où certaines informations de type peuvent être omises et reconstruites par l'algorithme d'inférence de type. Il ne s'agit pas d'insérer arbitrairement des arguments implicites; les endroits concernés doivent être

signalés par l'utilisation du symbole «?». Ainsi (*comp longueur premier*) est mal typé, et notre exemple précédent s'écrit:

$$(comp\ ?\ ?\ longueur\ ?\ premier)$$

Écriture qui reste plus lisible que le terme complètement explicite. Nous montrons dans la section 4.6 comment se passer des «?» et accepter directement le terme (*comp longueur premier*). La tâche de l'algorithme d'inférence est de remplacer chaque terme inconnu «?» par un terme explicite de telle sorte que le terme explicite résultant soit bien typé.

Dans le cas de certains PTS, où les types peuvent dépendre des termes, des termes aussi peuvent être automatiquement inférés. Donnons un exemple; soit le type des vecteurs défini dans la section 3.3, le constructeur qui rajoute un élément en tête d'un vecteur est:

$$consv : (A : *)(n : nat)A \to (vect\ A\ n) \to (vect\ A\ (S\ n))$$

Considérons maintenant un vecteur $v : (vect\ bool\ 5)$ de cinq booléens et un booléen $b : bool$. On peut alors écrire $(consv\ ?\ ?\ b\ v)$; le deuxième terme inconnu correspond à un terme, le nombre 5.

Nous utilisons de manière intensive les sous-termes implicites[1] dans notre développement de la théorie des catégories dans Coq (voir troisième partie de cette thèse). Ce chapitre constitue une documentation pour le mécanisme des sous-termes implicites de Coq. L'article [132] de R. Pollack décrit le principe général du mécanisme de sous-termes implicites implémenté dans LEGO; celui de Coq est cependant plus général (les différences sont signalées tout au long de ce chapitre). La donnée d'un algorithme précis reste le seul moyen de comprendre dans les détails ce mécanisme. Toutefois nous ne faisons pas l'étude méta-théorique de ce mécanisme.

Le mécanisme de sous-termes implicites de Coq a été conçu et implémenté par C. Murthy. Il repose de manière essentielle sur un algorithme d'unification de termes; celui-ci, initialement implémenté aussi par C. Murthy, a été sensiblement modifié (amélioré?) par C. Paulin-Mohring et l'auteur.

Notre contribution dans ce chapitre consiste en deux points:

- section 4.5: nous développons une heuristique pour un meilleur traitement de l'unification en présence d'enregistrements. Nous montrons aussi comment cette extension permet de surcharger des noms de fonctions (section 4.7).
- section 4.6: développement d'un mécanisme simple pour le calcul et l'insertion automatique des arguments implicites d'une fonction, cachant ainsi les disgracieux «?».

4.2 Principe et Définitions

Nous adoptons l'approche de R. Pollack: la sémantique du calcul implicite est donnée en définissant une translation du calcul implicite vers le calcul explicite. Cette translation est un algorithme d'inférence de types. Le langage implicite est donc seulement une interface pour le calcul explicite. L'approche de M. Hagiya et Y. Toda[67] est quelque peu différente de la nôtre. Dans [67], la vérification de type se fait directement dans le calcul implicite sans expliciter les termes implicites.

[1] Le terme consacré est plutôt *arguments implicites*. Mais le terme de *sous-termes implicites* est plus approprié car dans notre cas, les termes implicites ne sont pas toujours des arguments d'applications de fonctions.

Cette approche a notamment pour avantage de notifier de manière plus appropriée les erreurs de typage. Elle comporte néanmoins quelques complications, comme la définition de la réduction dans le calcul implicite. Cette réduction doit préserver l'unicité de la translation vers le calcul explicite: si M se réduit en N, et M a une translation unique M_e, alors N a aussi une translation unique N_e, et M_e se réduit en N_e dans le calcul explicite. Cette propriété permet de tester la conversion entre deux termes implicites en les réduisant à un terme implicite commun, sans les expliciter. Pour garantir la préservation de l'unicité de la translation vers le calcul explicite, plusieurs restrictions sont appliquées au calcul implicite, et des annotations (correspondant au cast) sont utilisées.

Le principe général est de laisser implicites certains sous-termes d'un terme, en les désignant par une inconnue «?». Nous obtenons ainsi un langage plus flexible. Il n'y a pas de restrictions quant à l'endroit d'apparition de ces inconnues. En LEGO, les inconnues sont toujours arguments d'une application; de plus, une inconnue ne peut pas être le dernier argument d'une application.

Définition 4.1 (Syntaxe concrète implicite(1))

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

La synthèse des inconnues repose sur une idée simple: pour chaque sous-terme, on confronte son type inféré avec son type attendu (s'il existe). Ces deux types peuvent, a priori, contenir des inconnues; il s'agit donc d'*unification*. Le cas typique est celui de l'application. Soit à typer $(a\ b)$, avec a de type $(x:B)A$ et b de type B'; B et B' sont respectivement les types attendu et inféré de b. Sans inconnues, il suffit de demander à ce que B et B' soient convertibles. En présence d'inconnues, il s'agit d'associer des termes (à trouver) à ces inconnues tel que les termes résultants soient convertibles. Il s'agit donc de résolution d'équations dans un λ-calcul typé, appelée aussi *unification d'ordre supérieur*[77, 56].

Voyons le processus sur un exemple. Soit le terme ci-dessous à typer:

$$[v:(vect\ ?\ 5)](consv\ ?\ ?\ true\ v)$$

Pour distinguer les inconnues, nous les numérotons dans l'ordre de leur apparition, de la gauche vers la droite. Le terme devient alors:

$$[v:(vect\ ?_1\ 5)](consv\ ?_2\ ?_3\ true\ v)$$

Le tableau ci-dessous montre le typage de chacun de ses sous-termes. D'après les types (définis dans le contexte) de *vect* et *consv*, les types des différentes inconnues sont $?_1 : *$, $?_2 : *$ et $?_3 : nat$. Ce sont les lignes du tableau possédant un type attendu mais pas de type inféré (étapes 2, 7 et 9). Les points cruciaux sont les étapes 11 et 13; le premier permet d'associer *bool* à $?_2$, et le second permet de déduire que $?_2 := ?_1$ et $?_3 := 5$ par unification de $(vect\ ?_1\ 5)$ et $(vect\ ?_2\ ?_3)$. Par transitivité, $?_1 := bool$. Le terme explicite est alors:

$$[v:(vect\ bool\ 5)](consv\ bool\ 5\ true\ v)$$

Étape	Terme	Type inféré	Type attendu
1	$vect$	$* \to nat \to *$	
2	$?_1$		$*$
3	$(vect\ ?_1)$	$nat \to *$	
4	5	nat	nat
5	$(vect\ ?_1\ 5)$	$*$	
6	$consv$	$(A:*)(n:nat)A \to (vect\ A\ n) \to (vect\ A\ (S\ n))$	
7	$?_2$		$*$
8	$(consv\ ?_2)$	$(n:nat)?_2 \to (vect\ ?_2\ n) \to (vect\ ?_2\ (S\ n))$	
9	$?_3$		nat
10	$(consv\ ?_2\ ?_3)$	$?_2 \to (vect\ ?_2\ ?_3) \to (vect\ ?_2\ (S\ ?_3))$	
11	$true$	$bool$	$?_2$
12	$(consv\ ?_2\ ?_3\ true)$	$(vect\ ?_2\ ?_3) \to (vect\ ?_2\ (S\ ?_3))$	
13	v	$(vect\ ?_1\ 5)$	$(vect\ ?_2\ ?_3)$
14	$(consv\ ?_2\ ?_3\ true\ v)$	$(vect\ ?_2\ (S\ ?_3))$	

L'idée est donc de transformer chaque inconnue «syntaxique» en une variable d'unification (on dit aussi «métavariable» ou «variable existentielle» pour les distinguer des variables du λ-calcul) distincte $?_i$ (nous commençons leur numérotation par 1).

Définition 4.2 (Syntaxe abstraite implicite(1))

$$T \quad ::= \quad s \mid x \mid c \mid [x:T]T \mid (T\ T) \mid (x:T)T \mid ?_i \quad (i \in \mathbb{N}^*)$$

Chaque métavariable a un type attendu, obtenu lors de sa première rencontre. Les métavariables sont instanciées par des termes lors du processus d'inférence. À chaque métavariable, nous associons un contexte répertoriant tous les noms (variables, constantes etc.) qui lui sont accessibles. Ainsi dans le terme (artificiel) suivant $[x:A](f...([y:B](g...?_1...))...?_2...)$, les variables x et y sont accessibles pour $?_1$, alors que seul x est accessible pour $?_2$. L'opération de α-conversion est ici capitale; dans $[x:A](f...([y:B](g...?_1...))...([y:C](h...?_2...))...)$, les deux métavariables $?_1$ et $?_2$ dépendent d'une variable y différente. Une façon simple de s'affranchir de tout souci de renommage lors de l'inférence de type, est de supposer que toutes les variables apparaissant dans les lieurs sont différentes (quitte à les renommer avant inférence).

Une *signature* enregistre toutes les métavariables rencontrées; les métavariables instanciées sont dites *définies* et celles qui ne le sont pas sont dites *déclarées*. Les signatures sont assez semblables aux contextes: les métavariables déclarées et les métavariables définies correspondent respectivement aux variables et aux constantes. La seule différence est que chaque métavariable possède son *contexte accessible*.

Définition 4.3 (Signature) *Une signature est une séquence finie de métavariables déclarées et de métavariables définies.*

$$\begin{array}{lcll} \Sigma & ::= & [] & \textit{signature vide} \\ & \mid & \Sigma;(\Gamma \vdash ?_i : A) & \textit{métavariable déclarée} \\ & \mid & \Sigma;(\Gamma \vdash ?_i : A := a) & \textit{métavariable définie} \end{array}$$

- *Les définitions vues dans 3.1 pour les contextes, Decl, Def, Dom, $\Sigma_{?_i}$ et $\Sigma^{val}_{?_i}$ s'appliquent aussi pour les signatures.*
- *Nous désignerons par $\Sigma^{ctxt}_{?_i}$ le contexte accessible d'une métavariable $?_i$ dans Σ.*
- *Nous notons par $MV(M)$ l'ensemble des métavariables de M.*

La notion de variable libre est plus difficile à définir pour des termes avec métavariables. En effet, on ne peut pas connaître a priori l'ensemble des variables libres d'un terme contenant des métavariables non définies. On peut par contre le «majorer», sachant que seules les variables de $\Sigma^{ctxt}_{?_i}$ peuvent apparaître dans la définition de $?_i$. D'où la définition de VL^* ci-dessous.

Définition 4.4 (Variables libres) *L'ensemble VL^* est défini relativement à une signature Σ.*

$$\begin{array}{rcl} VL^*_\Sigma(x) & = & \{x\} \\ VL^*_\Sigma(s) & = & \emptyset \\ VL^*_\Sigma(c) & = & \emptyset \\ VL^*_\Sigma(?_i) & = & \begin{cases} VL^*_\Sigma(\Sigma^{val}_{?_i}) & si\ ?_i \in Def(\Sigma) \\ \Sigma^{ctxt}_{?_i} & si\ ?_i \in Decl(\Sigma) \end{cases} \\ VL^*_\Sigma([x:A]b) & = & VL^*_\Sigma(A) \cup (VL^*_\Sigma(b) \setminus \{x\}) \\ VL^*_\Sigma((x:A)B) & = & VL^*_\Sigma(A) \cup (VL^*_\Sigma(B) \setminus \{x\}) \\ VL^*_\Sigma((a\ b)) & = & VL^*_\Sigma(a) \cup VL^*_\Sigma(b) \end{array}$$

La définition d'une métavariable se déroule en deux étapes. Elles sont d'abord déclarées puis instanciées. Ainsi nous disposons d'une opération pour définir une métavariable déjà déclarée.

Définition 4.5 (Instanciation d'une métavariable) *La signature $\Sigma[?_i := a]$ est obtenue par la transformation de la métavariable $?_i$ déclarée dans Σ en une métavariable définie avec la valeur a.*

$$\begin{array}{rcl} []?_i := a] & = & \bot \\ (\Sigma;(\Gamma \vdash ?_i : A))[?_i := a] & = & \Sigma;(\Gamma \vdash ?_i : A := a) \\ (\Sigma;(\Gamma \vdash ?_j : A))[?_i := a] & = & (\Sigma[?_i := a]);(\Gamma \vdash ?_j : A) \\ (\Sigma;(\Gamma \vdash ?_j : A := b))[?_i := a] & = & (\Sigma[?_i := a]);(\Gamma \vdash ?_j : A := b) \end{array}$$

4.3 Unification

Nous avons bien vu sur l'exemple précédent que les métavariables sont instanciées à travers un algorithme d'unification. Celui-ci est en fait au centre du processus d'inférence de types pour le langage implicite.
Deux termes d'une théorie sont dits *unifiables* s'il existe des valeurs pour les variables libres (ou métavariables) présentes dans ces termes, de façon à rendre ces termes égaux relativement aux axiomes de la théorie. Il s'agit dans notre cas d'unification d'ordre supérieur car notre théorie sous-jacente est un λ-calcul typé. Cette unification a été prouvée indécidable[77]: il n'existe pas d'algorithme qui décide si deux termes sont unifiables ou non. Huet[77] a toutefois développé un algorithme d'unification semi-décidable, qui termine et donne une solution si les deux termes sont unifiables, mais qui peut ne pas terminer en présence de termes non unifiables. Goldfarb[63] a montré que même l'unification du second ordre (restreignant le type des métavariables à être au

plus du second ordre, c'est-à-dire des fonctions simples) est indécidable. Des cas particuliers décidables de l'unification d'ordre supérieur ont été considérés, notamment le cas des *motifs*[109] (ou *patterns*, termes où chaque symbole de fonction n'est appliqué qu'à des variables toutes distinctes) où l'unification est décidable et où il existe unificateur principal (dont toutes les autres solutions peuvent se déduire).
D'autres algorithmes ont aussi été développés pour des extensions du λ-calcul simplement typé. Ainsi C. M. Elliott et D. Pym ont développé le premier algorithme d'unification prenant en compte les types dépendants. Dans sa thèse, G. Dowek[56] a développé un algorithme d'unification pour les PTS du cube de Barendregt (admettant les types dépendants, le polymorphisme et les constructeurs de types).

Nous adoptons pour notre part un algorithme ad hoc, non complet mais avec un bon comportement en pratique. Cet algorithme prend en entrée un contexte et une signature et une paire de termes formant une équation.

Définition 4.6 (Équation) *Une équation est une paire de termes (M, N) valides relativement à un contexte Γ et une signature Σ donnés. Une telle équation sera notée $M = N$ ou $\Gamma \vdash M = N$.*

La solution d'une unification réussie est une substitution associant à chaque métavariable une valeur. Ce rôle est tenu par les signatures. Nous définissons maintenant l'application d'une signature à un terme.

Définition 4.7 (Application d'une signature) *Nous notons par $\Sigma(M)$ le résultat de l'application de la signature Σ au terme M.*

$$\begin{array}{rcl}
\Sigma(x) & = & x \\
\Sigma(s) & = & s \\
\Sigma(c) & = & c \\
\Sigma(?_i) & = & \begin{cases} \Sigma(\Sigma^{val}_{?_i}) & si\ ?_i \in Def(\Sigma) \\ ?_i & si\ ?_i \in Decl(\Sigma) \end{cases} \\
\Sigma([x:A]b) & = & [x:\Sigma(A)]\Sigma(b) \\
\Sigma((x:A)B) & = & (x:\Sigma(A))\Sigma(B) \\
\Sigma((a\ b)) & = & (\Sigma(a)\ \Sigma(b))
\end{array}$$

Pour unifier deux termes, nous commençons par calculer leur forme normale de tête faible, sans expansion de constantes.

Définition 4.8 (Fonction de normalisation en forme normale de tête faible) *La fonction de normalisation (sans expansion de constantes) en forme normale de tête faible est:*

$$\begin{array}{rcl}
\mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(?_i) & = & \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(\Sigma^{val}_{?_i})\ si\ ?_i \in Def(\Sigma) \\
\mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((([x:A]b)a\ a_1...a_n)) & = & \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(b[x \leftarrow a]\ a_1...a_n) \\
\mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((?_i\ a_1...a_n)) & = & \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((\Gamma^{val}_{?_i}\ a_i...a_n))\ si\ ?_i \in Def(\Sigma) \\
\mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((L_i\ p_1...p_m\ a\ b_1...b_k)) & = & \begin{cases} \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((a_i\ b_1...b_k)) & si\ \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(a) \equiv \\ & \quad (R_Cons\ q_1...q_m\ a_1...a_n) \\ (L_i\ p_1...p_n\ a\ b_1...b_k) & sinon \end{cases} \\
\mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(M) & = & M\ pour\ tous\ les\ autres\ cas
\end{array}$$

• *Les fonctions* $\mathsf{Whnf}_{\Gamma,\Sigma}$ *et* $\downarrow_{\Gamma,\Sigma}$ *sont respectivement similaires aux définitions 3.8 et 3.9; les métavariables définies sont traitées comme des constantes, et les métavariables déclarées (et non définies) comme des variables.*

La réponse de l'algorithme d'unification n'est pas seulement binaire (oui ou non); elle consiste en l'ensemble des équations restant à unifier. En effet certaines équations ne peuvent être résolues immédiatement et sont retardées en attente de l'instanciation de certaines métavariables. Ce sont les équations dont l'un des deux membres est un terme *flexible*. Un cas particulier en est $?_i = ?_j$.

Définition 4.9 (Terme flexible) *Un terme est dit flexible s'il est de la forme* $(?_i\ a_1 ... a_n)$ *où* $?_i$ *est une métavariable non définie.*

La forme générale d'un terme flexible ne peut être connue qu'à l'instanciation de la métavariable de tête. En effet la réduction en forme normale de tête faible ne peut être poursuivie. La résolution des équations retardées est entreprise dès l'instanciation de leur métavariable de tête. Nous noterons par $Dep(?_i, \Gamma \vdash M = N)$ lorsque l'équation $M = N$ (valide sous le contexte Γ) est en attente de l'instanciation de $?_i$.

Définition 4.10 (Prédicat Dep**)** *Le prédicat* $Dep(?_i, .)$ *est défini par:*

$$Dep(?_i, \Gamma \vdash M = N) \text{ si et seulement si } M \equiv (?_i\ a_1 ... a_n) \vee N \equiv (?_i\ b_1 ... b_n) \quad (n \geq 0)$$

Nous organisons les équations restant à résoudre sous forme de séquence, que nous noterons par la lettre E (éventuellement avec indice).

Définition 4.11 (Algorithme d'unification)

$\mathsf{Unify}(\Sigma, E, \Gamma \vdash M = N) =$

$$\begin{cases} \Sigma, E & si\ M \equiv N \\ \begin{cases} \Sigma, E & si\ M \downarrow_{\Gamma,\Sigma} N \\ \bot & sinon \end{cases} & si\ MV(M) \cup MV(N) = \emptyset \\ \mathsf{Unify}'(\Sigma, E, \Gamma \vdash \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(M) = \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}(N)) & sinon \end{cases}$$

Unify' *est défini par analyse de cas comme suit:*

• $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash M = M) = \Sigma, E$

• $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash s_1 = s_2) = \begin{cases} \Sigma, E & si\ s_1 \equiv s_2 \\ \bot & sinon \end{cases}$

• $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash [x : A_1]b_1 = [x : A_2]b_2) = \mathsf{Unify}(\mathsf{Unify}(\Sigma, E, \Gamma \vdash A_1 = A_2), (\Gamma; x : A_1) \vdash b_1 = b_2)$

• $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (x : A_1)B_1 = (x : A_2)B_2) =$

$\mathsf{Unify}(\mathsf{Unify}(\Sigma, E, \Gamma \vdash A_1 = A_2), (\Gamma; x : A_1) \vdash B_1 = B_2)$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (x\ a_1 ... a_n) = (y\ b_1 ... b_m)) =$

$$\begin{cases} \Sigma_n, E_n & si\ x \equiv y\ et\ n \equiv m \\ & \quad avec\ \mathsf{Unify}(\Sigma, E, \Gamma \vdash a_1 = b_1) \equiv \Sigma_1, E_1 \\ & \quad \mathsf{Unify}(\Sigma_{i-1}, E_{i-1}, \Gamma \vdash a_i = b_i) \equiv \Sigma_i, E_i \ \ (i\ allant\ de\ 2\ à\ n) \\ \bot & sinon \end{cases}$$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (?_i\ a_1 ... a_n) = (?_j\ b_1 ... b_m)) = \Sigma, (E; \Gamma \vdash (?_i\ a_1 ... a_n) = (?_j\ b_1 ... b_m))$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash ?_i = M) =$

$$\begin{cases} \Sigma_n, E_n & si\ VL^*_\Sigma(M) \subseteq \Sigma^{ctxt}_{?_i} \\ & \quad avec\ \mathsf{Unify}(\Sigma[?_i := M], E \setminus (\Gamma_1 \vdash a_1 = b_1), \Gamma_1 \vdash a_1 = b_1) \equiv \Sigma_1, E_1 \\ & \quad \mathsf{Unify}(\Sigma_{j-1}, E_{j-1} \setminus (\Gamma_j \vdash a_j = b_j), \Gamma_j \vdash a_j = b_j) \equiv \Sigma_j, E_j \\ & \quad\ (j\ allant\ de\ 2\ jusqu'au\ plus\ petit\ n\ vérifiant \\ & \quad\ \ Prem_{Dep(?_i,.)}(E_n) \equiv \bot,\ Prem_{Dep(?_i,.)}(E) \equiv (\Gamma_1 \vdash a_1 = b_1) \\ & \quad\ \ et\ Prem_{Dep(?_i,.)}(E_{j-1}) \equiv (\Gamma_j \vdash a_j = b_j)\ \ (2 \leq j \leq n)) \\ \bot & sinon \end{cases}$$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash M = ?_i) = \mathsf{Unify}'(\Sigma, E, \Gamma \vdash ?_i = M)$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (M\ a_1 ... a_n) = (?_i\ b_1 ... b_m)) = \begin{cases} S_1 & si\ S_1 \not\equiv \bot \\ S_2 & sinon \end{cases}$ *avec*

$$S_1 = \begin{cases} \Sigma_{m+1}, E_{m+1} & si\ m \leq n \\ & \quad avec\ \mathsf{Unify}'(\Sigma, E, \Gamma \vdash ?_i = (M\ a_1 ... a_{n-m})) \equiv \Sigma_1, E_1 \\ & \quad \mathsf{Unify}(\Sigma_j, E_j, \Gamma \vdash a_{n-m+j} = b_j) \equiv \Sigma_{j+1}, E_{j+1} \\ & \quad\ (j\ allant\ de\ 1\ à\ m) \\ \Sigma_n, E_n & sinon \\ & \quad avec\ \mathsf{Unify}(\Sigma, (E; \Gamma \vdash M = (?_i\ b_1 ... b_{m-n})), \Gamma \vdash a_1 = b_{m-n+1}) \equiv \Sigma_1, E_1 \\ & \quad \mathsf{Unify}(\Sigma_{j-1}, E_{j-1}, \Gamma \vdash a_j = b_{m-n+j}) \equiv \Sigma_i, E_j \\ & \quad\ (j\ allant\ de\ 2\ à\ n) \end{cases}$$

$$S_2 = \begin{cases} \mathsf{Unify}'(\Sigma, E, \Gamma \vdash \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((\Gamma^{val}_M\ a_1 ... a_n)) = (?_i\ b_1 ... b_m)) & si\ M \in Def(\Gamma) \\ \bot & sinon \end{cases}$$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (?_i\ b_1 ... b_m) = (M\ a_1 ... a_n)) = \mathsf{Unify}'(\Sigma, E, \Gamma \vdash (M\ a_1 ... a_n) = (?_i\ b_1 ... b_m))$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (M\ a_1 ... a_n) = (c\ b_1 ... b_m)) = \begin{cases} S_1 & si\ S_1 \not\equiv \bot \\ S_2 & sinon \end{cases}$ *avec*

$$S_1 = \begin{cases} \Sigma_n, E_n & si\ c \equiv M\ et\ n \equiv m \\ & \quad avec\ \mathsf{Unify}(\Sigma, E, \Gamma \vdash a_1 = b_1) \equiv \Sigma_1, E_1 \\ & \quad \mathsf{Unify}(\Sigma_{i-1}, E_{i-1}, \Gamma \vdash a_i = b_i) \equiv \Sigma_i, E_i \\ & \quad (i\ allant\ de\ 2\ à\ n) \\ \bot & sinon \end{cases}$$

$$S_2 = \mathsf{Unify}'(\Sigma, E, \Gamma \vdash (M\ a_1 ... a_n) = \mathsf{Whnf}^{-\delta}_{\Gamma,\Sigma}((\Gamma^{val}_c\ b_1 ... b_m)))$$

- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash a :: A = M) = \mathsf{Unify}'(\Sigma, E, \Gamma \vdash a = M)$
- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash M = a :: A) = \mathsf{Unify}'(\Sigma, E, \Gamma \vdash M = a)$
- $\mathsf{Unify}'(\Sigma, E, \Gamma \vdash M = N) = \bot$ *pour tous les autres cas*

Il est possible d'avoir un seul paramètre pour Unify (autre que l'équation à résoudre) en combinant la signature, le contexte et les équations en attente dans une seule séquence. C'est la solution adoptée dans [56].

Nous ne prouvons aucune propriété de l'algorithme donné; nous nous contentons de spécifier la propriété de correction souhaitée. Pour cela, nous commençons par définir précisément la notion de solution d'une équation.

Définition 4.12 (Solution d'une équation)

- *Une signature* Σ *est une* solution *d'une équation* $\Gamma \vdash M = N$ *si et seulement si* $\Sigma(M) \cong \Sigma(N)$. *De plus* $\Sigma(M)$ *et* $\Sigma(N)$ *doivent être bien typés dans* Γ.
- *De la même manière,* Σ *est une solution d'un ensemble d'équations* E *si et seulement si* Σ *est une solution de chaque équation de* E.
- *Nous notons par* $Sol(E)$ *l'ensemble des solutions d'un ensemble d'équations* E.

Nous définissons aussi la notion d'extension d'une signature. Il s'agit d'une signature obtenue par déclaration et définition de nouvelles métavariables, et instanciation de métavariables déjà déclarées.

Définition 4.13 (Extension d'une signature) *Soient* Σ_1 *et* Σ_2 *deux signatures valides. On a* $\Sigma_2 \geq \Sigma_1$ *si et seulement si:*

- $\forall (\Gamma_i \vdash ?_i : A) \in \Sigma_1.\ ((\Gamma_i \vdash ?_i : A) \in \Sigma_2)\ \vee\ ((\Gamma_i \vdash ?_i : A := a) \in \Sigma_2)$, *et*
- $\forall (\Gamma_i \vdash ?_i : A := a) \in \Sigma_1.\ (\Gamma_i \vdash ?_i : A := a) \in \Sigma_2$.

La signature Σ_2 *est appelée* extension *de* Σ_1.

Propriété de correction L'énoncé de la propriété de correction est alors:

Soit $\mathsf{Unify}(\Sigma, E, \Gamma \vdash a = b) \equiv \Sigma', E'$. Nous devons avoir:

1. $\Sigma' \geq \Sigma$, et
2. $\forall \Sigma''$ (avec $\Sigma'' \geq \Sigma'$). $\Sigma'' \in Sol(E') \Longrightarrow \Sigma'' \in Sol(E)$.

4.4 Inférence de types

En plus du contexte, l'algorithme d'inférence de types prend en entrée une signature, un ensemble d'équations (restant à résoudre), un terme à typer (et à compléter), ainsi que son type attendu. Le symbole «$\bullet$» désigne l'absence de type attendu.

Définition 4.14 (Syntaxe abstraite implicite(2))

$$T \quad ::= \quad s \mid x \mid c \mid [x:T]T \mid (T\ T) \mid (x:T)T \mid ?_i \mid \bullet \qquad (i \in \mathbb{N}^*)$$

Nous étendons l'unification pour prendre en compte le terme $\bullet$.

Définition 4.15 (Unification (Suite 4.11))

- $\mathsf{Unify}(\Sigma, E, \Gamma \vdash M = \bullet) = \Sigma, E$
- $\mathsf{Unify}(\Sigma, E, \Gamma \vdash \bullet = M) = \Sigma, E$

La principale différence entre cet algorithme et un algorithme d'inférence pour le langage explicite est le remplacement de la conversion par l'unification. Les règles pour déterminer le type attendu d'un terme sont simples:

1. Soit à typer $(f\ a)$. Si f a pour type $(x:A)B$, alors A est le type attendu de a.
2. Dans le terme $a :: A$, A est le type attendu de a.
3. Si le terme $[x:A]b$ a pour type attendu $(x:A')B$, alors B est le type attendu de b.

Il existe un cas particulier où une métavariable est directement instanciée, lors de sa déclaration. En effet, si le terme $[x:?]b$ a pour type attendu $(x:A)B$, alors la valeur de $?_i$ (nouvelle métavariable associée à «?») est A.

L'algorithme d'inférence de types utilise deux fonctions auxiliaires Dom et Rng, calculant respectivement le domaine et le codomaine d'un type produit.

Définition 4.16 *Les fonctions Dom et Rng sont définies par:*

$$Dom_{\Gamma,\Sigma}(M) = \begin{cases} A & si\ \mathsf{Whnf}_{\Gamma,\Sigma}(M) \equiv (x:A)B \\ \bullet & sinon \end{cases}$$

$$Rng_{\Gamma,\Sigma}(M) = \begin{cases} B & si\ \mathsf{Whnf}_{\Gamma,\Sigma}(M) \equiv (x:A)B \\ \bullet & sinon \end{cases}$$

Pour typer les applications, il est plus judicieux de considérer des applications «aplaties» de la forme $(a\ b_1 ... b_n)$. En effet, avant toute unification, on commence par typer tous les arguments b_i; ce qui nous permet de glaner un maximum d'informations sur les métavariables.

Définition 4.17 (Inférence de types)

$$
\begin{array}{lcl}
\mathsf{Infer}(\Sigma, E, \Gamma, s_1 : M) & = & \Sigma, E, s_1 : s_2 \quad \textit{avec } \langle s_1, s_2\rangle \in \mathcal{A} \\
\mathsf{Infer}(\Sigma, E, \Gamma, x : M) & = & \Sigma, E, x : \Gamma_x \\
\mathsf{Infer}(\Sigma, E, \Gamma, c : M) & = & \Sigma, E, c : \Gamma_c \\
\mathsf{Infer}(\Sigma, E, \Gamma, ? : M) & = & (\Sigma; \Gamma \vdash ?_i : M), E, ?_i : M \quad \textit{avec } i \textit{ nouveau et } M \not\equiv \bullet \\
\mathsf{Infer}(\Sigma, E, \Gamma, ?_i : M) & = & \Sigma, E, ?_i : \Sigma_{?_i} \\
\mathsf{Infer}(\Sigma, E, \Gamma, (x : A)B : M) & = & \Sigma_2, E_2, (x : A_1)B_1 : R_{\langle s_1, s_2\rangle} \\
 & & \textit{avec } \mathsf{Infer}(\Sigma, E, \Gamma, A : \bullet) \equiv \Sigma_1, E_1, A_1 : N, \\
 & & \mathsf{Whnf}_{\Gamma,\Sigma_1}(N) \equiv s_1, \\
 & & x \notin Dom(\Gamma), \\
 & & \mathsf{Infer}(\Sigma_1, E_1, (\Gamma; x : A_1), B, \bullet) \equiv \Sigma_2, E_2, B_1 : P \ \textit{et} \\
 & & \mathsf{Whnf}_{\Gamma,\Sigma_2}(P) \equiv s_2 \\
\mathsf{Infer}(\Sigma, E, \Gamma, [x :?]b : M) & = & \Sigma_2, E_2, [x : A_1]b_1 : (x : A_1)B_1 \\
 & & \textit{avec } \mathsf{Whnf}_{\Gamma,\Sigma}(M) \equiv (x : A)B, \\
 & & \mathsf{Infer}(\Sigma, E, \Gamma, A : \bullet) \equiv \Sigma_1, E_1, A_1 : N, \\
 & & \mathsf{Whnf}_{\Gamma,\Sigma_1}(N) \equiv s_1, \\
 & & x \notin Dom(\Gamma), \\
 & & \mathsf{Infer}((\Sigma_1; \Gamma \vdash ?_i : s_1 := A_1), E_1, (\Gamma; x : A_1), b : Rng_{\Gamma,\Sigma_1}(M)) \\
 & & \quad \equiv \Sigma_2, E_2, b_1 : B_1, \\
 & & i \textit{ nouveau et} \\
 & & (\mathsf{Whnf}_{\Gamma,\Sigma_2}(B_1) \in \mathcal{A}_o \vee \mathsf{Whnf}_{\Gamma,\Sigma_2}(B_1) \notin \mathcal{S}) \wedge \\
 & & \quad \exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3\rangle \in \mathcal{R} \\
\mathsf{Infer}(\Sigma, E, \Gamma, [x : A]b : M) & = & \Sigma_2, E_2, [x : A]b_1 : (x : A_1)B_1 \\
 & & \textit{avec } \mathsf{Infer}(\Sigma, E, \Gamma, A : \bullet) \equiv \Sigma_1, E_1, A_1 : N, \\
 & & \mathsf{Whnf}_{\Gamma,\Sigma_1}(N) \equiv s_1, \\
 & & x \notin Dom(\Gamma), \\
 & & \mathsf{Infer}(\Sigma_1, E_1, \Gamma, b : Rng_{\Gamma,\Sigma_1}(M)) \equiv \Sigma_2, E_2, b_1 : B_1 \ \textit{et} \\
 & & (\mathsf{Whnf}_{\Gamma,\Sigma_2}(B_1) \in \mathcal{A}_o \vee \mathsf{Whnf}_{\Gamma,\Sigma_2}(B_1) \notin \mathcal{S}) \wedge \\
 & & \quad \exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3\rangle \in \mathcal{R} \\
\mathsf{Infer}(\Sigma, E, \Gamma, (a\ b) : M) & = & \mathsf{Unify}(\Sigma_2, E_2, \Gamma \vdash M = A[x \leftarrow b']), (a'\ b') : A[x \leftarrow b'] \\
 & & \textit{avec } \mathsf{Infer}(\Sigma, E, \Gamma, a, \bullet) \equiv \Sigma_1, E_1, a' : N, \\
 & & \mathsf{Infer}(\Sigma_1, E_1, \Gamma, b : Dom_{\Gamma,\Sigma_1}(N)) \equiv \Sigma_2, E_2, b' : B' \\
 & & Rng_{\Gamma,\Sigma_2}(N) \equiv (x : B)A \\
\mathsf{Infer}(\Sigma, E, \Gamma, a :: A : M) & = & \mathsf{Unify}(\Sigma_2, E_2, \Gamma \vdash A_1 = A_2), a_1 :: A_2 : A_1 \\
 & & \textit{avec } \mathsf{Infer}(\Sigma, E, \Gamma, A : \bullet) \equiv \Sigma_1, E_1, A_1 : N, \\
 & & \mathsf{Whnf}_{\Gamma,\Sigma_1}(N) \equiv s \ \textit{et} \\
 & & \mathsf{Infer}(\Sigma_1, E_1, \Gamma, a : A_1) \equiv \Sigma_2, E_2, a_1 : A_2
\end{array}
$$

Propriété de correction La propriété de correction pour Infer est:

Soit $\mathsf{Infer}(\emptyset, \emptyset, \Gamma, M : \bullet) \equiv \Sigma, E, M' : A$. On a:

1. $MV(\Sigma(M')) \cup MV(\Sigma(A)) = \emptyset$
2. $\Gamma \vdash \Sigma(M') : \Sigma(A)$
3. M' est obtenu à partir de M en y remplaçant chaque inconnue « ? » par un terme.

4.5 Heuristique pour l'unification en présence d'enregistrements

Nous remédions dans cette section à une faiblesse de l'algorithme de synthèse de sous-termes implicites. Soit un exemple simple d'enregistrement:

$Structure\ R\ :\ *\ :=\ \{Car : *;\ Ar : Car \rightarrow nat\}$

Nous rappelons que les projections ont pour type:

$$Car : R \rightarrow * \qquad \text{et} \qquad Ar : (r : R)(Car\ r) \rightarrow nat$$

Supposons la fonction Ar_1 (de type $bool \rightarrow nat$) définie par ailleurs, et considérons la constante ci-dessous:

$$r1\ :=\ (Build\_R\ bool\ Ar_1)$$

Notre algorithme d'inférence de type est incapable de typer des termes de la forme $(f\ ?\ b)$ dans le contexte:

$$...; f : (r : R)(x : (Car\ r))(list\ A);\ b : bool; ...$$

En effet, l'algorithme est incapable d'unifier $(Car\ ?_1)$ et $bool$. Notre technique consiste alors, dans ce cas, à chercher dans le contexte une constante de type enregistrement R et ayant pour valeur $bool$ au champ Car; cette constante est r_1. De manière générale, il s'agit de simplifier les équations de la forme $(Car\ t) = bool$ en $t = r_1$.

Nous allons traiter maintenant le cas le plus général; Soit R un enregistrement quelconque:

$$Structure\ R\ [p_1 : P_1; ...; p_m : P_m] : s\ :=\ R\_Cons\ \{L_1 : T_1; ...; L_n : T_n\}$$

Soit c une constante définie par:

$$c := [x_1 : B_1]...[x_k : B_k](R\_Cons\ a_1...a_m\ t_1...t_n)$$

Nous avons (voir 3.32), en posant $a^* \equiv a[x_k \leftarrow b_k]...[x_1 \leftarrow b_1]$:

$$(L_i\ a_1^*...a_m^*\ (c\ b_1...b_k)) =_{\beta\delta\iota} t_i^*$$

En particulier, pour tout t_i de la forme $(c_i\ u_{i,1}...u_{i,n_i})$ (c_i est une constante et $n_i \geq 0$), nous obtenons:

$$(L_i\ a_1^*...a_m^*\ (c\ b_1...b_k)) =_{\beta\delta\iota} (c_i\ u_{i,1}^*...u_{i,n_i}^*)$$

Les équations que nous pouvons maintenant espérer résoudre sont maintenant de la forme:

$$(L_i\ a_1'...a_m'\ t\ l_1...l_r) = (c_i\ u_{i,1}'...u_{i,n_i}'\ l_1'...l_r')$$

Il nous faut accéder dans le contexte, à la constante appropriée c à partir des noms L_i et c_i. C'est ce que réalise $\mathsf{Proj}_\Gamma(L_i, c_i)$ défini ci-dessous.

Définition 4.18 (La fonction Proj) *Soit L_i le champ d'ordre i d'un enregistrement R, et soit c_i un nom défini dans Γ. $\mathsf{Proj}_\Gamma(L_i, c_i)$ est défini par:*

$$\begin{aligned}
\mathsf{Proj}_{[]}(L_i, c_i) &= \bot \\
\mathsf{Proj}_{c:A:=[x_1:B_1]...[x_k:B_k](R_Cons\ a_1...a_m\ t_1...t_n);\Gamma}(L_i, c_i) &= \begin{cases} c & si\ t_i \equiv (c_i\ ...) \\ \mathsf{Proj}_\Gamma(L_i, c_i) & sinon \end{cases} \\
\mathsf{Proj}_{X;\Gamma}(L_i, c_i) &= \mathsf{Proj}_\Gamma(L_i, c_i)\ \textit{pour tous les autres cas}
\end{aligned}$$

Par abus de notation, nous écrirons $\mathsf{Proj}(L_i, c_i)$; le contexte sous-entendu est alors le contexte courant.

Remarquez que c_i peut être aussi une étiquette d'un champ d'enregistrement. Nous en tenons compte dans l'algorithme d'unification, en considérant d'abord $\mathsf{Proj}_\Gamma(L_i, c_i)$ puis $\mathsf{Proj}_\Gamma(c_i, L_i)$ si $\mathsf{Proj}_\Gamma(L_i, c_i)$ échoue.

Définition 4.19 (Modification de l'algorithme d'unification)

$$\mathsf{Unify}'(\Sigma, E, \Gamma \vdash (L_i\ a'_1...a'_m\ t\ l_1...l_r) = (c_i\ u'_{i,1}...u'_{i,n_i}\ l'_1...l'_r)) = \Sigma_{m+n_i+r+1}, E_{m+n_i+r+1}$$

avec $\mathsf{Unify}(\Sigma^*, E, \Gamma \vdash t = (c\ ?_1...?_k)) \equiv \Sigma_1, E_1$

$$\begin{cases} \mathsf{Unify}(\Sigma_1, E_1, \Gamma \vdash a'_1 = a^*_1) \equiv \Sigma_2, E_2 \\ \vdots \\ \mathsf{Unify}(\Sigma_m, E_m, \Gamma \vdash a'_m = a^*_m) \equiv \Sigma_{m+1}, E_{m+1} \end{cases}$$

$$\begin{cases} \mathsf{Unify}(\Sigma_{m+1}, E_{m+1}, \Gamma \vdash u'_{i,1} = u^*_{i,1}) \equiv \Sigma_{m+2}, E_{m+2} \\ \vdots \\ \mathsf{Unify}(\Sigma_{m+n_i}, E_{m+n_i}, \Gamma \vdash u'_{i,n_i} = u^*_{i,n_i}) \equiv \Sigma_{m+n_i+1}, E_{m+n_i+1} \end{cases}$$

$$\begin{cases} \mathsf{Unify}(\Sigma_{m+n_i+1}, E_{m+n_i+1}, \Gamma \vdash l_1 = l'_1) \equiv \Sigma_{m+n_i+2}, E_{m+n_i+2} \\ \vdots \\ \mathsf{Unify}(\Sigma_{m+n_i+r}, E_{m+n_i+r}, \Gamma \vdash l_r = l'_r) \equiv \Sigma_{m+n_i+r+1}, E_{m+n_i+r+1} \end{cases}$$

en posant:
$a^* \equiv a[x_k \leftarrow ?_k]...[x_1 \leftarrow ?_1]$ *et*
$\Sigma^* \equiv \Sigma;\ \Gamma \vdash ?_1 : A_1; ...;\ \Gamma \vdash ?_j : A_j[x_{j-1} \leftarrow ?_{j-1}]...[x_1 \leftarrow ?_1]; ...;\ \Gamma \vdash ?_k : A_k[x_{k-1} \leftarrow ?_{j-1}]...[x_1 \leftarrow ?_1]$

avec $\Gamma_c^{val} \equiv [x_1 : B_1]...[x_k : B_k](R\_Cons\ a_1...a_m\ t_1...(c_i\ u_{i,1}...u_{i,n_i})...t_n)$

obtenu par: $c = \begin{cases} \mathsf{Proj}_\Gamma(L_i, c_i) & si\ \mathsf{Proj}_\Gamma(L_i, c_i) \not\equiv \bot \\ \mathsf{Proj}_\Gamma(c_i, L_i) & sinon \end{cases}$

Remarquez que cette règle peut s'appliquer plusieurs fois de manière successive sur les équations engendrées. Nous donnons un exemple complet décrivant un tel cas d'utilisation dans la section 8.11.2.

4.6 Syntaxe concrète des applications et insertion de « ? »

Les sous-termes implicites les plus fréquents sont les arguments implicites de fonctions polymorphiques (comme notre exemple *comp*). Il est alors important d'avoir une syntaxe particulière pour les applications de fonctions nous évitant les disgracieux symboles « ? ».

R. Pollack[132] a introduit dans LEGO une notation spéciale indiquant les arguments implicites des fonctions. Ainsi $[x|A]b$ est une fonction avec un argument implicite, son type est alors de la forme $(x|A)B$. Les arguments implicites n'ont pas besoin d'être fournis lors de l'application. Par exemple, soit l'identité définie par $Id : (A|*)A \to A := [A|*][x : A]x$; l'application de Id à 3 est notée tout simplement par $(Id\ 3)$. Ce terme est traduit alors en $(Id\ ?\ 3)$ avec « ? » une inconnue devant être instanciée (en l'occurrence en *nat*) par l'algorithme d'inférence vu dans la section précédente. Notre exemple fétiche de la composition s'écrirait alors:

$$[A|*][B|*][f : A \to B][C|*][g : B \to C][x : A](g\ (f\ x))$$

Et son type est $(A|*)(B|*)(A \to B) \to (B \to C) \to (C|*)A \to C$. La notation $(f|a)$ permet d'expliciter un argument normalement implicite; exemple $(Id|nat\ 3)$.

Bien que similaire dans le principe, notre solution pour cacher les inconnues « ? » est différente de celle de LEGO. Dans notre mécanisme, le calcul des positions implicites est automatique; l'utilisateur n'a plus à les indiquer avec la notation $[x|A]b$ ou $(x|A)B$. La deuxième différence est une restriction: seules les constantes, types inductifs, enregistrements et variables globales (voir section 3.1.3, page 46) ont des positions implicites associées. Notre calcul est très simple: un argument est implicite s'il apparaît comme composante dans le type d'un argument suivant. Dans [44], G. Huet propose un calcul plus fin où seules les occurrences rigides (notion provenant de l'unification d'ordre supérieur) sont prises en compte.

Notation 4.1 *Par abus de notation, nous noterons quelquefois par* $(f\ [a_1; ...; a_n])$ *le terme* $(f\ a_1 ... a_n)$ *où les* a_i *sont des termes.*

Définition 4.20 (Calcul des positions implicites) *Soit une constante c de type A. La séquence de ses positions implicites* $\mathsf{Implicits}_\Gamma(c)$ *est calculée à partir de la forme normale faible de tête itérée* $\mathsf{Whnf}^*(A)$. *Soit* $(x_1 : A_1)...(x_n : A_n)B$ *la valeur de* $\mathsf{Whnf}^*(A)$.

$$\mathsf{Implicits}_\Gamma(c) = [i \mid x_i \in VL(A_j)\ avec\ j\ allant\ de\ i+1\ à\ n]$$

Les positions implicites sont rangées dans l'ordre croissant.

Par exemple, les positions implicites de la constante *comp* sont $[1; 2; 4]$.

Dans notre système, pour expliciter un argument implicite d'une application, il suffit de le précéder de « i! » où i est sa position. Ainsi pour donner explicitement le quatrième argument de $(comp\ longueur\ premier)$, il suffit d'écrire $(comp\ longueur\ 4!bool\ premier)$. Cette notation remédie à un inconvénient de LEGO où pour expliciter un argument implicite, on doit aussi expliciter tous les arguments implicites qui le précédent; notre exemple s'écrit $(comp|?\ |?\ longueur\ |bool\ premier)$ dans LEGO. Quant à la notation $(!c\ a_1 ... a_n)$, elle permet d'ignorer tous les arguments implicites de c. Tous les arguments de c doivent être fournis; ceci évite les trop longues listes d'arguments

implicites explicités. Nous donnons ci-dessous la syntaxe concrète de notre langage incluant les arguments implicites.

Nous disposons aussi d'une nouvelle syntaxe pour les abstractions implicites, de la forme $[x]a$.

Définition 4.21 (Syntaxe implicite(2))

$$\begin{array}{lcl} T & ::= & s \mid x \mid c \mid [x:T]T \mid (T\ T) \mid (x:T)T \mid ? \mid [x]T \mid (!c\ T...T) \mid (c\ A...A) \\ A & ::= & T \mid i!T \end{array}$$

L'opération d'insertion des inconnues « ? » dans le système de LEGO s'effectue en même temps que le typage. Nous donnons, pour illustrer notre propos, une forme simplifiée (sans nous occuper des conditions de typage et de la résolutions des inconnues « ? ») de l'explicitation des applications dans LEGO. Nous notons par $\overline{M}$ la forme explicite du terme M. Pour expliciter une application $(f\ a)$, nous devons connaître le type de $\overline{f}$. Ainsi:

$$\overline{(f\ a)} = \begin{cases} (\overline{f}\ ?\ \overline{a}) & \text{si } \overline{f} : (x|A)B \\ (\overline{f}\ \overline{a}) & \text{sinon} \end{cases}$$

$$\overline{(f|a)} = \begin{cases} (\overline{f}\ \overline{a}) & \text{si } \overline{f} : (x|A)B \\ \bot & \text{sinon} \end{cases}$$

La fonction d'insertion des « ? » dans Coq est alors quelque peu compliquée mais est complètement séparée du typage; elle est purement syntaxique.

Définition 4.22 (Insertion des inconnues « ? »)

$$\begin{array}{rcl} \overline{s} & = & s \\ \overline{x} & = & x \\ \overline{c} & = & c \\ \overline{[x:A]b} & = & [x:\overline{A}]\overline{b} \\ \overline{(x:A)B} & = & (x:\overline{A})\overline{B} \\ \overline{[x]b} & = & [x:?]\overline{b} \\ \overline{(c\ a_1...a_n)} & = & \begin{cases} \begin{cases} (c'\ \overline{a_1}...\overline{a_n}) & si\ \forall i.a_i \not\equiv !a'_i \\ \bot & si\ \exists i.a_i \equiv !a'_i \end{cases} & si\ c \equiv !c' \\ (c\ [a_1...a_n] \parallel_1 \mathsf{Implicits}_\Gamma(c)) & sinon \end{cases} \\ \overline{(a\ b)} & = & (\overline{a}\ \overline{b}) \end{array}$$

Le paramètre k de $\parallel_k$ correspond au rang de l'argument attendu pour l'application.

$$[a_i...a_n] \parallel_k [p_j...p_m] = \begin{cases} [] & si\ [a_i...a_n] \equiv [] \\ \begin{cases} [\overline{a'_i}] \cdot [a_{i+1}...a_n] \parallel_{k+1} [p_{j+1}...p_m] & si\ a_i \equiv k!a'_i \\ [?] \cdot [a_i...a_n] \parallel_{k+1} [p_{j+1}...p_m] & sinon \end{cases} & si\ k = j \\ \begin{cases} \bot & si\ a_i \equiv l!a'_i \\ [\overline{a_i}] \cdot [a_{i+1}...a_n] \parallel_{k+1} [p_j...p_m] & sinon \end{cases} & si\ k < j \\ \bot & sinon \end{cases}$$

L'apparente complexité de la fonction d'insertion ci-dessus provient du fait qu'un terme implicite peut être erroné pour différentes raisons:

- explicitation avec la notation $i!a$ d'un argument d'une application complètement explicite $(!c\ a_1 ... a_n)$.
- explicitation d'une position non implicite; argument de la forme $i!a$ avec $i \notin \mathsf{Implicits}_\Gamma(c)$.
- ordre des arguments implicites non respecté.

À cause de la restriction du calcul des positions implicites aux constantes, notre système s'avère moins puissant que celui de LEGO. Considérons le terme $(Id\ Id\ 3)$; notre fonction d'insertion le transforme en $(Id\ ?\ Id\ 3)$, qui est mal typé. Il nous faut écrire $(Id\ Id\ ?\ 3)$. En effet notre système ne prend pas en compte le changement d'arité (nombre d'arguments) des constantes polymorphes. C'est une faiblesse théorique importante mais peu pesante en pratique (du moins dans notre application de la formalisation de la théorie des catégories – voir la troisième partie de cette thèse). En LEGO, $(Id\ Id\ 3)$ est tout à fait correct puisque $(Id\ Id) : (A|*)A \to A$ a un argument implicite. D'où $(Id\ Id\ 3) : nat$, et sa forme explicite $(Id\ ((A|*)A \to A)\ Id\ nat\ 3)$ comporte deux termes implicites.

4.7 Surcharge de noms de fonctions

Le mécanisme que nous venons de développer (section 4.5) constitue aussi un mécanisme simple et efficace de *surcharge* de noms de fonctions. Nous entendons par surcharge la possibilité d'utiliser un même nom pour plusieurs fonctions différentes. L'exemple le plus courant est celui du *plus* (ou +): dans beaucoup de langages, il représente à la fois l'addition des entiers naturels, des réels, mais aussi la concaténation de chaînes de caractères et de listes. À la compilation (où à l'exécution pour certains langages, notamment orientés objets), la fonction appropriée est sélectionnée suivant le type de ses arguments.

Donnons un exemple. Soit à surcharger des noms de fonctions suivant le type de leur premier argument. Pour cela, nous devons définir le type des fonctions d'au moins un argument (toutes les fonctions).

$$Structure\ FunArg_1\ :\ *\ :=\ \{Car_1 : *;\ Car_2 : *;\ f : Car_1 \to Car_2\}$$

Supposons maintenant qu'on veuille surcharger les fonctions f_1, f_2 et f_3 présentes dans le contexte.

$$...A_1 : Type;\ A_2 : Type;\ f_1 : nat \to A_1;\ f_2 : bool \to A_2;$$
$$f_3 : (list\ nat) \to bool \to nat;\ f_4 : (list\ bool) \to bool \to nat;\ g : A_1 \to A_2 \to bool...$$

Il faut pour cela les encapsuler en des objets de type $FunArg_1$.

$$F_1\ :=\ (Build\_FunArg_1\ f_1)$$
$$F_2\ :=\ (Build\_FunArg_1\ f_2)$$
$$F_3\ :=\ (Build\_FunArg_1\ f_3)$$

Nous pouvons alors écrire f à la place des trois fonctions f_1, f_2 et f_3. Exemples:

- $(g\ (f\ 0)\ (f\ True))$ est équivalent à $(g\ (f_1\ 0)\ (f_2\ True))$, et
- $(g\ (f\ (f\ (nil\ nat)\ False))\ (f\ True))$ est équivalent à $(g\ (f_1\ (f_3\ (nil\ nat)\ False))\ (f_2\ True))$.

Explicitons le terme $(f\ 0)$; il se réécrit d'abord en $(f\ ?\ 0)$, puis le processus d'inférence nous amène à résoudre le problème d'unification $(Car_1\ ?_1) = nat$. Celui-ci, suivant la méthode de résolution décrite ci-dessus (définition 4.19), se résout par $?_1 := F_1$. Avec la même technique, les inconnues de $(g\ (f\ (f\ (nil\ nat)\ False))\ (f\ True))$ (soit $(g\ (f\ ?_1\ (f\ ?_2\ (nil\ nat)\ False))\ (f\ ?_3\ True))$ après explicitation) sont résolues par $?_1 := F_1$, $?_2 := F_2$ et $?_3 := F_3$.

Dans l'exemple précédent, il n'est pas possible de surcharger f_3 et une autre fonction f_4 (de type $(list\ bool) \rightarrow ...$) car seul le constructeur du champ Car_1 (ici $list$) importe, et non son type exact.

On peut de manière analogue définir d'autres types de fonctions (comme $FunArg_1$) de façon à autoriser la discrimination sur un numéro d'argument bien déterminé. Il est toutefois impossible d'avoir de stratégie plus sophistiquée de discrimination, comme la recherche du premier argument a_i de l'application $(f\ a_1 ... a_n)$ tel qu'il n'existe qu'une seule fonction surchargée $h : T_1 \rightarrow ... \rightarrow T_m$ avec les types T_j $(j \leq i)$ compatibles avec ceux de a_1 jusqu'à a_i.

Construisons la définition $FunArg_2$ nous permettant de surcharger des fonctions ayant au moins deux arguments. La discrimination se fera alors sur le type du second argument.

$Structure\ FunArg_2\ [C_1 : *]\ :\ *\ := \{C_2 : *;\ C_3 : *;\ h : C_1 \rightarrow C_2 \rightarrow C_3\}$

Soit une partie d'un contexte valide:

$$...h_1 : nat \rightarrow bool \rightarrow A_1;\ h_2 : nat \rightarrow nat \rightarrow A_2; ...$$

Pour surcharger h_1 et h_2, nous devons d'abord les transformer en des objets de type $FunArg_2$.

$H_1\ :=\ (Build\_FunArg_2\ h_1)$
$H_2\ :=\ (Build\_FunArg_2\ h_2)$

Nous pouvons alors écrire $(g\ (h\ 0\ True)\ (h\ 0\ 0))$ à la place de $(g\ (h_1\ 0\ True)\ (h_2\ 0\ 0))$.

