Méta-programmation pour le transfert de preuve en théorie des types dépendants

Enzo Crance

COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

Nantes Université

THÈSE DE DOCTORAT DE

NANTES UNIVERSITÉ

ÉCOLE DOCTORALE N° 641
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Enzo CRANCE

Méta-programmation pour le transfert de preuve en théorie des types dépendants

Thèse présentée et soutenue à Nantes, le 19 décembre 2023
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes (LS2N)

Rapporteurs avant soutenance :

Sandrine BLAZY — Professeur des universités, Université de Rennes, IRISA
Anders MÖRTBERG — *Associate professor*, Université de Stockholm

Composition du Jury :

Président :	Guillaume MELQUIOND	Directeur de recherche, Inria, Université Paris-Saclay, LMF
Examinateurs :	Sandrine BLAZY	Professeur des universités, Université de Rennes, IRISA
	Anders MÖRTBERG	*Associate professor*, Université de Stockholm
	Karl PALMSKOG	*Senior lecturer*, *KTH Royal Institute of Technology*, Stockholm
	Nicolas TABAREAU	Directeur de recherche, Inria, Nantes Université
	Enrico TASSI	Chargé de recherche, Inria Sophia Antipolis Méditerranée, Université Côte d'Azur
Dir. de thèse :	Assia MAHBOUBI	Directrice de recherche, Inria, LS2N, Nantes
Co-dir. de thèse :	Denis COUSINEAU	Chercheur industriel (docteur), Mitsubishi Electric R&D Centre Europe, Rennes

Invité :

Mathieu BOESPFLUG — Chercheur industriel (docteur), Modus Create

Thèse de doctorat

Méta-programmation pour le transfert de preuve en théorie des types dépendants

Enzo Crance

13 février 2024

Encadrée par Assia Mahboubi et Denis Cousineau

À feu Marcel Coignard, dit « Monsieur l'abbé »,
Un homme extraordinaire et un puits de culture,
qui regrettait de ne jamais avoir pu explorer les sciences exactes.

Remerciements

En premier lieu, je souhaite remercier mes encadrants et les chercheurs avec lesquels j'ai travaillé au long de cette thèse, sans qui ce document n'aurait jamais pu voir le jour.

Merci Assia, pour ton honnêteté intellectuelle, ta rigueur scientifique sans faille, ta patience, ta pédagogie, ta modestie, ton humanité et ton ouverture d'esprit.

Merci Denis, pour ta prise de recul permanente, tes remarques pertinentes, ton optimisme, ta sympathie et tes précieux conseils sur le plan professionnel.

Merci à vous deux d'avoir mis au point ce sujet de thèse, de m'avoir fait confiance pendant ces trois années et de toujours avoir été très disponibles.

Merci Cyril, pour ton esprit vif, tes idées innovantes et tes plans sur la comète, sources de motivation dans la recherche.

Merci Chantal, pour nos échanges productifs et cette facilité que tu as à travailler en équipe. Merci également à Louise, Valentin et Pierre pour leur travail sur notre publication commune.

Merci Enrico, pour ton aide dans l'apprentissage d'Elpi, tes nombreuses réponses détaillées à mes questions techniques, et ton accueil chaleureux en ~~Italie~~ région niçoise.

Merci à Sandrine et Anders pour votre travail rigoureux dans la relecture de ce document. Merci à tous les membres de mon jury de thèse pour leur présence à ma soutenance et leur intérêt pour mon travail.

Ensuite, je souhaite remercier les autres ex-doctorants, doctorants et futurs doctorants qui ont croisé ma route. Merci à vous tous pour les discussions variées, les pauses café, les soirées au bar.

Merci Martin d'avoir été mon compagnon de route tout au long de cette thèse. Merci pour ton optimisme, ta gentillesse, ton humour et tous les services rendus, dont le fait de m'avoir nourri lorsque j'étais malade et (plus ou moins) coincé dans une résidence étudiante à Édimbourg. Peut-être saurons-nous un jour qui est le vrai thésard d'Assia.

Merci Hamza d'avoir partagé mon bureau pendant des mois et d'avoir fait office de canard en plastique lorsque j'étais bloqué et que j'avais besoin d'une oreille attentive pour comprendre et résoudre mes problèmes.

Merci Xavier, pour ton accueil à mon arrivée, ta grande culture en informatique tant théorique que pratique, tes bons conseils ainsi que ta résistance à toute épreuve à l'angoisse des dates butoirs.

Merci Theo (Winterhalter), Meven et Loïc, pour votre accueil chaleureux, votre patience et votre pédagogie dans le partage de votre connaissance pointue de la théorie des types, ainsi que votre disponibilité pour répondre à mes questions même après votre départ.

Merci Sidney d'avoir été mon compagnon de *trolling*, de coinche et de pinte dans les situations d'urgence.

Merci Pierre, pour ta capacité remarquable à penser constamment en dehors de la boîte, comme on dit outre-Manche.

Merci Nils de m'avoir prouvé que les Allemands n'étaient pas tous méchants. À plus sous le bus.

Merci Théo (Laurent), pour nos discussions fructueuses sur la paramétricité, aussi bien en milieu académique qu'au bar ou dans le train.

Merci Arthur (Correnson) d'avoir été mon binôme de chambre à plusieurs reprises aux JFLA et mon repère rennais au milieu de tant de Parisiens.

Merci à Thomas, Peio, Robin, Simon, Koen, Yee Jian, Yann, Tomas^2, Mara, Arthur, Virgil, Axel, Josselin, Amélie, Davide, Emily et Stéphane.

Je remercie aussi les chercheurs, académiques et industriels, qui m'ont beaucoup appris.

Merci Pierre-Marie, pour ta capacité remarquable (du numéro 3, du medium) à adapter ton vocabulaire à la culture scientifique et technique de ton interlocuteur et ainsi pouvoir expliquer aisément énormément de notions techniques. Merci pour ton humour et ta culture linguistique, politique, cinématographique et musicale, qui ont enchanté mon quotidien au long de ces années passées chez Gallinette. Vive le Luxembourg !

Merci Matthieu (Piquerez), pour ta vision du monde du travail, ton calme et ta façon de travailler sereinement dans l'informatique en continuant d'être un matheux.

Merci Yannick, pour ta grande culture scientifique, ta précision germanique dans le raisonnement et ta manie de demander sans cesse quand aura lieu la fête.

Merci Nicolas, pour tous tes efforts pour faire vivre l'équipe Gallinette, et pour tes explications sur la paramétricité univalente.

Merci Guillaume, pour ta pensée scientifique indépendante et ton habileté à parler au degré 1.5.

Merci du fond du cœur à Alan, sans qui je n'aurais peut-être jamais continué mon chemin dans la recherche.

Merci Florian, de m'avoir apporté des connaissances sur les nombres flottants et de m'avoir raconté autant de faissoleries croustillantes.

Merci David, pour la liberté que j'ai pu avoir pendant ma thèse CIFRE chez Mitsubishi Electric et pour la possibilité de publier nos résultats en *open source*.

Merci à Kazuhiko, Matthieu (Sozeau), Guilhem, Kenji, Marie, Gaëtan, Benoît, François, Éric, Delphine, Frédéric et Thomas.

Merci à Éric (Tanter) et Quentin pour les discussions intéressantes lors de vos visites à Nantes.

Merci à tous les autres chercheurs avec qui j'ai échangé pendant ces trois années.

Merci aux enseignants hors du commun qui ont contribué à la construction de la personne que je suis aujourd'hui : Marie-Jeanne, Marie-Odile, M. André, M. et Mme Hesry et M. Kerambellec.

Merci à mes professeurs de l'INSA, et tout particulièrement Pascal Garcia qui m'a permis de développer ce goût de l'informatique bien faite.

Sur le plan personnel, je souhaite remercier ma famille pour tout l'amour qu'elle m'a apporté. Il s'agit bien du « centre autour duquel tout gravite et tout brille », comme écrivait Victor Hugo.

Merci à mes parents, ma sœur Sarah, mon frère Roman, mes grands-parents, mes oncles et tantes, mes arrière-grands-parents et mes cousins.

Merci à Octave et Julia qui font maintenant partie de la famille.

Merci à ma belle-famille : Jérôme, Nathalie, Alexis et Clarisse.

La rédaction d'une thèse est une rare occasion de pouvoir rédiger un texte qui persistera à travers le temps. Je profite donc de cette occasion pour adresser une pensée chaleureuse à mes potentiels enfants, neveux, nièces, et tous les descendants de ma famille qui pourraient tomber sur ce document un jour.

Je remercie également mes amis pour leurs encouragements et leur présence à mes côtés.

Merci à mes amis de longue date, Martin, Baptiste, Hugo, Timothé, Florent et mon petit hyyu, pour tous ces moments passés ensemble.

Merci à mes collègues de l'INSA de m'avoir si bien accompagné pendant mes études. Merci en particulier à Nominoë, Alexis, François, Gaël, Lucas et Antoine. À nos parties de cidre-pong et aux fous rires qui vont avec. Aux expatriés ou futurs expatriés : que la Bretagne et la France vivent dans vos cœurs, où que vous soyez dans le monde, et à nos retrouvailles.

Merci Thibault, la « pourriture communiste », toujours disponible pour aller boire une bière et refaire le monde.

Mes salutations fraternelles au camarade Franouch.

Merci à Gaël, Antoine, Taha, William, Olivier et Julien, pour nos échanges de conseils au long du chemin du doctorat.

Enfin, un énorme merci à Séverine, celle qui a passé le plus de temps avec moi pendant ces années.

Merci pour ton soutien sans faille.

Merci pour les balades et les vacances qui m'ont permis de changer d'air.

Merci d'avoir ramené Trusty et Umi à la maison, et par la même occasion, énormément d'amour et de bons moments, des sorties variées, de magnifiques photos, et une saine occupation sur notre temps libre.

Merci d'avoir accepté de regarder Star Wars.

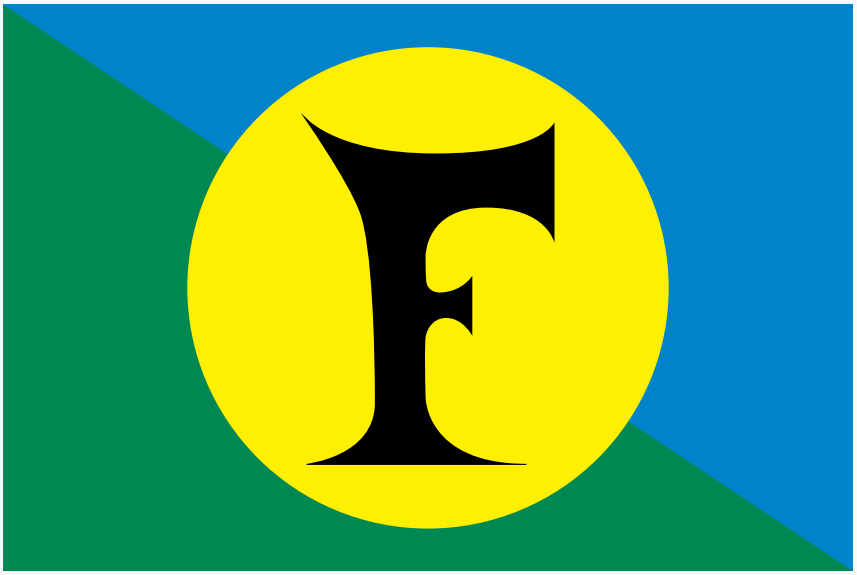
Merci d'avoir accepté que je passe parfois beaucoup de temps avec celle que tu appelles Thérèse et que tu peux à présent voir sous forme écrite et surtout terminée.

Je remercie chaleureusement tous ceux qui vont lire, citer, reprendre et faire vivre cette thèse à l'avenir.

Empire de la Basse Chesnaie

Ministère des Technologies nouvelles

« Ah tiens, d'habitude ça marche. »



Ad Majorem Propriam Gloriam

Cette thèse a été rédigée entièrement par un humain, sans l’aide d’outils d’intelligence artificielle.

Table des matières

Table des matières

1 Introduction **1**
1.1 Courte histoire de la logique . . . 2
1.1.1 Fondements historiques . . . 2
1.1.2 Vers les mathématiques et l'informatique . . . 3
1.1.3 Mécanisation de la logique . . . 3
1.2 Assistants de preuve et automatisation . . . 4
1.2.1 Une fiabilité à toute épreuve . . . 4
1.2.2 Des preuves encore très manuelles . . . 5
1.2.3 Contributions de cette thèse . . . 5

L'ASSISTANT DE PREUVE COQ, EN THÉORIE ET EN PRATIQUE **7**

2 Introduction à l'assistant de preuve Coq **10**
2.1 Des types pour les preuves et les programmes . . . 10
2.1.1 Le λ-calcul pur . . . 10
2.1.2 Types simples et correspondance de CURRY-HOWARD . . . 11
2.1.3 Le Calcul des Constructions . . . 14
2.2 Un langage de programmation expressif . . . 15
2.2.1 Univers et polymorphisme . . . 16
2.2.2 Types inductifs . . . 17

3 Assistance à la preuve **24**
3.1 Inférence . . . 24
3.1.1 L'unification . . . 24
3.1.2 Inférence et polymorphisme *ad hoc* . . . 27
3.2 Tactiques et automatisation . . . 29
3.2.1 Le mode preuve . . . 29
3.2.2 Tactiques de preuve automatique . . . 32
3.3 Réécriture et transfert de preuve . . . 34
3.3.1 Réécriture . . . 34
3.3.2 Prolongement à l'équivalence . . . 35

4 Méta-programmation pour Coq avec Coq-Elpi **37**
4.1 Un méta-langage logique pour Coq . . . 37
4.1.1 Un héritage de la programmation logique . . . 37
4.1.2 Encodage des termes Coq . . . 39
4.2 Une boîte à outils . . . 41
4.2.1 Bases de données . . . 41
4.2.2 Création de commandes et tactiques . . . 42

TRAKT : TRANSFERT DE PREUVE PAR CANONISATION **45**

5 Canonisation d'énoncés : objectifs et situation actuelle **47**
5.1 Contenu du pré-traitement souhaité . . . 47
5.1.1 Pré-traitement des théories . . . 47
5.1.2 Statut de la logique . . . 49

5.1.3 Polymorphisme et types dépendants . . . 50
5.2 La famille `zify` : fonctionnalités et limites . . . 51
5.2.1 Pré-traitement modulaire de l'arithmétique . . . 51
5.2.2 Pré-traitement de la logique . . . 52
5.2.3 L'extension `mczify` . . . 53
5.2.4 Limites de `zify` . . . 54

6 Fonctionnement théorique **56**
6.1 Collecte des informations de l'utilisateur . . . 56
6.1.1 Plongements de types . . . 56
6.1.2 Plongements logiques . . . 57
6.1.3 Plongements de symboles . . . 58
6.1.4 Clés de conversion . . . 58
6.2 Algorithme de pré-traitement . . . 59
6.2.1 Gestion des quantificateurs universels . . . 59
6.2.2 Traitement des connecteurs logiques . . . 62
6.2.3 Pré-traitement spécifique à une théorie . . . 63
6.2.4 La tactique `trakt` . . . 65

7 Conclusion et perspectives **67**
7.1 Écosystème d'automatisation dans Coq . . . 67
7.1.1 De la nécessité du pré-traitement . . . 67
7.1.2 Transformations modulaires de la tactique `scope` . . . 69
7.2 Succès de l'outil . . . 71
7.2.1 Exemples de buts traités . . . 71
7.2.2 Intégration de Trakt à d'autres outils . . . 72
7.3 Voies d'amélioration . . . 74
7.3.1 Polymorphisme et types dépendants . . . 74
7.3.2 Architecture du pré-traitement . . . 74

Trocq : transfert de preuve par paramétricité **76**

8 La paramétricité en théorie des types dépendants **79**
8.1 Motivation et définition . . . 79
8.1.1 Typage et propriétés des λ-termes . . . 79
8.1.2 Traduction de paramétricité brute . . . 80
8.1.3 Limites de la traduction brute . . . 81
8.2 La paramétricité univalente . . . 82
8.2.1 Enrichissement des témoins de paramétricité . . . 82
8.2.2 Équivalence de types et univalence . . . 83
8.2.3 Traduction de paramétricité univalente . . . 85
8.2.4 Omniprésence de l'axiome d'univalence . . . 87

9 L'équivalence de types, sur mesure **88**
9.1 Une nouvelle formulation de l'équivalence . . . 88
9.1.1 Décomposition de l'équivalence . . . 89
9.1.2 Recomposition hiérarchique des témoins de paramétricité . . . 92
9.2 Peuplement de la hiérarchie de relations . . . 93
9.2.1 Traitement des univers . . . 94
9.2.2 Traitement des produits dépendants . . . 94
9.2.3 Le cas des produits non dépendants . . . 95

10 Un calcul pour le transfert de preuve **96**
10.1 Séquents de paramétricité brute . . . 96

10.2 Séquents de paramétricité univalente . . . 98
10.3 Théorie des types annotés . . . 99
10.4 Le calcul TROCQ . . . 100
10.5 Constantes . . . 102

11 Conclusion et perspectives **104**

IMPLÉMENTATION D'OUTILS DE PRÉ-TRAITEMENT EN COQ-ELPI **106**

12 Architecture logicielle d'un greffon de pré-traitement **108**
12.1 Base de connaissances utilisateur . . . 109
12.1.1 Utilisation des bases de données COQ-ELPI . . . 109
12.1.2 Stockage des termes COQ . . . 110
12.2 Traversée du but initial . . . 110
12.2.1 Une tactique de traduction en COQ-ELPI . . . 111
12.2.2 TRAKT : leçons d'une première tentative . . . 112

13 Implémentation d'un greffon de paramétricité **115**
13.1 Générer et habiter la hiérarchie de paramétricité . . . 116
13.1.1 Génération de la hiérarchie et mise en place du greffon . . . 116
13.1.2 Flexibilité des témoins de paramétricité . . . 118
13.2 Implémentation de la relation de paramétricité . . . 119
13.2.1 Des règles d'inférence au programme logique . . . 120
13.2.2 Fonctionnalités utiles de COQ-ELPI . . . 122
13.3 Inférence de classes de paramétricité . . . 123
13.3.1 Définition du problème . . . 123
13.3.2 Solution retenue dans TROCQ . . . 126
13.3.3 Implémentation . . . 127
13.3.4 Affaiblissement et sous-typage . . . 129
13.4 Polymorphisme d'univers . . . 129
13.4.1 Levée de l'ambiguïté typique . . . 130
13.4.2 Univers algébriques et univers liés . . . 132

Conclusion et perspectives **134**

Références

Introduction 1

1.1 Courte histoire de la logique . 2
1.1.1 Fondements historiques 2
1.1.2 Vers les mathématiques et l'informatique 3
1.1.3 Mécanisation de la logique . . 3
1.2 Assistants de preuve et automatisation 4
1.2.1 Une fiabilité à toute épreuve . 4
1.2.2 Des preuves encore très manuelles 5
1.2.3 Contributions de cette thèse . 5

Il est d'usage, dans l'académie et dans l'industrie, de faire appel à différentes méthodes de *vérification* des résultats, dans le but d'augmenter la confiance de la société dans ces résultats. En effet, si la démonstration d'un théorème mathématique est comprise et acceptée par la communauté de son auteur, il peut alors être considéré comme acquis et réutilisé dans des travaux de recherche ultérieurs. Dans un milieu appliqué, des ingénieurs peuvent également utiliser des résultats théoriques dans la conception de produits industriels concrets. Les différents *tests* effectués sur ces produits avant qu'ils ne quittent les usines sont un autre moyen de garantir la confiance des futurs clients et utilisateurs. D'une manière générale, la vérification a pour objectif de gommer le caractère naturellement imparfait du travail humain, ainsi que de permettre à chaque génération de s'attaquer à des problèmes de plus en plus complexes, grâce aux travaux de la génération précédente auxquels elle accorde sa confiance.

> Nous tentons d'oublier que nous sommes des animaux, mais la nature nous le rappelle, parfois cruellement.
>
> Ieveni Sétine (personnage d'OSS 117)

Malheureusement, en dépit de toutes les procédures de vérification en vigueur, à maintes reprises des erreurs se sont glissées dans des travaux, qu'il s'agisse de preuves mathématiques ou de programmes informatiques. Ainsi, le 4 juin 1996, la fusée Ariane 5 termine son vol inaugural[1] au bout d'une trentaine de secondes par une explosion due à un *bug* informatique appelé *dépassement d'entier*,[2] causant une perte de centaines de millions d'euros. Plus récemment, en 2018 et 2019,[3] deux avions Boeing 737-MAX s'écrasent dans les minutes qui suivent leur décollage, à cause d'un logiciel conçu pour éviter les décrochages, à la fois défaillant et prioritaire sur le contrôle manuel de l'appareil. Cette fois, le bilan est humain et extrêmement lourd, s'élevant à plusieurs centaines de personnes.

1 : Vol 501.

2 : Sur une machine, les nombres sont représentés par une valeur binaire d'une certaine taille fixée afin de pouvoir les stocker en mémoire. Ainsi, ils ne peuvent dépasser une certaine valeur maximale. Un *dépassement d'entier* se produit lorsque le résultat d'une opération arithmétique dépasse cette valeur. La valeur binaire retenue pour le nombre devient alors très petite, ce qui est souvent source d'erreurs.

3 : Vols Lion Air 610 et Ethiopian Airlines 302.

> Le test logiciel montre la présence de *bugs*, pas leur absence.
>
> Edgser W. Dijkstra (informaticien néerlandais)

La conclusion évidente est que le test, *a fortiori* effectué par des humains, ne suffit plus, et que l'ère est à la vérification *formelle*, c'est-à-dire l'utilisation d'outils permettant de *certifier* l'absence d'erreurs dans une preuve mathématique ou dans un programme informatique. C'est la promesse des *méthodes formelles*, un domaine de recherche donnant un cadre théorique pour effectuer des preuves sur ordinateur. Dans cette veine, la fin du XX$^{\text{ème}}$ siècle voit apparaître une famille de logiciels appelée *prouveurs automatiques*. Il s'agit d'implémentations numériques d'algorithmes de recherche de preuve dans une théorie logique donnée, permettant à un humain d'entrer un énoncé à prouver et de laisser l'ordinateur déterminer s'il est vrai ou non.

> *Quis custodiet ipsos custodes ?*[4]
>
> Juvénal (poète romain)

4 : « Qui garde ces gardiens ? »

Cependant, une inquiétude demeure dans le fait que ces prouveurs ne sont eux-mêmes pas infaillibles car issus d'un code écrit par des humains, et de ce fait, pouvant contenir des erreurs. Les *assistants de preuve interactifs*, fleuron des

méthodes formelles, sont une réponse à ce problème. Il s'agit d'une famille de logiciels conçus autour d'un *noyau logique*, une petite quantité de code implémentant directement les règles d'une théorie logique et à laquelle les utilisateurs font confiance. Divers outils y sont mis à disposition de l'utilisateur pour effectuer des preuves, chaque preuve étant ultimement vérifiée par le noyau, garantissant à tout instant une confiance sans faille dans tous les développements effectués à l'aide d'un tel logiciel. Conceptuellement, c'est un compromis optimal entre l'humain et la machine. En effet, un ordinateur n'étant pas réellement capable de produire un raisonnement *original*,[5] cette tâche est laissée à l'humain. En revanche, la machine excelle dans l'application mécanique des règles logiques du noyau pour vérifier qu'une preuve est correcte, là où un humain pourrait commettre des erreurs.

5 : Malgré des résultats impressionnants récemment dans ce domaine, l'apprentissage automatique consiste essentiellement à compiler et exploiter de la meilleure manière possible une très grande quantité d'information. Cette information a beau être nettement plus massive que les connaissances d'un humain, il ne s'agit pas de donner à la machine les traits humains que sont l'originalité, la créativité, *etc.*

Le prix à payer pour cette sécurité supplémentaire est le caractère interactif des assistants de preuve. En effet, afin de pouvoir représenter des programmes informatiques et des théories mathématiques très abstraites au sein d'un assistant de preuve et garantir leur polyvalence, la théorie logique sous-jacente est souvent bien plus complexe que dans un prouveur automatique. De plus, pour avoir un noyau lisible et digne de confiance, les assistants de preuve ne bénéficient pas des heuristiques et optimisations agressives présentes dans le code des prouveurs automatiques. Par conséquent, l'utilisateur doit bien souvent expliciter des détails peu intéressants dans la confection des preuves, et la moindre automatisation est une tâche ardue.

Cette thèse s'inscrit dans une série de travaux de recherche dans l'automatisation des preuves, pour faciliter le travail des utilisateurs d'assistants de preuve, l'objectif final étant de répandre l'usage de ces outils à la place du test logiciel, partout où cela est possible et pertinent. Elle se situe donc à la frontière entre l'informatique et les mathématiques, et oscille entre contributions théoriques et travaux d'implémentation, car il est important de fournir des outils tangibles aux utilisateurs le plus rapidement possible. Pour situer ce travail dans un contexte large, dans ce chapitre, nous retraçons rapidement l'histoire de la logique (§ 1.1) avant de présenter les assistants de preuve, leur niveau d'automatisation actuel, ainsi que les contributions majeures (§ 1.2).

1.1 Courte histoire de la logique

La logique est l'étude des règles formelles à mettre en place pour déterminer si un raisonnement est valide. Cette discipline a été fondée pendant l'Antiquité, comme branche de la philosophie. Plus récemment, elle s'est rapprochée des mathématiques puis de l'informatique théorique en prenant la forme de systèmes logiques formels. Enfin, avec l'arrivée des ordinateurs, la logique s'est mécanisée. Dans cette section, nous donnons quelques détails sur ces différentes étapes.

1.1.1 Fondements historiques

En Occident, les travaux fondateurs dans le domaine de la logique remontent à la Grèce antique. En effet, cette discipline, alors appelée *λογική*, y tient une place centrale dans la vie publique, et de nombreux concepts encore utilisés aujourd'hui dans ce domaine proviennent des penseurs de cette époque, notamment Aristote et Euclide.

Dans son œuvre de l'*Organon*, Aristote définit la structure d'un raisonnement logique. Il y différencie les notions d'*être* et de *prédicat*,[6] de *cause* et de *conséquence*, ou encore d'*affirmation* et de *négation*. Il y présente aussi une construction logique appelée *syllogisme*, définie par l'enchaînement de deux prémisses et d'une conclusion par déduction.[7]

De son côté, Euclide introduit des définitions liées à la démonstration, comme les *postulats* ou *axiomes*, hypothèses non démontrées que l'on prend comme base d'un système logique, ou bien les *propositions* qui sont les énoncés que l'on peut prouver. À ceci s'ajoutent de nombreux théorèmes, notamment en géométrie et en théorie des nombres, pour former l'œuvre des *Éléments*, devenue ensuite une véritable référence académique. D'une manière générale, jusqu'à la fin du Moyen-Âge, la logique est enseignée avec ces œuvres fondatrices.

6 : Les *êtres* sont les entités et les *prédicats* représentent ce que l'on peut dire d'icelles.

7 : Le syllogisme le plus connu est probablement le suivant :

- Tous les hommes sont mortels ;
- Or Socrate est un homme ;
- Donc Socrate est mortel.

1.1.2 Vers les mathématiques et l'informatique

Après des siècles d'avancées scientifiques sans précédent à travers le monde, le XIX^{ème} siècle est marqué par une quête de formalisation de la logique. En effet, l'objectif est alors de construire un langage commun pour les mathématiques. C'est dans ce cadre que les langages *formels*[8] sont mis au point. Par exemple, Frege introduit dans l'*Idéographie* [1] le concept de *quantification*[9] et le *calcul des prédicats*,[10] encore utilisés de nos jours. Peano propose quant à lui une axiomatisation de l'arithmétique sur les entiers naturels [2], d'où provient le *raisonnement par récurrence* enseigné aujourd'hui en mathématiques. Enfin, Cantor crée la *théorie des ensembles* [3], permettant de décrire tous les objets mathématiques de l'époque dans un cadre commun. La logique, traditionnellement une discipline proche de la philosophie, devient alors une branche des mathématiques.

Au cours du XX^{ème} siècle, la théorie des langages formels donne naissance aux *langages de programmation*, utilisés pour implémenter des *algorithmes*.[11] Les règles associées à ces langages sont alors des règles de *calcul* permettant de véritablement exécuter les programmes pour obtenir un résultat. Cependant, plusieurs scientifiques identifient des liens permettant de donner à ces langages une interprétation logique : on appelle ce parallèle la correspondance de Curry-Howard. Cette découverte marque cette fois une convergence entre la logique et cette science naissante qu'est l'informatique théorique.

8 : Ce sont des langages définis par une syntaxe, c'est-à-dire un ensemble fini de symboles que l'on peut employer pour créer des formules selon des règles précises et explicites. On peut ensuite donner à ces symboles une interprétation logique et définir des règles pour construire un raisonnement, également explicites, afin de ne laisser aucune ambiguïté dans la manipulation du langage dans ce contexte.

[1] : Frege (1882), « Begriffsschrift : Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens »

9 : Il introduit deux *quantificateurs* :

- le quantificateur universel (« pour tout x ») ;
- le quantificateur existentiel (« il existe x »).

10 : On parle également de *logique du premier ordre*.

[2] : Peano (1889), « Arithmetices principia : Nova methodo exposita »

[3] : Cantor (1883), « Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen »

1.1.3 Mécanisation de la logique

Le développement des ordinateurs permet une mise en œuvre concrète de divers systèmes formels étudiés précédemment. Cette avancée annonce une nouvelle ère pour la logique et les mathématiques, dans laquelle l'humain et la machine travaillent de concert. Ainsi, le système Prolog [4] voit le jour au début des années 1970. Dans ce langage de programmation, le développeur définit une base de faits et de règles pour que des affirmations soient valides, et l'utilisateur pose des questions au système. Prolog a beaucoup été utilisé dans le traitement du langage, mais la possibilité de raisonner par récurrence sur des nombres et des arbres le rend également utilisable pour traiter des formules logiques. Ainsi, les conjectures de l'utilisateur peuvent être exprimées comme des requêtes dont le système pourra vérifier la validité en les exécutant.

11 : Un algorithme est une suite d'opérations à exécuter pour résoudre un problème particulier.

[4] : Colmerauer *et coll.* (1973), « Un système de communication homme-machine en français »

Les implémentations de systèmes logiques incluent également les prouveurs automatiques, dont le représentant le plus répandu est le solveur SAT.[12] Il reçoit en entrée une formule en *logique propositionnelle*[13] et détermine s'il en existe une *valuation*[14] qui la rend vraie. Les solveurs SAT connaissent un franc succès en raison de la grande variété de problèmes qu'ils permettent de représenter : planification, conception de circuits électroniques, cryptanalyse, *etc.* En ajoutant des symboles et un fonctionnement dédiés à une ou plusieurs théories, par exemple l'arithmétique ou les vecteurs de bits, on obtient un solveur SMT.[15] Cette extension élargit la portée des solveurs SAT à des problèmes de vérification de programmes.

12 : Satisfiabilité booléenne.

13 : Formée à partir de variables et de connecteurs de négation, conjonction ou disjonction.

14 : Affectation d'une valeur de vérité (vrai ou faux) à chaque variable de la formule.

15 : Satisfiabilité Modulo Théorie.

Des outils dédiés à la certification d'algorithmes et à la preuve de théorèmes mathématiques ont été développés par la suite, comme les outils de *model checking* à l'instar de TLA+ [5]. De tels logiciels offrent aux utilisateurs la possibilité de représenter les objets sur lesquels ils souhaitent raisonner ainsi que de décrire les énoncés qu'ils souhaitent prouver, tout ceci dans un langage formel expressif plus proche du langage naturel. Ainsi, ces outils permettent aussi bien de prouver la correction d'algorithmes concurrents et distribués, que de prouver qu'un automate ne peut jamais être dans un état invalide (par exemple, un ascenseur à l'arrêt entre deux étages). Certains outils comme Why3 [6] ou Liquid Haskell [7] permettent d'écrire des programmes dans un langage de la famille ML et les différents énoncés à prouver directement dans le même fichier. Dans ce cadre, une partie des preuves est déléguée à des solveurs SMT, offrant un certain niveau d'automatisation, si bien que pour certains cas d'utilisation, cette famille d'outils est considérée comme un bon compromis à l'heure actuelle.

[5] : Lamport (1994), « The Temporal Logic of Actions »

[6] : Filliâtre *et coll.* (2013), « Why3 — Where Programs Meet Provers »

[7] : Vazou (2016), *Liquid Haskell : Haskell as a Theorem Prover*

1.2 Assistants de preuve et automatisation

Malgré le niveau d'automatisation fourni par un solveur SMT, ces outils sont souvent utilisés comme des boîtes noires, n'étant souvent pas capables d'expliquer leur raisonnement. La confiance que l'on peut placer dans ces outils reste donc limitée, d'autant plus que des erreurs sont parfois découvertes dans leur code très optimisé mais difficile à mettre à jour de façon correcte. Dans les applications les plus critiques, ce niveau de fiabilité supplémentaire requis est fourni par les *assistants de preuve*, des logiciels à l'approche plus radicale mais plus manuelle pour l'utilisateur. Dans cette section, nous présentons ces différents aspects des assistants de preuve avant de lister les contributions majeures de cette thèse.

1.2.1 Une fiabilité à toute épreuve

Les assistants de preuve, ou prouveurs interactifs, sont des logiciels conçus pour effectuer des preuves sur la base d'une collaboration entre l'humain et la machine. Ces logiciels s'articulent autour d'un *noyau logique*, une petite quantité de code représentant les règles du système logique implémenté par l'assistant de preuve. Ce noyau est la seule base de code à laquelle l'utilisateur doit faire confiance, car il vérifie toutes les preuves effectuées dans l'assistant de preuve. Chaque preuve validée par le noyau peut alors être utilisé dans d'autres preuves, permettant de progressivement construire des bibliothèques entières de preuves dans un domaine donné.

Il existe une variété de systèmes logiques à la base des assistants de preuve. Par exemple, l'assistant de preuve Isabelle/HOL [8] est basé sur la logique d'ordre

[8] : Nipkow *et coll.* (2002), *Isabelle/HOL : a proof assistant for higher-order logic*

supérieur. Dans cette thèse, on s'intéresse à une famille d'assistants de preuve basée sur la théorie des types, un langage expressif, fruit d'une exploitation poussée de la correspondance de Curry-Howard, servant à la fois à programmer, exprimer les énoncés à prouver, et effectuer les preuves. Des exemples de logiciels de cette famille sont Agda [9], Lean [10], ou encore Coq [11], l'assistant de preuve sur lequel ce travail se base. Dans ce cadre, les preuves sont représentées par des termes de preuve, et la vérification du noyau est en réalité la vérification de typage[16] du langage sous-jacent.

[9] : Norell (2008), « Dependently Typed Programming in Agda »

[10] : De Moura *et coll.* (2021), « The Lean 4 Theorem Prover and Programming Language »

[11] : The Coq Development Team (2022), *The Coq Proof Assistant*

1.2.2 Des preuves encore très manuelles

La contrepartie d'un tel niveau de confiance est que les preuves redeviennent manuelles, car l'assistant de preuve demande que toutes les étapes d'une preuve soient rendues explicites par un terme de preuve. Or, le langage de l'assistant de preuve Coq a beau être très expressif, il reste très différent du langage naturel employé par les humains, et il est difficile pour un utilisateur d'entrer à la main des termes de preuve dans le logiciel. De plus, certaines étapes de preuve doivent être explicitées sans pour autant être intéressantes pour l'utilisateur. Par exemple, expliciter pour l'assistant de preuve des manipulations mathématiques pourtant triviales sur le papier, comme la commutativité de l'addition,[17] prend un temps considérable et ralentit l'utilisateur dans son travail de preuve.

Pour résoudre ces problèmes, le logiciel fournit une boîte à outils qui rapproche son langage formel du langage naturel utilisé sur les preuves papier. Ces outils incluent notamment des fonctionnalités d'inférence, grâce auxquelles l'utilisateur peut fournir des termes incomplets et laisser le logiciel en déterminer les morceaux manquants. Par exemple, l'utilisateur n'a pas besoin d'expliciter les types de toutes les valeurs, et l'assistant de preuve permet de définir des notations pour que les termes entrés par l'utilisateur se rapprochent des notations qu'il emploierait dans une preuve papier. Par ailleurs, des outils de preuve automatique sont mis à disposition de l'utilisateur pour prouver en quelques commandes une certaine classe d'énoncés correspondant aux étapes de la preuve sur lesquelles un mathématicien expert ne souhaite pas passer la majeure partie de son temps. Ces outils sont ce qui font de ces logiciels de véritables *assistants* à la preuve.

16 : Sur un ordinateur, les valeurs sont toutes représentées en binaire. Or, il serait impraticable d'écrire des programmes complexes ne manipulant que des nombres. Afin d'élever le niveau d'abstraction, de nombreux langages de programmation encodent en binaire différentes structures de données tout en offrant au développeur un moyen de les manipuler d'une manière différente que comme des nombres, en leur donnant un *type*. On peut alors manipuler des chaînes de caractères, des listes, *etc.* Les langages à valeurs typées sont alors munis d'un vérificateur de typage permettant de valider que les opérations associées à un type ne sont pas utilisées sur une valeur d'un autre type, ce qui ferait perdre son sens au programme.

17 : $a + b = b + a$.

1.2.3 Contributions de cette thèse

La mécanisation des preuves n'est possible qu'en mettant en place des solutions d'automatisation. Une approche est de connecter les assistants de preuve à des prouveurs automatiques, largement utilisés dans les méthodes formelles, afin de faciliter les preuves des énoncés qui sont à leur portée. Cela demande toutefois d'aligner les formules utilisées de part et d'autre en faisant correspondre la logique, les types de données, les opérations, *etc.* Plus généralement, le problème du *transfert de preuve*, à l'échelle d'un même formalisme logique, est celui de l'expression d'un même concept mathématique de plusieurs manières différentes, sans impact pour les preuves, c'est-à-dire que lorsqu'une preuve a été effectuée à l'aide d'une représentation d'un concept mathématique au sein de l'assistant de preuve, on souhaite ne pas avoir à refaire cette preuve manuellement en utilisant une autre représentation du même concept. Dans l'objectif de résoudre une instance du problème du transfert de preuve dans l'assistant de preuve Coq, nous proposons Trakt [12], un outil de pré-traitement des énoncés Coq faisant conver-

[12] : Blot *et coll.* (2023), « Compositional preprocessing for automated reasoning in dependent type theory »

ger les énoncés d'une théorie donnée vers une forme canonique qui correspond au format idéal attendu par un outil de preuve automatique pour cette théorie.

D'autres propriétés des langages formels utilisés dans les assistants de preuves peuvent être exploitées pour effectuer du transfert de preuve, comme la *paramétricité* [13]. Il s'agit d'une interprétation des types comme des relations, permettant de construire des outils qui lient un énoncé Coq à un énoncé associé, cible du transfert de preuve. Dans des versions riches de la paramétricité, comme la *paramétricité univalente* [14], on peut extraire du témoin[18] de la relation entre les deux énoncés un terme de preuve exploitable pour effectuer concrètement le transfert de preuve. Cependant, la paramétricité univalente introduit des axiomes dans Coq pour fonctionner correctement, y compris dans les cas où un traitement manuel n'utilisant pas d'axiome est possible. Nous présentons alors Trocq [15], un second greffon de transfert de preuve pour Coq, plus général que Trakt, ayant pour but d'égaler la puissance de la paramétricité univalente dans un maximum de cas, tout en analysant plus finement l'énoncé et en utilisant les axiomes dans un plus petit nombre de cas.

[13] : Reynolds (1983), « Types, Abstraction and Parametric Polymorphism »

[14] : Tabareau *et coll.* (2021), « The marriage of univalence and parametricity »

18 : Le *témoin* d'une relation R entre deux valeurs a et b est une preuve que ces deux valeurs sont bien liées dans la relation, c'est-à-dire une preuve de $R\,a\,b$.

[15] : Cohen *et coll.* (2024), « Trocq : Proof Transfer for Free, With or Without Univalence »

Ces deux outils prennent la forme de greffons pour l'assistant de preuve Coq. En tant que tels, ils reposent sur des techniques spécifiques de méta-programmation et leur implémentation pose des problèmes particuliers indépendants de leur conception théorique. Le méta-langage retenu pour ces implémentations est Coq-Elpi [16], un langage de programmation logique qui propose un haut niveau d'abstraction dans la manipulation de termes Coq.

[16] : Tassi (2018), « Elpi : an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect) »

Le reste de cette thèse est organisé en quatre parties : une introduction technique définissant les différents concepts manipulés par la suite (§ I), une présentation des deux prototypes d'outils de pré-traitement développés, Trakt (§ II) puis Trocq (§ III), ainsi que d'une partie dédiée aux questions d'implémentation (§ IV).

L'assistant de preuve Coq, en théorie et en pratique

Introduction

Les méthodes formelles sont des outils de choix pour augmenter le niveau de confiance que l'on peut placer dans les programmes informatiques. Par l'introduction d'un raisonnement rigoureux, elles apportent plus de garanties que des méthodes plus classiques d'assurance de qualité logicielle telles que le test logiciel, et sont plébiscitées dans le développement de logiciels critiques. Parmi les nombreux outils formels disponibles, ceux dont l'approche est la plus radicale sont les assistants de preuve, car dans de tels logiciels, les preuves elles-mêmes sont garanties correctes formellement. Au prix d'une confiance à placer dans un noyau logique composé de quelques centaines de lignes de code implémentant le système logique au cœur de l'assistant de preuve, on peut alors utiliser un tel logiciel pour vérifier la conformité de programmes vis-à-vis d'une spécification ou la validité d'une preuve mathématique.

Au sein d'un assistant de preuve, on utilise un langage de programmation pour décrire les objets mathématiques sur lesquels on souhaite raisonner[19] et un langage de preuve permettant de prouver pas à pas des énoncés exprimés à partir des différentes structures de données définies au préalable. Cette thèse est centrée sur l'assistant de preuve Coq [11], né en France dans les années 1980. Dans ce logiciel, l'expressivité du langage de programmation utilisé est telle qu'il est possible d'effectuer également les preuves dans ce même langage, incluant la vérification des preuves dans la vérification du typage. Cette expressivité est un réel avantage pour l'utilisateur, d'abord car il ne doit accorder sa confiance qu'à un seul langage, mais également car il est possible de représenter dans Coq des objets mathématiques très abstraits et de raisonner sur les théories associées, ce qui n'est pas le cas de tous les outils formels. En revanche, cette expressivité rend les preuves très difficiles à effectuer, car elles sont représentées par des termes de preuve dans ce langage très avancé, dont le très haut niveau d'abstraction dans les tâches de programmation devient un très bas niveau d'abstraction dans l'élaboration des preuves.

19 : Par exemple, un programme informatique est représenté par un arbre de syntaxe abstrait obtenu à partir de son code source.

[11] : The Coq Development Team (2022), *The Coq Proof Assistant*

L'automatisation des preuves dans Coq est donc la motivation de nombreux travaux de recherche en informatique. Ces travaux ont essentiellement pour but de construire une couche d'abstraction au-dessus du langage de programmation de Coq, afin que l'utilisateur n'ait pas à écrire de preuves manuellement dans ce langage. L'assistant de preuve possède alors un langage de *tactiques* permettant de faire progresser la preuve pas à pas, en exécutant des actions intuitives pour l'utilisateur et autorisées par Coq suite à la vérification d'un fragment du terme de preuve final, fourni par la tactique. Ces tactiques passant d'un état de preuve à un autre sont implémentées au niveau méta, à l'aide d'un des méta-langages disponibles dans Coq.

Certaines tactiques sont l'implémentation d'une procédure de décision, c'est-à-dire un algorithme permettant de décider si un énoncé contenu dans une théorie est vrai ou faux, en fabriquant automatiquement le terme de preuve associé. Par exemple, c'est le cas de la tactique `lia` [17] qui permet de prouver automatiquement un énoncé vrai qui appartient à la théorie de l'arithmétique de Presburger. D'autres tactiques reformulent l'énoncé à prouver pour le rendre plus simple à prouver manuellement ou plus adapté à d'autres tactiques d'automatisation. Parmi ces dernières existent les outils de transfert de preuve, permettant de reformuler des preuves ou des énoncés exprimés à l'aide d'un encodage d'un

[17] : Besson (2006), « Fast Reflexive Arithmetic Tactics the Linear Case and Beyond »

objet mathématique en les exprimant avec un autre encodage du même objet, que l'utilisateur juge plus adapté à la preuve qu'il est en train d'effectuer. Le transfert de preuve est une autre sorte d'abstraction dans les preuves, permettant de gommer les différences inhérentes à l'encodage d'un même objet mathématique de plusieurs façons différentes dans un assistant de preuve.

Les contributions principales de cette thèse sont la conception de deux outils de pré-traitement d'énoncés Coq, chacun répondant selon un angle précis au problème du transfert de preuve. Cette partie apporte les définitions techniques nécessaires pour suivre la présentation détaillée de ces outils. Nous y présentons l'assistant de preuve Coq, les fonctionnalités réelles d'assistance à la preuve que le logiciel implémente, ainsi que le greffon de méta-programmation pour Coq utilisé tout au long de cette thèse, Coq-Elpi [16].

[16] : Tassi (2018), « Elpi : an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect) »

Introduction à l'assistant de preuve Coq

2

Dans ce chapitre, nous présentons brièvement le langage de Coq pour introduire les concepts auxquels nous faisons référence dans le reste de ce manuscrit. Nous présentons d'abord graduellement le cœur de la théorie des types implémentée par le noyau et l'interprétation logique associée (§ 2.1), puis nous présentons les fonctionnalités que Coq ajoute au-dessus de cette base théorique (§ 2.2).

2.1 Des types pour les preuves et les programmes 10
2.1.1 Le λ-calcul pur 10
2.1.2 Types simples et correspondance de Curry-Howard . . . 11
2.1.3 Le Calcul des Constructions . . 14
2.2 Un langage de programmation expressif 15
2.2.1 Univers et polymorphisme . . 16
2.2.2 Types inductifs 17

2.1 Des types pour les preuves et les programmes

L'assistant de preuve Coq s'articule autour d'un noyau dont le composant essentiel est le vérificateur de typage d'un λ-calcul à types dépendants. Le logiciel permet à l'utilisateur de déclarer des types personnalisés appelés types inductifs, grâce auxquels il peut représenter les objets mathématiques sur lesquels il souhaite raisonner. Le langage de Coq peut aussi être interprété comme un système logique, ce qui en fait un langage de preuve. Cette section étudie ces différents aspects.

2.1.1 Le λ-calcul pur

Introduit dans les années 1930 par Church [18], le λ-*calcul* est un langage de programmation minimaliste dont la syntaxe définit seulement trois classes de termes : variable, abstraction, application.

[18] : Church (1933), « A set of postulates for the foundation of logic »

$$t, u \quad ::= \quad x \mid \lambda x.\, t \mid t\, u$$

L'objet de base d'un tel langage est la fonction, représentée par le cas de l'abstraction $\lambda x.\, t$ où x est une variable *liée*[1] et t le corps de la fonction. Ce qui fait de ce langage un calcul est la règle de β-réduction, permettant de calculer l'application d'une fonction à un argument[2] en effectuant une substitution de la variable liée de la fonction par l'argument :

$$(\lambda x.\, t)\, u \quad \rightsquigarrow \quad t[x := u]$$

1 : Une variable *liée*, par opposition aux variables *libres*, n'a de sens qu'au sein d'une fonction. Elle permet de représenter le futur argument qui sera potentiellement donné à cette fonction lors de son application, en liant cet argument à un nom.

2 : Un tel terme est appelé β-*rédex*.

La substitution est définie de manière à éviter les captures, c'est-à-dire qu'une variable libre avant substitution devienne liée après substitution, car cela donnerait un terme avec un sens différent de celui du terme attendu. Deux λ-termes qui ne diffèrent que dans le nom de leurs variables liées ont le même comportement vis-à-vis de la β-réduction. On dit alors qu'ils sont α-équivalents. On peut définir une équivalence entre termes appelée *conversion*[3] par la fermeture réflexive symétrique transitive de la β-réduction et de l'α-équivalence.

3 : On note $t \equiv u$ pour « t est convertible à u ».

Il est possible d'encoder de nombreux concepts de programmation à l'aide de λ-termes : entiers, booléens, paires, listes, arbres, *etc.* Il est même possible d'encoder la récursivité grâce à des combinateurs de point fixe, comme le combinateur Y :

$$Y \quad := \quad \lambda f.\, (\lambda x.\, f\, (x\, x))\, (\lambda x.\, f\, (x\, x))$$

En effet, $Y\ f$ se réduit en $f\ (Y\ f)$, puis $f\ (f\ (Y\ f))$, et ainsi de suite indéfiniment. Il est en fait possible d'encoder n'importe quel programme Turing-calculable dans un λ-terme.[4]

4 : La Turing-calculabilité d'un programme correspond à la possibilité de l'encoder sur une machine de Turing. Turing lui-même prouve que le λ-calcul est équivalent à sa machine [19], c'est-à-dire qu'il est Turing-complet.

La réduction d'un terme s'effectue en appliquant la règle de β-réduction autant que possible dans ce terme. À une étape donnée, il est possible que la règle puisse être appliquée à différents endroits. Le chemin emprunté lors de la réduction d'un terme n'est donc pas forcément unique. Si un chemin de réduction arrive à un terme qui ne contient plus aucun rédex, c'est-à-dire qu'il ne peut plus être β-réduit, on dit que ce terme final est une *forme β-normale*. La propriété dite de *normalisation* est l'existence d'une telle forme normale pour tout terme du calcul.[5] Comme le montre le combinateur Y, le λ-calcul ne respecte pas cette propriété car il permet d'encoder la *récursivité générale*, c'est-à-dire des boucles arbitraires.

5 : Si tous les chemins de réduction mènent à une forme normale, on parle de *normalisation forte*.

2.1.2 Types simples et correspondance de Curry-Howard

Dans l'objectif d'obtenir plusieurs propriétés intéressantes dont la normalisation, on peut restreindre l'ensemble des termes définissables dans le λ-calcul. On introduit alors des *types* [20], c'est-à-dire des annotations sur les termes, avec des *règles de typage* permettant d'inférer ou de vérifier ces types, et par extension de décrire les termes que l'on souhaite autoriser dans le calcul, un terme mal typé étant rejeté.

[20] : Church (1940), « A formulation of the simple theory of types »

Définition du λ-calcul simplement typé La fonction étant l'objet de base du λ-calcul, on définit un type comme étant soit une variable α appartenant à un ensemble fini de types de base, soit un type fonctionnel d'un domaine vers un codomaine à l'aide d'une flèche :

$$\tau \quad ::= \quad \alpha \mid \tau \to \tau$$

On annote alors les fonctions en ajoutant un type à leur variable liée :

$$t, u \quad ::= \quad x \mid \lambda x : \tau . t \mid t\ u$$

Un terme ne peut être bien typé que dans un contexte de typage, c'est-à-dire une liste d'associations entre des variables et des types :

$$\Gamma \quad ::= \quad \langle\rangle \mid \Gamma, x : \tau$$

Un jugement de typage $\Gamma \vdash t : \tau$ affirme que le terme t a le type τ dans le contexte Γ. Pour obtenir un jugement de typage sur un terme t, on compose des règles de typage ensemble par induction sur la syntaxe de t, pour construire une *dérivation de typage* dont la conclusion est le jugement de typage sur le terme t. Les règles de typage de ce λ-calcul dit *simplement typé* et noté $\lambda_{\to}$ sont alors les suivantes :

Le calcul $\lambda_{\to}$ permet d'écrire des programmes sensés dont on peut obtenir le résultat en calculant leur forme normale. Par exemple, on peut ajouter au calcul un type de base $\mathbb{N}$ pour représenter les nombres entiers naturels, ainsi que deux constantes $0 : \mathbb{N}$ et $S : \mathbb{N} \to \mathbb{N}$ pour représenter le zéro et le successeur, les deux composantes de base de l'arithmétique de Peano. En utilisant ces valeurs, on peut alors construire des programmes qui manipulent des entiers.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ (\textsc{Var}_{\to}) \qquad \frac{\Gamma, x : \tau \vdash t : \tau'}{\Gamma \vdash \lambda x : \tau.\, t : \tau \to \tau'}\ (\textsc{Lam}_{\to})$$

$$\frac{\Gamma \vdash t : \tau \to \tau' \qquad \Gamma \vdash u : \tau}{\Gamma \vdash t\, u : \tau'}\ (\textsc{App}_{\to})$$

Fig. 2.1 : Typing rules for $\lambda_{\to}$

La correspondance de Curry-Howard En associant la flèche $\to$ du type fonctionnel à l'implication logique, le λ-calcul simplement typé correspond en fait au système logique de déduction naturelle. Ce lien crucial entre logique et programmation est appelé correspondance de Curry-Howard. En effet, en considérant les types de base comme des variables propositionnelles, les types simples peuvent être vus comme des énoncés et les règles de typage comme des règles du système logique associé. Le contexte de typage correspond à un ensemble d'hypothèses, les variables libres et les constantes étant alors des axiomes (propriétés déclarées prouvées avant le début de toute démonstration). Ainsi, la règle $\textsc{Var}_{\to}$ décrit le cas des hypothèses : si une propriété est dans le contexte, alors elle est prouvable. La règle $\textsc{Lam}_{\to}$ permet d'introduire une implication en remarquant que si τ' est prouvable à partir d'une preuve de τ dans un contexte, alors dans ce contexte τ implique τ'. De plus, la règle $\textsc{App}_{\to}$ est la fameuse règle logique du *modus ponens* : si la proposition τ est prouvable et que τ implique τ', alors τ' est également prouvable. Si l'on augmente $\lambda_{\to}$ avec des constructions supplémentaires comme le produit[6] ou la somme,[7] on peut pousser la correspondance jusqu'au calcul propositionnel. En effet, un type $\tau_1 \times \tau_2$ correspond à une conjonction de deux propositions, et un type $\tau_1 + \tau_2$ correspond à une disjonction. Enfin, le faux (aussi noté $\bot$) est associé à ce qui n'est pas prouvable, c'est-à-dire aux types qui n'ont pas d'habitant.

Par ailleurs, cette correspondance entre les énoncés et les types est aussi une correspondance entre les preuves et les programmes. Dans une fonction $\lambda x : \tau.\, t$ de type $\tau \to \tau'$ qui construit un habitant de τ' à partir d'un habitant de τ, les termes x et t sont donc des termes de preuve. Un énoncé prouvé correspond à l'existence d'un terme qui habite le type de cet énoncé. Les constantes que l'on ajoute au contexte global sont alors des axiomes. En effet, une constante $k : \tau$ est une variable k dont on fait l'hypothèse qu'elle habite le type τ. L'énoncé correspondant à τ est donc prouvé, mais par un témoin qui n'a aucun contenu calculatoire. L'analogie s'étend naturellement aux autres constructions. Ainsi, un terme qui habite un type produit $\tau_1 \times \tau_2$ est une paire de preuves (t_1, t_2) où t_1 est une preuve de τ_1 et t_2 une preuve de τ_2, et un terme qui habite un type somme $\tau_1 + \tau_2$ est soit $\mathrm{inl}\, t_1$ où t_1 est une preuve de τ_1, soit $\mathrm{inr}\, t_2$ où t_2 est une preuve de τ_2.

6 : Le type produit permet de représenter des paires de valeurs. Une valeur de type $A \times B$ est l'association d'une valeur de type A et d'une valeur de type B. Il s'agit de la brique de base pour représenter en théorie le concept de tuple présent dans de nombreux langages de programmation. Son constructeur est le suivant :

$$(\cdot, \cdot) : A \to B \to A \times B$$

7 : La somme $A + B$ est une construction qui contient une valeur d'un type parmi deux possibilités A et B. Dans les langages à types algébriques, on s'en sert pour effectuer des disjonctions de cas, par exemple entre un résultat valide et une erreur (`result` en OCaml, `Either` en Haskell, *etc.*). Traditionnellement, une somme est construite par l'une des constantes suivantes :

$$\mathrm{inl} : A \to A + B$$
$$\mathrm{inr} : B \to A + B$$

Le cube de Barendregt Le λ-calcul simplement typé est le point de départ de nombreuses généralisations. En effet, il bénéficie de nombreuses propriétés intéressantes, dont la préservation du typage par la réduction,[8] l'unicité du typage, ou encore la *cohérence logique*.[9] En particulier, son système de typage rejette tous les termes dont la réduction ne termine pas, tels que les combinateurs de point fixe. Par exemple, la définition du combinateur Y applique un sous-terme x à lui-même, ce qui est impossible dans $\lambda_{\to}$, x ne pouvant à la fois avoir un type fonctionnel $\tau \to \tau'$ et le type de son domaine τ. En revanche, il manque d'expressivité par rapport au λ-calcul pur. Par exemple, dans le λ-calcul pur, le

8 : Si un terme $t : \tau$ se réduit en un terme t', alors $t' : \tau$.

9 : On ne peut prouver $\bot$ à partir d'un contexte vide dans le système logique.

terme $\lambda x.\, x$ représente la fonction identité et peut être appliqué à n'importe quel terme. En λ-calcul simplement typé, chaque fonction est définie sur un type unique. Si l'on souhaite l'appliquer à des termes de types différents, il faut alors construire une version distincte de l'identité pour chaque type utilisé. Différentes extensions de $\lambda_{\to}$ ont été conçues dans la deuxième moitié du XX$^{\text{ème}}$ siècle, dans l'objectif d'obtenir un langage de programmation ou système logique plus expressif, les deux points de vue étant disponibles grâce à l'isomorphisme de Curry-Howard.

Barendregt [21] propose en 1991 de représenter ces extensions comme les arêtes d'un cube[10] prenant $\lambda_{\to}$ pour origine et introduisant une forme d'abstraction différente sur chaque axe. Ces formes d'abstraction peuvent être décrites grâce au concept de *sorte*, c'est-à-dire le type d'un type. On introduit la sorte $\star$ et on déclare que tous les types simples ont pour sorte $\star$. Par exemple, on a $\mathbb{N} : \star$ et $\mathbb{N} \to \mathbb{N} : \star$. Ainsi, on possède un moyen de caractériser des constructions plus abstraites. Par exemple, un terme qui construirait un type à partir d'un autre type aurait la sorte $\star \to \star$. On introduit la sorte $\square$ comme sorte de toutes les sortes. Par exemple, $\star : \square$, $\star \to \star : \square$ et $\mathbb{N} \to \star : \square$. On identifie alors les différentes formes d'abstraction d'un λ-calcul par l'autorisation d'une ou plusieurs des classes de types fonctionnels dans le calcul, définies par une paire de symboles parmi $\star$ et $\square$, représentant la sorte du domaine et du codomaine du type fonctionnel autorisé :

- La classe $(\star, \star)$ décrit les termes qui dépendent de termes. Par exemple, c'est le cas de l'addition des entiers naturels $+_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Cette classe de termes est définissable dans $\lambda_{\to}$ et toutes ses extensions.
- La classe $(\square, \square)$, autorisée dans le calcul $\lambda\underline{\omega}$ et ses extensions, décrit les types qui dépendent de types. Il s'agit des *constructeurs de types*. Par exemple, on peut imaginer un terme list permettant de décrire le type d'une liste lorsqu'on lui fournit le type des élements de la liste. Ainsi, $\mathrm{list}\,\mathbb{N}$ est le type des listes d'entiers naturels.
- La classe $(\square, \star)$, autorisée dans le calcul λ_2 et ses extensions, décrit les termes qui dépendent de types. Il s'agit des *constructeurs* de termes appartenant à des types polymorphes. Par exemple, on peut imaginer une constante cons qui ajoute un élément à une liste. Cette constante dépend du type des éléments de la liste.
- La classe $(\star, \square)$, autorisée dans le calcul λ_P et ses extensions, décrit les types qui dépendent de termes. Cette abstraction est celle des *types dépendants*, la moins répandue dans les langages de programmation fonctionnelle. Elle permet de construire par exemple un type de tableau d'entiers de longueur fixe à l'aide d'une constante narray qui dépend d'un entier n décrivant la taille des tableaux qui auront le type $\mathrm{narray}\, n$.

Toutes ces abstractions peuvent alors être représentées sur trois axes à l'aide d'un cube illustré en Figure 2.2.[11] En généralisant ce cube à des calculs à plus de deux sortes, on obtient la famille des *Systèmes de Typage Purs*[12] [22, 23].

Chaque forme d'abstraction ajoute de l'expressivité au langage mais affaiblit potentiellement ses garanties théoriques. Les langages fonctionnels ayant fait l'objet d'une implémentation concrète et étant utilisés à un niveau industriel, comme Haskell, peuvent difficilement être placés sur le cube, car leur système de typage inclut de nombreuses fonctionnalités pragmatiques ne correspondant pas forcément à une extrémité du cube (types algébriques généralisés (GADT), polymorphisme à la Hindley-Milner, *etc.*) ou s'éloignant d'une interprétation logique

[21] : Barendregt (1991), « Introduction to generalized type systems »

10 : Alors appelé λ-cube ou cube de Barendregt.

11 : On ne représente sur le cube que les langages mentionnés ici.

12 : *Pure Type Systems*.

[22] : Berardi (1988), « Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube »

[23] : Terlouw (1989), « Een nadere bewijstheoretische analyse van GSTT's »

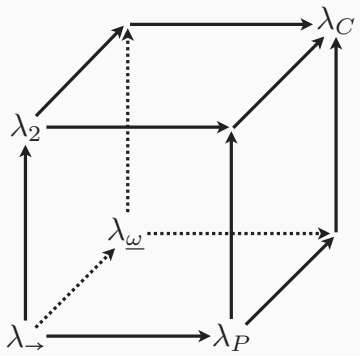


Fig. 2.2 : Cube de Barendregt.

(non-terminaison). Leur base théorique est cependant une restriction de λ_2 permettant de conserver la décidabilité du typage et l'inférence complète pour des raisons pratiques. L'assistant de preuve Coq, quant à lui, contient une implémentation du langage situé sur le point du cube le plus éloigné de l'origine, λ_C ou le Calcul des Constructions (CoC)[13] [24]. Il possède par conséquent toutes les abstractions présentées plus tôt en conservant une interprétation logique cohérente, celle du calcul des prédicats intuitionniste.

13 : Abréviation en partie à l'origine du nom de l'assistant de preuve.

[24] : Coquand *et coll.* (1986), « The calculus of constructions »

2.1.3 Le Calcul des Constructions

La syntaxe du Calcul des Constructions ajoute deux constructions à celle du λ-calcul simplement typé, le produit dépendant ou Π-type et l'univers :

$$A, B, t, u \quad ::= \quad x \mid \lambda x : A. t \mid t\,u \mid \Pi x : A. B \mid \square$$

Produit dépendant Le produit dépendant permet de décrire les types fonctionnels dépendants. En effet, dans le type flèche $A \to B$ du λ-calcul simplement typé, le codomaine B ne dépend pas lui-même du domaine A, ces termes étant définis tous deux en dehors de ce type fonctionnel. Dans un λ-calcul dépendant, on utilise un produit dépendant $\Pi x : A. B$ qui est un lieur à la manière d'une abstraction. Ainsi, tout comme le corps t d'une fonction $\lambda x. t$ est défini en fonction de l'argument x que l'on va lui fournir lors de son application, le codomaine B d'un produit dépendant peut dépendre de la variable liée x sur laquelle il quantifie.[14] En effet, l'interprétation logique du produit dépendant est la quantification universelle. Par exemple, en notant $\mathrm{narray}\ n$ un tableau d'entiers naturels de taille n, le terme construisant un tableau de taille n en répétant une valeur donnée a le type suivant :

$$\mathrm{nreplicate} \ : \ \Pi n : \mathbb{N}. \mathbb{N} \to \mathrm{narray}\ n$$

Le terme $\mathrm{nreplicate}$ a donc le type $\mathbb{N} \to \mathrm{narray}\ n$ *pour tout* entier n fourni en premier paramètre. Le terme $\mathrm{nreplicate}\ 4\ 0$ représente le tableau $\langle 0, 0, 0, 0\rangle$ de type $\mathrm{narray}\ 4$. Ce quantificateur prend pleinement son sens logique lors de son utilisation dans le type de propriétés à démontrer :

$$\mathrm{natpos} \ : \ \Pi n : \mathbb{N}. n \geq 0$$

14 : Lorsque B ne dépend pas de x, le fait de nommer la variable est inutile, et on peut alors employer la notation $A \to B$ comme sucre syntaxique pour $\Pi\_ : A. B$.

Univers et hiérarchie d'univers On remarque que les types ne sont plus une catégorie syntaxique à part entière, mais bien des termes comme les autres, et en

tant que tels, ils peuvent être manipulés comme des valeurs de première classe dans le langage. Il s'agit d'un trait des langages à types dépendants. Étant des termes comme les autres, il faut pouvoir écrire les types à gauche d'un jugement de typage. C'est la raison de l'existence de l'univers $\Box$, qui est une sorte, c'est-à-dire le type des types. Ainsi, on peut généraliser le type de tableaux narray et la fonction associée $\mathrm{nreplicate}$ aux tableaux polymorphes :

$$\begin{aligned} \text{array} &: \Box \to \mathbb{N} \to \Box \\ \text{replicate} &: \Pi A : \Box. \Pi n : \mathbb{N}. A \to \text{array } A\, n \end{aligned}$$

Comme les types d'un λ-calcul dépendant sont également des termes, la sorte $\Box$ peut elle-même être à gauche d'un jugement de typage. Or, le fait d'autoriser $\Box : \Box$ brise la cohérence logique de la théorie : il s'agit du paradoxe de Girard [25], l'équivalent en théorie des types du paradoxe de Russell[15] de la théorie des ensembles, selon lequel il ne peut exister aucun ensemble contenant tous les ensembles. Une solution est d'utiliser une version augmentée du Calcul des Constructions appelée CC_ω, dans laquelle chaque univers $\Box_i$ est annoté par un entier naturel i représentant son *niveau*.[16] On peut ainsi postuler en toute sécurité que chaque univers est inclus dans le suivant, d'où le nom de *hiérarchie d'univers* :

$$\frac{}{\Gamma \vdash \Box_i : \Box_{i+1}}$$

[25] : Girard (1972), « Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur »

15 : Russell explique initialement le paradoxe dans une lettre envoyée à Frege en 1902. Le contenu de la lettre sera ensuite publié par ce dernier [26].

16 : Dans le reste de cette thèse, pour plus de lisibilité, lorsqu'un niveau d'univers n'est pas important dans le discours, nous le laissons implicite et notons $\Box$.

Règles de typage Les règles de typage de CC_ω sont disponibles en Figure 2.3. La règle Lam diffère de sa version simplement typée par le fait que le type de t peut à présent dépendre de la variable x, donc l'abstraction est typée par un produit dépendant. Pour la même raison, la règle App effectue maintenant une substitution dans le type d'une application, afin d'instancier la variable liée x dans le codomaine B.

$$\frac{}{\Gamma \vdash \Box_i : \Box_{i+1}}\ (\textsc{Sort}) \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\textsc{Var})$$

$$\frac{\Gamma \vdash A : \Box \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A. t : \Pi x : A. B}\ (\textsc{Lam})$$

$$\frac{\Gamma \vdash t : \Pi x : A. B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[x := u]}\ (\textsc{App})$$

$$\frac{\Gamma \vdash A : \Box_i \qquad \Gamma, x : A \vdash B : \Box_j}{\Gamma \vdash \Pi x : A. B : \Box_{\max(i,j)}}\ (\textsc{Pi}) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}\ (\textsc{Conv})$$

Fig. 2.3 : Typing rules for CC_ω

2.2 Un langage de programmation expressif

Plusieurs assistants de preuve, comme Coq ou Lean, sont basés sur des variantes proches du Calcul des Constructions, mais effectuent des choix différents sur les détails de leur formalisme sous-jacent. Cette section détaille les choix majeurs implémentés dans Coq concernant les univers et les types inductifs.

2.2.1 Univers et polymorphisme

Bien qu'initialement introduits pour éviter le paradoxe de Girard et maintenir la cohérence de la théorie logique, les multiples univers présents dans une théorie des types influencent son comportement et sa puissance logique, en fonction de la nouvelle définition des règles de typage Sort et Pi.

Imprédicativité, cumulativité La hiérarchie d'univers de CC_ω est dite *prédicative* en raison de la règle de typage Pi. En effet, cette règle stipule qu'un produit dépendant vit dans un univers plus grand que ceux de son domaine et de son codomaine. Cela signifie que chaque quantification dans un terme fait augmenter son niveau d'univers, en fonction de l'univers du type sur lequel on quantifie. L'assistant de preuve Coq implémente cette hiérarchie en représentant $\square_i$ par le terme `Type@{i}`,[17] mais il inclut également d'autres univers, dont un univers $\mathbb{P}$ représenté par le terme `Prop` appelé l'univers des *propositions*. Cet univers a la particularité d'être *imprédicatif*, c'est-à-dire que lorsque le codomaine d'un produit dépendant est $\mathbb{P}$, alors la règle Pi place le produit dépendant entier dans $\mathbb{P}$ quel que soit l'univers du domaine. Le comportement de cet univers dans le typage étant différent de celui des univers prédicatifs, on ne considère sa présence que dans le premier prototype développé pendant cette thèse.

17 : Dans de nombreux cas, on laisse l'univers implicite et on écrit `Type`. Cette fonctionnalité appelée *ambiguïté typique* est présentée avec les autres fonctionnalités d'inférence de Coq, en section 3.1.

La règle Sort, quant à elle, définit la politique d'inclusion entre les univers. Coq définit alors une version supplémentaire de la règle pour définir le comportement de $\mathbb{P}$, mais également une autre règle dite de *cumulativité*, permettant d'inclure un univers de la hiérarchie dans n'importe quel univers situé au-dessus :

$$\frac{i < j}{\Gamma \vdash \square_i : \square_j}\ (\textsc{Cumul})$$

Cette généralisation de la règle Sort présentée en Figure 2.3 rend le calcul plus flexible, mais elle rend également le raisonnement sur les niveaux d'univers plus difficile du fait des nombreux cas supplémentaires qu'elle autorise.

Polymorphisme d'univers Les règles de typage mentionnant des niveaux d'univers posent de fait des contraintes sur ces univers. Historiquement, dans Coq, chaque occurrence d'un univers est associée à une variable globale qui représente son niveau. Ces contraintes d'ordre entre les univers qui apparaissent lors de la vérification du typage des termes sont alors stockées dans un graphe de contraintes global qui doit toujours être valide, c'est-à-dire acyclique. En effet, on ne cherche pas à déterminer un entier exact à affecter à chaque niveau d'univers, mais la validité du graphe de contraintes d'univers garantit l'existence d'une solution, ce qui est suffisant pour maintenir la cohérence logique du développement en cours. Par exemple, voici la définition d'un type et d'un constructeur pour ce type :

$$\begin{aligned} \mathrm{Box} &: \square_\alpha \to \square_\beta \\ \mathrm{box} &: \Pi A : \square_\gamma.\ A \to \mathrm{Box}\ A \end{aligned}$$

Les niveaux α, β et γ sont alors contraints par toutes les utilisations de ces deux constantes.[18] Dans certains cas, cette façon de faire peut causer des erreurs de typage.

18 : La définition même de box contient une occurrence de Box, posant la contrainte $\gamma < \alpha$ par les règles App et Cumul.

Par exemple, l'identité peut conceptuellement être appliquée à n'importe quel terme :

$$\mathrm{id} : \Pi A : \square_\delta. A \to A$$
$$\mathrm{id}\,\mathbb{N}\,0 : \mathbb{N}$$
$$\mathrm{id}\,(\mathbb{N} \to \mathbb{N})\,(\lambda n : \mathbb{N}. n) : \mathbb{N} \to \mathbb{N}$$

En revanche, dans cette théorie des types, l'appliquer à elle-même est impossible. En effet, si on applique l'identité à elle-même, le premier paramètre de l'application est le paramètre de type qui doit vivre dans $\square_\delta$. Or, ce paramètre est le type de l'identité $\Pi A : \square_\delta. A \to A$. Son domaine est $\square_\delta$ de type $\square_{\delta+1}$ et son co-domaine est $A \to A$ de type $\square_\delta$. Par la règle Pi, le type de l'identité vit donc dans l'univers $\square_{\delta+1}$ qui est supérieur à $\square_\delta$. Par conséquent, il est impossible d'appliquer l'identité à elle-même.

Dans les termes qui n'exploitent pas un niveau d'univers en particulier mais la relation entre les différents univers au sein d'un même terme, une manière d'obtenir le comportement souhaité est de considérer les univers comme des variables liées. Ce faisant, on définit alors non pas une seule constante mais une famille de constantes indexée par une liste d'univers liés que l'on nomme alors *instance d'univers*. L'identité devient le terme suivant, indexé par une variable d'univers :

$$\mathrm{id}_i : \Pi A : \square_i. A \to A$$

La version bien typée de l'application de l'identité à elle-même est alors $\mathrm{id}_{i+1}\,\mathrm{id}_i$. On appelle cette fonctionnalité le *polymorphisme d'univers* [27], et on dit que l'identité est une constante polymorphe sur les univers, soit *univers-polymorphe*. Chaque occurrence d'une constante univers-polymorphe est alors contrainte indépendamment dans le graphe de contraintes d'univers, plutôt que globalement dans le cas monomorphe.

[27] : Sozeau *et coll.* (2014), « Universe polymorphism in Coq »

Néanmoins, l'implémentation pratique du polymorphisme d'univers pose des questions algorithmiques non triviales. La réponse choisie par Coq n'est donc pas encore arrêtée et continue d'évoluer dans les versions actuelles du logiciel.

2.2.2 Types inductifs

Les langages de programmation de haut niveau offrent au développeur la possibilité de déclarer des types de données personnalisés. Dans les langages fonctionnels statiquement typés, il s'agit traditionnellement des types algébriques, voire des types algébriques généralisés. L'assistant de preuve Coq possède une version des types algébriques étendue aux types dépendants et cohérente avec la théorie logique, appelée *types inductifs* [28].

[28] : Paulin-Mohring (1996), « Définitions Inductives en Théorie des Types »

Définition et filtrage par motif On considère par exemple la définition en Coq du type inductif `nat`, correspondant au type théorique des entiers naturels $\mathbb{N}$ pris en exemple plus d'une fois précédemment :[19]

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

Cette définition unique comporte la définition du constructeur de types `nat`, des deux constructeurs `O` et `S`, ainsi que l'enregistrement de ces constantes dans des

19 : Ce type serait déclaré de la manière suivante, en Haskell et en OCaml :

```
data Nat = O | S Nat
type nat = O | S of nat
```

règles de typage et de réduction de Coq spécifiques aux types inductifs.[20] La définition précédente affirme qu'une valeur de type `nat` est construite à partir de l'un de ses deux constructeurs. Il est donc possible d'effectuer une disjonction de cas sur toute valeur de type `nat` pour savoir quel est son constructeur de tête, en faisant un *filtrage par motif* :

```
Fixpoint add (n1 n2 : nat) : nat :=
  match n2 with
  | O ⇒ n1
  | S n ⇒ S (add n2 n)
  end.
```

Selon la correspondance de Curry-Howard, le filtrage représente également la disjonction de cas de la théorie de la démonstration. Par conséquent, la règle de typage du filtrage doit en vérifier l'*exhaustivité*, afin de rendre impossible l'oubli d'un cas.[21] Coq interdit donc la définition de fonctions partielles, tandis qu'un langage fonctionnel plus éloigné d'une interprétation logique donnerait simplement un avertissement du compilateur dans de tels cas. Le mot-clé `Fixpoint` indique que la définition est récursive. Le λ-terme sous-jacent utilise alors un opérateur de récursivité `fix` dont la règle de typage vérifie que les appels récursifs sont *structurellement décroissants*, c'est-à-dire que l'on rappelle la fonction uniquement sur des sous-termes de l'argument de l'appel courant. Ceci permet d'autoriser la récursivité tout en garantissant que tous les programmes terminent, puisque la non-terminaison correspond à une incohérence logique.

Ces fonctionnalités sont cruciales dans la justification de la confiance supplémentaire que l'on peut placer dans une preuve Coq par rapport à une preuve sur papier. Toutefois, les termes considérés dans les parties suivantes de cette thèse ne contiennent pas de filtrage, c'est pourquoi la section précédente traite uniquement du cœur de la théorie, CC_ω, et le reste de cette thèse ne mentionne que ponctuellement le filtrage.

Structures de données Les types inductifs permettent de définir de nombreuses structures de données, y compris des encodages différents de mêmes concepts mathématiques. Par exemple, voici un exemple d'encodage binaire des entiers naturels :

```
Inductive bin_nat : Type :=
  | bO : bin_nat
  | bpos : positive → bin_nat.

Inductive positive : Type :=
  | pH : positive
  | pI : positive → positive
  | pO : positive → positive.
```

Le nombre binaire est soit $\mathbf{0}$ représenté par `bO`, soit un nombre strictement positif encodé en partant du bit de poids faible, `pO` représentant un $\mathbf{0}$, `pI` un $\mathbf{1}$, et `pH` le premier $\mathbf{1}$ en tête de nombre, par lequel tout nombre commence puisque le cas `bO` a été évacué. Voici quelques nombres binaires et leur représentation sous ce format :

$\mathbf{1}$ $(0b1)$ $\mapsto$ `bpos pH`
$\mathbf{2}$ $(0b10)$ $\mapsto$ `bpos (pO pH)`
$\mathbf{6}$ $(0b110)$ $\mapsto$ `bpos (pO (pI pH))`

20 : La définition d'un type inductif inclut également la définition de principes d'induction sur ce type, présentés en fin de chapitre.

21 : On peut même déclarer un type sans constructeur. Comme il n'y a aucun moyen d'obtenir une valeur de ce type d'une manière constructive, ce type est alors un encodage de $\bot$.

Il est possible d'utiliser les deux types `nat` et `bin_nat` pour représenter les entiers naturels. Cette diversité offerte par l'assistant de preuve est le cœur des questions traitées pendant cette thèse.

On peut également définir des structures de données exploitant pleinement le polymorphisme et les types dépendants. Par exemple, voici une définition des listes chaînées et une fonction calculant la longueur d'une liste en Coq :

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.

Fixpoint length (A : Type) (l : list A) : nat :=
  match l with
  | nil _ ⇒ O
  | cons _ _ l' ⇒ S (length l')
  end.
```

Le type `list` prend un argument `A` dans sa définition, il s'agit donc d'un type polymorphe : on peut construire des valeurs de type `list nat`, `list (list nat)`, `list (nat → nat)`, *etc.* Le fait que l'argument soit situé avant l'annonce de la sorte dans laquelle vit le type inductif dans la tête de la définition (le caractère `:`) en fait un *paramètre*, c'est-à-dire que tous les constructeurs listés en-dessous construisent invariablement des valeurs du type `list A`. Les types des constructeurs contiennent donc une quantification sur le paramètre, laissée implicite dans la définition du type inductif mais visible lors de son utilisation : les branches du filtrage dans la définition de `length` incluent ce premier argument, sous forme ignorée `_` car il ne sert pas dans la construction de la valeur de sortie de la fonction. En effet, les vrais types des constructeurs de `list` sont les suivants :

```
nil : forall (A : Type), list A
cons : forall (A : Type), A → list A → list A
```

Enfin, on peut définir des types inductifs dépendants, comme le type des vecteurs de taille fixe :

```
Inductive vec (A : Type) : nat → Type :=
  | vnil : vec A O
  | vcons : forall (n : nat), A → vec A n → vec A (S n).
```

Ici, le second argument à donner à `vec` afin d'obtenir un type est une valeur de type `nat` correspondant à la longueur du vecteur. Comme cette valeur est située dans le type de `vec`, après le caractère `:`, elle peut varier d'un constructeur à l'autre. Il s'agit par conséquent d'un *indice* et non d'un paramètre. Le filtrage sur une valeur `v` de type `vec A n` est alors un filtrage dépendant, puisqu'il effectue à la fois une disjonction de cas sur le constructeur de tête de `v` et sur le constructeur de tête de la valeur `n` située dans le *type* de `v`. De plus, le filtrage dépendant permet d'éliminer des cas impossibles. Pour illustrer ce point, voici la définition d'une fonction dépendante qui récupère une valeur en tête d'un vecteur :

```
Definition head (A : Type) (n : nat) (v : vec A (S n)) : A :=
  match v in vec _ m
  return match m with O ⇒ nat | S _ ⇒ A end
  with
  | vnil _ ⇒ O
  | vcons _ _ a _ ⇒ a
  end.
```

Cette fonction n'ayant pas de sens dans le cas d'un vecteur vide, elle n'est définie que sur les vecteurs de type `vec A (S n)` pour un `n` donné. Le filtrage de Coq permet de préciser dans quel type vit la valeur inspectée par une clause `in` et dans quel type vit la valeur renvoyée en sortie du filtrage, par une clause `return`. Dans le cas d'un filtrage dépendant, on peut faire dépendre le type de la valeur de retour du type de la valeur inspectée. Dans le cas de `head`, on lie la taille du vecteur à une variable `m` et on annonce que le filtrage renverra une valeur d'un type qui dépend de la valeur de `m`. Si `m` vaut `O`, on renverra un entier naturel, sinon une valeur de type `A`. Les cas du filtrage reflètent cette disjonction de cas dans le type de retour, car la branche du vecteur vide renvoie l'entier `O` et l'autre branche renvoie bien la valeur de tête `a` du vecteur, qui est bien de type `A`. Comme la valeur `v` n'est jamais un vecteur vide, le filtrage ne prendra en réalité jamais la première branche, mais elle doit être définie pour des raisons d'exhaustivité. Comme le cas du vecteur vide est dépourvu de sens mais doit quand même être défini, on utilise pour ce cas dans la clause `return` un type de retour arbitraire mais trivial à habiter, afin que la branche associée soit facile à remplir. Dans ce cas, nous choisissons `nat` et donnons l'habitant `O`. Grâce à la puissance expressive des types dépendants, cette fonction est donc bien typée et le système de typage garantit qu'elle ne pourra jamais être appliquée à des vecteurs vides.

Encodage de constructions supplémentaires Les types inductifs permettent également d'implémenter dans Coq d'autres constructions fréquemment utilisées en théorie des types. Voici des exemples de définitions pour le type produit $\times$ et le type somme $+$:

```
Inductive product (A B : Type) : Type :=
  | pair : A → B → product A B.

Inductive sum (A B : Type) : Type :=
  | inl : A → sum A B
  | inr : B → sum A B.
```

Une paire de type $A \times B$ est bien construite à l'aide d'une valeur de type A et une valeur de type B, et une somme de type $A + B$ possède deux cas, soit une valeur de type A, soit une valeur de type B.

Les types inductifs dépendants permettent également d'encoder des constructions plus avancées. Une *paire dépendante* $\Sigma x : A.B$ est une paire $(x ; b)$ dans laquelle x est une valeur de type A et b une valeur d'un type B *autorisé à dépendre de* x. Une telle construction peut être considérée comme l'une des constructions de base du langage étudié, au même statut que le Π-type ou l'abstraction par exemple, ou bien encodée grâce à des termes du langage. Dans Coq, on peut représenter une paire dépendante par le type suivant :

```
Inductive sigma (A : Type) (B : A → Type) : Type :=
  | dpair : forall (a : A), B a → sigma A B.
```

Une paire `dpair a b` contient bien une valeur `a` de type `A` et une valeur `b` d'un type `B a` qui dépend de `a`.

Une seconde construction avancée définissable grâce aux types inductifs dépendants est l'égalité, dans une version appelée *égalité propositionnelle*. On peut en effet encoder une égalité $x = y$ comme un terme `eq A x y`, où `A` représente le type de `x` et `y` :

```
Inductive eq (A : Type) (a : A) : A → Prop :=
  | refl : eq A a a.
```

L'égalité n'a donc qu'un seul constructeur décrivant le fait que la seule valeur possiblement égale à une valeur `a` est `a` elle-même. Grâce au filtrage dépendant, dans une preuve contenant une hypothèse `e` de type `eq A x y`, lorsque le typage l'autorise, le fait d'effectuer une disjonction de cas sur `e` expose le seul cas possible, c'est-à-dire que `y` est exactement le terme `x`, dans le sens où ces termes deviennent *convertibles*. Cette analyse de cas permet alors de remplacer toutes les occurrences de `y` par `x` et les occurrences de `e` par `refl A x`. Inversement, lorsqu'il faut prouver une propriété de type `eq A x x`, il suffit de fournir le terme `refl A x` pour terminer la preuve. Puisque la conversion inclut les règles de réduction du calcul, le constructeur `refl` peut être utilisé y compris lorsque l'égalité présente deux termes syntaxiquement différents mais convertibles. Par exemple, on peut utiliser `refl` pour prouver la propriété suivante, représentant l'égalité $1 + 1 = 2$:

```
eq nat (add (S O) (S O)) (S (S O))
```

Le vérificateur de typage se charge alors d'effectuer le calcul pour vérifier que les deux termes sont convertibles.

Enregistrements Grâce aux types inductifs, on peut également encoder les *enregistrements*, présents dans de nombreux langages de programmation et utilisés pour structurer les données. Voici la définition Coq d'un type représentant des coordonnées positives en deux dimensions, sous la forme d'un enregistrement :

```
Record Coord := { x : nat; y : nat }.
```

Un enregistrement est alors défini en donnant une valeur à chaque champ :

```
Definition coord_origin := {| x := O ; y := O |}.
```

La définition d'un type d'enregistrement est équivalente à la définition d'un type inductif à un seul constructeur prenant un argument par champ présent dans l'enregistrement, ainsi que deux fonctions de projection permettant d'extraire chaque champ à partir d'un habitant du type fraîchement créé.