Nous donnons maintenant la structure permettant la discrimination sur l'argument d'ordre n.

$Structure\ FunArg_n\ [C_1 : *; ....; C_{n-1} : *]\ :\ *\ := \{C_n : *;\ C_{n+1} : *;\ h : C_1 \rightarrow ... \rightarrow C_n \rightarrow C_{n+1}\}$

D'autres exemples de structures plus intéressantes sont possibles, comprenant des types dépendants et des propriétés sur les fonctions; voir les exemples de l'égalité et de la composition dans 8.11.2.

Chapitre 5

Coercions Implicites

5.1 Introduction

Une différence fondamentale entre la théorie des ensemble et la théorie des types est que cette dernière ne possède pas une notion de sous-type correspondant à la notion de sous-ensemble. Alors qu'un élément peut appartenir à plusieurs ensembles différents, en théorie des types, un terme n'a qu'un seul type unique (à la convertibilité près). Cet état de fait a poussé les chercheurs à considérer diverses notions de sous-typage. Le principe de base du sous-typage est que si A est un sous-type de B, noté $A \leq B$, alors les termes de type A peuvent être considérés comme étant de type B aussi.

$$\frac{\Gamma \vdash a : A \qquad A \leq B}{\Gamma \vdash a : B}$$

Ceci induit une nouvelle forme de polymorphisme : une fonction peut être appliquée à tout terme dont le type est «inférieur» à son type attendu.

$$\frac{\Gamma \vdash f : B \to C \qquad \Gamma \vdash a : A \qquad A \leq B}{\Gamma \vdash (f\ a) : C}$$

L'approche basée sur les *coercions*[1] permet de garder une théorie fortement typée (chaque terme possède au plus un type). On introduit une constante unique $c_{A,B}$, appelée coercion, pour chaque paire de types A et B avec $A \leq B$. Dans certaines théories, ces coercions sont définissables et correspondent alors tout simplement à des fonctions. Les deux règles ci-dessus deviennent alors :

$$\frac{\Gamma \vdash a : A \qquad c_{A,B} : A \leq B}{\Gamma \vdash (c_{A,B}\ a) : B}$$

$$\frac{\Gamma \vdash f : B \to C \qquad \Gamma \vdash a : A \qquad c_{A,B} : A \leq B}{\Gamma \vdash (f\ (c_{A,B}\ a)) : C}$$

L'utilisation des coercions permet une forme de *surcharge*; par exemple le nombre 3 peut être considéré, suivant les circonstances, comme un entier naturel, un entier relatif, un nombre rationnel, un nombre réel ou un nombre complexe. Il y a en fait cinq objets différents avec le nom 3. Ils

[1] «coercition» semble être le terme consacré dans la langue française; nous lui préférons toutefois l'anglicisme «coercion».

sont tous référencés par 3 mais un objet particulier est utilisé à chaque référence suivant le contexte d'utilisation du nom. Contrairement aux mécanismes de surcharge habituels, les coercions permettent d'expliciter les connexions entre tous ces objets.

Une forme de sous-typage fort répandue en pratique est l'*héritage* dans les langages de programmation à objets. Ce sont des langages développés avec comme préoccupation principale la réutilisation de code et la factorisation des propriétés. Depuis Simula 67, le premier langage à objets de l'histoire, plusieurs langages ont vu le jour; les plus importants sont Smalltalk, Eiffel, C++ et Java. Il y a plusieurs conceptions de l'héritage suivant le langage considéré. Dans ces langages, les programmes sont organisés en *classes* (s'apparentant à des enregistrements), qui encapsulent les données et les méthodes (ou fonctions). Une classe A peut être *spécialisée* en lui rajoutant de nouvelle composantes (ou champs) pour obtenir une nouvelle classe B; on dit que B hérite ou dérive de A. Une méthode de A peut alors être *invoquée* (appliquée) sur des objets de B. L'héritage peut être «simulé» par l'utilisation de coercions. Toutefois, certains traits importants des langages à objets ne peuvent pas être représentés par le sous-typage, comme par exemple la redéfinition de méthodes et la spécification d'accès aux champs d'une classe. Avec le mécanisme de redéfinition de méthodes, une classe peut redéfinir le comportement de certaines méthodes héritées. Quant au mécanisme de spécification d'accès, il permet de restreindre l'accès de certains champs pour les classes dérivées (classes qui héritent) dans des buts de sécurité et de confidentialité.

Le sous-typage a été intensivement étudié dans le domaine de la programmation fonctionnelle. Des études sémantiques ont été aussi menées dans des théories de type sans types dépendants comme le système F, notamment [28] utilisant une approche basée sur les coercions. Une étude plus récente du sous-typage dans le système F est [101]. Les études sur le sous-typage dans des théories de type avec types dépendants sont par contre peu nombreuses; nous citons le travail de Pfenning[126] sur les «refinement types», le travail de Aspinall et Compagnoni[12] sur la décidabilité de Edinburgh LF avec sous-typage, et enfin les travaux de Chen et Longo[39].
Bien que le sous-typage soit conceptuellement naturel, il n'est pas encore clair comment introduire un mécanisme de sous-typage dans des théories de types avec types dépendants, sans compromettre les bonnes propriétés de la théorie. Un tel mécanisme devrait représenter l'héritage entre théories mathématiques et permettre de raisonner à propos de sous-ensembles. L'absence de mécanisme de sous-typage pratique dans les systèmes de preuves est certainement l'un des principaux obstacles à leur application dans la formalisation de parties conséquentes des mathématiques.
L'utilisation des coercions implicites est tellement fréquente et naturelle en mathématiques que la plupart des mathématiciens ne se rendent même pas compte de leur existence; elles sont intégrées au langage mathématique. En effet beaucoup d'abus de notations correspondent à l'utilisation implicite de coercions, notamment entre structures mathématiques (algébriques). Ainsi par exemple, toute construction sur les ensembles peut être appliquée à (l'ensemble support d') un groupe. Cet abus de notation est généralement admis car tout groupe peut être «coercé» en un ensemble par une opération canonique qui extrait l'ensemble support sous-jacent d'un groupe.
Le cas des sous-ensembles est similaire à celui des structures car un sous-ensemble d'un ensemble A est représenté par une structure $\{x \in A \mid P(x)\}$ où P est un prédicat sur A. Un objet $\langle a, p\rangle$ de $\{x \in A \mid P(x)\}$ peut être vu comme un objet de A en considérant la première projection comme coercion.

Notre mécanisme peut être vu comme une approche à la fois conceptuellement simple et puis-

sante pour l'introduction du sous-typage dans les théories de types avec types dépendants. Nous ne sommes pas intéressés par augmenter le puissance d'expression de notre théorie, mais seulement par améliorer son confort d'utilisation. Étant donné un terme, éventuellement mal typé, nous nous intéressons au problème de savoir s'il peut être bien typé modulo l'insertion de coercions appropriées. Un tel algorithme d'inférence est donné dans [8] dans le contexte d'un λ-calcul simplement typé avec types récursifs.

Les travaux de P. Aczel[2, 3] constituent le point de départ et la principale source d'inspiration pour l'élaboration de notre système. Ces travaux étaient pionniers dans l'utilisation de l'héritage dans les systèmes de preuves dans le but d'imiter des abus de notations utilisés en mathématiques. Le mécanisme décrit par P. Aczel a été étendu par G. Barthe [16] au cas de l'*héritage multiple*. Aucune implémentation n'existe de ces deux mécanismes.
Dans une première étape et en collaboration avec A. Bailey, nous avons défini et implémenté dans LEGO un troisième système d'héritage avec héritage multiple et *classes avec paramètres* (non admis par les systèmes précédents). Dans ce chapitre, nous présentons un nouvel système d'héritage améliorant tous les systèmes précédents. La contribution majeure consiste en l'introduction de deux classes abstraites FUNCLASS et SORTCLASS permettant de capturer encore plus d'abus de notations.

Ce chapitre est organisé comme suit. Nous décrivons dans la deuxième section les travaux de P. Aczel et G. Barthe. La troisième section est consacrée à motiver les extensions que nous proposons dans notre nouveau système. La section suivante décrit ce système dans le détail. La cinquième section est consacrée à la construction du graphe d'héritage. L'algorithme d'inférence et ses propriétés sont les sujets abordés dans la sixième section.

5.2 Travaux antérieurs

Nous ne rappelons que les principaux travaux d'héritage basés sur les coercions implicites, et relatifs aux systèmes de preuves.

5.2.1 Proposition de Peter Aczel

Dans le cadre de son projet de formalisation de la théorie de Galois, P. Aczel a proposé l'utilisation de l'héritage dans les systèmes de preuves basés sur la théorie des types. Il entendait par héritage, la possibilité d'appliquer une fonction $f : A \to B$ à un terme $a : A'$ car A' peut être vu dans un certain sens comme un *sous-type* de A. Par exemple, un groupe peut être utilisé comme ensemble, monoïde ou tout autre sous-structure de groupe, suivant le contexte. Il décrivait ce mécanisme comme une forme disciplinée de *surcharge* de définitions. En effet, la fonction f ci-dessus agit de manière uniforme sur tout objet d'un sous-type de A, sans avoir à la redéfinir.

P. Aczel a proposé une notion de *classe* et *méthode* où chaque méthode est associée à seulement une classe mais est héritée par ses sous-classes. À chaque classe est associé le type de ses *instances* $\hat{C}$, et chaque méthode m sur C sera donnée sous forme d'une fonction sur $\hat{C}$. La relation d'héritage est alors définie en utilisant des fonctions appelées *coercions*: une classe C' hérite d'une classe C s'il existe une coercion de $\hat{C}'$ vers $\hat{C}$. On dit aussi que C' est une *classe-enfant* de C ou bien que C est la *classe-père* de C'.

Ainsi toute fonction définie sur $\hat{}C$ induit une fonction sur $\hat{}C'$. En particulier, chaque méthode définie sur C induit une fonction sur le type $\hat{}C'$ des instances de toute classe-enfant C' de C.

L'ensemble des classes forme une forêt de classes, c'est-à-dire un ensemble fini d'arbres finis dont les nœuds représentent des classes, et les arcs représentent des fonctions de coercion. On distingue deux sortes de définitions de classes:

- définition d'une classe correspondant à une racine d'un nouvel arbre dans la forêt.

$$C := \mathsf{rootclass}\ T$$

- définition d'une sous-classe à une sous-classe déjà existante. Ainsi:

$$C := \mathsf{childclass}\ D\ [x : T]b$$

  créé une nouvelle feuille C, immédiatement au-dessous de la classe D. La fonction $[x : T]b$ est la coercion entre $\hat{}C$ et $\hat{}D$.

Dans les deux cas, nous avons implicitement:

$$\hat{}C := T$$

Une classe C' est une sous-classe de C s'il existe un chemin de C' vers C dans un arbre de la forêt. La coercion $\hat{}C' \to \hat{}C$ est alors obtenue en composant dans l'ordre les coercions du chemin menant de C' vers C.

Une définition d'une méthode m pour une classe C a la forme:

$$m := \mathsf{method}[x : \hat{}C]e$$

Une fois cette méthode définie, il devient possible d'écrire $(m\ a)$ pourvu que $a : \hat{}C'$ où C' est une sous-classe de C. Le terme $(m\ a)$ doit être traité comme étant le terme $e[x \leftarrow (f\ a)]$ où $f : \hat{}C' \to \hat{}C$ est la coercion entre C' et C.

Ce système permet la définition incrémentale de classes et coercions. Toutefois une classe racine ne peut hériter d'aucune classe: les arbres d'héritage ne croîssent que dans un sens. Une autre faiblesse est que ce système ne supporte que l'*héritage!simple*, c'est-à-dire toute classe est classe-enfant d'au plus une classe. Cette notion est opposée à celle d'*héritage multiple* où une classe peut être classe-enfant de plusieurs classes. Cette restriction est dûe au fait que la déclaration d'une classe-enfant se fait en même temps que sa coercion. De plus, pour remédier au premier problème, il faut abolir la distinction entre classe-racine et classe-enfant. Nous obtenons ainsi deux primitives:

- Déclaration d'une classe C, en créant un arbre formé de cette seule classe.

$$C := \mathsf{class}\ T$$

- Déclaration de $[x : T]b$ comme coercion entre C et D pourvu que $[x : T]b : \hat{}C \to \hat{}D$. Un arc est alors créé dans la forêt d'héritage entre les noeuds C et D.

$$[x : T]b\ \mathsf{coercion\ from}\ C\ \mathsf{to}\ D$$

Cette nouvelle présentation pose toutefois un nouveau problème: il peut exister plusieurs coercions entre deux classes. Laquelle utiliser lors de l'application d'une méthode? Cette question est abordée dans la section suivante.

5.2.2 PTS avec héritage

Dans le système proposé par Gilles Barthe[16], la notion de classe n'existe plus; la notion centrale est celle de coercion.

Définition 5.1 (Coercion) *Une coercion est une paire de termes clos* $\langle [x : A]i, A \to B \rangle$, *que nous écrirons* $[x : A]i : A \to B$.

Les classes sont maintenant tout simplement des termes, ils correspondent aux types des instances $\hat{}C$ de [2]. Les coercions sont composées, modulo β-conversion, afin d'en construire de nouvelles. La fermeture transitive Δ^+ d'un ensemble de coercions Δ est l'ensemble de toutes les compositions de coercions – il correspond à la notion de forêt d'héritage de [2].

Définition 5.2 *La fermeture* Δ^+ *d'un ensemble* Δ *de coercions est le plus petit ensemble tel que, si pour* $i = 1, ..., n$, $c_i : A_i \to B_i \in \Delta$, *et* $A_{i+1} =_\beta B_i$ *pour* $i = 1, ..., n-1$, *alors* $[x : A_1](c_n\ (c_{n-1} ... (c_1\ x))) : A_1 \to B_n \in \Delta^+$.

Remarquons que tous les éléments de Δ^+ sont aussi des coercions.

Définition 5.3 *Soit* Δ *un ensemble de coercions et* A *et* B *deux termes. Une coercion* $[x : A']i : A' \to B' \in \Delta^+$ Δ*-lie* A *à* B *si et seulement si* $A =_\beta A'$ *et* $B =_\beta B'$.

Étant donnés un terme $M : A$ et un type B, la question qu'on va se poser est la suivante: peut-on transformer M en un terme de type B en insérant une coercion appropriée de Δ^+? Si plusieurs coercions $i_1, ... i_n$ conviennent, on voudrait qu'elles donnent toutes le même terme, c'est-à-dire $(i_k\ M) =_\beta (i_l\ M)$ pour $k, l = 1, ..., n$. C'est la condition dite de *cohérence*.

Définition 5.4 (Cohérence) *Un ensemble* Δ *est cohérent si:*

1. *pour toutes coercions* $i : A \to B$ *et* $j : A' \to B'$ *dans* Δ^+, $A =_\beta A' \wedge B =_\beta B' \Rightarrow (i\ x) =_\beta (j\ x)$ *où* x *est une variable fraîche,*
2. *pour toute coercion* $i : A \to B \in \Delta^+$, $A =_\beta B \Rightarrow (i\ x) =_\beta x$ *où* x *est une variable fraîche.*

L'opération d'insertion de coercions est capturée par une règle de calcul $\epsilon(\Delta)$.

Définition 5.5 ($\epsilon(\Delta)$-réduction)

- *La relation* $\to_{\epsilon(\Delta)}$ *est la plus petite relation telle que:*
  - $(M\ u) \to_{\epsilon(\Delta)} (M\ (i\ u))$ *où* $i : A \to B \in \Delta$,
  - $M \to_{\epsilon(\Delta)} N$, *alors* $(M\ b) \to_{\epsilon(\Delta)} (N\ b)$, $(a\ M) \to_{\epsilon(\Delta)} (a\ N)$, $[x : M]b \to_{\epsilon(\Delta)} [x : N]b$, $[x : A]M \to_{\epsilon(\Delta)} [x : A]N$, $(x : M)B \to_{\epsilon(\Delta)} (x : N)B$ *et* $(x : A)M \to_{\epsilon(\Delta)} (x : A)N$.
- *La relation de* $\epsilon(\Delta)$*-réduction* $\to^*_{\epsilon(\Delta)}$ *est la plus petite relation réflexive et transitive contenant la relation* $\to_{\epsilon(\Delta)}$.
- *La relation de* $\epsilon(\Delta)$*-équivalence* $=_{\epsilon(\Delta)}$ *est la plus petite relation d'équivalence contenant* $\to_{\epsilon(\Delta)}$.

- *La réduction $\rightarrow_{\epsilon(\Delta)}$ est étendue aux contextes par:*
  - $[] \rightarrow_{\epsilon(\Delta)} []$
  - $M \rightarrow_{\epsilon(\Delta)} N \Rightarrow \Gamma; x : M; \Gamma' \rightarrow_{\epsilon(\Delta)} \Gamma; x : N; \Gamma'$

Les règles des PTS avec héritage comportent plusieurs différences avec les PTS:

- D'abord, la règle (Meth) nous permet d'appliquer $a : (x : B)A$ à tous les termes dont le type est Δ-lié à B. Remarquez aussi que le type de l'application est $A[x \leftarrow (i\ b)]$ et non pas $A[x \leftarrow b]$, car ce dernier n'est pas toujours bien typé.
- La règle (Ccoe) permet d'introduire des coercions à condition que l'ensemble résultant de coercions soit cohérent. Le typage se fait dans un contexte vide et sans coercions.
- Nous avons deux règles de conversion; l'une pour $=_\beta$ et l'autre pour $=_{\epsilon(\Delta)}$. Les β-réductions ne sont faites que sur des termes *explicites*, c'est-à-dire typables sans coercions. En effet, la combinaison de β-réductions et de $\epsilon(\Delta)$-réductions n'a pas les bonnes propriétés, puisque $\rightarrow_{\epsilon(\Delta)}$ n'est ni fortement normalisable, ni confluente. De plus, elle ne préserve pas le typage.

Nous donnons maintenant les règles d'inférence des PTS avec héritage. La métathéorie de ce système est présentée dans [16].

Définition 5.6 (Règles des PTS avec héritage) *Les jugements $\Delta|\Gamma \vdash$ et $\Delta|\Gamma \vdash t : T$ sont définis de manière mutuellement inductive.*

$$(\text{CVide})\ \frac{}{\emptyset|[] \vdash}$$

$$(\text{CVar})\ \frac{\Delta|\Gamma \vdash A : s \qquad s \in \mathcal{S} \qquad x \notin Dom(\Gamma)}{\Delta|\Gamma; x : A \vdash}$$

$$(\text{CCoe})\ \frac{\emptyset|[] \vdash i : A \rightarrow B \qquad \Delta \cup \{i : A \rightarrow B\}\ est\ coh\acute{e}rent}{\Delta; i : A \rightarrow B|\Gamma \vdash}$$

$$(\text{Axiom})\ \frac{s_1 : s_2 \in \mathcal{A}}{\Delta|\Gamma \vdash s_1 : s_2}$$

$$(\text{Name})\ \frac{x \in Dom(\Gamma)}{\Delta|\Gamma \vdash x : \Gamma_x}$$

$$(\text{Lam})\ \frac{\Delta|\Gamma; x : A \vdash b : B \qquad \Delta|\Gamma \vdash (x : A)B : s}{\Delta|\Gamma \vdash [x : A]b : (x : A)B}$$

$$(\text{Prod})\ \frac{\Delta|\Gamma \vdash A : s_1 \qquad \Delta|\Gamma; x : A \vdash B : s_2 \qquad \langle s_1, s_2, s_3\rangle \in \mathcal{R}}{\Delta|\Gamma \vdash (x : A)B : s_3}$$

$$(\text{App})\ \frac{\Delta|\Gamma \vdash a : (x : B)A \qquad \Delta|\Gamma \vdash b : B}{\Delta|\Gamma \vdash (a\ b) : A[x \leftarrow b]}$$

$$(\text{Meth})\ \frac{\Delta|\Gamma \vdash a : (x : B)A \qquad \Delta|\Gamma \vdash b : B' \qquad B'\ est\ \Delta\text{-}lié\ à\ B\ par\ i}{\Delta|\Gamma \vdash (a\ b) : A[x \leftarrow (i\ b)]}$$

$$(\text{Conv-}\beta)\ \frac{\Delta|\Gamma \vdash M : B \qquad \emptyset|\Gamma' \vdash A : s \qquad \emptyset|\Gamma' \vdash B : s \qquad A =_\beta B \qquad \Gamma \rightarrow^*_{\epsilon(\Delta)} \Gamma'}{\Delta|\Gamma \vdash M : A}$$

$$(\text{Conv-}\epsilon(\Delta))\ \frac{\Delta|\Gamma \vdash M : B \qquad \Delta|\Gamma \vdash A : s \qquad A =_{\epsilon(\Delta)} B}{\Delta|\Gamma \vdash M : A}$$

Ce système admet l'héritage multiple, et permet la définition de coercions *opposées* $i : A \rightarrow B \in \Delta^+$ et $j : B \rightarrow A \in \Delta^+$. Les coercions opposées sont très utiles en pratique car elles permettent de manipuler de manière transparente plusieurs définitions équivalentes d'une même notion mathématique. Malheureusement, ce système est difficile à étendre au cas des coercions de type $(x : A)B$ à cause de la vérification de la condition de cohérence. En effet, deux coercions $i : (x : A)B$ et $j : (x : A)C$ peuvent avoir le même codomaine $B[x \leftarrow a] =_\beta C[x \leftarrow a]$ pour certaines valeurs a de x, et être différentes pour certaines autres valeurs.

5.3 Motivations des extensions introduites

Avant de détailler notre système, nous essayons de motiver par des cas pratiques chacune des extensions que nous introduisons.

Classes avec paramètres. La première extension intéressante à considérer est le sous-typage entre enregistrements. Ils s'apparentent dans ce cas aux classes des langages de programmation à objets; on peut construire une classe A à partir d'une classe B en l'étendant par de nouveaux champs. Lorsqu'un terme de type A est utilisé, on peut tout simplement «oublier» ses champs supplémentaires pour retrouver un terme de type B. Ainsi il existe une coercion d'«oubli» implicite entre A et B. Dans notre formalisme, on peut avoir des coercions entre enregistrements avec paramètres. Ainsi dans la déclaration ci-dessous, supposons qu'on veuille déclarer L_i comme coercion entre deux enregistrements A et B.

$$Structure\ A\ [p_1 : P_1; ...; p_m : P_m] : s\ :=\ R\_Cons\ \{...; L_i : (B\ b_1 ... b_n); ...\}$$

Le type de cette coercion est :

$$L_i : (p_1 : P_1)....(p_m : P_m)(x : (R\ p_1 ... p_m))(B\ b_1^* ... b_n^*)$$

$$\text{avec } b_j^* = b_j[L_1 \leftarrow (L_1\ p_1 ... p_m\ x)]...[L_{i-1} \leftarrow (L_{i-1}\ p_1 ... p_m\ x)]$$

Les systèmes précédents ne savent pas traiter cette coercion car il s'agit d'une coercion paramétrée, concernant deux familles de types. Ce type de coercions n'intervient pas seulement entre enregistrements; nous étendons ce schéma de coercions à tous les noms (types inductifs, constantes, variables). Nous donnons l'exemple de coercions «opposées» entre les listes et les vecteurs:

$$list\_vect : (A : Type)(l : (list\ A))(vect\ A\ (longueur\ A\ l))$$

$$vect\_list : (A : Type)(n : nat)(vect\ A\ n) \rightarrow (list\ A)$$

Coercions entre fonctions. Sans aucune modification, les systèmes précédents peuvent prendre en compte la règle de sous-typage contravariante à gauche des types fonctionnels.

$$\frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{(A_1 \to A_2) \leq (B_1 \to B_2)}$$

Cette règle peut être adaptée au cas des coercions assez aisément comme suit:

$$\frac{p : B_1 \to A_1 \qquad q : A_2 \to B_2}{[f : A_1 \to A_2][x : B_1](q\ (f\ (p\ x))) : (A_1 \to A_2) \to (B_1 \to B_2)}$$

Remarquons toutefois que dans le cas des PTS avec héritage, la vérification de la cohérence du graphe d'héritage devient problématique. Nous montrerons dans les prochaines sections comment étendre cette règle au cas des familles de types, et quelles sont ses répercussions sur la cohérence du graphe d'héritage.

Autres types de coercions. Dans les deux systèmes précédents, les coercions ne sont appliquées qu'à des arguments d'applications de la forme $(f\ a)$. Nous introduisons deux nouveaux cas d'application de coercions correspondant à des abus de notations mathématiques usuels.

- Soit G un groupe, on a l'habitude d'utiliser la notation $x \in G$, qui est pourtant formellement incorrecte car G n'est pas un ensemble; $x \in G$ est un abus de notation pour $x \in \overline{G}$ où $\overline{G}$ est l'ensemble sous-jacent de G. En théorie des types aussi, on ne peut pas écrire $x : G$ avec $G : Group$ (voir la définition de $Group$ dans 3.4) car $Group$ n'est pas une sorte (ou de manière équivalente, G n'est pas un type). Cependant, un tel mécanisme d'abus de notations est aussi possible en théorie des types; il suffit de considérer des coercions vers des sortes, et qui seraient insérées lors de la déclaration de variables. Dans le cas des groupes, cette coercion est $CarGrp$, et la déclaration correcte pour $x : G$ est $x : (CarGrp\ G)$.

- La deuxième extension concerne la notation fonctionnelle en mathématiques. De nombreuses constructions mathématiques peuvent être appliquées à des arguments sans être formellement des fonctions; exemples: morphismes de groupes (anneaux etc.), foncteurs, transformations etc. En théorie des types, la notation fonctionnelle est réservée aux termes dont le type est fonctionnel (c'est-à-dire $(x : A)B$). L'idée alors est d'étendre cette notation à tous les termes dont le type n'est pas fonctionnel mais qui peut être «coercé» vers un type fonctionnel. Donnons un exemple; nous définissons le type des morphismes de groupe. Un morphisme entre les groupes A et B est une fonction entre les types sous-jacents de A et B, préservant la structure de groupe de A.

  $$\begin{array}{l} Structure\ MorGrp\ [A, B : Group]\ :\ Type\ := \\ \{FctGrp : (CarGrp\ A) \to (CarGrp\ B); \\ \ PresGrp : (x, y : (CarGrp\ A))(FctGrp\ (OpGrp\ A\ x\ y)) = \\ \qquad\qquad\qquad\qquad (OpGrp\ B\ (FctGrp\ x)\ (FctGrp\ y))\wedge \\ \qquad (x : (CarGrp\ A))(FctGrp\ (InvGrp\ A\ x)) = (InvGrp\ B\ (FctGrp\ x))\wedge \\ \qquad (FctGrp\ (IdGrp\ A)) = (IdGrp\ B)\} \end{array}$$

  Donnons la définition en utilisant la coercion $CarGrp$ et en omettant certains arguments, que nous appelons «arguments implicites».

$$
\begin{array}{l}
Structure\ MorGrp\ [A, B : Group]\ :\ Type\ := \\
\{FctGrp : A \to B; \\
\ PresGrp : (x, y : A)(FctGrp\ (OpGrp\ x\ y)) = (OpGrp\ (FctGrp\ x)\ (FctGrp\ y)) \wedge \\
\qquad\qquad (x : A)(FctGrp\ (InvGrp\ x)) = (InvGrp\ (FctGrp\ x)) \wedge \\
\qquad\qquad (FctGrp\ (IdGrp\ A)) = (IdGrp\ B)\}
\end{array}
$$

L'énoncé $f(a + a^{-1}) = e_B$ où f un morphisme de groupes entre A et B, s'écrit alors $(f\ (OpGrp\ a\ (InvGrp\ a))) = (IdGrp\ B)$ avec $f : (MorGrp\ A\ B)$.

5.4 Langage des coercions

5.4.1 Classes

Nous adoptons l'approche de [2] en exigeant que les classes soient toutes nommées (variables globales, constantes, types inductifs, enregistrements ou projections). En pratique, toute notion importante est nommée même s'il s'agit d'un cas particulier d'une notion plus générale. Ces classes peuvent avoir des paramètres; néanmoins une classe avec paramètres est considérée comme étant une seule classe et non comme une famille de classes. En plus de ces classes *définies*, nous avons deux classes *abstraites* SORTCLASS et FUNCLASS, respectivement classe des sortes et des fonctions. Elles sont dites abstraites dans le sens où, contrairement aux classes définies, elles ne correspondent pas à un terme de la théorie.

Définition 5.7 (Classe)

- *Une classe à n paramètres est tout nom déclaré dans le contexte courant dont le type a une forme normale de la forme* $(x_1 : A_1)...(x_n : A_n)s$. *Ces classes sont dites* classes définies.
- SORTCLASS *et* FUNCLASS *sont des classes sans paramètres.*

Remarquez que par leur type, les constructeurs de types inductifs et d'enregistrements, ne peuvent pas être des classes.

La détermination de la classe d'un type est une opération purement syntaxique. En particulier, $A \cong B$ n'implique pas que $Cl(A) = Cl(B)$.

Définition 5.8 (Classe d'un terme) *La classe d'un terme bien typé* $M : A$ *est* $Cl(A)$, *définie par:*

$$
\begin{array}{rcl}
Cl(s) & = & \mathsf{SORTCLASS} \\
Cl((x : A)B) & = & \mathsf{FUNCLASS} \\
Cl((C\ a_1...a_n)) & = & C \quad si\ C\ est\ une\ classe\ définie\ avec\ n\ paramètres \\
Cl(M) & = & \bot \quad sinon
\end{array}
$$

Une autre approche est possible concernant le traitement des sortes. Il s'agit de considérer chaque sorte comme une classe, ainsi SORTCLASS n'existerait pas.

C'est l'égalité syntaxique qui est utilisée pour comparer deux classes; ainsi deux classes de noms différents sont considérées comme différentes même si leur valeur est la même. Exemple: soit une constante $c := nat \to bool$, nous avons $Cl(c) := c$ et $Cl(nat \to bool) =$ FUNCLASS.

Définition 5.9 (Égalité de classes) *Deux classes sont égales si et seulement si elles ont le même nom.*

Sur le plan de l'implémentation, ceci est un gain d'efficacité appréciable puisque comparer deux noms est beaucoup plus rapide que tester la convertibilité de deux termes.

5.4.2 Coercions

La présence des classes avec paramètres complique quelque peu la définition des coercions. Une première approche est de considérer comme coercion entre deux classes C et D, toute fonction dont le type principal est de la forme:

$$(x_1 : A_1)...(x_k : A_k)(y : M)N$$

avec $C = Cl(M)$ et $D = Cl(N)$. Pour rester cohérent avec notre choix de considérer une classe avec paramètres comme une seule classe, il est raisonnable d'exiger que toute coercion sur C s'applique à tout objet de C. Ce n'est évidemment pas le cas pour cette définition de coercion. En effet, soit $C : nat \to s$ une classe et $f : (n : nat)(C\ (S\ n)) \to N$ une coercion entre C et $Cl(N)$, cette coercion ne s'applique pas aux objets de C de type $(C\ 0)$. Il faut alors exiger que les coercions d'une classe définie C vers une classe D aient pour type principal $(x_1 : A_1)...(x_n : A_n)(y : (C\ x_1...x_n))N$ avec $D = Cl(N)$.
Considérons maintenant le cas des classes abstraites. Une coercion sur SORTCLASS de type $(x_1 : T_1)...(x_k : T_k)(y : s_1)N$, ne peut pas s'appliquer sur des objets de SORTCLASS dont le type est différent de s_1. Un problème similaire concerne FUNCLASS. Nous décidons ainsi de ne pas accepter de coercions sur les classes abstraites.

Définition 5.10 (Coercion) *Soit C une classe définie et D une classe. Une coercion est tout nom déclaré dans le contexte, vérifiant la* condition d'héritage uniforme, *c'est-à-dire ayant un type principal de la forme:*

$$(x_1 : A_1)...(x_n : A_n)(y : (C\ x_1...x_n))N$$

avec $D = Cl(N)$.
Une coercion f entre C et D est notée $f : C \longmapsto D$, et les classes C et D sont respectivement appelées classe source *et* classe cible *de f.*

Dans le cas où chaque sorte est une classe, il aurait été possible de considérer les coercions $s \longmapsto Cl(N)$ de la forme $(y : s)N$.

Définition 5.11 (Application d'une coercion) *Soient $M : (C\ a_1...a_n)$ un objet de la classe définie C et $f : C \longmapsto D$ une coercion. L'application de f à M est définie par:*

$$f@(M : (C\ a_1...a_n)) = (f\ a_1...a_n\ M)$$

Le terme obtenu est un objet de D. Nous notons son type par $f\{M : (C\ a_1...a_n)\}$. Nous noterons respectivement $f@(M : (C\ a_1...a_n))$ et $f\{M : (C\ a_1...a_n)\}$ par $f@M$ et $f\{M\}$ lorsque le type de M peut se déduire du contexte.

Sans la condition d'héritage uniforme, il aurait fallu définir $f@M$ par $(f\ ?...?\ M)$. Les inconnues « ? » devant alors être résolues par un algorithme d'arguments implicites (non complet).

5.4.3 Coercions identité

La condition d'héritage uniforme n'est pas aussi restrictive qu'on pourrait le penser. En effet, il est possible de la contourner en utilisant les *coercions identité*.

Supposons qu'on veuille déclarer une coercion entre C et D ne vérifiant pas la condition d'héritage uniforme ($D = Cl(N)$):

$$f : (x_1 : A_1)...(x_k : A_k)(y : (C\ a_1....a_n))N$$

Cette coercion ne s'applique qu'à certains objets de C, plus précisément aux objets de type $(C\ a_1....a_n)$ où $x_1, ...x_k$ sont des variables quelconques. Nous allons donc commencer par définir une classe C' à k paramètres contenant ces objets:

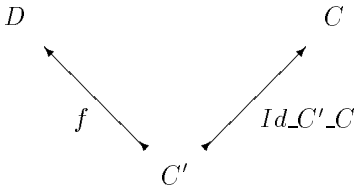
$$C' := [x_1 : A_1]...[x_k : A_k](C\ a_1....a_n)$$

Nous définissons alors une coercion entre C' et C consistant en une fonction identité:

$$Id\_C'\_C := [x_1 : A_1]...[x_k : A_k][y : (C'\ x_1....x_k)](y :: (C\ a_1....a_n))$$

La fonction f peut alors être déclarée comme coercion entre C' et $Cl(N)$, en modifiant son type en:

$$(x_1 : A_1)...(x_k : A_k)(y : (C'\ x_1....x_k))N$$



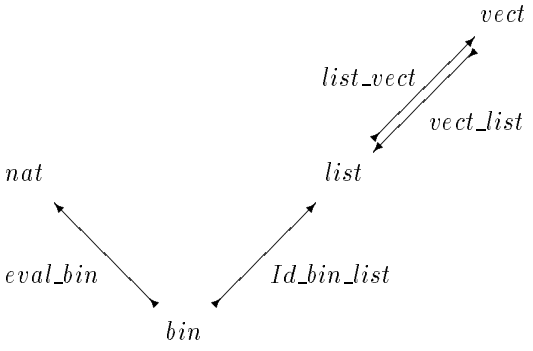
Reprenons notre exemple des listes et vecteurs. On définit une sous-classe des listes correspondant aux nombres binaires. Un nombre binaire est une suite ordonnée de booléens correspondants aux valeurs (digits) 0 et 1. Nous définissons ainsi la classe *bin* des nombres binaires.

$$bin := (list\ bool)$$

On définit une coercion entre les nombres binaires et les entiers naturels (en base décimale) correspondant à la fonction d'évaluation.

$$eval\_bin : bin \rightarrow nat$$

Seule la classe *bin* hérite de *nat*; la classe *list* n'en hérite pas. Et *bin* hérite aussi de toutes les classes dont hérite *list* grâce à la coercion identité $Id\_bin\_list : bin \rightarrow (list\ bool)$.



5.5 Graphe d'héritage

Nous formalisons maintenant la représentation adoptée lors de l'exemple précédent. Soit Δ un ensemble de coercions. Cet ensemble peut être représenté sous forme d'un *graphe d'héritage* $\langle \mathcal{N}, \mathcal{A} \rangle$:

- $\mathcal{N}$ est l'ensemble des classes définies dans Δ
- $C \xrightarrow{f} D \in \mathcal{A}$ si et seulement si $f : C \longmapsto D \in \Delta$.

Nous commençons par spécifier les conditions de validité portant sur l'ensemble des coercions.

Définition 5.12 (Ensemble valide de coercions) *Soit un ensemble Δ de coercions; chaque élément de Δ est de la forme $\langle p : A, C, D \rangle$ où p est une coercion, A son type et C et D sont respectivement ses classes de départ et d'arrivée.*
Un ensemble de coercions Δ est valide relativement à un contexte Γ si et seulement si $p \in Dom(\Gamma)$ et $\Gamma \vdash p : A$ pour tout $\langle p : A, C, D \rangle \in \Delta$. Nous noterons cette relation par $\Gamma \models \Delta$.

5.5.1 Composition de coercions

Nous représentons le graphe d'héritage par l'ensemble des chemins (au sens de la théorie des graphes) du graphe $\langle \mathcal{N}, \mathcal{A} \rangle$. Chaque chemin définit une coercion: les chemins à un seul arc correspondent à des coercions simples (appartenant à Δ), alors que les chemins plus longs sont obtenus par composition de coercions simples.

Nous montrons maintenant comment composer les coercions existantes pour obtenir de nouvelles coercions.

Définition 5.13 (Chemins du graphe d'héritage) *Soit Δ un ensemble de coercions.*

- *Un chemin de coercions est toute liste non vide $[f_1; ...; f_n]$ ordonnée de coercions telles que $f_i : C_i \longmapsto C_{i+1} \in \Delta$ pour $i = 1, ..., n$. On écrit alors $[f_1; ...; f_n] : C_1 \longmapsto C_{n+1}$. Le nombre de paramètres de $[f_1; ...; f_n] : C_1 \longmapsto C_{n+1}$ est le nombre de paramètres de C_1.*
- *On étend l'application @ des coercions au cas des chemins de coercions par:*

$$[f_1]@t = f_1@t \text{ et } [f_1; ...; f_n]@t = [f_2; ...; f_n](f_1@t)$$

- *La composition de chemins de coercions est définie pour tous $[f_1; ...; f_n] : C \longmapsto D$ et $[g_1; ...; g_m] : D \longmapsto E$ par:*

$$[f_1; ...; f_n] \cdot [g_1; ...; g_m] = [f_1; ...; f_n; g_1; ...; g_m]$$

- *Nous notons l'ensemble des chemins de coercions dans Δ par Δ^+. Il est défini comme étant le plus petit ensemble vérifiant:*
  - $f : C \longmapsto D \in \Delta \Rightarrow [f] : C \longmapsto D \in \Delta^+$,
  - $f : C \longmapsto D \in \Delta \wedge p : D \longmapsto E \in \Delta^+ \Rightarrow [f] \cdot p : C \longmapsto D \in \Delta^+$.
  - *On dit que C est une* sous-classe *de D s'il existe $p : C \longmapsto D \in \Delta^+$.*

- *Nous appelons graphe tout ensemble fini de chemins de coercions. Nous étendons la notation* $\models$ *aux graphes;* $\Gamma \vdash \mathcal{G}$ *si et seulement si* $\Gamma \vdash \Delta$ *où* Δ *est l'ensemble des coercions apparaissant dans* $\mathcal{G}$.
- *Nous notons par* $\mathcal{G}_n$ *l'ensemble des chemins de coercions ayant n paramètres.*

Notation 5.1 *Pour mentionner explicitement les classes source C et cible D d'une coercion p d'un graphe* $\mathcal{G}$, *nous écrirons* $(p : C \longmapsto D) \in \mathcal{G}$.

Nous énonçons quelques lemmes simples sur la structure des coercions et le typage de leur application.

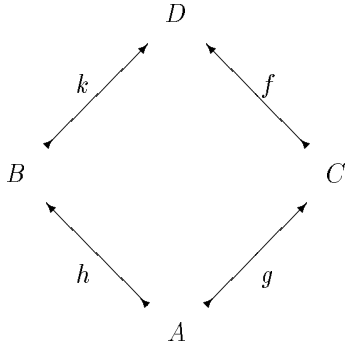
Lemme 5.1 *Soient* $\mathcal{G}$ *un graphe et* Γ *un contexte tels que* $\Gamma \models \mathcal{G}$.

1. *pour tout* $(p : C \longmapsto D) \in \mathcal{G}$, *si* $\Gamma \vdash a : (C\ a_1 ... a_n)$, *alors* $\Gamma \vdash p@a : p\{a : (C\ a_1 ... a_n)\}$.
2. *Si* $(p : C \longmapsto \mathsf{SORTCLASS}) \in \mathcal{G}$ *et* $\Gamma \vdash a : (C\ a_1 ... a_n)$, *alors* $p\{a : (C\ a_1 ... a_n)\} \in \mathcal{S}$.
3. *Si* $(p : C \longmapsto \mathsf{FUNCLASS}) \in \mathcal{G}$ *et* $\Gamma \vdash a : (C\ a_1 ... a_n)$, *alors il existe* A_1 *et* A_2 *tels que* $p\{a : (C\ a_1 ... a_n)\} \equiv (x : A_1)A_2$.
4. *Si* $(p : C \longmapsto D) \in \mathcal{G}$ *avec* $D \not\equiv \mathsf{SORTCLASS}$ *et* $D \not\equiv \mathsf{FUNCLASS}$ *et* $\Gamma \vdash a : (C\ a_1 ... a_n)$, *alors il existe des termes* $b_1, ..., b_m$ *tels que* $p\{a : (C\ a_1 ... a_n)\} \equiv (D\ b_1 ... b_m)$.

PREUVE Immédiat par l'hypothèse $\Gamma \models \mathcal{G}$ qui nous garantit le bon typage des coercions. Les jugements de typage sont démontrés par utilisation de la règle (APP) de typage de l'application. ■

5.5.2 Cohérence

Le problème général à résoudre est: étant donné un terme M de type $(C\ a_1 ... a_n)$ et un type $(D\ b_1 ... b_m)$, peut-on convertir M en un terme de type $(D\ b_1 ... b_m)$ en lui appliquant des coercions. Le même problème de cohérence vu en section 5.2.2 se pose pour notre système. En effet, on peut avoir deux coercions différentes entre deux classes C et D. Cette situation se présente notamment dans le cas du «diagramme du losange» ci-dessous, caractéristique de l'héritage multiple.



On peut contourner ce problème en adaptant la condition de cohérence de [16] au cas des classes avec paramètres.

Définition 5.14 (Cohérence) *Un ensemble Δ est cohérent si pour toute paire $(p, q : C \longmapsto D) \in \Delta^+$, on a $p@x \cong q@x$ avec $x : (C\ x_1 ... x_n)$, et $x_1,...,x_n$ de nouvelles variables quelconques.*

Cette solution n'est pas satisfaisante car elle est à la fois trop restrictive et trop permissive, car il existe une distorsion entre la conversion qui est une notion non syntaxique et la notion de classe qui est syntaxique.

- elle est trop restrictive car elle ne prend pas en compte le type d'arrivée des coercions. On peut en effet avoir deux coerions toujours différentes de la forme $p : (x : A)(C\ x) \to (D\ true)$ et $q : (x : A)(C\ x) \to (D\ false)$ qui pourtant ne vérifient pas la condition de cohérence.
- elle est trop permissive car elle ne prend pas complètement en compte la convertibilité des classes. Deux coercions $p : C \longmapsto D$ et $q : C' \longmapsto D'$ peuvent co-exister dans un graphe cohérent Δ même si C' et D' sont respectivement identiques à C et D, car la notion de classe est purement syntaxique.

Définition 5.15 (Déclaration de coercion) *Soient Δ un ensemble cohérent de coercions, et $f : C \longmapsto D$ une nouvelle coercion. Le graphe d'héritage obtenu après la déclaration de f est $\mathcal{G}$:*

$$\mathcal{G} = \begin{cases} (\Delta \cup f)^+ & si\ (\Delta \cup f)\ est\ coherent \\ \Delta^+ & sinon \end{cases}$$

Nous pouvons toujours avoir plusieurs chemins possibles entre deux classes dans $\mathcal{G}$, à condition qu'ils soient tous convertibles.

5.5.3 Une alternative à la condition de cohérence

Il existe plusieurs stratégies, autres que la condition de cohérence, pour garantir l'unicité de la coercion entre deux classes quelconques. Nous présentons la stratégie que nous avons choisie dans l'implémentation de notre système. Dans le cas de chemins de coercion multiples entre deux classes, seul le plus ancien chemin est valide. L'ordre de déclaration des coercions devient donc important. Nous considérons $\mathcal{G}$ l'ensemble des coercions (chemins) valides. Nous notons par $\mathcal{G} + f$ l'extension de $\mathcal{G}$ par une nouvelle coercion f.

Définition 5.16 *Soient $\mathcal{G}$ un ensemble de coercions valides, et $f : C \longmapsto D$ une nouvelle coercion.*

- *L'ensemble des coercions engendrées par la déclaration de f est :*

$$\begin{aligned} New(\mathcal{G}, f) = \quad & \{[f]\} \cup \\ & \{p \cdot [f] \mid (p : B \longmapsto C) \in \mathcal{G}\} \cup \\ & \{[f] \cdot q \mid (q : D \longmapsto E) \in \mathcal{G}\} \cup \\ & \{p \cdot [f] \cdot q \mid (p : B \longmapsto C) \in \mathcal{G} \ \wedge (q : D \longmapsto E) \in \mathcal{G}\} \end{aligned}$$

- *L'ensemble $\mathcal{G} + f$ est défini par:*

$$\mathcal{G} + f = \mathcal{G} \ \cup \ \{p \mid (p : X \longmapsto Y) \in New(\mathcal{G}, f) \wedge \ \nexists (q : X \longmapsto Y) \in \mathcal{G}\}$$

L'avantage d'une telle stratégie, outre sa simplicité, est la monotonie de la construction du graphe. Cette propriété n'est pas vérifiée dans le cas d'une stratégie favorisant le plus récent chemin. De plus, l'opération de déclaration de coercion préserve la propriété de validité du graphe.

Lemme 5.2 *Soient $\mathcal{G}$ un ensemble de coercions valides, et $f : C \longmapsto D$ une nouvelle coercion. Nous avons:*

- $\mathcal{G} \subseteq \mathcal{G} + f$.
- *Si $\mathcal{G}$ est valide, alors $\mathcal{G} + f$ l'est aussi.*

PREUVE Par construction de $\mathcal{G} + f$. ■

Notre présentation suggère un calcul statique des chemins de coercions, effectué à chaque déclaration de coercion. C'est un gain d'efficacité pour notre implémentation car les chemins de coercions sont calculés une fois pour toutes et non pas à chaque accès au graphe d'héritage.

Le comportement des deux stratégies que nous avons présentées diffère par leur ensemble de chemins valides. Avec la condition de cohérence, si la déclaration d'une coercion provoque l'incohérence du graphe, tous les chemins qu'elle engendre sont rejetés. Avec notre seconde stratégie, seuls les chemins multiples sont rejetés. Comme exemple, considérons le cas du «diagramme du losange» ci-dessus, où f est la dernière coercion déclarée. Supposons que $[g; f]$ et $[h; k]$ ne sont pas convertibles. Avec la condition de cohérence, le graphe obtenu est $\{[h], [k], [h; k], [g]\}$, alors qu'il est égal à $\{[h], [k], [h; k], [g], [f]\}$ avec notre seconde stratégie.

5.6 Fonction d'explicitation de termes

Après avoir défini notre langage des coercions, nous nous intéressons maintenant à son interprétation dans le langage explicite (langage officiel des PTS). Dans le système de G. Barthe[16], les coercions ne pouvaient être insérées qu'en position d'arguments dans les applications. Les positions d'applications dans notre système sont plus nombreuses; elles correspondent aux termes encadrés: $(a\ \boxed{b})$, $(\boxed{a}\ b)$, $[x : \boxed{A}]b$, $(x : \boxed{A})B$, $(x : A)\boxed{B}$, $a :: \boxed{A}$ et $\boxed{a} :: A$. Ces positions peuvent être combinées, donnant les cas $(\boxed{a}\ \boxed{b})$, $(x : \boxed{A})\boxed{B}$ et $\boxed{a} :: \boxed{A}$.

Nous n'adoptons pas la démarche de G. Barthe[16] qui consiste à définir une nouvelle théorie avec coerions implicites. Nous optons pour une interprétation directe des termes implicites dans les PTS (sans coercions). Nous restons fidèles à notre politique de considérer le langage implicite comme une simple interface du langage explicite. Cette interprétation prend la forme d'un algorithme d'inférence de types que nous appelerons aussi *algorithme d'explicitation.*

5.6.1 Insertion de coercions

Nous en venons maintenant aux fonctions d'application de coercions. Nous commençons par affiner notre fonction calculant la classe d'un terme $M : A$ de telle sorte que si A n'est pas une classe, nous calculons la classe de sa forme normale. Nous appelons Cl^* cette nouvelle fonction. Il ne s'agit toutefois que d'une heuristique puisqu'en toute généralité, un terme $M : A$ peut avoir plusieurs classes, correspondant à tous les noms de classe C_i pour lesquels il existe des termes $a_1^i ... a_{n_i}^i$ avec $A \cong (C_i\ a_1^i ... a_{n_i}^i)$.

Définition 5.17 (Classe d'un terme (fonction améliorée))

$$Cl^*(\mathcal{G}, \Gamma, A) = \begin{cases} Cl(A) & si\ Cl(A) \not\equiv \bot \\ Cl(\mathsf{Whnf}_\Gamma(A)) & sinon \end{cases}$$

Cette fonction est intéressante car elle est compatible avec la conversion dans le cas où l'on interdit les classes constantes et les projections (ces classes correspondent à des types non normaux, pouvant se simplifier). Nous ferons dorénavant référence à ce système par le *système restreint* 5.18. C'est un système que nous allons beaucoup étudier car il jouit de meilleures propriétés que le système général.

Définition 5.18 (Système restreint) *Nous appelons* système restreint *tout langage de coercions dont le graphe d'héritage ne contient ni des classes constantes ni des classes projections.*

Ce système contient le cas standard où les classes sont des structures et les coercions des projections entre structures. Mais il est encore plus général puisque les coercions ne sont pas astreintes à être exclusivement des projections, et les classes peuvent aussi êtres des variables globales (axiomes), des types inductifs (les exemples abondent: *nat*, *bool* ou *list* etc.) ou encore les classes abstraites SORTCLASS et FUNCLASS.

Nous montrons la compatibilité de la fonction Cl^* avec l'égalité définitionnelle.

Lemme 5.3 *Soient* Γ *un contexte et* $\mathcal{G}$ *vérifiant la restriction 5.18 n'est ni une constante, ni une projection. Nous avons:*

$$A \cong B \Rightarrow Cl^*(\mathcal{G}, \Gamma, A) = Cl^*(\mathcal{G}, \Gamma, B)$$

PREUVE Nous montrons d'abord que $Cl^*(\mathcal{G}, \Gamma, A) \equiv Cl(\mathsf{Whnf}_\Gamma(A))$. La preuve est par récurrence sur la structure du terme A. Un seul cas est intéressant, celui de l'application. Nous considérons trois sous-cas:

- nous avons $Cl^*(\mathcal{G}, \Gamma, (c\ a_1 ... a_n)) = Cl(\mathsf{Whnf}_\Gamma((c\ a_1 ... a_n)))$ lorsque c est une constante ou une projection; en effet $Cl((c\ a_1 ... a_n)) = \perp$ car $\mathcal{G}$ ne contient pas de classes constantes, ni de classes projections.
- si c peut être une classe (une variable globale, un type inductif, un constructeur, ou un enregistrement) alors nous avons $Cl^*(\mathcal{G}, \Gamma, (c\ a_1 ... a_n)) = Cl(\mathsf{Whnf}_\Gamma((c\ a_1 ... a_n))) = c$ si c est une classe et $Cl^*(\mathcal{G}, \Gamma, (c\ a_1 ... a_n)) = Cl(\mathsf{Whnf}_\Gamma((c\ a_1 ... a_n))) = \perp$ sinon (car $\mathsf{Whnf}_\Gamma((c\ a_1 ... a_n)) = (c\ a_1 ... a_n)$ d'après le lemme 3.8).
- pour les autres cas, nous avons $Cl^*(\mathcal{G}, \Gamma, (c\ a_1 ... a_n)) = Cl(\mathsf{Whnf}_\Gamma((c\ a_1 ... a_n))) = \perp$.

Puisque $A \cong B$, nous en déduisons que $Cl(\mathsf{Whnf}_\Gamma(A)) \equiv Cl(\mathsf{Whnf}_\Gamma(B))$ (puisque $\mathsf{Whnf}_\Gamma(A)$ et $\mathsf{Whnf}_\Gamma(B)$ ont même forme). Finalement:

$$Cl^*(\mathcal{G}, \Gamma, A) = Cl(\mathsf{Whnf}_\Gamma(A)) = Cl(\mathsf{Whnf}_\Gamma(B)) = Cl^*(\mathcal{G}, \Gamma, B)$$

■

Nous définissons maintenant des fonctions qui transforment un terme a de type A en une forme requise: une sorte, un type fonctionnel ou un type spécifique B. Dans les deux premiers cas, les fonctions retournent un couple comprenant le terme transformé et son type (une sorte pour Coerce_SORT et un type fonctionnel pour Coerce_FUN). La dernière fonction Coerce retourne seulement le terme transformé.
Il faut remarquer que les coercions ne sont appliquées que si a n'est pas déjà sous la forme requise.

Définition 5.19 (Fonctions de coercion)

• Coercion vers une sorte.

$$\mathsf{Coerce\_SORT}(\mathcal{G},\Gamma,a,A) = \begin{cases} a, s & si\ \mathsf{Whnf}_\Gamma(A) \equiv s \\ p@a, p\{a\} & si\ (p : Cl^*(\mathcal{G},\Gamma,A) \longmapsto \mathsf{SORTCLASS}) \in \mathcal{G} \\ \bot & sinon \end{cases}$$

• Coercion vers un type fonctionnel.

$$\mathsf{Coerce\_FUN}(\mathcal{G},\Gamma,a,A) = \begin{cases} a, (x : A_1)A_2 & si\ \mathsf{Whnf}_\Gamma(A) \equiv (x : A_1)A_2 \\ p@a, p\{a\} & si\ (p : Cl^*(\mathcal{G},\Gamma,A) \longmapsto \mathsf{FUNCLASS}) \in \mathcal{G} \\ \bot & sinon \end{cases}$$

• Coercion vers un type.

$$\mathsf{Coerce}(\mathcal{G},\Gamma,a,A,B) = \begin{cases} a & si\ A \cong B \\ p@a & si\ (p : Cl^*(\mathcal{G},\Gamma,A) \longmapsto Cl^*(\mathcal{G},\Gamma,B)) \in \mathcal{G}\ et\ p\{a\} \cong B \\ [x : B_1]\mathsf{Coerce}(\mathcal{G},(\Gamma; x : B_1),(a\ \mathsf{Coerce}(\mathcal{G},(\Gamma; x : B_1),x,B_1,A_1)),A_2,B_2) & \\ \qquad si\ \mathsf{Whnf}_\Gamma(A) \equiv (x : A_1)A_2\ et\ \mathsf{Whnf}_\Gamma(B) \equiv (x : B_1)B_2 & \\ \bot & sinon \end{cases}$$

La deuxième alternative dans la définition de Coerce correspond à l'application de la règle de sous-typage (contravariante) pour les types fonctionnels généralisée aux types dépendants comme suit:

$$\frac{p : B_1 \longmapsto A_1 \qquad q : A_2 \longmapsto B_2 \qquad a : (x : A_1)A_2}{[x : B_1](q@(a\ (p@x))) : (x : B_1)B_2}$$

On peut aussi introduire une optimisation permettant d'éviter des abstractions inutiles dans certains cas particuliers, suivant la règle:

$$\frac{q : A_2 \longmapsto B_2 \qquad [x : A_1]a_1 : (x : A_1)A_2}{[x : A_1](q@a_1) : (x : A_1)B_2}$$

Sans cette optimisation, on aurait obtenu le terme $[x : A_1](q@([x : A_1]a_1\ x))$. Il faut alors rajouter le nouveau cas ci-desous pour la fonction Coerce:

$$\begin{array}{rcl} \mathsf{Coerce}(\mathcal{G},\Gamma,a,A,B) & = & [x : A_3]\mathsf{Coerce}(\mathcal{G},(\Gamma; x : B_1),a_3,A_2,B_2) \\ & & \text{si } a \equiv [x : A_3]a_1,\ \mathsf{Whnf}_\Gamma(A) \equiv (x : A_1)A_2, \\ & & \mathsf{Whnf}_\Gamma(B) \equiv (x : B_1)B_2 \text{ et } A_3 \downarrow_\Gamma B_1 \end{array}$$

Propriétés Pour lier le terme implicite en entrée avec le résultat des fonctions d'insertion de coercions, nous définissons une règle de réduction ayant pour comportement d'effacer les coercions.