```
Inductive Coord : Type := BuildCoord : nat → nat → Coord.
Definition x (c : Coord) : nat := match c with BuildCoord x _ ⇒ x end.
Definition y (c : Coord) : nat := match c with BuildCoord _ y ⇒ y end.
```

Dans des langages de programmation non dépendants, on peut encoder un enregistrement par un tuple, puisqu'il s'agit simplement d'une construction permettant de réunir plusieurs valeurs dans un même terme, en les nommant grâce aux projections. Dans un contexte dépendant comme celui de Coq, un encodage analogue des enregistrements requiert des Σ-types. En effet, les enregistrements étant en réalité des types inductifs, ils peuvent naturellement être polymorphes ou dépendants, et chacun de leurs champs est autorisé à dépendre des champs précédents. Cette fonctionnalité est exploitée dans la définition de structures mathématiques dans l'assistant de preuve. Par exemple, un monoïde est un type M muni d'une opération associative $\diamond : M \to M \to M$ et d'un élément $m_0 : M$ neutre pour $\diamond$. En Coq, on peut représenter cette structure mathématique par un type d'enregistrement qui, en vertu de la correspondance de Curry-Howard, contient à la fois des données (les définitions de m_0 et $\diamond$) et des propriétés[22] (associativité de $\diamond$ et neutralité de m_0 pour $\diamond$) :

22 : Aussi appelées *lois*.

```
Record Monoid (M : Type) := {
  mzero : M;
  mconcat : M → M → M;
  mconcat_assoc : forall (m₁ m₂ m₃ : M),
    mconcat m₁ (mconcat m₂ m₃) = mconcat (mconcat m₁ m₂) m₃;
  mzero_neut_mconcat_l : forall (m : M), mconcat mzero m = m;
  mzero_neut_mconcat_r : forall (m : M), mconcat m mzero = m
}.
```

Une instance de structure mathématique sur un type donné est donc une instance du type d'enregistrement, ses données pouvant ainsi être utilisées dans des programmes et ses propriétés dans des preuves. Cette approche diffère notamment de celle choisie par Haskell, dont la bibliothèque standard définit plusieurs structures, parmi lesquelles le monoïde, mais aussi le foncteur ou encore la monade, seulement par leurs données [29]. En effet, Haskell n'étant pas utilisé pour effectuer des preuves de manière interne au langage, le respect des différentes lois des structures mathématiques dépend de la discipline de l'utilisateur dans la définition des instances de ces structures. Il est alors possible de définir des instances illicites qui, malgré leur utilité (ces structures fournissent des abstractions utiles en programmation), rendent invalide tout raisonnement impliquant les lois de la structure effectué sur un programme qui utilise ces instances. Comme les lois sont directement dans le type d'enregistrement, toute instance de structure définie en Coq est contrainte de les respecter, car sa définition en inclut les preuves.

[29] : Wadler (1995), « Monads for functional programming »

À partir d'une définition inductive, Coq génère et prouve automatiquement un *principe d'induction*, c'est-à-dire un lemme utilisable dans les preuves pour effectuer un raisonnement par induction sur une valeur de ce type inductif. Voici le principe d'induction généré lors de la définition du type `nat` :[23]

```
Lemma nat_rect : forall (P : nat → Type),
  P O → (forall (n : nat), P n → P (S n)) → forall (n : nat), P n.
```

23 : En réalité, un principe d'induction distinct est généré pour chaque sorte existante dans le calcul, afin de couvrir tous les codomaines possibles pour la propriété `P`, mais nous ne nous intéressons qu'à la version pour `Type` dans cette thèse.

On remarque que ce schéma d'induction décrit le traditionnel raisonnement par récurrence sur les entiers naturels : si une propriété est vraie pour 0 et se transmet de tout nombre à son successeur, alors elle est vraie pour tous les entiers. Le schéma d'induction pour les listes est similaire, avec une hypothèse pour la liste vide et une hypothèse pour l'ajout d'une valeur en tête de liste. Le type des listes étant polymorphe, le principe d'induction généré l'est également :

```
Lemma list_rect : forall (A : Type) (P : list A → Type),
  P (nil A) → (forall (a : A) (l : list A), P l → P (cons A a l)) →
    forall (l : list A), P l.
```

Bonne formation et schémas d'induction Outre la présence des types dépendants, la différence majeure entre les types inductifs de Coq et les types algébriques disponibles dans d'autres langages de programmation est que lors de leur définition, l'assistant de preuve s'assure qu'il est possible de raisonner par induction sur ces types. Cela signifie que l'on ne peut pas définir n'importe quel type inductif dans Coq. Par exemple, le type inductif suivant, dont les valeurs sont construites à partir d'une fonction qui produit des valeurs dans ce même type, est invalide en Coq :

```
Fail Inductive I : Type :=
  | K : (I → I) → I.
```

En effet, toutes les occurrences de `I` dans le type de `K` ne sont pas covariantes, et la définition de ce type inductif dans Coq permettrait de prouver le principe d'induction suivant :

```
I_rect :
  forall (P : I → Type),
    (forall (f : I → I) (i : I), P (f i) → P (K f)) →
      forall i : I, P i
```

Or, en instanciant `P` avec `fun _ ⇒ False`, il deviendrait possible de prouver $\bot$ et ainsi casser la cohérence logique :

```
I_rect (fun _ ⇒ False) (fun _ _ F ⇒ F) (K (fun i ⇒ i)) : False
```

Par conséquent, Coq possède un *vérificateur de positivité*, c'est-à-dire un critère de bonne formation des types inductifs, qui assure que la cohérence logique est maintenue si tous les types inductifs le respectent.[24]

24 : Ce critère étant correct mais pas complet, il existe des types inductifs théoriquement valides qui ne sont pas acceptés par l'assistant de preuve.

3 Assistance à la preuve

Cette section traite des différentes fonctionnalités faisant de Coq un véritable *assistant* à la preuve. Il s'agit d'un aperçu partiel couvrant les éléments qui entrent en compte dans la suite de cette thèse. Premièrement, Coq présente des fonctionnalités d'inférence permettant à l'utilisateur de n'écrire que partiellement les termes et déléguer le reste du travail à l'assistant de preuve (§ 3.1). Deuxièmement, le logiciel propose à l'utilisateur un mode preuve muni d'un langage dédié, afin qu'il soit possible de faire progresser une preuve pas à pas et en visualiser l'état courant à tout moment, sans avoir à écrire les termes de preuve manuellement (§ 3.2). Troisièmement, l'assistant de preuve fournit un moyen d'exploiter facilement des relations d'égalité et plus généralement d'équivalence entre des termes, afin qu'ils soient interchangeables d'une manière relativement transparente (§ 3.3).

3.1 Inférence 24
3.1.1 L'unification 24
3.1.2 Inférence et polymorphisme *ad hoc* 27
3.2 Tactiques et automatisation . 29
3.2.1 Le mode preuve 29
3.2.2 Tactiques de preuve automatique 32
3.3 Réécriture et transfert de preuve 34
3.3.1 Réécriture 34
3.3.2 Prolongement à l'équivalence . 35

3.1 Inférence

La plupart des langages statiquement typés viennent avec des fonctionnalités d'inférence : les langages à la Hindley-Milner [30, 31] en tête de file, leur base théorique étant conçue dans l'objectif d'une inférence de type *complète*, mais également les langages plus communs. Par exemple, dans les versions modernes de C++, l'utilisation du mot-clé `auto` permet de laisser le compilateur tenter de trouver le type d'une variable ou le type de retour d'une fonction. Dans le cas de Coq, le langage de programmation est extrêmement complexe, donc les systèmes d'inférence doivent être plus avancés. Dans cette section, nous présentons diverses fonctionnalités permettant à l'utilisateur de laisser *implicitement* des *trous* dans les termes lorsque certaines informations peuvent être inférées à partir des éléments qu'il fournit.

[30] : Hindley (1969), « The principal type-scheme of an object in combinatory logic »

[31] : Milner (1978), « A theory of type polymorphism in programming »

3.1.1 L'unification

La complexité du langage de Coq apporte une certaine verbosité dans les termes, qui peut parfois s'apparenter à de l'obfuscation. Par exemple, voici un terme Coq correspondant à la concaténation de deux listes d'entiers $[1] \diamond [2, 3]$, dans sa version brute présentée au chapitre précédent :

```
mconcat (list nat) (list_Monoid nat)
  (cons nat (S O) (nil nat))
  (cons nat (S (S O)) (cons nat (S (S (S O))) (nil nat)))
```

On suppose pour cela l'existence d'une instance de monoïde pour les listes dans le contexte :

```
list_Monoid : forall (A : Type), Monoid (list A)
```

Variables d'unification La répétition du type `nat` $n+1$ fois dans la construction d'une liste de n valeurs semble superflue pour l'utilisateur, mais elle est nécessaire pour que le terme obtenu soit bien typé. L'information est effectivement redondante, puisqu'il suffit par exemple de donner systématiquement comme premier argument de `cons` le type de la valeur ajoutée en tête de liste, c'est-à-dire de l'argument suivant. L'instanciation de `list_Monoid` à `nat` peut également être inférée à partir du contexte. L'implémentation réelle de Coq possède un constructeur de terme particulier appelé *variable d'unification* conçu spécialement pour pouvoir laisser des trous dans les termes. L'exemple précédent peut alors être noté ainsi :

```
mconcat _ (list_Monoid _)
  (cons _ (S O) (nil _))
  (cons _ (S (S O)) (cons _ (S (S (S O))) (nil _)))
```

Pour chaque trou laissé dans le terme, Coq crée une variable d'unification. Le vérificateur de typage doit alors résoudre tous les problèmes d'unification et remplir les trous avant d'accepter le terme. Par exemple, le problème d'unification pour la première liste est le suivant :

```
cons ?T (S O) (nil ?T) : list ?T
```

Grâce au type de `S O`, on peut inférer que `?T` est le type `nat` et savoir que le reste de la liste doit avoir le type `list nat`, ce qui déclenche le problème d'unification suivant `nil ?T : list nat`. À la fin du processus d'inférence, toutes les annotations ont été ajoutées dans le terme sans l'aide de l'utilisateur.

Arguments implicites Les fonctionnalités d'inférence de Coq permettent d'aller au delà de la création explicite de variables d'unification en les laissant *implicites*. En effet, comme le premier argument des constructeurs du type `list`, le paramètre de type, est inférable à partir du contexte, on peut le rendre implicite en utilisant une syntaxe particulière lors de la déclaration, ou bien en exécutant des commandes Coq dédiées. Dans le cas d'un type inductif, il n'est pas forcément souhaitable de rendre le paramètre implicite partout, car cela permettrait de déclarer des valeurs de type `list` sans mentionner ce paramètre, ce qui peut apporter de la confusion. Dans ce cas, on utilise alors les commandes suivantes pour les constructeurs :

```
Arguments nil {_}.
Arguments cons {_}.
```

Ainsi, le constructeur de type `list` a toujours besoin d'un paramètre explicite, mais une valeur de type `list nat` peut être construite en écrivant simplement `nil`. On peut également déclarer implicite le paramètre de `list_Monoid` et le paramètre de `mconcat`. L'exemple ci-dessus devient alors le suivant :

```
mconcat list_Monoid
  (cons (S O) nil)
  (cons (S (S O)) (cons (S (S (S O))) nil))
```

Dans la définition d'une fonction polymorphe, on peut utiliser une syntaxe qui indique que le paramètre polymorphe est implicite. Ainsi, la tête de la déclaration de la fonction `length` définie dans le chapitre précédent devient la suivante, où les accolades déclarent un argument comme implicite :

```
Fixpoint length {A : Type} (l : list A) : nat.
```

On écrit alors `length l` comme dans un langage non dépendant.

Dans le cadre des types dépendants, il est même possible qu'une valeur soit inférable sans pour autant être un type. Par exemple, la fonction `head` récupérant la tête d'un vecteur prend en argument un entier qui apparaît dans le type de l'argument suivant, le vecteur dont on veut extraire la tête. Cet entier peut donc être déclaré implicite et inféré pour chaque occurrence de `head`. Ainsi, on peut écrire `head (vcons 4 vnil)` sans préciser aucune taille de vecteur, dans les constructeurs du type `vec` comme dans la fonction `head`.

Notations Afin que les termes soient encore plus lisibles, Coq propose un système de *notations* ajoutant des cas à l'analyseur syntaxique pour associer une syntaxe particulière à certains termes. Ces notations peuvent être infixes, avec des priorités entre elles définies par l'utilisateur. La bibliothèque standard définit notamment des notations sur l'arithmétique et les listes, afin d'obtenir des termes dans une syntaxe très proche des autres langages de programmation. Des notations `+` et `*` sont associées aux opérations d'addition et de multiplication sur les entiers naturels, et les constantes numériques peuvent être écrites naturellement en base 10. Une notation récursive est associée aux listes pour qu'elles puissent être exprimées dans la syntaxe d'OCaml. Ainsi, l'exemple de cette sous-section devient le suivant :

```
mconcat list_Monoid [1] [2; 3]
```

Dans le reste de ce document, nous utiliserons des notations classiques de Coq, telles que `=` pour l'égalité, `*` pour le produit, *etc.*

Coercions Classiquement, la programmation fonctionnelle statiquement typée interdit par défaut les plongements implicites entre les types. Un booléen ne peut alors pas être utilisé dans une position où le typage attend un entier. Cependant, l'assistant de preuve donne à l'utilisateur la possibilité de déclarer une fonction comme *coercion*, c'est-à-dire fonction de plongement implicite. Dans ce cas, une fonction `nat_of_bool` de type `bool → nat` qui associerait `true` à `1` et `false` à `0` peut être déclarée comme coercion grâce à la commande suivante :

```
Coercion nat_of_bool : bool >-> nat.
```

Ainsi, tout problème de typage `b : nat` où `b` est une valeur dans `bool` devient un problème d'unification `?f b : nat` où `?f` est une coercion. Coq tente alors de construire cette coercion, possiblement par transitivité. Ce mécanisme puissant apporte une flexibilité supplémentaire dans la syntaxe mais augmente le risque de faire valider un terme par le vérificateur de typage dans un cas où l'utilisateur s'attendrait à une erreur de typage, le vérificateur insérant une coercion dénuée de sens. Par exemple, on peut définir une coercion qui encode un entier naturel par une paire d'entiers définissant son quotient et son reste dans une division euclidienne par une constante donnée. Dans un tel cas, si l'utilisateur entre *par erreur* un booléen dans un contexte où une paire d'entiers est attendue, Coq peut composer les coercions et valider le terme, pouvant causer ultérieurement des erreurs difficiles à tracer, alors qu'une erreur de typage indiquerait directement le problème.

Ambiguïté typique La majorité des exemples de termes Coq dans cette thèse ne précise pas les niveaux d'univers. En effet, Coq possède une fonctionnalité appelée *ambiguïté typique* permettant de les inférer automatiquement. Toutes les contraintes d'univers associées sont alors vérifiées avant de déclarer que le terme est bien typé. Dans de nombreux cas, cette inférence est suffisante et donne à l'utilisateur l'illusion que toute la hiérarchie d'univers prédicatifs n'est qu'un seul univers, ce qui est plus naturel et rend plus facile l'apprentissage de Coq, bien qu'inexact en théorie. En revanche, dans le cadre du polymorphisme d'univers, cette inférence est imparfaite et doit parfois être désactivée et les annotations faites à la main. Typiquement, lorsque que le développement inclut des raisonnements circulaires dans lesquels un terme se contient lui-même, il peut être nécessaire d'activer le polymorphisme d'univers. Pour être certain de définir les termes aux bons niveaux d'univers, on peut imposer une instance d'univers de façon stricte et diminuer la portée de l'inférence d'univers. Dans cette thèse, l'implémentation du prototype Trakt utilise l'ambiguïté typique, mais celle du prototype Trocq précise de nombreux univers manuellement.

3.1.2 Inférence et polymorphisme *ad hoc*

Les différentes fonctionnalités d'inférence présentées ici prennent leurs informations dans le contexte du problème d'unification, mais il est aussi possible de les récupérer à partir d'une base de données au niveau méta alimentée par l'utilisateur. Ceci permet notamment d'implémenter le polymorphisme *ad hoc* dans certains langages de programmation, c'est-à-dire la définition de certaines valeurs, non pas en quantifiant universellement sur un type comme c'est le cas pour le polymorphisme général présenté précédemment, mais pour un sous-ensemble fini de types. Ainsi, on peut utiliser une opération générique sur n'importe quel habitant d'un type contenu dans ce sous-ensemble, et le témoin d'appartenance est inféré par le système de typage.

Classes de types La manière la plus répandue d'implémenter le polymorphisme *ad hoc* est l'utilisation de *classes de types*. Il s'agit de structures dont on peut enregistrer les instances comme habitants canoniques de leur type. D'une façon similaire à Haskell [32] ou Scala [33], Coq possède cette fonctionnalité [34], il suffit pour cela d'utiliser le mot-clé `Class` en tête de déclaration de la structure. Si le monoïde est défini comme une classe de types, on peut alors enregistrer des instances en les déclarant avec le mot-clé `Instance`. On peut alors définir des notations introduisant des problèmes d'inférence qui seront résolus automatiquement à l'aide de la base de données d'instances de la classe de types :

[32] : Hall *et coll.* (1996), « Type classes in Haskell »

[33] : Oliveira *et coll.* (2010), « Type classes as objects and implicits »

[34] : Sozeau *et coll.* (2008), « First-class type classes »

```
Notation "x <> y" := (@mconcat _ _ x y).
```

Ici, le premier trou est le type déclaré comme monoïde, et le second trou est l'instance de type `Monoid _`.

Il est également possible de déclarer des *fonctions* au niveau des types qui fabriquent des instances canoniques de classes de types à partir d'autres instances. Par exemple, si deux types A et B sont des monoïdes, alors le type produit $A \times B$ en est un également. La tête de la définition en Coq est la suivante :

```
Instance product_Monoid {A B : Type}
  `{M_A : Monoid A} `{M_B : Monoid B} : Monoid (A * B).
```

Cette méthode d'inférence permet d'utiliser de façon fiable une véritable généricité dans la syntaxe. Par exemple, grâce à cette notation et la déclaration de `list_Monoid` comme fonction canonique de fabrication d'instances de monoïdes sur des listes, l'exemple de la sous-section précédente s'écrit de la manière suivante, soit exactement la même syntaxe que la notation mathématique :

```
[1] <> [2; 3]
```

Certaines difficultés demeurent, comme le problème analogue de l'héritage en losange en programmation orientée objet, où une instance peut être inférée par plusieurs chemins, ou bien l'inférence de termes complexes dans le cadre des types dépendants, une instance pouvant être indexée par autre chose qu'un type.

Structures canoniques Les structures mathématiques sont définies comme un type appelé *type porteur*, muni de données particulières (valeurs et/ou opérations) liées à ce type, ainsi que de lois portant sur ces données. Une façon d'encoder ces structures dans un langage de programmation est d'utiliser un type enregistrement contenant les opérations et les lois, et d'utiliser le type porteur comme paramètre de cet enregistrement pour le rendre polymorphe, comme c'est le cas dans la définition de `Monoid` dans le chapitre précédent. Pour gagner en automatisation, on peut alors transformer ce type enregistrement en classe de types.

Une autre solution est de placer le type porteur directement dans la structure, en tant que premier champ. Comme les enregistrements sont dépendants, la définition des autres champs n'en est pas affectée. Cette solution est la voie choisie par la bibliothèque MathComp [35], comme le montre la structure `eqType` représentant les types munis d'une égalité décidable. Voici une déclaration similaire à celle qui en est faite dans la bibliothèque :[1]

```
Record eqType : Type := {
  carrier : Type;
  eq_op : carrier → carrier → bool;
  eq_op_equality : forall (x y : carrier), x = y <-> eq_op x y = true
}
```

On peut dans ce cas définir la première projection comme une coercion, permettant ainsi d'écrire des preuves qui quantifient en apparence sur une instance de la structure, mais réellement sur le type porteur. Afin de retrouver les fonctionnalités des classes de types, Coq propose un moyen de déclarer des *structures canoniques*[2] [37] pour résoudre automatiquement le problème d'unification suivant, avec `T` un type concret et `?E` une variable d'unification que Coq doit remplir avec la bonne structure :

$$\texttt{T} \equiv \texttt{carrier ?E}$$

On peut alors définir des notations génériques, comme `=` pour `eq_op`, utilisable pour n'importe quelle instance d'`eqType`. Par exemple, dans le lemme suivant, les valeurs T_1 et T_2 apparaissant dans le type de `u` et `v` cachent des coercions implicites vers leur type porteur, et on utilise la notation générique pour `eq_op` sur ces valeurs :[3]

```
Lemma pair_eq1 : forall (T₁ T₂ : eqType) (u v : T₁ * T₂),
  u = v → u.1 = v.1.
```

Ici, `u.1 = v.1` est donc en réalité `eq_op T₁ u.1 v.1`, l'inférence de structures canoniques ayant rempli le terme automatiquement.

[35] : Mahboubi *et coll.* (2021), *Mathematical Components*

1 : La déclaration réelle expose une complexité qu'il est inutile de présenter ici. Notamment, elle fait appel à une autre bibliothèque, Hierarchy Builder [36], qui automatise la création de structures imbriquées, non abordée dans cette thèse.

2 : Il s'agit aussi du nom de la fonctionnalité.

[37] : Mahboubi *et coll.* (2013), « Canonical structures for the working Coq user »

3 : Ici, `.1` est une notation pour une fonction générique définie sur les produits permettant d'en extraire la première composante.

3.2 Tactiques et automatisation

L'inférence présentée dans la section précédente rend le langage de Coq plus flexible donc plus facile à écrire pour l'utilisateur. Ce qui en revanche fait de Coq un véritable assistant de preuve est l'exploitation de l'inférence dans un *mode preuve* affichant à l'utilisateur l'état courant de la preuve jusqu'à ce qu'elle soit terminée, lui permettant d'identifier plus facilement la prochaine action à effectuer. Ces actions sont représentées par des commandes spécifiques au mode preuve appelées *tactiques*. Ainsi, la preuve progresse pas à pas sans que l'utilisateur n'ait à écrire les termes de preuve à la main, et la preuve finale vient sous la forme d'un script dans ce langage de tactiques appelé Ltac [38], plus lisible et plus proche d'une preuve sur papier. Certaines tactiques font plus qu'effectuer un pas de preuve et construisent des preuves entières d'énoncés contenus dans un certaine théorie qu'elles savent traiter, offrant ainsi une véritable automatisation des preuves. Cette section présente le mode preuve et ces tactiques avancées.

[38] : Delahaye (2000), « A tactic language for the system Coq »

3.2.1 Le mode preuve

Lors d'une définition, après avoir écrit la tête de la définition contenant le nom et le type du terme à déclarer, on peut donner ce terme sous forme brute comme nous l'avons fait plus tôt dans les exemples ou bien entrer dans le *mode preuve* avec le mot-clé `Proof`. Le type du terme que l'on déclare devient alors un type à habiter appelé *but*. La preuve s'effectue en exécutant une série de *tactiques* qui génèrent chacune un terme de preuve. Chaque terme de preuve est donné à Coq et fait progresser plus ou moins la preuve selon son type. Si son type n'est pas convertible au but, alors la tactique échoue et l'utilisateur doit changer de stratégie ou annuler la preuve. Si son type est convertible au but, alors on regarde s'il contient des variables d'unification. S'il n'y a aucune variable d'unification, la preuve est terminée. Dans le cas contraire, chaque variable d'unification est un trou qui reste à remplir pour terminer la preuve : Coq crée donc un sous-but par variable d'unification et demande à l'utilisateur de prouver tous les sous-buts. La tactique primitive reflétant exactement ce comportement est `refine`, mais cette présentation du mode preuve se concentre sur des tactiques plus proches du raisonnement logique.

Illustrons le mode preuve en analysant la preuve du lemme suivant dans Coq :

```
Theorem length_append {A : Type} : forall (l₁ l₂ : list A),
  length (append l₁ l₂) = length l₁ + length l₂.
```

Tout d'abord, voici une définition de la fonction de concaténation de listes :

```
Fixpoint append {A : Type} (l₁ l₂ : list A) : list A :=
  match l₁ with
  | nil ⇒ l₂
  | cons a l ⇒ cons a (append l l₂)
  end.
```

L'état initial de la preuve est le suivant, représenté par des hypothèses en haut et le but à prouver en bas :

```
                                A : Type
──────────────────────────────────────────────────────────────────────
forall (l₁ l₂ : list A), length (append l₁ l₂) = length l₁ + length l₂
```

Dans un premier temps, on peut utiliser la tactique `intros` [4] pour passer sous les

4 : `intros l₁ l₂.`

quantificateurs et introduire dans le contexte deux listes `l1` et `l2`.

```
A : Type
l1, l2 : list A
──────────────────────────────────────────────────
length (append l1 l2) = length l1 + length l2
```

En effet, pour prouver qu'une propriété est valide sur toute paire de listes, on peut nommer ces valeurs et prouver la propriété spécialisée à ces valeurs, c'est-à-dire en instanciant les quantificateurs.

Ensuite, comme la propriété à prouver est liée aux longueurs de ces listes, on peut faire une disjonction de cas sur `l1` avec la tactique `destruct`.[5] Coq divise alors la preuve en deux sous-cas, l'un dans lequel la liste a été remplacée par une liste vide, et l'autre dans lequel la liste a été remplacée par une liste `a :: l`.

5 : `destruct l1 as [|a l].`

```
A : Type
l2 : list A
──────────────────────────────────────────────────  (G1)
length (append nil l2) = length nil + length l2
```

```
A : Type
a : A
l : list A
──────────────────────────────────────────────────  (G2)
length (append (a :: l) l2) = length (a :: l) + length l2
```

Dans le premier sous-but G1, comme dans le cas d'une liste vide, la longueur est nulle d'après la définition de `length`, l'addition est la seconde valeur et la concaténation est la seconde liste d'après la définition d'append, le but est convertible à `length l2 = length l2`. Par conséquent, il suffit d'appliquer la tactique `reflexivity` qui termine la preuve en indiquant à Coq qu'il s'agit d'une égalité triviale. Cet appel de tactique est équivalent à l'application du constructeur de l'égalité `refl` par la tactique `exact`.[6] Cette tactique permet d'appliquer un terme de preuve exprimé dans le langage de Coq. Grâce à la tactique `apply`,[7] il est aussi possible de signaler à Coq qu'il suffit d'appliquer `refl` à un argument, et laisser l'inférence se charger de trouver cet argument. Ceci apporte plus de lisibilité dans le script de preuve final. La variante `eapply`[8] ajoute de la flexibilité en autorisant Coq à ajouter des variables d'unification dans le cas où l'inférence ne trouve pas le bon argument, mais elle n'est pas nécessaire ici.

6 : `exact (refl (length l2)).`

7 : `apply refl.`

8 : Et en général, les tactiques préfixées par un `e`.

Pour prouver le sous-but G2, qui devient alors le seul but restant, on peut effectuer un pas de réduction avec la tactique `simpl`[9] pour voir que la présence de `a` correspond à une incrémentation des longueurs de part et d'autre de l'égalité.

9 : `simpl.`

```
A : Type
a : A
l, l2 : list A
──────────────────────────────────────────────────
S (length (append l l2)) = S (length l + length l2)
```

On remarque alors que le but est impossible à prouver en l'état. En effet, à la différence de l'ajout de `a : A` dans le contexte et de `S` devant les deux côtés de l'égalité (que l'on peut supprimer à l'aide de la tactique `f_equal`), la preuve est dans le même état que l'état initial. Cette boucle est symptomatique de l'utilisation d'un raisonnement trop faible. Ici, il faut utiliser un raisonnement par induction sur la liste au lieu d'une simple disjonction de cas. Le premier cas G1 ne change pas car il est le cas de base de l'induction, mais le second G2 devient un cas inductif avec une hypothèse d'induction dans son contexte. On utilise alors la tactique

`induction`[10] à la place de `destruct`.

10 : `induction l1 as [|a l IHl].`

```
A : Type
a : A
l, l2 : list A
IHl : length (append l l2) = length l + length l2
──────────────────────────────────────
   length (append l l2) = length l + length l2
```

Grâce à l'hypothèse d'induction, on dispose de l'élément nécessaire pour conclure la preuve. Le script de preuve final est le suivant :

```
Theorem length_append {A : Type} : forall (l1 l2 : list A),
  length (append l1 l2) = length l1 + length l2.
Proof.
  intros l1 l2.
  induction l1 as [|a l IHl].
  - reflexivity.
  - simpl. f_equal. exact IHl.
Qed.
```

Les sous-buts créés par la tactique `induction` sont présentés sous la forme d'une liste à points, pour plus de lisibilité mais aussi afin d'indiquer à Coq que l'on se concentre d'abord sur l'un, puis sur l'autre. Les appels à `simpl` peuvent être laissés dans le script si l'étape de réduction contribue à une preuve plus facile à comprendre pour un humain qui la réexécuterait pas à pas, mais ces appels ne sont pas strictement nécessaires puisque le typage de Coq inclut la conversion. L'étape initiale d'introduction d'éléments dans le contexte peut aussi être faite avant la preuve, dans la tête de la définition :

```
Theorem length_append {A : Type} (l1 l2 : list A) :
  length (append l1 l2) = length l1 + length l2.
```

Enfin, la preuve se conclut par la mention `Qed`, ligne qui lorsqu'elle est exécutée envoie le terme de preuve global au noyau pour une seconde vérification avant l'enregistrement de la déclaration. Il existe une subtilité dans la fermeture du mode preuve : le mot-clé `Qed` rend la preuve *opaque*. Cela signifie qu'après la déclaration, le type de l'énoncé prouvé est bien considéré comme habité, et le témoin est le terme déclaré, mais sa *définition*, c'est-à-dire le terme de preuve, est en fait inaccessible, comme s'il avait été oublié par l'assistant de preuve. Ceci peut avoir un intérêt pour distinguer les preuves à caractère calculatoire des preuves purement logiques pour lesquelles on ne souhaite pas voir Coq introduire des termes à rallonge dans le but lors d'une réduction ambitieuse. Si l'on souhaite exposer le terme de preuve, on le rend *transparent* en refermant le mode preuve avec le mot-clé `Defined`.

Dans la majorité des cas, les déclarations d'ordre calculatoire se font en programmant le terme, car ce mode est plus naturel, et les preuves s'effectuent en mode preuve pour la même raison. Cependant, la frontière entre les deux n'est pas nette. Avec les types dépendants, une preuve peut dépendre de données et une donnée peut dépendre d'une preuve. Le meilleur mode pour effectuer la déclaration n'est donc pas toujours évident. Pour un contrôle maximal sur le terme de preuve généré, il peut être intéressant voire nécessaire[11] de fournir manuellement les termes de preuve, *a minima* partiellement en utilisant la tactique `exact` en mode preuve pour fournir des sous-termes bruts et sélectionner les parties déléguées à l'inférence de Coq.

11 : C'est notamment le cas de certaines preuves dans le deuxième prototype développé dans cette thèse, Trocq, présenté en § III.

3.2.2 Tactiques de preuve automatique

Le mode preuve apporte à l'utilisateur un niveau d'abstraction par-dessus le λ-calcul de Coq dans la confection de preuves, mais il peut aussi être utilisé pour les automatiser. En effet, les tactiques reçoivent l'état courant de la preuve et renvoient un terme justifiant la transition vers un nouvel état de preuve, qui est l'état final si le terme est suffisant pour terminer la preuve. Les tactiques présentées plus tôt sont utilisées pour effectuer des pas de preuve atomiques, mais rien n'empêche le développement de tactiques plus avancées, capables de réduire des preuves à une seule ligne dans le script de preuve.

Certaines tactiques effectuent alors de la recherche de preuve en sélectionnant des lemmes dans le contexte ou dans des bases de données dédiées. C'est le cas de la tactique `auto` dans la bibliothèque standard, mais également d'outils nettement plus complexes comme les *marteaux*, inspirés du greffon Sledgehammer [39] pour l'assistant de preuve Isabelle/HOL [8]. Les marteaux commencent par exécuter une procédure de filtrage du contexte global pour sélectionner les lemmes accessibles les plus pertinents pour la preuve à effectuer. Ils envoient ensuite ce contexte ainsi que le but à prouver à des prouveurs automatiques, et déterminent le sous-contexte minimal nécessaire pour faire la preuve. Ce contexte minimal est ensuite donné à une procédure de reconstruction qui effectue de nouveau la preuve, cette fois au sein de l'assistant de preuve. Le représentant de cette famille d'outils dans l'écosystème Coq est le projet CoqHammer [40].

[39] : Böhme *et coll.* (2010), « Sledgehammer : judgement day »

[8] : Nipkow *et coll.* (2002), *Isabelle/HOL : a proof assistant for higher-order logic*

[40] : Czajka *et coll.* (2018), « Hammer for Coq : Automation for dependent type theory »

D'autres tactiques sont des implémentations de *procédures de décision*, des algorithmes capables de prouver automatiquement des énoncés dans une théorie donnée. Par exemple, la tactique `lia` [17] de la bibliothèque standard est l'implémentation d'une procédure de décision pour l'arithmétique linéaire sur les entiers. Elle permet de prouver en un seul appel de tactique n'importe quel énoncé qui appartient à cette théorie. Un autre exemple est le projet SMTCoq [41] qui connecte Coq à des solveurs SMT. Le but à prouver est d'abord encodé dans le langage SMT-LIB [42], un format d'entrée classique pour cette famille de solveurs, puis le problème est donné à un solveur SMT instrumenté pour fournir une trace de son exécution donnant des indices sur la manière de prouver le but. Cette trace est ensuite donnée à une procédure de reconstruction dans Coq, qui fabrique le terme de preuve pour le but initial.

[17] : Besson (2006), « Fast Reflexive Arithmetic Tactics the Linear Case and Beyond »

[41] : Ekici *et coll.* (2017), « SMTCoq : A Plug-In for Integrating SMT Solvers into Coq »

[42] : Barrett *et coll.* (2010), « The SMT-LIB Standard : Version 2.0 »

Un exemple de preuve où une tactique d'automatisation peut être utilisée est la preuve d'une instance de l'enregistrement `Monoid` pour `nat` :

```
Definition nat_Monoid : Monoid nat.
```

La tactique `econstructor` permet d'appliquer le constructeur de l'enregistrement sans le nommer, en créant des variables d'unification pour tous les champs. L'inférence attend alors plus d'information de la part de l'utilisateur pour remplir ces variables. Ici, on souhaite remplir les champs un par un manuellement, donc on transforme ces variables d'unification en sous-buts avec la tactique `unshelve`.[12]

12 : `unshelve econstructor.`

Initialement, ces buts mentionnent les variables d'unification :

```
------------------ (G1)
?mzero : nat

                ?mzero : nat
------------------------------------------------ (G2)
?mconcat : nat → nat → nat

            ?mzero : nat
            ?mconcat : nat → nat → nat
------------------------------------------------------------ (G3)
forall (a b c : nat),
  ?mconcat a (?mconcat b c) = ?mconcat (?mconcat a b) c

            ?mzero : nat
            ?mconcat : nat → nat → nat
------------------------------------------------------------ (G4)
forall (m : nat), ?mconcat ?mzero m = m

            ?mzero : nat
            ?mconcat : nat → nat → nat
------------------------------------------------------------ (G5)
forall (m : nat), ?mconcat m ?mzero = m
```

En se focalisant tour à tour sur chaque sous-but dans l'ordre des champs de l'enregistrement, à chaque étape, on prouve un sous-but sans variable d'unification. On choisit de déclarer un monoïde basé sur l'addition comme accumulateur et le zéro comme valeur neutre. On donne donc ces deux valeurs avec la tactique `exact`.[13] Les trois derniers sous-buts deviennent alors les suivants :

13 : `exact 0.` / `exact add.`

```
------------------------------------------------------------ (G3)
forall (a b c : nat), a + (b + c) = (a + b) + c

------------------------------------ (G4)
forall (m : nat), 0 + m = m

------------------------------------ (G5)
forall (m : nat), m + 0 = m
```

Le sous-but G3 peut être prouvé à la main par induction sur les entiers, mais si l'on remarque qu'il tombe dans la théorie de l'arithmétique linéaire, on peut déléguer intégralement la preuve à la tactique `lia`.[14] Enfin, les deux derniers sous-buts sont des propriétés classiques sur les entiers naturels, prouvées dans des lemmes de la bibliothèque standard. Dans ces cas, un outil d'automatisation tel que `lia` n'est pas nécessaire, mais peut simplifier le script de preuve. En effet, on peut préciser sur quels buts on souhaite appliquer une tactique en préfixant l'appel par les numéros des buts ciblés, la mention `all` permettant d'appliquer la tactique à tous les buts restants. On peut donc dès le troisième sous-but clore la preuve en une ligne appelant `lia` sur tous les champs restants.[15] Le script de preuve final est donc le suivant :

14 : `lia.`

15 : `all: lia.`

```
Definition nat_Monoid : Monoid nat.
Proof.
  unshelve econstructor.
  exact 0. exact add.
  all: lia.
Defined.
```

On remarque que la preuve se conclut par `Defined` car on souhaite pouvoir extraire de ce terme la définition de `mzero` et `mconcat` à une fin calculatoire.

3.3 Réécriture et transfert de preuve

Une autre catégorie de manipulations fréquemment effectuées dans les preuves mathématiques est la *réécriture*, c'est-à-dire la substitution de valeurs, structures mathématiques, *etc.*, identifiées comme étant similaires. Dans les preuves formelles, on doit donner à ce concept de similarité une définition formelle, afin de justifier auprès du noyau de l'assistant de preuve le passage de l'état initial à l'état substitué. L'égalité est un exemple de relation pouvant être utilisée pour représenter cette notion de similarité. Par exemple, lors de la résolution d'un système d'équations, isoler une variable dans l'une des équations permet de l'exprimer, avec une égalité, en fonction de toutes les autres, et la remplacer par cette nouvelle expression dans toutes les autres équations, permettant ainsi de diminuer la taille du problème.

Dans la pratique des mathématiques, des relations d'équivalence plus générales que l'égalité peuvent être utilisées pour représenter la similarité entre les objets. Ce raisonnement à *équivalence* près donne une grande liberté dans les preuves concernant la représentation des objets mathématiques. Par exemple, en fonction du contexte et de la preuve à réaliser, il peut être intéressant de voir un entier naturel à travers le prisme d'un encodage unaire ou d'un encodage binaire, mais les résultats obtenus sur les entiers unaires peuvent naturellement être exploités dans des preuves sur les entiers binaires, et inversement, car ces deux encodages sont équivalents. Dans une preuve Coq, la situation n'est pas aussi simple, car les relations d'équivalence doivent être formalisées et le système de typage est moins flexible. En effet, dans le cadre des types dépendants, remplacer un type par un autre n'est pas anodin et peut rendre un terme mal typé. Le fait de rendre une preuve effectuée avec un encodage d'un objet mathématique disponible dans le contexte d'un autre encodage de cet objet est appelé *transfert de preuve*, et il s'agit d'une tâche non triviale bien que transparente sur une preuve papier. Cette section traite du fonctionnement de la réécriture et de son extension au transfert de preuve dans Coq.

3.3.1 Réécriture

L'égalité de Coq, représentée par le type inductif `eq`, est basée sur le principe d'*identité des indiscernables*, attribué à Leibniz,[16] selon lequel l'égalité correspond au fait que les deux éléments se comportent de la même manière dans tous les contextes. Formellement, cette propriété se traduit par le principe d'induction de l'égalité : si une propriété est vraie pour `x` et `x = y`, alors la même propriété est vraie pour `y`. Voici le type du principe d'induction `eq_rect` dans la bibliothèque standard de Coq :[17]

16 : D'où le nom d'égalité de Leibniz.

```
eq_rect : forall (A : Type) (x : A) (P : A → Type),
  P x → forall (y : A), x = y → P y
```

17 : Dans le cas précis du type `eq`, le principe d'induction utilisé par défaut dans Coq est non dépendant, c'est-à-dire que la propriété `P` ne quantifie pas aussi sur une preuve d'égalité mais uniquement sur la valeur de l'indice de type `A`.

Ce terme peut alors être utilisé pour faire de la réécriture dans un but. On suppose que l'on souhaite effectuer une réécriture entre deux valeurs `y` et `x` de type `A` à partir d'une égalité `e : x = y` entre les deux termes. Il suffit pour cela d'abstraire dans le but les occurrences de `y`, pour obtenir un prédicat de type `A → Type`, qui sera précisément l'argument `P` du principe d'induction de l'égalité. Le terme `eq_rect A x P ?p`$_x$ `y e` est alors une preuve du but, avec une variable d'unification `?p`$_x$ correspondant à un nouveau but `P x`, c'est-à-dire le même but dans lequel les occurrences sélectionnées de `y` ont été remplacées par `x`.

La tactique permettant d'effectuer des réécritures en utilisant `eq_rect` comme base du terme de preuve sous-jacent est `rewrite`. Elle prend en paramètre le sens de la réécriture (`x` vers `y` ou `y` vers `x`), la preuve d'égalité qui la justifie, ainsi que des indications éventuelles sur les occurrences que l'on souhaite réécrire et celles que l'on souhaite laisser intactes,[18] ou le terme dans lequel on veut réécrire (par défaut, le but). L'équivalent en tactique du terme de preuve de réécriture est donc `rewrite → e` ou `rewrite e`.

18 : Par défaut, la tactique essaie de réécrire la valeur partout où il est possible de le faire.

3.3.2 Prolongement à l'équivalence

Grâce à la réécriture telle que présentée ci-dessus, on peut exploiter des égalités dans des preuves Coq. Cependant, il existe d'autres situations dans lesquelles on effectuerait une réécriture dans une preuve papier, en n'exploitant pas une égalité mais une autre relation d'équivalence. En effet, Coq définit des tactiques `reflexivity`, `symmetry` et `transitivity`, permettant respectivement d'appliquer `refl`, de retourner une égalité et de couper une preuve d'égalité en deux, en passant par une égalité avec un troisième terme. Or, les propriétés de réflexivité, symétrie et transitivité sont communes à toutes les relations d'équivalence. Par conséquent, Coq est équipé d'une fonctionnalité de réécriture *généralisée* [43] qui étend ces tactiques ainsi que la tactique `rewrite` à des *setoïdes* [44], c'est-à-dire des types munis d'une relation d'équivalence. Par exemple, on considère l'équivalence propositionnelle `↔`, soit une implication dans les deux sens entre deux propositions dans `Prop`. Cette relation est une équivalence sur `Prop`, il est donc possible de déclarer `Prop` comme un setoïde avec cette relation. Grâce à la réécriture généralisée, on peut alors prouver `A ↔ C` avec `transitivity` et des preuves de `A ↔ B` et `B ↔ C`, pour une proposition `B` bien choisie.

[43] : Sozeau (2009), « A New Look at Generalized Rewriting in Type Theory »

[44] : Barthe *et coll.* (2003), « Setoids in type theory »

Bien qu'étendue aux setoïdes, la réécriture s'applique uniquement à des relations d'équivalence, donc des relations homogènes. La réécriture dans les types est donc un acte fragile, car elle peut modifier des types dans un énoncé mais pas les *valeurs* qui habitent ces types. En effet, on peut lier les encodages unaires et binaires des entiers naturels `nat` et `bin_nat`, déclarés en § 2.2.2, par une relation d'équivalence au niveau des types, mais la relation ne peut s'étendre à d'éventuelles valeurs dans ces types qui pourraient être présentes dans un but. Par exemple, on considère le but suivant :

```
forall (n : nat), 0 ⩽ n
```

Ici, on ne peut réécrire `nat` en `bin_nat` sans toucher au reste du but, car les occurrences de `n` changent de type dans le processus. Dans ce cas, on a besoin d'un moyen de lier la relation d'ordre `⩽` à un équivalent sur `bin_nat`, et les constantes `0` et `b0` entre elles, afin de *transférer* l'entièreté du but depuis `nat` vers `bin_nat`. Cette opération globale est appelée *transfert de preuve* et peut être utilisée pour porter des preuves existantes ou reformuler des buts, le long de preuves d'équivalence.

Le projet CoqEAL [45] permet de travailler avec des relations hétérogènes, qui peuvent être utilisées pour relier plusieurs représentations d'un même objet mathématique, en particulier une représentation adaptée à la preuve de propriétés sur l'objet et une représentation plus performante pour exprimer des programmes utilisant l'objet. On parle alors de *raffinement*. Ce greffon fonctionne sur des relations fonctionnelles hétérogènes, mais effectue du transfert uniquement sur des termes clos sans quantificateurs. Le reste de cette thèse explore différentes options de transfert de preuve, d'abord dans un objectif de pré-traitement des buts avant passage d'une tactique de preuve automatique (§ II), puis dans

[45] : Cohen *et coll.* (2013), « Refinements for Free ! »

l'objectif de pousser plus loin différentes approches récentes basées sur la *paramétricité* [13], offrant un niveau de généralité maximal incluant les types dépendants mais introduisant parfois des axiomes de manière sous-optimale (§ III). La dernière partie traite de l'implémentation de tels techniques de transfert de preuve dans Coq (§ IV).

[13] : Reynolds (1983), « Types, Abstraction and Parametric Polymorphism »

Méta-programmation en Coq avec Coq-Elpi

4

4.1 Un méta-langage logique pour Coq 37
4.1.1 Un héritage de la programmation logique 37
4.1.2 Encodage des termes Coq . . . 39
4.2 Une boîte à outils 41
4.2.1 Bases de données 41
4.2.2 Création de commandes et tactiques 42

Le mode preuve rend possible l'automatisation des preuves en permettant à l'utilisateur d'exécuter des tactiques pour générer des termes de preuve à sa place. Les scripts de preuve sont alors écrits dans le langage de tactiques Ltac qui permet de chaîner des appels de tactiques et structurer la preuve. Les tactiques sont écrites dans un méta-langage, le choix exact n'ayant pas d'importance pour Coq puisque les tactiques peuvent être vues comme des fonctions d'un état de la preuve à un autre. La seule contrainte sur la tactique est qu'elle doit pouvoir fournir un terme de preuve Coq en mesure d'être vérifié par le noyau par la suite.

Plusieurs méta-langages sont disponibles dans Coq, chacun ayant des atouts particuliers : Ltac, OCaml, MetaCoq, Ltac 2, Coq-Elpi, *etc.* Le langage Ltac possède des fonctionnalités d'inspection de l'état courant de la preuve et de création de termes Coq, ce qui en fait un méta-langage utilisable pour écrire des tactiques. Le méta-langage le plus primitif est OCaml, puisqu'il s'agit du langage utilisé pour l'implémentation de Coq lui-même. Les interactions avec Coq se font alors avec l'API interne du mode preuve et on manipule les termes Coq dans leur représentation interne, ce qui donne une liberté maximale mais expose toute la complexité de Coq au méta-programme. Le projet MetaCoq [46] permet de manipuler des termes Coq directement dans Coq. Il est donc utile lorsque l'on souhaite certifier le méta-programme, par exemple en prouvant sa complétude. Le langage Ltac 2 [47] a pour objectif d'être un méta-langage syntaxiquement proche de Ltac, tout en y ajoutant des fonctionnalités auxquelles on s'attend dans un langage moderne, telles qu'un système de typage ou un moyen de déclarer des structures de données. Tous les développements réalisés pendant cette thèse ont été effectués dans le méta-langage Coq-Elpi [16], présenté dans ce chapitre. Nous nous intéressons d'abord à ses fonctionnalités puis à l'outillage qui l'accompagne pour interagir avec Coq.

[46] : Sozeau *et coll.* (2020), « The MetaCoq Project »

[47] : Pédrot (2019), « Ltac2 : tactical warfare »

[16] : Tassi (2018), « Elpi : an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect) »

4.1 Un méta-langage logique pour Coq

En réalité, le greffon Coq-Elpi est une extension autour d'un langage appelé Elpi [48] pour en faire un méta-langage complet pour Coq. Cette section présente ce langage et les différentes fonctionnalités qui le rendent intéressant dans le cadre de la méta-programmation pour Coq. On étudie également la manière de représenter des termes Coq en Elpi.

[48] : Dunchev *et coll.* (2015), « ELPI : Fast, Embeddable, λProlog Interpreter »

4.1.1 Un héritage de la programmation logique

Le langage Elpi fait partie d'une famille de langages appelés langages *logiques*. Ce paradigme de programmation s'est développé dans la deuxième partie du XX$^{\text{ème}}$ siècle avec l'avènement de son représentant le plus célèbre, Prolog [4]. L'objet de base d'un tel langage est le *prédicat* et les programmes ont une interprétation logique dans un sous-ensemble de la logique du premier ordre.

[4] : Colmerauer *et coll.* (1973), « Un système de communication homme-machine en français »

Fonctionnement d'un programme logique Plutôt qu'une série d'instructions à exécuter, un programme logique est représenté par une base de connaissances et une requête. La *base de connaissances* est une liste de déclarations de faits (prédicats vrais inconditionnellement) et de règles (prédicats vrais si un ensemble de prémisses est vrai). Chaque déclaration d'un prédicat est appelée une *instance* de ce prédicat, et peut avoir des arguments qui sont soit des *atomes* (valeurs constantes de base du langage, comme les nombres ou les chaînes de caractères), soit des *variables*, c'est-à-dire des emplacements nommés qui ne contiennent pas encore de valeur concrète.[1]

1 : On les note en commençant par une majuscule.

La *requête* est une liste d'instances de prédicats que l'on souhaite rendre vraies. Un moteur d'inférence explore ensuite la base de connaissances pour vérifier s'il existe une solution. Pour chaque prédicat à rendre vrai, on parcourt les instances de ce prédicat dans la base de connaissances. Pour chaque instance, on tente d'unifier syntaxiquement la tête de l'instance (le fait ou la conclusion de la règle) avec le prédicat à rendre vrai dans la requête. Si l'unification réussit, on remplace ce prédicat dans la requête par le corps de l'instance (les conditions pour que la règle soit vraie), qui doit alors aussi réussir pour que la requête ait une solution. Si l'unification échoue, on passe à l'instance suivante. Si toutes les options sont épuisées, alors la requête ne peut pas être satisfaite.

Les langages logiques sont donc dits déclaratifs, car on décrit le problème et la caractérisation d'une solution plutôt que la procédure de calcul de cette solution. Plus précisément, Elpi est une implémentation de λProlog [49], une extension de Prolog y ajoutant des λ-termes (les prédicats se comportent comme des fonctions), des quantificateurs et un opérateur d'implication.

[49] : Miller *et coll.* (1987), *A logic programming approach to manipulating formulas and programs*

Exemple Illustrons le fonctionnement d'Elpi avec un encodage du λ-calcul simplement typé, présenté en § 2.1.2.

```
kind lterm type.
type abs (lterm → lterm) → lterm.
type app lterm → lterm → lterm.
```

Le code ci-dessus déclare un nouveau type `lterm` représentant les λ-termes, avec deux constructeurs, `abs` pour l'abstraction et `app` pour l'application. Il s'agit d'un encodage en HOAS (*Higher Order Abstract Syntax* [50]), c'est-à-dire que les variables du langage sont les variables Elpi et les fonctions (l'argument de `abs`) sont les fonctions Elpi.

[50] : Pfenning *et coll.* (1988), « Higher-Order Abstract Syntax »

On peut définir un type Elpi pour représenter les types simples de ce λ-calcul :

```
kind ltype type.
type arrow ltype → ltype → ltype.
```

De la même manière, le cas de la variable de type est représenté par une variable Elpi. À partir de ces déclarations, on peut implémenter un prédicat de typage, prenant un terme du calcul et renvoyant un type simple.

```
pred type-of i:lterm, o:ltype.
type-of (app T1 T2) B :-
  type-of T1 (arrow A B),
  type-of T2 A.
type-of (abs F) (arrow A B) :-
  pi a\ type-of a A ⇒ type-of (F a) B.
```

La première ligne déclare le type du prédicat, en précisant si chaque argument est une entrée ou une sortie. Chaque instance reflète une règle de typage du calcul, la tête de l'instance représentant la conclusion de la règle et le corps de l'instance ses prémisses. On remarque que le cas de l'abstraction utilise les opérateurs `pi` et `=>`, représentant respectivement la quantification universelle et l'implication. En effet, comme l'abstraction est représentée par une méta-fonction, afin d'inspecter le corps de cette fonction, on doit lui fournir un argument. L'opérateur `pi` introduit alors localement une variable `a` appelée *constante universelle*. Le terme `F a` est alors le corps de l'abstraction dans lequel la variable liée est `a`. On vérifie ensuite que ce terme a un type `B`, mais cela n'est possible que si l'on donne un type à la variable `a` fraîchement introduite. Le rôle de l'implication est l'extension de contexte et correspond exactement à cette situation : on ajoute l'hypothèse que `a` est de type `A` dans le contexte d'exécution du prédicat à droite de la flèche. Ainsi, dans l'exécution de `type-of (F a) B`, lorsque l'on cherchera le type de `a`, l'hypothèse que l'on a fournie sera une instance supplémentaire du prédicat `type-of` permettant de donner un type à cette variable. La règle $\textsc{Var}_{\rightarrow}$ qui fait intervenir le contexte est alors implicite puisqu'elle exploite le contexte d'Elpi.

Règles de gestion de contraintes Une fonctionnalité cruciale d'Elpi est le langage des *règles de gestion de contraintes* [51] (CHR). Elle permet de complexifier le flot de contrôle des programmes Elpi, en autorisant le *gel* de certaines requêtes sur une variable, c'est-à-dire leur mise en attente jusqu'à ce que la variable en question prenne une valeur concrète. On met en attente une requête `R` sur une liste de variables `L` en appelant le prédicat suivant :

[51] : Frühwirth (1994), « Constraint Handling Rules »

```
declare_constraint R L
```

Au delà du gel des requêtes, il est possible de déclarer des *règles* permettant de détecter la présence simultanée d'un ensemble de requêtes gelées et exécuter du code Elpi. Par exemple, on considère un prédicat binaire p_1 et un prédicat unaire p_2. On peut déclarer une règle de gestion de contraintes commune à ces deux prédicats de la manière suivante :

```
constraint p1 p2 {
  rule (p1 X Y) \ (p2 1) | Cond ⟺ Code.
}
```

La règle se déclenche dès qu'il existe simultanément dans l'ensemble des prédicats gelés une instance de p_1 appliquée à deux valeurs arbitraires et une instance de p_2 appliquée à la constante `1`. Lorsque la règle se déclenche, la condition `Cond` est testée pour savoir si le code de la règle (représenté par la variable `Code`) doit être exécuté. Si la condition réussit, le code est exécuté ; sinon, on inspecte les autres règles. Le caractère `\` permet de supprimer des requêtes en attente une fois la règle exécutée : tous les prédicats situés à gauche de ce caractère sont maintenus, tous ceux à droite sont supprimés. Il est tout à fait possible d'écrire une règle qui ne supprime aucun prédicat identifié ou une règle qui les supprime tous.

4.1.2 Encodage des termes Coq

Le greffon Coq-Elpi connecte Elpi à Coq en définissant des prédicats internes écrits en OCaml permettant de donner aux développeurs Elpi l'accès à l'API de l'assistant de preuve, afin d'interagir avec directement depuis le méta-langage. Lorsque le corps d'un prédicat est écrit en OCaml, il est nécessaire de définir un

moyen de passer des valeurs OCaml aux valeurs Elpi et inversement. Chaque structure de donnée manipulée doit alors avoir une représentation dans les deux langages, en particulier les termes Coq.

Le λ-calcul de Coq est encodé par un type Elpi `term`, dont voici les constructeurs utilisés dans cette thèse :

```
type sort sort → term.
type fun name → term → (term → term) → term.
type prod name → term → (term → term) → term.
type app list term → term.
type global gref → term.
type pglobal gref → univ-instance → term.
```

Présentons ces constructeurs dans l'ordre.

Tout d'abord, les sortes sont encodées par un constructeur `sort`. On distingue un cas `sort prop` permettant d'encoder l'univers imprédicatif $\mathbb{P}$ et un autre cas `sort (typ I)` qui encode l'univers prédicatif $\square_i$ en associant i à `I` à l'aide d'un autre type Elpi représentant les univers.

Ensuite, les encodages des lieurs Π et λ sont respectivement les constructeurs `prod` et `fun`. Il s'agit ici également d'un encodage en HOAS, où les lieurs utilisent des méta-fonctions. Il n'y a par conséquent pas de cas pour les variables liées ni pour les variables d'unification. Les variables liées sont représentées par des constantes universelles comme dans l'exemple du λ-calcul simplement typé ci-dessus, et les variables d'unification sont représentées par des variables Elpi.[2] Ces deux cas sont notoirement difficiles à traiter dans les encodages plus classiques des termes Coq, car leur manipulation est délicate. En effet, ces variables n'ont de sens que dans un contexte donné. Pour ne pas avoir à prendre les noms des variables liées en compte dans le raisonnement, on privilégie les encodages rendant syntaxiquement égaux les termes α-équivalents. Ainsi, les encodages communs du λ-calcul représentent les variables liées par des *indices de* De Bruijn [52], c'est-à-dire un constructeur $\texttt{var}_n$ où n est un entier naturel représentant la distance au lieur dont ce constructeur pointe la variable. Dans ce cadre, les lieurs sont représentés simplement par le type de la variable liée et le terme lié. Par exemple, si l'on note `lam` le constructeur qui encode λ et `A` l'encodage d'un type A, l'identité $\lambda x : A.\, x$ est encodée par $\texttt{lam A var}_1$, et l'encodage de $\lambda x : A.\, \lambda y : A.\, x$ est $\texttt{lam A (lam A var}_2\texttt{)}$. L'encodage HOAS choisi par Coq-Elpi est un autre moyen de représenter les termes modulo α-équivalence.[3] En effet, l'utilisation d'une méta-fonction fixe la position du lieur auquel la variable fait référence, et a même l'avantage sur les indices de De Bruijn qu'il est impossible qu'une variable échappe à sa portée, tandis que sans implication supplémentaire du système de typage du méta-langage, un indice de De Bruijn peut *a priori* être plus élevé que le nombre de lieurs présents dans le terme. De plus, l'abstraction de termes ou la réduction exigent une certaine rigueur dans les décalages d'indices, tandis que ces opérations sont triviales en Elpi.

Concernant les constructeurs restants, on remarque que l'application est n-aire pour refléter le type OCaml des termes Coq, et les constantes sont représentées par deux constructeurs différents en fonction de leur polymorphisme d'univers. Les noms des constantes sont encodés dans un type `gref`[4] qui possède trois cas, pour représenter les constructeurs de types inductifs (`indt`), les constructeurs de termes dans ces types inductifs (`indc`) et le reste des définitions (`const`). D'autres constructeurs de termes sont disponibles, comme `match` ou `fix`, mais ne sont pas abordés dans cette thèse.

2 : Un terme à trous dans Coq est par conséquent également un terme à trous en Elpi.

[52] : De Bruijn (1972), « Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem »

3 : Les premiers paramètres de type `name` des lieurs sont simplement présents pour des raisons d'affichage mais ne sont pas pris en compte dans l'unification d'Elpi.

4 : Pour *global reference*.

Voici un exemple de terme Coq et son encodage en Elpi :[5]

```
fun (X : Type@{u}) (f : Type@{u} -> A) => f X
```

```
fun `X` (sort (typ «u»)) x\
  fun `f` (prod `_` (sort (typ «u»)) _\ global (indt «A»)) f\
    app [f, x]
```

Un outil Coq-Elpi reçoit et envoie les termes Coq (arguments des commandes, buts à prouver, termes de preuve, *etc.*) dans ce format.

5 : On suppose ici que `A` est un type inductif univers-monomorphe et `u` un univers nommé.

4.2 Une boîte à outils

Au delà des fonctionnalités offertes par le langage Elpi et de l'encodage haut niveau des termes Coq, le greffon Coq-Elpi fournit de nombreux points d'entrée vers l'API de Coq, par exemple pour chercher la définition d'un terme dans le contexte Coq dans lequel s'exécute le méta-programme, faire des appels au vérificateur de typage ou encore créer de nouvelles définitions. On peut également créer des bases de données pour stocker des termes au niveau méta ainsi que des commandes et tactiques pour interagir avec l'assistant de preuve, faisant d'Elpi un outil complet de méta-programmation.

Dans cette section, on prendra l'exemple simple d'une fonctionnalité d'affichage de termes, c'est-à-dire l'association à chaque type d'une fonction qui crée une chaîne de caractères à partir d'une valeur de ce type. Pour cela, le langage Haskell utilise une classe de types `Show` et inclut une fonctionnalité de dérivation d'instances pour les nouveaux types de données avec le mot-clé `deriving` :

```
class Show a where
  show :: a -> String

data Maybe a = Just a | Nothing
  deriving Show
```

On peut obtenir cette fonctionnalité en Coq grâce à une base de données stockant les fonctions d'affichage propres à chaque constante et une commande permettant de générer de nouvelles fonctions à partir des fonctions existantes et de la déclaration d'un type.

4.2.1 Bases de données

En Coq-Elpi, une base de données est une banque d'instances de prédicats. On peut en effet utiliser un prédicat comme moyen de regrouper des données ou les associer entre elles. Dans notre cas, on veut associer à des constantes des fonctions Coq permettant de les afficher. On peut utiliser un prédicat `show` prenant en argument deux constantes, l'une pour le type à afficher, l'autre pour la fonction d'affichage.

Création d'une base de données La création d'une base de données Coq-Elpi se fait avec la commande `Elpi Db`, à laquelle on donne le nom de la base de données ainsi qu'un bloc de code déclarant les différents prédicats représentant les données que l'on va y stocker par la suite. Par exemple, voici la déclaration d'une base de données destinée à contenir des instances de fonctions d'affichage :

```
Elpi Db show.db lp:{{
  pred show o:gref, o:constant.
}}.
```

Ici, `show.db` est le nom de la base de données. On remarque que le premier argument peut être n'importe quelle constante (type inductif, constructeur, définition) tandis que le second est forcément une définition, puisqu'il s'agit d'une fonction dont le codomaine est `string`.

Ajout d'informations L'ajout d'une instance du prédicat `show` dans la base de données se fait avec un prédicat interne à Coq-Elpi appelé `coq.elpi.accumulate`. Il prend divers arguments permettant de préciser la base de données dans laquelle stocker l'instance, à quel endroit dans cette base plus précisément, *etc.*, ainsi que l'instance en question. Ainsi, si l'utilisateur déclare un terme `show_nat` comme fonction d'affichage pour le type `nat`, il est possible de l'ajouter à la base de données `show.db` avec la requête suivante dans Coq-Elpi :

```
global Nat = {{ nat }},
global (const ShowNat) = {{ show_nat }},
coq.elpi.accumulate _ "show.db" (clause _ _ (show Nat ShowNat))
```

La syntaxe `{{·}}` est l'opération de *réification* permettant d'utiliser la syntaxe Coq au sein d'un programme Coq-Elpi, le terme étant ensuite traduit en Elpi. L'opération inverse, permettant d'écrire un terme exprimé dans le méta-langage au milieu d'un terme Coq et de l'*évaluer* pour obtenir un terme entièrement exprimé en Coq, est accessible par la syntaxe `lp :{{·}}`. Naturellement, dans un greffon abouti pour Coq, l'utilisateur n'est pas contraint d'écrire ce code Elpi manuellement pour peupler une base de données. Il utilisera plutôt une commande Coq mise à disposition par le développeur du greffon. Ceci présente aussi l'avantage d'ajouter une indirection entre l'entrée utilisateur et le code de modification concrète de la base de données, par exemple pour vérifier que les termes fournis par l'utilisateur en argument de la commande sont bien typés et correspondent bien au format des données que l'on souhaite stocker.

Requêtes dans la base de données Une fois que l'utilisateur a rempli la base de données avec les informations souhaitées, n'importe quel code Coq-Elpi connecté à la base peut l'exploiter en faisant des requêtes. Comme les données sont représentées par des instances de prédicats, effectuer une requête revient à appeler un prédicat de la base de données. Si l'instance recherchée existe, alors le prédicat réussit, sinon il échoue.

4.2.2 Création de commandes et tactiques

L'intérêt principal d'un méta-langage pour Coq est de permettre le développement d'outils pour interagir plus efficacement avec Coq, c'est-à-dire des commandes et des tactiques. Les commandes permettent d'enregistrer des données, de chercher et afficher des informations, et d'automatiser des déclarations. Les

tactiques permettent d'effectuer une preuve plus rapidement qu'en laissant l'utilisateur la faire seul, en exécutant des algorithmes implémentés dans le méta-langage pour automatiser les étapes de la preuve. Dans le cas de l'exemple donné plus tôt, on souhaite avoir une commande pour enregistrer des fonctions d'affichage et une commande permettant d'en générer de nouvelles.

Commandes Une commande permettant à l'utilisateur de déclarer des fonctions d'affichage peut être la suivante :

```
Show Declare nat show_nat.
```

Le rôle principal de cette commande est de remplacer l'appel manuel au prédicat interne `coq.elpi.accumulate`. On la déclare comme ceci :

```
Elpi Command Show.
Elpi Accumulate Db show.db.
Elpi Accumulate lp:{{
  main [str "Show", str "Declare", trm T, trm F] :- % ...
}}.
```

Après avoir connecté la commande à la base de données, on donne le prédicat principal de la commande, avec les arguments qu'elle attend et le code à exécuter lorsqu'elle est appelée. Ici, la variable `T` contiendra le terme correspondant au type pour lequel on souhaite ajouter une fonction d'affichage. Cette fonction est alors représentée par la variable `F`. Ce prédicat principal fait appel au prédicat interne `coq.elpi.accumulate`, mais il peut vérifier préalablement que `T` et `F` sont bien typés par exemple, voire que `F` a bien le type d'une fonction d'affichage pour `T`. De cette manière, on est certain que la base de données contient des données bien formées à tout instant.

Une commande peut être déclarée d'une façon similaire pour générer une fonction d'affichage sur un type inductif `I` à partir de sa définition. On appelle alors un prédicat interne `coq.env.indt` pour récupérer la déclaration, donnant entre autres le type du constructeur de type `I` et des constructeurs de valeurs dans `I`. Ensuite, la fonction peut être générée en inspectant ces types syntaxiquement. Une fois la fonction d'affichage construite, on peut appeler le prédicat interne `coq.env.add-const` pour déclarer ce terme comme une constante `show_I` par exemple.

Tactiques La création d'une tactique avec Coq-Elpi se fait grâce à la commande suivante, où `t` est le nom de la tactique :

```
Elpi Tactic t.
```

Tout comme les commandes, une tactique a un prédicat principal,[6] que l'on définit en accumulant du code Elpi, après avoir connecté la tactique aux différentes bases de données auxquelles son code fait appel :

6 : Dans le cas des tactiques, il s'appelle `solve`.

```
Elpi Accumulate lp:{{
  solve InitialGoal NewGoals :- % ...
}}.
```

Le prédicat `solve` prend le but initial et doit renvoyer une liste de buts contenant le but associé qui reste à prouver ainsi que les éventuelles obligations de preuve que la tactique laisse derrière elle.

Le but initial est représenté par une valeur du type Elpi `goal`, contenant à la fois le contexte du but, le type à habiter pour le prouver, ainsi que les différents arguments passés à la tactique. Ce type `goal` contient également des variables de type `term` représentant les preuves à appliquer pour prouver le but. Ce sont initialement des variables indéfinies et toute contrainte d'unification qui leur est appliquée se traduit par une action sur le terme de preuve dans Coq. Toucher directement ces termes est donc un acte délicat, mais une API est disponible dans Coq-Elpi pour agir sur le terme de preuve d'une façon contrôlée. Dans le cadre de cette thèse, nous n'utilisons que la fonction `refine` qui effectue essentiellement la même action que la tactique Coq du même nom, c'est-à-dire appliquer un terme de preuve à trous, les trous représentant les nouvelles obligations de preuve. Coq vérifie que le terme à trous est bien typé pourvu que l'utilisateur les remplisse par la suite.

Trakt :
Transfert de preuve par canonisation

Introduction

Les procédures de décision qui permettent d'automatiser les preuves d'énoncés contenus dans une théorie sont définies de façon abstraite, à l'aide des objets mathématiques présents dans la *signature* de cette théorie, c'est-à-dire la liste des symboles appartenant à la théorie ainsi que les différentes équations qui la régissent. Or, dans l'assistant de preuve Coq, il est possible de modéliser un même objet mathématique de différentes manières, comme le montre la définition des types inductifs `nat` et `bin_nat` en § 2.2.2. Cette diversité dans les représentations des objets mathématiques est disponible à la fois pour le développeur de tactiques et l'utilisateur de l'assistant de preuve.[7] Ainsi, lors de l'implémentation d'une procédure de décision pour Coq, le développeur effectue un choix de représentation pour chacun des symboles de la théorie concernée. La tactique d'automatisation, fruit de cette implémentation, est alors initialement biaisée envers l'une des représentations : si l'utilisateur se sert des mêmes structures de données que le développeur de la tactique pour représenter la théorie, alors la procédure de décision peut s'exécuter normalement et fournir l'automatisation attendue; dans le cas contraire, la tactique ne reconnaît pas les différents symboles, ce qui la rend incapable de remplir son rôle correctement.

7 : Pour ce dernier, elle est importante, car certaines preuves se font plus facilement avec certaines représentations.

Afin d'augmenter la compatibilité des buts avec les procédures de décision, une solution est d'ajouter une phase de pré-traitement entre ces buts et les tactiques associées. Un énoncé pré-traité idéal exprimerait tous les symboles de la signature d'une théorie avec des termes définis à l'aide de la même structure de données, celle qui est reconnue par la tactique d'automatisation que l'on souhaite exécuter après le pré-traitement. Les diverses représentations que l'on peut trouver dans les énoncés doivent par conséquent toutes *converger* vers la signature cible. Les énoncés pré-traités sont alors exprimés sous une forme *canonique* qui maximise les chances de réussite de la tactique d'automatisation qui s'exécute ensuite. Afin de remplacer le but d'origine par le but pré-traité, il est nécessaire de prouver que le second implique le premier. En effet, si l'on souhaite prouver G' quand Coq attend une preuve de G, il faut appliquer la règle du *modus ponens* à l'aide d'une fonction de type $G' \to G$, qui effectue le remplacement de but.

Cette partie présente Trakt[8] [12], un greffon pragmatique de pré-traitement de buts pour Coq dont l'objectif est de permettre à plus d'énoncés d'être prouvés par des implémentations existantes de procédures de décision. Nous y présentons tout d'abord la spécification de l'outil de pré-traitement souhaité et nous situons les outils existants par rapport à cet objectif (§ 5). Ensuite, nous présentons dans le détail le pré-traitement effectué par Trakt sur le plan théorique (§ 6). Enfin, nous faisons le bilan de cet outil, en le positionnant par rapport à un écosystème d'outils de pré-traitement dans Coq, et en identifiant les perspectives d'amélioration (§ 7).

8 : L'implémentation est disponible dans le dépôt :

`https://github.com/ecranceMERCE/trakt`

[12] : Blot *et coll.* (2023), « Compositional preprocessing for automated reasoning in dependent type theory »

Canonisation d'énoncés : objectifs et situation actuelle

5

5.1 Contenu du pré-traitement souhaité 47
5.1.1 Pré-traitement des théories . . 47
5.1.2 Statut de la logique 49
5.1.3 Polymorphisme et types dépendants 50
5.2 La famille `zify` : fonctionnalités et limites 51
5.2.1 Pré-traitement modulaire de l'arithmétique 51
5.2.2 Pré-traitement de la logique . 52
5.2.3 L'extension `mczify` 53
5.2.4 Limites de `zify` 54

Le cadre du pré-traitement d'énoncés dans Coq est large, et la nature des réécritures effectuées dépend de la procédure de décision que l'on souhaite exécuter après le pré-traitement, ainsi que de la flexibilité de l'implémentation de cette procédure de décision. Nous restreignons ce cadre à la *canonisation* des énoncés, c'est-à-dire l'exploitation d'équivalences entre des types et des opérations dans une réécriture du but, afin qu'il utilise la même implémentation de la signature d'une théorie que celle choisie par le développeur de l'outil d'automatisation que l'on souhaite exécuter pour prouver le but. Cependant, même dans ce cadre réduit, le pré-traitement parfait n'existe pas, chaque outil traitant un sous-problème précis et pas nécessairement d'une manière complète. Le développement des outils de pré-traitement est alors incrémental, chaque outil étant justifié par l'identification de limites dans les outils existants qui traitent le même problème. Dans ce chapitre, nous définissons le problème de pré-traitement auquel nous nous intéressons dans la première partie de cette thèse (§ 5.1), puis nous étudions les fonctionnalités d'une famille d'outils de pré-traitement basés sur la tactique `zify`, ainsi que leurs limites au regard du problème à résoudre (§ 5.2).

5.1 Contenu du pré-traitement souhaité

Une manière efficace de définir l'objectif d'un outil de pré-traitement d'énoncés est de se baser sur la tactique d'automatisation que l'on souhaite exécuter pour prouver les énoncés contenus dans une théorie donnée, et d'identifier les différences entre les buts fréquemment rencontrés par l'utilisateur dans cette théorie et l'implémentation de la signature de la théorie dans la tactique d'automatisation. Dans le cas de Trakt, l'objectif général est d'avoir un outil de pré-traitement pour les théories de la famille SMT, et en particulier d'améliorer le pré-traitement du greffon SMTCoq [41], un outil permettant de connecter Coq à des solveurs SMT. On souhaite cependant que l'outil soit compatible avec d'autres tactiques d'automatisation, telles que `lia`, en le rendant flexible sur la théorie pré-traitée et en ne fixant pas une signature particulière en dur dans le greffon. Dans cette section, nous expliquons point par point les objectifs de la canonisation attendue dans ce contexte.

[41] : Ekici *et coll.* (2017), « SMTCoq : A Plug-In for Integrating SMT Solvers into Coq »

5.1.1 Pré-traitement des théories

Le premier objectif de la canonisation est d'effacer la variabilité dans les représentations des objets mathématiques dans Coq, due à la liberté et l'expressivité du langage de l'assistant de preuve. Cela revient à traduire un but utilisant une ou plusieurs implémentations de la signature d'une théorie vers un autre but qui utilise une implémentation de la signature définie comme canonique. Ceci implique de savoir traiter individuellement tous les éléments possibles d'une signature et leurs occurrences dans le but. On considère dans les exemples la tactique d'automatisation `lia` [17] pour l'arithmétique linéaire sur les entiers.

[17] : Besson (2006), « Fast Reflexive Arithmetic Tactics the Linear Case and Beyond »

Types de données L'implémentation canonique sélectionnée par `lia` pour la signature de l'arithmétique linéaire sur les entiers est basée sur le type `Z` des entiers relatifs binaires :

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

La représentation choisie utilise le type `positive` défini en § 2.2.2.[1] Comme il s'agit du type canonique pour les entiers, une valeur dans `Z` est reconnue comme un entier dans toute exécution de la tactique `lia`. En revanche, les entiers exprimés avec d'autres encodages ne sont pas reconnus comme tels, et peuvent faire échouer la tactique.[2] Par exemple, la bibliothèque MathComp fournit un type `int` avec un encodage unaire collant deux exemplaires de l'espace des entiers naturels l'une contre l'autre :

```
Variant int : Set := Posz of nat | Negz of nat.
```

Bien que cette multiplicité de représentations offre une plus grande liberté à l'utilisateur, les tactiques restent rigides en raison des choix nécessaires à l'implémentation. La tactique `lia` ne traite pas par défaut les valeurs dans `int` comme des entiers. La phase de pré-traitement souhaitée vient remplacer les valeurs de divers encodages par des valeurs dans un encodage cible. Dans le cas de l'arithmétique, par exemple, on souhaite remplacer les valeurs dans `int` par des valeurs dans `Z` pour améliorer la prise en charge du but par `lia`. Dans le cas des énoncés quantifiés, on veut que le type des variables liées change également en conséquence, au moins pour les quantifications sur des types simples.

1 : Il s'agit en fait de `bin_nat` avec un cas supplémentaire pour les entiers strictement négatifs.

2 : Ici, on considère une version de `lia` sans pré-traitement. Dans la version standard de Coq, cette tactique est fournie avec un pré-traitement intégré dont Trakt augmente les fonctionnalités.

Opérations et constantes Dans le pré-traitement souhaité, le remplacement des valeurs dans un type source par des valeurs dans un type cible doit s'effectuer à un niveau de granularité plus fin. On doit pouvoir associer une opération du type source à une opération équivalente dans le type cible. En effet, dans le cas contraire, une valeur `a + b` dans un type source (par exemple `int`) pourrait être traduite d'un seul bloc, ce qui n'en ferait plus une addition dans le type cible. Si les opérations sont associées, l'outil de pré-traitement peut alors traduire l'opération, puis ses arguments. Ce traitement s'applique également aux constantes dans le type source, qui peuvent être considérées comme des opérations d'arité nulle. Si l'on peut associer les constructeurs d'un type source à des constantes et opérations dans le type cible, on peut alors traduire les constantes numériques.

Sous-typage Certains buts rencontrés dans les preuves peuvent contenir des types pouvant être plongés dans un type cible plus grand, sans leur être équivalent. C'est le cas par exemple du type `nat` et du type `Z`. En effet, il peut être intéressant de voir les entiers naturels comme un cas particulier des entiers relatifs pour faciliter le travail d'un outil de preuve automatique comme `lia`. On s'attend alors à ce que l'énoncé traduit présente une propriété qui explicite l'information de sous-typage, afin que cet énoncé associé reste prouvable si l'énoncé d'origine l'est.

Pour résumer, on considère le but suivant :

```
forall (a b : nat), a + b = b + a
```

Ce but doit pouvoir être traduit vers le but suivant :[3]

3 : On utilise la même notation `+` pour l'addition dans les deux types, pour plus de lisibilité.

```
forall (a b : bin_nat), a + b = b + a
```

On constate que les valeurs sont traitées indépendamment : on conserve les additions dans le but pré-traité et les quantificateurs sont mis à jour avec des variables liées dans le type cible. Si l'outil d'automatisation requiert que les valeurs soient exprimées dans un type cible plus grand, on s'attend à un but similaire à celui-ci, ajoutant des conditions permettant de conserver l'information de positivité de `a` et `b` :

```
forall (a b : Z), a ⩾ 0 → b ⩾ 0 → a + b = b + a
```

5.1.2 Statut de la logique

Le pré-traitement que l'on souhaite construire se concentre sur la traduction des symboles d'une théorie, mais lors de la traduction d'un énoncé Coq, on ne peut ignorer le traitement de la logique, exprimée à travers des quantifications, relations, connecteurs logiques, *etc.* Même dans un but simple, on a besoin de la logique pour exprimer ce que l'on souhaite prouver.

Relations Par exemple, dans le dernier but présenté ci-dessus, la propriété à prouver est une égalité entre deux expressions arithmétiques. Le type inductif `eq` utilisé pour représenter l'égalité est une relation binaire sur un type donné. Dans le but associé, l'égalité est devenue celle du type cible. On cherche donc à construire un pré-traitement qui substitue aussi les relations. On note que ce fonctionnement doit pouvoir s'étendre à des relations d'arité arbitraire, dont les prédicats, d'arité 1.

Représentations de la logique L'expressivité de Coq permet d'exprimer la logique de différentes manières, toutes n'étant pas forcément traitées par les tactiques de preuve automatique. En toute généralité, les propriétés peuvent être exprimées grâce à des types inductifs dans `Prop` ou `Type`. Or, un utilisateur ou un développeur de bibliothèque peut utiliser dans ses énoncés des relations dans d'autres représentations qui ne sont pas reconnues par toutes les tactiques d'automatisation. En effet, les relations décidables sont en général encodées comme des booléens, sur lesquels il est facile de faire une disjonction de cas.[4] Les types de la bibliothèque MathComp [35] viennent accompagnés de définitions permettant de les utiliser avec la logique booléenne (par exemple, un test d'égalité ou une relation d'ordre). Ces encodages booléens ont moins de chances d'être reconnus par une tactique d'automatisation que leurs versions dans `Prop` définis dans la bibliothèque standard de Coq. Un pré-traitement capable d'exprimer la logique dans `Prop` est donc intéressant.

4 : Le système logique de Coq n'étant pas classique, il n'est, par défaut, pas toujours possible de faire une disjonction de cas sur une valeur dans `Prop`.

[35] : Mahboubi *et coll.* (2021), *Mathematical Components*

À l'inverse, si l'on utilise une tactique d'automatisation faisant intervenir un système logique de plus bas niveau que celui de Coq, dans lequel toute relation est décidable, il faut que l'énoncé qu'elle reçoit expose au maximum l'information de décidabilité des relations. Par exemple, le projet SMTCoq, qui cible des solveurs SMT, associe directement la logique booléenne de Coq à la logique du langage SMT-LIB utilisé comme format d'entrée des solveurs. Ainsi, l'énoncé d'entrée optimal pour la tactique d'automatisation de SMTCoq est un énoncé où toute la logique est exprimée dans `bool`.

Connecteurs logiques L'utilisateur de Coq peut déclarer de nouveaux connecteurs logiques, par exemple un OU exclusif, et les utiliser dans ses énoncés. Si ces connecteurs ont une version booléenne, le pré-traitement doit pouvoir cibler une version ou l'autre, afin que toute la partie logique de l'arbre de syntaxe représentant l'énoncé soit exprimée dans la représentation souhaitée de la logique.

Pour résumer, prenons le but suivant, exprimé à l'aide de la logique boolénne et des entiers de MathComp :[5]

```
forall (x y z : int), x ⩽? y && y ⩽? z ⟶ x ⩽? z
```

Les relations d'ordre, l'implication et la conjonction sont exprimées dans une version booléenne. Le pré-traitement souhaité est capable de traduire cet énoncé en ciblant à la fois un autre type pour représenter les entiers, comme `Z`, et une représentation de la logique dans `Prop`.