Définition 5.20 *La relation $\rightarrow_\nu$ est la plus petite congruence engendrée par la règle:*

$$\mathcal{G} \vdash (a\ a_1 ... a_n\ b) \rightarrow_\nu b\ si\ a \in \mathcal{G}_n$$

Cette règle seule n'est pas suffisante à exprimer le lien entre les termes explicite et implicite. En effet, soit le terme $[x : B_1](q@(a\ (p@x))) : (x : B_1)B_2$ obtenu à partir de $a : (x : A_1)A_2$ avec la règle de sous-typage des types fonctionnels. Ce terme peut se réduire comme suit (en supposant, pour simplifier, que le terme a n'est pas une coercion et qu'il est en forme normale): $\mathcal{G} \vdash [x : B_1](q@(a\ (p@x))) \rightarrow_\nu^* [x : B_1](a\ x)$. La η-réduction est alors nécessaire pour poursuivre la réduction en a.

Nous commençons par établire des résultats de correction pour les fonctions d'insertion de coercions. D'abord un lemme technique concernant la bonne fondation de la récurrence utilisée dans Coerce. Nous rappelons la définition de la notion de relation bien fondée.

Définition 5.21 (relation bien fondée) *Une relation $>$ est dite* bien fondée *s'il n'existe pas de chaîne infinie de la forme $a_1 > a_2 > ....$*

Définition 5.22

- *Soit $>^{st}$ la relation transitive des sous-termes définie par:*

$$(x : A)B >^{st} A,\ (x : A)B >^{st} B,\ [x : A]b >^{st} A,\ [x : A]b >^{st} b,\ (a\ b) >^{st} a \text{ et } (a\ b) >^{st} b$$

- *On désigne par $>^{st\beta\delta}$ la relation obtenue à partir de $>^{st}$ et $\rightarrow_{\beta\delta}$.*
- *On désigne par $>_p^{st\beta\delta}$ la relation sur les paires de termes définie par:*

$$(a_1, b_1) >_p^{st\beta\delta} (a_2, b_2) \Leftrightarrow (a_1 >^{st\beta\delta} a_2\ \wedge\ b_1 >^{st\beta\delta} b_2)\ \vee\ (a_1 >^{st\beta\delta} b_2\ \wedge\ b_1 >^{st\beta\delta} a_2)$$

Lemme 5.4 *Les relations $>^{st\beta\delta}$ et $>_p^{st\beta\delta}$ sont bien fondées.*

PREUVE Il est clair que la relation $>^{st}$ est bien fondée, puisque la taille des termes (nombre de symboles) diminue par $>^{st}$. Nous rappelons aussi que nous considérons des PTS avec la relation $\rightarrow_{\beta\delta}$ fortement normalisable (c'est à dire bien fondée).

1. Pour montrer la bonne fondation de $>^{st\beta\delta}$, nous montrons d'abord que $\rightarrow_{\beta\delta}$ et $>^{st}$ commutent de la manière suivante: si $a >^{st} b \rightarrow_{\beta\delta} c$ alors $a \rightarrow_{\beta\delta} b >^{st} c$. En effet:

$$\begin{array}{rcl}
(x : A)B >^{st} A \rightarrow_{\beta\delta} A' & \Longrightarrow & (x : A)B \rightarrow_{\beta\delta} (x : A')B >^{st} A' \\
(x : A)B >^{st} B \rightarrow_{\beta\delta} B' & \Longrightarrow & (x : A)B \rightarrow_{\beta\delta} (x : A)B' >^{st} B' \\
[x : A]b >^{st} A \rightarrow_{\beta\delta} A' & \Longrightarrow & [x : A]b \rightarrow_{\beta\delta} [x : A']b >^{st} A' \\
[x : A]b >^{st} b \rightarrow_{\beta\delta} b' & \Longrightarrow & [x : A]b \rightarrow_{\beta\delta} [x : A]b' >^{st} b' \\
(a\ b) >^{st} a \rightarrow_{\beta\delta} a' & \Longrightarrow & (a\ b) \rightarrow_{\beta\delta} (a'\ b) >^{st} a' \\
(a\ b) >^{st} b \rightarrow_{\beta\delta} b' & \Longrightarrow & (a\ b) \rightarrow_{\beta\delta} (a\ b') >^{st} b'
\end{array}$$

   Supposons maintenant que $>^{st\beta\delta}$ n'est pas bien fondée; il existe donc une chaîne infinie de la forme $a_1 >^{st\beta\delta} a_2 >^{st\beta\delta} ....$ D'après la propriété de commutation ci-dessus, on peut ramener toutes les $\rightarrow_{\beta\delta}$-réductions en début de cette chaîne: $a_{i_1} \rightarrow_{\beta\delta} a_{i_2} \rightarrow_{\beta\delta} ...a_{j_1} >^{st} a_{j_2} >^{st} ....$ On extrait alors de cette suite une suite infinie de $\rightarrow_{\beta\delta}$-réductions ou une suite infinie ordonnée par $>^{st}$. Les deux cas contredisent la bonne fondation de $\rightarrow_{\beta\delta}$ et de $>^{st}$. Conclusion: la relation $>^{st\beta\delta}$ est bien fondée.

2. Montrons la bonne fondation de $>_p^{st\beta\delta}$. Supposons que $>_p^{st\beta\delta}$ n'est pas bien fondée; il existe donc une chaîne infinie $(a_1, b_1) >_p^{st\beta\delta} (a_2, b_2) >_p^{st\beta\delta}$ .... De cette chaîne, on peut construire une autre chaîne infinie $a_1 >^{st\beta\delta} X_2 >^{st\beta\delta} X_3$... où X_i est a_i ou b_i suivant la «valeur» de $(a_{i-1}, b_{i-1}) >_p^{st\beta\delta} (a_i, b_i)$. Or nous venons de démontrer que $>^{st\beta\delta}$ est une relation bien fondée. Conclusion: la relation $>_p^{st\beta\delta}$ est bien fondée.

■

Lemme 5.5 *Soient $\mathcal{G}$ un graphe d'héritage et Γ un contexte tels que $\Gamma \models \mathcal{G}$.*

1. *Si $\Gamma \vdash a : A$, alors* $\mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, a, A) \equiv \perp$ *ou* $\mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, a, A) \equiv a', s$ *avec $\Gamma \vdash a' : s$ et $\mathcal{G} \vdash a' \rightarrow_\nu^* a$.*
2. *Si $\Gamma \vdash a : A$, alors* $\mathsf{Coerce\_FUN}(\mathcal{G}, \Gamma, a, A) \equiv \perp$ $\mathsf{Coerce\_FUN}(\mathcal{G}, \Gamma, a, A) \equiv a', (x : A_1)A_2$ *avec $\Gamma \vdash a' : (x : A_1)A_2$ et $\mathcal{G} \vdash a' \rightarrow_\nu^* a$.*
3. *Si $\Gamma \vdash a : A$ et $\Gamma \vdash B : s$, alors* $\mathsf{Coerce}(\mathcal{G}, \Gamma, a, A, B) \equiv \perp$ *ou $\Gamma \vdash \mathsf{Coerce}(\mathcal{G}, \Gamma, a, A, B) : B$ et $\mathcal{G} \vdash \mathsf{Coerce}(\mathcal{G}, \Gamma, a, A, B) \rightarrow_{\nu\eta}^* a$.*

PREUVE

1. Par simple analyse de cas, et en utilisant le lemme 5.1.
2. idem.
3. Par analyse de cas:
   (a) si $A \cong B$, alors $\Gamma \vdash a : B$ par la règle (CONV).
   (b) si $(p : Cl^*(\mathcal{G}, \Gamma, A) \longrightarrow Cl^*(\mathcal{G}, \Gamma, B)) \in \mathcal{G}$ et $p\{a\} \cong B$, alors $\Gamma \vdash p@a : p\{a\}$ par le lemme 5.1. Par la règle (CONV), $\Gamma \vdash a : B$. De plus $\mathcal{G} \vdash p@a \rightarrow_\nu^* a$.
   (c) supposons $\mathsf{Whnf}_\Gamma(A) \equiv (x : A_1)A_2$ et $\mathsf{Whnf}_\Gamma(B) \equiv (x : B_1)B_2$. Puisque $\Gamma \vdash a : A$, on a $\Gamma \vdash a : (x : A_1)A_2$ par la règle (CONV). Aussi $\Gamma \vdash (x : B_1)B_2 : s$, par le lemme 2.8 avec $\Gamma \vdash B : s$. Par les lemmes de génération (2.5), il existe des sortes s_1, s_2, s_1', $s_2' \in \mathcal{S}$ telles que $\Gamma \vdash A_1 : s_1$, $\Gamma \vdash A_2 : s_2$, $\Gamma \vdash B_1 : s_1'$ et $\Gamma \vdash B_2 : s_2'$. Par conséquent le contexte $\Gamma; x : B_1$ est bien formé.
   La preuve se fait par récurrence sur les termes A et B, avec l'ordre $>_p^{st\beta\delta}$. Par récurrence (car $(A, B) >_p^{st\beta\delta} (B_1, A_1)$), nous avons:

$$\Gamma; x : B_1 \vdash \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1) : A_1$$

$$\mathcal{G} \vdash \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1) \rightarrow_{\nu\eta}^* x$$

   Aussi, par induction (car $(A, B) >_p^{st\beta\delta} (A_2, B_2)$), nous obtenons:

$$\Gamma; x : B_1 \vdash \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_E 1), (a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1)), A_2, B_2) : B_2$$

$$\mathcal{G} \vdash \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), (a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1)), A_2, B_2)$$

$$\downarrow_{\nu\eta}^*$$

$$(a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1))$$

Par la règle (LAM), et puisque $\Gamma \vdash (x : B_1)B_2 : s$, nous concluons par:

$\Gamma \vdash [x : B_1]\mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), (a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1)), A_2, B_2) : (x : B_1)B_2$

Finalement, par la règle (CONV), nous aboutissons au résultat souhaité:

$$\Gamma \vdash [x : B_1]\mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), (a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1)), A_2, B_2) : B$$

De plus,

$$\begin{array}{c} \mathcal{G} \vdash [x : B_1]\mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), (a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1)), A_2, B_2) \\ \downarrow^*_{\nu\eta} \\ [x : B_1](a\ \mathsf{Coerce}(\mathcal{G}, (\Gamma; x : B_1), x, B_1, A_1)) \\ \downarrow^*_{\nu\eta} \\ [x : B_1](a\ x) \\ \downarrow_{\eta} \\ a \end{array}$$

■

5.6.2 Algorithme d'inférence de types

Dans notre nouvel algorithme d'inférence de types, les fonctions d'insertion de coercions qu'on vient de définir remplacent Whnf et la conversion; plus exactement, Coerce_SORT et Coerce_FUN tiennent le rôle de Whnf et Coerce remplace la conversion. Le typage se fait relativement à un contexte et à un graphe recensant tous les chemins de coercions valides. Le typage d'un terme (implicite) M s'effectue par $\mathsf{Infer}(\mathcal{G}, \Gamma, M)$; le résultat retourné est un terme (explicite) M' et un type (explicite aussi) A.

En toutes généralité, à un terme implicite peut correspondre plusieurs termes explicites correspondant à toutes les insertions de coercions valides possibles. À cela deux raisons:

1. Nous savons que dans les PTS, un terme M peut avoir plusieurs types, disons A et A', modulo la convertibilité. Comme la détermination de la classe d'un type (les fonctions Cl et Cl^*) ne respecte pas la convertibilité, ces types peuvent correspondre à des classes différentes $Cl(A)$ et $Cl(A')$. Ainsi le terme M peut être transformé en deux termes différents $p@M$ et $q@M$ avec les coercions $p : Cl(A) \longmapsto C$ et $q : Cl(A') \longmapsto D$. Ce probème ne se pose pas lorsque les constantes et les projections ne peuvent pas être des classes, puisque le lemme 5.3 nous garantit l'unicité de la classe d'un terme (par Cl^*) dans ce cas.

2. Il est possible d'inserer des coercions dans un terme déjà bien typé. Soit M un terme explicite bien typé, on peut le transformer en tous les termes $p@M$ bien typé (p est une coercion appropriée).

Pour garantir l'unicité de la procédure d'explicitation, il nous faut trouver une parade à chacune de ces deux situations:

1. La détermination de la classe d'un terme est réalisée seulement à partir de son type inféré.
2. On interdit l'insertion de coercions dans des termes déjà bien typés.

Définition 5.23 (Algorithme d'inférence)

$$
\begin{array}{lcl}
\mathsf{Infer}(\mathcal{G},\Gamma,s_1) & = & s_1,s_2 \quad avec\ \langle s_1,s_2\rangle \in \mathcal{A} \\
\mathsf{Infer}(\mathcal{G},\Gamma,x) & = & x,\Gamma_x \\
\mathsf{Infer}(\mathcal{G},\Gamma,c) & = & c,\Gamma_c \\
\mathsf{Infer}(\mathcal{G},\Gamma,(x:A)B) & = & (x:A')B',\mathcal{R}_{\langle s_1,s_2\rangle} \\
 & & avec\ \mathsf{Coerce\_SORT}(\mathcal{G},\Gamma,\mathsf{Infer}(\mathcal{G},\Gamma,A))\equiv A',s_1\ et \\
 & & \mathsf{Coerce\_SORT}(\mathcal{G},(\Gamma;x:A'),\mathsf{Infer}(\mathcal{G},(\Gamma;x:A'),B))\equiv B',s_2 \\
\mathsf{Infer}(\mathcal{G},\Gamma,[x:A]b) & = & [x:A']b',(x:A')B' \\
 & & avec\ \mathsf{Coerce\_SORT}(\mathcal{G},\Gamma,\mathsf{Infer}(\mathcal{G},\Gamma,A))\equiv A',s_1, \\
 & & \mathsf{Infer}(\mathcal{G},\Gamma;x:A',b)\equiv b',B'\ et \\
 & & (\mathsf{Whnf}_\Gamma(B')\in\mathcal{A}_o \vee \mathsf{Whnf}_\Gamma(B')\notin\mathcal{S})\wedge\exists s_2,s_3\in\mathcal{S}.\langle s_1,s_2,s_3\rangle\in\mathcal{R} \\
\mathsf{Infer}(\mathcal{G},\Gamma,(a\ b)) & = & (a'\ b'),A'[x\leftarrow b'] \\
 & & avec\ \mathsf{Coerce\_FUN}(\mathcal{G},\Gamma,\mathsf{Infer}(\mathcal{G},\Gamma,a))\equiv a',(x:B')A'\ et \\
 & & \mathsf{Coerce}(\mathcal{G},\Gamma,\mathsf{Infer}(\mathcal{G},\Gamma,b),B')\equiv b' \\
\mathsf{Infer}(\mathcal{G},\Gamma,a::A) & = & a'::A',A' \\
 & & avec\ \mathsf{Coerce\_SORT}(\mathcal{G},\Gamma,\mathsf{Infer}(\mathcal{G},\Gamma,A))\equiv A',s\ et \\
 & & \mathsf{Coerce}(\mathcal{G},\Gamma,\mathsf{Infer}(\mathcal{G},\Gamma,a),A')\equiv a'
\end{array}
$$

5.6.3 Propriétés

Correction La première propriété de correction consiste à vérifier que le terme explicite est bien typé. De plus, nous devons établir que le terme explicite est obtenu à partir du terme implicite par insertion de coercions.

Théorème 6 (Correction)

Si $\mathsf{Infer}(\mathcal{G},\Gamma,M)\equiv M',A$, *alors* $\Gamma\vdash M':A$ *et* $\mathcal{G}\vdash M'\rightarrow^*_{\nu\eta} M$.

PREUVE Nous procédons par récurrence sur la structure du terme à typer. Aucun cas n'est véritablement difficile à démontrer, mais nous préférons tout de même détailler la preuve de tous les cas.

- Les trois premiers cas, relatifs aux sortes, variables et constantes sont triviaux.
- Cas du produit. Nous avons $\mathsf{Infer}(\mathcal{G},\Gamma,(x:A)B)\equiv(x:A')B',\mathcal{R}_{\langle s_1,s_2\rangle}$ avec:
  $(H_1):\mathsf{Infer}(\mathcal{G},\Gamma,A)\equiv M,X$
  $(H_2):\mathsf{Coerce\_SORT}(\mathcal{G},\Gamma,M,X)\equiv A',s_1$
  $(H_3):\mathsf{Infer}(\mathcal{G},(\Gamma;x:A'),B)\equiv N,Y$
  $(H_4):\mathsf{Coerce\_SORT}(\mathcal{G},(\Gamma;x:A'),N,Y)\equiv B',s_2$
  Le lemme 5.5 appliqué à (H_2) et à (H_4) nous fournit les énoncés suivants:

$(H_5) : \Gamma \vdash A' : s_1$
$(H_6) : \mathcal{G} \vdash A' \rightarrow^*_{\nu\eta} M$
$(H_7) : \Gamma; x : A' \vdash B' : s_2$
$(H_8) : \mathcal{G} \vdash B' \rightarrow^*_{\nu\eta} N$

Nous remarquons d'abord que le contexte $\Gamma; x : A'$ est bien formé car (H_5). Par hypothèse de récurrence sur (H_1), nous obtenons:

$(H_9) : \mathcal{G} \vdash M \rightarrow^*_{\nu\eta} A$

Aussi par hypothèse de récurrence sur (H_3):

$(H_{10}) : \mathcal{G} \vdash N \rightarrow^*_{\nu\eta} B$

En combinant (H_5) et (H_7) avec la règle (PROD), nous obtenons $\Gamma \vdash (x : A')B' : \mathcal{R}_{\langle s_1, s_2\rangle}$.
Par transitivité de la relation $\rightarrow^*_{\nu\eta}$, nous obtenons:

$(H_6) + (H_9) : \mathcal{G} \vdash A' \rightarrow^*_{\nu\eta} A$
$(H_8) + (H_{10}) : \mathcal{G} \vdash B' \rightarrow^*_{\nu\eta} B$

Puis par congruence: $\mathcal{G} \vdash (x : A')B' \rightarrow^*_{\nu\eta} (x : A)B$.

- Cas de l'abstraction. Nous avons $\mathsf{Infer}(\mathcal{G}, \Gamma, [x : A]b) \equiv [x : A']b', (x : A')B'$ avec:

$(H_1) : \mathsf{Infer}(\mathcal{G}, \Gamma, A) \equiv M, X$
$(H_2) : \mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, M, X) \equiv A', s_1$
$(H_3) : \mathsf{Infer}(\mathcal{G}, (\Gamma; x : A'), b) \equiv b', B'$
$(H_4) : (\mathsf{Whnf}_\Gamma(B') \in \mathcal{A}_o \vee \mathsf{Whnf}_\Gamma(B') \notin \mathcal{S}) \wedge \exists s_2, s_3 \in \mathcal{S}.\langle s_1, s_2, s_3\rangle \in \mathcal{R}$

Le lemme 5.5 appliqué à (H_2), nous fournit:

$(H_5) : \Gamma \vdash A' : s_1$
$(H_6) : \mathcal{G} \vdash A' \rightarrow^*_{\nu\eta} M$

Par hypothèse de récurrence sur (H_1) et (H_3):

$(H_7) : \mathcal{G} \vdash M \rightarrow^*_{\nu\eta} A$
$(H_8) : \Gamma; x : A' \vdash b' : B'$
$(H_9) : \mathcal{G} \vdash b' \rightarrow^*_{\nu\eta} b$

Nous obtenons alors:

$(H_5) + (H_8) + (H_4) + (\text{LAM}) : \Gamma \vdash [x : A']b' : (x : A')B'$

Enfin, par transitivité et congruence de $\rightarrow_{\nu\eta}$:

$(H_6) + (H_7) + (H_9) : \mathcal{G} \vdash [x : A']b' \rightarrow^*_{\nu\eta} [x : A]b$

- Cas de l'application. Nous avons $\mathsf{Infer}(\mathcal{G}, \Gamma, (a\ b)) \equiv (a'\ b'), A'[x \leftarrow b']$ avec:

$(H_1) : \mathsf{Infer}(\mathcal{G}, \Gamma, a) \equiv M, X$
$(H_2) : \mathsf{Coerce\_FUN}(\mathcal{G}, \Gamma, M, X) \equiv a', (x : B')A'$
$(H_3) : \mathsf{Infer}(\mathcal{G}, \Gamma, b) \equiv N, Y$
$(H_4) : \mathsf{Coerce}(\mathcal{G}, \Gamma, N, Y, B') \equiv b'$

Par hypothèse de récurrence, nous avons:

$(H_5) : \mathcal{G} \vdash M \rightarrow^*_{\nu\eta} a$
$(H_6) : \mathcal{G} \vdash N \rightarrow^*_{\nu\eta} b$

Et par application du lemme 5.5 aux hypothèses (H_2) et (H_4), nous obtenons:

$(H_7) : \Gamma \vdash a' : (x : B')A'$
$(H_8) : \mathcal{G} \vdash a' \rightarrow^*_{\nu\eta} M$
$(H_9) : \Gamma \vdash b' : B'$
$(H_{10}) : \mathcal{G} \vdash b' \rightarrow^*_{\nu\eta} N$

Puis finalement:

$(H_7) + (H_8) + (\textsc{App}) : \Gamma \vdash (a'\ b') : A'[x \leftarrow b']$

Enfin, par transitivité et congruence de $\rightarrow_{\nu\eta}$:

$(H_8) + (H_5) + (H_{10}) + (H_6) : \mathcal{G} \vdash (a'\ b') \rightarrow^*_{\nu\eta} (a\ b)$

- Cas du cast. Nous avons $\mathsf{Infer}(\mathcal{G}, \Gamma, a :: A) \equiv a' :: A', A'$ avec:

  $(H_1) : \mathsf{Infer}(\mathcal{G}, \Gamma, A) \equiv M, X$

  $(H_2) : \mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, M, X) \equiv A', s$

  $(H_3) : \mathsf{Infer}(\mathcal{G}, \Gamma, a) \equiv N, Y$

  $(H_4) : \mathsf{Coerce}(\mathcal{G}, \Gamma, N, Y, A') \equiv a'$

  Par hypothèse de récurrence, nous avons:

  $(H_5) : \mathcal{G} \vdash M \rightarrow^*_{\nu\eta} A$

  $(H_6) : \mathcal{G} \vdash N \rightarrow^*_{\nu\eta} a$

  Et par application du lemme 5.5 aux hypothèses (H_2) et (H_4), nous obtenons:

  $(H_7) : \Gamma \vdash A' : s$

  $(H_8) : \mathcal{G} \vdash A' \rightarrow^*_{\nu\eta} M$

  $(H_9) : \Gamma \vdash a' : A'$

  $(H_{10}) : \mathcal{G} \vdash a' \rightarrow^*_{\nu\eta} N$

  Puis finalement:

  $(H_7) + (H_9) + (\textsc{Cast}) : \Gamma \vdash a' :: A' : A'$

  Enfin, par transitivité et congruence de $\rightarrow_{\nu\eta}$:

  $(H_8) + (H_5) + (H_{10}) + (H_6) : \mathcal{G} \vdash a' :: A' \rightarrow^*_{\nu\eta} a :: A$

■

Conservation Une deuxième propriété importante est celle qui garantit que des coercions ne sont insérées que lorsque c'est strictement nécessaire. Ainsi l'explicitation d'un terme bien typé est lui-même.

Théorème 7 (Conservation)

Si $\Gamma \vdash M : A$, *alors* $\mathsf{Infer}(\mathcal{G}, \Gamma, M) \equiv M, B$ *avec* $A \cong B$.

PREUVE Par récurrence sur la structure des dérivations de la relation $\vdash$.

- Les cas (AXIOM) et (NAME) sont triviaux.

- Cas (PROD). Nous avons $\Gamma \vdash (x : A)B : R_{\langle s_1, s_2\rangle}$ avec:

  $(H_1) : \Gamma \vdash A : s_1$

  $(H_2) : \Gamma; x : A \vdash B : s_2$

  Par récurrence sur (H_1) et (H_2):

  $(H_3) : \mathsf{Infer}(\mathcal{G}, \Gamma, A) \equiv A, X$

  $(H_4) : s_1 \cong X$

  $(H_5) : \mathsf{Infer}(\mathcal{G}, (\Gamma; x : A), B) \equiv B, Y$

  $(H_6) : s_2 \cong Y$

  Nous avons:

  $(H_7) : \mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, A, X) \equiv A, s_1$ car (H_4)

  $(H_8) : \mathsf{Coerce\_SORT}(\mathcal{G}, (\Gamma; x : A), B, Y) \equiv B, s_2$ car (H_6)

  D'où:

  $(H_3) + (H_7) + (H_5) + (H_8) : \mathsf{Infer}(\mathcal{G}, \Gamma, (x : A)B) \equiv (x : A)B, \mathcal{R}_{\langle s_1, s_2\rangle}$

- Cas (LAM). Nous avons $\Gamma \vdash [x : A]b : (x : A)B$ avec:
  $(H_1) : \Gamma \vdash A : s_1$
  $(H_2) : \Gamma; x : A \vdash b : B$
  $(H_3) : (B \in \mathcal{A}_o \vee B \notin \mathcal{S}) \wedge \langle s_1, s_2, s_3 \rangle \in \mathcal{R}$
  Par récurrence:
  $(H_4) : \mathsf{Infer}(\mathcal{G}, \Gamma, A) \equiv A, X$
  $(H_5) : s_1 \cong X$
  $(H_6) : \mathsf{Infer}(\mathcal{G}, (\Gamma; x : A), b) \equiv b, Y$
  $(H_7) : B \cong Y$
  Nous avons:
  $(H_8) : \mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, A, X) \equiv A, s_1$ car (H_5)
  $(H_9) : (\mathsf{Whnf}_\Gamma(Y) \in \mathcal{A}_o \vee \mathsf{Whnf}_\Gamma(Y) \notin \mathcal{S}) \wedge \langle s_1, s_2, s_3 \rangle \in \mathcal{R}$ car $(H_3) + (H_7)$
  D'où:
  $(H_4) + (H_8) + (H_6) + (H_9) : \mathsf{Infer}(\mathcal{G}, \Gamma, [x : A]b) \equiv [x : A]b, (x : A)Y$
  et $(x : A)B \cong (x : A)Y$ car (H_7).

- Cas (APP). Nous avons $\Gamma \vdash (a\ b) : A[x \leftarrow b]$ avec:
  $(H_1) : \Gamma \vdash a : (x : B)A$
  $(H_2) : \Gamma \vdash b : B$
  Par récurrence:
  $(H_3) : \mathsf{Infer}(\mathcal{G}, \Gamma, a) \equiv a, X$
  $(H_4) : (x : B)A \cong X$
  $(H_5) : \mathsf{Infer}(\mathcal{G}, \Gamma, b) \equiv b, Y$
  $(H_6) : B \cong Y$
  On a:
  $(H_7) : \mathsf{Coerce\_FUN}(\mathcal{G}, \Gamma, a, X) \equiv a, (x : B)A$ car (H_4)
  $(H_8) : \mathsf{Coerce}(\mathcal{G}, \Gamma, b, Y, B) \equiv b$ car (H_6)
  En conclusion:
  $(H_3+)(H_7) + (H_5) + (H_8) : \mathsf{Infer}(\mathcal{G}, \Gamma, (a\ b)) \equiv (a\ b), A[x \leftarrow b]$

- Cas (CAST). Nous avons $\Gamma \vdash a :: A : A$ avec:
  $(H_1) : \Gamma \vdash A : s$
  $(H_2) : \Gamma \vdash a : A$
  Par récurrence:
  $(H_4) : \mathsf{Infer}(\mathcal{G}, \Gamma, A) \equiv A, X$
  $(H_5) : s \cong X$
  $(H_6) : \mathsf{Infer}(\mathcal{G}, \Gamma, a) \equiv a, Y$
  $(H_7) : A \cong Y$
  On a:
  $(H_8) : \mathsf{Coerce\_SORT}(\mathcal{G}, \Gamma, A, X) \equiv A, s$ car (H_5)
  $(H_9) : \mathsf{Coerce}(\mathcal{G}, \Gamma, a, Y, A) \equiv a$ car (H_7)
  D'où:
  $(H_4) + (H_8) + (H_6) + (H_9) : \mathsf{Infer}(\mathcal{G}, \Gamma, a :: A) \equiv a :: A, A$

■

Monotonie L'algorithme d'inférence Infer souffre d'un défaut assez invalidant. Il n'est pas monotone, c'est à dire un terme bien typé dans un graphe $\mathcal{G}$ ne l'est pas forcément dans $\mathcal{G} + p$.

Nous commençons par donner un exemple de cas où $\mathsf{Infer}(\mathcal{G},\Gamma,M) \equiv M', A$ et $\mathsf{Infer}(\mathcal{G}+p,\Gamma,M) \equiv M'', B$ tels que $M' \ncong M''$ et $A \ncong B$. Considérons le contexte suivant:

$$...;\ A : nat \to *;\ B : (A\ 1) \to *;\ A'' : nat \to *;\ A' := (A''\ 0);\ f : (x : (A\ 1))(B\ x);\ a : A';$$
$$p : (x : nat)(A''\ x) \to (A\ x + 1);\ q : A' \to (A\ 1);\ r : A' \to (A\ 3)$$

Le terme implicite $(f\ a)$ est bien typé dans le graphe $\mathcal{G} = [p : A'' \longmapsto A]$; en effet $\mathsf{Infer}(\mathcal{G},\Gamma,(f\ a)) = (f\ (p\ 0\ a)),(B\ (p\ 0\ a))$. Dans le graphe étendu $\mathcal{G} + q = [p : A'' \longmapsto A;\ q : A' \longmapsto A]$, le résultat de $\mathsf{Infer}(\mathcal{G},\Gamma,(f\ a)) = (f\ (q\ a)),(B\ (q\ a))$ est différent.
Pire encore, $(f\ a)$ est mal typé dans $\mathcal{G} + r = [p : A'' \longmapsto A;\ r : A' \longmapsto A]$. La raison est simple: Cl^* n'est pas monotone. Ainsi:

- $Cl^*(\mathcal{G},\Gamma,A') = A''$ et
- $Cl^*(\mathcal{G}+q,\Gamma,A') = Cl^*(\mathcal{G}+r,\Gamma,A') = A'$.

L'inférence de types est par contre monotone pour le système restreint 5.18.

Lemme 5.6 *Nous nous plaçons dans le cadre du système restreint 5.18. Soient* Γ *et* Γ' *deux contextes,* $\mathcal{G}$ *et* $\mathcal{G}'$ *deux graphes,* A *un terme bien typé dans* Γ *(il existe un type* T *tel que* $\Gamma \vdash A : T$*) et* C *une classe (*$C \not\equiv \bot$*) tels que* $\Gamma \subseteq \Gamma'$, $\mathcal{G} \subseteq \mathcal{G}'$, $\Gamma \models \mathcal{G}$, $\Gamma' \models \mathcal{G}'$. *On a:*

$$Cl^*(\mathcal{G},\Gamma,A) \equiv C \Rightarrow Cl^*(\mathcal{G}',\Gamma',A) \equiv C$$

PREUVE Découle de 5.3 qui stipule que dans le système restreint 5.18, tout terme a au plus une classe. ■

Théorème 8 (Monotonie) *Nous nous plaçons dans le cadre du système restreint 5.18. Soient* Γ *et* Γ' *deux contextes,* $\mathcal{G}$ *et* $\mathcal{G}'$ *deux graphes et* M, M' *et* A *des termes tels que* $\Gamma \subseteq \Gamma'$, $\mathcal{G} \subseteq \mathcal{G}'$, $\Gamma \models \mathcal{G}$, $\Gamma' \models \mathcal{G}'$. *On a:*

$$\mathsf{Infer}(\mathcal{G},\Gamma,M) \equiv M', A \Rightarrow \mathsf{Infer}(\mathcal{G}',\Gamma',M) \equiv M', A$$

PREUVE Par récurrence sur la dérivation de $\mathsf{Infer}(\mathcal{G},\Gamma,M) \equiv M', A$, puis en construisant une dérivation de $\mathsf{Infer}(\mathcal{G}',\Gamma',M) \equiv M', A$ en utilisant le résultat $Cl^*(\mathcal{G},\Gamma,X) \equiv C \Rightarrow Cl^*(\mathcal{G}',\Gamma',X) \equiv C$ (lemme 5.6). ■

5.7 Coercions entre types inductifs paramétrés

Notre mécanisme d'héritage ne prend pas en compte les types de données. Il ne peut pas par exemple transformer une liste d'entiers naturels en une liste de rationnels, même s'il existe une coercion entre les naturels et les rationnels. La situation est similaire pour les autres types de données tels que les types produits, les arbres, les vecteurs etc. À chaque type de donnée correspond une règle de sous-typage, par exemple:

$$\frac{A \le B}{(list\ A) \le (list\ B)}$$

$$\frac{A \le B \qquad A' \le B'}{A * A' \le B * B'}$$

Tous ces types de données sont des cas particuliers de types inductifs paramétrés. Ainsi la définition de liste est donnée dans la section 3.3, et voici la définition du type produit:

$Inductive\ prod\ [A:*;B:*]\ :*:=pair:A\to B\to (prod\ A\ B)$

Nous suggérons dans cette section une manière possible de prendre en compte les coercions entre types de données.

Dans le cas des listes, la coercion $(Iter\_list\ A\ B\ f)$ entre $(list\ A)$ et $(list\ B)$ est recursivement définie pour toute fonction f entre A et B par:

$$\begin{array}{lcl}(Iter\_list\ A\ B\ f\ (nil\ A)) & = & (nil\ B)\\ (Iter\_list\ A\ B\ f\ (cons\ A\ a\ l_1)) & = & (cons\ B\ (f\ a)\ (Iter\_list\ A\ B\ f\ l_1))\end{array}$$

Ce qui donne dans le calcul des constructions inductives:

$$\begin{array}{lll}Fixpoint\ Iter\_list\ :=\ [A:*][B:*][f:A\to B][l:(listA)]\ <(listB)>Case\ l\ of & & \\ \quad (nil\ B) & & \\ \quad |\ [a:A][l_1:(list\ A)](cons\ B\ (f\ a)\ (Iter\_list\ A\ B\ f\ l_1))\ end. & & \end{array}$$

Pour les produits, nous avons deux fonctions en paramètres (correspondant au nombre de paramètres de $prod$) $f:A\to B$ et $f':A'\to B'$.

$$(Iter\_prod\ A\ B\ f\ A'\ B'\ f'\ (pair\ a\ a'))\ =\ (pair\ (f\ a)\ (f'\ a'))$$

Ces définitions peuvent être automatiquement construites pour tout type inductif paramétré, suivant la forme de ses constructeurs.

Voyons maintenant comment utiliser ces coercins dans notre système. Soient C et D deux classes, et l : $(list\ (C\ a_1...a_n))$ un terme à transformer en un terme de type $(list\ (D\ b_1...b_m))$. S'il existe une coercion p : $(x_1:A_1)...(x_n:A_n)(y:(C\ a_1...a_n))(D\ u_1...u_m)$ entre C et D, alors $(Iter\_list\ (C\ a_1...a_n)\ (D\ b_1...b_m)\ (p\ a_1...a_n))$ est la coercion à appliquer à l à condition que le type de $(p\ a_1...a_n)$ soit compatible avec $(D\ b_1...b_m)$.

Chapitre 6

Tactiques de Réécriture

6.1 Introduction

La réécriture est le processus qui consiste à utiliser des équations comme des règles de transformation afin de simplifier des termes. Elle peut aussi être utilisée pour démontrer des égalités. C'est notamment le cas lorsqu'on dispose d'un ensemble *complet* de règles de réécriture, capable de réécrire n'importe quel terme en sa forme normale[78]. Pour tester la validité d'une équation, il suffit alors de comparer syntaxiquement les formes normales des deux membres de l'équation.

La réécriture est aussi une technique utilisée dans les systèmes de preuves et les programmes de calcul formel. Il est toutefois impossible, dans le cadre des systèmes de preuve, de concevoir un mécanisme de réécriture assez général pour convenir aux différentes situations susceptibles de se présenter. En effet, il y a une grande variété de stratégies de réécriture correspondant entre autres à l'ordre d'application des règles, au sens de parcours des termes (du haut vers le bas ou du bas vers le haut), ainsi qu'à l'ordre d'orientation des équations (de la droite vers la gauche ou de la gauche vers la droite). La solution consistant à appliquer de manière exhaustive et répétitive chaque règle à tous les nœuds du terme est souvent mauvaise. D'abord parce qu'elle n'exploite pas la structure des règles: certaines formes de règles permettent des simplifications en une seule passe. Un autre problème est que ce processus peut boucler: c'est le cas de la règle de commutativité.

Le processus de réécriture, dans le cadre des systèmes de preuve, a de plus deux particularités:

- Toute étape de réécriture doit être justifiée. Il est donc hors de question d'utiliser un programme de calcul formel pour tester la validité d'équations qu'on va ensuite poser en axiomes.
- Le contexte traditionnel de la réécriture est une théorie équationnelle avec une égalité qui est une congruence sur les termes de cette théorie. Dans les systèmes de preuves basés sur la théorie des types, il n'existe pas qu'une seule relation d'égalité. L'utilisateur peut définir de nouvelles relations d'égalité qu'il peut utiliser pour réécrire.

Pour répondre à toutes ces exigences, L. Paulson[121] a introduit un langage de *conversions* et de *conversionals*, dans le même esprit que celui des tactiques et tacticals. Ce langage consiste en un bibliothèque modulaire de fonctions ML pouvant être combinées de différentes manières, permettant ainsi à l'utilisateur de définir ses propres stratégies de réécriture (éventuellement très sophistiquées). Cette technique a été adoptée dans les systèmes HOL, Isabelle et Nuprl.
Dans [21], D. Basin a étendu dans Nuprl ce langage au cas de relations quelconques (dans LCF, il

n'était utilisé que pour deux égalités particulières). Toujours dans Nuprl, P. Jackson[88] a réécrit un nouvelle bibliothèque de réécriture en y incluant l'égalité définitionnelle. Cette dernière bibliothèque a été utilisée avec succès dans plusieurs preuves[88]. La bibliothèque que nous avons implémentée dans Coq est similaire à celui de [88]. Nous remédions ainsi à une lacune importante de Coq.

Nous commençons par donner un panorama des égalités fréquemment utilisées en théorie des types. Puis nous introduisons les notions de tactiques et tacticals, et enfin nous détaillons le bibliothèque de réécriture. Nous terminons en montrant comment utiliser cette bibliothèque pour le raisonnement équationnel.

6.2 Principales égalités

6.2.1 Égalité de Leibniz

L'égalité de Leibniz, appelée aussi égalité propositionnelle, est inductivement engendrée par l'axiome de réflexivité; cette définition est dûe à Ch. Paulin-Mohring[119]. On écrira $x = y$ au lieu de $(eq\ A\ x\ y)$, en rendant implicite le type commun A de x et y.

$$Inductive\ eq\ [A : Set; x : A]\ :\ A \to Prop\ :=\ refl\_equal\ :\ (eq\ A\ x\ x)$$

Son principe d'élimination est le suivant:

$$\begin{aligned} eq\_ind1 : &(A : Set)(x : A)(P : (y : A)x = y \to Prop) \\ &(P\ x\ (refl\_equal\ A\ x)) \to (y : A)(e : x = y)(P\ y\ e) \end{aligned}$$

Il peut être spécialisé au cas de prédicats non dépendants afin d'obtenir la règle de *substitutivité* *eq_ind* ci-dessous. C'est la propriété de Leibniz: si un prédicat P est vérifié en x, alors il l'est aussi en y si $x = y$.

$$\begin{aligned} eq\_ind := &[A : Set][x : A][P : A \to Prop][p : (P\ x)][y : A][e : x = y] \\ &(eq\_ind1\ A\ x\ ([y : A][e : x = y](P\ y))\ p\ y\ e) \\ : &\ (A : Set)(x : A)(P : A \to Prop)(P\ x) \to (y : A)x = y \to (P\ y) \end{aligned}$$

L'opérateur *eq_ind* nous permet de prouver plusieurs propriétés de eq. Commençons par prouver que eq est la plus petite relation réflexive. Soit E une relation réflexive quelconque définie dans le contexte:

$$...; A : Set;\ E : A \to A \to Prop;\ r : (x : A)(E\ x\ x); ...$$

On prouve que E contient eq:

$$[x, y : A][H : x = y](eq\_ind\ A\ [a : A](E\ x\ a)\ (r\ x)\ y\ H) : (x, y : A)x = y \to (E\ x\ y)$$

L'égalité eq est une relation d'équivalence. En effet:

- eq est réflexive par définition.

  $$Refl := refl\_equal : (A : Set)(x : A)x = x$$

- eq est symétrique.

$$\begin{aligned} Sym &:= [A:Set][x,y:A][e:x=y](eq\_ind\ A\ x\ ([a:A]a=x)\ (refl\_equal\ A\ x)\ y\ e) \\ &: (A:Set)(x,y:A)x=y \to y=x \end{aligned}$$

- eq est transitive.

$$\begin{aligned} Trans &:=[A:Set][x,y,z:A][e_1:x=y][e_2:y=z](eq\_ind\ A\ x\ ([a:A]a=x)\ e_2\ z\ e_1) \\ &: (A:Set)(x,y,z:A)x=y \to y=z \to x=z \end{aligned}$$

Une autre propriété remarquable est que toute fonction $f : A \to B$ préserve eq. Remarquez que dans le type de f_equal ci-dessous, $x = y$ est une abréviation de $(eq\ A\ x\ y)$, alors que $(f\ x) = (f\ y)$ est une abréviation de $(eq\ B\ (f\ x)\ (f\ y))$.

$$\begin{aligned} f\_equal := &[A,B:Set][f:A\to B][x,y:A][H:x=y] \\ &(eq\_ind\ A\ x\ ([a:A](f\ x)=(f\ a))\ (refl\_equal\ B\ (f\ x))\ y\ H) \\ : &(A,B:Set)(f:A\to B)(x,y:A)x=y\to(f\ x)=(f\ y) \end{aligned}$$

L'égalité de Leibniz est substitutive sans être extentionnelle; on ne peut pas substituer un égal par un égal sous une abstraction. On ne peut pas prouver l'axiome d'extentionnalité des fonctions:

$$(A,B:Set)(f,g:A\to B)((x:A)(f\ x)=(g\ x))\to f=g$$

L'égalité de Leibniz a cette propriété fondamentale qu'elle reflète l'égalité définitionnelle dans le sens que deux objets sont définitionnellement égaux si et seulement s'ils sont prouvablement égaux par l'égalité de Leibniz (dans le contexte vide). Cette propriété est évidemment fausse dans un contexte non vide, pouvant contenir des égalités ou bien encore des objets définis inductivement.

Lemme 6.1 (Réflexion de l'égalité) *Supposons que* $\vdash a : A$ *et* $\vdash b : A$. *On a* $a \cong b$ *si et seulement s'il existe* M *tel que* $\vdash M : a =_A b$.

PREUVE voir [103]. ■

Z. Luo[103] montre aussi que la prouvabilité de l'inégalité de Leibniz est incomplète par rapport à l'égalité définitionnelle, dans le sens qu'il existe un type A avec des objets a et b tels que $a \ncong b$ mais $\neg(a =_A b)$ est non prouvable.

6.2.2 Autres égalités

Tout d'abord, il existe la version de l'égalité de Leibniz sur la sorte $Type$, notée $==$. Ensuite, on peut généraliser l'égalité de Leibniz au cas de familles indexées de types. Nous nous contenterons de donner sa définition. Dans la définition de eq_dep, F est une famille de types indexée par A.

$$\begin{aligned} &Inductive\ eq\_dep[A:Set;F:A\to Set;i:A;x:(F\ i)]:(j:A)(F\ j)\to Prop \\ &\qquad := eq\_dep\_intro:(eq\_dep\ A\ F\ i\ x\ i\ x) \end{aligned}$$

Une présentation de ses propriétés est donnée dans [50].

D'autres relations sont aussi utilisées, comme par exemple $\leq$ entre les entiers, $\leftrightarrow$ entre les propositions, etc. On peut aussi définir une égalité par filtrage pour tout type de données défini inductivement. Enfin, on peut utiliser des égalités abstraites introduites comme axiomes dans le contexte. Nous verrons dans 7.3 une telle égalité.

6.3 Notions de réécriture (Rappels)

Pour fixer la terminologie, nous rappelons les notions de base de la théorie de la réécriture. Le lecteur trouvera un exposé plus complet de cette théorie dans [78]. Ces notions nous seront utiles pour la réécriture avec des notions catégoriques (8.10 et 9.3.4).

Définition 6.1 (Relation de réduction) *Soit A un ensemble et $\rightarrow \subseteq A \times A$ une relation binaire. Nous appelons $\rightarrow$ relation de réduction et notons $(a,b) \in \rightarrow$ par $a \rightarrow b$.*

- *L'élément $a \in A$ est en forme* normale *si il n'existe pas de $b \in A$ avec $a \rightarrow b$.*
- *La relation $\rightarrow^*$,* fermeture réflexive transitive *de $\rightarrow$, est la plus petite relation vérifiant:*

$$a \rightarrow a$$

$$a \rightarrow b \Rightarrow a \rightarrow^* b$$

$$a \rightarrow b \text{ et } b \rightarrow^* c \Rightarrow a \rightarrow^* c$$

- *La relation $\longleftrightarrow^*$,* fermeture réflexive symétrique transitive *de $\rightarrow$, est la fermeture réflexive transitive de $\longleftrightarrow$, définie par $a \longleftrightarrow b$ si et seulement si $a \rightarrow b$ ou $b \rightarrow a$.*
- *L'*inverse *de la relation $\rightarrow$ est une relation de réduction, notée par $\leftarrow$; elle est définie par $b \leftarrow a$ si et seulement si $a \rightarrow b$.*
- *Soient $\rightarrow_1$ et $\rightarrow_2$ deux relations de réduction sur A, Leur composition est une relation de réduction $\rightarrow_1 \circ \rightarrow_2$ telle que $a \rightarrow_1 \circ \rightarrow_2 c$ si et seulement s'il existe $b \in A$ tel que $a \rightarrow_1 b$ et $b \rightarrow_2 c$.*

Nous notons par $\equiv$ l'égalité syntaxique dans l'ensemble A.

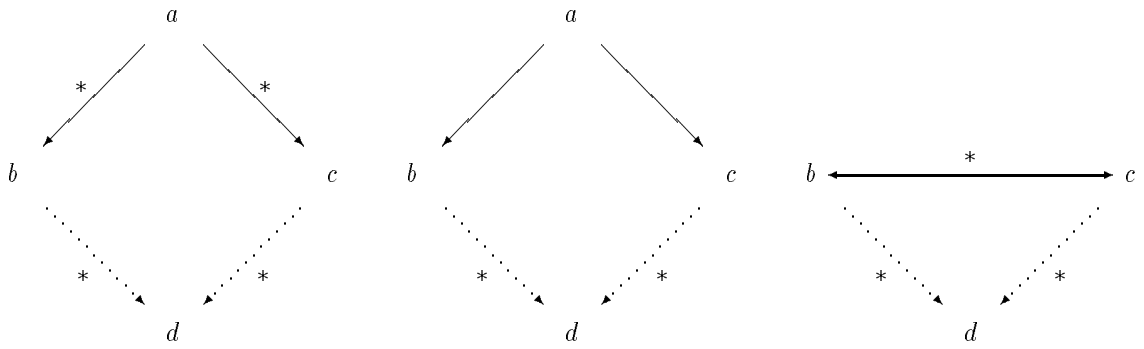
Définition 6.2 (Propriétés des relations de réduction) *Une relation de réduction $\rightarrow$ est:*

- confluente *si et seulement si* $\leftarrow^* \circ \rightarrow^* \subseteq \rightarrow^* \circ \leftarrow^*$,
- localement confluente *si et seulement si* $\leftarrow \circ \rightarrow \subseteq \rightarrow^* \circ \leftarrow^*$,
- Church Rosser $\longleftrightarrow^* \subseteq \rightarrow^* \circ \leftarrow^*$,
- nœthérienne *s'il n'existe pas de séquence infinie d'éléments de A telle que* $s_1 \rightarrow s_2 \rightarrow \ldots$.

Les relations nœthériennes et confluentes sont dites canoniques ou complètes.

La propriété de confluence signifie l'indépendance du résultat du "calcul" vis-à-vis du chemin de réduction suivi. Quant à la nœthérianité, elle nous garantit la terminaison des calculs quelle que soit la stratégie de réduction adoptée. Les relations nœthériennes sont aussi appelées fortement normalisables (notamment dans le contexte du λ-calcul), relations qui terminent (*terminating* en anglais) ou encore relations bien fondées.

Les trois premières propriétés s'expriment par les diagrammes suivants, où un trait plein indique une hypothèse ("si ...") et un trait pointillé une conclusion ("il existe ..."):



Le lemme ci-dessous nous donne immédiatement une procédure de décision, en effet $a \longleftrightarrow^* b$ si et seulement si $\overline{a} \equiv \overline{b}$.

Lemme 6.2 (existence et unicité de la forme normale) *Soit $\rightarrow$ une relation de réduction sur A. Si $\rightarrow$ est complète alors chaque élément $a \in A$ possède une forme normale unique $\overline{a}$.*

Signalons aussi que les notions de confluence et de Church Rosser sont équivalentes, alors que la confluence locale n'implique pas la confluence (voir le contre-exemple de [78]).
Nous donnons maintenant une condition nécessaire et suffisante pour qu'une relation localement confluente soit confluente. C'est un résultat important car la confluence locale est nettement plus facile à montrer que la confluence.

Lemme 6.3 (Lemme de Newman) *Si $\rightarrow$ est nœthérienne, alors elle est confluente si et seulement si elle est localement confluente.*

Définition 6.3 (Signature) *Une signature $\Sigma = (V, F, ar)$ est définie par la donnée d'un ensemble V infini dénombrable de variables, un ensemble F non vide d'opérateurs tel que $F \cap V = \emptyset$, et une fonction d'arité $ar : F \rightarrow N$.*

Définition 6.4 (Termes) *L'ensemble T_Σ des termes construits sur une signature $\Sigma = (V, F, ar)$ est le plus petit ensemble tel que $V \subseteq T_\Sigma$, et $f(t_1, ..., t_{ar(f)}) \in T_\Sigma$ pour tous $f \in F$ et $t_1, ..., t_{ar(f)} \in T_\Sigma$.*
Nous notons par $var(t)$ l'ensemble des variables d'un terme t.

Définition 6.5 (Substitution)

- *Soit $\Sigma = (V, F, ar)$ une signature. Une substitution sur Σ est une fonction $\sigma : V \rightarrow T_\Sigma$.*
  Le terme $\sigma(t) \in T_\Sigma$ est le résultat de la substitution de chaque variable, disons $v \in V$, de $t \in T_\Sigma$ par son image $\sigma(v)$.
  Nous noterons par $[v \leftarrow u]$ les substitutions ne modifiant qu'une seule variable v. Leur application à un terme t est notée $t[v \leftarrow u]$.
- *La composition de substitutions est définie par: $(\sigma_1 \circ \sigma_2)(v) = \sigma_2(\sigma_1(v))$.*
- *Un contexte est un terme avec un "trou". Le terme $C[t]$ dénote le contexte C avec le terme t à la place du trou. Le remplacement de cette occurrence de t par un terme u donne le terme $C[u]$.*

Définition 6.6 (Système de réécriture) *Soit $\Sigma = (V, F, ar)$ une signature. Une règle de réécriture est une paire ordonnée de termes $(s,t) \in T_\Sigma \times T_\Sigma$ telle que $s \notin V$ et $var(t) \subseteq var(s)$.*
Un système de réécriture (Σ, R) est spécifié par la donnée d'une signature et d'un ensemble de règles de réécriture sur Σ.
La paire $(s,t) \in R$ se note $s \rightarrow_R t$ ou plus simplement $s \rightarrow t$, s'il n'y a pas risque de confusion.

Définition 6.7 (Relation de réécriture) *Soit (Σ, R) un système de réécriture. La relation de réécriture, notée $\rightarrow_R$, est une relation de réduction sur T telle que $u \rightarrow_R v$ si et seulement s'il existe un contexte C, une substitution σ et une règle $s \rightarrow_R t$ tels que $C[\sigma(s)] \equiv u$ et $v \equiv C[\sigma(t)]$.*

Définition 6.8 *Un système de réécriture (Σ, R) est dit complet (ou canonique) si la relation de réécriture $\rightarrow_R$ est canonique.*

6.4 Activité de preuve

Nous nous intéressons maintenant à la manière de communiquer une preuve mathématique à un système de preuve. Il est clair que les preuves «informelles» qu'on trouve dans les ouvrages de mathématiques sont hors de portée des ordinateurs; il s'agit en effet d'un savant mélange de symboles formels et de langage naturel, considérant souvent comme acquise une certaine connaissance du domaine étudié. D'un autre coté, nous avons vu en section 2.5, que grâce à la correspondance de Curry-Howard-de Bruijn, les preuves formelles dans les logiques intuitionnistes correspondent à des termes de λ-calculs typés. Ces preuves «complètement» formelles sont pratiquement impossibles à lire et à écrire par un humain. Une troisième sorte de preuve, visant à combler le fossé entre les deux premières sortes de preuves précitées, est introduite dans LCF. Une preuve dans LCF est un *script* constitué d'une succession de commandes appelées *tactiques*. Chaque tactique est un programme indiquant au système de preuve comment construire une preuve formelle. Les scripts de tactiques ainsi obtenus sont «presque» aussi faciles à lire et à écrire que les preuves informelles.
De nombreux systèmes de preuves (Coq, LEGO, NuPRL etc.) ont adopté le style de preuve de LCF. Nous ferons référence à ces systèmes par «systèmes à la LCF».

6.4.1 Langage ML

Le langage ML (ou un de ses dérivés) est le langage d'implémentation des procédures de preuves (tactiques et tacticals). Il a d'ailleurs été développé dans ce but par Robin Milner en 1978. C'est un langage fonctionnel fortement typé, avec un système de type polymorphique et offrant la possibilité de définir de nouveaux types. Il constitue une implémentation du λ-calcul faible (pas de réduction sous une abstraction) où les fonctions sont des objets de première classe; on peut définir des fonctions (dites d'ordre supérieur) prenant des fonctions comme arguments ou les retournant comme résultat. Il contient aussi des traits impératifs, notamment pour les opérations d'entrée-sortie et les exceptions. Il n'est pas dans notre but de présenter le détail de ces caractéristiques; le lecteur est encouragé à consulter les nombreux ouvrages consacrés au sujet (notamment [110] pour SML et [99] pour CAML).

Dans un souci de clarté, nous répugnons à inclure des programmes ML dans notre exposé. La syntaxe que nous utilisons pour les expressions ML est approximative, et nous permet de faire abstraction de nombreux détails techniques d'implémentation, tout en restant très exacte sur le plan du principe général et des idées. Nous aurons besoin de très peu de constructions ML.