```
forall (x y z : Z), x ⩽ y ⋀ y ⩽ z → x ⩽ z
```

5 : Une coercion de `bool` vers `Prop` définie dans cette bibliothèque pour plus de lisibilité doit également être prise en compte par l'outil de pré-traitement.

5.1.3 Polymorphisme et types dépendants

L'expressivité de Coq permet à l'utilisateur de prouver des énoncés très généraux vrais pour une famille de types. Par exemple, on peut prouver des énoncés portant sur une structure de données quelle que soit sa taille ou le type des éléments que cette structure contient. Les fonctionnalités du pré-traitement définies précédemment doivent alors s'étendre à ce cadre, pour que ces énoncés plus abstraits puissent également être prouvés par des outils d'automatisation.

Polymorphisme Si l'on sait associer une valeur dans un type `A` à une valeur dans un type `B` dans l'énoncé de sortie, il peut être intéressant de propager cette association dans une structure de données contenant des valeurs dans `A`, afin d'obtenir dans l'énoncé associé une structure similaire avec ses valeurs dans `B`. Ainsi, une liste d'entiers `list int` peut être associée à une liste d'entiers `list Z` dans le but associé, ce qui permet alors à une tactique capable de traiter la théorie des listes et l'arithmétique dans `Z` de prouver le but.

Sans s'intéresser au type des valeurs contenues dans une structure de données, on peut aussi effectuer un pré-traitement d'ordre logique, par exemple en propageant une égalité décidable à travers la structure pour en obtenir une sur la structure elle-même. Les associations entre relations détaillées précédemment sont alors paramétrées par d'autres relations.

Par ailleurs, il peut être intéressant d'associer plusieurs structures de données entre elles. Ainsi, passer d'une liste d'entiers `list int` à un arbre d'entiers `tree Z` peut se faire en réorganisant les valeurs de la liste.

Enfin, des formes de polymorphisme *ad hoc* peuvent être présentes dans les énoncés, comme les notations génériques de la bibliothèque MathComp permettant d'utiliser une même notation pour plusieurs types munis d'une même opération. Par exemple, l'addition de MathComp est notée `+` pour tous les types qui sont en réalité des instances d'une structure mathématique d'anneau, encodée par des structures canoniques.[6] Cette généricité ne doit pas entraver le fonctionnement de l'outil de pré-traitement.

6 : Concept présenté en § 3.1.2.

Types dépendants Au delà du polymorphisme, certaines structures de données sont indexées par des valeurs qui ne sont pas des types. Par exemple, on peut imaginer un type `bitvector n` qui représente les vecteurs de bits de taille `n`, où `n` est un entier naturel dans le type `nat`. Dans ce cas, le pré-traitement d'un énoncé contenant ce type est plus subtil. En effet, si la structure de données ne change pas pendant le pré-traitement, alors la valeur `n` doit rester dans le type source `nat`, car un changement de type rendrait l'énoncé associé mal typé. Si la structure change, par exemple pour une autre structure indexée par un entier dans `Z`, alors il faut permettre à l'utilisateur d'associer de telles structures dépendantes.

5.2 La famille `zify` : fonctionnalités et limites

La tactique `zify` [53] est un outil de pré-traitement d'énoncés arithmétiques et logiques pour Coq, conçu pour la tactique `lia`. Il s'agit du point de départ du travail effectué sur Trakt et d'une source d'inspiration pour certains choix de conception. Elle traduit certaines représentations des entiers (`nat`, `positive`, *etc.*) vers le type `Z`, représentation des entiers utilisée par la tactique ciblée. Cette tactique est certifiante et ne laisse pas d'obligation de preuve à l'utilisateur. Dans cette section, nous présentons le fonctionnement global de cet outil et évaluons sa réponse au cahier des charges défini à la section précédente.

[53] : Besson (2017), « ppsimpl : a reflexive Coq tactic for canonising goals »

5.2.1 Pré-traitement modulaire de l'arithmétique

La modularité vis-à-vis d'une signature unique et fixe pour une théorie peut être représentée par un type enregistrement ou un type de module Coq. Ainsi, pour l'arithmétique, on pourrait avoir un enregistrement défini par un type porteur, les opérations d'addition, de multiplication, *etc.* Cependant, pour permettre à l'utilisateur de ne pré-traiter que partiellement la théorie, ou d'ajouter de nouveaux symboles, c'est-à-dire de nouvelles opérations arithmétiques à prendre en compte, la tactique `zify` offre un degré d'extensibilité sous la forme de classes de types.[7] Ainsi, il existe une classe de types pour chaque catégorie de symboles à pré-traiter.

7 : Concept présenté en § 3.1.2.

Toutes les instances de classes de types sont ensuite exploitées lors de la traversée du but d'origine, chaque symbole étant remplacé par son symbole associé, en ajoutant un sous-terme à la preuve globale, qui ultimement est une preuve d'implication entre le nouveau but obtenu et le but d'origine.

Plongement de types La première classe, `InjTyp`, permet de déclarer un *plongement* entre deux types Coq :

```
Class InjTyp (S T : Type) := {
  φ : S → T;
  P : T → Prop;
  π : forall (x : S), P (φ x)
}.
```

La fonction `φ` est une injection du type source `S` vers le type cible `T`. Le plongement peut être partiel grâce aux deux champs suivants : `P` est une propriété sur les valeurs du type cible, et `π` une preuve que toute valeur injectée respecte cette propriété. Ceci permet de représenter un sous-typage, comme celui de `nat` dans `Z` évoqué précédemment. Dans le cas de deux types équivalents, comme `int` et

Z, les champs P et π sont remplis d'une façon triviale, par exemple la suivante :

```
P _ := True
π _ := I
```

Dans le cas du plongement *partiel* de nat dans Z, la propriété est la positivité :

```
φ := Z.of_nat
P (n : nat) := Z.of_nat n ⩾ 0
π : forall (n : nat), Z.of_nat n ⩾ 0
```

Plongement de symboles Pour que le pré-traitement remplace une opération par l'opération canonique associée, on utilise des classes pour diverses arités : CstOp, UnOp, BinOp. Voici la définition de la classe BinOp :

```
Class BinOp
  {S₁ S₂ S₃ T₁ T₂ T₃ : Type}
  `{InjTyp S₁ T₁} `{InjTyp S₂ T₂} `{InjTyp S₃ T₃}
  (op : S₁ → S₂ → S₃)
    := {
      op' : T₁ → T₂ → T₃;
      π₂ : forall (x₁ : S₁) (x₂ : S₂), φ (op x₁ x₂) = op' (φ x₁) (φ x₂)
    }.
```

On lie une opération binaire op, définie grâce à trois types déclarés plongeables, à une opération op', définie avec les trois types cibles associés, par une preuve de morphisme π_2.

Exemple 5.2.1
Voici la déclaration d'un plongement de l'addition de nat vers l'addition de Z :

```
op' := Z.add
π₂ : forall (n₁ n₂ : nat),
        Z.of_nat (n₁ + n₂) = Z.of_nat n₁ + Z.of_nat n₂
```

5.2.2 Pré-traitement de la logique

Le traitement de la logique dans zify est similaire au traitement de l'arithmétique. Tout comme il existe des classes de types pour les opérations, le greffon définit des classes de types pour différentes arités de relations.

Plongement des relations La tactique zify est également capable de pré-traiter des relations binaires homogènes grâce à la classe BinRel :

```
Class BinRel {S T : Type} (R : S → S → Prop) `{InjTyp S T} := {
  R' : T → T → Prop;
  πR2 : forall (x₁ x₂ : S), R x₁ x₂ ↔ R' (φ x₁) (φ x₂)
}
```

Ainsi, on peut plonger, par exemple, une relation d'ordre sur le type positive vers la relation d'ordre sur Z :

```
R' := Z.ge
πR2 : forall (p1 p2 : positive), p1 ⩾ p2 ↔ Zpos p1 ⩾ Zpos p2
```

Traitement général de la logique D'autres fonctionnalités sont proposées, par exemple un mécanisme de saturation, afin de retrouver des propriétés perdues lors du plongement, telles que la préservation de la positivité par la multiplication des entiers naturels, ou encore la possibilité de déclarer des morphismes pour l'équivalence dans `Prop`, pour permettre de gérer différents connecteurs logiques. Le pré-traitement de la logique en général est possible *via* l'utilisation des classes `BinRel` et `BinOp`. En effet, les connecteurs booléens doivent être déclarés comme des opérateurs binaires de la classe `BinOp` (avec `S3 := bool`) et associés soit à eux-mêmes, soit à leur version dans `Prop` grâce à un plongement trivial de `bool` dans `Prop`, soit à des opérateurs dans un type entier, certains buts utilisant la logique booléenne au milieu de calculs arithmétiques.

Exemple 5.2.2
Avec `zify`, il est possible de pré-traiter le but suivant pour en obtenir une version prouvable par la tactique `lia` :

```
forall (n : nat), n + n ⩾ n
```

Le but associé obtenu est alors le suivant :

$$\frac{\texttt{n : nat} \qquad \texttt{p}_\texttt{n}\texttt{ : Z.of\_nat n} \geqslant \texttt{0}}{\texttt{Z.of\_nat n + Z.of\_nat n} \geqslant \texttt{Z.of\_nat n}}$$

Ici, la propriété $\texttt{p}_\texttt{n}$ représente la partialité du plongement de `n` dans `Z`. L'opérateur de comparaison utilisé en sortie est celui du type `Z`.

5.2.3 L'extension `mczify`

Il existe une extension de `zify` appelée `mczify` [54], ayant pour but de gérer les différents types d'entiers définis dans la bibliothèque MathComp, ainsi que les opérateurs associés. Elle a pu être construite grâce au caractère extensible de `zify`, puisqu'elle consiste en une série de déclarations d'instances des classes de types présentées précédemment. Les déclarations liées aux opérateurs MathComp font de `mczify` un travail d'orfèvre, car il s'agit d'opérateurs génériques.

[54] : Sakaguchi (2019–2022), *Micromega tactics for Mathematical Components*

Exemple 5.2.3
Grâce à l'utilisation de la surcouche `mczify`, le but suivant peut être traité par `zify` :

```
forall (x : int), x ⩾ 1 → x * x ⩾ x
```

Le sous-terme `x * x` est en réalité `@GRing.mul int_Ring x x`, où `GRing.mul` est la projection permettant d'obtenir l'opération de multiplication à partir de l'instance d'anneau `int_Ring` déclarée dans la bibliothèque MathComp pour le type `int`. De la même manière, `x ⩾ 1` est en réalité `@Order.ge int_porderType x (@GRing.one int_Ring)`, où `int_porderType` est l'instance pour `int` de la structure MathComp représentant les relations d'ordre. On note que la valeur `1` est également un champ des structures d'anneau. Les déclarations ajoutées par `mczify` permettent donc d'utiliser le polymorphisme *ad hoc* de MathComp, sur la théorie de l'arithmétique comme sur la logique booléenne, en restant compatible avec `zify`.

5.2.4 Limites de `zify`

Il est clair que la tactique `zify` remplit son rôle d'outil de pré-traitement pour `lia`, car les déclarations de base présentes dans le greffon pour les différentes classes de types présentées ciblent exactement l'espace des buts reconnaissables par la tactique d'automatisation. L'outil, tout comme la tactique ciblée, est capable de passer sous les différents quantificateurs et connecteurs logiques présents dans le but, ainsi que de pré-traiter les opérations et valeurs de la signature de l'arithmétique de Presburger, ceci pour les types d'entiers de la bibliothèque standard, les plus fréquemment utilisés. Grâce à son extensibilité, le greffon n'est pas limité à ces types de référence. Comme le montre l'extension `mczify`, il peut être étendu aux types et opérateurs d'une bibliothèque personnalisée, y compris ceux d'une certaine complexité.

Néanmoins, cet outil *ad hoc* conçu spécifiquement pour `lia` peut difficilement être utilisé dans un contexte différent. Par exemple, au sein de la classe de buts décidables, fréquents dans les preuves et peu intéressants à prouver pour l'humain,[8] se trouvent les problèmes SMT, dans lesquels apparaissent des *symboles non interprétés*, en particulier des fonctions.[9] Plusieurs greffons pour Coq permettent d'envoyer ces énoncés à des solveurs SMT, comme Itauto [55] ou SMTCoq, mais ces interfaces ont également une signature d'entrée à respecter.[10] En ce sens, elles souffrent du même problème que la tactique `lia`. Or, sur de tels énoncés, les outils de la famille `zify` sont insuffisants par nature, car ils n'ont pas été conçus dans le but de traiter ces cas qui dépassent le fragment logique prouvable par `lia`. Par exemple, dans le cas d'une fonction non interprétée `f` à valeurs dans `int` et appliquée à un entier `x`, la tactique `zify` considère le terme `f x` comme une seule valeur de type `int`. Cette situation est acceptable dans le cadre de l'usage de `lia`, car le comportement de la tactique est aligné avec celui de `zify`, mais si l'énoncé associé est donné à un solveur SMT, il peut ne pas réussir à le prouver à cause de cette perte d'information lors du pré-traitement.

De plus, dans le cadre de l'usage d'un outil d'automatisation qui nécessite de faire disparaître tout élément logique exprimé dans `Prop`, tel que SMTCoq, la tactique `zify` effectue un traitement incomplet. En effet, bien que les relations sur les entiers puissent être remplacées par leur équivalent booléen, un connecteur logique dans `Prop` ne peut être changé en sa version booléenne, car la classe `BinRel` exigerait dans ce cas que `Prop` soit plongeable dans `bool`, ce qui n'est pas vrai dans le cas général. Il semble intéressant de construire un pré-traitement pour la logique indépendant de ce qui est fait sur les signatures des théories.

Une autre limite conceptuelle de `zify` est l'utilisation de termes Coq pour stocker les informations utilisateur. En effet, utiliser des classes de types permet d'exploiter les fonctionnalités d'inférence de Coq, mais oblige le développeur à modéliser ses données dans un type Coq, ce qui peut manquer de flexibilité. Par exemple, il faut une classe de type par arité possible sur les opérations et les relations. De plus, il peut devenir difficile de capturer sous un même type Coq les différentes associations possibles entre des types et des symboles, en particulier si l'on souhaite construire un outil de pré-traitement qui prend en charge le polymorphisme et les types dépendants, ce qui n'est pas le cas de `zify`. Par conséquent, il existe un espace à combler entre les buts initiaux entrés dans Coq par l'utilisateur et les outils de preuve automatique.

Exemple 5.2.4
Le but suivant entre dans la théorie UFLIA :[11]

8 : Il s'agit de la classe de buts où un travail d'automatisation est le plus attendu.

9 : On parle de théorie de l'égalité, de théorie de la congruence, ou dans la terminologie SMT, de la théorie UF (*Uninterpreted Functions*).

[55] : Besson (2021), « Itauto : An Extensible Intuitionistic SAT Solver »

10 : Le projet SMTCoq dispose d'une phase de pré-traitement mais elle est sommaire et peu extensible.

11 : *Uninterpreted Functions and Linear Integer Arithmetic*.

```
forall (f : int → int) (x : int), f (2 * x) ⩽? f (x + x)
```

En théorie, il est prouvable par un solveur SMT, mais il est en dehors du fragment accepté par les tactiques de la famille `zify`. Sans pré-traitement, les greffons en charge de déléguer des preuves à un solveur SMT ne peuvent pas le prouver. TRAKT a pour objectif de faire le pont entre ce type de buts et ces greffons.

Fonctionnement théorique 6

Le premier prototype développé dans cette thèse, Trakt [12], est un outil de pré-traitement par canonisation des énoncés dans Coq, ayant pour objectif d'étendre le pré-traitement effectué par la tactique `zify` aux énoncés de la famille SMT. On souhaite donc dépasser les limites de `zify` sans introduire de régression dans les fonctionnalités déjà présentes. Ce chapitre traite de la manière dont la canonisation s'effectue dans Trakt. Dans un premier temps, nous listons les différentes informations que l'utilisateur peut fournir à l'outil (§ 6.1), puis nous détaillons l'algorithme utilisé pour répondre au problème théorique (§ 6.2).

6.1 Collecte des informations de l'utilisateur 56
6.1.1 Plongements de types 56
6.1.2 Plongements logiques 57
6.1.3 Plongements de symboles . . 58
6.1.4 Clés de conversion 58
6.2 Algorithme de pré-traitement 59
6.2.1 Gestion des quantificateurs universels 59
6.2.2 Traitement des connecteurs logiques 62
6.2.3 Pré-traitement spécifique à une théorie 63
6.2.4 La tactique `trakt` 65

[12] : Blot *et coll.* (2023), « Compositional pre-processing for automated reasoning in dependent type theory »

6.1 Collecte des informations de l'utilisateur

La canonisation n'est rendue possible que par la connaissance des différents plongements existant à partir des termes trouvables dans le but d'entrée de l'outil de pré-traitement. Trakt permet de déclarer différentes sortes d'informations : les plongements de types, les plongements logiques, les plongements de symboles et les clés de conversion.

6.1.1 Plongements de types

Comme expliqué en § 3.3.2, pour faire un transfert de preuve, il est nécessaire de définir une relation entre les types sources et les types cibles. Dans Trakt, cette relation est la bijection et elle est définie pour des types inductifs sans paramètre ni indice tels que les types d'entiers. On déclare un type A plongeable dans un type B lorsqu'il existe une bijection entre eux :[1]

$$\Sigma(\phi : A \to B)(\psi : B \to A).\, (\psi \circ \phi \doteq \mathrm{id}) \times (\phi \circ \psi \doteq \mathrm{id})$$

Une commande est mise à disposition de l'utilisateur pour déclarer ces informations :[2]

```
Trakt Add Embedding A B φ ψ π1 π2.
```

Cette déclaration permet de plonger toute valeur de type A ou n'importe quel type fonctionnel contenant A vers le type cible associé. Par exemple, une valeur de type $A \to A$ peut être plongée vers une valeur de type $B \to B$. En particulier, Trakt est capable de traiter les symboles non interprétés présents dans les énoncés de la classe SMT, ainsi que les variables quantifiées universellement ayant un type éligible au plongement.

Plongements partiels Il existe des pré-traitements pertinents pour lesquels les deux fonctions de plongement ne sont pas inverses l'une de l'autre. C'est le cas du plongement de $\mathbb{N}$ dans $\mathbb{Z}$ par exemple, pour lequel la fonction de plongement pseudo-inverse n'est pas injective puisqu'elle associe une valeur par défaut aux entiers négatifs, n'ayant pas d'équivalent dans l'espace des entiers naturels. Dans ce cas, on ne peut prouver qu'une version affaiblie de la rétraction, dans laquelle la propriété n'est vraie que sur les valeurs pour lesquelles ψ n'effectue

1 : On note $\doteq$ l'égalité point à point, c'est-à-dire :

$$f \doteq g \quad := \quad \Pi x.\, f\, x = g\, x$$

2 : Ici, π_1 et π_2 représentent les preuves de section et rétraction montrant que φ et ψ sont inverses l'une de l'autre (respectivement les première et seconde preuves de la paire située sous le Σ-type ci-contre).

pas de troncature, c'est-à-dire s'il existe un antécédent dans A pour ces valeurs. Pour le plongement de $\mathbb{N}$ dans $\mathbb{Z}$, cette condition appelée *condition de plongement* sera la positivité. On représente la condition de plongement par des données supplémentaires dans la preuve fournie par l'utilisateur : la condition ainsi qu'une preuve que tout plongement depuis A la respecte. Le plongement partiel est donc défini comme suit :

$$\Sigma(\phi : A \to B)(\psi : B \to A)(P : B \to \mathbb{P}).$$
$$(\psi \circ \phi \doteq \mathrm{id}) \times (\Pi b : B.\, P\, b \to \phi\, (\psi\, b) = b) \times (\Pi a : A.\, P\, (\phi\, a))$$

La commande Coq permettant d'ajouter les plongements de types à Trakt accepte également les plongements partiels :[3]

```
Trakt Add Embedding A B φ ψ P π1 π2P πP.
```

Lors du plongement d'une variable, la condition sera explicitée de sorte qu'il soit toujours possible de prouver que le but final implique le but initial. Ainsi, une fonction non interprétée $f : \mathbb{N} \to \mathbb{N}$ sera remplacée par une fonction associée $f' : \mathbb{Z} \to \mathbb{Z}$ accompagnée de la propriété suivante :

$$\Pi x : \mathbb{Z}.\quad x \geq 0 \to f'\, x \geq 0$$

3 : Ici, π_{2P} est la preuve de rétraction conditionnelle et π_P est la preuve que tout plongement depuis `A` vérifie la condition de plongement.

6.1.2 Plongements logiques

Dans un énoncé de la classe SMT, les sous-termes contenus dans une des théories traitées sont des atomes dans l'arbre qui représente la formule logique, les nœuds de cet arbre étant les connecteurs logiques, égalités et autres prédicats d'arité arbitraire. Selon la tactique d'automatisation ciblée après le passage de Trakt, ces prédicats doivent aussi être traduits, soit vers une version booléenne, soit vers une version dans $\mathbb{P}$. Trakt permet de déclarer un plongement entre deux prédicats P et Q ayant les types suivants, où chaque type T_i' est soit T_i lui-même, soit un plongement à partir de T_i :

$$P : T_1 \to \cdots \to T_n \to L$$
$$Q : T_1' \to \cdots \to T_n' \to L'$$

La déclaration se fait en fournissant une preuve d'équivalence entre les deux prédicats :

$$\Pi x_1 \cdots x_n.\quad P\, x_1 \cdots x_n \quad \bowtie_{L,L'} \quad Q\, (\phi_1^?\, x_1) \cdots (\phi_n^?\, x_n)$$

L	L'	$\bowtie_{L,L'}$
$\mathbb{B}$	$\mathbb{B}$	$=$
$\mathbb{P}$	$\mathbb{B}$	$\lambda P\, b.\ P \leftrightarrow b = 1_{\mathbb{B}}$
$\mathbb{B}$	$\mathbb{P}$	$\lambda b\, P.\ b = 1_{\mathbb{B}} \leftrightarrow P$
$\mathbb{P}$	$\mathbb{P}$	$\leftrightarrow$

Les codomaines logiques des prédicats L et L' sont soit $\mathbb{P}$ soit $\mathbb{B}$, et $\bowtie_{L,L'}$ est une manière d'exprimer l'équivalence en fonction de ces codomaines. $\phi_i^?$ désigne une fonction de plongement entre T_i et T_i', optionnelle au niveau méta : si les deux types sont identiques, ce terme est absent.

Exemple 6.1.1
Dans le cas d'un plongement de l'égalité sur le type `int` vers l'égalité booléenne sur `Z`, la preuve à déclarer à Trakt a le type suivant :

```
forall (x y : int), x = y <-> Z_of_int x =? Z_of_int y = true
```

Dans ce but, `Z_of_int` est la fonction de plongement de `int` vers `Z` et `=?` l'égalité booléenne sur `Z`.

La commande mise à disposition de l'utilisateur pour les déclarations de plongements logiques est la suivante :

```
Trakt Add Relation n P Q π_L.
```

La valeur `n` est l'arité des prédicats déclarés,[4] `P` et `Q` sont les relations source et cible, et π_L est la preuve d'équivalence entre les deux prédicats.

4 : On demande cette information pour qu'il soit possible de déclarer des relations particulières s'exprimant avec plusieurs termes, comme l'égalité qui prend un argument de type.

6.1.3 Plongements de symboles

La base d'une théorie mathématique est souvent un ensemble muni d'opérations. La signature d'une théorie dans Coq est donc la donnée d'un type dit *porteur* accompagné de valeurs et d'opérations définies sur ce type. Le plongement de types permet de gérer la variabilité sur les types porteurs dans les différents énoncés, mais il reste à traiter les symboles de la signature. Pour ce faire, Trakt rend également possible de déclarer des plongements entre les différents symboles, quelle que soit leur arité, par la donnée d'un symbole source et d'un symbole cible, ainsi que la propriété de morphisme :

$$\begin{aligned} s &: T_1 \to \cdots \to T_{n+1} \\ s' &: T'_1 \to \cdots \to T'_{n+1} \end{aligned}$$

$$\Pi x_1 \cdots x_n. \quad \phi^?_{n+1}\,(s\,x_1 \cdots x_n) = s'\,(\phi^?_1\,x_1) \cdots (\phi^?_n\,x_n)$$

Exemple 6.1.2
On peut plonger l'addition sur le type `nat` vers l'addition sur `Z` en donnant une preuve de l'énoncé de l'Exemple 5.2.1, et le zéro de `nat` vers celui de `Z` en explicitant à Trakt que `Z.of_nat O = Z0`.

La déclaration d'un symbole se fait grâce à la commande suivante :

```
Trakt Add Symbol S S' π_S.
```

Les valeurs `S` et `S'` sont les symboles source et cible, et π_S est la preuve de morphisme.

6.1.4 Clés de conversion

Par défaut, pour des raisons de performance à l'implémentation, la reconnaissance de termes dans Trakt se fait de manière purement syntaxique. En revanche, afin de maintenir la fonctionnalité supplémentaire apportée par `mczify` et présentée à l'Exemple 5.2.3, si l'utilisateur déclare des plongements sur des opérations concrètes ensuite empaquetées dans des structures, il faut que l'outil de pré-traitement soit capable de détecter que les projections génériques sur ces structures donnent les mêmes termes que ceux qui ont été déclarés. Il faut alors que la conversion de Coq puisse être utilisée localement pour ces termes que l'on appelle *clés de conversion*.

Dans l'Exemple 5.2.3, si l'utilisateur déclare un plongement à partir de la multiplication concrète sur le type `int`, alors la présence de notations génériques dans les buts ne doit pas dégrader le pré-traitement. Par conséquent, le terme `@GRing.mul int_Ring` doit être pré-traité comme si l'opération concrète avait été utilisée à cet endroit, ce qui est possible en déclarant `GRing.mul` comme clé de conversion. Un tel comportement est possible car la projection se réduit précisément vers l'opération concrète. Le type de la preuve ne sera donc pas invalide.

La commande utilisée pour déclarer des clés de conversion est la suivante :

```
Trakt Add Conversion K.
```

où `K` est le terme à déclarer comme clé de conversion.

6.2 Algorithme de pré-traitement

À partir de toutes les informations fournies à Trakt par l'utilisateur, l'outil est en mesure d'effectuer le pré-traitement du but d'entrée. Cette section détaille le fonctionnement de l'algorithme de pré-traitement sur les quantificateurs universels, les connecteurs logiques et les sous-termes contenus dans une théorie, avant de présenter la tactique `trakt` qui l'implémente.

L'algorithme prend un but à pré-traiter en entrée, et est paramétré par le type cible désiré par l'utilisateur pour exprimer la logique ainsi que le type cible pour la théorie à pré-traiter dans le but. Il génère ensuite un but de sortie ainsi qu'une preuve qu'il implique le but d'entrée. En raison de la polarité des connecteurs logiques, sur certains sous-termes, cette preuve d'implication doit être générée dans le sens inverse, subtilité gérée par l'algorithme.

Sauf explicitement indiqué, dans les exemples, on considère un plongement vers le type `Z` et une expression de la logique dans `Prop`.

6.2.1 Gestion des quantificateurs universels

La première construction rencontrée dans un but à pré-traiter est souvent un quantificateur universel. Ce cas est traité par un appel récursif sur le sous-terme et un combinateur pour étendre la preuve obtenue au quantificateur. En revanche, le type de la variable liée dans le nouveau but doit exploiter au maximum les plongements de types déclarés par l'utilisateur, les types fonctionnels étant aussi pris en compte, comme expliqué en § 6.1.1. Le traitement est donc différent selon le type de la variable liée avant et après traduction, et la polarité au moment de traduire le quantificateur.

Type inchangé Si le type de la variable liée ne change pas, alors on doit prouver

$$\Pi x : A.\, B' \;\to\; \Pi x : A.\, B$$

à partir de la preuve $p : B' \to B$ obtenue sur le sous-terme.[5] Le combinateur à utiliser est le suivant, quelle que soit la polarité :

$$\lambda (H : \Pi (x : A).\, B')(x : A).\, p\,(H\,x)$$

5 : Le cas contravariant inverse B et B' dans les types des preuves.

Type plongé, cas covariant Si le type change, alors toutes les occurrences de la variable x font l'objet d'un plongement $\phi_{A \rightsquigarrow A'}$ lors de la traduction du sous-terme.[6] Le sous-terme de sortie au niveau du quantificateur est donc obtenu en faisant abstraction de ces plongements, c'est-à-dire en remplaçant toutes les occurrences de $\phi_{A \rightsquigarrow A'}\ x$ par une variable x' du même type A', afin d'obtenir un nouveau sous-terme ne dépendant que de la nouvelle variable x' et de pouvoir refermer le terme quantifié. Ainsi, dans le cas covariant, la preuve à fournir au niveau du quantificateur est

$$\Pi x' : A'.\, B'' \quad \rightarrow \quad \Pi x : A.\, B$$

et l'on part d'une preuve $p : B' \rightarrow B$.[7] Ici, B'' est la traduction du sous-terme B' dans lequel on a remplacé les plongements de x par x'.[8] Par conséquent, le combinateur peut instancier l'hypothèse avec le plongement de la variable x, fournissant une preuve de $B''[x' := \phi_{A \rightsquigarrow A'}\ x] \equiv B'$. La preuve p peut alors être directement appliquée. Le nouveau combinateur est le suivant :

$$\lambda(H : \Pi(x' : A').\, B'')(x : A).\, p\ (H\ (\phi_{A \rightsquigarrow A'}\ x))$$

Dans le cas où le passage de A à A' pour x contient un plongement partiel, la variable liée x' dans le but de sortie doit être accompagnée d'une propriété $P\,x'$ combinant différentes conditions de plongements définies en § 6.1.1. La preuve à fournir est donc la suivante :

$$\Pi x' : A'.\, P\,x' \rightarrow B'' \quad \rightarrow \quad \Pi x : A.\, B$$

Comme le combinateur instancie x' avec le plongement de x, la propriété est toujours vraie, par composition des preuves fournies par l'utilisateur.[9] En notant π^*_P cette combinaison de preuves, le combinateur dans le cas de plongements partiels est le suivant :

$$\lambda(H : \Pi(x' : A').\, P\,x' \rightarrow B'')(x : A).\, p\ (H\ (\phi_{A \rightsquigarrow A'}\ x)\ (\pi^*_P\ x))$$

On considère l'énoncé suivant, représentant un lemme permettant d'effectuer une preuve qu'une fonction est constante par récurrence sur son domaine, les entiers naturels :

```
forall (f : nat → nat) (k : nat),
  f 0 = k → (forall (n : nat), f (S n) = f n) →
    forall (n : nat), f n = k
```

Lors d'un pré-traitement où `nat` est déclaré plongeable dans `Z`, le premier quantificateur est changé en une nouvelle variable liée `f' : Z → Z` avec une propriété combinant deux fois la condition de plongement de `nat` dans `Z` :

```
forall (x' : Z), x' ⩾ 0 → f' x' ⩾ 0
```

Lors de la preuve de pré-traitement du quantificateur, on prouve cette propriété grâce au fait que la valeur concrète pour `f'` est une composition de `f` avec les fonctions de plongement entre `nat` et `Z` :

```
f' := fun (x : Z) ⇒ Z.of_nat (f (Z.to_nat x))
```

En particulier, toute application de `f'` à un terme `t` peut être vue comme celle de `Z.of_nat` à `f (Z.to_nat t)`, donc elle est positive grâce à la preuve de la condition de plongement donnée lors de la déclaration du plongement entre `nat` et `Z` :

6 : Ici, on désigne par $\phi_{T \rightsquigarrow T'}$ un combinateur ajoutant toutes les fonctions de plongement nécessaires pour passer du type T potentiellement fonctionnel au type T' associé.

7 : Le cas des plongements partiels est traité au paragraphe suivant.

8 : $B'' \equiv B'[\phi_{A \rightsquigarrow A'}\ x := x']$

9 : La valeur π_P dans chaque déclaration de plongement partiel.

```
forall (n : nat), Z.of_nat n ⩾ 0
```

Type plongé, cas contravariant [10]

10 : On ne traite ici que le cas avec plongement partiel, les autres cas étant des simplifications de ce dernier.

Le cas contravariant supprime le besoin de prouver les conditions de plongement, car l'implication devant être prouvée dans l'autre sens, elles deviennent des hypothèses supplémentaires et non des arguments à fournir pour utiliser une hypothèse. Cependant, ces conditions demeurent utiles pour supprimer des identités de plongement qui apparaissent lors de l'instanciation des hypothèse. En effet, la preuve à construire a le type suivant :

$$\Pi x : A.\, B \quad \to \quad \Pi x' : A'.\, P\, x' \to B''$$

Le combinateur a donc à sa disposition une variable $x' : A'$ ainsi qu'une preuve de $P\, x'$, et il doit instancier une hypothèse $H : \Pi(x : A).\, B$. La solution est le plongement inverse de x' que l'on peut noter $\psi_{A \hookleftarrow A'}$. On obtient alors une preuve de $B[x := \psi_{A \hookleftarrow A'}\, x']$, sachant que l'on dispose de la preuve sur le sous-terme $p : B \to B'$. On peut remarquer que p est construite à partir de la variable x obtenue en passant sous le quantificateur, donc on peut l'exprimer en fonction de x. Ainsi, en substituant $\psi_{A \hookleftarrow A'}\, x'$ à x dans p par une opération au niveau méta, on peut appliquer cette preuve à l'hypothèse instanciée précédemment avec cette même valeur de plongement inverse, donnant une preuve de $B'[x := \psi_{A \hookleftarrow A'}\, x']$. Or, il reste à prouver B'', c'est-à-dire $B'[\phi_{A \rightsquigarrow A'}\, x := x']$ par définition.

Pour unifier ces deux types, on peut remarquer que comme la substitution effectuée dans le premier type remplace les occurrences de x par le plongement inverse de x', elle remplace aussi les occurrences du plongement de x par une composition de plongements appliquée à x' :

$$x := \psi_{A \hookleftarrow A'}\, x' \quad \Longrightarrow \quad \phi_{A \rightsquigarrow A'}\, x := (\phi_{A \rightsquigarrow A'} \circ \psi_{A \hookleftarrow A'})\, x'$$

Grâce à la preuve de la condition de plongement disponible pour x', on peut systématiquement réécrire cette composition en l'identité partout où elle apparaît dans la preuve de $B'[x := \psi_{A \hookleftarrow A'}\, x']$, et la preuve finale obtenue est bien B''. Le combinateur contravariant est donc le suivant :

$$\lambda(H : \Pi(x : A).\, B)(x' : A')(c : P\, x').\, \pi_c^{\mathrm{rw}}\ (p[x := \psi_{A \hookleftarrow A'}\, x']\ (H\ (\psi_{A \hookleftarrow A'}\, x')))$$

où π_c^{rw} est la preuve de réécriture de toutes les compositions $\phi_{A \rightsquigarrow A'} \circ \psi_{A \hookleftarrow A'}$ en identités dans le type de la preuve en argument, à l'aide de la preuve dont c est le témoin.

Prenons un énoncé simple sur tout entier naturel :

```
forall (n : nat), n * 0 = 0
```

Son pré-traitement donnera l'énoncé suivant :

```
forall (n' : Z), n' ⩾ 0 → n' * 0 = 0
```

Une preuve contravariante donnera une preuve sur le sous-terme suivant :

```
p : n * 0 = 0 → Z.of_nat n * 0 = 0
```

La preuve à effectuer pour autoriser la substitution d'énoncé est la suivante :

```
H : forall (n : nat), n * 0 = 0
n' : Z
c : n' ⩾ 0
──────────────────────────────
          n' * 0 = 0
```

Comme expliqué précédemment, on construit une nouvelle preuve basée sur `p` en remplaçant la variable `n` par `Z.to_nat n'` et on l'applique à `H (Z.to_nat n')`. On obtient une preuve de `Z.of_nat (Z.to_nat n') * 0 = 0`, et la preuve `c` de la condition de plongement sur `n'` permet de réécrire cette composition de plongements en une identité grâce à un lemme utilisateur,[11] puis de conclure.

11 : La valeur π_{2p} lors de la déclaration du plongement partiel.

6.2.2 Traitement des connecteurs logiques

Dans un premier temps, l'algorithme traverse les quantificateurs mais aussi les parties logiques du but, c'est-à-dire les différents connecteurs logiques, jusqu'à atteindre les prédicats ou relations marquant le passage vers les sous-termes spécifiques à une théorie. Au niveau de chaque connecteur, on applique une preuve associée permettant de faire un ou plusieurs appels récursifs sur les sous-termes. L'objectif étant de générer une preuve d'implication entre les buts, on utilise des propriétés de morphisme de l'implication par rapport aux connecteurs logiques. Ainsi, la construction d'une preuve d'implication en traversant un connecteur $K\ P_1\ \cdots\ P_n$ peut se faire à partir des preuves d'implication construites sur les arguments P_i, dans un sens ou l'autre selon la polarité de la position de chaque argument pour ce connecteur.

Le lemme de morphisme attendu pour K a le type suivant :

$$\Pi P_1\ \cdots\ P_n\ P'_1\ \cdots\ P'_n.\\ (P_1\ \diamond_1^K\ P'_1) \to \cdots \to (P_n\ \diamond_n^K\ P'_n) \to (K\ P'_1\ \cdots\ P'_n \to K\ P_1\ \cdots\ P_n)$$

$$\text{où } \diamond_i^K := \begin{cases} \to & \text{si la position } i \text{ est covariante pour le connecteur } K \\ \leftarrow & \text{sinon} \end{cases}$$

Pré-traiter une disjonction `A ∨ B` revient à générer respectivement `A'` et `B'` ainsi qu'une preuve de `A' ∨ B' → A ∨ B` à partir des sous-termes `A` et `B`. Dans ce cas, Trakt utilise donc un lemme montrant que l'implication est un morphisme pour la disjonction :

```
Lemma or_impl_morphism : forall (A B A' B' : Prop),
  (A' → A) → (B' → B) → (A' ∨ B' → A ∨ B).
```

Il s'agit d'une instance du cas général ci-dessus, où $n = 2$ et $\diamond_i^\vee = \to$ pour tout i car toutes les positions sont covariantes, tandis que dans le cas d'une implication, le premier argument serait en position contravariante.

Gestion de la logique booléenne Si le type logique ciblé est différent du type logique du connecteur inspecté, alors on tente d'exprimer tous les arguments du connecteur dans le type ciblé. Si c'est possible, alors on peut remplacer le connecteur par sa version associée en ajoutant un lemme d'implication entre les deux.

Si l'on cible `bool` pour la logique et que le but contient une disjonction dans `Prop`, on tente de pré-traiter chaque argument en un booléen injecté dans `Prop`. Lorsque c'est possible, on peut appliquer le lemme suivant qui permet d'utiliser la disjonction booléenne `||` dans le but de sortie :

```
Lemma orb_or_impl : forall (b1 b2 : bool),
  b1 || b2 = true → b1 = true ⋁ b2 = true.
```

Par induction, il est ainsi possible de transférer tout un arbre logique de `Prop` à `bool` ou inversement, lorsque tous les atomes le permettent.

Atomes logiques Le parcours du terme se poursuit jusqu'à trouver un atome logique, c'est-à-dire soit `True` ou `False`, soit un prédicat déclaré dans Trakt sous lequel se trouvent des termes contenus dans la théorie à pré-traiter. Il s'agit du cas des plongements logiques définis en § 6.1.2, où la preuve donnée par l'utilisateur est une équivalence logique à partir de laquelle on peut obtenir une preuve d'implication. Dans le cas où les arguments du prédicat sont dans un type éligible à un plongement, la version de la relation utilisée dans le but de sortie introduit des plongements devant ces sous-termes, ouvrant la voie au pré-traitement spécifique détaillé à la prochaine sous-section.

On considère le but suivant :

```
forall (x : int), x = x
```

On suppose que l'utilisateur a donné une preuve de plongement de l'égalité sur `int` vers l'égalité booléenne sur `Z`, présentée dans l'Exemple 6.1.1. À partir de cette preuve, on peut obtenir une implication dans le sens souhaité (ici, on suppose le sens covariant) :

```
forall (x y : int), Z_of_int x =? Z_of_int y = true → x = y
```

Ainsi, en passant sous ce nœud, on justifie le passage à la relation cible (l'égalité booléenne) et on lance deux appels récursifs de pré-traitement sur `Z_of_int x`, de part et d'autre de l'égalité.

6.2.3 Pré-traitement spécifique à une théorie

Une fois passé sous les atomes logiques, l'objectif de Trakt est de plonger un maximum de valeurs dans le type cible souhaité. L'algorithme va donc inspecter chaque nœud et exploiter les différentes preuves de morphisme déclarées par l'utilisateur. Toutes les valeurs inconnues sont traversées et laissées intactes.

Descente des fonctions de plongement Inspiré de `zify`, l'algorithme de Trakt introduit des fonctions de plongement dès que possible dans le but initial avant de les pousser vers les feuilles du terme. Ces fonctions de plongement sont le déclencheur permettant d'effectuer toutes les réécritures. En effet, les lemmes de morphisme sont utilisés pour remplacer leur membre gauche dans le terme d'entrée par leur membre droit dans le terme de sortie, et le membre gauche a un plongement en tête lorsque c'est possible. L'introduction d'un plongement, par exemple grâce à une preuve d'équivalence de prédicat, permet d'utiliser tous les lemmes de morphisme en cascade jusqu'aux feuilles de l'arbre.

Exemple 6.2.1
On considère le but suivant :

```
forall (x y : int), x * y + x = x * (y + 1)
```

Si l'utilisateur a déclaré un plongement logique à partir de l'égalité sur le type `int` vers l'égalité dans `Prop` sur le type `Z` par exemple, alors la preuve d'équivalence permet de lancer l'algorithme sur les deux membres de l'égalité précédés d'un plongement. Si toutes les opérations ont été déclarées plongeables vers leurs équivalents dans `Z`, alors le membre de gauche va subir cette liste de réécritures, poussant tous les plongements vers le bas :

```
Z_of_int (x * y + x)
Z_of_int (x * y) + Z_of_int x
Z_of_int x * Z_of_int y + Z_of_int x
```

Toutes les preuves d'égalité sont étendues à l'atome logique dans lequel elles sont utilisées, afin de pouvoir les composer par transitivité et obtenir une preuve d'égalité unique entre l'atome logique d'entrée et l'atome logique de sortie.

Lorsque le type d'un argument d'une fonction non interprétée est plongeable, afin de pré-traiter cet argument correctement, on insère une identité de plongement devant cet argument.[12] De cette façon, on s'assure que l'argument est précédé d'une fonction de plongement avant d'être pré-traité par Trakt.

12 : On utilise ici la première identité fournie par l'utilisateur, vraie pour tout plongement de manière inconditionnelle :

$$\psi \circ \phi \doteqdot \mathrm{id}$$

Traitement des feuilles de l'arbre Arrivé aux feuilles de l'arbre, Trakt fait disparaître un maximum de plongements pour obtenir un but de sortie entièrement exprimé dans le type cible. Ces feuilles sont soit des constantes d'arité nulle, soit des variables.

Dans le premier cas, la preuve fournie avec la constante permet de faire disparaître le plongement restant au profit d'une nouvelle constante dans le type cible. C'est le cas de l'Exemple 6.1.2 où le zéro du type `nat` est plongé vers celui du type `Z` à l'aide d'un lemme impliquant un plongement sur le membre gauche.

Dans le second cas, celui d'une variable $x : T$, si le type T est plongeable vers T', alors la variable est nécessairement l'argument d'une composition de fonctions de plongement $\phi_{T \rightsquigarrow T'}$. Il suffit alors de remplacer le terme $\phi_{T \rightsquigarrow T'}\ x$ par la variable de sortie $x' : T'$. Cette substitution est valide, la preuve correspondante étant effectuée lors de la fermeture du but de sortie. En effet, replacer le nouveau quantificateur au-dessus d'un terme ouvert traduit t', contenant des plongements sur la variable x, revient à étendre une preuve de $t' \to t$, obtenue par appel récursif, à l'implication suivante entre les types quantifiés :

$$\Pi x' : T'.\, t'[\phi_{T \rightsquigarrow T'} x := x'] \quad \to \quad \Pi x : T.\, t$$

Exemple 6.2.2
Si l'on reprend le but de l'exemple précédent, la descente des plongements dans les deux membres de l'égalité donne les deux termes suivants :

```
Z_of_int x * Z_of_int y + Z_of_int x
Z_of_int x * (Z_of_int y + Z_of_int 1)
```

Pour terminer le pré-traitement, on applique le lemme permettant de passer de la valeur `1` dans le type `int` à son équivalent dans `Z`, et on referme le terme

avec les nouveaux quantificateurs `x'` et `y'`, pour obtenir le but final suivant :

```
forall (x' y' : Z), x' * y' + x' = x' * (y' + 1)
```

6.2.4 La tactique trakt

L'algorithme présenté précédemment a fait l'objet d'une implémentation sous la forme d'une tactique `trakt`. Dès le passage en mode preuve, on appelle `trakt` pour pré-traiter le but, avant de laisser une tactique de preuve automatique terminer la preuve.

Cette tactique prend deux arguments correspondant aux paramètres de l'algorithme : le type cible pour la théorie traitée et le type d'expression de la logique. Ainsi, pour que les entiers soient exprimés dans `Z` et la logique dans `Prop`, on appellera `trakt Z Prop`.

Plusieurs des buts arithmétiques présentés dans cette partie peuvent être prouvés grâce à diverses déclarations ainsi que la combinaison de tactiques suivante :

```
Proof. trakt Z Prop; lia. Qed.
```

Lorsque le but requiert la théorie de l'égalité, on peut utiliser une tactique d'appel à un solveur SMT à la place de `lia`, telle que la tactique `smt` de SMTCoq ou bien celle du projet ITAUTO.

Il est également possible de pré-traiter un but uniquement pour réécrire sa partie logique, en omettant le premier argument. On passe alors vers un but booléen en appelant `trakt bool`.

Dans le cas d'une théorie pour laquelle le seul pré-traitement pertinent avant passage d'un outil de preuve automatique est d'exploiter la décidabilité des prédicats, il est inutile d'activer les fonctionnalités de pré-traitement spécifique à une théorie dans TRAKT. On se contente donc d'un pré-traitement logique en retirant le premier argument.

A priori, les informations fournies à TRAKT doivent être des termes déclarés avant la preuve, car ces informations sont stockées dans une base de données qui persiste après la fin de la preuve. Or, dans certaines preuves, il peut être pertinent que certaines informations soient portées à la connaissance de TRAKT, notamment des relations. L'outil possède une fonctionnalité pour couvrir ce cas, avec une nouvelle syntaxe :

`trakt T L with rel (R, R', π_R).`

où `T` et `L` sont les types cibles pour la théorie et la logique, et le triplet correspond à une déclaration de relation telle que présentée en § 6.1.2.

L'exemple le plus parlant pour illustrer le besoin de cette fonctionnalité est la décidabilité d'une relation sur un type local. En effet, si un but quantifie sur un type A et des informations impliquant par exemple que l'égalité sur A est décidable, alors il peut être pertinent de les introduire dans le contexte de preuve, d'expliciter la preuve de décidabilité et d'appeler `trakt` avec cette information supplémentaire, afin de pré-traiter le reste du but et peut-être obtenir une preuve plus simple à terminer.

Enfin, l'algorithme implémenté dans la tactique `trakt` permettant de raisonner dans les deux sens grâce à une gestion de la polarité présentée précédemment, il est possible de l'utiliser pour pré-traiter en chaînage avant sans effort supplémentaire. On peut donc pré-traiter une hypothèse plutôt que le but, cette fois en générant une preuve d'implication de la nouvelle hypothèse à partir de l'ancienne. La syntaxe proposée par Trakt dans ce cas est la suivante :

```
trakt_pose T L : H as H'.
```

où `H` est l'hypothèse à pré-traiter et `H'` le nom à utiliser pour la nouvelle hypothèse obtenue. Les autres fonctionnalités de Trakt sont également disponibles dans ce sens.

Conclusion et perspectives 7

Le greffon Trakt dont les concepts ont été présentés au chapitre précédent a été implémenté dans Coq.[1] Ce chapitre effectue le bilan de cet outil qui vient compléter un écosystème d'outils d'automatisation disponibles pour l'utilisateur Coq (§ 7.1), et peut interagir avec certains d'entre eux. Trakt améliore la réponse apportée par `zify` au problème du pré-traitement par canonisation pour les énoncés de la classe SMT (§ 7.2). Le greffon est partiellement aligné avec la spécification définie en § 5, mais il possède aussi quelques failles et pourrait être amélioré (§ 7.3).

7.1 Écosystème d'automatisation dans Coq 67
7.1.1 De la nécessité du pré-traitement 67
7.1.2 Transformations modulaires de la tactique scope 69
7.2 Succès de l'outil 71
7.2.1 Exemples de buts traités 71
7.2.2 Intégration de Trakt à d'autres outils 72
7.3 Voies d'amélioration 74
7.3.1 Polymorphisme et types dépendants 74
7.3.2 Architecture du pré-traitement 74

1 : Les questions liées à cette implémentation sont traitées en § 12.

7.1 Écosystème d'automatisation dans Coq

Sans automatisation, un logiciel aussi complexe qu'un assistant de preuve ne peut être utilisé convenablement. Les outils d'assistance à la preuve sont de plus en plus nombreux pour aider l'utilisateur. Ainsi, Trakt s'insère dans un écosystème d'outils utilisés soit pour prouver automatiquement les buts, soit pour les pré-traiter afin que les autres outils d'automatisation fonctionnent encore mieux. Dans cette section, nous exposons la difficulté d'effectuer des preuves en Coq et la nécessité du pré-traitement, en citant quelques outils de preuve automatique avec lesquels Trakt peut fonctionner ; puis nous présentons rapidement `scope`, une autre tactique de pré-traitement qui peut s'associer à Trakt.

7.1.1 De la nécessité du pré-traitement

Les étapes de preuve techniques mais inintéressantes sont le pain quotidien de la vérification de programmes. Heureusement, de nombreux énoncés élémentaires sont facilement prouvés par les prouveurs automatiques modernes. Les étapes de preuve *formelle* correspondantes peuvent alors être automatisées, par exemple en utilisant les *marteaux*, une architecture puissante pour connecter des prouveurs externes à des environnements de preuve formelle interactifs. Le greffon CoqHammer [40] équipe Coq d'une instance de marteau en fournissant une tactique `hammer` qui combine des heuristiques avec des appels à des prouveurs externes pour la logique du premier ordre, afin d'obtenir des indices que l'on espère suffisants, y compris des lemmes pertinents dans le contexte actuel, pour prouver le but. La preuve formelle réelle est ensuite reconstruite à partir de ces indices grâce à des variantes de la tactique `sauto` et `hammer` produit un script de preuve correspondant, robuste et indépendant de l'oracle utilisé.

[40] : Czajka *et coll.* (2018), « Hammer for Coq : Automation for dependent type theory »

Par exemple, on considère une propriété sur la longueur de la concaténation inversée de deux listes :

```
Lemma length_rev_app : forall (B : Type) (l l' : list B),
  length (rev (l ++ l')) = length l + length l'.
```

Ce lemme peut être prouvé avec CoqHammer, qui fournit le script suivant, en utilisant les lemmes auxiliaires `app_length` et `rev_app_length` de la section de la bibliothèque standard chargée des listes :

```
Proof. scongruence use: app_length, rev_length. Qed.
```

Cependant, dans sa version 1.3.2, CoqHammer n'est pas conçu pour exploiter un raisonnement spécifique à une théorie, et ne peut donc pas prouver cette légère variante, où `b :: l'` remplace `l'`, parce qu'il n'a pas de gestion spécifique de l'arithmétique :

```
Lemma length_rev_app_cons : forall (B : Type) (l l' : list B) (b : B),
  length (rev (l ++ (b :: l'))) = length l + length l' + 1.
```

Dans ce cas, les utilisateurs peuvent recourir au greffon SMTCoq, qui implémente un vérificateur de certificats pour les témoins de preuve produits par des solveurs SMT. Ces derniers sont en effet conçus pour trouver des preuves combinant le raisonnement propositionnel, la congruence et les procédures de décision spécifiques à une théorie, par exemple pour l'arithmétique linéaire. Cependant, ni CoqHammer ni SMTCoq ne peuvent en général raisonner par analyse de cas ou induction.

Une variante de la tactique `sauto` peut certes prouver le premier but ci-dessous concernant une concaténation de listes, mais pas le second, dans lequel la première liste est inversée :

```
Lemma app_nilI : forall (B : Type) (l l' : list B),
  l ++ l' = [] → l = [] ⋀ l' = [].