- $t_1 * t_2$ est le produit cartésien de types.
- $t\ list$ est le type des listes d'éléments de type t.
- $t_1 + t_2$ est l'union disjointe de type.
- $unit$ est le type unité à un seul habitant $\langle\rangle$.
- $\langle a_1, ..., a_n \rangle$ est un n-uplet.
- $type\ t = ...$ définit un type.
- $let\ x = ...$ définit une valeur, éventuellement une fonction, auquel cas ses paramètres sont à la suite du nom de la fonction $let\ f\ x_1 ... x_n = ...$
- $[a_1; ...; a_n]$ est la liste formée des valeurs a_1,...,a_n.

6.4.2 Tactiques

Le noyau des systèmes à la LCF consiste en l'implémentation des règles de la théorie des types implémentée, appelées *règles primitives*. Ces règles de preuves représentent de très petites étapes de preuves. Il est heureusement possible d'utiliser toute la puissance de ML pour écrire de nouvelles procédures de preuve appelées tactiques (*tactics*). Des programmes quelconques peuvent être écrits pour prouver un énoncé; la seule restriction est que ces programmes soient bien typés par ML. Il n'est pas nécessaire que les tactiques prouvent de manière uniforme un énoncé; la preuve peut dépendre de la forme du but et la tactique peut utiliser différents algorithmes. Les tactiques permettent la mise en œuvre de la technique de preuve par raffinement successif (i.e. chaînage arrière): le but est réduit en sous-buts plus simples, qui sont à leur tour réduits en sous-buts, jusqu'à ce que tous les sous-buts soient triviaux à démontrer. Ainsi une tactique est une fonction ML qui transforme un but en une liste de sous-buts et retourne une fonction de validation.

$type\ validation = proof\ list \rightarrow proof$
$type\ tactic = goal \rightarrow (goal\ list * validation)$

Une validation produit une preuve du but originel à partir des preuves de ses sous-buts. Le type *goal* correspond au type des buts; un but est exprimé sous forme d'un séquent, formé d'une conclusion et d'un contexte d'hypothèses.
Il est important de remarquer que les tactiques ne compromettent en rien ni la correction, ni la simplicité des systèmes à la LCF. En effet, la correction des systèmes à la LCF repose uniquement sur leur noyau. C'est le seul responsable de la construction de tous les terme-preuves. C'est lui qui interprète le résultat de toutes les tactiques; toute tactique est ultimement décomposée en une succession de règles primitives.

Les tactiques les plus simples correspondent à l'application des règles primitives.

- *Intro* implémente la règle (LAM). Appliquée à un but $\Gamma \vdash (x : U)T$, elle engendre un sous-but $\Gamma; x : U \vdash T$.

- *Apply t* implémente la règle (APP). Cette tactique essaie de filtrer la conclusion du but avec la conclusion du type du terme t. Si elle réussit, elle retournera autant de sous-buts que de premisses instanciées du type de t.

- les tactiques les plus sophistiquées correspondent à des procédures de décision permettant d'enchaîner plusieurs milliers de règles primitives. Beaucoup de tactiques évoluées sont basées sur des algorithmes de «recherche». La tactique se déroule alors en deux étapes. La première étape, généralement longue et complexe, est chargée de trouver une preuve (généralement courte). La deuxième étape traduit cette preuve en termes de tactiques déjà existantes.

Des améliorations sont apportées à l'approche purement LCF, par certains systèmes de preuves (notamment Coq et NuPRL). Ainsi on utilise des variables existentielles[56] permettant de différer des instantiations, comme dans le cas de la tactique *Apply*. Aussi, en plus du terme-preuve, un arbre de preuve représentant la dérivation est construit.

Une qualité importante des tactiques est leur modularité; des fonctions appelées *tacticals* permettent de combiner les tactiques existantes pour en former de nouvelles. Les plus usuelles sont *Then* et *Orelse*.

- tac_1 *Then* tac_2 applique tac_1, puis applique tac_2 à tous les sous-buts résultants.

- tac_1 *Orelse* tac_2 applique tac_1; si elle échoue alors elle applique tac_2.

6.5 Réécriture Générique

6.5.1 Principe Général

Notre bibliothèque de réécriture utilise des conversions de type ML *conv* comme suit.

$type\ conv = term \rightarrow term * ((rel * term\ list * tactic) + unit)$

où *rel* est le type ML des noms de relations. Une relation étant tout object de type $T \rightarrow T \rightarrow s$ où s est une sorte. Une relation peut avoir des paramètres, comme par exemple eq; son type est alors de la forme $(\vec{x} : \vec{A})T \rightarrow T \rightarrow s$. Le cas *unit* est réservé au cas de l'égalité définitionnelle.

La tactique principale est *Rewrite c t* où c est une conversion et t est un terme. L'application de cette tactique à un but G a pour effet d'appliquer c à t, donnant ainsi un triplet de la forme $\langle t', \langle R, \vec{p}, tac\rangle\rangle$ ou $\langle t', \langle\rangle\rangle$. Dans le premier cas, *tac* est une tactique devant prouver $(R\ \vec{p}\ t\ t')$. Cette dernière égalité est alors rajoutée au contexte local de G, et peut être ainsi utilisée pour prouver G. Dans le second cas, il s'agit de l'égalité définitionnelle $t \equiv t'$ (cette égalité n'a pas besoin de justification), elle est alors utilisée pour réécrire la conclusion du but G.

6.5.2 Conversions de base

Les conversions basiques sont construites à partir de lemmes ayant la forme $(\vec{x} : \vec{A})(R\ \vec{p}\ t_1\ t_2)$ où R est une relation. Dans la présentation habituelle des systèmes de réécriture (comme dans 6.3), les règles contiennent des variables qui sont implicitement universellement quantifiées. Dans le cas de nos lemmes, elles le sont explicitement.

La fonction ML $ruleC : lemma \rightarrow conv$ transforme un tel lemme en une conversion. Soit l un nom de lemme de la forme précédente, l'application de la conversion $ruleC(l)$ à un terme t se déroule comme suit:

1. t est filtré par t_1, donnant une substitution σ liant toutes les variables $\vec{x}$. Le filtrage peut bien entendu échouer.
2. le triplet $\langle \sigma(t_2), \langle R, \sigma(\vec{p}), tac \rangle \rangle$ où tac correspond à la tactique d'application du lemme l.

La fonction $ruleC$ a des paramètres supplémentaires: un paramètre pour indiquer le sens de la réécriture (de gauche à droite ou de droite à gauche), et éventuellement une liste de liaisons pour quelques variables de $\vec{x}$ pour effectuer l'opération de filtrage.

Les lemmes de réécriture conditionnelle sont aussi acceptés. Leur forme est $(\vec{x} : \vec{A})T_1 \rightarrow ... \rightarrow T_n \rightarrow (R\ \vec{p}\ t_1\ t_2)$. Le type complet de $ruleC$ est alors:

$ruleC : lemma \rightarrow dir \rightarrow (var, term)\ list \rightarrow tactic\ list \rightarrow conv$

où dir indique le sens de la réécriture et $(var, term)\ list$ est une liste de liaisons. On doit alors lui fournir une liste de tactiques pour prouver les instantiations $\sigma(T_i)$ des conditions T_i.

Des conversions de base spéciales correspondent aux différentes réductions de l'égalité définitionnelle. Ainsi par exemple $betaC : conv$ appliquée à un terme $([x : A]t\ u)$ donne $\langle t[x \leftarrow u], \langle\rangle\rangle$. On peut aussi définir des règles de réécriture associées à l'égalité définitionnelle grâce à la commande $Conv$. Elle possède trois arguments: un contexte C (une suite de déclarations de variables), et deux termes t_1 et t_2 dont toutes les variables sont dans C.

$Conv : context \rightarrow term \rightarrow term \rightarrow conv$

Si t_1 et t_2 sont convertibles, alors $(Conv\ C\ t_1\ t_2)$ est une conversion transformant tout terme t tel que $t = \sigma(t_1)$, en un terme $\sigma(t_2)$ qui lui est définitionnellement égal. Un exemple d'application est donné dans la section 8.10.4.

Deux autres conversions sont importantes:

- la conversion identité IdC. Elle transforme un terme t en $\langle t, \langle\rangle\rangle$.
- la conversion $FailC$ qui échoue toujours (utilise le mécanisme d'exceptions de ML).

6.5.3 Lemmes utilisés

Notre bibliothèque utilise une base de données d'informations concernant les relations. Cette base de données est étendue de manière dynamique par l'utilisateur qui peut y ajouter divers lemmes.

- Lemmes de transitivité. Ils sont de la forme:

  $(\vec{x} : \vec{A})(a, b, c : T)(R_1\ \vec{p}_1\ a\ b) \rightarrow (R_2\ \vec{p}_2\ b\ c) \rightarrow (R_3\ \vec{p}_3\ a\ c)$

- Lemmes de congruence. Un lemme de congruence par rapport à une fonction f a la forme:

$(\vec{x} : \vec{A})(\vec{a}, \vec{b} : \vec{T})(R_1\ \vec{p_1}\ a_1\ b_1) \to ... \to (R_n\ \vec{p_n}\ a_n\ b_n) \to (R\ \vec{p}\ (f\ \vec{q_1}\ \vec{a})\ (f\ \vec{q_2}\ \vec{b}))$

Des lemmes plus généraux sont aussi acceptés comme:

$(\vec{x} : \vec{A})(f : (\vec{y} : \vec{B})U \to V)(\vec{a}, \vec{b} : \vec{T})$
$(R_1\ \vec{p_1}\ a_1\ b_1) \to ... \to (R_n\ \vec{p_n}\ a_n\ b_n) \to (R\ \vec{p}\ (f\ \vec{q_1}\ \vec{a})\ (f\ \vec{q_2}\ \vec{b}))$

ou bien encore des lemmes de substitutivité:

$(\vec{x} : \vec{A})(a : T)(P : T \to s)(P\ a) \to (b : T)(R\ \vec{p}\ a\ b) \to (P\ b)$

- Lemme d'inclusion. Une relation R est inclue dans une relation R' (notation: $R \subset R'$) si:

  $(\vec{x} : \vec{A})(a, b : T)(R\ \vec{p}\ a\ b) \to (R'\ \vec{q}\ a\ b)$

  Ces lemmes sont utilisés dans un souci d'optimisation: on essaiera toujours de réécrire avec la plus petite relation possible, car c'est elle qui donne le plus d'information. Ainsi par exemple $a < b$ est plus informatif que $a \leq b$ ($< \subset \leq$).

Nous décrirons dans ce qui suit l'utilisation de chacun de ces lemmes.

6.5.4 Composition de conversions

Des combinateurs appelés *conversionals* permettent de combiner des conversions pour en former de nouvelles. Les principaux combinateurs sont $ThenC$, $SubC$ et $OrelseC$.

Pour simplifier, nous ne traiterons que le cas où toutes les conversions utilisées ci-dessous renvoient des tactiques. Les autres cas consistant à combiner des tactiques avec l'égalité définitionnelle sont faciles à traiter, et ne font que compliquer l'exposé.

6.5.4.1 ThenC

La conversion $c_1\ ThenC\ c_2$ compose séquentiellement les conversions c_1 et c_2. L'application de $c_1\ ThenC\ c_2$ à un terme t se déroule comme suit:

1. c_1 est appliqué à t, donnant un résultat $\langle t_1, \langle R_1, \vec{q_1}, tac_1 \rangle \rangle$.
2. c_1 est appliqué à t_1, donnant un résultat $\langle t_2, \langle R_2, \vec{q_2}, tac_2 \rangle \rangle$.
3. on cherche dans la base de données un lemme de transitivité de la forme $(\vec{x} : \vec{A})(a, b, c : T)(R'_1\ \vec{p_1}\ a\ b) \to (R'_2\ \vec{p_2}\ b\ c) \to (R_3\ \vec{p_3}\ a\ c)$ avec $R_1 \subset R'_1$, $R_2 \subset R'_2$ et R_3 la relation la plus petite possible. Les termes $\vec{q_1}$ et $\vec{q_2}$ doivent alors être filtrés par respectivement $\vec{p_1}$ et $\vec{p_2}$; soit σ la substitution obtenue.
4. le résultat global est $\langle t_2, \langle R_3, \sigma(\vec{p_3}), tac \rangle \rangle$ où tac_1, tac_2, le lemme de transitivité et les lemmes d'inclusion pour $R_1 \subset R'_1$ et $R_2 \subset R'_2$ sont combinés de manière appropriée en tac.

La conversion $c_1\ ThenC\ c_2$ échoue si l'une des conversions c_1 ou c_2 échoue, ou bien si aucun lemme de transitivité convenable n'est trouvé.

6.5.4.2 SubC

La conversion $SubC\ c$, appliquée à un terme $(f\ \vec{p}\ t_1...t_n)$, commence par appliquer c à chacun des sous-termes $t_1...t_n$ obtenant des triplets $\langle u_1, \langle R_1, \vec{p}_1, tac_1\rangle\rangle...\langle u_n, \langle R_n, \vec{p}_n, tac_n\rangle\rangle$. On cherche alors dans la base de données tous les lemmes de congruence pour f liant des relations R_1^i, ..., R_n^i à une relation R^i et vérifiant $R_1 \subset R_1'...R_n \subset R_n'$. Le lemme de congruence sélectionné est celui possédant la plus petite relation R^i.

6.5.4.3 OrelseC

La conversion $c_1\ OrelseC\ c_2$ appliquée à t correspond à $c_1\ t$ s'il réussit, sinon il applique c_2 à t. $c_1\ OrelseC\ c_2$ échoue si les deux conversions c_1 et c_2 échouent.

6.5.4.4 Autres combinateurs de conversions

Nous présentons les combinateurs les plus utilisés en pratique. Ces combinateurs sont complètement génériques dans le sens qu'ils sont indépendants de la relation utilisée pour réécrire.

- $TryC\ c$ essaie d'appliquer c. Si c échoue, $TryC\ c$ se comporte comme IdC.

  $let\ TryC\ c = c\ OrelseC\ IdC$

- $c_1\ OrThenC\ c_2$ est similaire à $c_1\ ThenC\ c_2$ sauf que $c_1\ OrThenC\ c_2$ n'échoue que si c_1 et c_2 échouent.

  $let\ c_1\ OrThenC\ c_2 = (c_1\ ThenC\ TryC\ c_2)\ OrelseC\ c_2$

- L'opérateur $FirstC$ est la version n-aire de $OrelseC$: $FirstC\ [c_1;...;c_n]$ retourne le résultat de la première conversion qui réussit.

- $ProgressC\ c$ se comporte comme c, mais échoue si le résultat de $c(t)$ est identique à t.

- Quant à $RepeatC\ c$, il applique c tant que la réécriture progresse.

  $let\ rec\ RepeatC\ c\ t = TryC\ (ProgressC\ c\ ThenC\ RepeatC\ c)\ t$

  $Repeat1C\ c$ échoue si la première application de c échoue.

  $Repeat1C\ c = c\ ThenC\ RepeatC$

- $(SweepDnC\ c\ t)$ commence par appliquer c à t puis applique $(SweepDnC\ c\ t)$ à ses sous-termes immédiats. Il réécrit ainsi t en le parcourant une seule fois de haut vers le bas.

  $let\ rec\ SweepDnC\ c\ t = (c\ OrThenC\ SubC\ (SweepDnC\ c))\ t$

- $SweepUpC$ est similaire à $SweepDnC$ mais procède du bas vers le haut.

  $let\ rec\ SweepUpC\ c\ t = (SubC\ (SweepUpC\ c)\ OrThenC\ c)\ t$

- $(ReTopC\ c\ t)$ applique c à tous les sous-termes de t en commençant par le haut. Cette opération est itérée jusqu'à échec. Il existe aussi une conversion similaire agissant du bas vers le haut.

$$let\ rec\ ReTopC\ c\ t = TryC\ (Repeat1C\ c\ OrThenC\ SubC\ (ReTopC\ c)\ OrThenC\ (c\ ThenC\ TryC\ (ReTopC\ c)))\ t$$

6.6 Raisonnement Équationnel

Le langage de conversions et conversionals peut être utilisé pour résoudre des équations. Soit un but à résoudre de la forme $\Gamma \vdash (R\ \vec{p}\ s\ t)$. Nous disposons des tactiques suivantes:

- $SimplEqL\ c$ qui réduit le but en $\Gamma \vdash (R\ \vec{p}\ s'\ t)$ où s' est le réduit de s par la conversion c. La tactique commence par réécrire s avec $Rewrite\ c\ s$. On obtient le but $\Gamma; (R'\ \vec{p'}\ s\ s') \vdash (R\ \vec{p}\ s\ t)$, puis un lemme de transitivité approprié.
- $SimplEqR\ c$ qui réduit le but en $\Gamma \vdash (R\ \vec{p}\ s\ t')$ où t' est le réduit de t par la conversion c. Elle nécessite un lemme de symmétrie pour R.
- $SimplEq\ c_1\ c_2$ combine les deux tactiques précédentes pour simplifier le but en $\Gamma \vdash (R\ \vec{p}\ s'\ t')$ où s' et t' sont respectivement les réduits de s et t par respectivement c_1 et c_2.

Toutes ces tactiques essaient de prouver le but en appliquant un lemme de réflexivité (s'il existe) pour R.

En pratique, on a souvent affaire à des systèmes complets de règles de réécriture, i.e. confluents et fortement normalisables. Ces propriétés nous assurent que tout terme a une forme normale unique et que toute stratégie de réécriture aboutit à cette forme normale. Soient r_1,..., r_n les règles d'un tel système sur un ensemble de termes T, et c_1,...,c_n les conversions correspondantes obtenues par $ruleC$ (les règles sont présentées sous forme de lemmes). La conversion c définie par $(ReTopC\ (FirstC\ [c_1; ..., c_n]))$ transforme tout terme en sa forme normale. Ainsi $SimplEq\ c\ c$ est une procédure de décision sur T. La conversion $NormalizeC$ normalise un terme relativement à une liste de conversions.

$$let\ NormalizeC\ l = (ReTopC\ (FirstC\ l))$$

6.7 Réflexion

L'approche LCF peut être inadéquate, car inefficace, dans certains cas comme dans des tâches de vérification (par exemple d'invariants) où les formules à prouver sont de taille très importante. Par contre, c'est loin d'être le cas pour la formalisation de mathématiques (qui nous intéresse tout particulièrement). Une alternative à l'approche LCF est l'utilisation du principe de *réflexion*. Dans une monographie[69], J. Harrison présente et compare les différentes approches basées sur la réflexion. L'utilisation de ce principe remonte aux travaux de Gödel qui a "codé" (représenté) les formules, les preuves et des opérations syntaxiques comme la substitution et la prouvabilité comme des objets (des entiers) de la logique (contenant l'arithmétique), et sur lesquels il peut

dorénavant raisonner. Dans les systèmes de preuves, la réflexion est utilisée pour le développement de procédures de décision efficaces pour des classes de problèmes particulières. Dans ce cas, il s'agit de coder la procédure de décision elle-même comme une fonction récursive du système, et de prouver sa correction, toujours dans le système. Ce théorème de correction, en faisant appel à la procédure de décision, constitue alors un moyen de construire des preuves plus concises pour résoudre un problème. L'objectif de la réflexion n'est donc pas de rendre un système formel plus puissant, mais plutôt de rendre plus efficace le processus de preuve en évitant de construire complétement certaines preuve. Les exemples les plus populaires d'utilisation de la réflexion concernent le raisonnement équationnel. Par exemple, si on veut justifier l'égalité $(a_1 + b_1) + ... + (a_n + b_n) = (a_1 + ... + a_n) + (b_1 + ... + b_n)$ dans une structure de groupe, nous pouvons directement utiliser la procédure de décision sur les groupes. Dans la méthode LCF, il faut réécrire délicatement l'égalité en utilisant les axiomes (notamment de commutatitivité et d'associativité). Un cas intéressant de la réflexion est celui de la *réflexion calculatoire* où un programme est extrait de la preuve de correction, et est ajouté dans l'implémentation du système sans justification (puisque la procédure a été prouvée correcte!).

Le principe de réflexion a été intensivement étudié, mais il existe peu de réalisations pratiques l'exploitant. Nous devons toutefois citer une application[27] récente dans Coq au cas des anneaux abéliens où les résultats sont très satisfaisants (du point de vue de la taille des preuves et du temps de leur production). Une généralisation de la méthode au cas des systèmes de réécriture canoniques y est aussi présentée. Le principal défaut de cette méthode est son manque de flexibilité. En effet, il faut refaire tout le codage et les preuves pour chaque nouveau système de réécriture ou pour prendre en compte une nouvelle équation en hypothèse.

Troisième partie

Application: Formalisation de la Théorie des Catégories

Introduction

La théorie des catégories est un langage mathématique dont l'origine remonte aux études de topologie algébrique de Eilenberg et Mac Lane en 1942. Cependant très vite, elle est devenue une discipline à part entière. La raison est qu'elle fournit un langage mathématique unifié et économique. Elle a emprunté à d'autres disciplines mathématiques des notions essentielles qu'elle a su généraliser. Grâce à sa généralité, le langage des catégories permet de transporter (via un «foncteur») des problèmes d'un domaine des mathématiques vers un autre où les solutions sont plus faciles à trouver. Toutefois cette généralité lui a valu auprès de certains mathématiciens l'appellation de « Abstract Nonsense ».
La théorie des catégories est la théorie de la composition typée de morphismes, elle est de ce point de vue une théorie très pauvre. Elle peut par contre être considérée de différents points de vue. Elle peut par exemple être comprise comme une théorie abstraite des fonctions, une théorie abstraite de structures et d'applications préservant ces structures, ou bien encore comme une théorie abstraite de déduction ($f : a \to b$ est alors interprété comme la preuve du séquent $a \vdash b$). Elle puise sa force dans de puissants et subtils mécanismes de description que sont les limites (et colimites) et les adjonctions. La structure d'objets mathématiques est étudiée en montrant comment ces objets sont liés entre eux. Cette description fait abstraction de la structure interne des objets, rompant ainsi totalement avec le mode usuel des mathématiques qui consiste justement à s'appuyer sur la connaissance de la structure interne des objets. C'est en ce sens que les constructions catégoriques sont abstraites: une même construction s'applique à différents objets mathématiques, plus exactement à tous les objets mathématiques pouvant être considérés comme «objets» d'une catégorie.

La théorie des catégories a été appliquée avec succès dans la formulation et la résolution de problèmes en topologie, algèbre, géométrie et analyse fonctionnelle. De plus depuis les travaux pionniers de Lawvere, dans les années 60, plusieurs travaux de recherche visant à imposer la théorie des catégories comme fondement de toutes les mathématiques, ont vu le jour.

Plus récemment, l'informatique a découvert la théorie des catégories. Elle a rapidement trouvé des applications en sémantique algébrique, en théorie des langages de programmation (notamment fonctionnels), puis plus récemment en théorie des automates. Elle joue notamment un rôle important dans l'étude de la sémantique des calculs logiques et des théories des types[128]. Elle a des connexions très fortes avec une autre théorie des fonctions, le λ-calcul. En effet il y a une correspondance entre les catégories cartésiennes closes[85] et le λ-calcul. Cette correspondance fournit une traduction des langages fonctionnels avec variables en un langage de combinateurs catégoriques[53] sans variables, qui est à la base de la machine abstraite utilisée pour l'implémentation de certains langages fonctionnels.

Nous avons choisi comme application la formalisation de la théorie des catégories dans Coq.

Notre but a été de concevoir une bibliothèque de notions et de résultats suffisamment riche pour permettre son utilisation dans d'autres développements de divers domaines. Nous avons fait le choix de n'exposer qu'une partie de cette bibliothèque; nous présentons dans la section 9.5 la liste des autres notions formalisées et résultats prouvés. Cette partie contient toutefois la formalisation des principales constructions catégoriques de base. Pour valider cette formalisation, nous présentons la preuve du théorème de Freyd[61] pour l'existence d'un adjoint gauche.
Notre principale référence pour la théorie des catégories est [106]. Nous avons aussi consulté d'autres ouvrages ([11, 137, 52, 127, 129, 15]) très intéressants.

L'objectif de cette présentation est double:

1. montrer l'adéquation de la théorie des types pour la formalisation de théories mathématiques abstraites. Nous faisons notamment grand usage des types dépendants (exemple: voir la définition de la composition dans une catégorie en 8.1) et de la hiérarchie d'univers (voir par exemple les définitions de SET (8.2) et CAT (8.7)). Nous confirmons par ailleurs l'importance des enregistrements dépendants pour la représentation des structures mathématiques.
   Nous montrons aussi que notre formalisation permet non seulement une représentation fidèle des notions catégoriques, mais aussi des techniques habituellement utilisées en théorie des catégories, comme la définition par universalité (9.1.1) et la définition par dualité (9.1.2).

2. mettre en pratique les outils vues dans les chapitres précédents: univers flottants, sous-termes implicites, coercions implicites et tactiques de réécriture. Pour chacun de ces outils, nous discutons un exemple mettant en évidence son importance dans la faisabilité de notre développement.

Une grande partie de notre présentation est consacrée à la discussion des difficultés rencontrées. Ces difficultés sont de différents ordres:

- problèmes de typage, dûs essentiellement au manque d'extentionnalité de l'égalité définitionnelle.
- incohérence d'univers, notamment dans le cas de CAT (section 8.7).
- certains comportements «bizarres» dûs à des définitions inductives avec types dépendants.

Cette partie est organisée en trois chapitres. Le premier définit la syntaxe de Coq et le langage des Setoïdes qui sous-tend notre formalisation. Le second est consacré à la définition des notions de catégorie, foncteur et transformation naturelle, et à la construction de quelques exemples de catégories. En fin de chapitre sont décrites les techniques de preuve mises en œuvre pour la preuve d'égalités de morphismes. Quant au dernier chapitre, il concerne la définition des notions de limites, adjonctions et à la construction de la preuve du théorème de Freyd. Nous terminons cette partie par un chapitre de conclusions où nous présentons notamment les formalisations réalisées dans d'autres systèmes de preuves.

Avertissement. Notre point de départ a été le développement réalisé par Peter Aczel dans LEGO en 1993, puis adapté à Coq par Gérard Huet. Ce développement comprenait essentiellement la définition de catégorie et la construction de la catégorie des ensembles (Setoïdes). La première partie de notre développement a été réalisé en collaboration avec G. Huet.

Chapitre 7

Préliminaires

Nous présentons dans ce chapitre toutes les notions préliminaires à la définition de la notion de catégorie. Nous commençons par décrire la syntaxe concrète des termes et des commandes de Coq que nous utilisons tout au long de cette partie.
La seconde partie est consacrée à la définition du langage des Setoïdes sur lequel notre formalisation est basée. Ce langage est maintenant largement utilisé pour la formalisation de l'algèbre dans la théorie des types, notamment par l'équipe de Peter Aczel[1, 13, 17] dans LEGO. Il est aussi utilisé par M. Hofmann[75] pour traduire des théories de types étendues (notamment par des types quotients) vers la théorie des types originale en interprétant un type comme un Setoïde. Enfin, il est utilisé comme langage de spécification algébrique par Luo[103].

7.1 Syntaxe Coq

Nous ne donnons ici que la partie de la syntaxe de Coq nécessaire à la compréhension du développement que nous présentons dans cette partie.

7.1.1 Syntaxe des termes

Les termes et les types de Coq sont décrits par une seule grammaire, leur syntaxe concrète est similaire à celle utilisée dans les précédents chapitres.

T	::=	*sorte*	correspond aux symboles `Prop` et `Type`
	|	*nom*	constante, variable, nom ou constructeurs de types inductifs
	|	`?`	terme implicite
	|	`[`*nom*`:`T`]` T	abstraction
	|	`[`*nom*`]` T	abstraction avec type implicite
	|	`[`*nom*`=`T`]` T	définition locale
	|	`(`*nom*`:`T`)` T	produit dépendant
	|	T`->`T	type fonctionnel ou implication logique
	|	`(`$T^{\#}$ T^{*}...T^{*}`)`	application
	|	T`::`T	cast

Nous disposons aussi d'une syntaxe particulière pour les abstractions et les produits imbriqués.

- `[`nom_1`,...,`nom_n`:`T`]` U est interprété comme `[`nom_1`:`T`]...[`nom_n`:`T`]`U.
- `(`nom_1`,...,`nom_n`:`T`)` U est interprété comme `(`nom_1`:`T`)...(`nom_n`:`T`)`U.

Les conventions concernant les arguments implicites sont celles décrites dans la section 4.6. La grammaire $T^{\#}$ définit la forme des termes en position fonctionnelle; `!`nom correspond à la version explicite de nom.

$T^{\#}$ ::= `!`nom | T

Quant à T^*, elle permet de donner explicitement certains arguments implicites d'une application par la syntaxe n`!`T où n est la position de cet argument.

T^* ::= n`!`T | T

Nous avons enfin les termes construits par filtrage d'un terme M, objet d'un type inductif. À chaque filtre F_i est associé un résultat (terme) U_i. Le terme T constitue le type du résultat; les dépendances de ce type avec M sont indiquées par des abstractions (vous trouverez plus de détails dans [49]). Il est optionnel; en effet il peut être automatiquement synthétisé dans le cas où tous les U_i sont de même type. Signalons enfin que ces constructions peuvent être imbriquées. La forme générale de `Cases` supporte plusieurs filtrages simultanés; sa description est disponible dans [51].

`<` T `>` `Cases` t `of` F_1 `=>` U_1 `|...|` F_n `=>` U_n `end`

7.1.2 Commandes

Les deux principales activités dans Coq sont la définition et la démonstration. Nous disposons aussi de commandes pour déclarer des coercions et poser des axiomes.

Déclaration

La déclaration d'un axiome consiste à introduire une nouvelle variable dans le contexte. On doit alors fournir son nom et son type. La syntaxe est:

`Axiom` nom `:` T`.`

Définition

Nous distinguons trois sortes de définitions.

- Définition d'une constante:

  `Definition` nom `:=` T`.`

  Où T est la valeur de la constante nom. On utilisera aussi la commande ci-dessous lorsque cette valeur est de la forme T_1`::`T_2.

  `Definition` nom `:` T_2 `:=` T_1`.`

- Définition d'un type enregistrement:

  $$\texttt{Structure}\ nom\ \texttt{[}p_1\texttt{:}T_1\texttt{;...;}\ p_n\texttt{:}T_n\texttt{]}\ \texttt{:}\ sorte\ \texttt{:=}\ \texttt{\{}ch_1\texttt{:}U_1\texttt{;...;}ch_m\texttt{:}U_m\texttt{\}}\texttt{.}$$

  La commande ci-dessus définit l'enregistrement nom avec n paramètres et m champs.

- Définition d'un type inductif:

  $$\texttt{Inductive}\ nom\ \texttt{[}p_1\texttt{:}T_1\texttt{;...;}\ p_n\texttt{:}T_n\texttt{]}\ \texttt{:}\ T\ \texttt{:=}\ c_1\texttt{:}U_1\texttt{|...|}c_m\texttt{:}U_m\texttt{.}$$

  Cette commande correspond à la définition d'un type inductif de nom nom ayant n paramètres et les constructeurs $c_1...c_m$.

Démonstration

Le but à démontrer est introduit par la commande ci-dessous où nom est le nom du lemme et T son type (la proposition à démontrer). Le mot clé `Theorem` est un synonyme de `Lemma`.

$$\texttt{Lemma}\ nom\ \texttt{:}\ T\texttt{.}$$

La preuve, construite de manière interactive en utilisant les tactiques, consiste en un terme habitant le terme T. Ainsi la démonstration est un cas particulier de l'activité de définition où la valeur de la constante n'est pas donnée à l'avance, mais construite construite par les commandes (tactiques) qui suivent l'énoncé à démontrer.

Coercions

La déclaration d'une coercion se fait par la donnée de son nom (préalablement déclaré ou défini) et de ses classes de départ et d'arrivée.

$$\texttt{Coercion}\ nom\ \texttt{:}\ C_1\ \texttt{>->}\ C_2\texttt{.}$$

La syntaxe de la déclaration d'une coercion identité est

$$\texttt{Identity Coercion}\ nom\ \texttt{:}\ C_1\ \texttt{>->}\ C_2\texttt{.}$$

Cette déclaration englobe les deux opérations décrites dans 5.4.3. Elle commence par vérifier que C_1 est défini en termes de C_2 par $C_1 := [p_1 : T_1]...[p_n : T_n](C_2\ u_1...u_m)$. Elle définit alors la constante nom comme une fonction identité, et la déclare comme coercion.

Nous utiliserons aussi les abréviations ci-dessous pour combiner les opérations de définition et de déclaration de coercions. Dans la dernière commande, la coercion identité, engendrée automatiquement, a pour nom `Id_`C_1`_`C_2.

$$\texttt{Coercion}\ nom\ \texttt{:=}\ T\ \texttt{.}$$
$$\texttt{Coercion}\ nom\ \texttt{:}T_2\ \texttt{:=}\ T_1\ \texttt{.}$$
$$\texttt{SubClass}\ C_1\ \texttt{:=}\ \texttt{[}p_1\texttt{:}T_1\texttt{]...}\ \texttt{[}p_n\texttt{:}T_n\texttt{](}C_2\ u_1...u_m\texttt{)}\texttt{.}$$

Nous utiliserons aussi très souvent l'abréviation suivante:

$$\texttt{Structure}\ \texttt{>}nom\ \texttt{[}p_1\texttt{:}T_1\texttt{;...;}\ p_n\texttt{:}T_n\texttt{]}\ \texttt{:}\ sorte\ \texttt{:=}\ \texttt{\{...;}ch_i\texttt{:>}\ U_i\texttt{;...\}}\texttt{.}$$

La syntaxe `>`nom indique que le constructeur `Build_`nom est déclaré comme coercion entre $Cl(U_m)$ (le dernier champ de l'enregistrement) et nom. Quant au symbole `:>`, il marque les projections déclarées comme coercions.

Divers

- Coq permet de définir une notation infixe pour les noms ayant deux arguments non implicites. Dans la syntaxe donnée ci-dessous, n indique le niveau de précédence pour l'utilisation de parenthèses dans le cas de notations infixes imbriquées. La chaîne de caractères *op* est le symbole infixe utilisé. Les opérateurs ainsi définis sont associatifs à gauche.

  `Infix` n "*op*" nom .

- Comme vu dans la section 4.5, l'unification prend en compte certaines définitions d'objets de type enregistrement pour synthétiser certaines variables existentielles. Ces définitions sont signalées à l'algorithme d'unification en les précédant par le symbole `@`.

7.1.3 Mécanisme des sections

Les développements effectués dans Coq sont souvent organisés en sections. Le nôtre n'échappe pas à cette règle. Une section constitue un contexte de déclarations de variables, de définitions et de démonstrations. Les variables sont déclarées par :

`Variable` nom : T . ou `Hypothesis` nom : T .

La commande `Hypothesis` est souvent réservée pour l'introduction de variables de type propositionnel. Il est aussi possible de déclarer plusieurs variables à la fois, même de types différents :

`Variables` $nom_{1,1}$,...,nom_{1,r_1} : T_1 ; ...;$nom_{n,1}$,...,nom_{n,r_n} : T_n .

Le contexte d'une section est délimité par les commandes « `Section` nom_sec . » indiquant le début de la section nom_sec et « `End` nom_sec . » indiquant sa fin. Le nommage des sections permet leur imbrication. Certaines constantes peuvent être définies localement à la section; elles se comportent comme des macros, elles n'existent plus à la sortie de la section. Leur syntaxe est:

`Local` nom `:=` T . ou `Local` nom : T_2 `:=` T_1 .

À la fermeture d'une section, les définitions et les lemmes contenus dans cette section sont abstraits par rapport aux variables dont ils dépendent. On dit aussi que les variables sont *déchargées*. Un axiome peut alors être perçu comme une variable permanente, jamais déchargée. Un exemple illustrant le fonctionnement du mécanisme des sections se trouve dans la section suivante.

7.2 Relations

Nous commençons par quelques notions usuelles concernant les relations binaires. Une relation sur un type U est représentée par son prédicat caractéristique, c'est-à-dire un terme de type $U \rightarrow U \rightarrow Prop$. Nous nous contentons de la définition de la notion de relation d'équivalence (réflexive, symétrique et transitive).

```
Section Orderings.
Variable U:Type.
Definition Relation :=  U -> U -> Prop.
Variable R:Relation.
Definition Reflexive := (x: U) (R x x).
Definition Transitive := (x,y,z: U) (R x y) -> (R y z) -> (R x z).
Definition Symmetric := (x,y: U) (R x y) -> (R y x).
Structure Partial_equivalence : Prop :=
 {Prf_trans : Transitive;
  Prf_sym   : Symmetric}.
Structure Equivalence : Prop :=
 {Prf_refl   :  Reflexive;
  Prf_pequiv :> Partial_equivalence}.
End Orderings.
```

À la sortie de la section, la valeur et le type de `Reflexive` deviennent:

```
[U:Type][R:(Relation U)](x: U) (R x x) : (U:Type)(Relation U) -> Prop
```

7.3 Ensembles

Le choix de la représentation de la notion d'ensemble est essentiel pour tout développement mathématique. Informellement, un ensemble est une collection d'objets. Dans les théories des ensembles (comme ZF), ces objets sont eux-même des ensembles, et des règles précises (axiomes) régissent la construction des ensembles afin d'éviter les paradoxes. Des formalisations de théories des ensembles (exemples, [7] et [156]) ont été réalisées dans des systèmes de preuves basés sur la théorie des types. Il est possible de prendre ces formalisations comme point de départ pour développer des théories mathématiques.

En pratique, les types sont considérés comme des représentations adéquates des ensembles; c'est d'ailleurs notre approche pour la définition des relations. Un sous-ensemble d'un «ensemble/type» U est alors tout simplement son prédicat caractéristique, de type $U \rightarrow Prop$. La relation d'appartenance est immédiate; soit $x : U$ et A un sous-ensemble de U, $x \in A$ si et seulement si $(A\ x)$. L'égalité de Leibniz est utilisée pour tester l'égalité de deux objets d'un même ensemble.

Cette approche s'avère trop restrictive, car l'égalité de Leibniz n'est pas toujours adéquate. Par exemple, dans le cas où les objets sont eux-mêmes des ensembles, deux ensembles sont égaux si et seulement s'ils contiennent les mêmes éléments, mais ne sont pas forcément identifiables par l'égalité de Leibniz. Un cas analogue est celui des ensembles de fonctions où l'égalité convenable est l'égalité extensionnelle des fonctions. Un autre exemple est celui de l'ensemble $\mathbb{Z}$ des entiers relatifs représentés comme paires d'entiers naturels. Une infinité de paires distinctes (au sens de l'égalité de Leibniz) représentent le même entier relatif.

En conclusion, l'égalité à utiliser depend de la nature des objets considérés. Il est alors naturel de paramétrer l'égalité associée à un type; cette égalité consiste en une relation d'équivalence sur ce type. Nous arrivons ainsi au developpement des «Setoïdes». Le langage des Setoïdes peut être considéré comme intermédiaire entre les deux notions de types et d'ensembles. En effet tout ensemble est muni d'une égalité extensionnelle, alors qu'un type est une collection non munie d'égalité autre que l'égalité définitionnelle. Un Setoïde, quant à lui, dispose d'une égalité, mais pas forcément extensionnelle.

Un Setoïde est un triplet composé d'un Type, d'une relation sur ce type et d'une preuve que cette relation est une relation d'équivalence. Un Setoïde est donc un ensemble considéré comme le quotient d'un Type par une congruence. Les Setoïdes ont d'abord été étudiés par M. Hofmann dans le cadre de la théorie des types de Martin-Löf[75].

```
Structure >Setoid : Type :=
 {Carrier   :> Type;
  Equal     :  (Relation Carrier);
  Prf_equiv :> (Equivalence Equal)}.
```

Pour avoir une notation plus agréable, nous introduisons une règle de grammaire permettant d'analyser `(Carrier A)` comme `|A|`. Dans les présentations informelle et formelle (grâce aux coercions), nous confondrons souvent le Setoïde avec son type sous-jacent.

Nous noterons par `=_S` l'égalité des Setoïdes. Il s'agit d'une égalité générique, puisque le type de ses éléments peut être en général déduit du contexte.

```
Infix 2 "=_S" Equal.
```

Cette approche est en fait un raffinement de celle de ensemble/type. Nous allons nous inspirer de la définition de sous-ensemble pour définir les sous-Setoïdes.Un sous-Setoïde d'un Setoïde U est tout prédicat $A : U \rightarrow Prop$ respectant son égalité: si $x \in A$ alors $y \in A$ pour tout y avec $x = y$. Là aussi $a \in A$ s'entend comme $(A\ a)$.

```
Section sub_setoid.
Variables U:Setoid.
Definition Reg_law := [A:U->Prop](x,y:U) x =_S y -> (A x) -> (A y).
Structure >Setoid_pred : Type :=
 {Pred    :> U -> Prop;
  Prf_reg :> (Reg_law Pred)}.
```

Un sous-Setoïde A donne lieu à un Setoïde dont les éléments consistent en des objets de U avec une preuve qu'ils sont dans A; l'égalité étant héritée de celle de U.

```
Variable A:Setoid_pred.
Structure SubType : Type :=
 {Elt_sub    : U;
  Prf_constr : (A Elt_sub)}.
Definition Equal_SubType := [a,b:SubType](Elt_sub a) =_S (Elt_sub b).
Lemma Equal_SubType_equiv : (Equivalence Equal_SubType).
@Definition SubSetoid : Setoid := Equal_SubType_equiv.
End sub_setoid.
```

Alternativement, nous pourrions construire des Setoïdes Partiels, où la relation d'équivalence est remplacée par une relation d'équivalence partielle. Il est cependant montré dans [17] qu'ils ne constituent pas une représentation fidèle et adéquate des ensembles.

7.4 Applications entre deux Setoïdes

Nous définissons maintenant une application entre le Setoïde `A` et le Setoïde `B` comme une fonction de `|A|` vers `|B|` qui préserve l'égalité.

```
Section maps.
Variables A,B:Setoid.
Definition Map_law := [f:A->B](x,y:A) x =_S y -> (f x) =_S (f y).
Structure >Map : Type := Build_Map
 {Ap   :> A->B;
  Pres :> (Map_law Ap)}.
```

Une application `f` de `A` vers `B` est alors similaire à une paire, comprenant une fonction `(Ap f)` (de type `|A|->|B|`) et une preuve `(Pres f)` que cette fonction préserve l'égalité.

Deux applications `f` et `g` sont dites égales si et seulement si elles sont extensionnellement égales, c'est-à-dire $\forall x.f(x) = g(x)$. Nous noterons cette égalité par `=_M`. Nous avons ainsi tous les ingrédients pour définir le Setoïde des applications entre deux Setoïdes .

```
Definition Ext := [f,g:Map](x:A)(f x) =_S (g x).
Lemma Ext_equiv : (Equivalence Ext).
@Definition Map_setoid : Setoid := Ext_equiv.
End maps.
Infix 2 "=_M" Ext.
Infix Assoc 6 "==>" Map_setoid.
```

Cette dernière commande nous permet d'écrire `A==>B` pour le Setoïde des applications entre les Setoïdes `A` et `B`.

Nous terminons cette section par la définition de l'application binaire (curryfiée) de Setoïdes. Une application binaire entre les Setoïdes A, B et C est une application entre A et $B \Rightarrow C$. Nous montrons ci-dessous comment construire une telle application à partir d'une fonction à deux arguments vérifiant les lois de congruence. La définition usuelle du produit est la version non-curryfiée, consistant en une application entre le Setoïde produit $A \times B$ et C.

```
Section fun2_to_map2.
Variables A,B,C:Setoid.
Definition Map2 := (Map A B==>C).
Variable f:A -> B -> C.
Definition Map2_congl_law := (b1,b2:B)(a:A)
                             (b1 =_S b2) -> (f a b1) =_S (f a b2).
Definition Map2_congr_law := (a1,a2:A)(b:B)
                             (a1 =_S a2) -> (f a1 b) =_S (f a2 b).
Definition Map2_cong_law := (a1,a2:A)(b1,b2:B)
           (a1 =_S a2) -> (b1 =_S b2) -> (f a1 b1) =_S (f a2 b2).
Hypothesis pgcl : Map2_congl_law.
Hypothesis pgcr : Map2_congr_law.
Lemma Map2_map_law1 : (a:A)(Map_law (f a)).
@Definition Map2_map1 := [a:A](Build_Map (Map2_map_law1 a)).
Lemma Map2_map_law2 : (Map_law Map2_map1).
Definition Build_Map2 : Map2 := (Build_Map Map2_map_law2).
End fun2_to_map2.
```

Nous définissons finalement un opérateur d'application pour les applications binaires qui nous sera utile plus tard.

```
Coercion Ap2 := [A,B,C:Setoid][f:(Map2 A B C)][a:A][b:B]((f a) b).
Identity Coercion Map2_Map : Map2 >-> Map.
```

Nous comparons dans le tableau ci-dessous le langage informel avec les notations implicites obtenues par l'utilisation des coercions.

Notation Usuelle	Langage Implicite	Langage Explicite
$x \in A$	`x:A`	`x:(Carrier A)`
$A \to B$	`A -> B`	`(Carrier A) -> (Carrier B)`
$f(x)$ avec $f : A \to B$	`(f x)` avec `f:(Map A B)`	`(Ap A B f x)` avec `f:(Map A B)`
$f(x, y)$ avec $f : A \to B \to C$	`(f x y)` avec `f:(Map2 A B C)`	`(Ap2 A B C f x y)` avec `f:(Map2 A B C)`

Chapitre 8

Catégories, Foncteurs et Transformations naturelles

Dans ce chapitre, nous définissons les trois notions de base de la théorie des catégories que sont les catégories, les foncteurs et les transformations naturelles. Certaines catégories intéressantes sont construites comme la catégorie des ensembles SET, la catégorie des catégories CAT et la catégorie des foncteurs. La dernière partie du chapitre est consacrée à l'utilisation des techniques de réécriture pour la preuve d'égalités de morphismes.

8.1 Catégories

La notion de fonction est l'une des notions les plus importantes en mathématiques. En manipulant des fonctions, nous devons considérer deux entités différentes : les ensembles et les fonctions entre ces ensembles. Une catégorie a une structure abstraite : une collection d'objets et une collection de morphismes. L'opération principale entre ses morphismes étant la composition.

Une catégorie comprend une collection d'objets, et pour toute paire d'objets a et b, une collection de flèches (appelées morphismes) notée $Hom(a,b)$, appelée *hom-set* (nous l'appelons dans notre cas hom-Setoïde). Nous écrirons $f : a \to b$ (`f:a-->b` sur machine) au lieu de $f \in Hom(a,b)$. Les objets a et b sont respectivement appelés domaine et codomaine de f.

```
Section cat.
Variables Ob:Type; Hom:Ob -> Ob -> Setoid.
Infix 6 "-->" Hom.
```

La prochaine composante d'une catégorie est une opération de composition assignant à toute paire de morphismes $f : a \to b$ et $g : b \to c$, un morphisme noté $f \circ g : a \to c$; nous adoptons une notation contraire à l'usage le plus répandu qui est $g \circ f$. Cette opération satisfait la loi d'associativité: pour tous morphismes $f : a \to b$, $g : b \to c$ et $h : c \to d$,

$$f \circ (g \circ h) = (f \circ g) \circ h$$

Nous écrivons cet opérateur en `o` infixe avec les objets a, b et c implicites.

```
Variable Op_comp:(a,b,c:Ob)(Map2 a-->b b-->c a-->c).
Definition Cat_comp := [a,b,c:Ob][f:a-->b][g:b-->c]((Op_comp a b c) f g).
Infix 6 "o" Cat_comp.
Definition Assoc_law := (a,b,c,d:Ob)(f:a-->b)(g:b-->c)(h:c-->d)
                        (f o (g o h)) =_S ((f o g) o h).
```

La dernière composante d'une catégorie est, pour tout objet a, un morphisme identité $Id_a : a \to a$, qui est l'identité pour la composition. Plus explicitement, les deux équations suivantes doivent être satisfaites pour tout morphisme $f : a \to b$,

$$Id_a \circ f = f \quad \text{et} \quad f = f \circ Id_b$$

```
Variable Id:(a:Ob)a-->a.
Definition Idl_law := (a,b:Ob)(f:a-->b)((Id ?) o f) =_S f.
Definition Idr_law := (a,b:Ob)(f:b-->a)f =_S (f o (Id ?)).
End cat.
```

Nous sommes maintenant en mesure de donner la définition synthétique de la notion de Catégorie .

```
Structure Category : Type :=
 {Ob      :> Type;
  Hom     :  Ob -> Ob -> Setoid;
  Op_comp :  (a,b,c:Ob)(Map2 (Hom a b) (Hom b c) (Hom a c));
  Id      :  (a:Ob)(Hom a a);
  Prf_ass :  (Assoc_law Op_comp);
  Prf_idl :  (Idl_law Op_comp Id);
  Prf_idr :  (Idr_law Op_comp Id)}.
```

Nous définissons l'application de l'opérateur de composition d'une catégorie, de sorte à utiliser la notation infixe `o` dans le contexte d'une catégorie quelconque `C`.

```
Definition Comp := [C:Category](Cat_comp (!Op_comp C)).
Infix 6 "-->" Hom.
Infix 6 "o" Comp.
```

Moralement, un opérateur de composition n'est rien d'autre qu'une fonction binaire vérifiant les lois de congruence pour chacun de ses arguments. Nous fournissons ainsi une méthode générale permettant de construire un opérateur de composition à partir d'une telle fonction. Nous allons systématiquement utiliser cet outil pour toute définition de catégorie.

```
Section composition_to_operator.
Variables A:Type; H:A -> A -> Setoid; Cfun:(a,b,c:A)(H a b) -> (H b c) -> (H a c).
Definition Congl_law := (a,b,c:A)(f,g:(H b c))(h:(H a b))
           f =_S g -> (Cfun h f) =_S (Cfun h g).
Definition Congr_law := (a,b,c:A)(f,g:(H a b))(h:(H b c))
           f =_S g -> (Cfun f h) =_S (Cfun g h).
Definition Cong_law := (a,b,c:A)(f,f':(H a b))(g,g':(H b c))
           f =_S f' -> g =_S g' -> (Cfun f g) =_S (Cfun f' g').
Hypothesis pcgl : Congl_law.
Hypothesis pcgr : Congr_law.
Definition Build_Comp := [a,b,c:A](Build_Map2 (!pcgl a b c) (!pcgr a b c)).
End composition_to_operator.
```

8.2 Catégorie des Setoïdes

Nous définissons maintenant la catégorie des Setoïdes avec les applications comme morphismes. Nous devons d'abord définir la composition et l'identité d'applications. La composition d'application est définie à partir de la composition de leur fonction sous-jacente. Nous utilisons la notation infixe `o_M`.

```
Section mcomp.
Variables A,B,C:Setoid; f:(Map A B); g:(Map B C).
Definition Comp_fun := [x:A](g (f x)).
Lemma Comp_fun_map_law : (Map_law Comp_fun).
@Definition Comp_map : (Map A C) := Comp_fun_map_law.
End mcomp.
Infix 6 "o_M" Comp_map.
```

L'opérateur `Comp_map` est seulement une fonction. Nous devons maintenant prouver qu'il est extensionnel en ses deux arguments pour obtenir un opérateur de composition sur les applications.

```
Lemma Comp_map_congl : (Congl_law Comp_map).
Lemma Comp_map_congr : (Congr_law Comp_map).
Definition Comp_SET := (Build_Comp Comp_map_congl Comp_map_congr).
Lemma Assoc_SET : (Assoc_law Comp_SET).
```

Après la vérification de l'associativité de notre opérateur de composition, nous définissons l'application identité à partir de la fonction identité $x \mapsto x$. Nous vérifions aussi les autres lois des catégories.

```
 Section id_map_def.
 Variable A:Setoid.
 Definition Id_fun := [x:A]x.
 Lemma Id_fun_map_law : (Map_law Id_fun).
 @Definition Id_map : (Map A A) := Id_fun_map_law.
 End id_map_def.
Definition Id_SET := Id_map.
Lemma Idl_SET : (Idl_law Comp_SET Id_SET).
Lemma Idr_SET : (Idr_law Comp_SET Id_SET).
```

Nous avons enfin tous les ingrédients pour former SET, la catégorie des Setoïdes.

```
@Definition SET := (Build_Category Assoc_SET Idl_SET Idr_SET).
```

Avertissement. Une grande partie des lemmes énoncés jusqu'à maintenant peuvent être considérés comme triviaux; ils correspondent aux preuves liées au langage des Setoïdes, c'est-à-dire preuves d'équivalence pour les Setoïdes et de préservation d'égalité pour les applications, ainsi que les preuves de congruence pour les opérateurs de composition. Pour ne pas lasser le lecteur, nous omettrons dorénavant (sauf rares exceptions) tous ces énoncés, ainsi que ceux correspondant aux lois des catégories.

8.3 Catégorie Duale

Étant donnée une catégorie C, un moyen de construire une nouvelle catégorie est d'inverser tous les morphismes de C. La nouvelle catégorie obtenue est appelée la catégorie duale de C et est notée C°. Plus précisément, les objets de C° sont les mêmes que ceux de C. Quant aux morphismes, si $f : a \to b$ est un morphisme de C, alors $f^\circ : b \to a$ est un morphisme de C°. Tous les morphismes de C° sont obtenus de cette manière. Ainsi, par définition $Hom_{C^\circ}(a,b) = Hom_C(b,a)$.

```
Section d_cat.
Variable C:Category.
Definition DHom := [a,b:C]b-->a.
```

La composition $\circ^\circ$ est définie comme attendu, $f^\circ \circ^\circ g^\circ = g^\circ \circ f^\circ$.

```
Definition Comp_Darrow := [a,b,c:C][df:(DHom a b)][dg:(DHom b c)]dg o df.
```

L'identité reste quant à elle inchangée. Nous écrivons `(Dual C)` pour C° .

```
@Definition Dual := (Build_Category Assoc_Dual Idl_Dual Idr_Dual).
End d_cat.
```

8.4 Propriétés simples sur les Catégories

Lorsque nous raisonnons à propos d'ensembles et de fonctions, nous nous intéressons souvent à des fonctions avec des propriétés supplémentaires. Des propriétés analogues jouent un rôle important en théorie des catégories. La difficulté réside dans le fait qu'il faut les exprimer sans faire intervenir la structure interne des objets, mais seulement en utilisant des équations entre morphismes.

Un morphisme $f : a \to b$ est *épi* lorsque pour tous morphismes $g, h : b \to c$, l'égalité $f \circ g = f \circ h$ implique $g = h$.

$$a \xrightarrow{f} b \underset{h}{\overset{g}{\rightrightarrows}} c \tag{1}$$

```
Section epic_monic_def.
Variables C:Category; a,b:C.
Definition Epic_law := [f:a-->b](c:C)(g,h:b-->c)(f o g) =_S (f o h) -> g =_S h.
Structure >Epic : Type :=
 {Epic_mor   :  a-->b;
  Prf_isEpic :> (Epic_law Epic_mor)}.
```

Un morphisme $f : b \to a$ est *monic* lorsque pour tous morphismes $g, h : c \to b$, l'égalité $g \circ f = h \circ f$ implique $g = h$.

$$c \underset{h}{\overset{g}{\rightrightarrows}} b \xrightarrow{f} a \tag{2}$$

```
Definition Monic_law := [f:b-->a](c:C)(g,h:c-->b)(g o f) =_S (h o f) -> g =_S h.
Structure >Monic : Type :=
 {Monic_mor   :  b-->a;
  Prf_isMonic :> (Monic_law Monic_mor) }.
End epic_monic_def.
```

Deux morphismes $f : a \to b$ et $f^{-1} : b \to a$ sont *iso* s'ils sont mutuellement inverses, c'est-à-dire $f^{-1} \circ f = Id_b$ and $f \circ f^{-1} = Id_a$.

```
Section iso_def.
Variable C:Category.
Definition RIso_law := [a,b:C][f:a-->b][f1:b-->a](f1 o f) =_S (Id b).
Variable a,b:C.
Definition AreIsos := [f:a-->b][f1:b-->a] (RIso_law f f1)/\(RIso_law f1 f).
```

Nous ne pouvons pas à proprement parler définir la propriété `IsIso` d'être un isomorphisme car dans ce cas le champ `Inv_iso` correspondant au morphisme inverse devient inaccessible (voir 3.4.3).

```
Structure IsIso [f:a --> b] : Prop :=
 {Inv_iso : b --> a;
  Prf_Iso :> (AreIsos f Inv_iso)}.
```

Nous rencontrons ce problème aussi avec les notions d'objet terminal et d'objet initial. Dans tous ces cas, nous remédions à ce problème en transformant le champ incriminé en paramètre de l'enregistrement. Les définitions deviennent ainsi peu naturelles. Une autre solution est de bannir la sorte *Prop* en définissant toutes les notions au niveau non-imprédicatif *Type*. Nous disposons d'ailleurs d'une version complètement prédicative de notre développement.

Nous disons que deux objets a et b sont *isomorphes* ($a \cong b$) s'ils sont connectés par une paire de morphismes isos. C'est une notion importante car les constructions catégoriques déterminent souvent un objet à isomorphisme près.

```
Structure >Iso : Type :=
 {Iso_mor : a-->b;
  Inv_iso : b-->a;
  Prf_Iso :> (AreIsos Iso_mor Inv_iso)}.
End iso_def.
```

Nous disons qu'un objet a est *initial* dans la Category C si et seulement si pour tout objet b, il existe un unique morphisme dans $Hom(a, b)$.

```
Section initial_def.
Variable C:Category.
Definition IsInitial := [a:C][h:(b:C)a-->b](b:C)(f:a--> b)f =_S (h b).
Structure >Initial : Type :=
 {Initial_ob    :  C;
  MorI          : (b:C) Initial_ob-->b;
  Prf_isInitial :> (IsInitial MorI)}.
```

Comme exercice, nous prouvons que deux objets initiaux dans une catégorie sont isomorphes.

```
Lemma I_unic : (i1,i2:Initial)(Iso (Initial_ob i1) (Initial_ob i2)).
End initial_def.
```

De manière duale, nous définissons la notion d'objet *terminal*: b est terminal dans une catégorie C si pour tout objet a, il existe un unique morphisme dans $Hom(a,b)$.

```
Section terminal_def.
Variable C:Category.
Definition IsTerminal := [b:C][h:(a:C)a-->b](a:C)(f:a-->b)f =_S (h a).
Structure >Terminal : Type :=
 {Terminal_ob    :  C;
  MorT           : (a:C)a-->Terminal_ob;
  Prf_isTerminal :> (IsTerminal MorT)}.
End terminal_def.
```

Nous prouvons que la propriété d'être terminal pour un objet est duale à celle d'être initial: un objet initial dans C est un objet terminal dans C°.

```
Lemma Initial_dual : (C:Category)(a:C)(h:(b:C)a --> b)(IsInitial h) ->
                     (IsTerminal 1!(Dual C) h).
Coercion Initial_dual : IsInitial >-> IsTerminal.
```

Dans la catégorie SET, nous avons les correspondances suivantes:

Langage catégorique	Correspondance dans SET
Morphisme épi	Application surjective
Morphisme monic	Application injective
Morphisme iso	Application bijective
Objet initial	Ensemble vide
Objet terminal	Ensemble singleton

8.5 Foncteurs

Les Foncteurs entre deux catégories C et D sont définis de manière usuelle, avec deux composantes, une fonction des objets de C vers les objets de D, et une application des hom-Setoïdes de C vers les hom-Setoïdes de D. Remarquez que la théorie des types nous permet d'exprimer ces contraintes de manière naturelle, sans codage arbitraire.

```
Section funct_def.
Variables C,D:Category.
 Section funct_laws.
 Variables FOb:C -> D; FMap:(a,b:C)(Map a-->b (FOb a)-->(FOb b)).
```

Les foncteurs doivent préserver la structure de catégorie, ils vérifient donc les deux lois: $F(f \circ g) = F(f) \circ F(g)$ et $F(Id_a) = Id_{F(a)}$.

```
 Definition Fcomp_law := (a,b,c:C)(f:a-->b)(g:b-->c)
            ((FMap a c) (f o g)) =_S (((FMap a b) f) o ((FMap b c) g)).
 Definition Fid_law := (a:C)((FMap a a) (Id a)) =_S  (Id (FOb a)).
 End funct_laws.
Structure Functor : Type :=
 {FOb           :> C -> D;
  FMap          :  (a,b:C)(Map a-->b (FOb a)-->(FOb b));
  Prf_Fcomp_law :  (Fcomp_law FMap);
  Prf_Fid_law   :  (Fid_law  FMap)}.
```

Nous définissons enfin une abréviation de manière à écrire `(FMor F f)` pour $F(f)$.

```
Definition FMor := [F:Functor][a,b:C][f:a-->b]((FMap F a b) f).
End funct_def.
```

8.6 Égalité de morphismes

La première difficulté que nous rencontrons intervient lors de la définition de l'égalité de foncteurs. Celle-ci est extensionnelle sur la composante de l'application entre hom-Setoïdes. Soient $F, G : C \to D$ deux foncteurs, $F = G$ si et seulement si $\forall f : a \to b.\ F(f) = G(f)$. Or les deux morphismes $F(f)$ et $G(f)$ de D appartiennent à deux hom-Setoïdes différents, respectivement $Hom(F(a), F(b))$ et $Hom(G(a), G(b))$. Ainsi l'énoncé `(FMor F f) =_S (FMor G f)` est mal typé. Cette difficulté est fréquente en théorie des types avec types dépendants; la solution que nous donnons est générique. Nous contournons cette difficulté en définissant une notion plus générale d'égalité de morphismes dans une catégorie .

```
Inductive Equal_hom [C:Category;a,b:C;f:a-->b] : (c,d:C)(c-->d) -> Prop :=
Build_Equal_hom : (g:a-->b)f =_S g -> (Equal_hom f g).
Infix 6 "=_H" Equal_hom.
```

Le prédicat `Equal_hom` prend comme arguments une catégorie `C`, des objets `a,b,c,d` de `C`, et des morphismes `f:a-->b` et `g:c-->d`. Puisque la seule construction possible pour cette égalité est avec `Build_Equal_hom`, qui nécessite que le second morphisme `g` ait le même type que le premier morphisme `f`, il semble suffisant de restreindre le type de `Equal_hom` aux morphismes de même type. Cependant cette généralité est nécessaire parce que nous voulons être capables d'énoncer l'égalité de deux morphismes dont les domaines respectifs ne sont pas définitionnellement égaux, mais qui le seront pour certaines instanciations de paramètres. Nous sommes maintenant capables d'énoncer $F(f) = G(f)$ par `(FMor F f) =_H (FMor G f)` qui va forcer les objets $F(a)$ et $G(a)$ à être définitionnellement égaux (de même pour $F(b)$ et $G(b)$), mais il n'y a aucun moyen de spécifier F et G avec une déclaration de type telle que $F(a) = G(b)$. Ceci nécessiterait une extension de la théorie des types avec des contraintes définitionnelles, qui pourraient être problématiques pour la décidabilité de l'égalité définitionnelle. Cette extension n'est pas réellement nécessaire si l'on prend soin d'écrire les égalités dépendantes avec un type suffisamment général.
Nous nous proposons maintenant d'étudier cette nouvelle égalité de morphismes. Nous aborderons trois principaux points.

1. Vérifions d'abord que `=_H` est une relation d'équivalence. Remarquez qu'on ne peut pas exprimer cette propriété avec les définitions déjà vues pour les relations tels que `Equivalence`, `Reflexive`, `Symmetric` et `Transitive`, car les deux principaux arguments de `=_H` sont de types différents.

   ```
   Section equal_hom_equiv.
   Variables C:Category; a,b:C; f:a-->b.
   Lemma Equal_hom_refl : f =_H f.
   Variables c,d:C; g:c-->d.
   Lemma Equal_hom_sym : f =_H g -> g =_H f.
   Variables i,j:C; h:i-->j.
   Lemma Equal_hom_trans : f =_H g -> g =_H h -> f =_H h.
   End equal_hom_equiv.
   ```

2. La définition `=_H` n'échappe pas au dysfonctionnement constaté dans certaines définitions mélangeant les types inductifs et les types dépendants. Ce phénomène a été intensivement étudié dans le cas de l'égalité de Leibniz, par M. Hofmann et T. Streicher (notamment dans [74]). Dans ce cas, il s'avère impossible de prouver l'unicité de la preuve de reflexivité; ils proposent alors d'y remédier en introduisant un nouveau schéma d'élimination appelé K.

   Dans notre cas, on constate qu'on ne peut pas «inverser» `=_H`, c'est-à-dire on ne peut pas démontrer:

```
(C:Category)(a,b:C)(f,g:a-->b)f =_H g -> f =_S g.
```

   Ceci est dû au schéma d'induction trop faible engendré pour `=_H`:

```
(C:Category)(a,b:C)(f:a-->b)
    (P:(c,d:C)(g:c-->d)f =_H g -> Prop)
       ((g:a-->b)(e:f =_S g)(P a b g (Build_Equal_hom e))) ->
           (c,d:C)(g:c-->d)(e:f =_H g)(P c d g e)
```

   Dans le même esprit que K, nous proposons ci-dessous un nouveau schéma d'élimination pour `=_H` indiquant que `(Build_Equal_hom e)` est l'unique preuve de `f =_H g` lorsque `f` et `g` appartiennent au même hom-Setoïde et avec `e` une preuve de `f =_S g`.

```
(C:Category)
    (P:(a,b:C)(f,g:a-->b)f =_H g -> Prop)
       ((a,b:C)(f,g:a-->b)(e:f =_S g)(P a b f g (Build_Equal_hom e))) ->
          (a,b:C)(f,g:a-->b)(p:f =_H g)(P a b f g p)
```

   Cet axiome s'avère toutefois inutile dans notre cas, car on n'y aura jamais recours dans notre développement.

3. La définition de `=_H` peut être généralisée au cas de Setoïdes quelconques; nous appelons cette égalité `Equal_gen`. Celle-ci peut être alors instanciée au cas particulier des hom-Setoïdes; nous noterons par `=_H1` l'égalité résultante.

```
Inductive Equal_gen [A:Setoid;a:A]: (B:Setoid)B -> Prop :=
  Build_Equal_gen : (b:A)a =_S b -> (Equal_gen a b).
Definition Equal_hom1 := [C:Category][a,b,c,d:C][f:a-->b][g:c-->d](Equal_gen f g).
Infix 2 "=_H1" Equal_hom1.
```

   Malheureusement, on échoue à montrer que les deux égalités `=_H` et `=_H1` sont équivalentes. Un seul sens de l'équivalence est démontrable:

```
Lemma Equal_hom_gen : (C:Category)(a,b:C)(f:a-->b)(c,d:C)(g:c-->d)
                      f =_H g -> f =_H1 g.
```

   Quant à l'autre sens de l'équivalence, il est à notre connaissance non démontrable dans Coq (en toutes rigueur, il faut prouver dans la méta-théorie que la proposition correspondante est non habitée).

```
(C:Category)(a,b:C)(f:a-->b)(c,d:C)(g:c-->d)f =_H1 g -> f =_H g.
```

8.7 La Catégorie des Catégories

Dans cette section, nous refléchissons la théorie dans elle-même: les catégories forment le type d'une catégorie des catégories CAT, les morphismes étant les foncteurs. On vérifie que le type des foncteurs possède une structure de Setoïde, en utilisant l'égalité définie plus haut. On écrira l'égalité de foncteurs par `=_F` avec la synthèse d'arguments appropriée. La définition de `Functor_setoid` (le Setoïde des foncteurs) en présence de la constante `SET` provoque une incohérence d'univers. Nous définissons alors `Functor_setoid` de type `Setoid'`; `Setoid'` étant une copie de `Setoid`. Toutes les notions utilisant le type `Setoid` ont été redéfinies pour utiliser le type `Setoid'`. Le nom de chaque copie est obtenue en accolant une apostrophe « ' » à la fin du nom de la notion originelle correspondante .

```
Section funct_setoid.
Variables C,D:Category.
Definition Equal_Functor := [F,G:(Functor C D)] (a,b:C)(f:a-->b)(FMor F f) =_H (FMor G f).
Lemma Equal_Functor_equiv : (Equivalence Equal_Functor).
@Definition Functor_setoid : Setoid' := Equal_Functor_equiv.
End funct_setoid.
Infix 2 "=_F" Equal_Functor.
```

Tout ce que l'on a à faire est de définir la composition et l'identité de foncteurs, et de prouver quelques lemmes exhibant la structure de catégorie de CAT. La première étape consiste en la définition de la composition de deux foncteurs. Nous composons les foncteurs $G : C \longrightarrow D$ et $H : D \longrightarrow E$ pour former un foncteur $G \circ H : C \longrightarrow E$, en composant séparément leur fonction sur les objets et leur application sur les morphismes. Nous écrivons `o_F` pour cette composition de foncteur.

```
Section Comp_F.
Variables C,D,E:Category; G:(Functor C D); H:(Functor D E).
Definition Comp_FOb := [a:C](H (G a)).
Definition Comp_FMor := [a,b:C][f:a-->b](FMor H (FMor G f)).
@Definition Comp_Functor := (Build_Functor Comp_Functor_comp_law Comp_Functor_id_law).
End Comp_F.
Infix 6 "o_F" Comp_Functor.
```

Comme précédemment, nous construisons un opérateur de composition après vérification des lois de congruence. Pour toute catégorie C, nous construisons le foncteur identité Id_C à partir de la fonction identité sur les objets, et de l'application identité sur les morphismes.

```
Section idCat.
Variable C:Category.
Definition Id_CAT_ob := [a:C]a.
Definition Id_CAT_map := [a,b:C](Id_map a-->b).
@Definition Id_CAT := (Build_Functor Id_CAT_comp_law Id_CAT_id_law).
End idCat.
```

Nous avons maintenant tous les ingrédients pour reconnaître dans CAT la structure de catégorie. Tout ce qu'on doit faire est de prendre une seconde copie de la notion de catégorie, appelée `Category'`. Le mécanisme d'ajustement implicite des univers nous assure que son type référence un univers plus grand. La définition de `CAT` de type `Category` est illégale car elle provoque une incohérence d'univers (même dans un contexte ne contenant pas la constante `SET`).

```
Structure Category' : Type :=
 {Ob'      :> Type;
  Hom'     :  Ob' -> Ob' -> Setoid';
  Opcomp'  :  (a,b,c:Ob')(Map2' (Hom' a b) (Hom' b c) (Hom' a c));
  Id'      :  (a:Ob')(Hom' a a);
  Prf_ass' :  (Assoc_law' Opcomp');
  Prf_idl' :  (Idl_law' Opcomp' Id');
  Prf_idr' :  (Idr_law' Opcomp' Id')}.
Definition CAT : Category' := (Build_Category' Assoc_CAT Idl_CAT Idr_CAT).
```

Nous en venons maintenant au problème bien connu de «taille» des catégories . Une catégorie est dite *petite* si sa collection d'objets est en correspondance avec un ensemble, elle est dite *grande* sinon (en opposition avec la notion de classe). Une catégorie est dite *localement petite* si pour tous objets a et b, $Hom(a,b)$ est en correspondance avec un ensemble. Les univers flottants nous affranchissent de ces «détails»; nous n'avons pas de petites et grandes catégories, mais des catégories «relativement petites» . Il n'y a pas un unique CAT, il y a une famille de CAT_i, et chaque CAT_i est une $Category_j$ avec $i < j$. La construction ci-dessus est alors cohérente avec l'analyse de Coquand[43] concernant les paradoxes liés à la catégorie des catégories.

On doit aussi remarquer que cet exemple justifie le mécanisme appelé «polymorphisme d'univers» (voir 3.2.3) défini par Harper et Pollack[68]. Avec ce mécanisme, nous pourrions définir directement `CAT` comme une catégorie (de type `Category`), sans avoir à faire une copie explicite de cette notion, la copie étant implicitement réalisée pour chaque occurrence du nom `Category`. La version actuelle de Coq n'implémente pas le polymorphisme d'univers, parce que ce mécanisme est plutôt coûteux en espace, et rarement utilisé en pratique.

8.8 Transformation naturelle

Le type des foncteurs de la catégorie C vers la catégorie D admet une structure de catégorie. Les morphismes correspondants sont appelés *Transformations Naturelles*. Une transformation naturelle T de $F : C \to D$ vers $G : C \to D$ associe à tout objet a de C, un morphisme T_a de l'objet $F(a)$ vers l'objet $G(a)$ dans D, tel que la loi de naturalité suivante est vérifiée: $F(f) \circ T_b = T_a \circ G(f)$.

$$\begin{array}{ccc} a & F(a) & \xrightarrow{T_a} & G(a) \\ \Big\downarrow f & \Big\downarrow F(f) & & \Big\downarrow G(f) \\ b & F(b) & \xrightarrow[T_b]{} & G(b) \end{array} \tag{3}$$

Notez que les transformations naturelles sont définies comme fonctions et non comme applications, puisque les objets n'ont pas une structure de Setoïde.

```
Section nt_def.
Variable C,D:Category; F,G:(Functor C D).
Definition NT_law := [T:(a:C)(F a)-->(G a)] (a,b:C)(f:a-->b)
                     ((FMor F f) o (T b)) =_S ((T a) o (FMor G f)).
Structure >NT : Type :=
```

```
 {ApNT       :> (a:C)(F a)-->(G a);
  Prf_NT_law :> (NT_law ApNT)}.
End nt_def.
```

Nous définissons maintenant le Setoïde des transformations naturelles entre les foncteurs F et G. L'égalité de transformations naturelles est aussi extensionnelle. Ainsi, deux transformations naturelles T_1 et T_2 sont dites égales lorsque leurs composantes sont égales pour un objet arbitraire: $\forall a.T_1(a) = T_2(a)$. Comme d'habitude, nous écrivons `=_NT` pour cette égalité. Pour remédier aux problèmes d'incohérence d'univers (toujours avec `SET`), nous définissons une nouvelle copie `Setoid''` de la notion de Setoïde, et nous définissons `NT_setoid` avec le type `Setoid''` .

```
Section setoid_nt.
Variable C,D:Category; F,G:(Functor C D).
Definition Equal_NT := [T,T':(NT F G)](a:C)(T a) =_S (T' a).
@Definition NT_setoid : Setoid'' := Equal_NT_equiv.
End setoid_nt.
Infix 2 "=_NT" Equal_NT.
```

8.9 Catégorie des Foncteurs

Nous avons maintenant tous les outils pour définir la catégorie des foncteurs de C vers D. Les objets sont les foncteurs et les morphismes sont les transformations naturelles. La catégorie des foncteurs joue un rôle essentiel; en effet beaucoup de propriétés des foncteurs ou des transformations naturelles s'expriment simplement en tant que propriétés d'objets ou de morphismes de la catégorie des foncteurs. Nous définissons la composition de deux transformations naturelles T et T' par $(T \circ_v T')_a = T_a \circ T'_a$. L'indice v signifie «verticale», car nous allons définir plus tard une autre composition «horizontale» .

La notion de catégorie et toutes les notions s'y rattachant doivent être aussi une deuxième fois dupliquées pour prendre en compte la définition de `Setoid''`.

```
Section cat_functor.
Variables C,D:Category.
 Section compnt.
 Variables F,G,H:(Functor C D); T:(NT F G); T':(NT G H).
 Definition Comp_tau := [a:C] (T a) o (T' a).
 @Definition CompV_NT : (NT_setoid F H) := Comp_tau_nt_law.
 End compnt.
Definition Comp_CatFunct := (Build_Comp'' CompV_NT_congl CompV_NT_congr).
```

À tout foncteur F, nous associons une transformation identité Id_F définie par $\lambda a.Id_{F(a)}$:

```
 Section id_catfunct_def.
 Variable F:(Functor C D).
 Definition Id_CatFunct_tau := [a:C](Id (F a)).
 @Definition Id_CatFunct := (Build_NT Id_CatFunct_nt_law).
 End id_catfunct_def.
```

Ayant vérifié toutes les propriétés catégoriques, nous définissons la catégorie des foncteurs, que nous notons $FUNCT(C, D)$. La modification du type de la catégorie des foncteurs de `Category` en `Category''` aura d'importantes répercussions sur le reste du développement car c'est une notion très utilisée en théorie des catégories.

```
@Definition FUNCT :=(Build_Category'' Assoc_CatFunct Idl_CatFunct Idr_CatFunct).
End cat_functor.
Infix 6 "o_NTv" CompV_NT.
```

8.10 Techniques de preuve - Complétion

Cette section a pour objectif de construire des procédures de décision pour l'égalité entre morphismes de catégories. Nous commençons par faire une digression méta-théorique pour décrire la méthode adoptée.

8.10.1 Description de la méthode

Il est d'usage de décrire les équations entre morphismesd'une catégorie sous forme de diagrammes. Un diagramme catégorique est un graphe dont les sommets et les arcs sont respectivement libellés par les objets et les morphismes de la catégorie et tel que les sommets de départ et d'arrivée d'un arc sont libellés par les objets domaine et codomaine du morphisme libellant cet arc. Tout chemin dans le diagramme détermine un morphisme en composant les morphismes le long de ce chemin. On dit qu'un diagramme commute si tous les chemins entre deux mêmes objets déterminent un même morphisme. La preuve d'énoncés catégoriques (égalités de morphismes) se fait alors en vérifiant la commutation de diagrammes et en juxtaposant des diagrammes commutatifs le long de chemins communs. Cet aspect graphique des preuves a suggéré de nouveaux langages de programmation pouvant manipuler des diagrammes en deux dimensions (voir les travaux de Y. Lafont[95]).

Il nous faut citer les travaux de D. Wätjan et W. Struckmann[154] concernant le développement d'un algorithme pour la vérification d'équations entre morphismes en présence d'un nombre quelconque d'équations mettant en jeu des foncteurs et des transformations naturelles. La vérification est réalisée en transformant, étape par étape, le premier membre de l'équation à vérifier de façon à retrouver le second membre. À chaque étape, une sous-séquence du premier membre est remplacée par un terme suivant une équation donnée. Un arbre (partiel) de toutes les transformations possibles est construit, et son parcours s'effectue par retour-arrière (backtracking).

Nous adoptons pour notre part l'approche algébrique, en considérant le raisonnement catégorique comme déduction équationnelle typée, nous inspirant ainsi de [82]. Nous sommes intéressés par décider l'égalité entre deux morphismes donnés d'une catégorie. Partant d'une caractérisation équationnelle, nous tentons d'obtenir un système de réécriture canonique et équivalent garantissant l'existence et l'unicité d'une forme normale, et pouvant ainsi être utilisé comme procédure de décision. Les systèmes de réécriture présentés dans cette section sont obtenus par complétion, en utilisant une procédure de Knuth-Bendix[91]. Nous décrivons ci-dessous sommairement cette procédure. Une présentation plus complète se trouve dans [79].

Définition 8.1 (Spécification équationnelle) *Soit $\Sigma = (V, F, ar)$ une signature. Une équation est une paire non ordonnée de termes $(s, t) \in T_\Sigma \times T_\Sigma$.*

Une spécification équationnelle est alors un couple (Σ, E) comportant un ensemble E d'équations. Nous notons $(s,t) \in E$ par $s =_E t$ ou bien $s = t$ s'il n'y a pas risque d'ambiguïté.

Définition 8.2 (Système de déduction équationnelle) *Soit (Σ, E) une spécification équationnelle. Les règles d'inférence ci-dessous constituent le* Système de déduction équationnelle *de (Σ, E).*

$$\text{(AXIOME)}\ \frac{s = t \in E}{s = t} \qquad \text{(REFLEXIVITÉ)}\ \frac{}{t = t}$$

$$\text{(SYMÉTRIE)}\ \frac{s = t}{t = s} \qquad \text{(TRANSITIVITÉ)}\ \frac{s = t \quad t = u}{s = u}$$

$$\text{(CONGRUENCE)}\ \frac{s_1 = t_1 ... s_{ar(f)} = t_{ar(f)}}{f(s_1, ..., s_{ar(f)}) = f(t_1, ..., t_{ar(f)})} \qquad \text{(REMPLACEMENT)}\ \frac{s = t}{\sigma(s) = \sigma(t)}$$

Construire une preuve en utilisant les règles d'inférence ci-dessus n'est pas facile: à chaque étape, on peut avoir plusieurs possibilités d'application de règles. Il faut faire aussi attention à l'utilisation de la règle de symétrie. La solution consiste à n'utiliser les équations que dans un sens; nous les transformons ainsi en règles de réécriture. Pour obtenir un système équivalent, il faudra généralement augmenter le nombre de règles. Nous utilisons alors les techniques de réécriture pour les systèmes canoniques, développées en 6.3.

On va montrer que la confluence locale peut se ramener à l'examen d'un certain nombre de couples de termes appelées *paires critiques*.

Définition 8.3 (Unificateur) *Soit $\Sigma = (V, F, ar)$ une signature. Un* unificateur *de deux termes $s, t \in T_\Sigma$ est une substitution θ telle que $\theta(s) \equiv \theta(t)$ (les termes s et t sont préalablement renommés afin de vérifier $var(s) \cap var(t) = \emptyset$).*
Un unificateur principal *de deux termes est un unificateur θ tel que pour tout autre unificateur θ', il existe une substitution σ avec $\theta' = \theta \circ \sigma$.*

Définition 8.4 (Paires Critiques) *Soient $s_1 \to t_1$ et $s_2 \to t_2$ deux règles d'un système de réécriture (Σ, R), et n'ayant pas de variables communes (modulo renommage). S'il existe un sous-terme $u \notin V$ de s_1 ($s_1 \equiv C[u]$) unifiable avec s_2, c'est-à-dire $\theta(u) \equiv \theta(s_2)$ avec θ l'unificateur le plus général de u et s_2, alors $(\theta(t_1), \theta(C[t_2]))$ est appelée paire critique.*
Une paire critique (u, v) converge *si et seulement si il existe un terme w tel que $u \to^* w \leftarrow^* v$.*

Lemme 8.1 (Lemme des paires critiques) *Un système de réécriture est localement confluent si et seulement si toutes ses paires critiques convergent.*

L'objectif de la complétion est de transformer une spécification équationnelle (Σ, E) en un système de réécriture fini (Σ, R) tel que $\to_R$ soit complet et $=_E$ et $\longleftrightarrow^*$ soient équivalents.

Définition 8.5 (Procédure de Knuth-Bendix) *Soit (Σ, E) une spécification équationnelle, et un ordre $\prec$ sur T_Σ. Nous présentons la procédure de Knuth-Bendix sous formes de règles d'inférence ayant comme conclusion des paires $E; R$ où E est un ensemble d'équations et R un ensemble de règles. La configuration de départ est $E; \emptyset$. $CP(R)$ est l'ensemble des paires critiques des règles de R.*

$$(\text{DÉDUCTION})\ \frac{E;R}{E \cup CP(R);R} \qquad (\text{ORIENTATION})\ \frac{E \cup \{s=t\};R}{E;R \cup \{s \to t\}}\ si\ s \prec t$$

$$(\text{ÉLIMINATION})\ \frac{E \cup \{s=s\};R}{E;R} \qquad (\text{SIMPLIFICATION})\ \frac{E \cup \{s=u\};R}{E \cup \{s=t\};R}\ si\ s \to t \in R$$

$$(\text{COMPOSITION})\ \frac{E;R \cup \{u \to s\}}{E;R \cup \{u \to t\}}\ si\ s \to t \in R$$

Cette procédure prend en entrée une spécification équationnelle (Σ, E) et un ordre bien fondé sur les termes (sous forme d'une fonction d'interprétation). Ces équations sont alors orientées suivant cet ordre, et d'autres équations sont engendrées correspondant aux paires critiques de manière à garantir la confluence locale. Si la procédure termine, elle fournit un système de réécriture canonique. Une preuve complète de la procédure de Knuth-Bendix se trouve dans [80]. Cette procédure a été étendue de différentes manières, notamment pour inclure des opérateurs associatifs et commutatifs.

Une particularité des systèmes que nous considérons est qu'ils sont typés; un morphisme est typé par sa catégorie et ses deux objets domaine et codomaine. La technique adoptée dans ce cas-là est décrite dans [82] et appliquée dans [85] au cas des catégories cartésiennes closes. Cette technique consiste à compléter la spécification équationnelle en ignorant les informations de type, puis à revérifier la confluence locale du système canonique obtenu en y intégrant le typage. Cette méthode a été intensivement utilisée dans [62] sur différentes constructions catégoriques et notamment les monades.

Les réécritures se font exclusivement avec l'égalité des Setoïdes `=_S`; les autres égalités étant toutes définies en fonctions d'elle. Une étape préalable à toute preuve par réécriture est alors de transformer le but à prouver en une égalité de morphismes.

Une fois le système de réécriture défini, nous utilisons les tactiques décrites au chapitre 6 pour construire une conversion lui correspondant. Cependant on n'utilisera pas systématiquement les conversions génériques de normalisation développées dans le chapitre 6 pour les systèmes canoniques. En effet on préférera dans certains cas simples développer des stratégies de normalisation plus adaptées pour le système étudié, car plus efficaces au vu du nombre de réécritures effectuées pour atteindre la forme normale.

8.10.2 Cas des Catégories

Le système de réécriture pour les catégories est le suivant:

$$\begin{array}{lrcl} (Ass) & (f \circ g) \circ h & \to & f \circ (g \circ h) \\ (Idl) & Id_a \circ f & \to & f \\ (Idr) & f \circ Id_b & \to & f \end{array}$$

La terminaison de ce système se montre par l'utilisation de l'ordre défini par: $t > u$ si et seulement si $\phi(t) > \phi(u)$ où ϕ est une interprétation polynomiale à valeurs dans $\mathbb{N}^*$.

$$\begin{array}{rcl} \phi(Id_a) & = & 1 \\ \phi(f \circ g) & = & 2 \times \phi(f) + \phi(g) \end{array}$$

Les trois règles du système de réécriture correspondent à des orientations des axiomes d'une catégorie. Soient $AssC$, $IdlC$ et $IdrC$ les conversions associées à ces règles. Elles sont obtenues par application de $ruleC$, par exemple:

$let\ AssC = ruleC$ `Prf_ass` RL [] []

Rappelons le type de `Prf_ass`:

```
(C:Category)(a,b,c,d:C)(f:a-->b)(g:b-->c)(h:c-->d)
                        f o (g o h) =_S (f o g) o h
```

Le système de réécriture est présenté pour une catégorie fixée. Les lemmes de réécriture sont toutefois abstraits par rapport à la catégorie; il est inutile de la fournir explicitement pour chaque réécriture, elle sera inférée à partir du terme à réécrire.

L'opérateur de composition $\circ$ correspond au symbole de fonction `Comp`. La catégorie et les différents objets intervenant en sont les paramètres. Ceci est plus visible sur le type «sans abus de notation» de `Prf_ass`:

```
(C:Category)(a,b,c,d:(Ob C))
  (f:(Carrier (Hom C a b)))(g:(Carrier (Hom C b c)))(h:(Carrier (Hom C c d)))
        (Equal (Hom C a d) (Comp C a b d f (Comp C b c d g h))
                           (Comp C a c d (Comp C a b c f g) h))
```

Nous définissons d'abord une conversion qui élimine les identités.

$let\ ElimIdC\ idlC\ idrC = TryC\ (SweepUpC\ (FirstC\ [idlC; idrC]))$

Puis nous définissons une conversion qui associe les compositions à droite.

$let\ RAssC\ assC = TryC\ (SweepDnC\ (RepeatC\ assC))$

Ces deux conversions sont alors combinées:

$let\ MonC\ assC\ idlC\ idrC = (ElimIdC\ idlC\ idrC)\ ThenC\ (RAssC\ assC)$

Nous venons ainsi de définir une conversion de normalisation pour les monoïdes. Le cas des catégories n'est qu'un cas particulier:

$let\ CatC = MonC\ AssC\ IdlC\ IdrC$

Il existe un autre système pour les axiomes des catégories qui consiste à associer les compositions à gauche.

8.10.3 Cas des Foncteurs

Le système de réécriture pour les foncteurs est le suivant:

$$\begin{array}{lrcl}
(Ass) & (f \circ g) \circ g & \rightarrow & f \circ (g \circ h) \\
(Idl) & Id_a \circ f & \rightarrow & f \\
(Idr) & f \circ Id_b & \rightarrow & f \\
(FId) & F(Id_a) & \rightarrow & Id_{F(a)} \\
(FComp) & F(f \circ g) & \rightarrow & F(f) \circ F(g)
\end{array}$$

Nous étendons l'interprétation ϕ pour prendre en compte l'opérateur F.

$$\phi(F(f)) \quad = \quad 2 \times \phi(f)$$

Les règles (FId) et $(FComp)$ correspondent respectivement aux conversions $FIdC$ et $FCompC$

Dans notre formalisme, on peut considérer F non pas comme un opérateur mais comme le paramètre de l'opérateur d'application des foncteurs. Cet opérateur correspond à `FMor`, dont voici le lemme de congruence:

```
Prf_FCong: (C,D:Category)(F:(Functor C D))(a,b:C)(f,g:a-->b)
           f =_S g -> (FMor F f) =_S (FMor F g)
```

Voilà la conversion de normalisation pour les foncteurs:

$$let\ FunC = TryC\ (SweepDnC\ FCompC)\ ThenC\ TryC\ (SweepUpC\ FIdC)\ ThenC\ CatC$$

Cette conversion normalise les termes mettant en jeu plusieurs catégories et plusieurs foncteurs; exemple: $G(F(Id_a) \circ H(f))$ se normalise en $G(H(f))$.

Il existe un autre ensemble de règles complet pour les foncteurs:

$$\begin{array}{lrcl}
(Ass) & (f \circ g) \circ g & \rightarrow & f \circ (g \circ h) \\
(Idl) & Id_a \circ f & \rightarrow & f \\
(Idr) & f \circ Id_b & \rightarrow & f \\
(FId) & F(Id_a) & \rightarrow & Id_{F(a)} \\
(FComp1) & F(f) \circ F(g) & \rightarrow & F(f \circ g) \\
(FComp2) & F(f) \circ (F(g) \circ h) & \rightarrow & F(f \circ g) \circ h
\end{array}$$

8.10.4 Cas des Transformations Naturelles

Le système de réécriture pour les transformations naturelles est le suivant:

$$\begin{array}{lrcl}
(Ass) & (f \circ g) \circ g & \rightarrow & f \circ (g \circ h) \\
(Idl) & Id_a \circ f & \rightarrow & f \\
(Idr) & f \circ Id_b & \rightarrow & f \\
(FId) & F(Id_a) & \rightarrow & Id_{F(a)} \\
(FComp1) & F(f) \circ F(g) & \rightarrow & F(f \circ g) \\
(FComp2) & F(f) \circ (F(g) \circ h) & \rightarrow & F(f \circ g) \circ h \\
(NT1) & T_a \circ G(f) & \rightarrow & F(f) \circ T_b \\
(NT2) & T_a \circ (G(f) \circ h) & \rightarrow & F(f) \circ (T_b \circ h)
\end{array}$$

Pour montrer la terminaison de ce système, nous redéfinissons l'interprétation polynomiale comme suit:

$$\begin{array}{rcl} \phi(Id_a) & = & 1 \\ \phi(f \circ g) & = & \phi(f) \times \phi(g) + \phi(f) \\ \phi(F(f)) & = & \phi(f) + 1 \\ \phi(G(f)) & = & 3 \times \phi(f) + 1 \\ \phi(T_a) & = & 1 \end{array}$$

Nous traduisons directement ce système de réécriture en conversion en utilisant $Normalize$.

$let\ NTC = Normalize\ [AssC; IdlC; IdrC; FIdC; FComp1C; FComp2C; NT1C; NT2C]$

Parfois, il est nécessaire de modifier quelque peu le but avant d'appliquer les conversions de normalisation. En effet, certains morphismes f apparaîssant dans le but correspondent à des termes de la forme $F(f)$, où F est un foncteur «concret», c'est-à-dire défini comme constante (par opposition aux foncteurs abstraits, déclarés comme variables). Dans ces cas, l'application de certains lemmes de conversion peut échouer, suite à l'échec du filtrage.
Prenons un exemple; soient C une catégorie, $F : C \to C$ un foncteur, $Id_C : C \to C$ le foncteur identité et $T : F \to Id_C$ une transformation naturelle, la conversion $(NT1)$ ne réécrit pas $T_a \circ f$ en $F(f) \circ T_b$ alors que f est convertible en $Id_C(f)$. C'est là où le combinateur $Conv$ défini à la section 6.5.2, vient à notre secours; il nous permet de réécrire tous les termes de la forme $T_a \circ f$ en $T_a \circ Id_C(f)$.

```
SweepUpC (Conv [C:Category;F:C->C;a,b:C;f:a-->b]
               ((T a) o f)  ((T a) o (FMor (Id_CAT C) f)))
```

Il nous reste à signaler les situations où on n'utilise pas les conversions de normalisation.

- lorsqu'on a à décider de égalité entre deux termes de petite taille, il est plus économique (en temps et en taille de la preuve) d'appliquer une conversion spécifique à cette égalité sans se soucier de la normalisation (souvent plus coûteuse).

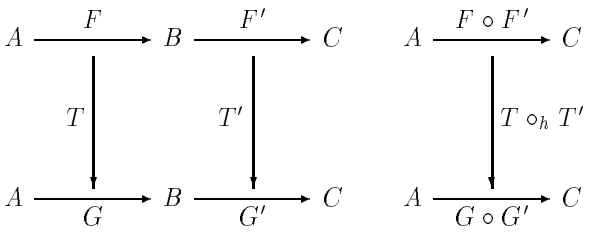
- en présence d'équations supplémentaires comme hypothèses, il est nécessaire de combiner les conversions de normalisation avec de nouvelles conversions tenant compte des hypothèses.

8.11 Exemple – La loi d'échange

8.11.1 Énoncé

Dans le but de tester nos constructions catégoriques, nous prouvons la *loi d'échange*. C'est l'une des lois des 2-catégories (catégories dont les morphismes forment une catégorie).

Soient A, B et C des catégories, $F, G : A \longrightarrow B$ et $F', G' : B \longrightarrow C$ des foncteurs. Nous définissons la *composition horizontale* des transformations naturelles $T : F \longrightarrow G$ et $T' : F' \longrightarrow G'$ par $(T \circ_h T')_a = T'_{F(a)} \circ G'(T_a)$. Nous vérifions que $T \circ_h T'$ est en effet une transformation naturelle de $F \circ F'$ vers $G \circ G'$. Nous écrirons `o_NTh` pour la composition horizontale .



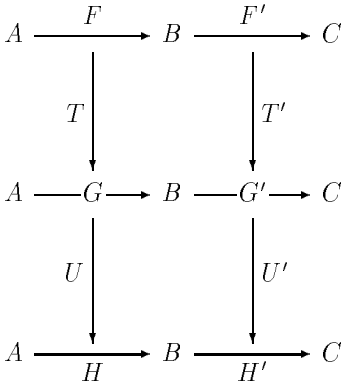
(4)

```
Section horz_comp.
Variables A,B,C:Category; F,G:(Functor A B); F',G':(Functor B C);
          T:(NT F G); T':(NT F' G').
Definition Ast : (a:A)((F o_F F') a)-->((G o_F G') a) := [a](T' (F a)) o (FMor G' (T a)).
@Definition CompH_NT := (Build_NT Ast_nt_law).
End horz_comp.
Infix 6 "o_NTh" CompH_NT.
```

Nous vérifions maintenant une propriété importante, la loi d'échange, qui lie les compositions verticale et horizontale. Soient A, B, C des catégories, $F, G, H : A \to B$, $F', G', H' : B \to C$ des foncteurs, et $T : F \to G$, $T' : F' \to G'$, $U : G \to H$, $U' : G' \to H'$ des transformations naturelles. La loi d'échange s'énonce par:

$$(T \circ_h T') \circ_v (U \circ_h U') = (T \circ_v U) \circ_h (T' \circ_v U')$$

```
Section interchangelaw.
Variables A,B,C:Category; F,G,H:(Functor A B); F',G',H':(Functor B C).
Variables T:(NT F G); T':(NT F' G'); U:(NT G H); U':(NT G' H').
Lemma InterChange_law : ((T o_NTh T') o_NTv (U o_NTh U')) =_NT
                        ((T o_NTv U) o_NTh (T' o_NTv U')).
End interchangelaw.
```



(5)

8.11.2 Sous-termes implicites

À titre de comparaison, et pour montrer l'importance du mécanisme des sous-termes implicites, nous donnons l'énoncé explicite de la loi d'échange.

```
(Equal_NT A C (Comp_Functor A B C F F') (Comp_Functor A B C H H')
          (CompV_NT A C (Comp_Functor A B C F F') (Comp_Functor A B C G G')
                    (Comp_Functor A B C H H') (CompH_NT A B C F G F' G' T T')
                    (CompH_NT A B C G H G' H' U U'))
          (CompH_NT A B C F H F' H' (CompV_NT A B F G H T U)
                    (CompV_NT B C F' G' H' T' U')))
```

Nous pouvons aussi écrire l'énoncé de la loi d'échange en remplaçant `o_NTv` par `o''` et `=_NT` par `=_S''`. Il devient alors:

```
((T o_NTh T') o'' (U o_NTh U')) =_S'' ((T o' U) o_NTh (T' o'' U'))
```

En effet `o''` (ou bien `Comp''`) est le nom générique pour toutes les compositions dans les catégories `Category''`, alors que `=_S''` (ou bien `Equal''`) est le nom générique pour toutes les égalités dans les Setoïdes `Setoid''`. En fait, il s'agit du mécanisme de surcharge défini dans la section 4.7. Nous donnons les principales étapes de la procédure d'inférence de type sur le terme `T o'' U`. C'est un exemple intéressant car il montre comment peuvent s'enchaîner deux étapes de résolution par la règle d'unification de la définition 4.19.

Nous avons à typer ($\texttt{Comp}''\ ?_1\ ?_2\ ?_3\ ?_4\ \texttt{T}\ \texttt{U}$). Rappelons le type (explicite) de `Comp''`:

```
(C:Category'')(a,b,c:(Ob'' C))(Carrier'' (Hom'' a b)) ->
             (Carrier'' (Hom'' b c)) -> (Carrier'' (Hom'' a c))
```

Nous avons la signature $\Gamma \vdash ?_1 : \texttt{Category}''$; $\Gamma \vdash ?_2 : (\texttt{Ob}''\ ?_1)$; $\Gamma \vdash ?_3 : (\texttt{Ob}''\ ?_1)$; $\Gamma \vdash ?_4 : (\texttt{Ob}''\ ?_1)$. La première unification que nous rencontrons intervient lors du typage de `T`:

$$(\texttt{NT A B F G}) = (\texttt{Carrier}''\ (\texttt{Hom}''\ ?_1\ ?_2\ ?_3))$$

Elle est simplifiée en utilisant la règle (4.19) avec (voir la définition de `NT_setoid` page 147):

$$\mathsf{Proj}(\texttt{Carrier}'', \texttt{NT}) \equiv \texttt{NT\_setoid}$$

Nous obtenons: $\begin{cases} (\texttt{Hom}''\ ?_1\ ?_2\ ?_3) = (\texttt{NT\_setoid}\ ?_5\ ?_6\ ?_7\ ?_8) \\ ?_5 = \texttt{A} \\ ?_6 = \texttt{B} \\ ?_7 = \texttt{F} \\ ?_8 = \texttt{G} \end{cases}$

avec $\Gamma \vdash ?_5 : \texttt{Category}$; $\Gamma \vdash ?_6 : \texttt{Category}$; $\Gamma \vdash ?_7 : (\texttt{Functor}\ ?_5\ ?_6)$; $\Gamma \vdash ?_8 : (\texttt{Functor}\ ?_5\ ?_6)$

Seule la première équation est difficile à résoudre. On utilise la même règle (4.19) avec (voir la définition de `FUNCT` page 147):

$$\mathsf{Proj}(\texttt{Hom}'', \texttt{NT\_setoid}) \equiv \texttt{FUNCT}$$

Nous obtenons: $\begin{cases} ?_1 = (\texttt{FUNCT}\ ?_9\ ?_{10}) \\ ?_9 = ?_5 \\ ?_{10} = ?_6 \\ ?_2 = ?_7 \\ ?_3 = ?_8 \end{cases}$

avec $\Gamma \vdash ?_9 : \texttt{Category}$; $\Gamma \vdash ?_{10} : \texttt{Category}$

Nous obtenons finalement la signature:

$$\Gamma \vdash ?_1 : \texttt{Category}'' := (\texttt{FUNCT A B});\ \Gamma \vdash ?_2 : (\texttt{Ob}''\ ?_1) := \texttt{F};$$
$$\Gamma \vdash ?_3 : (\texttt{Ob}''\ ?_1) := \texttt{G};\ \Gamma \vdash ?_4 : (\texttt{Ob}''\ ?_1); \ldots$$

La valeur de $?_4$ ($:= \texttt{H}$) est obtenue par analyse de `U`. Le terme explicite obtenu est alors:

$$(\texttt{Comp}''\ (\texttt{FUNCT A B})\ \texttt{F G H T U})$$

Un autre exemple Considérons un exemple plus simple; celui de la définition de la propriété d'idempotence pour les applications. Remarquez que dans cet exemple, les applications sont composées en utilisant la composition dans les catégories, et comparées avec l'égalité des Setoïdes.

```
Definition idemp := [A:Setoid][f:(Map A A)] (f o f) =_S f.
```

La forme explicite de `(f o f) =_S f` est `(Equal (Map_setoid A A) (Comp SET A A A f f) f)`. L'heuristique (4.19) est utilisée plusieurs fois:

- pour «se souvenir» que la «composition catégorique» d'applications peut s'effectuer dans la catégorie `SET`, en vertu de la définition ci-dessous (page 139, donnée ici sous forme explicite):

  ```
  @Definition SET := (!Build_Category Setoid Map_setoid Comp_SET Id_SET
                                      Assoc_SET Idl_SET Idr_SET).
  ```

- pour comparer deux applications dans le Setoïde approprié, `(Map_setoid A A)` (page 135).

  ```
  @Definition Map_setoid := [A,B:Setoid](!Build_Setoid (Map A B) (Ext_equiv A B)).
  ```

Ces deux constantes sont automatiquement trouvées, lors du processus d'unification, par la fonction `Proj` avec respectivement $\texttt{Proj}(\texttt{Hom}, \texttt{Map\_setoid}) \equiv \texttt{SET}$ et $\texttt{Proj}(\texttt{Carrier}, \texttt{Map}) \equiv \texttt{Map\_setoid}$.

8.11.3 Preuve par réécriture

Nous commentons maintenant la preuve du but de la loi d'échange dans Coq:

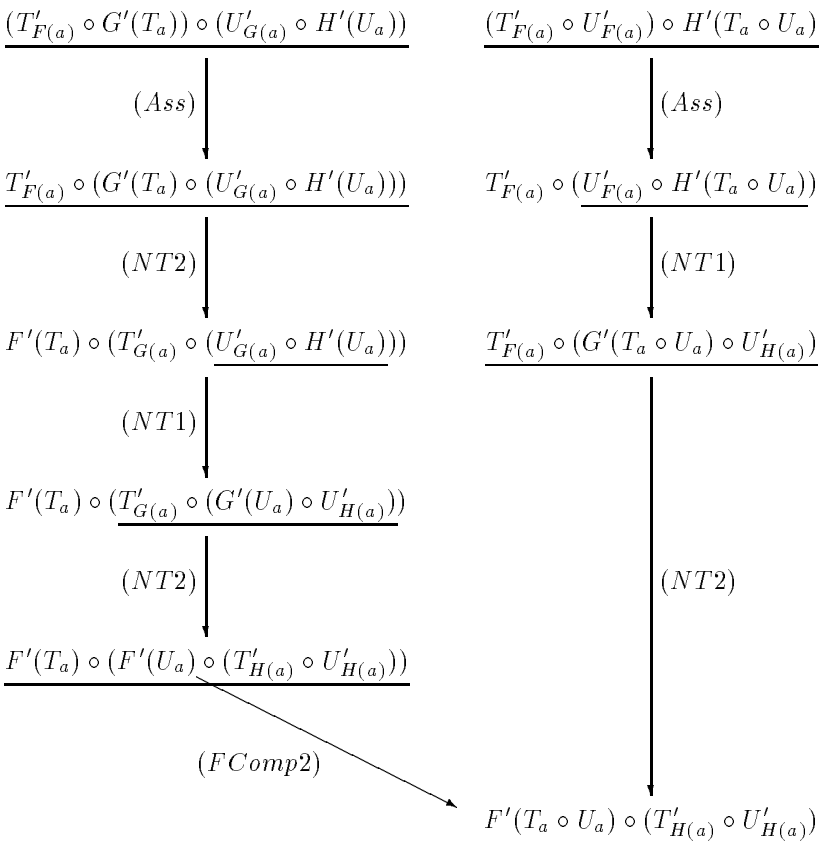
```
============================
 (T o_NTh T') o_NTv (U o_NTh U') =_NT (T o_NTv U) o_NTh (T' o_NTv U')
```

Après expansion des différentes constantes (`=_NT`, `o_NTv` et `o_NTh`), nous obtenons le but:

```
 a : A
============================
 ((T' (F a)) o (FMor G' (T a))) o ((U' (G a)) o (FMor H' (U a)))
  =_S ((T' (F a)) o (U' (F a))) o (FMor H' ((T a) o (U a)))

InterChange_law < SimplEq NTC NTC.
Subtree proved!
```

Ce but est prouvé par la tactique *SimplEq* en effectuant les réécritures suivantes:




```
End interchangelaw.
```

8.11.4 Associativité de la composition horizontale

Nous terminons cette section en montrant que la composition horizontale est associative. Là intervient une difficulté.

Étant donnés les catégories A, B, C, les foncteurs F, G de A vers B et les foncteurs F', G' de B vers C, la composition horizontale des transformations naturelles $T : F \to G$ et $T' : F' \to G'$ a pour résultat une transformation naturelle $T \circ_h T' : F \circ F' \to G \circ G'$. Pour exprimer l'associativité de la composition horizontale, nous devons identifier, comme types, disons $(F \circ F') \circ F''$ et $F \circ (F' \circ F'')$. Mais là nous rencontrons un problème, puisque ces deux termes sont égaux dans le sens de l'égalité de foncteurs, mais ne sont pas définitionnellement égaux. Nous sommes alors incapables même d'écrire l'énoncé d'associativité de la composition horizontale: l'énoncé est mal typé.

Pour contourner le problème, à l'instar de 8.6, nous définissons une égalité entre transformations moins contraignante `=_NTH`, comme suit.

```
Definition EqualH_NT := [C,D:Category][F,G,F',G':(Functor C D)]
                        [T:(NT F G)][T':(NT F' G')](a:C)(T a) =_H (T' a).
Infix 2 "=_NTH" EqualH_NT.

Section assoc_horz_comp.
```

```
Variables A,B,C,D:Category; F,G:(Functor A B);
          F',G':(Functor B C); F'',G'':(Functor C D).
Variables T:(NT F G); T':(NT F' G'); T'':(NT F'' G'').
Lemma CompH_NT_assoc : ((T o_NTh T') o_NTh T'') =_NTH (T o_NTh (T' o_NTh T'')).
End assoc_horz_comp.
```

Chapitre 9

Constructions Universelles et Adjonctions

Nous formalisons dans ce chapitre des notions plus avancées de la théorie des catégories telles que les limites et les adjonctions. Le principal théorème vérifié dans cette partie est le théorème de Freyd pour l'existence d'un adjoint gauche[61]. C'est une preuve non-triviale qu'on présente de manière «modulaire», nécessitant la preuve de plusieurs résultats intermédiaires intéressants. C'est un bon test pour notre formalisation car le théorème met en jeu la plupart des notions catégoriques formalisées.

9.1 Constructions Universelles et Limites

9.1.1 Morphisme Universel

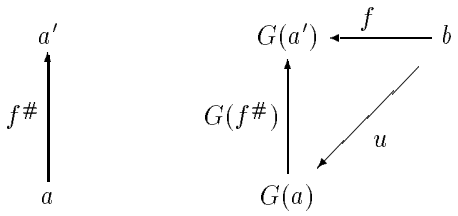
La définition par universalité est une technique usuelle en mathématiques qui, en théorie des catégories, fournit la principale façon de caractériser des structures. Son mécanisme général consiste à définir une classe d'objets vérifiant une propriété puis un critère pour distinguer un objet particulier de cette classe; les deux parties étant toujours décrites en termes de morphismes. La forme standard du critère pour distinguer un objet particulier est l'unicité d'un morphisme vérifiant certaines conditions données.

L'énoncé de plusieurs constructions catégoriques a la forme suivante «pour tout...il existe un unique...tel que...». C'est notamment le cas des notions de produit, égalisateur, objet terminal et objet initial. Toutes ces constructions sont des cas particuliers d'un concept général, le morphisme universel et son dual, le morphisme co-universel.

Soient A et B deux catégories, $b : B$ et $F : A \to B$ un foncteur.

```
Section ua_def.
Variables A,B:Category; b:B; G:(Functor A B).
```

Un couple $\langle a : A, u : b \to G(a) \rangle$ est dit *morphisme universel* de b vers G si pour tout a' et tout $f : b \to G(a')$, il existe un unique $f^{\#} : a \to a'$ tel que le diagramme ci-dessous commute.



(1)

```
Section isua_def.
Variables a:A; u:b-->(G a).
```

On va définir un morphisme universel par la donnée d'une opération $\_^{\#}$ qui associe à tout morphisme $f : b \to G(a')$ un morphisme $f^{\#} : a \to a'$.

```
  Section ua_laws.
  Variable diese:(a':A)b-->(G a') -> a-->a'.
```

Cette opération doit vérifier deux équations. La première traduit la commutation du diagramme et la deuxième, l'unicité de $f^{\#}$.

$$f = u \circ G(f^{\#}) \tag{2}$$

$$\forall g : a \to a', f = u \circ G(g) \Rightarrow g = f^{\#} \tag{3}$$

```
  Definition UA_eq := [a':A][f:b-->(G a')][g:a-->a'] (u o (FMor G g)) =_S f.
  Definition UA_law1 := (a':A)(f:b-->(G a'))(UA_eq f (diese f)).
  Definition UA_law2 := (a':A)(f:b-->(G a'))(g:a-->a')(UA_eq f g) -> g =_S (diese f).
  End ua_laws.
```

Toutes ces données sont les conditions nécessaires et suffisantes pour affirmer que $\langle a, u \rangle$ est un morphisme universel de b vers G. On l'écrira `(IsUA a u)`.

```
 Structure IsUA : Type :=
  {UA_diese      : (a':A)b-->(G a') -> a-->a';
   Prf_isUA_law1 : (UA_law1 UA_diese);
   Prf_isUA_law2 : (UA_law2 UA_diese)}.
 End isua_def.
```

On définit aussi le «Type des morphismes universels de b vers G», noté `(UA b G)`.

```
Structure >UA : Type :=
 {UA_ob    :  A;
  UA_mor   :  b-->(G UA_ob);
  Prf_IsUA :> (IsUA UA_mor)}.
End ua_def.
```

La définition par universalité définit des objets à isomorphisme près. Si $\langle a_1, u_1 \rangle$ et $\langle a_2, u_2 \rangle$ sont des morphismes universels de b vers G, alors $a_1 \cong a_2$. Les morphismes isos entre a_1 et a_2 sont $u_1^{\#_2}$ et $u_2^{\#_1}$.

```
Section ua_iso_def.
Variables A,B:Category; b:B; G:(Functor A B); u1,u2:(UA b G).
Definition UA_iso_mor := (UA_diese u1 (UA_mor u2)).
Definition UA_iso_mor_1 := (UA_diese u2 (UA_mor u1)).
Lemma UA_iso_law1 : (AreIsos UA_iso_mor UA_iso_mor_1).
Definition UA_iso : (Iso (UA_ob u1) (UA_ob u2)) := UA_iso_law1.
End ua_iso_def.
```

C'est une caractéristique importante de la théorie des catégories: les objets sont vus de manière abstraite, indépendamment d'une représentation particulière. Dans la suite du développement, lorsque nous parlons d'un morphisme universel `x:(UA b G)`, il s'agit d'un représentant quelconque de la classe d'objets isomorphes; son choix exact n'a aucune importance. Par contre, lors de la construction d'un morphisme universel, un représentant unique est construit.