Lemma app_nil_rev : forall (B : Type) (l l' : list B),
  rev l ++ l' = [] → l = [] ⋀ l' = [].
```

On peut utiliser le greffon SMTCoq pour prouver des propriétés d'arithmétique linéaire sur les entiers, mais seulement lorsqu'elles sont énoncées en utilisant le type `Z` de la bibliothèque standard de Coq pour représenter les entiers :

```
Lemma eZ : forall (z : Z), z ⩾ 0 → z < 1 → z = 0.
```

Jusqu'à sa version 2.0, SMTCoq ignore cependant toute représentation alternative de l'arithmétique des entiers, par exemple le type `int` des entiers unaires de la bibliothèque MathComp, déjà évoqué précédemment.

```
Lemma eint : forall (z : int), z ⩾ 0 → z < 1 → z = 0.
```

Heureusement, Coq distribue la tactique `lia`, spécialisée dans l'arithmétique linéaire des entiers, qui peut également prouver des lemmes tels que `eZ`. De plus, `lia` peut être adaptée à une instance d'arithmétique utilisateur grâce à la tactique de pré-traitement dédiée `zify` présentée en § 5.2. Une fois correctement configurée pour le type `int` grâce à l'extension `mczify`, `lia` est tout aussi puissante sur le type `int` que sur le type `Z` et prouve à la fois `eZ` et `eint`. Cependant, aussi puissante qu'elle puisse être sur l'arithmétique linéaire des entiers, la tactique ignore par nature la théorie de l'égalité. Ainsi, bien qu'elle puisse prouver l'égalité `eintC` ci-dessous, elle est incapable de prouver la variante `cong_eintC`, parce que cette dernière implique une congruence sur l'opération `_ :: nil` qui est étrangère à la théorie de l'arithmétique linéaire.

```
Lemma eintC : forall (z : int), z + 1 = 1 + z.
Lemma cong_eintC : forall (z : int), (z + 1) :: nil = (1 + z) :: nil.
```

Prouver la propriété exprimée par `cong_eintC` nécessite de *combiner* différentes théories, dans ce cas l'arithmétique des entiers et la théorie de l'égalité, comme le font les solveurs SMT. Pourtant, dans ce cas également, le plugin SMTCoq ne nous est d'aucune aide, parce que l'énoncé est formulé en utilisant le type `int` au lieu du type `Z`. Le récent solveur SAT itauto [55], implémenté dans Coq, explore une autre approche de la satisfiabilité modulo théorie formellement vérifiée, et

[55] : Besson (2021), « Itauto : An Extensible Intuitionistic SAT Solver »

organise une coopération entre les tactiques indépendantes `lia`, pour l'arithmétique des entiers, et `congruence`, pour l'égalité. Par conséquent, la tactique `smt` construite au-dessus de ITAUTO peut bénéficier des facilités de pré-traitement de `lia`. Par exemple, dès lors que le pré-traitement de `lia` est correctement configuré pour le type `int`, la tactique `smt` est capable de prouver le lemme `cong_eintC`.

Cependant, les possibilités de pré-traitement de `lia` sont inconnues du reste de la procédure de décision SMT. Ainsi, bien que le premier but ci-dessous soit prouvable par la tactique `smt`, parce que `lia` a été informé de l'égalité booléenne `=` disponible sur le type `int`, la même tactique échoue sur la variante `cong_eintCb`, comportant un symbole non interprété `f` :

```
Lemma eintCb : forall (z : int), (z + 1 = 1 + z) = true.
Lemma cong_eintCb : forall (f : int → int) (z : int),
  (f (z + 1) = f (1 + z)) = true.
```

Il s'avère que, bien que les utilisateurs de l'assistant de preuve COQ disposent d'une variété de tactiques mettant en place un raisonnement automatisé, trouver l'arme appropriée pour s'attaquer à un but donné reste un défi. Il est souvent difficile d'anticiper la compétence exacte des tactiques basées sur un raisonnement automatisé du premier ordre, et d'en interpréter les échecs. Par conséquent, dans le cadre d'efforts de formalisation à grande échelle, les utilisateurs peuvent finir par développer leurs propres outils d'automatisation spécifiques, comme la tactique `list_solve` dans la VERIFIED SOFTWARE TOOLCHAIN [56, 57], qui automatise le raisonnement sur les listes et l'arithmétique, ce qui augmente encore le nombre de tactiques disponibles, souvent de manière redondante. Une forme de pré-traitement générique telle que celle ciblée par TRAKT semble donc idéale pour rendre plus flexibles les outils d'automatisation existants.

[56] : APPEL (2011), « Verified Software Toolchain - (Invited Talk) »

[57] : APPEL *et coll.* (2022), *Verifiable C*

7.1.2 Transformations modulaires de la tactique scope

La tactique `scope` [12] est une combinaison de différentes tactiques ayant pour objectif de ramener les énoncés COQ contenus dans le fragment SMT à la théorie logique manipulée par les solveurs SMT, qui se situe à un plus bas niveau.[2] Le pré-traitement effectué par `scope` est orthogonal et complémentaire par rapport aux fonctionnalités de TRAKT, si bien que les deux outils peuvent travailler ensemble dans le pré-traitement des énoncés SMT. Ici, nous présentons trois des différentes transformations effectuées par `scope` : la génération du principe d'inversion des relations inductives, l'élimination du filtrage et la monomorphisation d'hypothèses.

[12] : BLOT *et coll.* (2023), « Compositional preprocessing for automated reasoning in dependent type theory »

2 : Par exemple, les quantifications sont prénexes et il n'y a pas de types dépendants.

Principe d'inversion des relations inductives Une relation inductive est un type inductif représentant une relation entre des termes, dont le codomaine est souvent `Prop`. Lorsqu'une hypothèse est un témoin de la relation entre des termes, il est possible de déterminer à partir de ces termes les différents constructeurs qui ont pu être utilisés pour construire le témoin. La tactique `inversion` utilise cette propriété pour ajouter des hypothèses au contexte, mais dans le cadre de l'utilisation d'un solveur SMT, il peut être intéressant de conserver le principe d'inversion en tant qu'hypothèse supplémentaire. La transformation associée dans `scope` effectue ce travail.

On considère la relation inductive représentant le graphe de la fonction d'addition entre deux entiers naturels :

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall (n : nat), add O n n
  | addS : forall (n m k : nat), add n m k → add (S n) m (S k).
```

Un appel à cette tactique de transformation ajoute une nouvelle hypothèse au contexte avec le type suivant :

```
forall (n m k : nat), add n m k ↔
  (exists (n' : nat), n = O ∧ m = n' ∧ k = n') ∨
  (exists (n' m' k' : nat),
    add n' m' k' ∧ n = S n' ∧ m = m' ∧ k = S k')
```

Élimination du filtrage Le filtrage par motif, disponible dans un langage de haut niveau, est une construction inconnue d'un solveur SMT. Lorsqu'une hypothèse contient un filtrage, cette transformation le sépare en autant de nouvelles hypothèses que le filtrage contient de cas.

On considère l'accès au n-ième élément d'une liste. Cette fonction peut être écrite de manière totale soit en utilisant une valeur de retour optionnelle pour prendre en compte le cas où n est supérieur à la taille de la liste, soit en utilisant une valeur de retour par défaut. Une hypothèse donnant la définition de la seconde (`nth_default`) à partir de la première (`nth`) :

```
forall (A : Type) (d : A) (l : list A) (n : nat),
  nth_default A d l n =
    match nth l n with
    | Some x ⇒ x
    | None ⇒ d
    end
```

est remplacée par deux hypothèses, chacune concernant un cas du filtrage :

```
H1 : forall A d l n, nth l n = Some x → nth_default d l n = x
H2 : forall A d l n, nth l n = None   → nth_default d l n = d
```

Monomorphisation La plupart des prouveurs automatiques ne gèrent pas le polymorphisme. Or, de nombreux lemmes dans Coq sont polymorphes. Il est donc utile d'implémenter une transformation qui instancie les hypothèses polymorphes avec les types présents dans le but, afin que le solveur puisse les exploiter. L'instanciation des lemmes est effectuée par une heuristique qui sélectionne différents types apparaissant dans le but comme instances potentiellement intéressantes.

Dans le contexte de preuve suivant, en instanciant `H` avec `option Z` et `list unit`, la preuve devient triviale pour un solveur SMT.

```
H : forall (A B : Type) (x1 x2 : A) (y1 y2 : B),
      (x1, y1) = (x2, y2) → x1 = x2 ∧ y1 = y2
──────────────────────────────────────────────
      Z.of_nat n + Z.of_nat n ⩾ Z.of_nat n
```

7.2 Succès de l'outil

Compte tenu du peu d'outils existant dans cet objectif de canonisation des énoncés pour faire le lien entre l'expressivité du langage de Coq et les différentes procédures de décision, la présence de Trakt améliore le niveau d'automatisation de plusieurs tactiques disponibles pour les utilisateurs de Coq. L'usage de Trakt ne présente aucune régression notable par rapport à `zify`, son point de comparaison principal, vis-à-vis de la spécification identifiée en § 5.1. Les notations liées au polymorphisme *ad hoc* telles que celles de MathComp présentées dans l'Exemple 5.2.3 sont prises en charge par Trakt, qui offre une fonctionnalité de niveau comparable. Enfin, le pré-traitement dédié à la logique dans Trakt améliore la situation pour le greffon SMTCoq, objectif initial dans la conception de cet outil. Cette section prend quelques buts en exemple pour démontrer les fonctionnalités effectives de Trakt, puis montre comment le greffon peut être utilisé de manière efficace avec SMTCoq.

7.2.1 Exemples de buts traités

Dans le chapitre précédent, les exemples sont choisis pour illustrer un certain aspect du fonctionnement de Trakt. Ici, prenons un exemple concret et détaillons tout le processus de pré-traitement du point de vue de l'utilisateur.

Le premier but d'exemple est le suivant :

```
forall (x : int), x * x ≥ 0
```

En affichant les projections génériques de MathComp, le but complet est le suivant :

```
forall (x : int),
  @Order.ge int_porderType
    (@GRing.mul int_Ring x x) (@GRing.zero int_Ring) = true
```

Tout d'abord, il faut déterminer le type canonique des entiers pour la tactique d'automatisation considérée ainsi que le type logique idéal pour que cette tactique fonctionne au mieux. Dans ce cas précis, on peut utiliser la tactique `lia`, donc cibler le type `Z` pour les entiers et `Prop` pour la logique par exemple.

Un plongement doit donc être déclaré entre `int` et `Z`. Ensuite, les opérations utilisées dans le but doivent aussi être déclarées. Ici, il s'agit de la multiplication `mulz` dont l'équivalent dans `Z` est `Z.mul`. Les constantes positives du type `int`, en particulier le zéro, sont représentées par des plongements de `nat` dans `int` *via* la fonction `Posz`. Si l'on déclare un plongement de `nat` dans `Z`, alors il suffit de montrer que `Posz` se plonge vers l'identité dans `Z` pour que toutes les constantes du type `int` soient plongeables dans `Z` avec Trakt. La relation d'ordre sur `int` doit aussi être déclarée plongeable vers `Z.ge`. Enfin, les projections génériques de MathComp doivent être déclarées comme des clés de conversion. Voici l'équivalent en utilisant les commandes de Trakt :

```
Trakt Add Embedding int Z Z_of_int Z_to_int π1 π2.
Trakt Add Symbol mulz Z.mul π3.
Trakt Add Embedding nat Z Z.of_nat Z.to_nat π4 π5.
Trakt Add Symbol Posz (@id Z) π6.
Trakt Add Relation (@Order.ge int_porderType) Z.ge π7.
Trakt Add Conversion GRing.mul.
```

```
Trakt Add Conversion GRing.zero.
Trakt Add Conversion Order.ge.
```

Les preuves utilisées ont les types suivants :

```
π1 : forall (x : int), Z_to_int (Z_of_int x) = x
π2 : forall (x' : Z), Z_of_int (Z_to_int x') = x'
π3 : forall (x y : int), Z_of_int (x * y) = Z_of_int x * Z_of_int y
π4 : forall (n : nat), Z.to_nat (Z.of_nat n) = n
π5 : forall (n' : Z), n' ⩾ 0 → Z.of_nat (Z.to_nat n') = n'
π6 : forall (n : nat), Z_of_int (Posz n) = Z.of_nat n
π7 : forall (x y : int), x ⩾ y ↔ Z_of_int x ⩾ Z_of_int y
```

On peut ensuite appeler la tactique `trakt Z Prop` et obtenir le but suivant, prouvable par la tactique `lia`, contrairement au but initial sans pré-traitement :

```
forall (x' : Z), x' * x' ⩾ 0
```

À présent, montrons comment deux autres buts rencontrés plus tôt peuvent être prouvés grâce à un pré-traitement effectué par Trakt.

Exemple 7.2.1
On reprend le but de l'Exemple 5.2.4 :

```
forall (f : int → int) (x : int), f (2 * x) ⩽? f (x + x)
```

Sous réserve de déclarations similaires à l'exemple précédent, un pré-traitement vers `Z` et la logique booléenne avec `trakt Z bool` permet d'obtenir le but suivant, prouvable par une tactique de délégation de preuve à un solveur SMT :

```
forall (f' : Z → Z) (x' : Z), f' (2 * x') ⩽? f' (x' + x') = true
```

Exemple 7.2.2
On reprend à présent le but de l'Exemple 6.2.1, en remplaçant `int` par `nat` :

```
forall (x y : nat), x * y + x = x * (y + 1)
```

Un pré-traitement avec `trakt Z Prop` permet d'obtenir le but suivant, prouvable par `lia` :

```
forall (x' : Z), x' ⩾ 0 → forall (y' : Z), y' ⩾ 0 →
  x' * y' + x' = x' * (y' + 1)
```

Il est visible sur ces exemples que l'association de Trakt avec les tactiques d'automatisation existantes permet effectivement d'étendre leur domaine d'entrée en rendant plus de signatures intelligibles.

7.2.2 Intégration de Trakt à d'autres outils

Trakt peut être utilisé comme pré-traitement unique avant usage d'un prouveur automatique, mais il peut aussi être utilisé avec plusieurs autres outils de pré-traitement. Par exemple, Trakt a été intégré à la suite de transformations `scope` présentée en § 7.1.2, puis utilisé en association avec SMTCoq, donnant le greffon Sniper [12]. Le rôle de Trakt dans ce contexte est de canoniser l'arithmétique et de s'assurer que la logique est exprimée dans `bool` autant que possible, afin

[12] : Blot *et coll.* (2023), « Compositional preprocessing for automated reasoning in dependent type theory »

d'exploiter au maximum la décidabilité des prédicats présents dans le but. Nous étudions ici un exemple de formalisation fortement automatisée par l'usage de Sniper.

Exemple 7.2.3

Cet exemple traite d'une formalisation de propriétés de plusieurs variantes du λ-calcul (comme la normalisation forte), à partir de la bibliothèque MathComp. Dans cette formalisation,[3] on retrouve des encodages profonds de langages à lieurs, dans lesquels on utilise les indices de De Bruijn pour représenter les variables liées. Le prix à payer est la nécessité de prouver des propriétés techniques et peu intéressantes sur la substitution et le décalage de variables. Les énoncés dans un tel contexte contiennent souvent à la fois de l'arithmétique et du raisonnement logique, et les preuves nécessitent de raisonner par induction. Par exemple, les termes du λ-calcul non typé habitent le type inductif suivant :

3 : Nous devons cet exemple à Kazuhiko Sakaguchi.

```
Inductive term : Type :=
  | var of nat
  | app of term * term
  | abs of term.
```

avec les fonctions de décalage et de substitution suivantes :

```
Fixpoint shift d c t : term :=
  match t with
  | var n ⇒ var (if c ⩽ n then n + d else n)
  | app t1 t2 ⇒ app (shift d c t1) (shift d c t2)
  | abs t1 ⇒ abs (shift d c.+1 t1)
  end.
```

Le terme `shift d c t` est le terme `t` dans lequel on a décalé de `d` crans les variables avec un indice au-dessus du seuil `c`.

```
Notation substv ts m n :=
  (shift n 0 (nth (var (m - n - size ts)) ts (m - n))).
Fixpoint subst n ts t : term :=
  match t with
  | var m ⇒ if n ⩽ m then substv ts m n else m
  | app t1 t2 ⇒ app (subst n ts t1) (subst n ts t2)
  | abs t' ⇒ abs (subst n.+1 ts t')
  end.
```

Le terme `subst n ts t` est le terme `t` dans lequel on a remplacé les variables avec un indice au-dessus d'un seuil `n` par des termes contenus dans la liste `ts`.

On note que ces définitions utilisent l'addition, la soustraction et la comparaison sur les entiers naturels définies dans MathComp. En les ajoutant à la base de données de Trakt, puis en effectuant une induction sur les termes du calcul suivie d'un appel à `snipe`, la tactique principale du projet Sniper, on peut prouver automatiquement un certain nombre de propriétés sur ce λ-calcul.

```
Lemma shift_add d d' c c' t :
  c ⩽? c' → c' ⩽? c + d →
    shift d' c' (shift d c t) = shift (d' + d) c t.
Proof. revert d d' c c'; induction t; snipe. Qed.

Lemma shift_shift_distr d c d' c' t :
  c' ⩽? c →
    shift d' c' (shift d c t) = shift d (d' + c) (shift d' c' t).
Proof. revert d d' c c'; induction t; snipe. Qed.
```

7.3 Voies d'amélioration

Bien que Trakt trouve une utilité dans l'écosystème des outils de pré-traitement pour Coq, l'outil demeure limité sur certains aspects. Cette section donne un aperçu de ces limites pour justifier un travail ultérieur.

7.3.1 Polymorphisme et types dépendants

Malgré les différents efforts pour que la traduction gagne en généralité, en tant que point de départ de cette thèse, Trakt est conçu sur le modèle de `zify`. De ce fait, le greffon hérite des divers aspects *ad hoc* de ce dernier, étant destiné à pré-traiter l'arithmétique prouvable par `lia`. En particulier, les plongements suivent le modèle des classes de types de `zify`. Trakt y apporte certes un certain niveau de flexibilité du fait que le stockage des termes soit externe (au niveau méta) contrairement aux instances de classes de types qui doivent se conformer strictement à un type Coq, mais par conception, la classe de termes pouvant faire l'objet de déclarations est proche de celle de `zify`.

En effet, les plongements de types définis en § 6.1.1 concernent des types simples exprimés en un seul terme, ce qui peut être limitant dans le cas où l'on voudrait déclarer des plongements polymorphes ou à base de types dépendants. On peut citer l'exemple des nombres ordinaux, entiers naturels bornés toujours partiellement plongeables dans `nat` ou un type entier plus grand comme `Z`, la condition de plongement étant le respect de la borne supérieure de l'ordinal source.

Un autre exemple intéressant est la canonisation dans le cadre de l'utilisation de types conteneurs sur des entiers. Le but suivant :

```
forall (l : list int), sum_int l ⩾ 0
```

pourrait faire l'objet d'un plongement vers `Z` pour ce qui relève de l'arithmétique, sans toucher le reste. En effet, si $\texttt{sum}_{\texttt{int}}$ est défini comme `fold_left (+) 0 l`, alors on peut plonger l'addition et le zéro vers ceux de `Z` pour obtenir la somme cible. Cependant, ce type de plongement avec utilisation de types polymorphes n'est pas permis par Trakt.

Un second problème est l'impossibilité de ne pas traiter une valeur. En effet, dans des buts impliquant des types dépendants que l'on souhaite laisser tels quels dans le but de sortie, certaines valeurs peuvent être utilisées comme arguments ne devant pas changer de type lors de la traduction, bien qu'elles soient d'un type plongeable. On peut citer le cas des vecteurs de bits dont la taille entière doit rester dans le type initialement utilisé pour l'encoder, car un pré-traitement plongeant cette valeur dans un autre type introduit des risques d'avoir un but mal typé en sortie. Trakt ne permet pas de gérer ces cas particuliers et effectue toujours les plongements quand ils sont possibles.

7.3.2 Architecture du pré-traitement

La structure de l'algorithme de pré-traitement limite également les possibilités de Trakt. En effet, l'outil a été conçu de façon incrémentale. D'abord construit à partir du principe de composition des preuves de morphisme pour pré-traiter les sous-termes appartenant à une théorie dont on sait plonger la signature, les différentes fonctionnalités supplémentaires se sont greffées à la fonction récursive principale de l'algorithme, ajoutant des arguments pour représenter diverses

informations à garder en mémoire dans les appels récursifs. Cette approche a l'avantage d'être pragmatique et de permettre d'avoir rapidement un outil fonctionnel et utilisable dans un contexte réel, mais elle a aussi quelques inconvénients.

Premièrement, en faisant abstraction de la théorie de la congruence qui peut toujours être traitée à la volée, Trakt ne permet de pré-traiter qu'une seule théorie à la fois. Si le but contient un mélange de théories, il faut appeler plusieurs fois Trakt avec un pré-traitement pour une théorie à chaque fois.

Deuxièmement, le passage de `bool` à `Prop` se fait en deux phases. En effet, pour faire passer un sous-terme booléen dans `Prop`, il faut que le sous-terme `b` soit plongé dans `Prop` avec une égalité : `b = true`, `false = b`, *etc.* Si le sous-terme est sous un prédicat non interprété de type `bool → Prop`, alors il ne sera pas possible de l'exprimer dans `Prop` dans le but de sortie. Dans l'état actuel de Trakt, cette information n'est pas suivie au cours de la traduction. Lors de la traduction d'une valeur booléenne, il n'est donc pas possible de savoir s'il est possible de la remplacer par son équivalent dans `Prop` . Même si l'égalité booléenne `Nat.eqb` sur les entiers naturels est plongeable vers l'égalité dans `Prop` sur le type `Z`, en arrivant au nœud de la relation, il ne sera pas possible de savoir si le plongement peut être effectué. Ainsi, pour faire un passage de `bool` à `Prop`, une phase spécialisée à la logique est d'abord exécutée, avec la capacité de regarder au-dessus d'un terme pour voir s'il est plongé dans `Prop` d'une manière qui autorise la réécriture. Une fois que la première passe est faite, le but est exprimé dans `Prop` et les plongements restants sont possibles. On passera donc de `Nat.eqb` à `@eq nat` puis à `@eq Z`, nécessitant deux déclarations utilisateur au lieu d'une.

Enfin, les preuves de réécriture prennent la forme d'une égalité entre un sous-terme avant et après réécriture. Leur composition est faite par transitivité, donc toutes les preuves d'égalité sont étendues pour avoir le même contexte et pouvoir être composées. Par conséquent, on fait une répétition du contexte avec une légère variation à chaque réécriture, ce qui donne à la preuve globale sur un sous-terme une complexité quadratique en espace en fonction du nombre de réécritures à effectuer dans ce sous-terme.

En définitive, les failles identifiées dans cette section peuvent être corrigées avec des solutions *ad hoc* de méta-programmation, ce qui permet de conserver le prototype actuel. Alternativement, riches des enseignements venant de la conception de Trakt, nous pouvons également reprendre la problématique du transfert de preuve, présentée en § 3.3, et concevoir un nouvel outil de pré-traitement avec une approche plus générale permettant d'englober les cas qui se situent à la frontière de ce que peut traiter Trakt. C'est la solution retenue pour le deuxième prototype mis au point pendant cette thèse, présenté dans la partie suivante.

Trocq :
Transfert de preuve par paramétricité

Introduction

Les limites identifiées en faisant le bilan de Trakt montrent un besoin de généralisation par rapport à l'approche *ad hoc* héritée de `zify`. Plutôt que de partir de relations concrètes entre les termes[4] et construire autour un algorithme qui exploite les fonctions de plongement pour créer le but associé, on peut abstraire ces relations, d'abord construire l'algorithme qui les exploite, et ensuite leur donner du contenu. On dit alors que deux types A et B sont reliés s'il existe une relation R de type $A \to B \to \square$. On étudie ensuite la possibilité de propager ces relations par induction sur le typage.

4 : Dans Trakt, il s'agit par exemple des bijections ou plongements partiels définis en § 6.1.1 pour relier des types, ou des preuves de morphisme vis-à-vis de la fonction de plongement définies en § 6.1.3 pour relier des opérations.

Cette étude est le sujet d'une lignée de travaux sur le concept de *paramétricité* ou *relations logiques* [58], dont l'objectif a d'abord été de dériver des propriétés sur les termes à partir de leur type dans des λ-calculs polymorphes. En donnant aux types une interprétation relationnelle, on peut obtenir des théorèmes dits « gratuits ». En effet, si deux termes a et b sont reliés, alors on peut trouver une relation entre deux termes $C[a]$ et $C'[b]$, où C est un contexte et C' le contexte associé. Ceci permet par exemple d'affirmer qu'une relation entre deux types A et B peut être étendue à des listes de valeurs dans A et B, ce qui correspond concrètement à des opérations comme `List.for_all2` en OCaml. L'extension de la paramétricité aux types dépendants puis à la théorie des types de Coq permet d'internaliser les témoins de paramétricité, c'est-à-dire les preuves que deux termes sont liés par une relation. Dans un tel contexte, les théorèmes gratuits deviennent réellement des preuves Coq, plutôt que des propriétés au niveau méta comme précédemment.

[58] : Mitchell (1986), « Representation Independence and Data Abstraction »

Une traduction de paramétricité [59] est une fonction qui prend un terme à traduire (dans notre cas, le but) et produit deux termes en sortie, un terme traduit (le but associé) ainsi qu'un témoin qui prouve que le terme initial est lié au terme traduit. Dans le contexte vide et sur des termes clos, la traduction de paramétricité dans Coq n'est autre qu'une identité profonde sur le but, obtenue en le traversant par induction sur la syntaxe. Pour traduire des constantes, on ajoute dans le contexte de la traduction des relations entre chaque constante source c et une constante cible associée c'. Ainsi, pour traduire l'addition dans `nat` vers l'addition dans `bin_nat`, on fournit une relation R entre les deux types ainsi qu'une relation entre les deux additions, c'est-à-dire une preuve que l'addition de termes liés par R donne des termes liés par R. Tout but contenant des valeurs et additions dans `nat` peut alors être traduit vers un but associé mentionnant ces termes dans `bin_nat`.

[59] : Boulier *et coll.* (2017), « The next 700 syntactical models of type theory »

Dans l'objectif d'effectuer un transfert de preuve, on doit pouvoir extraire une fonction du témoin de paramétricité. On souhaite alors enrichir la relation propagée à travers la syntaxe pendant la traduction de paramétricité, afin de profiter du cadre général que cette technique apporte tout en obtenant plus d'information dans le témoin de paramétricité obtenu en sortie de la traduction. Parmi tous les enrichissements possibles, les enrichissements asymétriques ne passent pas à travers toutes les constructions, pour des raisons de polarité, notamment pour le Π-type. Un enrichissement stable est possible à travers la paramétricité dite *univalente* [14], au prix de l'ajout du principe d'*univalence* à Coq, nécessaire pour traduire les univers. Le témoin obtenu est alors dense et riche en information, et contient en particulier la fonction nécessaire au transfert de preuve.

[14] : Tabareau *et coll.* (2021), « The marriage of univalence and parametricity »

Effectuer un transfert de preuve en faisant une traduction de paramétricité est donc possible. Une telle traduction prend en charge toutes les constructions du langage et permet de pré-traiter de nombreux buts. Cependant, dans ce cadre, les relations sur les constantes ajoutées au contexte par l'utilisateur avant la traduction requièrent un témoin univalent qui, en raison de la richesse de ce témoin, n'est pas toujours facilement prouvable. De plus, le principe d'univalence vient dans la version standard de Coq sous la forme d'un axiome qu'il est regrettable de devoir utiliser dans le cadre du pré-traitement d'un but qui pourrait être effectué sans axiome si l'utilisateur le faisait à la main, en faisant des manipulations de l'ordre de ce qui est fait automatiquement par Trakt. Cette limite est la motivation majeure pour la mise au point de Trocq [15], une implémentation d'un nouveau cadre de paramétricité plus flexible et modulaire, afin de conserver la généralité de la paramétricité tout en faisant un usage parcimonieux des axiomes, idéalement réduit aux cas où ils sont strictement nécessaires pour traiter le terme d'entrée.

[15] : Cohen *et coll.* (2024), « Trocq : Proof Transfer for Free, With or Without Univalence »

Cette partie présente les concepts théoriques de Trocq. Dans un premier temps, nous présentons de manière plus détaillée le contexte dans lequel s'inscrit ce nouveau greffon,[5] c'est-à-dire le chemin depuis les origines de la paramétricité jusqu'à la paramétricité univalente, qui constitue la base du travail effectué sur Trocq, au même titre que `zify` pour Trakt (§ 8). Dans un second temps, nous étudions la décomposition du témoin de paramétricité univalente pour en exposer les informations, ainsi que leur recomposition dans une *hiérarchie* de témoins de paramétricité et la construction d'un cadre unique qui fait fonctionner tous ces témoins ensemble (§ 9). Enfin, nous formulons cette relation comme un programme logique, afin d'exposer au maximum les informations requises sur le contexte, dans le but de l'implémenter en tant que traduction dans une tactique Coq par la suite (§ 10).

5 : L'implémentation est disponible dans le dépôt :

`https://github.com/ecranceMERCE/trocq`

La paramétricité en théorie des types dépendants

8

Les limites de TRAKT poussent à chercher une approche plus générale pour effectuer du transfert de preuve en COQ. Ce chapitre présente la *paramétricité*, un concept de la théorie des langages de programmation qui donne aux types du λ-calcul une interprétation relationnelle, permettant de repenser les liens existant entre les termes d'entrée et de sortie dans le cadre d'un outil de pré-traitement. Nous y définissons le concept d'origine et son extension au λ-calcul de COQ (§ 8.1). Ensuite, nous présentons la *paramétricité univalente*, une version enrichie permettant d'effectuer du transfert de preuve généralisé dans COQ avec l'axiome d'univalence, et nous montrons en quoi cette technique est intéressante vis-à-vis de la spécification que nous avons donnée pour TRAKT initialement (§ 8.2).

8.1 Motivation et définition . . . 79
8.1.1 Typage et propriétés des λ-termes 79
8.1.2 Traduction de paramétricité brute 80
8.1.3 Limites de la traduction brute . 81
8.2 La paramétricité univalente . 82
8.2.1 Enrichissement des témoins de paramétricité 82
8.2.2 Équivalence de types et univalence 83
8.2.3 Traduction de paramétricité univalente 85
8.2.4 Omniprésence de l'axiome d'univalence 87

8.1 Motivation et définition

La notion de *paramétricité* remonte à l'apparition du polymorphisme dans le λ-calcul. À l'origine un outil pour raisonner sur les propriétés des fonctions polymorphes [13], elle est à présent utilisée pour effectuer des traductions de termes COQ. Cette section présente ces deux aspects.

[13] : REYNOLDS (1983), « Types, Abstraction and Parametric Polymorphism »

8.1.1 Typage et propriétés des λ-termes

Dans le λ-calcul simplement typé, les termes peuvent avoir des types fonctionnels simples ou « types flèches », gouvernés par la règle de typage $\text{LAM}_{\to}$:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B}$$

Toutes les fonctions y sont *monomorphes* et ont un domaine A et un codomaine B concrets. Ainsi, en définissant un type $\mathbb{B}$ et des opérations primitives $\neg$ et $\wedge$ pour représenter les booléens et les connecteurs logiques de négation et de conjonction, la fonction représentant la porte logique NAND a un type unique :

$$\Gamma \vdash \lambda b_1\, b_2.\, \neg\,(b_1 \wedge b_2) : \mathbb{B} \to \mathbb{B} \to \mathbb{B}$$

Or, dans le λ-calcul à la CURRY,[1] il existe des fonctions pour lesquelles plusieurs types sont possibles. Un exemple primitif est la fonction identité, $\lambda x.\, x$. En effet, comme elle peut être appliquée de façon valide à n'importe quel terme du langage, cette fonction possède une infinité de types, chacun dépendant du type que l'on choisit pour la variable liée : on dit que la fonction est *polymorphe*. Quand ce type peut être abstrait et représenté par une variable, on parle de polymorphisme *paramétrique*. Le système F permet de représenter ces variables à l'aide d'un lieur supplémentaire, Λ. Ainsi, l'identité peut être typée de manière unique :

1 : Il s'agit de la présentation dans laquelle les fonctions ne possèdent pas d'annotation de typage.

$$\Gamma \vdash \lambda x.\, x : \Lambda\alpha.\, \alpha \to \alpha$$

L'observation principale de la *paramétricité* est que le corps d'une fonction polymorphe n'exploite jamais le type de la variable liée, c'est-à-dire le *paramètre de*

type. Ceci permet d'extraire des propriétés sur le comportement des fonctions polymorphes à partir de leur type, c'est-à-dire sans avoir besoin d'inspecter le corps de la fonction. Par exemple, le type polymorphe de la fonction identité permet de déterminer de façon unique son implémentation. En effet, la fonction reçoit en entrée une valeur d'un type dont elle ne peut exploiter la structure, car il est polymorphe, et doit renvoyer une valeur du même type. Il n'y a donc aucune autre implémentation possible que celle qui renvoie la valeur d'entrée inchangée. Ces propriétés disponibles directement au niveau du type sont qualifiées de « théorèmes gratuits » dans la littérature [60].

[60] : Wadler (1989), « Theorems for Free ! »

8.1.2 Traduction de paramétricité brute

La théorie des types permet de représenter les propriétés de paramétricité et leurs preuves de manière *interne*, c'est-à-dire dans le même calcul que celui des termes à l'étude. Ainsi, les résultats précédemment obtenus par une analyse manuelle au niveau méta peuvent dans ce cadre être donnés automatiquement par une *traduction syntaxique* du calcul vers lui-même. De telles traductions peuvent prendre en compte les types dépendants [61], les types inductifs [62], ainsi que le Calcul des Constructions Inductives[2] tout entier, y compris sa sorte impredicative [63].

[61] : Bernardy *et coll.* (2011), « Realizability and Parametricity in Pure Type Systems »

[62] : Bernardy *et coll.* (2012), « Proofs for free - Parametricity for dependent types »

Les travaux de Bernardy *et coll.*, Keller et Lasson permettent de définir ce que l'on appellera par la suite la traduction de *paramétricité brute*, qui définit essentiellement une relation logique $[\![T]\!]$ pour tout type T, par induction sur la syntaxe :

$$
\begin{aligned}
[\![\langle\rangle]\!] &:= \langle\rangle \\
[\![\Gamma, x : A]\!] &:= [\![\Gamma]\!], x : A, x' : A', x_R : [\![A]\!]\, x\, x' \\
\\
[\![\square_i]\!] &:= \lambda A\, A'.\, A \to A' \to \square_i \\
[\![x]\!] &:= x_R \\
[\![\Pi x : A.\, B]\!] &:= \lambda f\, f'.\, \Pi (x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\, [\![B]\!]\, (f\, x)\, (f'\, x') \\
[\![\lambda x : A.\, t]\!] &:= \lambda (x : A)(x' : A')(x_R : [\![A]\!]\, x\, x').\, [\![t]\!] \\
[\![t\, u]\!] &:= [\![t]\!]\, u\, u'\, [\![u]\!]
\end{aligned}
$$

Fig. 8.1 : Traduction de paramétricité brute pour CC_ω.

Cette présentation utilise la convention classique selon laquelle t' est le terme obtenu à partir d'un terme t en y remplaçant chaque variable x par une variable fraîche x'. Une variable x est traduite par une variable x_R dont le nom est frais. Cette traduction préserve le typage en vertu du résultat suivant :

Théorème 8.1.1 (Théorème d'abstraction)
Si $\Gamma \vdash t : T$, alors $[\![\Gamma]\!] \vdash t : T$, $[\![\Gamma]\!] \vdash t' : T'$ et $[\![\Gamma]\!] \vdash [\![t]\!] : [\![T]\!]\, t\, t'$.

Démonstration. Voir par exemple [63]. ■

Cette traduction génère précisément les énoncés attendus pour une famille de types ou un programme paramétrique. Par exemple, la traduction d'un produit dépendant donnée ci-dessus est une relation qui lie deux fonctions f et f' si

2 : Il s'agit du Calcul des Constructions, défini en § 2.1.3, muni des types inductifs définis en § 2.2.2.

[63] : Keller *et coll.* (2012), « Parametricity in an Impredicative Sort »

elles produisent des termes reliés lorsqu'on leur donne des termes reliés en entrée.

8.1.3 Limites de la traduction brute

Dans notre contexte, la paramétricité brute présente deux limites principales que sont la non-préservation des égalités définitionnelles et la faiblesse des témoins de paramétricité.

Dans le premier cas, Tabareau *et coll.* prennent l'exemple d'une preuve du faux[3] à partir d'une égalité contradictoire sur les entiers naturels :

3 : C'est-à-dire, un type inductif vide appelé `False`.

```
forall (n : nat), 0 = S n → False
```

On peut prouver cette propriété en définissant une fonction `P : nat → Type` qui vaut `0 = 0` en `0` et `False` sinon. Ensuite, on introduit `n` et l'égalité `e`, et on doit prouver `False`. La preuve s'obtient alors par induction dépendante sur `e` : si l'on peut prouver une propriété `Q 0 refl`, alors on a `Q (S n) e`. En posant `Q n _ := P n`, on doit alors prouver `P 0` pour obtenir une preuve de `P (S n)`. Comme `P 0` est défini comme `0 = 0`, il suffit de fournir `refl` pour cette preuve, puis on obtient la preuve de `P (S n)`, c'est-à-dire `False`. Or, dans cette dernière étape, nous exploitons le fait que la valeur `P` est définie à l'aide du principe d'induction sur `nat`, lui-même défini à l'aide d'un filtrage par motif sur la valeur de type `nat` qui lui est fournie. Ainsi, dès que le constructeur de tête de l'argument de `P` est connu, on peut effectuer une étape de ι-réduction[4] pour sélectionner la bonne branche du filtrage. Cette étape de conversion est nécessaire pour conclure que le terme final est bien typé. On dit en effet que `P 0` et `P (S n)` sont *définitionnellement* égaux à (respectivement) `0 = 0` et `False`.

4 : Il s'agit de la règle de réduction qui gère les filtrages par motif.

Si l'on utilise la paramétricité brute pour traduire le terme de preuve final, on va associer chaque constante sur `nat` à une constante sur un type associé à `nat`, par exemple `bin_nat`. En particulier, le principe d'induction de `nat` va devenir un principe d'induction sur `bin_nat` dont la structure du type est la même, c'est-à-dire que l'induction se fait de successeur en successeur comme sur le type `nat`, alors que les constructeurs de `bin_nat` encodent des valeurs binaires. Par conséquent, le principe d'induction que l'on associe à `nat_rect` n'est, contrairement à ce dernier, pas défini par induction sur son argument. Ainsi, le terme de preuve traduit ne pourra pas exploiter la ι-réduction dans la vérification de typage. En effet, Coq devra montrer que `refl` a le type `P' b0`, ce qui est impossible si l'on ne sait pas que ce dernier terme est en réalité `b0 = b0`. L'utilisation de la conversion dans le typage est puissante, mais pénalise toutes les traductions qui ne préservent pas les égalités définitionnelles, comme c'est le cas de la paramétricité brute.

La seconde limite de cette traduction est la faiblesse des témoins de paramétricité. En effet, bien que la traduction brute soit en mesure de générer le but souhaité après pré-traitement,[5] dans ce cadre, les témoins de paramétricité liant les termes d'entrée et de sortie sont toujours des relations ou témoins de relation. Générer le but associé est une partie du travail nécessaire pour le transfert de preuve, mais savoir que les deux buts sont liés par une relation n'est pas suffisant pour remplacer le premier par le second. En effet, pour effectuer la réécriture du but, il faut une fonction de l'un vers l'autre, ce que ne fournit pas la traduction de paramétricité brute.

5 : Pourvu que le contexte de paramétricité contienne les différentes relations décrivant les substitutions souhaitées par l'utilisateur.

8.2 La paramétricité univalente

Pour surmonter les limites de la paramétricité brute, une solution est d'enrichir les témoins de paramétricité afin qu'il soit toujours possible d'obtenir une implication entre le but de sortie et le but d'entrée après traduction. Il s'agit de la promesse de la *paramétricité univalente* [14], traduction de paramétricité plus puissante basée sur l'exploitation de l'axiome d'*univalence* [64] et de déclarations d'équivalences de types par l'utilisateur. Dans cette section, nous détaillons l'enrichissement progressif du type des témoins de paramétricité, nous explicitons la définition de l'équivalence utilisée dans ce contexte, puis nous présentons la traduction de paramétricité univalente.

[14] : Tabareau *et coll.* (2021), « The marriage of univalence and parametricity »

[64] : Univalent Foundations Program (2013), *Homotopy Type Theory : Univalent Foundations of Mathematics*

8.2.1 Enrichissement des témoins de paramétricité

Pour définir une traduction de paramétricité, on définit d'abord la traduction de l'univers, car elle donne la structure de tous les témoins de paramétricité qui relient des types. On définit ensuite le reste des cas de la traduction qui étendent des preuves sur les sous-termes à de nouvelles constructions. Ainsi, dans la traduction de paramétricité brute, le témoin permettant de lier deux types A et B est une relation $A \to B \to \square$, et la traduction du produit dépendant correspond à une preuve de la propriété suivante, qui indique qu'à partir de témoins sur A et A' et B et B', on peut construire un témoin sur les produits dépendants $\Pi a : A.\, B\, a$ et $\Pi a' : A'.\, B'\, a'$:

$$\begin{aligned}
&\Pi(A\, A' : \square)(A_R : A \to A' \to \square).\\
&\Pi(B : A \to \square)(B' : A' \to \square).\\
&\quad (\Pi(a : A)(a' : A').\, A_R\, a\, a' \to (B\, a \to B'\, a' \to \square)) \to\\
&\qquad (\Pi a : A.\, B\, a) \to (\Pi a' : A'.\, B'\, a') \to \square
\end{aligned}$$

Si l'on change la définition du témoin sur les univers pour enrichir la traduction de paramétricité, les autres cas de la traduction disposent de plus d'informations sur les sous-termes, mais doivent aussi en propager plus. Ainsi, si l'on ajoute une fonction au témoin de paramétricité sur les univers en le définissant comme une paire $(A \to B \to \square) \times (B \to A)$, la propriété à prouver pour un type flèche[6] devient alors la suivante :

6 : On étudie d'abord le type flèche, puis le produit dépendant, pour exposer deux problèmes différents apparaissant avec l'enrichissement du témoin par une fonction.

$$\begin{aligned}
&\Pi(A\, A' : \square)(A_R : (A \to A' \to \square) \times (A' \to A)).\\
&\Pi(B\, B' : \square)(B_R : (B \to B' \to \square) \times (B' \to B)).\\
&\quad ((A \to B) \to (A' \to B') \to \square) \times ((A' \to B') \to (A \to B))
\end{aligned}$$

La partie gauche de la paire à construire sur le type flèche, la relation, peut être obtenue de la même façon que pour la traduction brute. La partie droite, en revanche, revient à construire une valeur dans $A \to B$ à partir des trois valeurs suivantes :

$$f' : A' \to B' \qquad \psi_A : A' \to A \qquad \psi_B : B' \to B$$

Or, on voit que la contravariance du domaine du type flèche nous empêche d'effectuer cette preuve. En effet, cette preuve serait faisable si la fonction ψ_A était dans l'autre sens, c'est-à-dire une fonction ϕ_A de type $A \to A'$. On construirait la fonction attendue en prenant une valeur dans A, puis en lui appliquant tour à tour ϕ_A, f', puis ψ_B, pour obtenir une valeur dans B. Le fait de casser

la symétrie en n'introduisant une fonction que dans un seul sens nous oblige à orienter les témoins de paramétricité et avoir une traduction qui gère deux types de témoins.

Cette situation est acceptable, bien qu'elle complexifie la traduction de paramétricité. En revanche, l'enrichissement par une fonction pose un autre problème lié aux types dépendants.[7] L'exemple précédent traite du type flèche, mais la preuve à effectuer pour un produit dépendant est la suivante :

7 : Ce problème existerait également en rendant le témoin symétrique en enrichissant le témoin brut avec une fonction dans chaque sens.

$$
\begin{aligned}
&\Pi(A\,A' : \square)(A_R : (A \to A' \to \square) \times (A' \to A)).\\
&\Pi(B : A \to \square)(B' : A' \to \square).\\
&\quad (\Pi(a : A)(a' : A').\, A_R.1\,a\,a' \to ((B\,a \to B'\,a' \to \square) \times (B'\,a' \to B\,a)))\\
&\qquad \to ((\Pi a : A.\, B\,a) \to (\Pi a' : A'.\, B'\,a') \to \square)\\
&\qquad\quad \times ((\Pi a' : A'.\, B'\,a') \to (\Pi a : A.\, B\,a))
\end{aligned}
$$

On s'intéresse de nouveau à la partie droite, qui revient à construire une valeur dans $\Pi a : A.\, B\,a$ à partir des trois valeurs suivantes :

$$
\begin{aligned}
&f' : \Pi a' : A'.\, B'\,a'\\
&\psi_A : A' \to A\\
&\psi_B : \Pi(a : A)(a' : A').\, A_R.1\,a\,a' \to (B'\,a' \to B\,a)
\end{aligned}
$$

La contravariance donne certes le mauvais type pour ψ_A, mais ici, on s'intéresse au type de ψ_B, qui dans le cas d'un produit dépendant, devient dépendant de deux valeurs a et a' ainsi qu'un témoin de paramétricité entre elles. On doit construire une valeur de type $\Pi a : A.\, B\,a$, c'est à dire une valeur de type $B\,a$ en introduisant une valeur $a : A$ dans le contexte. Or, même en disposant d'une valeur de type A' à fournir comme second argument de ψ_B, on ne pourrait créer de témoin de paramétricité entre a et cette valeur, puisque a n'est qu'une variable locale sur laquelle on ne dispose d'aucune autre information. On a donc besoin d'enrichir davantage le témoin de paramétricité, par exemple en liant la relation à la fonction. Le témoin (ou plutôt l'un des témoins orientés) devient une paire dépendante à trois valeurs :

$$\Sigma(R : A \to B \to \square)(\phi : A \to B).\, \Pi(a : A).\, R\,a\,(\phi\,a)$$

Ce nouveau témoin fournit alors un moyen de construire le témoin manquant pour instancier ψ_B ci-dessus et créer la fonction sur les produits dépendants. Cependant, comme le témoin de paramétricité a été enrichi, on doit également prouver la dernière propriété sur le produit dépendant, qui elle aussi demande davantage d'informations. Le témoin de paramétricité univalente peut être vu comme l'aboutissement d'une itération sur ce problème. On obtient un témoin à la fois symétrique et stable par la traduction, c'est-à-dire qui passe à travers toutes les constructions sans changer de nature.

8.2.2 Équivalence de types et univalence

Au cœur de la paramétricité univalente se trouve le principe d'univalence, défini à l'aide de l'équivalence de types, une notion très répandue avec de nombreuses définitions existantes. Ici, nous explicitons les définitions choisies par Tabareau *et coll.*, qui servent de base à notre travail : isomorphisme, équivalence, univalence. Ce sont des définitions classiques que l'on peut retrouver dans le livre *Homotopy Type Theory* [64].

[64] : Univalent Foundations Program (2013), *Homotopy Type Theory : Univalent Foundations of Mathematics*

Définition 8.2.1 (Isomorphisme)
Une fonction $\phi : A \to B$ est un *isomorphisme*, noté $\mathrm{IsIso}(\phi)$, si elle possède un inverse à gauche et à droite :

$$\mathrm{IsIso}(\phi) \quad :\equiv \quad \Sigma(\psi : B \to A).\ (\psi \circ \phi \doteq \mathrm{id}) \times (\phi \circ \psi \doteq \mathrm{id})$$

On note que le plongement de types de Trakt, introduit en § 6.1.1, est défini à l'aide d'un isomorphisme dans le cas d'un plongement total.

Définition 8.2.2 (Équivalence)
L'ajout de la propriété dite d'*adjonction* à un isomorphisme $\phi : A \to B$ permet d'obtenir une *équivalence*, notée $\mathrm{IsEquiv}(\phi)$:

$$\begin{aligned}\mathrm{IsEquiv}(\phi) \quad :\equiv \quad & \Sigma(\psi : B \to A) \\ & (\mathrm{sec} : \psi \circ \phi \doteq \mathrm{id}) \\ & (\mathrm{ret} : \phi \circ \psi \doteq \mathrm{id}). \\ & \quad \mathrm{ap}\,\phi \circ \mathrm{sec} \doteq \mathrm{ret} \circ \phi\end{aligned}$$

où $\mathrm{ap}\ f\ p : f\ x = f\ y$ pour $p : x = y$.

La propriété d'adjonction relie ensemble les preuves de *section* et de *rétraction* en montrant qu'une preuve de $\phi \circ \psi \circ \phi \doteq \phi$ peut être obtenue en retirant dans le membre de gauche soit la composition extérieure $\phi \circ \psi$ à l'aide de la preuve de rétraction, soit en retirant la composition intérieure $\psi \circ \phi$ à l'aide de la preuve de section.

Parfois, on sait définir la fonction inverse et les preuves de section et de rétraction, mais l'adjonction est difficile à prouver. Dans ce cas, il existe une méthode classique de HoTT pour obtenir une équivalence à partir d'un isomorphisme.

Lemme 8.2.3
Un isomorphisme est une équivalence.

Définition 8.2.4 (Équivalence de types)
On dit que deux types A et B sont équivalents, noté $A \simeq B$, lorsqu'il existe une équivalence $\phi : A \to B$:

$$A \simeq B \quad :\equiv \quad \Sigma\phi : A \to B.\ \mathrm{IsEquiv}(\phi)$$

Une équivalence de types $e : A \simeq B$ contient donc deux *fonctions de transport* que l'on peut aussi noter $\uparrow_e : A \to B$ et $\downarrow_e : B \to A$. Elles peuvent être utilisées pour effectuer du transfert de preuve du type A vers le type B, en utilisant $\uparrow_e$ pour les occurrences covariantes et $\downarrow_e$ pour les occurrences contravariantes.[8]

Le *principe d'univalence* affirme que les types équivalents sont indiscernables.

Définition 8.2.5 (Principe d'univalence)
Pour tous types A et B, la fonction canonique de type $A = B \to A \simeq B$ est une équivalence.

Dans les variantes de CC_ω, le principe d'univalence peut être postulé comme axiome, sans contenu calculatoire explicite, comme c'est le cas par exemple dans la bibliothèque HoTT [65] pour l'assistant de preuve Coq. Certaines variantes plus

8 : Ce fonctionnement n'est pas sans rappeler celui de Trakt, où le rôle des fonctions de transport est joué par les fonctions de plongement.

[65] : Bauer *et coll.* (2017), « The HoTT library : a formalization of homotopy type theory in Coq »

récentes de la théorie des types dépendants intègrent un principe d'univalence non axiomatique et sont utilisées pour implémenter des assistants de preuve expérimentaux tels que CUBICAL AGDA. Dans les deux cas, le principe d'univalence constitue un puissant principe de transfert de preuve de $\square$ à $\square$, car pour deux types A et B tels que $A \simeq B$, et tout $P : \square \to \square$, on obtient que $P\,A \simeq P\,B$ comme corollaire direct de l'univalence.[9] Concrètement, on obtient $P\,B$ à partir de $P\,A$ en plaçant de manière appropriée les fonctions de transport fournies par les preuves d'équivalence, un processus de transfert typiquement utile dans le contexte de l'ingénierie de la preuve [66].

9 : Pour $e : A \simeq B$ et u une preuve du principe d'univalence appliqué à A et B, on a :

$$\uparrow_u (\mathrm{ap}\, P\, (\downarrow_u e)) : P\,A \simeq P\,B.$$

[66] : RINGER *et coll.* (2021), « Proof repair across type equivalences »

8.2.3 Traduction de paramétricité univalente

L'observation majeure de la paramétricité univalente est qu'il est possible d'enrichir les témoins de paramétricité tout en préservant le théorème d'abstraction (8.1.1). En effet, tandis qu'un témoin de paramétricité brute lie deux types par une relation arbitraire, la paramétricité univalente requiert que la relation soit une équivalence entre ces types. Cet enrichissement nécessite toutefois une conception minutieuse lors de la traduction des univers.

DÉFINITION 8.2.6
L'interprétation relationnelle de l'univers dans la paramétricité univalente est la suivante :

$$\begin{aligned}\mathrm{Param}_i\,(A\,B : \square_i) \quad :\equiv \quad & \Sigma(R : A \to B \to \square_i)(e : A \simeq B).\\ & \Pi(a : A)(b : B).\, R\,a\,b \simeq (a =_{\downarrow_e} b)\end{aligned}$$

Ce type regroupe une relation R et une équivalence e telles que R est équivalente à la relation fonctionnelle associée à $\downarrow_e$. Une propriété essentielle de cette nouvelle traduction est la suivante :

LEMME 8.2.7
Sous l'axiome d'univalence, l'interprétation de l'univers dans la paramétricité univalente est équivalente à l'équivalence, c'est-à-dire qu'il existe un terme $\mathrm{ParamEquiv}_i$ tel que

$$\mathrm{ParamEquiv}_i : \Pi(A\,B : \square_i).\, \mathrm{Param}_i\, A\,B \simeq (A \simeq B).$$

Démonstration. Soient A et B deux types. On a :

$$\begin{aligned}
& \mathrm{Param}_i\, A\,B \\
& \quad \downarrow \text{définition} \\
\equiv\ & \Sigma(R : A \to B \to \square_i)(e : A \simeq B).\, \Pi(a : A)(b : B).\, R\,a\,b \simeq (a =_{\downarrow_e} b) \\
& \quad \downarrow \text{échange de lieurs non dépendants entre eux} \\
\simeq\ & \Sigma(e : A \simeq B)(R : A \to B \to \square_i).\, \Pi(a : A)(b : B).\, R\,a\,b \simeq (a =_{\downarrow_e} b) \\
& \quad \downarrow \text{principe d'univalence} \\
\simeq\ & \Sigma(e : A \simeq B)(R : A \to B \to \square_i).\, \Pi(a : A)(b : B).\, R\,a\,b = (a =_{\downarrow_e} b) \\
& \quad \downarrow \text{reformulation de l'égalité en style sans point} \\
\simeq\ & \Sigma(e : A \simeq B)(R : A \to B \to \square_i).\, R = \lambda(a : A)(b : B).\, (a =_{\downarrow_e} b) \\
& \quad \downarrow \text{type contractile dans le membre droit de la paire dépendante} \\
\simeq\ & A \simeq B
\end{aligned}$$

■

Cette observation est en fait un exemple d'une technique plus générale disponible pour construire des modèles syntaxiques de la théorie des types [59]. En effet, l'enrichissement du témoin de paramétricité sur l'univers change la structure de tous les témoins de paramétricité sur les types, en en faisant des paires dépendantes contrairement à la traduction brute où ces témoins sont des relations. En l'état, la traduction est donc mal formée et le théorème d'abstraction devient invalide. Dans ces modèles, une manière classique de récupérer le théorème d'abstraction consiste alors à raffiner la traduction en deux variantes, pour traiter correctement les termes qui sont aussi des types. Ainsi, la traduction d'un type $T : \square_i$ en tant que *terme*, notée $[T]_u$, est bien une paire dépendante contenant une relation ainsi que les données supplémentaires indiquées par l'interprétation de l'univers Param_i. La traduction de T en tant que *type*, $[\![T]\!]_u$, sera la relation elle-même, c'est-à-dire la projection de la paire dépendante $[T]_u$ sur sa première composante. La traduction complète de la paramétricité univalente est la suivante :

[59] : Boulier *et coll.* (2017), « The next 700 syntactical models of type theory »

$$
\begin{aligned}
[\![\langle\rangle]\!]_u &:= \langle\rangle \\
[\![\Gamma, x : A]\!]_u &:= [\![\Gamma]\!]_u, x : A, x' : A', x_R : [\![A]\!]_u\, x\, x' \\
\\
[\![A]\!]_u &:= [A]_u.1 \\
\\
[\square_i]_u &:= (\mathrm{Param}_i\ ;\ \mathrm{Equiv}_{\square_i}\ ;\ \mathrm{Coh}_{\square_i}) \\
[x]_u &:= x_R \\
[\Pi x : A.\, B]_u &:= \begin{pmatrix} R_\Pi\, A\, B\,; \\ \mathrm{Equiv}_\Pi\, A\, A'\, [\![A]\!]_u\, B\, B'\, [\![B]\!]_u\,; \\ \mathrm{Coh}_\Pi\, A\, A'\, [\![A]\!]_u\, B\, B'\, [\![B]\!]_u \end{pmatrix} \\
[\lambda x : A.\, t]_u &:= \lambda (x : A)(x' : A')(x_R : [\![A]\!]_u\, x\, x').\, [t]_u \\
[f\, t]_u &:= [f]_u\, t\, t'\, [t]_u
\end{aligned}
$$

Fig. 8.2 : Traduction de paramétricité univalente pour CC_ω.

Les cas les plus intéressants sont l'univers et le produit dépendant, les autres cas étant similaires à la traduction brute. Étant des types, leur traduction est donc un triplet dépendant, la première composante étant une relation, la seconde une preuve d'équivalence, et la dernière étant une preuve de cohérence entre les deux termes précédents. La relation R_Π a la même structure que dans la traduction brute, en utilisant la traduction univalente pour le domaine et le codomaine :

$$
R_\Pi\, A\, B \quad := \quad \lambda f\, f'.\, \Pi (x : A)(x' : A')(x_R : [\![A]\!]_u\, x\, x').\, [\![B]\!]_u\, (f\, x)\, (f'\, x')
$$

Les preuves d'équivalence ($\mathrm{Equiv}_{\square_i}$ et Equiv_Π) et de cohérence ($\mathrm{Coh}_{\square_i}$ et Coh_Π) sont disponibles dans l'article [14].

[14] : Tabareau *et coll.* (2021), « The marriage of univalence and parametricity »

On peut alors énoncer le théorème d'abstraction de la paramétricité univalente, où $\vdash_u$ fait référence à un jugement de typage supposant l'axiome d'univalence :

Théorème 8.2.1 (Théorème d'abstraction pour la paramétricité univalente)
Si $\Gamma \vdash t : T$, alors $[\![\Gamma]\!]_u \vdash_u [t]_u\ :\ [\![T]\!]_u\, t\, t'$.

On remarque toutefois que pour respecter le théorème d'abstraction, la définition de $[\Box_i]_u$ utilise le principe d'univalence d'une façon essentielle. En effet, puisque la relation sur l'univers est Param_i, on doit avoir :

$$\begin{aligned} & [\Box_i]_u : [\![\Box_{i+1}]\!]_u \, \Box_i \, \Box_i \\ \text{soit} \quad & [\Box_i]_u : \mathrm{Param}_{i+1} \, \Box_i \, \Box_i \end{aligned}$$

L'équivalence entre un univers et lui-même, $\mathrm{Equiv}_{\Box_i}$, est triviale et utilise l'identité pour les deux fonctions de transport. Ainsi, prouver la propriété de cohérence $\mathrm{Coh}_{\Box_i}$ revient à prouver que la relation est équivalente à l'égalité sur l'univers, c'est-à-dire :

$$\Pi A\, B : \Box_i. \, \mathrm{Param}_i \, A \, B \simeq (A = B).$$

La preuve se fait à partir du Lemme 8.2.7 et nécessite bien l'axiome d'univalence.

8.2.4 Omniprésence de l'axiome d'univalence

Prenons l'exemple de but suivant :

$$\Pi(P : \mathbb{N} \to \Box). \, P\, 0 \to P\, 0$$

Si l'on associe $\mathbb{N}$ à un autre type, par exemple un encodage binaire N des entiers naturels, alors le but associé par paramétricité sera le suivant, où 0_N est la constante associée à 0 :

$$\Pi(P' : N \to \Box). \, P'\, 0_N \to P'\, 0_N$$

Le témoin de paramétricité univalente est construit par induction sur la syntaxe du but initial. Pendant la traversée de ce terme, on est forcé de traduire $\Box$, donc d'invoquer la preuve $\mathrm{Coh}_{\Box}$ qui nécessite l'axiome d'univalence. Pourtant, on se situe dans un cas dans lequel la preuve d'implication entre les deux buts est faisable sans axiome. La preuve à effectuer est la suivante :

$$\frac{H' : \Pi(P' : N \to \Box). \, P'\, 0_N \to P'\, 0_N \qquad P : \mathbb{N} \to \Box \qquad p_0 : P\, 0}{P\, 0}$$

La preuve triviale p_0 est possible ici, mais la preuve générale valable pour d'autres buts est celle qui passe par H' en plaçant des fonctions de plongement dans un sens ou l'autre en fonction des types à habiter. On instancie alors H' avec $P' := P \circ \downarrow_{\mathbb{N}}$. Si l'on définit des fonctions de plongement qui envoient le zéro d'un type vers le zéro de l'autre, alors le second argument de H' peut être p_0 inchangé. Ainsi, on a une preuve d'implication entre les deux buts sans utiliser d'axiome. Ce cas se produit dès qu'une instance de $\Box$ est présente dans le but initial sans être problématique pour la mise au point d'une preuve manuelle.

L'équivalence de types, sur mesure 9

Comme expliqué dans le chapitre précédent, la paramétricité fournit un cadre général de mise en relation de termes dans un λ-calcul, l'exemple le plus poussé étant la paramétricité univalente. Cette traduction très puissante permet en effet, au prix de l'ajout d'un axiome, de générer des preuves d'équivalence par induction sur la syntaxe, et ce à partir de n'importe quel terme de CC_ω. En ajoutant les constantes au calcul, il est possible d'implémenter un outil dans lequel l'utilisateur peut ajouter des témoins de paramétricité hétérogènes, c'est-à-dire des équivalences entre des types différents, avant le début de la traduction, permettant de générer des équivalences entre des buts différents et ainsi faire du transfert de preuve.

Cependant, la nécessité d'ajouter l'axiome d'univalence au calcul pose problème pour deux raisons. Premièrement, comme exposé en § 8.2.4, de nombreux buts pourraient faire l'objet d'un pré-traitement au résultat identique à celui d'une traduction de paramétricité univalente, mais sans utiliser l'axiome d'univalence, tandis que la paramétricité univalente y fait appel systématiquement. Deuxièmement, dans un objectif pragmatique d'implémentation d'un nouveau greffon de paramétricité pour faciliter le transfert de preuve dans Coq, il faut s'assurer que le cadre de conception de la traduction de paramétricité soit logiquement cohérent avec la théorie logique sous-jacente de l'assistant de preuve. Or, il se peut que l'axiome d'univalence introduise des incompatibilités avec la version standard de Coq.[1]

Ce chapitre présente donc une nouvelle relation de paramétricité, basée sur une nouvelle formulation de l'équivalence de types (§ 9.1) qui expose toutes les informations d'une manière symétrique et atomique, contrairement à la formulation classique présentée en Définition 8.2.4. La particularité des témoins de paramétricité dans ce cadre est qu'ils contiennent une quantité variable d'informations, allant du témoin brut dans le cas le plus faible au témoin univalent dans le cas le plus fort. Il n'y a donc pas une seule traduction de paramétricité, mais un ensemble d'associations possibles (§ 9.2), l'objectif étant de moduler la taille du témoin de paramétricité et éviter de dépendre de l'axiome d'univalence lorsque c'est possible.

9.1 Une nouvelle formulation de l'équivalence 88
9.1.1 Décomposition de l'équivalence 89
9.1.2 Recomposition hiérarchique des témoins de paramétricité . 92
9.2 Peuplement de la hiérarchie de relations 93
9.2.1 Traitement des univers 94
9.2.2 Traitement des produits dépendants 94
9.2.3 Le cas des produits non dépendants 95

1 : On l'utilise en général dans la bibliothèque HoTT [65] pour travailler dans un cadre maîtrisé.

9.1 Une nouvelle formulation de l'équivalence

Comme montré précédemment, la Définition 8.2.6 décrit un témoin de paramétricité univalente très riche, car il demande systématiquement une équivalence, et très dense, car il contient seulement trois valeurs. De ce fait, la condition de cohérence dans le cas de l'univers demande que la définition du témoin soit équivalente à l'égalité entre types, ce qui induit un appel systématique à l'axiome d'univalence. Or, certains buts contiennent des occurrences de $\square$ pour lesquelles il est superflu d'exiger une équivalence pour effectuer un pré-traitement.

La situation suggère la recherche d'une relation de paramétricité hybride, ne demandant l'équivalence que dans les cas où elle est strictement nécessaire, et demandant moins d'informations lorsque c'est possible. Ceci implique de décomposer l'équivalence de types (§ 9.1.1), c'est-à-dire étaler les informations qu'elle

contient. Une fois la décomposition effectuée, il est possible de façonner une hiérarchie de témoins de paramétricité en sélectionnant différentes valeurs de ce Σ-type (§ 9.1.2).

9.1.1 Décomposition de l'équivalence

Tout d'abord, on peut observer que la Définition 8.2.4 de l'équivalence de types est asymétrique, bien que ce fait soit d'une certaine manière masqué par la notation infixe $A \simeq B$. En effet, premièrement, les données d'une équivalence $e : A \simeq B$ privilégient la direction gauche–droite, puisque $\uparrow_e$ est directement accessible à partir de e comme première projection, alors que l'accès au transport de droite à gauche nécessite une projection supplémentaire. Ensuite, l'énoncé de la propriété d'adjonction, disponible dans la Définition 8.2.2, est le suivant :

$$\mathrm{ap}\,\phi \circ \mathrm{sec} \doteqdot \mathrm{ret} \circ \phi$$

Cet énoncé utilise les preuves sec et ret, respectivement les propriétés de section et de rétraction de e, mais pas de façon symétrique, bien qu'il soit possible d'obtenir une définition équivalente en les permutant. Cet enchevêtrement tue tout espoir de pouvoir retracer les rôles respectifs de chaque fonction de transport, au cours d'une traduction de paramétricité univalente donnée. Cependant, l'exercice 4.2 du livre HoTT [64] suggère une présentation symétrique de l'équivalence de types, à l'aide de relations fonctionnelles.

[64] : Univalent Foundations Program (2013), *Homotopy Type Theory : Univalent Foundations of Mathematics*

Définition 9.1.1
Toute relation $R : A \to B \to \square_i$ est dite *fonctionnelle* lorsque chaque valeur de A est uniquement reliée à exactement une valeur de B dans R :

$$\mathrm{IsFun}(R) \quad :\equiv \quad \Pi a : A.\, \mathrm{IsContr}(\Sigma b : B.\, R\, a\, b)$$

où $\mathrm{IsContr}(\cdot)$ est le prédicat classique de contractilité :

$$\mathrm{IsContr}(T) \quad :\equiv \quad \Sigma t_0 : T.\, \Pi t : T.\, t = t_0$$

On peut à présent formuler l'équivalence de types de manière symétrique, comme une relation fonctionnelle dont la relation symétrique est également fonctionnelle :

Lemme 9.1.2
Pour tous types $A, B : \square_i$, le type $A \simeq B$ est équivalent à :

$$\Sigma R : A \to B \to \square_i.\, \mathrm{IsFun}(R) \times \mathrm{IsFun}(R^{-1})$$

où la relation $R^{-1} : B \to A \to \square_i$ permute simplement les arguments d'une relation arbitraire $R : A \to B \to \square_i$.

Effectuons la preuve de ce résultat, laissée comme exercice dans [64].

Le lemme suivant, qui explique pourquoi $\mathrm{IsFun}(\cdot)$ caractérise des relations fonctionnelles, est nécessaire :

Lemme 9.1.3
Pour tous types $A, B : \square_i$, on a :

$$(A \to B) \simeq \Sigma R : A \to B \to \square_i.\, \mathrm{IsFun}(R).$$

Démonstration. La preuve s'effectue en chaînant les équivalences suivantes :

$$\begin{aligned}
& \Sigma R : A \to B \to \square_i.\, \mathrm{IsFun}(R) \\
& \quad \downarrow \text{définition} \\
\equiv\ & \Sigma R : A \to B \to \square_i.\, \Pi a : A.\, \mathrm{IsContr}(\Sigma b : B.\, R\, a\, b) \\
& \quad \downarrow \text{échange de lieurs non dépendants entre eux} \\
\simeq\ & \Pi a : A.\, \Sigma R : A \to B \to \square_i.\, \mathrm{IsContr}(\Sigma b : B.\, R\, a\, b) \\
& \quad \downarrow \text{en posant } P := R\, a, \text{ le premier lieur n'est plus dépendant} \\
\simeq\ & A \to \Sigma P : B \to \square_i.\, \mathrm{IsContr}(\Sigma b : B.\, P\, b) \\
& \quad \downarrow \text{lemme standard de HoTT} \\
\simeq\ & A \to B
\end{aligned}$$

■

Preuve du Lemme 9.1.2. La preuve se fait en chaînant les équivalences suivantes :

$$\begin{aligned}
& A \simeq B \\
& \quad \downarrow \text{définition} \\
\simeq\ & \Sigma f : A \to B.\, \mathrm{IsEquiv}(f) \\
& \quad \downarrow \text{résultat classique de HoTT} \\
\simeq\ & \Sigma f : A \to B.\, \Pi b : B.\, \mathrm{IsContr}(\Sigma a : A.\, f\, a = b) \\
& \quad \downarrow \text{définition de } \mathrm{IsFun}(\cdot) \\
\simeq\ & \Sigma f : A \to B.\, \mathrm{IsFun}(\lambda (b : B)(a : A).\, f\, a = b) \\
& \quad \downarrow \text{Lemme 9.1.3} \\
\simeq\ & \Sigma (f : \Sigma (R : A \to B \to \square_i).\, \mathrm{IsFun}(R)).\, \mathrm{IsFun}(f.1^{-1}) \\
& \quad \downarrow \text{associativité de } \Sigma \\
\simeq\ & \Sigma R : A \to B \to \square_i.\, \mathrm{IsFun}(R) \times \mathrm{IsFun}(R^{-1})
\end{aligned}$$

■

La version symétrique de l'équivalence de types donnée par le Lemme 9.1.2 n'expose toutefois pas explicitement les deux fonctions de transport dans ses données, bien que ce contenu calculatoire puisse être extrait *via* des projections sur les preuves de contractilité. En fait, il est possible de concevoir une définition de l'équivalence qui donne directement les deux fonctions de transport tout en demeurant symétrique. L'ingrédient essentiel de cette reformulation est une caractérisation alternative des relations fonctionnelles.

Définition 9.1.4
Pour tous types $A, B : \square_i$, une relation $R : A \to B \to \square_i$ est une *application univalente*, notée $\mathrm{IsUMap}(R)$, lorsqu'il existe une fonction dont elle est exactement le graphe et que cette même propriété est accompagnée d'une

preuve de cohérence :

$$\begin{aligned}\mathrm{IsUMap}(R) \quad :=\quad & \Sigma(m : A \to B)\\ & (\pi_1 : \Pi(a : A)(b : B).\, m\, a = b \to R\, a\, b)\\ & (\pi_2 : \Pi(a : A)(b : B).\, R\, a\, b \to m\, a = b).\\ & \quad \Pi(a : A)(b : B).\, (\pi_1\, a\, b) \circ (\pi_2\, a\, b) \doteqdot id\end{aligned}$$

Le lemme central de cette sous-section, prouvé formellement dans le code de Trocq, est le suivant :

Lemme 9.1.5
Pour tous types $A, B : \square_i$ et toute relation $R : A \to B \to \square_i$,

$$\mathrm{IsFun}(R) \simeq \mathrm{IsUMap}(R).$$

Démonstration. La preuve s'effectue en réécrivant le côté gauche petit à petit :

$$\begin{aligned}& \mathrm{IsFun}(R)\\ & \quad \downarrow \text{définitions}\\ \simeq\ & \Pi a_0.\, \Sigma(c : \Sigma b.\, R\, a_0\, b).\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, c = p\\ & \quad \downarrow \text{associativité de } \Sigma\\ \simeq\ & \Pi a_0.\, \Sigma b_0.\, \Sigma r : R\, a_0\, b_0.\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, (b_0\,;\, r) = p\\ & \quad \downarrow \text{choix intuitionniste}\\ \simeq\ & \Sigma \phi : A \to B.\, \Pi a_0.\, \Sigma r : R\, a_0\, (\phi\, a_0).\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, (\phi\, a_0\,;\, r) = p\\ & \quad \downarrow \text{permutation des lieurs et abstraction de } a_0 \text{ dans } r\\ \simeq\ & \Sigma \phi.\, \Sigma(r : \Pi a.\, R\, a\, (\phi\, a)).\, \Pi a_0.\, \Pi(p : \Sigma b.\, R\, a_0\, b).\, (\phi\, a_0\,;\, r\, a_0) = p\\ & \quad \downarrow \text{découpage de } p\\ \simeq\ & \Sigma \phi.\, \Sigma r.\, \Pi a_0.\, \Pi b_0.\, \Pi(r' : R\, a_0\, b_0).\, (\phi\, a_0\,;\, r\, a_0) = (b\,;\, r')\\ & \quad \downarrow \text{décomposition de la preuve d'égalité sur une paire dépendante}\\ \simeq\ & \Sigma \phi.\, \Sigma r.\, \Pi a_0.\, \Pi b_0.\, \Pi r'.\, \Sigma e : \phi\, a_0 = b_0.\, r\, a_0 =_e r'\\ & \quad \downarrow \text{permutation des lieurs et abstraction de } a_0,\ b_0 \text{ et } r' \text{ dans } e\\ \simeq\ & \Sigma \phi.\, \Sigma r.\, \Sigma(e : \Pi a.\, \Pi b.\, R\, a\, b \to \phi\, a = b).\, \Pi a_0.\, \Pi b_0.\, \Pi r'.\, r =_{e\, a_0\, b_0\, r'} r'\end{aligned}$$

On identifie que ϕ est la valeur m dans la définition de $\mathrm{IsUMap}(R)$, et e est la valeur π_2. En renommant et en réorganisant les Σ-types, il reste à prouver la propriété suivante :

$$\begin{aligned}& \Sigma(\pi_1 : \Pi a.\, \Pi b.\, m\, a = b \to R\, a\, b).\, (\pi_1\, a_0\, b_0) \circ (\pi_2\, a_0\, b_0) \doteqdot \mathrm{id}\\ \simeq\ & \Sigma(r : \Pi a.\, R\, a\, (m\, a)).\, \Pi(r' : R\, a_0\, b_0).\, r\, a_0 =_{\pi_2\, a_0\, b_0\, r'} r'\end{aligned}$$

Nous renvoyons le lecteur au code associé. ■

Le corollaire direct est une nouvelle caractérisation de l'équivalence de types :

Théorème 9.1.1
Pour tous types $A, B : \square$, on a :

$$(A \simeq B) \simeq \mathrm{Param}^{\top}\, A\, B$$

où la relation $\mathrm{Param}^{\top}\, A\, B$ est définie comme :

$$\begin{aligned}\mathrm{Param}^{\top}\, A\, B \quad &:= \quad \Sigma R : A \to B \to \square.\\ &\qquad \mathrm{IsUMap}(R) \times \mathrm{IsUMap}(R^{-1})\end{aligned}$$

La collection de données obtenue est maintenant symétrique, puisque la version inverse de l'équivalence basée sur les applications univalentes peut être obtenue en inversant la relation et en échangeant les deux preuves de fonctionnalité. Si la règle η pour les enregistrements est vérifiée, cette symétrie est même définitionnellement involutive.

9.1.2 Recomposition hiérarchique des témoins de paramétricité

La Définition 9.1.4 des applications univalentes et la reformulation de l'équivalence de types qui en résulte suggèrent d'introduire une hiérarchie de structures pour les relations hétérogènes, qui expose à quel point une relation donnée est proche de l'équivalence de types. Cette distance est elle aussi décrite en termes de structure disponible respectivement sur les fonctions de transport de gauche à droite et de droite à gauche.

Définition 9.1.6
Pour $n, k \in \{0, 1, 2_{\mathrm{a}}, 2_{\mathrm{b}}, 3, 4\}$ et $\alpha = (n, k)$, la relation Param^{α} est définie comme :

$$\begin{aligned}\mathrm{Param}^{\alpha} \quad &:= \quad \lambda(A\, B : \square).\\ &\qquad \Sigma R : A \to B \to \square.\, \mathrm{Class}_{\alpha}\, R\end{aligned}$$

où la *classe applicative* $\mathrm{Class}_{\alpha}\, R$ se déplie en un type produit de deux *témoins unilatéraux*, l'un de A vers B, l'autre de B vers A :

$$(\mathrm{M}_n\, A\, B\, R) \times (\mathrm{M}_k\, B\, A\, R^{-1})$$

avec M_i défini comme :

$$\begin{aligned}\mathrm{M}_0\, A\, B\, R \quad &:= \quad .\\ \mathrm{M}_1\, A\, B\, R \quad &:= \quad A \to B\\ \mathrm{M}_{2_{\mathrm{a}}}\, A\, B\, R \quad &:= \quad \Sigma m : A \to B.\, G_{2_{\mathrm{a}}}\, A\, B\, m\, R\\ \mathrm{M}_{2_{\mathrm{b}}}\, A\, B\, R \quad &:= \quad \Sigma m : A \to B.\, G_{2_{\mathrm{b}}}\, A\, B\, m\, R\\ \mathrm{M}_3\, A\, B\, R \quad &:= \quad \Sigma(m : A \to B).\\ &\qquad (G_{2_{\mathrm{a}}}\, A\, B\, m\, R) \times (G_{2_{\mathrm{b}}}\, A\, B\, m\, R)\\ \mathrm{M}_4\, A\, B\, R \quad &:= \quad \Sigma(m : A \to B)\\ &\qquad (g_{\mathrm{a}} : G_{2_{\mathrm{a}}}\, A\, B\, m\, R)\\ &\qquad (g_{\mathrm{b}} : G_{2_{\mathrm{b}}}\, A\, B\, m\, R).\\ &\qquad\quad \Pi a\, b.\, (g_{\mathrm{a}}\, a\, b) \circ (g_{\mathrm{b}}\, a\, b) \div id\end{aligned}$$

avec

$$\begin{aligned}G_{2_{\mathrm{a}}}\, A\, B\, m\, R \quad &:= \quad \Pi(a : A)(b : B).\, m\, a = b \to R\, a\, b\\ G_{2_{\mathrm{b}}}\, A\, B\, m\, R \quad &:= \quad \Pi(a : A)(b : B).\, R\, a\, b \to m\, a = b\end{aligned}$$

Pour tous types A et B, et tout $r : \mathrm{Param}^{\alpha}\ A\ B$, on utilisera les notations $\mathrm{rel}(r)$, $\mathrm{map}(r)$ et $\mathrm{comap}(r)$ pour se référer respectivement à la relation, l'application de type $A \to B$, celle de type $B \to A$, incluse dans les données de r, pour un α suffisant.

Définition 9.1.7
Nous désignons par $\mathcal{A}$ l'ensemble $\{0, 1, 2_{\mathrm{a}}, 2_{\mathrm{b}}, 3, 4\}^2$, utilisé pour indexer les classes applicatives dans la Définition 9.1.6. Cet ensemble est partiellement ordonné pour l'ordre produit sur $\{0, 1, 2_{\mathrm{a}}, 2_{\mathrm{b}}, 3, 4\}$ défini à partir de l'ordre partiel $0 < 1 < 2_* < 3 < 4$ avec 2_* étant soit 2_{a} soit 2_{b}, et avec 2_{a} et 2_{b} incomparables.

Remarque 9.1.8
La relation $\mathrm{Param}^{(4,4)}$ de la Définition 9.1.6 coïncide avec la relation $\mathrm{Param}^{\top}$ introduite dans le Théorème 9.1.1, équivalente au type des témoins de paramétricité univalente. De même, nous désignons par $\mathrm{Param}^{\bot}$ la relation $\mathrm{Param}^{(0,0)}$, ce qui revient à avoir une relation $R : A \to B \to \square$ comme dans la traduction de la paramétricité brute. Une relation dotée de la structure $\mathrm{Param}^{(4,0)}\ A\ B$ (respectivement $\mathrm{Param}^{(3,3)}\ A\ B$) est le graphe d'une application univalente de A vers B (respectivement un isomorphisme entre A et B).