9.1.2 Définition d'un Morphisme Co-universel

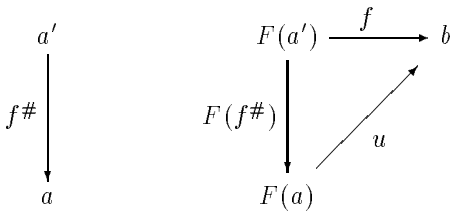
La théorie des catégories admet un principe de dualité, exprimé par l'invariance de la théorie par la modification de la direction des morphismes. Ainsi toute propriété catégorique a son duale obtenue en inversant tous les morphismes dans l'énoncé de la propriété. Ce principe de dualité peut être décrit en fonction d'opérations sur les catégories et les foncteurs. Nous avons déjà défini le dual d'une catégorie en 8.3; on définit le dual d'un foncteur $F : A \to B$, par $F^\circ(a) = F(a)$ et $F^\circ(f^\circ) = F(f)^\circ$.

```
Section dfunctor_def.
Variables A,B:Category; F:(Functor A B).
Definition DFunctor_ob : (Dual A) -> (Dual B) := [a:A](F a).
Definition DFunctor_map := [b,a:A](FMap F a b).
@Definition Dfunctor := (Build_Functor DFunctor_comp_law DFunctor_id_law).
End dfunctor_def.
```

Toute propriété ou construction catégorique peut alors être transformée en son duale en l'interprétant dans la catégorie duale. Pour illustrer ce principe, nous allons définir la notion duale de morphisme universel, que nous appelons *morphisme co-universel de b vers F* (cette appellation n'est pas très répandue, les catégoriciens préfèrent plutôt parler de morphisme universel de F vers b). Le couple $\langle a : A, u : F(a) \to b\rangle$ est dit morphisme co-universel de b vers F si et seulement si $\langle a : A^\circ, u : b \to F^\circ(a)\rangle$ est un morphisme universel de b vers F°. Remarquez la double «nature» de u, c'est à la fois un morphisme dans B et B°; ces deux types, respectivement $Hom_B(F(a), b)$ et $Hom_{B^\circ}(b, F^\circ(a))$ sont bien entendu convertibles.

```
Section coua_def1.
Variables A,B:Category; b:B; F:(Functor A B).
SubClass IsCoUA1 := [a:A][u:(F a)-->b](IsUA 4!(Dfunctor F) u).
SubClass CoUA1 := (UA b (Dfunctor F)).
End coua_def1.
```

On peut définir une notion basique de morphisme co-universel comme suit. Un couple $\langle a : A, u : F(a) \to b\rangle$ est dit co-morphisme universel de b vers F si pour tout a' et tout $f' : F(a') \to b$, il existe un unique $f^\# : a' \to a$ tel que:



(4)

Remarquez que ce diagramme est identique à celui du diagramme (1) après modification du sens des morphismes.

```
Section coua_def.
Variables A,B:Category; b:B; F:(Functor A B).
Structure IsCoUA [a:A;u:(F a)-->b]: Type :=
 {CoUA_diese      :> (a':A)(F a')-->b -> a'-->a;
  Prf_isCoUA_law1 :  (CoUA_law1 CoUA_diese);
  Prf_isCoUA_law2 :  (CoUA_law2 CoUA_diese)}.
Structure >CoUA : Type :=
 {CoUA_ob    :  A;
  CoUA_mor   :  (F CoUA_ob)-->b;
  Prf_IsCoUA :> (IsCoUA CoUA_mor)}.
End coua_def.
```

Il est possible de montrer l'équivalence des deux versions (duale et basique) de la notion de co-morphisme universel. On pourra alors utiliser indifféremment l'une ou l'autre des versions sans nous soucier du typage, en utilisant deux coercions opposées.

9.1.3 Limites et Colimites

Limites et colimites donnent un traitement unifié des constructions comme les produits et les égalisateurs (et bien d'autres encore). D'abord deux définitions préliminaires.

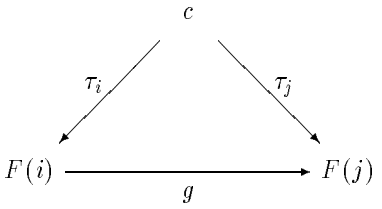
1. Soient A et B deux catégories. La fonction Δ associe à tout objet $b : B$ le foncteur constant $\Delta(b) : A \to B$ défini par $\Delta(b)(a) = b$ pour tout $a : A$, et $\Delta(b)(f) = Id_b$ où $f : a \to a'$ et $a, a' : A$.

   ```
   Section constFun.
   Variables A,B:Category; b:B.
   Definition Const_ob := [a:A]b.
   Definition Const_mor := [a1,a2:A][f:a1-->a2](Id b).
   @Definition Const := (Build_Functor Const_comp_law Const_id_law).
   End constFun.
   ```

2. On appelle *cône* de c verc F toute transformation naturelle de $\Delta(c)$ vers le foncteur F.

   ```
   Section def_cone.
   Variables J,C:Category; c:C; F:(Functor J C).
   SubClass Cone := (NT (Const J c) F).
   ```

Soit $\tau : \Delta(c) \to F$ un cône. On a $\tau_i : c \to F(i)$ pour tout $i : J$. Sa loi de naturalité se ramène à la commutation du diagramme suivant pour tous $i, j : J$ et $g : i \to j$.

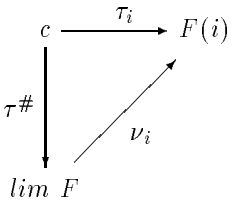


(5)

Le cône τ est ainsi complètement déterminé par la donnée d'une famille de morphismes $\{\tau_i : c \to F(i) | i : J\}$ vérifiant (5).

```
 Section cone_nt.
 Variable T:(a:J)c-->(F a).
 Definition Cone_law := (i,j:J)(g:i-->j)(T j) =_S ((T i) o (FMor F g)).
 Lemma Prf_Cone_nt_law : Cone_law -> (NT_law 3!(Const J c) T).
 Variable p:Cone_law.
 @Definition Build_Cone : Cone := (Build_NT (Prf_Cone_nt_law p)).
 End cone_nt.
End def_cone.
```

La *limite* d'un foncteur $F : J \to C$ est toute paire $\langle lim\ F, \nu \rangle$ composée d'un objet $lim\ F$, appelé *objet limite de* F, et d'un cône $\nu : \Delta(lim\ F) \to F$, appelé *cône limite* de F, et vérifiant la loi d'universalité suivante: pour tout cône $\tau : \Delta(c) \to F$, il existe un unique morphisme $\tau^{\#} : c \to lim\ F$ vérifiant le diagramme ci-dessous pour tout $i : J$.



(6)

```
Section limit_def.
Variables J,C:Category; F:(Functor J C).
 Section islimit_def.
 Variables lim:C; nu:(Cone lim F).
  Section limit_laws.
  Variable l_diese:(c:C)(Cone c F) -> (c-->lim).
  Definition Limit_eq := [c:C][tau:(Cone c F)][g:c-->lim]
                         (i:J)(g o (nu i)) =_S (tau i).
  Definition Limit_law1 := (c:C)(tau:(Cone c F))(Limit_eq tau (l_diese tau)).
  Definition Limit_law2 := (c:C)(tau:(Cone c F))(g:c-->lim)
                           (Limit_eq tau g) -> g =_S (l_diese tau).
  End limit_laws.
 Structure IsLimit : Type :=
  {Lim_diese     : (c:C)(Cone c F) -> (c-->lim);
```

```
   Prf_limit1    : (Limit_law1 Lim_diese);
   Prf_limit2    : (Limit_law2 Lim_diese)}.
 End islimit_def.
 Structure >Limit : Type :=
  {Lim           : C;
   Limiting_cone : (Cone Lim F);
   Prf_Islimit   :> (IsLimit Limiting_cone)}.
End limit_def.
```

Une catégorie C est dite *complète* si tout foncteur $F : J \to C$ admet une limite.

```
Definition Complete := [C:Category](J:Category)(F:(Functor J C))(Limit F).
```

Il aurait été plus élégant de définir la notion de limite comme un cas particulier de morphisme co-universel. En effet, la limite d'un foncteur $F : J \to C$ est un morphisme co-universel $\langle lim\ F, \nu \rangle$ du foncteur Δ vers F. Le foncteur $\Delta : C \to FUNCT(J,C)$ est obtenu en étendant la fonction Δ (définie au début de cette section) au cas des morphismes par: pour tout $f : c_1 \to c_2$ de C, $\Delta(f) : \Delta(c_1) \to \Delta(c_2)$ est une transformation naturelle avec les composantes $\Delta(f)_i = f$. Cette définition est mal typée puisque C est de type `Category`, alors que $FUNCT(J,C)$ est de type `Category''`. Elle est toutefois possible au prix de nombreuses duplications, notamment en définissant une nouvelle notion de foncteur de type `Catgory->Category''->Type`.

Il existe une autre alternative à notre définition de limite consistant à utiliser une notion de *diagramme* (un graphe particulier, voir [106] et [137]) à la place du foncteur J. L'utilisation de foncteurs nous évite l'introduction de ce nouveau concept. De plus, il est d'usage de considérer J comme une catégorie qui est petite ou même finie. Nous ne faisons pas cette hypothèse car c'est une restriction inutile puisque tous les résultats que nous avons prouvés sont valables pour une catégorie J quelconque.

9.1.4 Préservation de Limites

La composition d'un cône $T : \Delta(r) \to F$ avec un foncteur $H : C \to D$ est un cône $T \circ H : \Delta(H(r)) \to F \circ H$ défini par $(T \circ H)_i = H(T_i)$ pour tout $i : J$. C'est une instance de la composition horizontale (8.11).

```
Section comp_cone.
Variables J,C,D:Category; c:C; F:(Functor J C); T:(Cone c F); G:(Functor C D).
Definition Comp_cone_tau : (i:J)(G c) --> ((F o_F G) i) := [i](FMor G (T i)).
Lemma Comp_cone_tau_cone_law : (Cone_law Comp_cone_tau).
Definition Comp_cone := (Build_Cone Comp_cone_tau_cone_law).
End comp_cone.
Infix 6 "o_C" Comp_cone.
```

On dit qu'un foncteur $H : C \to D$ préserve les limites d'un foncteur $F : J \to C$ lorsque tout cône limite de F est transformé par composition avec H en un cône limite de $F \circ H$, c'est-à-dire si $\langle lim\ F, \nu \rangle$ est une limite de F alors $\langle H(lim\ F), \nu \circ H \rangle$ est une limite de $F \circ H$.

```
Section def_pres_limits.
Variables J,C,D:Category; F:(Functor J C); G:(Functor C D).
```

```
Definition Preserves_1limit := [l:(Limit F)](IsLimit ((Limiting_cone l) o_C G)).
Definition Preserves_limits := (l:(Limit F))(Preserves_1limit l).
End def_pres_limits.
```

Un foncteur $H : C \to D$ est dit *continu* s'il préserve toutes les limites de tous les foncteurs $F : J \to C$.

```
Definition Continuous := [C,D:Category][G:(Functor C D)]
                         (J:Category)(F:(Functor J C))(Preserves_limits F G).
```

9.1.5 Exemples de limites

Dans cette section, nous étudions quelques limites intéressantes correspondant à des constructions usuelles. Dans chacun des cas étudiés, nous donnons deux définitions; la première est purement équationnelle, alors que la deuxième s'obtient comme limite d'un foncteur particulier. Ces deux définitions sont équivalentes; dans cette section, nous relions la définition par les limites à la définition équationnelle.

9.1.5.1 Produits

Cette construction apparaît sous différents aspects en théorie des ensembles, en algèbre et en topologie. Nous pouvons, par exemple, former le produit de deux ensembles, de deux groupes ou de deux espaces topologiques. Le détail des constructions varie, mais la forme générale du produit est commune à tous les cas. Les produits sont décrits de manière abstraite en terme de projections du produit vers ses composantes.

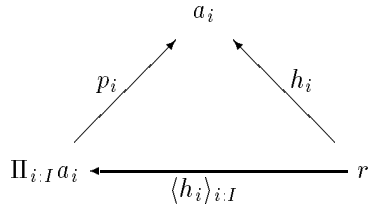
Définition équationnelle Pour tous objets $a, b : C$, leur produit est un objet $a \times b$ avec deux morphismes $p_1 : a \times b \to a$ et $p_2 : a \times b \to b$. On peut généraliser la définition au cas du produit d'une famille d'objets. Soit C une catégorie et $\{a_i : C \mid i : I\}$ une famille d'objets de C.

```
Section prodts_def.
Variables C:Category; I:Type; a:I -> C.
```

Le produit catégorique de cette famille est un objet $\Pi_{i:I} a_i$ avec une famille de morphismes $\{p_i : \Pi_{i:I} a_i \to a_i \mid i : I\}$.

```
 Section prodts_laws.
 Variables PI:C; Proj:(i:I) PI-->(a i).
```

Pour toute autre famille $\{h_i : r \to a_i \mid i : I\}$, il doit exister un unique morphisme $\langle h_i \rangle_{i:I}$ tel que pour tout $i : I$:



(7)

Nous décomposons cette condition en une condition d'existence et d'unicité .

```
 Variable Pd_diese:(r:C)((i:I)r-->(a i)) -> r-->PI.
 Definition Product_eq := [r:C][h:(i:I)r-->(a i)][g:r-->PI](i:I)(h i) =_S (g o (Proj i)).
 Definition Product_law1 := (r:C)(h:(i:I) r-->(a i))(Product_eq h (Pd_diese h)).
 Definition Product_law2 := (r:C)(h:(i:I) r-->(a i))(g:r-->PI)
                            (Product_eq h g) -> g =_S (Pd_diese h).
 End prodts_laws.
```

Ce sont toutes ces données rangées dans une structure qui forment le type «Produit de la famille $\{a_i : C \mid i : I\}$».

```
Structure Product : Type :=
 {Pi          : C;
  Proj        : (i:I) Pi-->(a i);
  Pd_diese    : (r:C)((i:I)r-->(a i)) -> r-->Pi;
  Prf_pd_law1 : (Product_law1 Proj Pd_diese);
  Prf_pd_law2 : (Product_law2 Proj Pd_diese)}.
End prodts_def.
```

Pour définir le produit de deux objets, il suffit d'instancier notre définition des produits de famille d'objets par la famille ne contenant que ces deux objets.

Définition par les limites On peut construire une catégorie à partir d'un ensemble quelconque U. C'est la catégorie discrète `(Discr U)` associée à cet ensemble. Elle est notée par $\hat{U}$.

```
Section discrete.
Variable U:Type.
```

Cette catégorie a pour objets les éléments de U et pour seuls morphismes, les identités Id_a pour tout $a : U$.

```
Inductive Discr_mor : U -> U -> Type := Build_Discr_mor : (a:U)(Discr_mor a a).
@Definition Discr := (Build_Category Assoc_Discr Idl_Discr Idr_Discr).
End discrete.
```

La définition `Discr_mor` est semblable à celle de l'égalité de Leibniz (sauf en ce qui concerne la disposition des paramètres, et la sorte de la définition). On avait déjà dit en 8.6 que l'axiome K pour l'égalité de Leibniz, stipulant que la preuve de réflexivité est unique, n'était pas démontrable dans le calcul des constructions mais que son ajout conservait la cohérence de la théorie. Pour la suite du développement, nous déclarons un axiome similaire pour `Discr_mor` énonçant que Id_a (soit `(Build_Discr_mor a)`) est l'unique morphisme de a vers a.

```
Axiom Discr_mor_ind1 : (U:Type)(a:U)(P:(Discr_mor a a)->Prop)
                       (P (Build_Discr_mor a)) -> (x:(Discr_mor a a))(P x).
```

Soit I un ensemble, C une catégorie et $\{a_i : C \mid i : I\}$ une famille d'objets de C indicée par I.

```
Section products_limit_def.
Variables C:Category; I:Type; a:I -> C.
```

Nous construisons un foncteur $F : \hat{I} \to C$ par $F(i) = a_i$ pour tout $i : I$, et $F(f) = Id_{a_i}$ pour $i, j : I$ et $f : i \to j$, car f ne peut être que l'identité, i et j étant identiques. Remarquez l'utilisation de l'opérateur de filtrage `Cases` sur un objet de type dépendant.

```
Definition FunDiscr_ob := [i:(Discr I)](a i).
Definition FunDiscr_arrow := [i,j:(Discr I)][f:i-->j]
                             <[d,d0:I][_:(Discr_mor d d0)](Carrier (a d)-->(a d0))>
                             Cases f of (Build_Discr_mor k) => (Id (a k)) end.
@Definition FunDiscr := (Build_Functor FunDiscr_comp_law FunDiscr_id_law).
```

Le produit de la famille $\{a_i : C \mid i : I\}$ d'objets de C est donné par la limite du foncteur F.

```
SubClass Product1 := (Limit FunDiscr).
```

Nous vérifions les lois des produits pour cette définition. Soit l un produit, limite du foncteur F ci-dessus. Ce produit est de la forme $\langle \Pi_{i:I} : C, \nu : \Delta(\Pi_{i:I}) \to F \rangle$. Considérons $\Pi_{i:I}$ comme l'objet produit de la famille $\{a_i : C \mid i : I\}$. Quant aux projections $p_i : \Pi_{i:I} \to a_i$, elles correspondent à ν_i.

```
Variable l:Product1.
Definition Pi1 := (Lim l).
Definition Proj1 : (i:I)Pi1-->(a i) := [i]((Limiting_cone l) i).
```

Vérifions maintenant la propriété d'unicité pour ce produit. Soit $f_i : c \to a_i$ une autre famille de projections d'un objet $c : C$. On construit le morphisme $\langle f \rangle_{i:I}$ en s'appuyant sur l'universalité de l. Plus précisément, $\langle f \rangle_{i:I} = \tau^{\#}$ avec $\tau : \Delta(c) \to F$ et $\tau_i = h_i$.

```
 Section pd1_diese_def.
 Variables c:C; f:(j:I)c-->(FunDiscr j).
 Definition Product_tau := [j:(Discr I)](f j).
 Definition Pd1_diese : c-->Pi1 := (Lim_diese l Product_nt).
 End pd1_diese_def.
@Definition Product1_to_Product := (Build_Product Prf_pd1_law1 Prf_pd1_law2).
End products_limit_def.
Coercion Product1_to_Product : Product1 >-> Product.
```

9.1.5.2 Égalisateurs

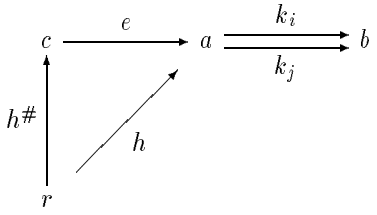
Définition équationnelle Soient C une catégorie, $a, b : C$ et $\{k_i : a \to b \mid i : I\}$ une famille de morphismes de C.

```
Section equaz_def.
Variables C:Category; a,b:C; I:Type; k:I -> a-->b.
```

Un égalisateur de cette famille est un morphisme $e : c \to a$ tel que $\forall i, j : I, e \circ k_i = e \circ k_j$.

```
 Section equaz_laws.
 Variables c:C; e:c-->a.
 Definition Equalizer_eq := [r:C][h:r-->a](i,j:I) (h o (k i)) =_S (h o (k j)).
 Definition Equalizer_law1 := (Equalizer_eq e).
```

De plus, pour tout morphisme $h : r \to a$ tel que $\forall i, j : I, h \circ k_i = h \circ k_j$, il doit exister un unique $h^{\#} : r \to c$ avec $h = h^{\#} \circ e$.



(8)

```
 Variable E_diese:(r:C)(h:r-->a)(Equalizer_eq h) -> (r-->c).
 Definition Equalizer_law2 := (r:C)(h:r-->a)(p:(Equalizer_eq h)) h =_S ((E_diese p) o e).
 Definition Equalizer_law3 := (r:C)(h:r-->a)(p:(Equalizer_eq h))(l:r-->c)
                              (h =_S (l o e)) -> l =_S (E_diese p).
 End equaz_laws.
```

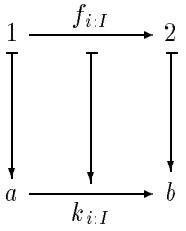
D'où le type «Égalisateur de la famille $\{k_i : a \to b \mid i : I\}$» .

```
Structure Equalizer : Type :=
 {E_ob       : C;
  E_mor      : E_ob-->a;
  Prf_E_law1 : (Equalizer_law1 E_mor);
  E_diese    : (r:C)(h:r-->a)(Equalizer_eq h) -> r-->E_ob;
  Prf_E_law2 : (Equalizer_law2 E_mor E_diese);
  Prf_E_law3 : (Equalizer_law3 E_mor E_diese)}.
End equaz_def.
```

Définition par les limites La catégorie «morphismes parallèles» U comporte deux objets notés 1 et 2 et une famille de morphismes $\{f_i : 1 \to 2 \mid i : I\}$.

```
Section pa_def.
Variable I:Type.
Inductive PA_ob : Type := PA1 : PA_ob | PA2 : PA_ob.
Inductive PA_mor : PA_ob -> PA_ob -> Type :=
   PA_I1 : (PA_mor PA1 PA1)
 | PA_I2 : (PA_mor PA2 PA2)
 | PA_f  : I -> (PA_mor PA1 PA2).
@Definition PA := (Build_Category Assoc_PA Idl_PA Idr_PA).
End pa_def.
```

Soient C une catégorie, $a, b : C$ et $\{k_i : a \to b \mid i : I\}$ une famille de morphismes de C. Nous construisons un foncteur $F : C \to U$ suivant la correspondance suivante:



```
Section equalizer_limit_def.
Variables C:Category; a,b:C; I:Type; k:I -> a-->b.
Definition FPA_ob := [x:(PA k)]Cases x of PA1 => a
                                       | PA2 => b end.
Definition FPA_mor := [x,y:(PA k)][f:x-->y]<[x',y':(PA k)](Carrier (FPA_ob x')-->(FPA_ob y'))>
                      Cases f of PA_I1    => (Id a)
                                 | PA_I2    => (Id b)
                                 | (PA_f i) =>  (k i) end.
@Definition FPA := (Build_Functor FPA_comp_law FPA_id_law).
```

L'égalisateur de la famille $\{k_i : a \to b \mid i : I\}$ est alors obtenu comme limite de F.

```
SubClass Equalizer1 := (Limit FPA).
```

Les deux définitions qu'on vient de voir ne sont pas en fait strictement équivalentes. Elles le sont pour des familles non vides de morphismes. Il nous faut alors supposer que l'ensemble I est non vide; il suffit pour cela de supposer l'existence d'un terme «témoin» (`Witness`) t de type I. Nous appelons `equaliseur2` le type des égalisateurs de familles non vides.

```
Structure Equalizer2 : Type :=
 {Prf_equalizer1 :> Equalizer1;
  Witness        :  I}.
```

Partant d'un tel égalisateur $\langle c : C, \nu : \Delta(c) \to F\rangle$, nous construisons un «égalisateur équationnel» en posant $e = \nu_1 : c \to a$. Pour tout $h : r \to a$ tel que $\forall i, j : I, h \circ k_i = h \circ k_j$, $h^{\#}$ est défini par $h^{\#} = \tau^{\#}$ où $\tau : \Delta(r) \to F$ est un cône défini par $\tau_1 = h$ et $\tau_2 = h \circ k_t$. Cette construction dépend ainsi du témoin choisi .

```
Variable l:Equalizer2.
Definition E1_ob := (Lim l).
Definition E1_mor : E1_ob-->a := ((Limiting_cone l) PA1).
 Section e1_diese_def.
 Variables r:C; h:r-->a.
 Hypothesis p:(Equalizer_eq k h).
 Definition E_tau := [x:(PA k)]<[x':(PA k)](Carrier r-->(FPA x'))>
                      Cases x of PA1 => h
                                 | PA2 => (h o (k (Witness l))) end.
 Definition E1_diese : r-->E1_ob := (Lim_diese l E_NT).
 End e1_diese_def.
@Definition Equalizer2_to_Equalizer :=
            (Build_Equalizer Prf_E1_law1 Prf_E1_law2 Prf_E1_law3).
End equalizer_limit_def.
Coercion Equalizer2_to_Equalizer : Equalizer2 >-> Equalizer.
```

Nous distinguons deux cas particuliers intéressants qui nous seront utiles plus loin, correspondants à l'égalisateur de deux morphismes de C (`Equalizer1_fg`), et à l'égalisateur de tous les morphismes entre deux objets de C (`Equalizer1_hom`).

```
Section equaz_fg_hom.
Variables C:Category; a,b:C; f,g:a-->b.
```

```
Definition K_fg := [i:TwoElts]Cases i of  Elt1 => f
                                        | Elt2 => g end.
Definition F_fg := (FPA K_fg).
SubClass Equalizer1_fg := (Equalizer1 K_fg).
Definition K_hom := [f:a-->b]f.
Definition F_hom := (FPA K_hom).
SubClass Equalizer1_hom := (Equalizer1 K_hom).
End equaz_fg_hom.
```

9.1.6 Existence d'Objet Initial

Quelques lemmes utiles concernant les égalisateurs

1. Un morphisme $h : a \to b$ ayant un inverse droit est forcément un épi. Soient $f, g : b \to c$ tels que $h \circ f = h \circ g$. En composant à droite par h^{-1}, on obtient $h^{-1} \circ h \circ f = h^{-1} \circ h \circ g$, d'où finalement $f = g$.

```
Lemma RightInv_epic : (C:Category)(a,b:C)(h:a-->b)(h1:b-->a)
                      (RIso_law h h1) -> (Epic_law h).
Coercion RightInv_epic : RIso_law >-> Epic_law.
```

2. Nous allons démontrer que tout égalisateur est monic. Soit $e : c \to a$ l'égalisateur de la famille $\{k_i : a \to b \mid i : I\}$. Soient $f, g : d \to c$ tels que $f \circ e = g \circ e$, on doit montrer que $f = g$.

$$d \underset{g}{\overset{f}{\rightrightarrows}} c \xrightarrow{e} a \underset{k_j}{\overset{k_i}{\rightrightarrows}} b \tag{9}$$

   On a pour tous $i, j : I$, $f \circ e \circ k_i = f \circ e \circ k_j$. En appliquant la conditions d'unicité de e pour $f \circ e$, on détermine un unique morphisme $(f \circ e)^{\#} : d \to c$ avec $f \circ e = (f \circ e)^{\#} \circ e$. Or, on a par hypothèse $f \circ e = g \circ e$, donc $g = (f \circ e)^{\#}$. Aussi, $f \circ e = f \circ e$, d'où $f = (f \circ e)^{\#}$. Finalement $f = g$.

```
Section equaz_prop.
Variables C:Category; a,b:C; I:Type; k:I -> a-->b; f:(Equalizer k).
Lemma E_monic : (Monic_law (E_mor f)).
```

3. Tout égalisateur épi est iso. Soit $e : c \to a$ l'égalisateur de la famille $\{k_i : a \to b \mid i : I\}$. Supposons de plus que e est épi. Comme e est épi, et comme $\forall i, j : I, e \circ k_i = e \circ k_j$, on en déduit que $\forall i, j : I, k_i = k_j$. Donc Id_a est aussi un égalisateur de $\{k_i : a \to b \mid i : I\}$, et il existe un unique $Id_a^{\#} : a \to c$ tel que $Id_a = Id_a^{\#} \circ e$.
   On a de plus, $e \circ Id_a^{\#} \circ e = Id_a \circ e = e \circ Id_c$. Comme e est monic (voir ci-dessus), alors $e \circ Id_a^{\#} = Id_c$. Ce qui achève notre preuve: $Id_a^{\#}$ est l'inverse de e.

```
Lemma Epic_Equalizer_id : (Epic_law (E_mor f)) -> (Equalizer_eq (Id a)).
Lemma Epic_Equalizer_iso : (p:(Epic_law (E_mor f)))
      [f1=(E_diese f (Epic_Equalizer_id p))](AreIsos (E_mor f) f1).
```

Puisqu'on a montré qu'un morphisme admettant un inverse droit est épi, et comme tout égalisateur épi est iso, on en déduit qu'avoir un inverse droit est une condition suffisante pour qu'un égalisateur soit iso.

```
Lemma Equalizer_iso : (h1:a-->(E_ob f))(p:(RIso_law (E_mor f) h1))
                      [f1=(E_diese f (Epic_Equalizer_id (RightInv_epic p)))]
                      (AreIsos (E_mor f) f1).
End equaz_prop.
```

Énoncé

Soit D une catégorie complète, D a un objet initial si et seulement si elle vérifie la condition suivante.
Condition de l'Ensemble Solution (1). Il existe un ensemble I et une famille d'objets $\{k_i \mid i : I\}$ de D tel que pour tout $d : D$, il existe un indice $i_d : I$ et un morphisme $f_d : k_{i_d} \to d$.

Dans ce qui suit, nous appellerons cette condition $SSC1$ (de l'anglais *Solution Set Condition*). Cette condition et l'ensemble d'indices I sont représentés par des types, et non comme des Setoïdes.

```
Section ssc1.
Variable D:Category.
Structure Cond1 [I:Type;k:I->D;d:D] : Type :=
 {Cond1_i : I;
  Cond1_f : (k Cond1_i)-->d}.
Structure SSC1 :Type :=
 {SSC1_I : Type;
  SSC1_k : SSC1_I->D;
  SSC1_p : (d:D)(Cond1 SSC1_k d)}.
Definition SSC1_i := [s:SSC1][d:D](Cond1_i (SSC1_p s d)).
Definition SSC1_f := [s:SSC1][d:D](Cond1_f (SSC1_p s d)).
End ssc1.
```

Condition nécessaire

Posons les hypothèses: Soit D une catégorie complète possédant un objet initial d_1.

```
Section thi1.
Variables D:Category; D_initial:(Initial D).
Definition Thi1_d1 := (Initial_ob D_initial).
```

La $SSC1$ est facile à construire. L'ensemble I est le type singleton $\{*\}$ et la famille se réduit à $k_* = d_1$.

```
Inductive UnitType : Type := Elt : UnitType.
Definition Thi1_I := UnitType.
Definition Thi1_k := [i:Thi1_I]Cases i of Elt => Thi1_d1 end.
```

Pour tout $d : D$, il existe $f_d : d_1 \to d$ puisque d_1 est initial (f_d est même unique).

```
Definition Thi1_verif_Cond1 := [d:D](Build_Cond1 3!Thi1_k 5!Elt (MorI D_initial d)).
```

La construction de la $SSC1$ est alors achevée.

```
@Definition Thi1_SSC1 := (Build_SSC1 Thi1_verif_Cond1).
End thi1.
```

Condition suffisante

Soit D une catégorie complète vérifiant la $SSC1$.

```
Section thi2.
Variables D:Category; D_complete:(Complete D); SSC1_D:(SSC1 D).
```

Comme D est complète, elle admet en particulier tous les produits et tous les égalisateurs. Soit $w = \Pi_{i:I} k_i$.

```
Definition Thi2_D_prod : (Product1 (SSC1_k 2!SSC1_D)) :=
                         (D_complete (FunDiscr (SSC1_k 2!SSC1_D))).
Definition Thi2_w := (Pi1 Thi2_D_prod).
```

Soit aussi $e : v \to w$ l'égalisateur de tous les morphismes dans $Hom(w, w)$.

```
Definition Thi2_D_E_hom := (Build_Equalizer2 (D_complete (F_hom Thi2_w Thi2_w))
                                             (Id Thi2_w)).
Definition Thi2_v := (E1_ob Thi2_D_E_hom).
Definition Thi2_e := (E1_mor Thi2_D_E_hom).
```

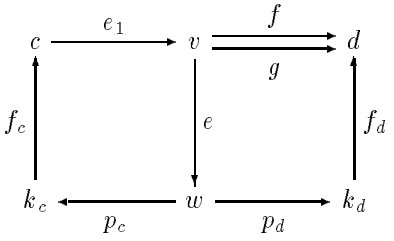
On va montrer que v est un objet initial dans D. Pour cela il nous faudra montrer qu'il existe un seul morphisme entre v et d pour tout $d : D$. Cet unique morphisme est $e \circ p_d \circ f_d : v \to d$.

```
Section thi2_mor_d_def.
Variable d:D.
Definition Thi2_p_i := (Proj1 Thi2_D_prod (SSC1_i SSC1_D d)).
Definition Thi2_f_d := (SSC1_f SSC1_D d).
Definition Thi2_mor_d := (Thi2_e o Thi2_p_i) o Thi2_f_d.
End thi2_mor_d_def.
```

Prenons maintenant deux morphismes $f, g : v \to d$ quelconques et montrons qu'ils sont égaux.

```
Section unique_mor_d.
Variables d:D; f,g:Thi2_v-->d.
```

Soit $e_1 : c \to v$ l'égalisateur de f et g.



(10)

```
Definition Thi2_D_E_fg := (Build_Equalizer2 (D_complete (F_fg f g)) Elt1).
Definition Thi2_c := (E1_ob Thi2_D_E_fg).
Definition Thi2_e1 := (E1_mor Thi2_D_E_fg).
```

Montrons que $e \circ s$, où $s = p_c \circ f_c$, est un inverse droit de e_1.

```
Definition Thi2_p_j := (Proj1 Thi2_D_prod (SSC1_i SSC1_D Thi2_c)).
Definition Thi2_f_c := (SSC1_f SSC1_D Thi2_c).
Definition Thi2_s := Thi2_p_j o Thi2_f_c.
```

Comme e est l'égalisateur de $s \circ e_1 \circ e$ et Id_w, on a $e \circ s \circ e_1 \circ e = e \circ Id_w = Id_v \circ e$. Or on a vu plus haut que e est monic, d'où $e \circ s \circ e_1 = Id_v$. $e \circ s$ est donc bien un inverse droit de e_1.

```
Lemma Thi2_e1_RightInv : (RIso_law Thi2_e1 (Thi2_e o Thi2_s)).
```

D'après le lemme démontré au début de la section, e_1 est iso. Soit e_1^{-1} son inverse. On a alors $f = e_1^{-1} \circ e_1 \circ f = e_1^{-1} \circ e_1 \circ g = g$.

```
Lemma Eq_fg : f =_S g.
End unique_mor_d.
```

La preuve s'achève ici puisqu'on a construit un objet initial v pour D.

```
Lemma Thi2_isInitial : (IsInitial Thi2_mor_d).
Definition Thi2_initial : (Initial D) := Thi2_isInitial.
End thi2.
```

9.2 Catégorie Comma

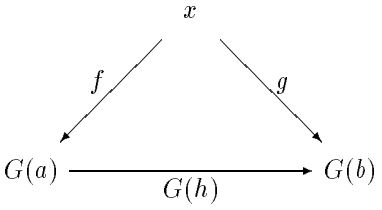
L'idée de base de la catégorie Comma est l'élévation des morphismes d'une catégorie en objets d'une autre. Notre définition, faisant intervenir un seul foncteur, est un cas particulier d'une définition plus générale faisant intervenir deux foncteurs (voir [137]).

9.2.1 Définition

Soient A, X deux catégories, $G : A \to X$ et x un objet de X. La catégorie Comma $(x \downarrow G)$ est celle dont les objets sont les morphismes de X de la forme $x \to G(a)$ où a est un objet de A. Plus exactement donc, ses objets sont des couples $\langle a, f : x \to G(a) \rangle$.

```
Section comma_def.
Variables A,X:Category; G:(Functor A X); x:X.
Structure Com_ob : Type :=
 {Ob_com_ob  : A;
  Mor_com_ob : x-->(G Ob_com_ob)}.
```

Un morphisme de $(x \downarrow G)$ entre $\langle a, f : x \to G(a) \rangle$ et $\langle b, g : x \to G(b) \rangle$ est un morphisme $h : a \to b$ (noté $\langle h \rangle$) tel que $g = f \circ G(h)$:



(11)

```
Section com_arrow_def.
Variables axf,bxg:Com_ob.
Definition Com_law := [h:(Ob_com_ob axf)-->(Ob_com_ob bxg)]
                      (Mor_com_ob bxg) =_S ((Mor_com_ob axf) o (FMor G h)).
Structure >Com_arrow : Type :=
 {Mor_com_arrow :  (Ob_com_ob axf)-->(Ob_com_ob bxg);
  Prf_com_law   :> (Com_law Mor_com_arrow)}.
End com_arrow_def.
```

On utilise ainsi l'égalité et la composition de morphismes de A pour la catégorie $(x \downarrow G)$. Pour tout objet $\langle a, f : x \to G(a)\rangle$, $\langle Id_a\rangle$ est aussi une identité pour cet objet.

```
@Definition Comma := (Build_Category Assoc_Comma Idl_Comma Idr_Comma).
End comma_def.
```

Nous associons à la catégorie Comma un foncteur de projection défini par $Q(\langle a, f\rangle) = a$ et $Q(\langle h\rangle) = h$.

```
Section comma_proj_def.
Variables A,X:Category; G:(Functor A X); x:X.
Definition Comma_proj_ob := [a:(Com_ob G x)](Ob_com_ob a).
Definition Comma_proj_mor := [a,b:(Comma G x)][f:a-->b](Mor_com_arrow f).
@Definition Comma_proj := (Build_Functor Comma_proj_comp_law Comma_proj_id_law).
End comma_proj_def.
```

9.2.2 Relation entre Catégorie Comma et Morphisme Universel

Nous pouvons utiliser la notion d'objet initial pour définir la notion de morphisme universel. En effet, $\langle a, u\rangle$ est un morphisme universel de x vers G si et seulement si il est un objet initial de $(x \downarrow G)$. C'est un exemple de l'interdéfinabilité des concepts d'universalité en théorie des catégories. Nous ne donnons ici que la construction transformant un objet initial de $(x \downarrow G)$ en un morphisme universel de x vers G.

```
Section ua_comma.
Variables A,X:Category; G:(Functor A X); x:X; axu:(Initial (Comma G x)).
Definition Com_diese := [a':A][f:x-->(G a')](Mor_com_arrow (MorI axu (Build_Com_ob f))).
Local axu_ob := (Initial_ob axu).
Lemma Com_isUA : (IsUA (Mor_com_ob axu_ob)).
@Definition Com_UA := (Build_UA Com_isUA).
End ua_comma.
```

9.2.3 Conditions Suffisantes à la Complétude d'une Catégorie Comma

Soient X, A et J des catégories et $G : A \to X$ un foncteur.

```
Section comma_complete.
Variables X,A:Category; G:(Functor A X); J:Category.
```

Supposons que A possède une limite pour tous les foncteurs $F : J \to A$ et que G préserve toutes ces limites.

```
Hypothesis A_comp_for_J : (F:(Functor J A))(Limit F).
Hypothesis G_pres_JA    : (F:(Functor J A))(Preserves_limits F G).
```

Nous allons montrer que $F : J \to (x \downarrow G)$ possède une limite.

```
Variables x:X; F:(Functor J (Comma G x)).
```

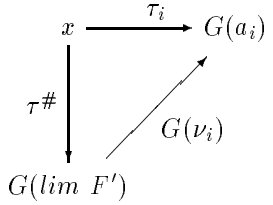
Soit $F' = F \circ Q : J \to A$ où $Q : (x \downarrow G) \to A$ est le foncteur de projection des catégories comma. Par hypothèse, F' possède une limite $\langle lim\ F', \nu \rangle$ et G la préserve, c'est-à-dire $\langle G(lim\ F'), G \circ \nu \rangle$ est une limite de $F' \circ G$.

```
Local F' := F o_F (Comma_proj G x).
Local l_F' := (A_comp_for_J F').
Local l_GF' := (G_pres_JA l_F').
```

Pour tout $i : J$, $F(i) : (x \downarrow G)$ est de la forme $\langle a_i, f_i \rangle$ où $a_i : A$ et $f_i : x \to G(a_i)$. Les morphismes f_i forment un cône $\tau : \Delta(x) \to F' \circ G$, en effet par définition $(F' \circ G)(i) = G(Q(F(i))) = G(a_i)$.

```
Definition Tc_tau := [i:J](Mor_com_ob (F i)).
Lemma Tc_tau_cone_law : (Cone_law 4!(F' o_F G) Tc_tau).
Definition Tc_cone := (Build_Cone Tc_tau_cone_law).
```

Il existe donc un morphisme unique $\tau^{\#}$ tel que pour tout $i : C$.



(12)

Définissons $lim\ F$ par $\langle lim\ F', \tau^{\#} \rangle$.

```
@Definition Tc_limF := (Build_Com_ob (l_GF' ? Tc_cone)).
```

Quant à son cône limite $\epsilon : \Delta(\langle lim\ F', \tau^{\#} \rangle) \to F$, il est défini par $\epsilon_i : \langle lim\ F', \tau^{\#} \rangle \to \langle a_i, f_i \rangle = \langle \nu_i \rangle$. La loi des comma étant vérifiée par ν_i d'après (12). Aussi, ϵ est un cône car ν est un cône.

```
Lemma Tc_limconeF_tau_com_law : (i:J)(Com_law 5!Tc_limF 6!(F i) ((Limiting_cone l_F') i)).
@Definition Tc_limconeF_tau := [i:J](Build_Com_arrow (Tc_limconeF_tau_com_law i)).
Lemma Tc_limconeF_cone_law : (Cone_law Tc_limconeF_tau).
Definition Tc_limconeF := (Build_Cone Tc_limconeF_cone_law).
```

Passons maintenant à la construction de $\_^{\#}$. Soit $t : \Delta(\langle a, f \rangle) \to F$ un cône, $t_i : \langle a, f \rangle \to \langle a_i, f_i \rangle$ est donc de la forme $\langle g_i \rangle$ avec $g_i : a \to a_i$ vérifiant:

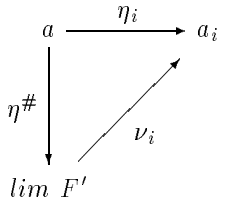
$$f_i = f \circ G(g_i) \tag{13}$$

```
Section ctdiese.
Variables axf:(Comma G x); t:(Cone axf F).
```

À partir de t, on construit un cône $\eta : \Delta(a) \to F'$ défini par $\eta_i = g_i$.

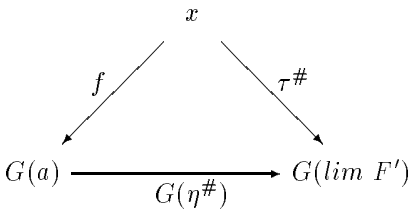
```
Definition Tc_t_tau := [i:J](Mor_com_arrow (t i)).
Lemma Tc_t_tau_cone_law : (Cone_law 3!(Ob_com_ob axf) 4!F' Tc_t_tau).
Definition Tc_t_cone := (Build_Cone Tc_t_tau_cone_law).
```

Il existe alors un unique $\eta^\#$ tel que pour tout $i : J$:



(14)

Remarquons que $\langle \eta^\# \rangle$ est un morphisme de $\langle a, f \rangle$ vers $\langle lim\ F', \tau^\# \rangle$. En effet $f \circ G(\eta^\#)$ vérifie (12), pour tout $i : J$, on a:



(15)

En effet $f \circ G(\eta^\#)$ vérifie (12) pour tout $i : J$.

τ_i	$=$	f_i	Définition de τ
	$=$	$f \circ G(g_i)$	(13)
	$=$	$f \circ G(\eta_i)$	Définition de η
	$=$	$f \circ G(\eta^\#) \circ G(\nu_i)$	(14)

Et par unicité de $\tau^\#$ (diagramme (12)), $\tau^\# = f \circ G(\eta^\#)$.

```
Lemma Tc_t_cone_com_law : (Com_law 5!axf 6!Tc_limF (Lim_diese l_F' Tc_t_cone)).
```

Nous définissons alors $t^\#$ par $t^\# = \langle \eta^\# \rangle$.

```
@Definition Tc_diese := (Build_Com_arrow Tc_t_cone_com_law).
End ctdiese.
```

Les propriétés d'universalité de $\langle lim\ F, \epsilon \rangle$ se déduisent facilement de celles de $\eta^\#$ (14).

```
Lemma Tc_UA_law1 : (Limit_law1 Tc_limconeF Tc_diese).
Lemma Tc_UA_law2 : (Limit_law2 Tc_limconeF Tc_diese).
Definition Comma_l_F := (Build_IsLimit Tc_UA_law1 Tc_UA_law2).
Definition Comma_l_F1 := (Build_Limit Comma_l_F).
End comma_complete.
```

Un cas particulier important est celui où A est complète et G est continu, on en déduit alors que $(x \downarrow G)$ est complet pour tout $x : X$.

```
Lemma Comma_complete : (X,A:Category)(G:(Functor A X))(Complete A) -> (Continuous G) ->
                       (x:X)(Complete (Comma G x)).
```

9.3 Adjonctions

Les adjonctions, introduites par Kan en 1958, constituent un outil de description très général, englobant différentes constructions mathématiques canoniques telles que les structures libres (algèbres libres), engendrées ou bien encore fermées (fermeture transitive d'un graphe). L'adjonction a dans ce sens, des liens très étroits avec les notions d'universalité vues dans les précédentes sections. Les adjonctions peuvent être définies de différentes façons équivalentes; nous en verrons deux dans cette section. Il y a des avantages à avoir plusieurs représentations d'une même notion. La principale est que certaines notions ou théorèmes sont plus faciles à établir avec une représentation qu'une autre. Le passage d'une représentation à une autre est réalisé de manière transparente par l'utilisation de coercions.

9.3.1 Définitions Préliminaires

Nous aurons besoin des définitions suivantes.

Catégorie Produit Étant données deux catégories quelconques A et B, leur catégorie Produit est notée $A \times B$. Ses objets sont tous les couples $\langle a, b\rangle$ où a est un objet de A et b un objet de B.

```
Section ProdCat.
Variables A,B:Category.
Structure POb : Type := {Ob_l : A; Ob_r : B}.
```

On dispose bien entendu de fonctions pour extraire les composantes d'un objet de $A \times B$. Ainsi, si `x` est un objet de $A \times B$, alors `(Ob_l x)` et `(Ob_r x)` accèdent respectivement aux objets gauche et droit de `x`.

Les morphismes de $A \times B$ sont aussi des couples. En effet, un morphisme entre $\langle a, b\rangle$ et $\langle c, d\rangle$ est un couple $\langle f, g\rangle$ où $f : a \to c$ dans A et $g : b \to d$ dans B.

```
Section pmor_setoid_def.
Variables u,t:POb.
Structure Pmor : Type :=
 {Hom_l : (Ob_l u)-->(Ob_l t);
  Hom_r : (Ob_r u)-->(Ob_r t)}.
@Definition Prod_Hom : Setoid := Equal_Pmor_equiv.
End pmor_setoid_def.
```

Comparer deux morphismes de $A \times B$ revient à les comparer composante par composante. La composition de deux morphismes de $A \times B$ lui aussi se fait composante par composante. Quant à l'identité, $Id_{\langle a,b\rangle} = \langle Id_a, Id_b\rangle$. Ces constructions vérifient les lois des catégories et nous permettent donc de construire la catégorie $A \times B$, notée `(PROD A B)`.

```
@Definition PROD := (Build_Category Assoc_PROD Idl_PROD Idr_PROD).
End ProdCat.
```

Nous définissons quelques abréviations pour manipuler des catégories de la forme $D^\circ \times C$.

```
Section abrev.
Variables C,D:Category.
Definition OB_l : (POb (Dual D) C) -> D := [dxc](Ob_l dxc).
Variables d1xc1,d2xc2:(POb (Dual D) C); foxg:(Pmor d1xc1 d2xc2).
```

```
Definition HOM_l : (OB_l d2xc2)-->(OB_l d1xc1) := (Hom_l foxg).
Definition Build_POb1 := [d:D][c:C](!Build_POb (Dual D) C d c).
Definition Build_Pmor1 := [c,c':C][d,d':D][f:d'-->d][g:c-->c']
                          (Build_Pmor 3!(Build_POb1 d c) 4!(Build_POb1 d' c') f g).
End abrev.
```

Hom-Foncteurs Contravariants Soient deux foncteurs $F : D \to C$ et $G : C \to D$. Le foncteur $Hom(F-,-) : D^\circ \times C \to SET$ est défini par $Hom(F-,-)(\langle d,c\rangle) = Hom(F(d),c)$ et $Hom(F-,-)(\langle f,g\rangle) = F(f) \circ h \circ g$ avec $f : d_2 \to d_1$, $g : c_1 \to c_2$ et $h : F(d_1) \to c_1$.

```
Section FunSet2_r.
Variables C,D:Category; F:(Functor D C).
Definition FunSET2_r_ob := [dxc:(POb (Dual D) C)](F (OB_l dxc))-->(Ob_r dxc).
 Section funset2_r_map_def.
 Variables d1xc1,d2xc2:(POb (Dual D) C); foxg:(Pmor d1xc1 d2xc2).
 Definition FunSET2_r_mor1 := [h:(FunSET2_r_ob d1xc1)]
                              ((FMor F (HOM_l foxg)) o h) o (Hom_r foxg).
 End funset2_r_map_def.
@Definition FunSET2_r := (Build_Functor Fun2_r_comp_law Fun2_r_id_law).
End FunSet2_r.
```

Quant au foncteur $Hom(-,G-) : D^\circ \times C \to SET$, il est défini par $Hom(-,G-)(\langle d,c\rangle) = Hom(d,G(c))$ et $Hom(-,G-)(\langle f,g\rangle) = f \circ h \circ G(g)$ avec $f : d_2 \to d_1$, $g : c_1 \to c_2$ et $h : d_1 \to G(c_1)$.

```
Section FunSet2_l.
Variables C,D:Category; G:(Functor C D).
Definition FunSET2_l_ob := [dxc:(POb (Dual D) C)](OB_l dxc)-->(G (Ob_r dxc)).
 Section funset2_l_map_def.
 Variables d1xc1,d2xc2:(POb (Dual D) C); foxg:(Pmor d1xc1 d2xc2).
 Definition FunSET2_l_mor1 := [h:(FunSET2_l_ob d1xc1)]
                              ((HOM_l foxg) o h) o (FMor G (Hom_r foxg)).
 End funset2_l_map_def.
@Definition FunSET2_l := (Build_Functor Fun2_l_comp_law Fun2_l_id_law).
End FunSet2_l.
```

Isomorphisme Naturel On appelle isomorphisme naturel un isomorphisme dans la catégorie des foncteurs. Il s'agit donc d'une transformation naturelle inversible.

```
Section natiso_def.
Variables A,B:Category.
Definition RNatIso_law := [F,G:(Functor A B)][T:(NT F G)][T1:(NT G F)]
                          (T1 o_NTv T) =_NT (Id_CatFunct G).
Variables F,G:(Functor A B).
Definition AreNatIsos := [T:(NT F G)][T1:(NT G F)]
                         (RNatIso_law T T1) /\ (RNatIso_law T1 T).
Structure >NatIso : Type :=
 {NatIso_mor :> (NT F G);
  NatIso_inv : (NT G F);
  Prf_NatIso :> (AreNatIsos NatIso_mor NatIso_inv)}.
End natiso_def.
```

Pour qu'une transformation naturelle T soit un isomorphisme naturel, il suffit que tous les morphismes T_a soient des isomorphismes. En effet, on définit $(T^{-1})_a$ par T_a^{-1}. On vérifie que ceci forme bien une transformation naturelle et que c'est bien l'inverse de T.

```
Section about_isIso.
Variables A,B:Category; F,G:(Functor A B); T:(NT F G); h:(a:A)(G a)-->(F a).
Hypothesis H:(a:A)(AreIsos (T a) (h a)).
Lemma NTinv_nt_law : (NT_law h).
@Definition NTinv := (Build_NT NTinv_nt_law).
Lemma NT_areIso : (AreNatIsos T NTinv).
@Definition NT_Iso : (NatIso F G) := NT_areIso.
End about_iso.
```

En fait, l'autre sens est aussi vrai. Si T est un isomorphisme naturel, on a forcément les T_a qui sont des isomorphismes. On définit leur inverse T_a^{-1} par $(T_{-1})_a$

9.3.2 Définition d'une Adjonction

Une adjonction met en jeu deux catégories C et D et deux foncteurs $F : D \to C$ et $G : C \to D$; elle est notée par $F \vdash G$. Elle définit une bijection entre les ensembles $Hom(F(d), c)$ et $Hom(d, G(c))$ «naturelle» en d et c. En d'autres termes, une adjonction entre ces deux foncteurs est un isomorphisme entre les foncteurs $Hom(F-,-)$ et $Hom(-,G-)$.

```
Section adj_def.
Variables C,D:Category; F:(Functor D C); G:(Functor C D).
SubClass Adj := (NatIso (FunSET2_r F) (FunSET2_l G)).
```

Une adjonction consiste donc en une transformation naturelle $\phi : Hom(F-,-) \to Hom(-,G-)$ inversible, c'est-à-dire qu'il existe $\phi^{-1} : Hom(-,G-) \to Hom(F-,-)$ tel que:

$$\phi \circ \phi^{-1} = Id_{C(F-,-)} \tag{16}$$

$$\phi^{-1} \circ \phi = Id_{D(-,G-)} \tag{17}$$

```
Variables phi:(NT (FunSET2_r F) (FunSET2_l G)); phi_1:(NT (FunSET2_l G) (FunSET2_r F)).
Variable phi_iso:(AreNatIsos phi phi_1).
Definition Build_Adj : Adj := (Build_NatIso phi_iso).
```

Il faut remarquer que si d est un objet de D et c un objet de C, alors $\phi_{\langle d,c\rangle} : Hom(F(d), c) \to Hom(d, G(c))$ est une application de SET. On peut donc l'appliquer à tout élément de l'ensemble $Hom(F(d), c)$. Si $f : F(d) \to c$, alors $\phi_{\langle d,c\rangle}(f) : d \to G(c)$. Comme d et c peuvent se déduire de f, on écrira $\phi(f)$.

Si `ad` est une adjonction entre F et G, alors `(ApAphi ad f)` et `(ApAphi_inv ad g)` représentent respectivement $\phi(f)$ et $\phi^{-1}(g)$.

```
Variables d:D; c:C; ad:Adj.
Definition ApAphi : (F d)-->c -> d-->(G c) := [f]((ad (Build_POb1 d c)) f).
Definition ApAphi_inv : d-->(G c) -> (F d)-->c := [g](((NatIso_inv ad) (Build_POb1 d c)) g).
End adj_def.
```

Un *adjoint gauche* de G est un foncteur F tel que $F \vdash G$ est une adjonction. Réciproquement, G est dit *adjoint droit* de F.

```
Structure LeftAdj [C,D:Category;G:(Functor C D)] : Type :=
 {Adjoint :  (Functor D C);
  Adj_l   :> (Adj Adjoint G)}.
Structure RightAdj [C,D:Category;F:(Functor D C)] : Type :=
 {CoAdjoint :  (Functor C D);
  Adj_r     :> (Adj F CoAdjoint)}.
```

Analysons de plus près cette définition de l'adjonction. Comme $Hom(F-,-)$ et $Hom(-,G-)$ sont des foncteurs de $D^\circ \times C$ vers SET, les deux équations (16) et (17) se simplifient en:

$$\forall f : F(d) \to c, \phi^{-1}(\phi(f)) = f \tag{18}$$

$$\forall g : d \to G(c), \phi(\phi^{-1}(g)) = g \tag{19}$$

La définition qu'on vient de donner est très dense, en fait elle cache plusieurs équations. Premièrement, ϕ est une transformation naturelle, donc vérifie la loi de naturalité, exprimée par l'équation ci-dessous obtenue après simplification. Pour tous $h : d' \to d$, $k : c \to c'$ et $f : F(d) \to c$,

$$\phi(F(h) \circ f \circ k) = h \circ \phi(f) \circ G(k) \tag{20}$$

De même, en utilisant la loi de naturalité pour ϕ^{-1}, on obtient pour tous $h : d' \to d$, $k : c \to c'$ et $g : d \to G(c)$,

$$\phi^{-1}(h \circ g \circ G(k)) = F(h) \circ \phi^{-1}(g) \circ k \tag{21}$$

```
Section adj_eqs.
Variables C,D:Category; F:(Functor D C); G:(Functor C D); ad:(Adj F G).
Variables d,d':D; c,c':C; h:d'-->d; k:c-->c'; f:(F d)-->c.
Lemma Adj_eq1 : (ApAphi ad (((FMor F h) o f) o k)) =_S
                ((h o (ApAphi ad f)) o (FMor G k)).
Variable g:d-->(G c).
Lemma Adj_eq2 : (ApAphi_inv ad ((h o g) o (FMor G k))) =_S
                (((FMor F h) o (ApAphi_inv ad g)) o k).
End adj_eqs.
```

En remplaçant dans la première équation h par Id_d, nous obtenons:

$$\phi(f \circ k) = \phi(f) \circ G(k) \tag{22}$$

Alors qu'en y remplaçant k par Id_c, nous prouvons:

$$\phi(F(h) \circ f) = h \circ \phi(f) \tag{23}$$

Quant à la deuxième équation, on en tire les deux cas particuliers suivants:

$$\phi^{-1}(g \circ G(k)) = \phi^{-1}(g) \circ k \tag{24}$$

$$\phi^{-1}(h \circ g) = F(h) \circ \phi^{-1}(g) \tag{25}$$

```
Section adj_eqs1.
Variables C,D:Category; F:(Functor D C); G:(Functor C D); ad:(Adj F G).
Variables d,d':D; c,c':C; h:d'-->d; k:c-->c'; f:(F d)-->c; g:d-->(G c).
Lemma Adj_eq3 : (ApAphi ad (f o k)) =_S ((ApAphi ad f) o (FMor G k)).
Lemma Adj_eq4 : (ApAphi ad ((FMor F h) o f)) =_S (h o (ApAphi ad f)).
Lemma Adj_eq5 : (ApAphi_inv ad (g o (FMor G k))) =_S ((ApAphi_inv ad g) o k).
Lemma Adj_eq6 : (ApAphi_inv ad (h o g)) =_S ((FMor F h) o (ApAphi_inv ad g)).
End adj_eqs1.
```

9.3.3 Une autre définition de l'adjonction

Nous donnons une autre définition possible de l'adjonction. Soient C et D deux catégories et $F : D \to C$ et $G : C \to D$ deux foncteurs.

```
Section adj1_def.
Variables C,D:Category; F:(Functor D C); G:(Functor C D).
```

Une adjonction entre F et G est définie par la donnée de deux transformations naturelles $\eta : Id_D \to F \circ G$ et $\epsilon : G \circ F \to Id_C$ vérifiant les deux équations ci-dessous où $\circ_h$ est la composition horizontale des transformations naturelles définie en 8.11.

$$(Id_G \circ_h \eta) \circ_v (\epsilon \circ_h Id_G) = Id_G \tag{26}$$

$$(\eta \circ_h Id_F) \circ_v (Id_F \circ_h \epsilon) = Id_F \tag{27}$$

Dans notre formalisation, ces équations ne sont pas «bien typées». Regardons de plus près la première équation. Nous distinguons deux problèmes différents:

1. Les deux sous-termes de son membre gauche ne peuvent pas être composés car $G \circ (F \circ G)$ et $(G \circ F) \circ G$ ne sont pas convertibles. En effet $Id_G \circ_h \eta : G \circ Id_D \to G \circ (F \circ G)$ et $\epsilon \circ_h Id_G : (G \circ F) \circ G \to Id_C \circ G$

2. De plus, se pose le même problème qu'en 8.11, celui d'écrire des égalités entre deux transformations naturelles de types différents (en supposant qu'on ait pu faire la composition ci-dessus). En effet ses deux membres ont des types différents (G et $Id_C \circ G$ ne sont pas définitionnellement égaux). En effet $(Id_G \circ_h \eta) \circ (\epsilon \circ_h Id_G) : G \circ Id_D \to Id_C \circ G$ et $Id_G : G \to G$

Ce problème se résout avec le même artifice qu'en 8.11, c'est-à-dire en utilisant une égalité moins contraignante que l'égalité habituelle des transformations naturelles. On verra toutefois qu'on n'y fera pas recours. En effet, revenons à notre premier problème, et essayons de simplifier ces équations. Après calculs, la première équation est équivalente à:

$$\forall c : C, \eta_{G(c)} \circ G(\epsilon_c) = Id_{G(c)} \tag{28}$$

Alors que l'équation (27) est équivalente (en faisant des calculs similaires) à:

$$\forall d : D, F(\eta_d) \circ \epsilon_{F(d)} = Id_{F(d)} \tag{29}$$

Ce sont ces équations qu'on va utiliser dans notre définition formelle, puisqu'elle sont équivalentes aux originelles et bien typées.

```
 Section adj1_laws.
 Variables eta:(NT (Id_CAT D) (F o_F G)); eps:(NT (G o_F F) (Id_CAT C)).
 Definition Adj1_law1 := (c:C)((eta (G c)) o (FMor G (eps c))) =_S (Id (G c)).
 Definition Adj1_law2 := (d:D) ((FMor F (eta d)) o (eps (F d))) =_S (Id (F d)).
 End adj1_laws.
Structure Adj1 : Type :=
 {A_eta         : (NT (Id_CAT D) (F o_F G));
  A_eps         : (NT (G o_F F) (Id_CAT C));
  Prf_Adj1_law1 : (Adj1_law1 A_eta A_eps);
  Prf_Adj1_law2 : (Adj1_law2 A_eta A_eps)}.
End adj1_def.
```

9.3.4 Adjonctions et réécriture

Quelle spécification équationnelle prendre pour les adjonctions? A priori, on en a deux correspondant aux deux définitions qu'on en a donné. La première est $E_1 = \{(18), (19), (20), (21)\}$, comportant aussi les équations relatives aux catégories C et D et aux foncteurs F et G. La deuxième, correspondant à la deuxième définition, est $E_2 = \{(28), (29)\}$ en plus des équations relatives aux catégories C et D, aux foncteurs F et G et aux transformations naturelles η et ϵ. Il est possible de simplifier E_1:

- $E_3 = \{(18), (19), (20)\}$ car (21) en est une conséquence.
- $E_4 = \{(18), (19), (21)\}$ car (20) en est une conséquence.
- $E_5 = \{(18), (19), (22), (23)\}$ car (20) est une conséquence de (22) et (23).
- $E_6 = \{(18), (19), (24), (25)\}$ car (21) est une conséquence de (24) et (25).

Supposons que des conversions existent pour ces spécifications. Le choix de la conversion à utiliser dépend de l'équation à démontrer, suivant que l'adjonction mise en jeu est de type `Adj` ou `Adj1`, c'est-à-dire dont les termes sont respectivement construits sur les signatures $\{\circ, F, G, \phi, \phi^{-1}\}$ ou $\{\circ, F, G, \eta, \epsilon\}$. Dans le cas où l'on a affaire à plusieurs adjonctions, il est possible que leur signature ne soit pas pour toutes la même. Les conversions de réécriture ne sont pas alors toujours utilisables. Pour s'abstraire complètement de la spécification utilisée, nous établissons une conversion permettant de passer de l'une vers l'autre.

Il faut d'abord prouver que les deux définitions qu'on a donné pour l'adjonction sont équivalentes. Il s'agit en fait d'établir une correspondance entre nos deux définitions de l'adjonction.

De `Adj` vers `Adj1`. Soient C et D deux catégories, et ϕ une adjonction de $F : D \to C$ et $G : C \to D$. Nous définissons η par $\eta_d = \phi(Id_{F(d)})$ et ϵ par $\epsilon_c = \phi^{-1}(Id_G(c))$. Nous vous épargnons les moult vérifications.

```
Section transf_adj.
Variables C,D:Category; F:(Functor D C); G:(Functor C D); ad:(Adj F G).
Definition Unit_tau := [d:D](ApAphi ad (Id (F d))).
@Definition Unit_NT := (Build_NT Unit_nt_law).
Definition CoUnit_tau := [c:C](ApAphi_inv ad (Id (G c))).
@Definition CoUnit_NT := (Build_NT CoUnit_nt_law).
@Definition Adj_to_Adj1 := (Build_Adj1 Unit_and_CoUnit_law1 Unit_and_CoUnit_law2).
```

De `Adj1` vers `Adj`. Partant d'une adjonction entre F et G définie par l'unité η et la co-unité ϵ, il nous faut construire un isomorphisme ϕ entre $Hom(F-,-)$ et $Hom(-,G-)$. Nous posons $\phi(f) = \eta_d \circ G(f)$ pour tout $f : F(d) \to c$ et $\phi^{-1}(g) = F(g) \circ \epsilon_c$ pour tout $g : d \to G(c)$.

```
Variable ad1:(Adj1 F G).
Definition Teta_arrow := [dxc:(POb (Dual D) C)][f:(F (OB_l dxc))-->(Ob_r dxc)]
                         ((A_eta ad1) (OB_l dxc)) o (FMor G f).
@Definition Teta := (Build_NT Teta_tau_NT_law).
Definition Teta_1_arrow := [dxc:(POb (Dual D) C)][g:(OB_l dxc)-->(G (Ob_r dxc))]
                           (FMor F g) o ((A_eps ad1) (Ob_r dxc)).
@Definition Teta_1 := (Build_NT Teta_1_tau_NT_law).
Lemma Teta_1_o_Teta : (dxc:(PROD (Dual D) C))(AreIsos (Teta dxc) (Teta_1 dxc)).
Definition Adj1_to_Adj := (Build_Adj (NT_Iso Teta_1_o_Teta)).
End transf_adj.
Coercion Adj_to_Adj1 : Adj >-> Adj1.
Coercion Adj1_to_Adj : Adj1 >-> Adj.
```

Le système de transformation `Adj` vers `Adj1` est:

$$\begin{array}{lrcl} (E\phi) & \phi(f) & \to & \eta_d \circ G(f) \\ (E\phi^{-1}) & \phi^{-1}(g) & \to & F(g) \circ \epsilon_c \end{array}$$

Il faut remarquer que les adjonctions de part et d'autre des règles ne sont pas les mêmes, mais ceci est aussi transparent pour l'utilisateur grâce aux coercions `Adj_to_Adj1` et `Adj1_to_Adj`.

La conversion correspondante est:

$$let\ Adj\_Adj1\ =\ SweepUp\ (E\phi C\ OrelseC\ E\phi^{-1}C)$$

Le système de transformation et la conversion correspondante de `Adj1` vers `Adj` sont:

$$\begin{array}{lrcl} (E\eta) & \eta_d & \to & \phi(Id_{F(d)}) \\ (E\epsilon) & \epsilon_c & \to & \phi^{-1}(Id_{G(c)}) \end{array}$$

$$let\ Adj1\_Adj =\ SweepUp\ (E\eta C\ OrelseC\ E\epsilon C)$$

Une autre solution consiste a définir directement un système de réécriture pour la spécification algébrique comportant toutes les équations sur la signature $\{\circ, F, G, \phi, \phi^{-1}, \eta, \epsilon\}$.

9.3.5 Construction d'un Adjoint Gauche

Un foncteur $G : C \to D$ admet un adjoint gauche si pour tout $d : D$, il existe un morphisme universel $\langle c_d, u_d \rangle$ de d vers G.

```
Section ua_to_ladj.
Variables C,D:Category; G:(Functor C D).
Hypothesis UA_of:(d:D)(UA d G).
```

Cet adjoint gauche F est défini par $F(d) = c_d$ et $F(f) = (f \circ u_{d'})^{\#}$ pour $f : d \to d'$.

```
Definition AdjointUA_ob := [d:D](UA_ob (UA_of d)).
Definition AdjointUA_mor := [d,d':D][f:d-->d']
                            (UA_diese (UA_of d) (f o (UA_mor (UA_of d')))).
```

On vérifie aussi les lois de fonctorialité.

```
@Definition AdjointUA := (Build_Functor AdjointUA_comp_law AdjointUA_id_law).
```

Il nous reste à construire une adjonction entre F et G. Nous définissons $\phi : Hom(F-,-) \to Hom(-,G-)$ en posant pour tout $f : F(d) \to c$, $\phi(f) = u_d \circ G(f)$.

```
Definition PhiUA_arrow := [dxc:(POb (Dual D) C)][f:(UA_ob (UA_of (OB_l dxc)))-->(Ob_r dxc)]
                          (UA_mor (UA_of (OB_l dxc))) o (FMor G f).
@Definition PhiUA := (Build_NT PhiUA_tau_nt_law).
```

Quant à ϕ^{-1}, on la définit par $\phi^{-1}(f) = f^{\#}$ avec $f : d \to G(c)$.

```
Definition PhiUA_1_arrow := [dxc:(POb (Dual D) C)][f:(OB_l dxc)-->(G (Ob_r dxc))]
                            (UA_diese (UA_of (OB_l dxc)) f).
@Definition PhiUA_1 := (Build_NT PhiUA_1_tau_nt_law).
```

La dernière étape consiste à prouver que ϕ et ϕ^{-1} sont mutuellement inverses.

```
Lemma PhiUA_1_o_PhiUA : (dxc:(POb (Dual D) C))(AreIsos (PhiUA dxc) (PhiUA_1 dxc)).
Definition AdjUA := (Build_Adj (NT_Iso PhiUA_1_o_PhiUA)).
@Definition LeftAdjUA := (Build_LeftAdj AdjUA).
End ua_to_ladj.
```

9.4 Théorème du Foncteur Adjoint de Freyd

9.4.1 Énoncé

Le théorème que nous allons prouver est le suivant:

Énoncé: Soit A une catégorie complète, un foncteur $G : A \to X$ a un foncteur adjoint si et seulement si il préserve toutes les limites et vérifie ce qui suit:
Condition de l'Ensemble Solution (2). Pour tout objet x de X, il existe un ensemble I et une famille indexée par I de morphismes $f_i : x \to G(a_i)$ telle que tout morphisme $h : x \to G(a)$ peut s'écrire $h = f_i \circ G(t)$ pour un certain indice i et un certain $t : a_i \to a$.

On va commencer par formaliser la condition ci-dessus. On lui donnera le nom $SSC2$.

```
Section ssc2_def.
Variables A,X:Category; G:(Functor A X); x:X.
Structure Cond2 [I:Type;l:I->A;f:(i:I)x-->(G (l i));a:A;h:x-->(G a)] : Type :=
 {Cond2_i       : I;
  Cond2_t       : (l Cond2_i)-->a;
  Prf_Cond2_law : h =_S ((f Cond2_i) o (FMor G Cond2_t))}.
Structure SSC2 : Type :=
 {SSC2_I : Type;
  SSC2_a : SSC2_I->A;
  SSC2_f : (i:SSC2_I)x-->(G (SSC2_a i));
  SSC2_p : (a:A)(h:x-->(G a))(Cond2 SSC2_f h)}.
```

```
Variables s:SSC2; a:A; h:x-->(G a).
Definition SSC2_i := (Cond2_i (SSC2_p s h)).
Definition SSC2_t := (Cond2_t (SSC2_p s h)).
Lemma Prf_SSC2_law : h =_S ((SSC2_f SSC2_i) o (FMor G SSC2_t)).
End ssc2_def.
```

9.4.2 Condition Nécessaire

Première Partie Soient C et D deux catégories. Si un foncteur $G : C \to D$ possède un adjoint gauche, alors il est continu.
Soit l une limite d'un foncteur $H : J \to C$. Soit $\nu : \Delta(lim\ H) \to H$ son cône limite.

```
Section ladj_pres.
Variables C,D:Category; G:(Functor C D); la:(LeftAdj G);
          J:Category; H:(Functor J C); l:(Limit H).
```

On doit montrer que G préserve l, en d'autres termes que $\langle G(lim\ H), \nu \circ G\rangle$ est une limite de $H \circ G : J \to D$. Il nous faut construire pour tout cône $\tau : \Delta(d) \to H \circ G$, un unique $\tau^{\#}$ tel que pour tout $i : J$,

$$\begin{array}{ccc} d & \xrightarrow{\tau_i} & G(H(i)) \\ {\scriptstyle \tau^{\#}}\downarrow & \nearrow_{G(\nu_i)} & \\ G(lim\ H) & & \end{array} \tag{30}$$

Soient $F : D \to C$ l'adjoint droit de G et $\phi : D(F-,-) \to C(-,G-)$ l'isomorphisme naturel de l'adjonction $F \vdash G$. Définissons le cône $\sigma : \Delta(d) \to H$ par $\sigma_i = \phi^{-1}(\tau_i) : F(d) \to H(i)$ pour tout $i : J$.

```
Section lp_diese.
Variables d:D; tau:(Cone d (H o_F G)).
Definition Lp_sigma_tau := [i:J] (ApAphi_inv la ((tau i)::(d-->(G (H i))))).
Lemma Lp_sigma_tau_cone_law : (Cone_law Lp_sigma_tau).
Definition Lp_sigma := (Build_Cone Lp_sigma_tau_cone_law).
```

Par universalité de l, il existe un unique $\sigma^{\#}$ tel que pour tout $i : J$.

$$\begin{array}{ccc} F(d) & \xrightarrow{\sigma_i} & H(i) \\ {\scriptstyle \sigma^{\#}}\downarrow & \nearrow_{\nu_i} & \\ lim\ H & & \end{array} \tag{31}$$

Prenons alors $\tau^{\#} = \phi(\sigma^{\#})$.

```
Definition Lp_diese : d-->(G (Lim l)) := (ApAphi la (Lim_diese l Lp_sigma)).
End lp_diese.
```

$\tau^\#$ vérifie (30) puisque:

$$\begin{array}{rcll} \tau^\# \circ G(\nu_i) & = & \phi(\sigma^\#) \circ G(\nu_i) & \text{définition de } \tau^\# \\ & = & \phi(\sigma^\# \circ \nu_i) & (22) \\ & = & \phi(\sigma_i) & (31) \\ & = & \phi(\phi^{-1}(\tau_i)) & \text{définition de } \sigma \\ & = & \tau_i & \phi \text{ est iso} \end{array}$$

```
Lemma Lp_coUAlaw1 : (Limit_law1 ((Limiting_cone l) o_C G) Lp_diese).
```

Pour montrer l'unicité de $\tau^\#$, supposons qu'il existe un autre morphisme $f : d \to G(lim\ H)$ vérifiant (30) et montrons que $\phi^{-1}(f)$ vérifie (31). On a pour tout $i : J$:

$$\begin{array}{rcll} \phi^{-1}(f) \circ \nu_i & = & \phi^{-1}(f \circ G(\nu_i)) & (24) \\ & = & \phi^{-1}(\tau_i) & \text{hypothèse} \\ & = & \sigma_i & \text{définition de } \sigma \end{array}$$

Par universalité de l, $\sigma^\# = \phi^{-1}(f)$. Et enfin pour tout $i : J$,

$$\begin{array}{rcll} \tau^\# & = & \phi(\sigma^\#) & \text{définition de } \tau^\# \\ & = & \phi(\phi^{-1}(f)) & \text{résultat ci-dessus} \\ & = & f & \phi \text{ est iso} \end{array}$$

Ce qui prouve que $\tau^\#$ est l'unique morphisme vérifiant (30).

```
Lemma Lp_coUAlaw2 : (Limit_law2 ((Limiting_cone l) o_C G) Lp_diese).
```

En résumé, G préserve l. Comme J, H et l sont quelconques, nous venons de prouver que G est continu. Si G a un foncteur adjoint, il preserve forcément toutes les limites.

```
Lemma Ladj_Pres1 : (Preserves_1limit G l).
End ladj_pres.
Lemma Ladj_continuous : (C,D:Category)(G:(Functor C D))(LeftAdj G) -> (Continuous G).
```

Deuxième Partie Il nous reste à vérifier la condition $SSC2$.

```
Section freyd_th_1.
Variables A,X:Category; G:(Functor A X); A_c:(Complete A); la:(LeftAdj G).
```

Pour tout x de X, nous prendrons comme ensemble index l'ensemble singleton $\{*\}$ (voir sa définition en page 171). La famille d'objets se réduit à un seul objet, nous le définissons par $a_* = F(x)$ où F est le foncteur adjoint de G.

```
Variable x:X.
Local I := UnitType.
Local a := [i:I]Cases i of Elt => ((Adjoint la) x) end.
```

Quant à la famille de morphismes f_i, on la définit par $f_* = \eta_x$ où η est l'unité définie par l'adjonction de $F \vdash G$.

```
Local f := [i:I]<[i':I](Carrier x-->(G (a i')))>
                Cases i of Elt => ((Unit_NT la) x) end.
```

Pour tout morphisme $h : x \to G(r)$, nous prenons $i = *$ (nous n'avons d'ailleurs pas d'autres choix), et $t = \phi^{-1}(h)$.

```
 Section cd2.
 Variables r:A; h:x-->(G r).
 Local i := Elt.
 Local t : (a i)-->r := (ApAphi_inv la h).
 Lemma FT_cd2_law : h =_S ((f i) o (FMor G t)).
 @Definition FT_cd2 := (Build_Cond2 5!I FT_cd2_law).
 End cd2.
Lemma AFT2 : (SSC2 G x).
End freyd_th_1.
```

9.4.3 Condition Suffisante

Il s'agit de construire un morphisme universel de x vers G pour tout $x : X$. Pour cela, nous allons exploiter le fait que construire un tel morphisme universel correspond exactement à construire un objet initial dans la catégorie comma $(x \downarrow G)$. Ce résultat est prouvé en (9.2.2). Or dans (9.1.6), nous avons prouvé un théorème donnant les conditions nécessaires à l'existence d'un objet initial dans une catégorie quelconque. Nous allons appliquer ce lemme à la catégorie comma $(x \downarrow G)$. Nous devons donc vérifier les prémisses de ce théorème. La première condition consiste à vérifier la condition $SSC1$ pour $(x \downarrow G)$.

```
Section freyd_th_2'.
Variables A,X:Category; G:(Functor A X); A_c:(Complete A); G_c:(Continuous G);
          s:(x:X)(SSC2 G x).
 Section uaxG'.
 Variable x:X.
```

Construisons donc les différents ingrédients de la condition $SSC1$. Ceux-ci ont obtenus à partir de la condition $SSC2$ que nous avons en hypothèse. Ainsi, l'ensemble I de $SSC1$ est I_x de $SSC2$.

```
  Section ssc2_to_ssc1.
  Local I := (SSC2_I (s x)).
  Local f := [i:I](SSC2_f i).
  Local a := [i:I](SSC2_a i).
  Local k := [i:I](Build_Com_ob (f i)).
   Section ssc2_to_cond1.
   Variable axh:(Comma G x).
   Definition SSC2_1_i := (SSC2_i (s x) (Mor_com_ob axh)).
   Definition SSC2_1_t := (SSC2_t (s x) (Mor_com_ob axh)).
   Lemma SSC2_1_f_com_law : (Com_law 5!(k SSC2_1_i) 6!axh SSC2_1_t).
   @Definition SSC2_1_f := (Build_Com_arrow SSC2_1_f_com_law).
   @Definition SSC2_1_cond := (Build_Cond1 2!I SSC2_1_f).
   End ssc2_to_cond1.
  @Definition SSC2_1 := (Build_SSC1 SSC2_1_cond).
  End ssc2_to_ssc1.
```

La deuxième condition à vérifier est la complétude de $(x \downarrow G)$. D'après le lemme (9.2.3), pour que $(x \downarrow G)$ soit complète, il suffit que A soit complète et G continu. Ces deux conditions sont vraies par hypothèse.

```
 Definition FT_UA' := (Com_UA (Thi2_initial (Comma_complete A_c G_c 6!x) SSC2_1)).
 End uaxG'.
Definition AFT1' : (LeftAdj G) := (LeftAdjUA FT_UA').
End freyd_th_2'.
```

9.5 Aperçu sur le reste du développement

Il nous est impossible, pour manque de place, de présenter la totalité des notions que nous avons formalisés et les résultats que nous avons prouvés dans Coq. Nous invitons le lecteur intéressé à consulter notre rapport [141] (disponible à l'adresse `http://pauillac.inira.fr/~saibi`) contenant la totalité de notre développement. Dans cette section, nous nous contentons d'énumérer la liste des notions formalisées et lemme prouvés constituant le reste du développement.

Catégories et constructions catégoriques

- Définition de la catégorie MON des monoïdes.
- Définition de la notion de sous-catégorie pleine: soit une catégorie C, la sous-catégorie pleine C_I contient les objets de la famille $\{a_i : (C \mid i : I\}$ et tous les morphismes de C entre ces objets.
- Définitions des notions de produits fibrés (pullbacks) et de CCC (catégories cartésiennes fermées).
- Exemples des diverses constructions catégoriques (égalisateurs, produits, produits fibrés, CCC) dans SET.

Foncteurs

- Définition des propriétés de foncteurs fidèle et plein.
- Définition des foncteurs de projection $Fst : C \times D \to C$ et $Snd : C \times D \to D$, et du foncteur d'inclusion $Inc : C_I \to C$.
- Définition du Hom-Foncteur $Hom(a, -) : C \longrightarrow SET$.
- Définition du foncteur d'oubli $U : MON \to SET$ et du foncteur $U' : SET \to MON$ qui pour tout Setoïde A associe le monoïde libre A^* engendré à partir de ce Setoïde. Les éléments de A^* sont des mots formés par l'alphabet A, et sont implémentés comme des listes.

Transformations naturelles

- Définition de la transformation naturelle $H^f : Hom(b, -) \longrightarrow Hom(a, -)$ avec $f : a \longrightarrow b$.
- Construction du plongement de Yoneda $Y : C^\circ \longrightarrow FUNCT(C, SET)$ qu'on a prouvé fidèle et plein.
- Preuve du lemme de Yoneda qui pour tout foncteur $F : C \to SET$ énonce l'existence d'une bijection entre l'ensemble $F(a)$ et l'ensemble des transformations naturelles entre $Hom(a, -)$ et F. Cette bijection est de plus naturelle en a et F.

Constructions universelles et limites

- Construction d'un exemple de morphisme universel: $\langle A^*, \eta \rangle$ est un morphisme universel de A vers $U : MON \to SET$ avec $\eta : A \to U(A^*)$ en considérant les éléments de A comme des lettres de A^* (listes avec un seul élément).
- Preuve d'isomorphisme dans SET entre $Cones(c, F)$ et $Hom(c, lim\ l)$ pour toute limite l du foncteur $F : J \to C$, et tout objet $c : C$.
- Définition des produits fibrés en utilisant des limites.
- Preuve du résultat: SET est complet.
- Construction de limites à partir de produits et égalisateurs:
  Soient C et J deux catégories. Si C possède un égalisateur pour chaque paire de morphismes, et tous les produits de familles d'objets de C indicées par J et $Mor(J)$ (l'ensemble des morphismes de J), alors C possède une limite pour tout foncteur $F : J \to C$.
- Les Hom-foncteurs $Hom(a, -)$ sont continus.

Adjonctions

- Preuve d'unicité de l'adjoint gauche.
- Construction à partir d'une adjonction entre $F : D \to C$ et $G : C \to D$, d'un morphisme universel de d vers vers G pour tout objet d de D.
- Exemple: U^* est l'adjoint gauche de U.
- Définition de CCC par les Adjonctions.
- Nous avons donné une deuxième preuve de la condition suffisante du théorème de Freyd, suivant la présentation d'Alex Poigné[129]. Elle est totalement différente de celle donnée en 9.4 car elle est plus directe et utilise les produits fibrés.

Chapitre 10

Analyse et Critiques

10.1 Quelques chiffres

Le développement complet est organisé en 98 modules (dont 6 modules consistant en des duplications de définitions afin de contourner les problèmes d'incohérence d'univers). Il se compile en entier en moins de 17 minutes sur une machine moderne (une alpha cadencée à 450 Mhz). Un gain considérable de temps d'exécution est dû à l'utilisation des mécanismes d'abus de notations. Il est en effet beaucoup plus rapide de typer un terme avec beaucoup d'informations implicites qu'un terme explicite, même si les algorithmes de typage sont plus compliqués dans le premier cas.

Le développement compte plus de 100 coercions déclarées pour plus de 350 chemins de coercions valides engendrés. Les mécanismes d'héritage et de sous-termes implicites sont intensivement utilisés. Ainsi le développement complet comporte 13392 sous-termes implicites (inconnues « ? »), et l'heuristique 4.5 (page 78) a été utilisée 407 fois, engendrant 926 nouvelles inconnues. Quant au mécanisme d'héritage, il a été utilisé 2167 fois.

10.2 Sur la forme

Il est difficile de juger de la fidélité de notre formalisation aux textes mathématiques standards traitant des notions basiques de la théorie des catégories. Il est certain que notre formalisation s'en éloigne quelquefois, notamment lors de l'utilisation d'artifices pour contourner des difficultés liées à la nature intensionnelle de l'égalité définitionnelle. À ce sujet, il semble possible de traiter ces problèmes de manière plus satisfaisante en utilisant la technique de M. Hofmann[75] consistant à représenter des traits extensionnels dans des théories des types intensionnelles.

Au niveau des énoncés

La première différence «visuelle» entre notre formalisation et un développement classique dans un ouvrage mathématique est à rechercher du coté des conventions de «nommage». Les textes mathématiques disposent d'un arsenal illimité de symboles et d'alphabets pour nommer les objets et les notions utilisés. Dans une implémentation informatique (telle que Coq), nous devons nous contenter des caractères ASCII. À cette limitation, s'ajoutent les contraintes techniques d'analyse lexicale et syntaxique, indispensables pour garantir la non-ambiguïté des noms. Pour y remédier, nous utilisons des identificateurs (noms) généralement plus longs, censés être plus parlants. Avec

l'inconvénient que le résultat obtenu est plus verbeux qu'un texte informel. De plus, le système Coq (et par conséquent notre développement en pâtit) souffre d'une limitation importante: il n'existe qu'un seul espace de noms. Ainsi tous les noms intervenant dans un développement doivent être distincts (l'intégration prochaine d'un mécanisme de modules devrait remédier à cet inconvénient).

La surcharge de noms est largement utilisée dans les textes informels. Elle évite la multiplication des identificateurs, et rend le texte plus lisible (à condition qu'elle ne soit pas utilisée à outrance). Certaines facettes de ce mécanisme sont capturées par notre formalisme, de trois manières différentes:

- Le polymorphisme constitue une forme de surcharge, puisque un même nom de fonction peut-être appliqué à des objets de différents types. Ainsi, $(Id\ 3)$ et $(Id\ true)$ sont tous deux valides. Notre mécanisme de sous-termes implicites permet de plus d'expliquer le polymorphisme en considérant les deux occurrences de Id comme respectivement $(Id\ nat)$ et $(Id\ bool)$.

- Les coercions implicites implémentent aussi une forme intéressante de surcharge (voir l'introduction du chapitre 5). Le cas typique consiste à autoriser l'application d'un nom de fonction $f : A \to B$ à un argument $a' : A'$ s'il existe une coercion $c' : A' \to A$. Il y'a deux façons de voir $(f\ a')$. La première, la plus répandue, est de considérer que a' est surchargé, et représente $(c'\ a')$. La seconde, plus proche de la notion usuelle de surcharge, est de considérer f comme un nom surchargé, représentant les différentes fonctions $f \circ c$ où $c : C \to A$ est une coercion.

- La dernière facette est la technique décrite dans la section 4.7. C'est la technique utilisée, par exemple, pour pouvoir utiliser le même nom `Equal` pour désigner diverses égalités: entre applications, foncteurs, transformations naturelles etc.

Au niveau des preuves

Les preuves de notre développement sont essentiellement constructives; il s'agit généralement de construire des structures vérifiant certaines propriétés (catégories, foncteurs, limites, adjonctions etc.). Ces preuves sont présentées de manière détaillée en utilisant des lemmes et des définitions intermédiaires afin d'expliciter chacune des composantes de la structure à construire. Nos preuves sont dans ce sens peu automatiques, et peuvent l'être davantage. Automatiser toute la recherche de preuve n'est toutefois pas souhaitable, car les preuves sont importantes en mathématiques (autant que l'énoncé démontré). On ne peut donc pas se contenter de l'existence d'une preuve trouvée par un système de preuve.
Certaines parties correspondant à des preuves «faciles» doivent être par contre automatisées. La notion de preuve facile est impossible à définir en général, car dépend du domaine étudié et de l'expérience du mathématicien. Nos procédures de preuves (tactiques) pour le raisonnement équationnel automatisent des étapes de preuves qui sont toujours considérées comme faciles. Il s'agit des preuves par simplification ou calcul. Réaliser ces preuves à la main est cependant très fastidieux. Notre approche basée sur la réécriture est l'équivalent équationnel de la technique de commutation des diagrammes. Cette dernière reste difficile à mettre en œuvre dans un cadre informatique. Notre approche constitue néanmoins une avancée par rapport à l'approche des tactiques car elle permet de décrire et de combiner des stratégies de réécriture; de plus elle décharge l'utilisateur de la mention des étapes de transitivité et de congruence (comme c'est le cas aussi en mathématiques standards).

10.3 Discussion - Égalité d'objets

La correction des preuves des théorèmes que nous venons de vérifier dans Coq dépend essentiellement de la correction de l'implémentation du système Coq par rapport à la théorie des types qu'il est sensé implémenter. Heureusement, on a vu en 6.4.2 que cette correction ne dépendait que du noyau, une partie infime du système. Une autre question plus difficile à trancher est celle de la correction de l'énoncé de nos théorème par rapport à leur énoncé intuitif en mathématiques informelles. En d'autres termes, est-ce que nous avons réellement prouvé le théorème que nous avions en tête? Cette fois, la correction dépend du «bien-fondé» de la représentation de chacune des notions que nous avons formalisée. Nous en profitons pour revenir sur notre choix de ne point utiliser d'égalité entre objets.

Il n'est pas courant en théorie des catégories d'utiliser des jugements d'égalité entre objets; on utilise plutôt des isomorphismes. Notre définition est conforme à cet usage en n'exigeant pas une structure de Setoïde pour la collection d'objets d'une catégorie. La seule égalité utilisée entre objets est l'égalité définitionnelle, qui est intrinsèque à notre formalisme. Cependant, la collection d'objets de certaines catégories est naturellement munie d'une égalité. C'est notamment vrai lorsque les objets de cette catégorie sont des morphismes d'une autre catégorie, comme pour la catégorie des foncteurs et la catégorie Comma. Dans la catégorie de foncteurs, deux objets (foncteurs) G et H peuvent être égaux par l'égalité des foncteurs sans être convertibles (comme vu en 8.11 pour $(F \circ F') \circ F''$ et $F \circ (F' \circ F'')$). Les hom-Setoïdes $Hom(F,G)$ et $Hom(F,H)$ où F est un objet quelconque, sont alors considérés comme distincts, et des morphismes (transformations naturelles) comme $T : F \to G$ et $U : H \to F'$ ne sont pas composables.

Pour prendre en compte cette égalité, nous allons présenter une nouvelle définition de la notion de catégorie, qui n'est pas basée sur la notion de hom-Setoïde. Une catégorie est formée d'une collection d'objets et d'une collection de morphismes. À chaque morphisme est associé un objet domaine et un objet codomaine, à travers respectivement les applications $Dom1$ et $Cod1$.

```
Section cat1.
Variables Ob1,Arr1:Setoid; Dom1,Cod1:(Map Arr1 Ob1).
```

Quant à la composition de morphismes, c'est une fonction partielle définie pour les morphismes f et g composables, c'est-à-dire vérifiant $(Cod1\ f) = (Dom1\ g)$.

```
Variable Comp1:(f,g:Arr1) (Cod1 f) =%S (Dom1 g) -> Arr1.
```

La composition est ainsi une fonction à trois arguments, où en plus des morphismes à composer, il faut fournir une preuve qu'ils sont composables. Dans des systèmes de type moins expressifs tels que ML, on a recours aux exceptions. Ainsi dans la définition ML de la notion de catégorie dans [137], la composition est simplement de type `Arr1 -> Arr1 -> Arr1`. La définition de la composition de chaque catégorie doit alors obéir au schéma suivant:

$$\text{Si } (Cod1\ f) = (Dom1\ g) \text{ Alors «retourner } f \circ g \text{» Sinon } raise \text{ erreur}$$

Il s'agit ici seulement de discipline de programmation, que le système de typage de ML n'est pas capable de garantir. Une définition analogue est possible dans Coq en utilisant le type `Option` ci-dessous. Dans le cadre général, `(Option A)` correspond à l'union disjointe de `A` et d'une valeur particulière `None` indiquant l'absence de résultat.

```
Inductive Option [A:Type] : Type := None : (Option A) | Some : A -> (Option A).
```

La composition est alors de type `Arr1 -> Arr1 -> (Option Arr1)` et le schéma de définition devient:

$$\text{Si } (Cod1\ f) = (Dom1\ g) \text{ Alors «retourner (}\texttt{Some}\ f \circ g\text{)» Sinon } \texttt{None}$$

Dans le système de typage de Coq, il est possible d'imposer une preuve de correction de la composition stipulant que $f \circ g$ est `None` si et seulement si f et g ne sont pas composables. Remarquez aussi que l'utilisation de la conditionnelle «Si $(Cod1\ f) = (Dom1\ g)$ Alors...Sinon...» impose que l'égalité des objets soit décidable.

Reprenons notre définition de catégorie avec notre définition initiale de la composition, et spécifions le comportement de la composition par rapport à l'égalité de morphismes et aux fonctions $Dom1$ et $Cod1$.

```
Variable Cong1 : (f,g,f',g':Arr1) (p:(Cod1 f) =%S (Dom1 g))(q:(Cod1 f') =%S (Dom1 g'))
                 f =%S f' -> g =%S g' -> (Comp1 p) =%S (Comp1 q).
Variable Dom_Comp1 : (f,g:Arr1)(p:(Cod1 f) =%S (Dom1 g))(Dom1 (Comp1 p)) =%S  (Dom1 f).
Variable Cod_Comp1 : (f,g:Arr1)(p:(Cod1 g) =%S (Dom1 f))(Cod1 (Comp1 p)) =%S  (Cod1 g).
```

La composition de plusieurs morphismes devient très vite problématique, les preuves de composabilité devenant de plus en plus compliquées. Nous donnons à titre d'exemple l'énoncé de l'associativité. L'énoncé informel (qu'on trouve dans la plupart des livres) est similaire à celui de notre définition en 8.1: pour tous morphismes $f : a \to b$, $g : b \to c$ et $h : c \to d$, $f \circ (g \circ h) = (f \circ g) \circ h$. Il est exprimé dans une théorie (des ensembles) extensionnelle; l'énoncé en théorie intensionnelle serait: Si $f : a \to b$, $g : b' \to c$ et $h : c' \to d$, tels que $b = b'$ et $c = c'$, alors $f \circ (g \circ h) = (f \circ g) \circ h$. Dans Coq, l'énoncé est encore plus obscur car les objets domaine et codomaine d'un morphisme ne font partie de son type et ne peuvent être manipulés que par les fonctions $Dom1$ et $Cod1$. De plus la composition dépend de la preuve de composabilité, d'où les définitions intermédiaires `left` et `right`.

```
Lemma left : (f,g,h:Arr1)(p:(Cod1 f) =%S (Dom1 g))(Cod1 g) =%S (Dom1 h) ->
             (Cod1 (Comp1 p)) =%S (Dom1 h).
Lemma right : (f,g,h:Arr1)(Cod1 f) =%S (Dom1 g) -> (q:(Cod1 g) =%S (Dom1 h))
              (Cod1 f) =%S (Dom1 (Comp1 q)).
Definition Ass1_law := (f,g,h:Arr1)(p:(Cod1 f) =%S (Dom1 g))(q:(Cod1 g) =%S (Dom1 h))
                       (Comp1 (right p q)) =%S (Comp1 (left p q)).
```

On définit ainsi, sur ce principe, une nouvelle notion de catégorie qu'on pourra d'ailleurs relier à l'ancienne définition. En effet, on définit un hom-Setoïde entre deux objets a et b par $Hom(a,b) = \{f \mid (Dom1\ f) = a \text{ et } (Cod1\ f) = b\}$. Dans l'autre sens, il suffit de définir l'ensemble des morphismes; pour cela, on encapsule les morphismes et ses objets dans une structure `Arrs`, et on les compare en utilisant `=%H`. Enfin on prend l'égalité de Leibniz comme égalité des objets.

```
Section arrs_setoid_def.
Variable C:Category.
Structure Arrs : Type :=
 {Dom   : C;
  Codom : C;
  Arrow : Dom --> Codom}.
Definition Equal_Arrs := [f,g:Arrs] (Arrow f) =%H (Arrow g).
@Definition Arrs_setoid : Setoid := Equal_Arrs_equiv.
End arrs_setoid_def.
```

10.4 Mizar

Avant la fin de cette thèse, nous aimerions parler plus longuement de Mizar. Il ne s'agit pas de comparer le système Coq étendu par nos outils avec le système Mizar[1]. Le choix de Mizar n'est pas fortuit. Mizar est en effet le «champion toutes catégories» des systèmes de preuves dédiés à la formalisation des mathématiques; sa très importante librairie de mathématiques formalisées en est une preuve édifiante. Cette librairie comporte en effet plus de 2000 définitions de concepts mathématiques, et 20000 théorèmes vérifiés.

Tout semble séparer les deux systèmes Coq et Mizar. Alors que l'objectif de Mizar est de fournir un environnement logiciel pour la rédaction d'articles mathématiques traditionnels, Coq semble préviligier la problématique de la certification de logiciels. Leurs fondements sont aussi différents: la logique classique avec la théorie des ensembles pour Mizar, et la logique intuitionniste avec la théorie des types pour Coq. Nous résumons dans le tableau ci-dessous les principales caractéristiques de ces deux systèmes.

	Coq	Mizar
fondement	théorie des types	théorie des ensembles
logique	ordre supérieur	1^{er} ordre
style de démonstration	procédural	déclaratif
automatisation	moyen	faible
interactivité	oui	faible (Batch)
extensibilité	oui	non

Partant des axiomes de la théorie des ensembles, et de quelques axiomes des réels, plus de 20000 théorèmes de différents domaines (algèbre, topologie, géométrie, théorie des catégories etc.) ont été prouvés. Le principal objectif du projet Mizar est de construire une encyclopédie complètement formelle des mathématiques.

La librairie de Mizar est organisée en articles. Tout article Mizar comporte deux partie: la déclaration de l'environnement et le texte proprement dit (*text proper*). L'environnement consiste en une liste de noms d'articles indiquant quels «objets» de la librairie Mizar, le *text proper* peut référencer. Ces objets sont de différente nature:

- les symboles, qui définissent le vocabulaire utilisable dans le *text proper*. Chaque symbole est défini avec une arité, une précédence, ainsi qu'un format (infixe etc.).
- les définitions et les théorèmes, qui servent à justifier les résultats énoncés dans le *text proper*. Mizar permet la définition de synonymes (plusieurs notations pour une même notion) ainsi que la surcharge de symboles.
- les *schémas*, qui sont des théorèmes particuliers dans lesquels des variables libres du second degré peuvent apparaître. Nous donnons en exemple le schéma de récurrence pour les entiers (nous omettons sa preuve).

```
scheme Ind { P[Nat] } :
```

[1] notamment car l'auteur n'est pas expert en Mizar.

```
    for k holds P[k]
      provided
   a1:   P[0] and
   a2:   for k st P[k] holds P[k + 1]
      proof let k; ...
```

Le *text proper* quant à lui, contient le développement mathématique. Il consiste en une séquence de définitions, de théorèmes et de schémas avec leur preuve. Les définitions incluent les définitions de fonctions, de prédicats et de structures. Une structure est une entité regroupant plusieurs champs accessibles par des sélecteurs (projections). Une structure peut avoir plusieurs «ancêtres» dont elle hérite les champs (on doit tout de même recopier ces champs dans la nouvelles structure).

Nous donnons l'exemple de la définition de la structure `CatStr` (définie dans [29]), correspondant à la notion de «pré-catégorie» (catégorie sans les propriétés catégoriques usuelles).

```
struct  CatStr
     <<Objects, Morphisms -> non empty set,
       Dom, Cod -> (Function of the Morphisms, The Objetcs),
       Comp -> (PartFunc of [: the Morphisms, the Morphisms :], the Morphisms),
       Id -> Function of the Objetcs, the Morphisms
      >>;
```

Bien que basé sur la théorie des ensembles, Mizar possède un système de types, intègrant aussi bien les structures mathématiques (comme `CatStr` ci-dessus) que les ensembles d'objets définis par compréhension. On peut ainsi définir de nouveaux *modes* (l'équivalent du mot «type» dans le jargon de Mizar) tels que `Real`, `Category` ou `Functor of C,D` (notez les paramètres `C` et `D` de mode `Category`). Par exemple, le mode `Category` est défini comme étant les objets de `CatStr` vérifiant les propriétés catégoriques usuelles.

```
definition let C be CatStr;
 attr C is Category-like means
:: CAT_1: def 8

     (for f,g being Element of the Morphisms of C holds
         [g,f] of dom(the Comp of C) iff (the Dom of C).g=(the Cod of C).f)
   & (for f,g being Element of the Morphisms of C
         st (the Dom of C).g=(the Cod of C).f holds
         (the Dom of C).((the Comp of C).[g,f]) = (the Dom of C).f &
         (the Cod of C).((the Comp of C).[g,f]) = (the Cod of C).g)
    &  (for f,g,h being Element of the Morphisms of C
         st  (the Dom of C).h = (the Cod of C).g &
             (the Dom of C).g = (the Cod of C).f
         holds  (the Comp of C).[h,(the Comp of C).[g,f]]
              = (the Comp of C).[(the Comp of C).[h,g],f] )
    &  (for b being Element of the Objects of C holds
         (the Dom of C).((the Id of C).b) = b &
         (the Cod of C).((the Id of C).b) = b  &
         (for  f being Element of the Morphisms of C st (the Cod of C).f = b
            holds (the Comp of C).[(the Id of C).b,f] = f ) &
         (for  g being Element of the Morphisms of C st (the Dom of C).g = b
```

```
            holds (the Comp of C).[g,(the Id of C).b] = g ) );
end;

definition
 mode Category is Category-like CatStr;
end;
```

Un exemple d'objet de mode `CatStr` nous est fourni lors de la preuve de correction du mode `Category-like CatStr` (cette preuve nous assure que la classe des structures de ce mode est non vide). La pré-catégorie exhibée en exemple est la pré-catégorie à un seul objet `0` et un seul morphisme `1 : 0 → 0`.

```
definition
 cluster Category-like CatStr;
   existence
    proof take C = CatStr<<{0},{1},{1}-->0,{1}-->0,(1,1).-->1,{0}-->1>> ;
     thus ...
```

Nous en venons maintenant au langage des preuves qui constitue la plus importante contribution du système Mizar. Une preuve Mizar se présente sous la forme d'un texte structuré suivant des formats prédéfinis. Chaque format correspond à une règle de déduction naturelle agrémentée de constructions en langage naturel (anglais). Les principaux formats sont:

- `for` x `be` A ; preuve de $P(x)$ est une preuve de $\forall x \in A.P(x)$ (soit «`for` x `being` A `holds` $P(x)$» dans le langage Mizar).
- `assume` A ; preuve de B est une preuve de $A \Rightarrow B$ (soit «A `implies` B» dans le langage Mizar).
- `take` a; preuve de $P(a)$ est une preuve de $\exists x \in A.P(x)$ (soit «`ex` x `being` A `st` $P(x)$» dans le langage Mizar).

Ainsi le script ci-dessous est une preuve de $\forall x \in A.\exists y \in B.P(x) \Rightarrow Q(y)$.

`for` x `be` A; `take` b; `assume` $P(x)$; preuve de $Q(b)$

Les étapes élémentaires d'une preuve sont démontrées dans Mizar en utilisant la construction:

$$A \texttt{ by } E_1, ..., E_n$$

L'énoncé A est considéré comme une «conséquence évidente» des faits référencés par les étiquettes E_1,...,E_n. Chacune de ces étiquettes référence soit un nom de théorème, soit une étape de la déduction actuelle. Le langage de preuves de Mizar est dit *déclaratif* car on n'explicite pas la manière dont on prouve A à partir des hypothèses E_1,...,E_n. Par opposition, le script de preuve de Coq est *procédural* car il consiste en une séquence de tactiques qui sont autant d'ordres précis à l'ordinateur sur la manière de construire une preuve.

D'autres constructions de preuves plus complexes existent comme le raisonnement par analyse de cas, la preuve par contradiction et le raisonnement équationnel. Nous incluons une preuve par raisonnement équationnel concernant les morphismes monics (toujours extraite de [29]).

```
definition let C,g;
   attr g is monic means
:: CAT_1:def 10
   for f1,f2 st dom f1 = dom f2 & cod f1 = dom g & cod f2 = dom g
         & (the Comp of C).[g,f1]=(the Comp of C).[g,f2]
   holds f1=f2;
end;

theorem
   for h being Morphism of a,b for g being Morphism of b,a
     st Hom(a,b) <> empty set  & Hom(b,a) <> empty set  & (the Comp of C).[h,g] = id b
    holds g is monic
 proof let h be Morphism of a,b; let g be Morphism of b,a such that
a1:  Hom(a,b) <> empty set  and
a2:  Hom(b,a) <> empty set  and
a3:  (the Comp of C).[h,g] = id b;
  now let c;
   let g1,g2 be Morphism of c,b such that
a4:  Hom(c,b) <> empty set  and
a5:  (the Comp of C).[g,g1] = (the Comp of C).[g,g2];
   thus g1 = (the Comp of C).[((the Comp of C).[h,g]),g1] by a3,a4,Th57
          .= (the Comp of C).[h,((the Comp of C).[g,g2])] by a5,a1,a2,a4,Th54
          .= (the Comp of C).[((the Comp of C).[h,g]),g2] by a1,a2,a4,Th54
          .= g2 by a3,a4,Th57;
  end;
  hence thesis by a2,Th60;
 end;
```

Remarquez que dans la définition de `monic`, aucun type n'est rattaché aux variables `C`, `g`, `f1` et `f2`. Mizar offre en effet la possibilité de réserver des identificateurs à des types. Si une variable a un identificateur réservé pour un type, et qu'aucun type explicite ne lui est rattaché, alors son type par défaut est le type pour lequel son identificateur a été réservé.

```
reserve B,C,D,E for Category;
reserve f,f1,f2 for Morphism of a,b;
reserve g,g1,g2 for Morphism of b,c;
```

Il est indéniable que les preuves Mizar sont beaucoup plus lisibles que les preuves Coq. On a vu (même sommairement) que les preuves Mizar s'inspiraient du langage des preuves présentées dans les ouvrages de mathématiques. Le langage de tactiques de Coq s'inscrit malheureusement dans une autre logique: lire un script de preuves Coq revient à comprendre le résultat de l'application de chaque tactique.

La puissance de Coq réside dans l'extensibilité de son langage de tactiques; de nouvelles procédures de preuves peuvent facilement (presque) lui être ajoutées. Par manque d'automatisation (la commande `by` réalise des étapes de preuves très simples), les preuves de Mizar sont extrêmement détaillées et longues. De plus, Mizar ne peut pas être étendu par de nouvelles procédures de preuves. Le meilleur des deux mondes est peut être possible; J. Harrison[71] a implémenté dans HOL un (mini) mode Mizar (ne reprenant pas toutes les subtilités de Mizar) dans lequel il est possible d'utiliser la construction `by` avec des tactiques de HOL.

L'activité de preuve en elle-même est différente dans les deux systèmes Mizar et Coq. Mizar n'admet que le mode *Batch* dans lequel l'utilisateur soumet son article à Mizar qui lui signale les erreurs à corriger. Le système Coq est lui, interactif: l'utilisateur construit interactivement ses preuves par application successive de tactiques.

10.5 Autres formalisations

Dans ML La formalisation la plus exhaustive de la théorie des catégories est celle de Burstall et Rydeheard dans [137]. Leur objectif ultime était de définir un cadre catégorique formel pour le développement de programmes et la spécification d'algorithmes. Nous citons tout particulièrement leur travail original concernant la dérivation d'un algorithme d'unification de termes à partir de constructions de colimites. En utilisant le langage SML (initialement HOPE), ils ont défini les types de nombreuses constructions catégoriques (catégories, limites, adjonctions, topos, etc.) et construit des fonctions pour notamment le calcul de limites et d'adjonctions dans différentes catégories. La validité de ces constructions est vérifiée informellement; en effet les axiomes associés à ces constructions ne peuvent pas être exprimés dans le système de type, trop faible, de ML. Ce système de type est d'ailleurs incapable de représenter fidèlement la nature d'application partielle de la composition; la «vérification informelle» de la composabilité des morphismes est laissée à la responsabilité de l'utilisateur (voir 10.3). A titre d'exemple, nous donnons ci-dessous leur définition de catégorie; il s'agit d'un 4-uplet de fonctions correspondant aux fonctions domaine, codomaine, identité et composition, et où `'o` et `'a` sont respectivement les types des objets et des morphismes.

```
datatype ('o,'a)Cat = cat of ('a->'o)*('a->'o)*('o->'a)*('a*'a->'a)
```

Par manque d'enregistrements dépendants (ou modules) et de hiérarchies d'univers, ML est aussi incapable de manipuler des structures «non-homogènes» comme les ensembles; car dans ce cas particulier, le type des éléments varie suivant l'ensemble (comme pour notre type des Setoïdes). Ainsi dans [137], la catégorie des ensembles n'est pas unique; elle est paramétrée par le type des éléments des ensembles.

Dans «Categorical ML» Certains des problèmes cités ci-dessus ont été résolus par [55] en définissant un nouveau langage, «Categorical ML» obtenu en étendant SML (ML avec modules) avec des structures catégoriques spécifiques: catégories, foncteurs, transformations naturelles et adjonctions. Chacune de ces structures a sa propre syntaxe pour définir des instances. Des niveaux d'univers supplémentaires sont aussi introduits pour permettre l'imbrication des structures. Une particularité intéressante de ce langage est la génération et la vérification automatiques des conditions de correction (sous forme d'équations) pour chaque instance des structures catégoriques. L'égalité utilisée est toutefois trop contraignante: deux objets sont égaux seulement s'ils sont la même instance d'une structure. Nous donnons comme exemple la définition de la catégorie des «petits» types (du plus bas univers). La signature (type) d'un module est introduite par le mot clé `sig`, alors que sa valeur l'est par `struct`. Quant à `structure`, `type` et `val`, ils servent à indiquer des champs de module dont la valeur est respectivement un module, un type ou une valeur simple.

```
category Type =
  cat
    object small_type = sig
```

```
                        type v
                      end
    arrow small_arrow = sig
                        structure Ts : small_type
                        and Tt : small_type
                        val Tfun : Ts.v -> Tt.v
                      end
    source(f) = f.Ts
    target(f) = f.Tt
    identity(T) = struct
                    structure Ts = T
                    and Tt = T
                    fun Tfun(x) = x
                  end
    compose(g,f,sharing g.Ts = f.Tt)
                  struct
                    structure Ts = f.Ts
                    and Tt = g.Tt
                    fun Tfun(x) = g.Tfun(f.Tfun(x))
                  end
end
```

Categorical ML engendre alors les conditions équationnelles requises pour que les fonctions données vérifient les axiomes d'une catégorie.

```
Type.source(Type.identity(a)) = a
Type.target(Type.identity(a)) = a
Type.source(Type.compose(f,g)) = Type.source(f)
Type.target(Type.compose(f,g)) = Type.target(g)
Type.compose(Type.compose(h,g),f) = Type.compose(h,Type.compose(g,f))
Type.compose(f,Type.identity(Type.source(f))) = f
f = Type.compose(Type.identity(Type.target(f)),f)
```

Dans GTTS La plus ancienne formalisation conséquente de la théorie des catégories en théories des types est, à notre connaissance, celle de R. Dyckhoff[60]. Elle a été réalisée dans un système expérimental «Göteborg Type Theory System» (GTTS), implémentant une théorie extensionnelle des types de Martin-Löf. R. Dyckhoff a étendu la théorie des types par de nouvelles constantes de types et de nouvelles règles d'inférence correspondant à des règles de formation, d'introduction, d'élimination et de calcul (dont les règles d'extensionnalité). Le fragment formalisé correspond à 40 règles, et comprend la définition des notions de catégorie, foncteur et transformation naturelle. Le seul exemple de catégorie est celui de la catégorie des foncteurs. Certaines preuves sont présentées incluant la preuve de réflexivité de l'isomorphisme et le fait que la composition de deux monics est un monic. Nous donnons ci-dessous quelques règles correspondant à la définition de catégorie. Les noms `U` et `U1` correpondent aux deux premiers niveaux de la hiérarchie d'univers.

```
CATform    ---------------
              CAT : U1
```

```
           O      : U
           M(X,Y) : U        [X:O, Y:O]
           i      : M(X,X)   [X:O]
           c(f,g) : M(X,Z)   [X:O, Y:O, Z:O, f:M(X,Y), g:M(Y,Z)]
           c(f,i) = f : M(X,Y)                  [X:O, Y:O, f:M(X,Y)]
           c(i,f) = f : M(X,Y)                  [X:O, Y:O, f:M(X,Y)]
           c(f,c(g,h)) = c(c(f,g),h) : M(W,Z)   [W:O, X:O, Y:O, Z:O,
                                                 f:M(W,X), g:M(X,Y), h:M(Y,Z)]
CATintr    ------------------------------------------------------------------
           Cat(O,M,i,c) : CAT


            C : CAT                                  Cat(O,M,i,c) : CAT
CATelim_Ob  -----------                  CATeq_Ob    ------------------------
            Ob(C) : U                                Ob(Cat(O,M,i,c)) = O : U


                                                     Cat(O,M,i,c) : CAT
            X : Ob(C)    Y : Ob(C)                   A : O    B : O
CATelim_Mor ----------------------       CATeq_Mor  ---------------------------------
            Mor(C,X,Y) : U                           Mor(Cat(O,M,i,c),A,B) = M(A,B) : U


                                                      Cat(O,M,i,c) : CAT
            f : Mor(C,X,Y)                            A : O    B : O   C : O
            g : Mor(C,Y,Z)                            f : M(A,B)    g : M(B,C)
CATeq_comp  ------------------------      CATeq_comp  --------------------------
            comp(f,g) : Mor(C,X,Z)                    comp(f,g) = c(f,g) : M(A,C)


            f : Mor(C,W,X)
            g : Mor(C,X,Y)
            h : Mor(C,Y,Z)
CATelim_Ass -------------------------------------------------
            comp(f,comp(g,h)) = comp(comp(f,g),h) : Mor(W,Z)
```

Il faut noter que l'ajout de nouvelles règles d'inférence peut compromettre la cohérence du système, ainsi une preuve méta-théorique de cohérence est nécessaire. Une autre approche, dite axiomatique, est brièvement décrite; elle correspond à l'utilisation de Σ-types, évitant l'extension de la théorie.

Dans Nuprl Une autre formalisation, menée dans une théorie similaire à celle de GTTS, a été réalisée par J. A. Altrucher et P. Panangaden [9] dans Nuprl. Plusieurs notions catégoriques ont été définies, menant à la preuve d'une version du théorème du foncteur adjoint de Freyd. Une seule direction du théoreme a été complètement réalisée correspondant à la construction de l'adjoint gauche. Il est toutefois difficile d'en dire plus puisque le papier [9] ne donne malheureusement que peu de détails sur la formalisation, et il ne reste plus de traces d'elle dans les archives de Nuprl. Voici ci-dessous la définition de catégorie où `U1` et `U2` sont des univers et où le produit dépendant `(x:A)B` est noté par `(x:A->B)` ou bien encore par `forall x:A.B` au niveau des propositions.

```
Category == Obj : U2
          # Mor : (Obj # Obj)->U1
          # Id : (o1:Obj->Mor(o1,o1))
          # o : (o1:Obj->o2:Obj->o1:Obj->Mor(o1,o2)->Mor(o2,o3)->Mor(o1,o3))
          # forall A,B:Obj.forall f:Mor(A,B).
            f o Id(A) {A,A,B} = f in Mor(A,B)
          # forall A,B:Obj.forall g:Mor(B,A).
            Id(A) o g {B,A,A} = g in Mor(B,A)
          # forall A,B,C,D:Obj.forall g:Mor(A,B).forall g:Mor(B,C).forall h:Mor(C,D).
            ((f o g {B,C,D}) o h {A,B,D}) = (f o (g o h {A,B,C}) {A,C,D}) in Mor(A,D)
```

Dans HOL Peu de notions catégoriques sont formalisées dans HOL. Dans Agerholm[4], ces notions sont formalisées en théorie des types et dans la théorie des ensembles (elle-même formalisée dans HOL), en vue de les comparer. À titre d'exemple, nous montrons la définition de catégorie.

```
DEF (Category)
  category(C_0:*o->bool,C_1:*a->bool,dom,cod,c) =
   (!f::C_1. (dom f)::C_0 /\ (cod f)::C_0) /\
   (!f g::C_1. (cod f = dom g) ==>
               let h = c g f in h::C_1 /\ (dom h = dom f) /\ (cod h =cod g)) /\
   (!f g h::C_1. (cod f = dom g) /\ (cod g = dom h) ==>
                 (c h (c g f) = c (c h g) f)) /\
   (!X::C_0. (?id::C_1. (dom id = X) /\ (cod id = X) /\
                  (!Y::C_0. (!f::C_1. (dom f = X) /\ (cod f = Y) ==> (c f id = f)) /\
                            (!f::C_1. (dom f = Y) /\ (cod f = X) ==> (c id f = f)))))
```

Dans Mizar Un grand nombre d'articles publiés dans *Formalized Mathematics*[29, 30, 31, 32, 33, 34, 35, 36, 37, 65, 66, 90, 92, 94, 112, 113, 114, 116, 146, 147, 148, 149, 150, 151, 158] sont consacrés à la formalisation de la théorie des catégories. Nous y remarquons beaucoup d'exemples de catégories mais peu de notions avancées telles que les limites et les adjonctions.

Il existe deux définitions différentes de la notion de catégorie dans Mizar. La première, due à C. Byliński[29] a été présentée dans la section 10.4. La seconde, due à A. Trybulec[149], est basée sur les Hom-sets. Une catégorie est un triplet $\langle O, \{\langle o_1, o_2\rangle\}_{o_1,o_2\in O}, \{\circ_{o_1,o_2,o_3}\}_{o_1,o_2,o_3\in O}\rangle$ avec $\circ_{o_1,o_2,o_3} : \langle o_2, o_3\rangle \times \langle o_1, o_2\rangle \to \langle o_1, o_3\rangle$ des opérateurs de composition vérifiant les conditions d'associativité et d'existence des identités.

```
struct(1-sorted) AltGraph
   <<carrier -> set,
     Arrows -> ManySortedSet of [:the carrier, the carrier:]>>;

struct(AltGraph) AltCatStr
   <<carrier -> set,
     Arrows -> ManySortedSet of [:the carrier, the carrier:],
     Comp -> BinComp of the Arrows>>;

definition
 mode category is transitive associative with_units (non empty AltCatStr);
end;
```

Autres T. Mohri[111] a comparé les trois débuts de formalisations dans Mizar, Nuprl et Coq. Il a réalisé la preuve du théorème de Yoneda dans ces trois systèmes.
Enfin, S. Agerholm[5] et I. Beylin et P. Dybjer[24] ont vérifié le théorème de cohérence pour les catégories monoïdales dans respectivement HOL et ALF. Dans [6], ces deux formalisations sont comparées.

Conclusion

Nous concluons cette thèse en énumérant nos réalisations, en dégageant une méthodologie pour la formalisation des mathématiques dans les théories de types, et enfin en discutant les travaux futurs potentiels.

Les principales contributions de cette thèse sont:

- Élaboration d'un mécanisme d'héritage basé sur les coercions implicites, et dont nous avons montré la correction. Ces coercions sont des fonctions reliant des classes. Les classes (éventuellement paramétrées) peuvent être des structures (enregistrements dépendants), des types inductifs ou des constantes définies. Nous avons aussi inclus deux classes abstraites SORTCLASS et FUNCLASS, respectivement classe des sortes et des fonctions. Parmi les caractéristiques de ce mécanisme, nous citons l'héritage multiple, les coercions opposées et la prise en compte de la règle de sous-typage contravariante à gauche des types fonctionnels.

- Formalisation d'une partie conséquente de la théorie des catégories. C'est à notre connaissance la plus importante formalisation de la théorie des catégories dans un système de preuves. Prévue initialement pour valider les outils dévelopés dans le cadre de cette thèse, cette formalisation en est devenue une contribution majeure. Cette formalisation comporte toutes les notions de base telles que les catégories, les foncteurs, les transformations naturelles, les limites et les adjonctions. Divers résultats ont été vérifiés dont le plus important est le théorème de Freyd pour l'existence d'un adjoint gauche. Notre développement a déjà été utilisé par d'autres personnes pour d'autres formalisations (notamment en France et au Portugal).

Des contributions moins importantes (mais cruciales pour mener à bien le projet de formalisation de la théorie des catégories) ont aussi été réalisées:

- Définition d'une syntaxe pour les enregistrements dépendants, automatiquement traduite en un type inductif approprié. Les projections aussi sont automatiquement construites.

- Description détaillée du mécanisme des sous-termes implicites conçu et implémenté dans Coq par Chet Murthy. Nous avons étendu ce mécanisme en développant une heuristique pour un meilleur traitement de l'unification en présence d'enregistrements dépendants. Cette extension permet de plus de surcharger des noms de fonctions. Nous avons enfin défini une nouvelle syntaxe implicite prenant automatiquement en compte les arguments implicites des applications de fonctions.

- Implémentation dans Coq d'une bibliothèque de tactiques de réécriture similaires à celles des systèmes HOL, Isabelle et Nuprl. Nous avons ainsi remédié à une lacune importante de Coq (du moins en partie). Nous rappelons ici qu'il s'agit d'une bibliothèque modulaire de stratégies

multi-relations, pouvant être combinées de différentes manières pour permettre à l'utilisateur de créer ses propres stratégies de réécriture.

À travers notre expérience de la formalisation de la théorie des catégories, nous pouvons dégager une méthodologie pour la formalisation de mathématiques (abstraites) en théorie des types avec types dépendants. Cette méthodologie s'appuie sur le langage des Setoïdes pour la représentation des ensembles (ou types extensionnels). Dans cette méthodologie, les structures mathématiques sont représentées sous forme d'enregistrements dépendants regroupant toutes leurs composantes (types, opérateurs et propriétés). Ces structures peuvent être étendues en les enrichissant par de nouvelles composantes. Le mécanisme de coercions implicites que nous avons développé, permet alors de définir une relation de sous-typage entre la structure étendue et la structure initiale. Notre développement a mis en évidence l'utilité pratique de diverses constructions comme les types dépendants, les constantes, les enregistrements dépendants et les univers flottants.

Cette méthodologie doit aussi s'appuyer sur un langage formel pour l'expression des textes mathématiques; ce langage doit être proche autant que possible des mathématiques informelles traditionnelles. Il est ainsi reconnu que l'abus de notation en mathématiques est indispensable. Mais nous pensons que l'abus de notation arbitraire est injustifié, car il existe des mécanismes génériques qui permettent d'utiliser des abus de notation uniformes et dont la désambiguation est mécanique. Les outils de coercions implicites et de sous-termes implicites (ainsi que les univers flottants) comptent parmi ces mécanismes; ils constituent des moyens disciplinés de simuler des abus de notation usuels en mathématiques informelles.

Durant le développement de la théorie des catégories, nous avons rencontré quelques difficultés qui sont autant de nouvelles voies à prospecter. La première difficulté est liée à la nature intensionnelle de l'égalité définitionnelle; l'introduction de concepts extensionnels dans les théories des types intensionnelles (tout en préservant leur décidabilité) est un sujet de recherche actif mais qui ne semble pas encore avoir de répercussions sur les systèmes de preuves. Il est aussi envisageable de prendre comme fondement pour la formalisation des mathématiques une théorie des ensembles elle-même formalisée dans une théorie des types intensionnelle. La deuxième difficulté est liée aux nombreuses incohérence d'univers rencontrées nous obligeant à dupliquer certaines définitions. Cette difficulté montre ainsi l'insuffisance du mécanisme des univers flottants et l'urgence de disposer du polymorphisme d'univers qui nous déchargerait des duplications de définitions. On ne sait pas encore implémenter efficacement un tel mécanisme; c'est à notre avis un problème encore mal compris.

Oui, la théorie des types avec types dépendants et hiérarchie d'univers est adéquate à la formalisation ... de fragments d'algèbre abstraite. Mais la question reste ouverte pour d'autres branches des mathématiques, manipulant beaucoup de structures concrètes. Qu'en est-il aussi des preuves combinant les techniques de récurrence et de réécriture? Beaucoup d'outils spécifiques restent à concevoir et à implémenter.

Bibliographie

[1] P. Aczel. *Galois: A Theory Development Project.* Turin workshop on the representation of mathematics in Logical Frameworks, January 1993.

[2] P. Aczel. *A notion of class for theory development in algebra (in a predicative type theory).* Presented at Workshop of Types for Proofs and Programs, Bastad, Sweden, June 1994.

[3] P. Aczel. *Simple overloading for type theories.* Privately circulated notes, 1994.

[4] S. Agerholm. *Experiments in formalizing basic category theory in higher order logic and set theory.* Draft, decembre 1995.

[5] S. Agerholm. *Formalising a proof of coherence for monoidal categories.* Draft, décembre 1995.

[6] S. Agerholm, I. Beylin et P. Dybjer. *A comparison of HOL and ALF formalizations of a categorical coherence theorem.* In TPHOL'96 proceedings, LNCS 1125, Springer-Verlag, Turku, Août 1996.

[7] G. Alexandre. *Une implémentation de Zermelo-Fraenkel en Coq.* Rapport technique LITP 96-05, 1996.

[8] R. Amadio, L. Cardelli. *Subtyping Recursive Types.* in ACM-TOPLAS, vol. 15, no 4, p. 575-631, 1993.

[9] J. A. Altucher and P. Panangaden. *A Mechanically Assisted Constructive Proof in Category Theory.* In proceedings of CADE 10, Springer-Verlag LNCS no 449, 1990.

[10] Anonymous. *The QED Manifesto.* Bundy, A. (ed.). 12th International Conference on Automated Deduction, Volume 814 of Lecture Notes in Computer Science, Nancy, France, Springer Verlag, 1994.

[11] R. Asperti and G. Longo. *Categories, Types, and Structures.* MIT Press, 1991.

[12] D. Aspinall et A. Compagnoni. *Subtyping dependent types.* In proceedings LICS96, 1996.

[13] A. Bailey. *Representing algebra in Lego.* Master Thesis, University of Edinburgh, 1993.

[14] A. Bailey. *Coercion Synthesis in Computer implementations of type-theoretic Frameworks.* In TYPES'96 proceedings, à paraître.

[15] M. Barr, C. Wells. *Category Theory for Computer Science.* International Series in Computer Science. Prentice Hall, 1990.

[16] G. Barthe. *Implicit coercions in type systems.* In TYPES'95 proceedings, LNCS 1158, Springer Verlag, 1996.

[17] G, Barthe. *Formalising algebra in type theory: fundamentals and application to group theory.* Draft.

[18] H. P. Barendregt. *The lambda calculus: its syntax and semantics.* Volume 103 of Studies in Logic and the Foundations of Mathematics. North-Holland, édition révisée, 1984.

[19] H. P. Barendregt. *Typed Lambda calculi.* In Handbook of Logic in Computer Science, S. Abramsky et al. (Eds.), Oxford University Press, 117-309, 1992.

[20] H. P. Barendregt. *The quest of correctness.* Draft, 1996.

[21] D. Basin. *Building Problem Solving Environments in Constructive Type Theory.* Ph.D. dissertation, Cornell University, Ithaca, NY, 1989.

[22] J. Bénabou. *Fibred Categories and the foundations of Naive Category Theory.* J. Symbolic Logic, 50, 1, pp. 10-37, 1985.

[23] G. Betarte. *Classes and overloading in type theory with record types.* Draft paper, 1995.

[24] I. Beylin et P. Dybjer. *Extracting a proof of coherence for monoidal categories from a formal proof of normalization for monoids.* In Types'95 proccedings. LNCS, 1996.

[25] E. Bishop. *Foundation of constructive analysis.* McGraw-Hill, 1967.

[26] N. Bourbaki. *Théorie des ensembles, fascicule de résultats.* Hermann, 1968.

[27] S. Boutin. *Reflexions sur les quotients.* Thèse Université Paris VII, 1997.

[28] V. Breazu-Tannen, C. Gunter, T. Coquand, A. Scedrov. *Inheritance as implicit coercion.* Information and Computation, Vol 93, pp 172-221, 1991.

[29] C. Byliński *Introduction to Categories and Functors.* In Formalized Mathematics journal, Vol. 1, 1989.

[30] C. Byliński *Subcategories and Products of Categories.* In Formalized Mathematics journal, Vol. 2, 1990.

[31] C. Byliński *Opposite Categories and Contravariant Functors.* In Formalized Mathematics journal, Vol. 2, 1990.

[32] C. Byliński *Category Ens.* In Formalized Mathematics journal, Vol. 3, 1991.

[33] C. Byliński et A. Darmochwal. *Comma Category.* In Formalized Mathematics journal, Vol. 4, 1992.

[34] C. Byliński *Products and Coproducts in Categories.* In Formalized Mathematics journal, Vol. 4, 1992.

[35] C. Byliński *Cartesian Categories.* In Formalized Mathematics journal, Vol. 4, 1992.

[36] C. Byliński *Categorial Categories and Slice Categories.* In Formalized Mathematics journal, Vol. 6, 1994.

[37] C. Byliński *Indexed Category.* In Formalized Mathematics journal, Vol. 7, 1995.

[38] L. Cardelli. *A polymorphic lambda calculus with Type:Type.* Rapport technique, DEC SRC, 1986.

[39] G. Chen et G. Longo. *Subtypin parametric and dependent types, an introduction.* Draft, 1996.

[40] A. Church. *A formulation of the simple theory of types.* Journal of Symbolic Logic, 5:56-68, 1940.

[41] L. Colson. *Représentation intensionnelles d'algorithmes dans les systèmes fonctionnels: une étude de cas.* Thèse de doctorat, Université Paris 7, 1991.

[42] R. Constable et *al. Implementing mathematics with NuPrl proof development system.* Prentice Hall, 1986.

[43] Th. Coquand. *An analysis of Girard's paradox.* Proceedings of LICS, Cambridge, Mass. July 1986, IEEE Press.

[44] T. Coquand et G. Huet. *Concepts mathématiques et informatiques dans le calcul des constructions.* Logic Colloquium'85. Elseiver Science Publishers, 1987.

[45] Th. Coquand. *Une théorie des Constructions.* Thèse de troisième cycle, Université Paris VII, 1985.

[46] Th. Coquand, G. Dowek, G. Huet et Ch. Paulin-Mohring. *The calculus of constructions: documentation and user's guide.* Rapport technique, projet Formel INRIA, juillet 1989.

[47] Th. Coquand, Ch. Paulin-Mohring. *Inductively defined types.* Proceedings of the International Conference on Computer Logic, P. Martin-Löf, G. Mints Eds. LNCS 417, pp. 50-66, 1988.

[48] C. Coquand, G. Huet. *The calculus of Constructions.* Information and Computation, 76:95-120, 1988.

[49] C. Cornes, J. Courant, J-C. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, B. Werner. *The Coq Proof Assistant Reference Manual, version 5.10.* Rapport technique INRIA 0177, 1995.

[50] C. Cornes, D. Terasse. *Automating Inversion of Inductive Predicates in Coq.* In Proceedings of TYPES Workshop, Turin 1995.

[51] C. Cornes. *Conception d'un langage de haut niveau de repésentation de preuves: Récurrence par filtrage de motifs, Unification en présence de types inductifs primitifs et Synthèse de lemmes d'inversion.* Thèse Université Paris VII, 1997.

[52] R. L. Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, 1993.

[53] P.-L. Curien. *Categorical combinators, sequential algorithms and functional programming.* Progression in theorical computer science. Birkhäuser, seconde edition, 1993.

[54] L. Damas et R. Milner. *Principal type schemes for functional programs.* In ninth symposium on Principles of Programming Languages, pages 207-212, 1982.

[55] E. Dennis-Jones et D. E. Rydeheard. *Categorical ML – category-theoretic modular programming.* Formal Aspects of Computing 5:337-366, 1993.

[56] G. Dowek. *Démonstration automatique dans le Calcul des Constructions.* Thèse Université Paris VII, 1991.

[57] G. Dowek. *La Logique.* Flammarions, coll. Dominos, 1995.

[58] M. Dummet. *Elements of intuitionnism.* Oxford Logic Series, Clarendon Press, Oxford, 1977.

[59] P. Dybjer et V. Gaspes. *Implementing a category of sets in ALF.* Draft, 1994.

[60] R. Dyckhoff. *Category theory as an extension of Martin-Löf type theory.* Internal Report CS 85/3, Dept. of Computational Science, University of St. Andrews, Scotland.

[61] P. Freyd. *Abelian categories: an introduction to the theory of functors.* New York, Harper and Row, 1964.

[62] W. Gehrke. *Decidability results for Categorical notions related to Monads by rewriting techniques.* PhD thesis, Johannes Kepler University, Linz, 1995.

[63] W. D. Goldfarb. *The undecidability of the second-order unification problem.* Theorical Computer Science, 13, pages 225-230, 1981.

[64] M. J. C. Gordon et T. F. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic.* Cambridge University Press, 1993.

[65] A. Grabowski. *On the Category of Posets.* In Formalized Mathematics journal, Vol. 8, 1996.

[66] A. Grabowski. *Examples of Category Structures.* In Formalized Mathematics journal, Vol. 8, 1996.

[67] M. Hagiya et Y. Toda. *On implicit arguments.* Rapport technique 95-1, 1995. Université de Tokyo.

[68] R. Harper and R. Pollack. *Type checking with universes.* Theoretical Computer Science 89, 1991.

[69] J. Harrison. *Metatheory and reflection in theorem proving: a survey and critique.* Rapport technique CRC-053, SRI Cambridge, 1995.

[70] J. Harrison. *Formalized mathematics.* Draft 1996.

[71] J. Harrison. *A Mizar mode for HOL.* Dans TPHOL'96 proceedings, LNCS 1125, Springer-Verlag, Turku, Août 1996.

[72] J. Harrison. *Proof style.* Dans TYPES'96 proceedings, à paraître.

[73] J. Harrison. *Theorem proving with the real numbers.* Phd of the University of Cambridge, 1997.

[74] M. Hofmann et T. Streicher. *A groupoïd model refutes uniqueness of identity proofs.* In proceddings of the 9th symposium on Logic in Computer Science (LICS), Paris, 1994.

[75] M. Hofmann. *Elimination of extensionality in Martin-Löf type theory.* Proceedings of workshop TYPES'93, Nijmegen, May 1993. In "Types for Proofs and Programs", Eds. H. Barendregt and T. Nipkow, LNCS 806, Springer-Verlag 1994.

[76] D. W. Howard. *The formula-as-types notion of construction.* In Seldin and Hindley[142], pages 479-490.

[77] G. Huet. *A unification algorithm for typed lambda calculus.* Theorical Computer Science, 1, pages 27-57, 1975.

[78] G. Huet. *Confluent reductions: abstract properties and applications to term rewriting.* Journal of the Association for Computing Machinery, 24(1), pp. 797-821, 1980.

[79] G. Huet et D. C. Oppen. *Equations and Rewrite rules: A Survey.* In R. Book ed., Formal Languages: Perspectives and Open problems, Academic Press, 1980.

[80] G. Huet. *A complete proof of the correctness of the Knuth-Bendix completion algorithm.* Journal of Computation Systems Sciences, 23:11-21, 1981.

[81] G. Huet. *Initiation à la Théorie des Catégories.* Notes de Cours, DEA Paris 7, Nov. 1985.

[82] G. Huet. *Formal structures for computation and deduction.* Papier de travail pour International Summer School on Logic of Programming and Calculi of Discrete Design, Marktoberdof, Germany 1986.

[83] G. Huet. *Extending the calculus of constructions with Type:Type.* Non publié, Avril 1987.

[84] G. Huet. *The constructive engine.* A Perspective in Theorical Computer Science, Commemorative volume for Gift Siromoney, R. Narasimhan Ed., World Scientific Publishing, 1989.

[85] G. Huet. *Cartesian closed categories and lambda-calculus.* In Logical Foundations of Functional Programming. University of Texas at Austin Programming Series, pp. 7-23, Addison Wesley, 1990.

[86] G. Huet. *A uniform approach to type theory.* Logic Foundation of Functional Programming, G. Huet Ed., Addison-Wesley, pp. 337-397, 1990.

[87] G. Huet, A. Saïbi. *Constructive Category Theory.* Présenté au CLICS-TYPES joint Workshop, 1995. À paraître dans un livre en l'honneur de Robin Milner.

[88] P. B. Jackson. *Enhancing the NuPRL proof development system and applying it to computational abstract algebra.* Ph.D. dissertation, Cornell University, Ithaca, NY, 1995.

[89] P. B. Jackson. *Exploring abstract algebra in constructive type theory.* In the proceedings of CADE-12, LNAI 814, 1994.

[90] W. Jaksch et C.Zinn. *Basic Properties about Functor Structures.* In Formalized Mathematics journal, Vol. 8, 1996.

[91] D. E. Knuth et P. B. Bendix. *Simple word problems in universal algebra.* In J. Leech editor, Computational Problems in Abstract Algebra, pp. 263-297. Pergamon Press 1970.

[92] A. Kornilowicz. *On the Baire Category Theorem.* In Formalized Mathematics journal, Vol. 9, 1997.

[93] A. Kornilowicz. *On the categories without the uniqueness of cod and dom. Some properties of morphisms and functors.* In Formalized Mathematics journal, Vol. 9, 1997.

[94] A. Kornilowicz. *The composition of functors and transformations in alternative categories.* In Formalized Mathematics journal, Vol. 10, 1998.

[95] Y. Lafont. *Equational reasoning with 2-dimensional diagrams.* Rapport technique, laboratoire de Mathématiques Discrètes, Université de Marseille, 1993.

[96] J. Lambek, P. J. Scott. *Introduction to higher-order categorical logic.* Number 7 in Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1986.

[97] J. Largeault. *L'intuitionnisme.* Presse universitaire de France, coll. «Que sais-je?», 1992.

[98] X. Leroy. *Typage polymorphe d'un langage algorithmique.* Thèse Université Paris VII, 1992.

[99] X. Leroy et P. Weis. *Le langage Caml.* InterEditions, Paris 1993.

[100] G. Longo. *Notes on the foundation of Mathematics and of Computer Science.* Rapport LIENS 90-20, 1990.

[101] G. Longo, K. Milsted et S. Soloviev. *logic of subtyping, extended abstract.* In proccedings of LICS'95, San Diego 1995.

[102] Z. Luo. *Coercive Subtyping.* In proccedings of CSL'96, Utrecht 1996.

[103] Z. Luo. *Computation and reasoning: a type theory for computer science.* No. 11 in International Series of Monographs on Computer Science. Oxford University Press, 1994.

[104] Z. Luo et R. Pollack. *LEGO proof development system: user's manual.* Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept. University of Edinburg, Mai 1992.

[105] D. Mackenzie. *Automation of proof: A historical and sociological exploration.* IEEE Annals of the History of Computing, 17(3):7-29, 1995.

[106] S. Mac Lane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[107] L. Magnusson, B. Nordström. *The Alf proof editor and its proof engine.* In the proceedings of Types for Proofs and Programs, LNCS 806, 1993.

[108] P. Martin-Löf. *An intuitionistic theory of types.* Bibliopolis, 1984.

[109] D. Miller. *A logic programming language with lambda-abstraction, function variables, and simple unification.* Journal of logic and computation, 1, 4, 1991 pp. 497-536.

[110] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML.* The MIT Press, 1991.

[111] T. Mohri. *On formalization of category theory.* A senior thesis, university of Tokyo, 1995.

[112] M. Muzalewski. *Categories of Groups.* In Formalized Mathematics journal, Vol. 3, 1991.

[113] M. Muzalewski. *Category of Rings.* In Formalized Mathematics journal, Vol. 3, 1991.

[114] M. Muzalewski. *Category of Left Modules.* In Formalized Mathematics journal, Vol. 3, 1991.

[115] R. P. Nederpelt, J. H. Geuvers et R. C. Vrijer eds. *Selected papers on Automath.* Volume 133 of Studies in Logic and the Foundations of Mathematics. North-Holland 1994.

[116] R. Nieszczerzewski. *Category of Functors between Alternative Categories.* In Formalized Mathematics journal, Vol. 9, 1997.

[117] B. Nordström, K. Petersson, J. Smith. *Programming in Martin-Löf's type theory.* OUP, 1990.

[118] S. Owre and N. Shankar and J.M. Rushby. *The PVS Specification Language.* Computer Science Laboratory, SRI International, 1993.

[119] Ch. Paulin-Mohring. *Inductive Definitions in the System Coq - Rules and Properties.* Proceedings TLCA 93, Ultrecht, March 93. LNCS 664, p 328–345.

[120] Ch. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse de Doctorat, Université Paris 7, January 1989.

[121] L. C. Paulson. *A higher order implementation of rewriting.* Science of Computer Programming, 3:119-149, 1983.

[122] L. C. Paulson. *Introduction to Isabelle.* Rapport technique 280, University of Cambridge, Computer Laboratory, 1993.

[123] F. Pfenning. *Elf: A language for logic definition and verified metaprogramming.* In proceedings of the Fourth Annual Symposium on LICS, Asilomar, Californie, 1989.

[124] F. Pfenning. *Logic programming in LF logical frameworks.* In proceedings of the first workshop on Logical Frameworks, 1991.

[125] F. Pfenning. *Unification and anti-unification ine the Calculus of Constructions.* In proceedins of Logic In Computer Science, 1991.

[126] F. Pfenning. *Refinement types for logical frameworks.* In Informal Proceedings of the Workshop on Types for Proofs and Programs, 1993.

[127] B. Pierce. *Basic Category Theory for Computer Scientists.* Foundations of Computing, MIT Press, 1991.

[128] A. Pitts. *Categorical logic.* In Handbook of Logic in Computer science, vol. VI. Oxford university Press, à paraître.

[129] A. Poigné. *Basic Category Theory.* In Handbook of Logic in Computer Science. Eds. S. Abramsky, D. M. Gabbay and T. S. E. Maibaum. Oxford Science Publications, pp. 413-641, 1992.

[130] E. Poll, P. Severi. *Pure type systems with definitions.* In Proceedings of LFCS'94, Vol 813, pages 316-328, St. Petersburg Russia.

[131] R. Pollack. *The theory of LEGO: a proof checker for the Extended Calculus of Constructions.* PhD thesis, university of Edinburgh, 1994.

[132] R. Pollack. *Implicit Synatx.* Informal Proceedings of First Workshop on Logical Frameworks, Antibes, Mai 1990.

[133] R. Pollack. *What is the meaning of a machine-checked proof?* Draft, 1996.

[134] R. Pollack. *On extensibility of proof-checkers.* In TYPES'94 proceedings, volume LNCS 996. P. Dybjer, B. Nordström and J. Smith eds. pages 140-161. Springer-Verlag.

[135] P. Rudnicki. *An overview of the MIZAR project.* Disponible dans le répertoire `pub/Mizar` par ftp à l'adresse `menaik.cs.alberta.ca`, 1992.

[136] B. Russel. *Mathematical logic as based on a theory of types.* American Journal of Mathematics, 30:222-262, 1908.

[137] D. E. Rydeheard and R. M. Burstall. *Computational Category Theory.* Prentice Hall, 1988.

[138] A. Saïbi. Axiomatization of a λ-calculus with explicit substitutions in Coq. Dans proceedings of the TYPES'94 (Bastad, Suède), LNCS 996, Juin 94.

[139] A. Saïbi. *Typing algorithm in type theory with inheritance.* In proccedings of POPL'97.

[140] A. Saïbi. *Réécriture dans Coq.* Dans les actes de JFLA'97, France 1997.

[141] A. Saïbi. *Formalisation constructive de la théorie des catégories.* En préparation.

[142] J. P. Seldin et J. R. Hindley eds. *To H. B. Curry: Essays in combinatory logic, lambda calculus, and formalisms.* Accademic Press, 1980.

[143] P. G. Severi. *Normalisation in Lambda Calculus and its relation to type inference.* Thèse de doctorat de l'université de Eindhoven, 1996.

[144] A. Tasistro. *Extension of Martin-Löf theory of types with record types and subtyping.* Draft paper, 1993.

[145] A. S. Toelestra et D. van Dalen. *Constructivism in mathematics, an introduction.* Vol. I, II. North-Holland, Amsterdam, 1986.

[146] A. Trybulec. *Natural transformations. Discrete categories.* In Formalized Mathematics journal, Vol. 2, 1990.

[147] A. Trybulec. *Isomorphisms of Categories.* In Formalized Mathematics journal, Vol. 3, 1991.

[148] A. Trybulec. *Some Isomorphisms Between Functor Categories.* In Formalized Mathematics journal, Vol. 4, 1992.

[149] A. Trybulec. *Categories without Uniqueness of cod and dom.* In Formalized Mathematics journal, Vol. 7, 1995.

[150] A. Trybulec. *Examples of Category Structures.* In Formalized Mathematics journal, Vol. 8, 1996.

[151] A. Trybulec. *Functors for Alternative Categories.* In Formalized Mathematics journal, Vol. 8, 1996.

[152] L. S. van Benthem Jutting, J. McKinna et R. Pollack. *Checking algorithms for Pure Type Systems.* In TYPES'93 proceedings, LNCS 806, pages 19-61. Springer-Verlag 1994.

[153] L. S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* In [115].

[154] D. Wätjan et W. Struckmann. *An algorithm for verifying equations of morphisms in a category.* Information Processing Letters, 14(3):104-108, Mai 1982.

[155] B. Werner. *Une théorie des Constructions Inductives.* Thèse Université Paris VII, 1995.

[156] B. Werner. *The encoding of Zermelo-Fraenkel set theory in Coq.* In TACS'97 proceedings, 1997.

[157] A. N. Whitehead et B. Russel. *Principia mathematica.* Cambridge University Press, 1925.

[158] M. Wojciechowski. *Yoneda Embedding.* In Formalized Mathematics journal, Vol. 9, 1997.

Index

@ (application d'une coercion), 92, 94
$\mathbb{C}$, 130
$(x \downarrow G)$ (catégorie comma), 171
$*$ (sorte), 29
$*$ (type produit ML), 115
$+$ (type union ML), 115
; (extension d'une séquence), 26
$=_\beta$ (β-équivalence), 25
$=_{\beta\delta}$ ($\beta\delta$-équivalence), 39
C° (catégorie duale), 138
[] (contexte vide), 26
[] (séquence vide), 26
$\Box$ (sorte), 29
$\overset{\frown}{C}$ (instance), 85
Δ (foncteur constant), 160
Δ-liaison, 87
Δ^+ (fermeture transitive d'un ensemble de coercions), 87, 94
$\mathcal{G} + f$ (graphe étendu), 96
$\mathcal{G}_n$ (chemins de coercions avec n paramètres), 95
Γ_c^{val} (valeur d'une constante dans un contexte), 39
Γ_x (type d'un nom dans un contexte), 26, 39
$\Pi_{i:I}$, 163
Σ-type, 54
$\Uparrow$, 49
$\_^\#$, 158
α-conversion, 24
β-équivalence ($=_\beta$), 25
β-réduction ($\rightarrow_\beta^*$), 25
β-réduction en une étape ($\rightarrow_\beta$), 25
$\beta\delta$-équivalence ($=_{\beta\delta}$), 39
$\bot$ (absurde), 32
$\cdot$ (composition de coercions), 94
$\cdot$ (concaténation de séquences), 26
$\cong$ (égalité définitionnelle), 59
$\cong$ (objets isomorphes), 139
$\downarrow$ (conversion), 42, 49
$\downarrow^\preceq$ (conversion avec cumulativité), 46
δ-réduction, 39
ϵ (co-unité), 179
$\epsilon(\Delta)$ (règle d'insertion de coercions), 87
$\equiv$ (égalité syntaxique), 24, 112
η (unité), 179
η-réduction, 25
$\geq$ (extension d'une signature), 73
$\in$ (prédicat d'appartenance à une séquence), 26
λ-calcul pur, 23
λ-terme, 23
λ-calcul simplement typé ($\lambda \rightarrow$), 25
$\lambda \rightarrow$ (λ-calcul simplement typé), 29
$\models$ (satisfaction de contraintes), 48
$\models$ (validité d'un graphe d'héritage), 94
$\neg$ (négation), 32
$>^{st}$ (relation), 100
$>^{st\beta\delta}$ (relation), 100
$>_p^{st\beta\delta}$ (relation), 100
$\langle h_i \rangle_{i:I}$, 163
$\langle\rangle$ (valeur de type *unit*), 115
π_1 (opérateur de projection), 55
π_2 (opérateur de projection), 55
$\preceq$ (cumulativité, 45
$\longmapsto$ (notation des coercions), 92
$\rightarrow_\beta$ (β-réduction en une étape), 25
$\rightarrow_\beta^*$ (β-réduction), 25
$\rightarrow_\iota$, 53, 57
$\rightarrow_\nu$ (effacement de coercions), 99
$\rightarrow_{\beta\delta}^{wh}$ (réduction de tête faible en une étape), 41
Σ^{ctxt} (contexte accessible d'une métavariable), 69
$\setminus$ (suppression d'un élément d'une séquence),

26
$\times$, 163, 175
$\vdash$ (adjonction), 177
$\vee$ (disjonction), 33
$\wedge$ (conjonction), 33
$\{*\}$ (ensemble singleton), 169
$f\{M : A\}$ (type de l'application d'une coercion), 92
$f\{M\}$ (type de l'application d'une coercion), 92
o (composition), 135
$\hat{U}$ (catégorie discrète), 164
? (terme inconnu), 67
`-->`, 135, 136
`:>`, 129
`==>`, 133
`=_F`, 143
`=_H`, 141
`=_M`, 133
`=_NT`, 145
`=_S`, 132
`>`, 129
`o`, 135, 136
`o_C`, 162
`o_NTh`, 151
`o_NTv`, 146
`=_NTH`, 155

$\mathcal{A}$ (axiomes de PTS), 27
$\mathcal{A}_0$ (sous-ensemble de sortes), 40
abstraction (λ-terme), 23
`Adj`, 177
`Adj1`, 180
`Adj1_law1`, 180
`Adj1_law2`, 180
`Adjoint`, 178
adjoint, 178
adjonction, 177, 179
  réécriture, 180
`A_eps`, 180
`A_eta`, 180
affectation de niveau, 48
ambiguïté de type, 47
`ApAphi`, 177
`ApAphi_inv`, 177
`ApNT`, 145
application
  λ-terme, 23
  binaire entre Setoïdes, 133
  entre Setoïdes, 132
  identité, 137
Apply (tactique), 116
arbre, 86
`AreIsos`, 139
`AreNatIsos`, 176
associativité, 155
`Assoc_law`, 136
Automath, 15
axiome
  de PTS ($\mathcal{A}$), 27
  variable globale, 44
`Axiom`, 128

betaC (conversion basique), 117
bien fondée (relation), 100
`Build_...`, 129
`Build_Adj`, 177
`Build_Comp`, 136
but, 115
  sous-but, 115

cône, 160
  limite, 161
Calcul des Constructions, 30
  Inductives, 51
Case...of...end (opérateur de filtrage), 52
`Cases` (filtrage), 128
cast, 38
`CAT`, 143
catégorie, 135
  comma, 171
  complète, 162
  des catégories, 143
  des foncteurs, 145
  des morphismes parallèles, 166
  des Setoïdes, 137
  diagramme, 146
  discrète, 164
  duale, 138
  grande, 144
  localement petite, 144

petite, 144
produit, 175
réécriture, 148
relativement petite, 144
taille, 144
Categorical ML, 197
Category, 136
Category', 143
Category'', 146
CC (Calcul des Constructions), 30
CC^{ω}, 45
CCI (Calcul des Constructions Inductives), 51
chaînage arrière, 115
champ, 56
chemin de coercions, 94
childclass, 86
Church Rosser, 112
cible (classe), 92
Cl (classe d'un terme), 91
Cl^* (classe d'un terme), 97
class, 86
classe, 85, 91
abstraite, 91
sous-classe, 94
classe-enfant, 85
classe-père, 85
CoAdjoint, 178
Coerce, 99
Coerce_FUN, 99
Coerce_SORT, 99
coercion, 85, 87, 92
entre fonctions, 90, 99
identité, 93
coercion from...to..., 86
Coercion, 129
cohérence, 49, 87, 96
Com_law, 172
Comma, 172
comma (catégorie), 171
complétion, 147
complète (catégorie), 162
Complete, 162
composition, 135
d'un cône avec un foncteur, 162
de coercions, 94
de foncteurs, 143
de transformations naturelles, 145, 151, 155
conclusion, 115
condition d'héritage uniforme, 92
Condition de l'Ensemble Solution, 169, 182
confluence, 27, 112
locale, 112
congruence, 117
constante, 37
définition, 44
globale, 38
locale, 38
constructeur
d'un enregistrement, 56
d'un type inductif, 52
contexte, 26, 38, 113
accessible, 68
bien formé, 26, 39, 88
d'hypothèses, 115
continu (foncteur), 163
Continuous, 163
contrainte, 48
contravariance, 90, 99
$Conv$ (conversion basique), 117
$conv$, 116
conversion, 42, 46, 49, 116
basique, 116
conversional, 118
Coq, 127
CoUA, 160
CoUA_diese, 160
CoUA_mor, 160
CoUA_ob, 160
cube de Barendregt, 29
Cum, 49
cumulativité ($\preceq$), 45
Curry-Howard (correspondance de), 32

déclaration de coercion, 96
déduction équationnelle, 147
$Decl$ (variables déclarées), 38
Déclaration (déclaration de variables), 44
Def (variables définies), 38
Définition (définition de constantes), 44

`Definition`, 128
Dep (prédicat), 71
diagramme catégorique, 146
`Discr`, 164
discrète (catégorie), 164
Dom (domaine d'une affectation de niveau), 48
Dom (fonction), 74
Dom (variables d'un contexte), 26, 38
`Dual`, 138
dualité (principe de), 159

`E1_diese`, 167
`E1_mor`, 167
`E1_ob`, 167
ECC, 45
échange (loi d'), 151
`E_diese`, 166
égalisateur, 165
égalité
  d'objets, 191
  définitionnelle, 34, 59
  dépendante, 111
  de classes, 92
  de foncteurs, 141, 143
  de Leibniz, 34, 110
  de morphismes, 141
  de transformations naturelles, 145
  propositionnelle, 34
  réflexion, 111
  syntaxique, 24, 112
`E_mor`, 166
`End`, 130
enregistrement dépendant, 56
ensemble dans Coq, 131
entiers de Church, 30
`E_ob`, 166
épi, 138
`Epic`, 138
`Epic_law`, 138
eq, 110
eq_ind, 110
`Equal_Functor`, 143
`Equal_hom`, 141
`Equalizer`, 166
`Equalizer1`, 167
`Equalizer1_fg`, 167
`Equalizer1_hom`, 167
`Equalizer2`, 167
`Equalizer_law1`, 165
`Equalizer_law2`, 166
`Equalizer_law3`, 166
`Equal_NT`, 145
equation
  équation
    entre morphismes, 146
équation, 70
`Equivalence`, 131
étiquette, 56
`Ext`, 133
extension d'une signature, 73
extensionnalité, 34

F (système), 29
F_ω (système), 30
FailC (conversion basique), 117
`Fcomp_law`, 140
fermeture
  réflexive symétrique transitive, 112
  réflexive transitive, 112
`Fid_law`, 140
filtrage, 128
FirstC (conversional), 119
Fixpoint (opérateur de point fixe), 53
flexible (terme), 71
`FMap`, 140
`FMor`, 141
`FOb`, 140
foncteur, 140
  égalité, 141, 143
  adjoint droit, 178
  adjoint gauche, 178, 181
  composition, 143
  constant, 160
  continu, 163
  Hom-foncteur, 176
  identité, 143
  projection comma, 172
  réécriture, 150
fonctionnel (PTS), 29

fonctions représentables, 30
forêt, 86
forme normale, 112
    de tête, 41, 58
FPA, 167
Freyd (théorème du foncteur adjoint), 182
FUNCLASS (classe), 91
FUNCT, 146
FUNCT, 146
Functor, 140
Functor_setoid, 143
FunDiscr, 165
FunSET2_l, 176
FunSET2_r, 176

graphe d'héritage, 95
GTTS, 198

héritage, 85
    multiple, 86
héritage simple, 86
hiérarchie d'univers, 45
HOL, 16, 200
Hom (Hom-foncteur), 176
Hom, 135, 136
Hom-foncteur, 176
Hypothesis, 130

Id (morphisme identité), 136
Id, 136
Id_..._..., 129
IdC (conversion basique), 117
Id_CAT, 143
Id_CatFunct, 145
identité
    application, 137
    foncteur, 143
    transformation naturelle, 145
morphisme, 136
Identity Coercion, 129
Idl_law, 136
Idr_law, 136
impicite
    argument, 128
implicite
    position, 78
    sous-terme, 152
    syntaxe, 67, 68, 74, 79
imprédicativité, 32
inclusion, 118
incohérence d'univers, 49, 143, 145
Inductive, 129
inférence de types, 37, 43, 44, 47, 50, 75, 103
Infer (inférence de types), 43, 44, 47, 50, 75, 103
Infix, 130
infixe (notation), 130
Initial, 139
initial (objet), 139
instance, 85
instanciation d'une métavariable, 69
intensionnalité, 34
Intro (tactique), 115
Isabelle, 16
IsCoUA, 160
IsInitial, 139
IsLimit, 161
Iso, 139
iso (isomorphisme), 139
isomorphes (objets), 139
isomorphisme, 139
    naturel, 176
IsTerminal, 140
IsUA, 158

jugement de typage, 26–28, 40, 41, 46, 56, 88

Knuth-Bendix (procédure), 147

LCF, 16, 114
LeftAdj, 178
Lemma, 129
lemme de Newman, 113
lemmes de génération, 30
Lim, 162
Lim_diese, 161
Limit, 162
limite, 161
    préservation de, 162
Limiting_cone, 162
Limit_law1, 161
Limit_law2, 161

list (type des liste ML), 115
`Local`, 130

métavariable, 68
    déclarée, 68
    définie, 68
    instanciation, 69
méthode, 85
`Map`, 132
`Map2`, 133
`Map_setoid`, 133
`method` , 86
Mizar, 15, 193, 200
ML, 114, 197
    définition de valeurs, 115
`Monic`, 138
monic, 138
`Monic_law`, 138
morphisme, 135
    égalité, 141
    épi, 138
    équation, 146
    co-universel, 159
    identité, 136
    iso, 139
    monic, 138
    universel, 157, 181
MV (ensemble des métavariables d'un terme), 69

`NatIso`, 176
`NatIso_inv`, 176
`NatIso_mor`, 176
naturalité (loi de), 144
New (coercions engendrées), 96
normal (terme), 25
normalisation
    en forme normale de tête faible (Whnf), 41, 42, 53, 58, 70
normalisation forte, 27
$NormalizeC$ (conversion), 120
NQTHM, 15
`NT`, 144
`NT_law`, 144
`NT_setoid`, 145
Nuprl, 199

nœthérianité, 112

`Ob`, 135
`Ob`, 136
objet, 135
    initial, 139
    limite, 161
    terminal, 140
opérateur
    de filtrage ($Case...of...end$), 52
    de point fixe ($Fixpoint$), 53
    de projection, 55
    de récursion, 31
optimisation des contextes valides, 43
$Orelse$ (tactical), 116
$OrelseC$ (conversional), 119
$OrThenC$ (conversional), 119

p_1, 163
p_2, 163
`PA`, 166
paire critique, 147
    lemme, 147
paire dépendante, 54
paramètre
    d'enregistrement, 56
    de classe, 89
    de type inductif, 52
paramètre de classe, 91
`Partial_equivalence`, 131
`Pd1_diese`, 165
`Pd_diese`, 163
Peter Aczel (proposition de), 85
`Pi`, 163
p_i, 163
`Pi1`, 165
polymorphisme, 29
    d'univers, 51, 144
position implicite, 78
précédence, 130
préservation
    de limites, 162
    du typage, 27
$Prem$, 26
`PROD`, 175
`Product`, 164

`Product1`, 165
`Product_law1`, 163
`Product_law2`, 163
produit, 163
    catégorie, 175
ProgressC (conversional), 119
`Proj`, 163
Proj (fonction), 77
`Proj1`, 165
projection, 57
    foncteur comma, 172
Prop, 32
propositions-as-types, 32
PTS, 27
    avec héritage, 87
    semi-plein, 40
PVS, 16

$\mathcal{R}$ (règles de PTS), 27
$\mathcal{R}_0$ (sous-ensemble de sortes), 40
$\mathcal{R}_1$ (sous-ensemble de couples de sortes), 40
réécriture, 154
    adjonction, 180
    catégorie, 148
    conditionnelle, 117
    foncteur, 150
    relation, 114
    système, 114
      canonique, 114
      complet, 114
    transformation naturelle, 150
réduction
    de tête faible en une étape ($\rightarrow_{\beta\delta}^{wh}$), 41
    relation, 112
réflexion, 120
réflexivité, 110
règle
    de PTS ($\mathcal{R}$), 27
    primitive, 115
raisonnement abstrait, 53
refl_equal, 110
`Reflexive`, 131
rel (type ML des relations), 116
relation, 112
    rel, 116
    canonique, 112
    complète, 112
    composition, 112
    dans Coq, 130
    inverse, 112
    réduction, 112
`Relation`, 131
Repeat1C (conversional), 119
RepeatC (conversional), 119
restreint (système), 98
ReTopC (conversional), 120
Rewrite, 116
`RightAdj`, 178
Rng (fonction), 74
rootclass, 86
ruleC (conversion basique), 117

$\mathcal{S}$ (sorte de PTS), 27
sémantique de Heyting, 32
séquence, 26
séquent, 115
satisfaction de contraintes ($\models$), 48
script, 114
section, 130
`Section`, 130
SET (catégorie des Setoïdes), 137
`SET`, 137
Setoïde, 132
    sous-Setoïde, 132
`Setoid`, 132
`Setoid'`, 143
`Setoid''`, 145
signature, 68, 113
    extension, 73
SimplEq (tactique), 120
SimplEqL (tactique), 120
SimplEqR (tactique), 120
singleton (type), 169
solution, 73
somme forte, 54
SORTCLASS (classe), 91
sorte de PTS ($\mathcal{S}$), 27
source (classe), 92
sous-but, 115
sous-classe, 94

sous-Setoïde, 132
sous-type, 85
spécification équationnelle, 146
$SSC1$ (Solution Set Condition), 169
$SSC2$ (Solution Set Condition), 182
`Structure`, 129
structure mathématique, 54
$SubC$ (conversional), 119
`SubClass`, 129
`SubSetoid`, 132
substitution, 24, 39, 113
    composition, 113
substitutivité, 110
    lemme, 118
surcharge, 80
$SweepDnC$ (conversional), 119
$SweepUpC$ (conversional), 119
symétrie, 110
`Symmetric`, 131
syntaxe implicite, 67, 68, 74, 79
système
    F, 29
    F_ω , 30
    T, 31
    de types purs (PTS), 27

T (système), 31
tactic, 115
tactical, 116
tactique, 115
taille des catégories, 144
terme, 113
    ambigu, 48
    bien typé, 27
    de ECC, 45
      avec univers flottants, 48
    de PTS, 28
    pour PTS
      avec constantes, 38
`Terminal`, 140
terminal (objet), 140
théorie
    des catégories, 125
    des types, 21
Then (tactical), 116
$ThenC$ (conversional), 118
`Theorem`, 129
transformation naturelle, 144
    égalité, 145
    composition horizontale, 151, 155
    composition verticale, 145
    identité, 145
    réécriture, 150
`Transitive`, 131
transitivité, 111
    lemme, 117
$TryC$ (conversional), 119
typage
    à la Church, 25
    à la Curry, 25
    explicite, 25
    implicite, 25
Type, 32
Type (prédicat), 26
type
    dépendant, 28
    inductif, 51
    inférence, 37
    liste, 115
    n-uplet, 115
    principal, 46
    produit, 115
    singleton, 169
    sous-type, 85
    union, 115
    unité, 115
    vérification, 37
$Type_i$ (univers), 45

`UA`, 158
`UA_diese`, 158
`UA_law2`, 158
`UA_law1`, 158
`UA_mor`, 158
`UA_ob`, 158
unificateur, 147
unification, 69, 71, 74, 77
Unify (unification), 71, 74, 77
unit (type unité ML), 115
univers, 45

flottant, 47
incohérence, 49, 143, 145
polymorphisme, 144
universalité, 157
universel (morphisme), 157, 181

vérification de type, 37
validation, 115
valide (ensemble de coercions), 94
variable, 23
déclaration, 44
de niveau, 48
globale (axiome), 44
liée, 24
libre, 24, 39, 69
`Variable`, 130
`Variables`, 130
VL (variables libres d'un terme), 24, 39
VL^* (ensemble des variables d'un terme implicite), 69
VN (ensemble des variables de niveaux d'un terme ou de contraintes), 48

Whnf (normalisation en forme normale de tête faible), 41, 42, 53, 58, 70
Whnf*, 49

Titre

Outils Génériques de Modélisation et de Démonstration
pour la Formalisation des Mathématiques en Théorie des Types.
Application à la Théorie des Catégories.

Resumé

L'objectif de cette thèse est de faciliter l'utilisation des systèmes de preuves basés sur la théorie des types pour la formalisation des mathématiques. Nous développons une méthodologie où les structures mathématiques sont représentées sous forme d'enregistrements dépendants regroupant toutes leurs composantes (types, opérateurs et propriétés). Cette méthodologie repose sur trois outils que nous avons développés et implémentés dans Coq (INRIA):

1. un mécanisme d'héritage basé sur les coercions implicites.
2. un mécanisme de sous-termes implicites.
3. une bibliothèque modulaire de tactiques de réécriture.

Ces outils ont pour objectif de combler la distance qui sépare les textes formels, compréhensibles par la machine, et les textes mathématiques usuels, plus lisibles et plus compréhensibles par l'humain.

Nous validons nos outils et notre méthodologie en formalisant une partie conséquente de la théorie des catégories dans Coq. Divers résultats ont été vérifiés dont le plus important est le théorème de Freyd pour l'existence d'un adjoint gauche.

Mots-clés

Théorie des types. Formalisation des mathématiques. Preuve formelle. Théorie des catégories. Systèmes de preuves. Héritage. Coercions. Réécriture. Langage implicite.

Title

Generic tools of Modelisation and Demonstration
for the Formalization of Mathematics in Type Theory.
Application to Category Theory.

Abstract

The goal of this thesis is to facilitate the use of proof systems based on type theory for formalization of mathematics. We develop a methodology where the mathematical structures are represented as dependant records including all their components (types, operations and properties). This methodology is based on three generic tools we developed and implemented in the Coq proof system (INRIA):

1- an inheritance mechanism based on implicit coercions.
2- an implicit subterms mechanism.
3- a modular library of rewriting tactics.

These tools try to fill the existing gap between the formal texts, readable by proof systems, and the usual mathematical texts, more comprehensible by humans.

We apply our tools and methodology by formalizing a significant parts of category theory in Coq. Several theorems are mechanically verified including the Freyd theorem for the left adjoint existence.

Keywords

Type theory. Mathematics formalization. Formal proof. Category theory. Proof systems. Inheritance. Coercions. Rewriting. Implicit Language.