Dans le code associé, le treillis correspondant à la collection des M_n est implémenté comme une hiérarchie de tuples dépendants (plus précisément, de types d'enregistrements). Chaque flèche de la Figure 9.1 représente l'inclusion des données de la structure source dans les données de la structure cible. De plus, les nœuds sont étiquetés avec les noms des champs d'enregistrement correspondants introduits par la structure plus riche.

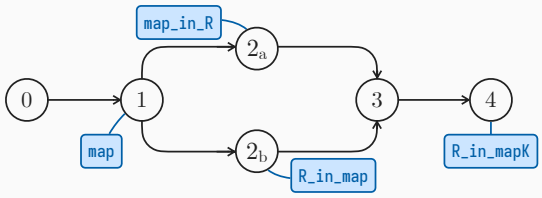


Fig. 9.1 : Implémentation de la hiérarchie de la Définition 9.1.6

9.2 Peuplement de la hiérarchie de relations

À présent, on peut revisiter les traductions de paramétricité de § 8. En particulier, la combinaison du Théorème 9.1.1 avec le théorème d'abstraction de la paramétricité univalente assure l'existence d'un terme $p_{\square_i}$ tel que :

$$\vdash_u p_{\square_i} : \mathrm{Param}^{\top}_{i+1}\ \square_i\ \square_i \quad \text{et} \quad \mathrm{rel}(p_{\square_i}) \simeq \mathrm{Param}^{\top}_i.$$

Autrement dit, la relation $\mathrm{Param}^{\top} : \square \to \square \to \square$ peut être dotée d'une structure $\mathrm{Param}^{\top}$, en supposant l'univalence. De même, l'équation pour les univers, dans la Figure 8.1 décrivant la traduction de paramétricité brute, peut être lue comme le fait que la relation $\mathrm{Param}^{\bot}$ sur les univers peut être dotée d'une structure $\mathrm{Param}^{\bot}\ \square\ \square$.

9.2.1 Traitement des univers

La hiérarchie des structures introduite par la Définition 9.1.6 permet désormais une analyse plus fine des interprétations relationnelles possibles des univers. Non seulement, cela regrouperait les traductions de paramétricité brute et univalente dans un même cadre, mais cela permettrait également de généraliser la paramétricité à une classe plus large de relations. Dans ce but, nous généralisons l'observation précédente sur l'ingrédient clé de la traduction des univers : pour chaque $\alpha \in \mathcal{A}$, la relation $\mathrm{Param}^{\alpha} : \square \to \square \to \square$ peut être dotée de plusieurs structures du treillis, et l'on doit identifier les quelles selon α. Autrement dit, on doit identifier les paires $(\alpha, \beta) \in \mathcal{A}^2$ pour lesquelles il est possible de construire un terme $p_{\square}^{\alpha,\beta}$ tel que :

$$\vdash_u p_{\square}^{\alpha,\beta} : \mathrm{Param}^{\beta}\ \square\ \square \quad \text{et} \quad \mathrm{rel}(p_{\square}^{\alpha,\beta}) \equiv \mathrm{Param}^{\alpha} \tag{9.1}$$

Notons que nous visons ici une égalité définitionnelle entre $\mathrm{rel}(p_{\square}^{\alpha,\beta})$ et Param^{α}, plutôt qu'une équivalence. Il est facile de voir qu'un terme $p_{\square}^{\alpha,\bot}$ existe pour tout $\alpha \in \mathcal{A}$, puisque $\mathrm{Param}^{\bot}$ n'exige aucune structure sur la relation. En revanche, il est impossible de construire un terme $p_{\square}^{\bot,\top}$, c'est-à-dire de transformer une relation arbitraire en une équivalence de types.

Définition 9.2.1
On note $\mathcal{D}_{\square}$ le sous-ensemble suivant de $\mathcal{A}^2$:

$$\mathcal{D}_{\square} = \{(\alpha, \beta) \in \mathcal{A}^2 \mid \alpha = \top \vee \beta \in \{0, 1, 2_{\mathrm{a}}\}^2\}$$

Le code associé [2] construit les termes $p_{\square}^{\alpha,\beta}$ pour chaque paire $(\alpha, \beta) \in \mathcal{D}_{\square}$, en utilisant un méta-programme pour les générer à partir d'une collection minimale de définitions manuelles. En particulier, en supposant l'univalence, il est possible de construire un terme $p_{\square}^{\top,\top}$, qui peut être considéré comme un analogue de la traduction $[\square]$ de la paramétricité univalente. Plus généralement, les termes fournis $p_{\square}^{\alpha,\beta}$ dépendent de l'univalence si et seulement si $\beta \notin \{0, 1, 2_{\mathrm{a}}\}^2$.

2 : Fichier `Param_Type.v`.

9.2.2 Traitement des produits dépendants

La question naturelle suivante est l'étude des structures possibles Param^{γ} qui peuvent équiper une relation associée à un produit dépendant $\Pi x : A.\, B$, lorsque les relations associées aux types A et B sont respectivement équipées des structures Param^{α} et Param^{β}.

Autrement dit, nous devons identifier les triplets $(\alpha, \beta, \gamma) \in \mathcal{A}^3$ pour lesquels il est possible de construire un terme p_{Π}^{γ} tel que :

$$\frac{\Gamma \vdash A_R : \mathrm{Param}^{\alpha}\ A\ A' \qquad \Gamma, x : A, x' : A', x_R : A_R\ x\ x' \vdash B_R : \mathrm{Param}^{\beta}\ B\ B'}{\Gamma \vdash p_{\Pi}^{\gamma}\ A_R\ B_R : \mathrm{Param}^{\gamma}\ (\Pi x : A.\, B)\ (\Pi x' : A'.\, B')} \quad \text{et}$$

$$\begin{aligned} \mathrm{rel}(p_{\Pi}^{\gamma}\ A_R\ B_R) \quad \equiv \quad & \lambda f\ f'.\, \Pi (x : A)(x' : A')(x_R : \mathrm{rel}(A_R)\ x\ x').\\ & \mathrm{rel}(B_R)\ (f\ x)\ (f\ x') \end{aligned}$$

La collection correspondante de triplets peut en fait être décrite comme une fonction $\mathcal{D}_{\Pi} : \mathcal{A} \to \mathcal{A}^2$, telle que $\mathcal{D}_{\Pi}(\gamma) = (\alpha, \beta)$ fournit les exigences *minimales*

sur les structures associées à A et B, en ce qui concerne l'ordre partiel sur $\mathcal{A}^2$. Le code associé [3] fournit une collection correspondante de termes p_Π^γ pour chaque $\gamma \in \mathcal{A}$, ainsi que tous les affaiblissements qui y sont associés. Une fois encore, ces définitions sont générées par un méta-programme. On observe en particulier que par symétrie, $p_\Pi^{(m,n)}$ peut être obtenu à partir de $p_\Pi^{(m,0)}$ et $p_\Pi^{(n,0)}$, en permutant ce dernier et en le collant au premier. Par conséquent, les valeurs de p_Π^γ et $\mathcal{D}_\Pi(\gamma)$ sont complètement déterminées par celles de $p_\Pi^{(m,0)}$ et $\mathcal{D}_\Pi(m,0)$. En particulier, pour tous $m, n \in \mathcal{A}$:

$$\mathcal{D}_\Pi(m,n) = ((m_A, n_A), (m_B, n_B))$$

pour $m_A, n_A, m_B, n_B \in \mathcal{A}$ définis comme

$$\mathcal{D}_\Pi(m,0) = ((0, n_A), (m_B, 0))$$
$$\mathcal{D}_\Pi(n,0) = ((0, m_A), (n_B, 0))$$

3 : Fichier `Param_forall.v`.

Nous résumons en Figure 9.2 les valeurs de $\mathcal{D}_\Pi(m,0)$.[4]

m	$\mathcal{D}_\Pi(m,0)_1$	$\mathcal{D}_\Pi(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,2_a)$	$(1,0)$
2_a	$(0,4)$	$(2_a,0)$
2_b	$(0,2_a)$	$(2_b,0)$
3	$(0,4)$	$(3,0)$
4	$(0,4)$	$(4,0)$

m	$\mathcal{D}_\to(m,0)_1$	$\mathcal{D}_\to(m,0)_2$
0	$(0,0)$	$(0,0)$
1	$(0,1)$	$(1,0)$
2_a	$(0,2_b)$	$(2_a,0)$
2_b	$(0,2_a)$	$(2_b,0)$
3	$(0,3)$	$(3,0)$
4	$(0,4)$	$(4,0)$

4 : Les cases grisées mettent en évidence une dépendance plus faible dans le cas d'un type flèche par rapport au cas général du produit dépendant.

Fig. 9.2 : Dépendances minimales pour les produits dépendants et non dépendants à la classe $(m,0)$

9.2.3 Le cas des produits non dépendants

Notons que dans le cas d'un produit non dépendant, la construction de $p_\to^\gamma$ nécessite moins de structure sur le domaine A d'un type flèche $A \to B$, ce qui motive l'introduction de la fonction $\mathcal{D}_\to(\gamma)$. L'utilisation du combinateur pour les produits dépendants pour interpréter un type flèche, bien que correcte, entraîne potentiellement des exigences de structure (et d'axiomes) inutiles. Le code associé [5] inclut une construction des termes $\mathcal{D}_t o^\gamma$ pour tout $\gamma \in \mathcal{A}$.

5 : Fichier `Param_arrow.v`.

Un calcul pour le transfert de preuve 10

10.1 Séquents de paramétricité brute 96
10.2 Séquents de paramétricité univalente 98
10.3 Théorie des types annotés . 99
10.4 Le calcul Trocq 100
10.5 Constantes 102

Ce chapitre présente Trocq, un cadre pour le transfert de preuve conçu comme une généralisation des traductions de paramétricité, de manière à permettre l'interprétation des types comme des instances des structures introduites en § 9.2.1. Nous le présentons à l'aide de séquents, en étroite correspondance avec le système de typage de CC_ω, tout en expliquant de manière cohérente les transformations des termes et des contextes. Cette présentation s'écarte de la littérature sur la paramétricité dans les Systèmes de Typage Purs, mais elle se rapproche des implémentations réelles, dont la gestion nécessaire des contextes de paramétricité est mise sous le tapis par les conventions de notation.

Dans cet objectif, nous introduisons successivement quatre calculs de complexité croissante. Nous commençons par introduire cette présentation en séquents en reformulant la traduction de la paramétricité brute (§ 10.1) et celle de la paramétricité univalente (§ 10.2). Nous introduisons ensuite CC^+_ω , un Calcul des Constructions avec des annotations sur les sortes et du sous-typage (§ 10.3), avant de définir le calcul Trocq (§ 10.4).

10.1 Séquents de paramétricité brute

Nous introduisons des *contextes de paramétricité*, sous la forme d'une liste de triplets regroupant des paires de variables avec un témoin de leur liaison :

$$\Xi ::= \varepsilon \mid \Xi,\ x \sim x' \because x_R$$

On écrit $(x, x', x_R) \in \Xi$ s'il existe Ξ' et Ξ'' tels que :

$$\Xi = \Xi',\ x \sim x' \because x_R,\ \Xi''$$

On note $\mathrm{Var}(\Xi)$ la liste des variables liées dans un contexte de paramétricité Ξ :

$$\mathrm{Var}(\varepsilon) = \varepsilon \qquad \mathrm{Var}(\Xi,\ x \sim x' \because x_R) = \mathrm{Var}(\Xi), x, x', x_R$$

Un contexte de paramétricité Ξ est *bien formé* (et l'on note $\Xi \vdash$) si la liste $\mathrm{Var}(\Xi)$ est sans doublon. Dans ce cas, on utilise la notation $\Xi(x) = (x', x_R)$ comme synonyme de $(x, x', x_R) \in \Xi$.

Un *jugement de paramétricité* relie un contexte de paramétricité Ξ et trois termes M, M' et M_R de CC_ω. Les jugements de paramétricité sont définis par les règles de la Figure 10.1. On les note et lit de la façon suivante :

$$\Xi \vdash M \sim M' \because M_R$$

Dans le contexte Ξ, le terme M se traduit par le terme M', car M_R.

Lemme 10.1.1
La relation associant un terme M à une paire (M', M_R) telle que $\Xi \vdash M \sim M' \because M_R$, avec Ξ un contexte de paramétricité bien formé, est *fonctionnelle* :

$$\frac{}{\Xi \vdash \square_i \sim \square_i \mathrel{\because} \lambda(A\,B : \square_i).\, A \to B \to \square_i}\ (\textsc{ParamSort})$$

$$\frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash x \sim x' \mathrel{\because} x_R}\ (\textsc{ParamVar})$$

$$\frac{\Xi \vdash M \sim M' \mathrel{\because} M_R \quad \Xi \vdash N \sim N' \mathrel{\because} N_R}{\Xi \vdash M\,N \sim M'\,N' \mathrel{\because} M_R\,N\,N'\,N_R}\ (\textsc{ParamApp})$$

$$\frac{\Xi, x \sim x' \mathrel{\because} x_R \vdash M \sim M' \mathrel{\because} M_R}{\Xi \vdash \lambda x : A.\, M \sim \lambda x' : A'.\, M' \mathrel{\because} \lambda x\, x'\, x_R.\, M_R}\ (\textsc{ParamLam})$$

$$\frac{\Xi \vdash A \sim A' \mathrel{\because} A_R \qquad \Xi, x \sim x' \mathrel{\because} x_R \vdash B \sim B' \mathrel{\because} B_R \qquad x, x' \notin \mathrm{Var}(\Xi)}{\Xi \vdash \Pi x : A.\, B \sim \Pi x' : A'.\, B' \mathrel{\because} \lambda f\, g.\, \Pi x\, x'\, x_R.\, B_R\,(f\,x)\,(g\,x')}\ (\textsc{ParamPi})$$

Fig. 10.1 : Param : traduction de paramétricité binaire en style relationnel

pour tout terme M et tout Ξ bien formé :

$$\begin{aligned}&\forall M', N', M_R, N_R,\\ &\quad \Xi \vdash M \sim M' \mathrel{\because} M_R \wedge \Xi \vdash M \sim N' \mathrel{\because} N_R \implies\\ &\qquad (M', M_R) = (N', N_R)\end{aligned}$$

Démonstration. Immédiate par induction sur la syntaxe de M. ■

Cette présentation de la paramétricité fournit donc une définition alternative de la traduction $[\![\cdot]\!]$ de la Figure 8.1, en explicitant la gestion de contexte masquée par la convention de notation « prime » de cette dernière.

Définition 10.1.2
Un contexte de paramétricité Ξ est *admissible* pour un contexte de typage bien formé Γ, noté $\Gamma \rhd \Xi$, lorsque Ξ est bien formé en tant que contexte de paramétricité et que Γ fournit des annotations de type cohérentes pour tous les termes de Ξ, c'est-à-dire que pour toutes variables x, x', x_R telles que $\Xi(x) = (x', x_R)$ et pour tous termes A' et A_R :

$$\Xi \vdash \Gamma(x) \sim A' \mathrel{\because} A_R \quad \implies \quad \Gamma(x') = A' \ \wedge\ \Gamma(x_R) \equiv A_R\,x\,x'$$

On peut désormais énoncer et prouver un théorème d'abstraction :

Théorème 10.1.1 (Théorème d'abstraction)

$$\frac{\Gamma \vdash M : A \qquad \Gamma \rhd \Xi \qquad \Xi \vdash M \sim M' \mathrel{\because} M_R \qquad \Xi \vdash A \sim A' \mathrel{\because} A_R}{\Gamma \vdash M' : A' \qquad \text{et} \qquad \Gamma \vdash M_R : A_R\,M\,M'}$$

Démonstration. Par induction sur la dérivation de $\Xi \vdash M \sim M' \mathrel{\because} M_R$. ■

10.2 Séquents de paramétricité univalente

Nous proposons maintenant en Figure 10.2 une version reformulée de la traduction de paramétricité univalente [14], en utilisant le même style relationnel et en remplaçant la traduction des univers par la relation équivalente $\mathrm{Param}^{\top}$. Dans cette variante, les jugements de paramétricité sont notés de la façon suivante, où Ξ est un contexte de paramétricité et M, M' et M_R sont des termes de CC_ω :

$$\Xi \vdash_u M \sim M' \mathrel{:\!\!\sim} M_R$$

L'indice u rappelle que les jugements de typage $\Gamma \vdash_u M : A$ impliqués dans le théorème d'abstraction associé sont des jugements de typage de CC_ω muni de l'axiome d'univalence.

[14] : Tabareau *et coll.* (2021), « The marriage of univalence and parametricity »

$$\frac{}{\Xi \vdash_u \square_i \sim \square_i \mathrel{:\!\!\sim} p_{\square_i}^{\top,\top}} \text{(UParamSort)}$$

$$\frac{(x, x', x_R) \in \Xi \qquad \Xi \vdash}{\Xi \vdash_u x \sim x' \mathrel{:\!\!\sim} x_R} \text{(UParamVar)}$$

$$\frac{\Xi \vdash_u M \sim M' \mathrel{:\!\!\sim} M_R \qquad \Xi \vdash_u N \sim N' \mathrel{:\!\!\sim} N_R}{\Xi \vdash_u M\, N \sim M'\, N' \mathrel{:\!\!\sim} M_R\, N\, N'\, N_R} \text{(UParamApp)}$$

$$\frac{\Xi \vdash_u A \sim A' \mathrel{:\!\!\sim} A_R \qquad \Xi, x \sim x' \mathrel{:\!\!\sim} x_R \vdash_u M \sim M' \mathrel{:\!\!\sim} M_R}{\Xi \vdash_u \lambda x : A.\, M \sim \lambda x' : A'.\, M' \mathrel{:\!\!\sim} \lambda x\, x'\, x_R.\, M_R} \text{(UParamLam)}$$

$$\frac{\Xi \vdash_u A \sim A' \mathrel{:\!\!\sim} A_R \qquad \Xi, x \sim x' \mathrel{:\!\!\sim} x_R \vdash_u B \sim B' \mathrel{:\!\!\sim} B_R}{\Xi \vdash_u \Pi x : A.\, B \sim \Pi x' : A'.\, B' \mathrel{:\!\!\sim} p_\Pi^{\top}\, A_R\, B_R} \text{(UParamPi)}$$

Fig. 10.2 : UParam : règles de paramétricité univalente

On peut à présent reformuler le théorème d'abstraction pour la paramétricité univalente.

Théorème 10.2.1 (Théorème d'abstraction univalente)

$$\frac{\Gamma \vdash M : A \qquad \Gamma \rhd \Xi \qquad \Xi \vdash_u M \sim M' \mathrel{:\!\!\sim} M_R \qquad \Xi \vdash_u A \sim A' \mathrel{:\!\!\sim} A_R}{\Gamma \vdash M' : A' \quad \text{et} \quad \Xi \vdash_u M_R : \mathrm{rel}(A_R)\, M\, M'}$$

Démonstration. Par induction sur la dérivation de $\Xi \vdash_u M \sim M' \mathrel{:\!\!\sim} M_R$. ■

Remarque 10.2.1

Dans le Théorème 10.2.1, le terme $\mathrm{rel}(A_R)$ est une relation de type $A \to A' \to \square$. En effet,

$$\frac{\Gamma \vdash A : \square_i \qquad \Xi \vdash_u A \sim A' \mathrel{:\!\!\sim} A_R \qquad \Gamma \rhd \Xi}{\Gamma \vdash_u A_R : \mathrm{rel}(p_{\square_i}^{\top,\top})\, A\, A'}$$

implique que A_R est de type

$$\begin{aligned} & \mathrm{rel}(p_{\square_i}^{\top,\top})\, A\, A' \\ \equiv\ & \mathrm{Param}^{\top}\, A\, A' \\ \equiv\ & \Sigma R : A \to A' \to \square.\, \mathrm{IsUMap}(R) \times \mathrm{IsUMap}(R^{-1}) \end{aligned}$$

10.3 Théorie des types annotés

Nous sommes maintenant prêts à généraliser l'interprétation relationnelle des types fournie par la traduction de paramétricité univalente, afin de permettre l'interprétation des sortes par des instances de structures plus faibles que l'équivalence. Dans cet objectif, nous introduisons une variante CC^+_ω de CC_ω où chaque univers est annoté par une étiquette indiquant la structure disponible sur son interprétation relationnelle. Rappelons que nous avons utilisé en § 9.2.1 les paires $\alpha \in \mathcal{A}^2$ pour identifier les différentes structures du treillis qui décompose l'équivalence de types : ce sont les étiquettes annotant les sortes de CC^+_ω, de sorte que si A a le type $\square^\alpha$, alors la relation associée A_R a le type $\mathrm{Param}^\alpha\, A\, A'$. La syntaxe de CC^+_ω est donc la suivante :

$$M, N, A, B \in \mathcal{T}_{CC^+_\omega} ::= \square^\alpha_i \mid x \mid M\,N \mid \lambda x : A.\, M \mid \Pi x : A.\, B$$
$$\alpha \in \mathcal{A} = \{0, 1, 2_a, 2_b, 3, 4\}^2 \qquad i \in \mathbb{N}$$

Avant d'achever la définition formelle du cadre de transfert de preuve de Trocq, illustrons de manière informelle comment ces annotations vont conduire l'interprétation des termes, et en particulier d'un produit dépendant $\Pi x : A.\, B$. Dans ce cas, avant de traduire B, trois termes représentant la variable liée x, sa traduction x' et le témoin de paramétricité x_R sont ajoutés au contexte. Le type de x_R est $\mathrm{rel}(A_R)\, x\, x'$ où A_R est le témoin de paramétricité reliant A à sa traduction A'. Le rôle de l'annotation α sur la sorte du type A est donc d'indiquer la quantité d'informations disponibles dans le témoin x_R, en déterminant le type de A_R. Cette intention se reflète dans les règles de typage de CC^+_ω, qui s'appuient sur la définition des ensembles $\mathcal{D}_\square$, $\mathcal{D}_\to$ et $\mathcal{D}_\Pi$, introduits en § 9.2.

Le typage des termes dans CC^+_ω nécessite de définir une relation de *sous-typage* $\preccurlyeq$, définie par les règles de la Figure 10.3. Les règles de typage de CC^+_ω sont disponibles en Figure 10.4 et suivent les présentations classiques [67]. La relation $\equiv$ dans la règle SubConv est la relation de *conversion*, définie comme la fermeture de l'α-équivalence et de la β-réduction sur cette variante du λ-calcul. Nous avons donc deux types de jugement dans ce calcul :

[67] : Aspinall *et coll.* (2001), « Subtyping dependent types »

$$\Gamma \vdash_+ A \preccurlyeq B \quad \text{et} \quad \Gamma \vdash_+ M : A$$

où M, A et B sont des termes de CC^+_ω et Γ est un contexte dans CC^+_ω.[1]

1 : $\Gamma ::= \varepsilon \mid \Gamma, x : A.$

$$\frac{\Gamma \vdash_+ A : K \qquad \Gamma \vdash_+ B : K \qquad A \equiv B}{\Gamma \vdash_+ A \preccurlyeq B}\ (\textsc{SubConv})$$

$$\frac{\alpha \geq \beta \qquad i \leq j}{\Gamma \vdash_+ \square^\alpha_i \preccurlyeq \square^\beta_j}\ (\textsc{SubSort}) \qquad \frac{\Gamma \vdash_+ M' N : K \qquad \Gamma \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ M\,N \preccurlyeq M'\,N}\ (\textsc{SubApp})$$

$$\frac{\Gamma, x : A \vdash_+ M \preccurlyeq M'}{\Gamma \vdash_+ \lambda x : A.\, M \preccurlyeq \lambda x : A.\, M'}\ (\textsc{SubLam})$$

$$\frac{\Gamma \vdash_+ \Pi x : A.\, B : \square^\gamma_i \qquad \Gamma \vdash_+ A' \preccurlyeq A \qquad \Gamma, x : A' \vdash_+ B \preccurlyeq B'}{\Gamma \vdash_+ \Pi x : A.\, B \preccurlyeq \Pi x : A'.\, B'}\ (\textsc{SubPi})$$

$$K ::= \square^\gamma_i \mid \Pi x : A.\, K$$

Fig. 10.3 : Règles de sous-typage pour CC^+_ω

$$\frac{\Gamma \vdash_+ M : A \qquad \Gamma \vdash_+ A \preccurlyeq B}{\Gamma \vdash_+ M : B}\ (\textsc{Conv}^+) \qquad \frac{(\alpha,\beta) \in \mathcal{D}_\square}{\Gamma \vdash_+ \square_i^\alpha : \square_{i+1}^\beta}\ (\textsc{Sort}^+)$$

$$\frac{(x, A) \in \Gamma \qquad \Gamma \vdash_+}{\Gamma \vdash_+ x : A}\ (\textsc{Var}^+) \qquad \frac{\Gamma \vdash_+ A : \square_i^\gamma \qquad x \notin \mathrm{Var}(\Gamma)}{\Gamma,\, x : A \vdash_+}\ (\textsc{Context}^+)$$

$$\frac{\Gamma \vdash_+ M : \Pi x : A.\, B \qquad \Gamma \vdash_+ N : A}{\Gamma \vdash_+ M\, N : B[x := N]}\ (\textsc{App}^+)$$

$$\frac{\Gamma, x : A \vdash_+ M : B}{\Gamma \vdash_+ \lambda x : A.\, M : \Pi x : A.\, B}\ (\textsc{Lam}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\to(\gamma) = (\alpha,\beta)}{\Gamma \vdash_+ A \to B : \square_i^\gamma}\ (\textsc{Arrow}^+)$$

$$\frac{\Gamma \vdash_+ A : \square_i^\alpha \qquad \Gamma, x : A \vdash_+ B : \square_i^\beta \qquad \mathcal{D}_\Pi(\gamma) = (\alpha,\beta)}{\Gamma \vdash_+ \Pi x : A.\, B : \square_i^\gamma}\ (\textsc{Pi}^+)$$

Fig. 10.4 : Règles de typage pour CC_ω^+

10.4 Le calcul Trocq

L'étape finale de la généralisation annoncée consiste à construire un analogue aux traductions de paramétricité disponibles dans les Systèmes de Typage Purs, mais pour la théorie des types annotée de § 10.3. Cet analogue est orienté vers le transfert de preuve et donc conçu pour *synthétiser* la sortie de la traduction à partir de son entrée, plutôt que pour *vérifier* que certaines paires de termes sont en relation. Cependant, la division de l'interprétation des univers en un treillis de structures relationnelles possibles signifie que le terme source de la traduction ne suffit pas à caractériser le résultat souhaité : la traduction doit recevoir des informations supplémentaires sur le résultat escompté de la traduction. Dans le calcul de Trocq, cette information supplémentaire est un type de CC_ω^+.

On définit donc les *contextes* de Trocq comme des listes de quadruplets :

$$\Delta ::= \varepsilon \mid \Delta,\ x \mathbin{@} A \sim x' \mathbin{\because} x_R \quad \text{où } A \in \mathcal{T}_{CC_\omega^+}$$

Nous introduisons également une fonction γ de conversion des contextes Trocq vers les contextes de CC_ω^+ :

$$\begin{aligned} \gamma(\varepsilon) &= \varepsilon \\ \gamma(\Delta,\ x \mathbin{@} A \sim x' \mathbin{\because} x_R) &= \gamma(\Delta),\ x : A \end{aligned}$$

Or, un jugement Trocq est une relation quaternaire, qui se note et se lit de la façon suivante :

$$\Delta \vdash_t M \mathbin{@} A \sim M' \mathbin{\because} M_R$$

Dans le contexte Δ, le terme M de type annoté A se traduit en M', car M_R.

Les jugements de Trocq sont définis par les règles de la Figure 10.5. Cette définition utilise une fonction d'affaiblissement pour les témoins de paramétricité, définie comme suit.

$$\frac{(\alpha,\beta)\in\mathcal{D}_{\square}}{\Delta\vdash_t \square_i^\alpha \mathbin{@} \square_{i+1}^\beta \sim \square_i^\alpha \mathbin{\because} p_{\square_i}^{\alpha,\beta}}\ (\textsc{TrocqSort})$$

$$\frac{(x,A,x',x_R)\in\Delta \qquad \gamma(\Delta)\vdash_+}{\Delta\vdash_t x \mathbin{@} A \sim x' \mathbin{\because} x_R}\ (\textsc{TrocqVar})$$

$$\frac{\Delta\vdash_t M \mathbin{@} \Pi x:A.B \sim M' \mathbin{\because} M_R \qquad \Delta\vdash_t N \mathbin{@} A \sim N' \mathbin{\because} N_R}{\Delta\vdash_t M\,N \mathbin{@} B[x:=N] \sim M'\,N' \mathbin{\because} M_R\,N\,N'\,N_R}\ (\textsc{TrocqApp})$$

$$\frac{\Delta\vdash_t A \mathbin{@} \square_i^\alpha \sim A' \mathbin{\because} A_R \qquad \Delta, x \mathbin{@} A \sim x' \mathbin{\because} x_R \vdash_t M \mathbin{@} B \sim M' \mathbin{\because} M_R}{\Delta\vdash_t \lambda x:A.M \mathbin{@} \Pi x:A.B \sim \lambda x':A'.M' \mathbin{\because} \lambda x\,x'\,x_R.M_R}\ (\textsc{TrocqLam})$$

$$\frac{\Delta\vdash_t A \mathbin{@} \square_i^\alpha \sim A' \mathbin{\because} A_R \qquad \Delta\vdash_t B \mathbin{@} \square_i^\beta \sim B' \mathbin{\because} B_R \qquad (\alpha,\beta)=\mathcal{D}_{\to}(\delta)}{\Delta\vdash_t A\to B \mathbin{@} \square_i^\delta \sim A'\to B' \mathbin{\because} p_{\to}^\delta\,A_R\,B_R}\ (\textsc{TrocqArrow})$$

$$\frac{\begin{array}{c}\Delta\vdash_t A \mathbin{@} \square_i^\alpha \sim A' \mathbin{\because} A_R \\ \Delta, x \mathbin{@} A \sim x' \mathbin{\because} x_R \vdash_t B \mathbin{@} \square_i^\beta \sim B' \mathbin{\because} B_R\end{array} \qquad (\alpha,\beta)=\mathcal{D}_{\Pi}(\delta)}{\Delta\vdash_t \Pi x:A.B \mathbin{@} \square_i^\delta \sim \Pi x':A'.B' \mathbin{\because} p_{\Pi}^\delta\,A_R\,B_R}\ (\textsc{TrocqPi})$$

$$\frac{\Delta\vdash_t M \mathbin{@} A \sim M' \mathbin{\because} M_R \qquad \gamma(\Delta)\vdash_+ A \preccurlyeq B}{\Delta\vdash_t M \mathbin{@} B \sim M' \mathbin{\because} \Downarrow_B^A M_R}\ (\textsc{TrocqConv})$$

Fig. 10.5 : Règles de Trocq

$$\Downarrow_{\square_{i'}^{\alpha'}}^{\square_i^\alpha} t_R := \downdownarrows_{\alpha'}^{\alpha} t_R \qquad\qquad \Downarrow_{A'\,M'}^{A\,M} N_R := \Downarrow_{A'}^{A} M\,M'\,N_R$$

$$\Downarrow_{\lambda x:A'.B'}^{\lambda x:A.B} M\,M'\,N_R := \Downarrow_{B'[x:=M']}^{B[x:=M]} N_R$$

$$\Downarrow_{\Pi x:A'.B'}^{\Pi x:A.B} M_R := \lambda x\,x'\,x_R.\Downarrow_{B'}^{B}(M_R\,x\,x'(\Downarrow_{A}^{A'} x_R)) \qquad\qquad \Downarrow_{A'}^{A} M_R := M_R$$

Fig. 10.6 : Affaiblissement des témoins de paramétricité

Définition 10.4.1
Pour tous $p,q\in\{0,1,2_a,2_b,3,4\}$, tels que $p\geq q$, on définit l'application $\downarrow_q^p\colon \mathsf{M}_p\to\mathsf{M}_q$ comme la fonction oubliant les champs de M_p qui ne sont pas présents dans M_q. Pour tous $\alpha,\beta\in\mathcal{A}$, tels que $\alpha\geq\beta$, la fonction $\downdownarrows_\beta^\alpha\colon \mathrm{Param}^\alpha\,A\,B\to\mathrm{Param}^\beta\,A\,B$ est définie comme :

$$\downdownarrows_{(p,q)}^{(m,n)}(R;(M^{\to},M^{\leftarrow})) \quad := \quad (R;(\downarrow_p^m M^{\to},\downarrow_q^n M^{\leftarrow})).$$

La fonction d'affaiblissement des témoins de paramétricité est définie en Figure 10.6 en étendant la fonction $\downdownarrows_\beta^\alpha$ à toutes les paires de types pertinentes de CC_ω^+, c'est-à-dire que $\Downarrow_U^T$ est définie pour $T,U\in\mathcal{T}_{CC_\omega^+}$ dès que $T\preccurlyeq U$.

Un théorème d'abstraction relie les jugements de Trocq bien formés et le typage dans CC_ω^+ :

Théorème 10.4.1 (Théorème d'abstraction de Trocq)

$$\frac{\gamma(\Delta) \vdash_+ \qquad \gamma(\Delta) \vdash_+ M : A \qquad\qquad \Delta \vdash_t M \mathbin{@} A \sim M' \mathbin{\because} M_R \qquad \Delta \vdash_t A \mathbin{@} \square_i^\alpha \sim A' \mathbin{\because} A_R}{\gamma(\Delta) \vdash_+ M' : A' \qquad \text{et} \qquad \gamma(\Delta) \vdash_+ M_R : \mathrm{rel}(A_R)\, M\, M'}$$

Démonstration. Par induction sur la dérivation $\Delta \vdash_t M \mathbin{@} A \sim M' \mathbin{\because} M_R$. ■

On note que le type A dans l'hypothèse de typage $\gamma(\Delta) \vdash_+ M : A$ du théorème d'abstraction est exactement l'information supplémentaire transmise à la traduction. Cette dernière peut donc également être considérée comme un algorithme d'inférence, qui déduit des annotations pour la sortie de la traduction à partir de celles de l'entrée.

Remarque 10.4.2
Comme, par définition de $p_\square^{\alpha,\beta}$ (Équation 9.1), on a $\vdash_t \square^\alpha \mathbin{@} \square^\beta \sim \square^\alpha \mathbin{\because} p_\square^{\alpha,\beta}$, en appliquant le Théorème 10.4.1 avec $\gamma(\Delta) \vdash_+ A : \square^\alpha$, on obtient :

$$\frac{\gamma(\Delta) \vdash_+ A : \square^\alpha \qquad \Delta \vdash_t A \mathbin{@} \square^\alpha \sim A' \mathbin{\because} A_R}{\gamma(\Delta) \vdash_+ A_R : \mathrm{rel}(p_\square^{\alpha,\beta})\, A\, A'}.$$

À présent, par la même définition, pour tout $\beta \in \mathcal{A}$, $\mathrm{rel}(p_\square^{\alpha,\beta}) = \mathrm{Param}^\alpha$, donc $\gamma(\Delta) \vdash A_R : \mathrm{Param}^\alpha\, A\, A'$, comme prévu par l'annotation $A : \square^\alpha$ en entrée de la traduction.

Remarque 10.4.3
En appliquant la Remarque 10.4.2 avec $\vdash_+ \square^\alpha : \square^\beta$, on obtient :

$$\vdash_+ p_\square^{\alpha,\beta} : \mathrm{Param}^\beta\, \square^\alpha\, \square^\alpha$$

comme prévu, pourvu que $(\alpha, \beta) \in \mathcal{D}_\square$.

10.5 Constantes

Les applications concrètes nécessitent d'étendre Trocq avec des constantes. Les constantes sont similaires aux variables, à ceci près qu'elles sont stockées dans un contexte global au lieu d'un contexte de typage. Une différence essentielle réside dans le fait qu'une constante peut se voir attribuer plusieurs types annotés différents dans CC_ω^+. Considérons par exemple une constante list, représentant le type des listes polymorphes. Comme $\mathrm{list}\ A$ est le type des listes dont les éléments sont de type A, elle peut être annotée avec le type $\square^\alpha \to \square^\alpha$ pour tout $\alpha \in \mathcal{A}$.

Chaque constante c déclarée dans l'environnement global est associée à une collection de types annotés possibles $T_c \subset \mathcal{T}_{CC_\omega^+}$. On exige que tous les types annotés possibles d'une même constante partagent le même effacement[2] vers CC_ω, c'est-à-dire :

$$\forall c, \forall A, \forall B, \quad A, B \in T_c \implies |A|^- = |B|^-$$

Par exemple, $T_{\mathrm{list}} = \{\square^\alpha \to \square^\alpha \mid \alpha \in \mathcal{A}\}$.

2 : Il s'agit d'une fonction $|\cdot|^-$ définie comme le retrait récursif de toutes les annotations sur les univers.

En outre, nous fournissons des traductions $\mathcal{D}_c(A)$ pour chaque type annoté possible A de chaque constante c dans le contexte global. Par exemple, la valeur $\mathcal{D}_{\text{list}}(\square^{(1,0)} \to \square^{(1,0)})$ est bien définie et égale à la traduction suivante :

$$\Big(\text{list}, \quad \lambda A\, A'\, A_R.\, (\,\text{List.All2}\; A_R\,;\, \text{List.map}\; \text{map}(A_R)\,)\Big)$$

où $\text{List.All2}\; A_R$ relie les listes dont les éléments sont reliés deux à deux par A_R, List.map est la fonction d'application standard sur les listes et la projection $\text{map}(A_R) : A \to A'$ est la fonction extraite du témoin $A_R : \text{Param}^{(1,0)}\; A\; A'$, soit $A_R : \Sigma R.\, A \to A'$. Une partie de ces traductions peut être générée automatiquement par affaiblissement.

Nous décrivons dans la Figure 10.7 les règles supplémentaires pour les constantes dans CC^{+}_{ω} et Trocq. On note que si le terme d'entrée contient des constantes, un mauvais choix d'annotation peut conduire à une traduction bloquée.

$$\frac{c \in \mathcal{C} \qquad A \in T_c}{\Gamma \vdash c : A}\ (\textsc{Const}^{+}) \qquad \frac{\mathcal{D}_c(A) = (c', c_R)}{\Delta \vdash c \mathbin{@} A \sim c' \mathbin{\because} c_R}\ (\textsc{TrocqConst})$$

Fig. 10.7 : Règles supplémentaires pour les constantes dans CC^{+}_{ω} et Trocq

11 Conclusion et perspectives

Les fonctionnalités du prototype de greffon présenté dans cette partie III peuvent être étendues dans plusieurs directions. Il serait particulièrement fructueux de le connecter à des outils capables d'automatiser la génération de preuves d'équivalence, tels que Pumpkin Pi [68]. D'autres améliorations, par exemple traitant le cas de la sorte imprédicative de Coq, posent des questions d'implémentation non triviales, liées à la gestion du polymorphisme d'univers dans Coq. Nous allons maintenant examiner comment ce prototype, dans son état actuel, se compare à d'autres approches du transfert de preuve implémentées dans des assistants de preuve interactifs, énumérées par ordre chronologique dans le tableau récapitulatif 11.1. Pour chacun de ces outils, le tableau indique si une caractéristique donnée est disponible (vert), non disponible (orange foncé) ou seulement partiellement disponible (jaune).

[68] : Ringer *et coll.* (2021), « Proof repair across type equivalences »

Dans le contexte de la théorie des types, l'idée que le contenu calculatoire des isomorphismes de types peut être utilisé pour le transfert de preuve apparaît déjà dans [69]. Le premier rapport d'implémentation d'un outil basé sur cette idée apparaît peu après [70]. Implémentée dans un méta-langage et basée sur la réécriture de preuves, cette traduction heuristique produit un terme de preuve candidat à partir d'un terme de preuve donné, sans aucune garantie formelle, pas même celle d'être bien typé. La réécriture généralisée [43], qui généralise la réécriture setoïdale aux préordres, est également une variante du transfert de preuve, bien qu'elle reste dans le même type. En tant que telle, elle permet en particulier la réécriture sous les lieurs. La restriction aux relations homogènes exclut cependant les applications aux relations d'équivalence quasi-partielle (QPER) [71], ou au changement de type de représentation des données.

[69] : Barthe *et coll.* (2001), « Type Isomorphisms and Proof Reuse in Dependent Type Theory »

[70] : Magaud (2003), « Changing Data Representation within the Coq System »

[71] : Krishnaswami *et coll.* (2013), « Internalizing Relational Parametricity in the Extensional Calculus of Constructions »

À notre connaissance, les autres méthodes de transfert de preuve traitent toutes du cas des relations hétérogènes. Elles peuvent donc également être utilisées pour le cas homogène, bien que ce cas particulier soit rarement mis en avant. Les greffons Coq Effective Algebra Library (CoqEAL) [45, 72] et Isabelle/HOL transfer [73-76] sont les premiers à utiliser des méthodes basées sur la parametricité pour le transfert de preuve, motivées par le raffinement de structures de données conçues pour la preuve vers des homologues optimisés pour le calcul. Avec une généralisation ultérieure de l'approche de CoqEAL [77], ces outils traitent le cas d'un transfert entre un sous-type d'un certain type A et un quotient d'un certain type B, c'est-à-dire le cas des QPER triviales dans lesquelles le morphisme zig-zag est une surjection partielle de A à B.

[73] : Lammich (2013), « Automatic Data Refinement »

[74] : Haftmann *et coll.* (2013), « Data Refinement in Isabelle/HOL »

[75] : Huffman *et coll.* (2013), « Lifting and Transfer : A Modular Design for Quotients in Isabelle/HOL »

[76] : Lammich *et coll.* (2019), « Automatic Refinement to Efficient Data Structures : A Comparison of Two Approaches »

Les deux colonnes suivantes du tableau concernent le transfert de preuve en présence du principe d'univalence, soit axiomatique dans le cas de la paramétricité univalente [14], soit calculatoire dans le cas de [78]. Les ingrédients clés de la paramétricité univalente sont déjà présents dans un travail antérieur apparemment non publié [79], implémenté à l'aide d'un ancêtre désuet de la bibliothèque MetaCoq [46].

[77] : Zimmermann *et coll.* (2015), « Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant »

[78] : Angiuli *et coll.* (2021), « Internalizing representation independence with univalence »

Le Tableau 11.1 indique les outils qui peuvent effectuer un transfert le long de *relations hétérogènes*, car il s'agit d'une condition préalable à la modification de la représentation des types, et ceux qui fonctionnent en prouvant un lemme d'implication *interne*, par opposition à une traduction monolithique d'un terme de preuve d'entrée. Nous empruntons la terminologie utilisée dans [14], dans

[79] : Anand *et coll.* (2017), *Revisiting Parametricity : Inductives and Uniformity of Propositions*

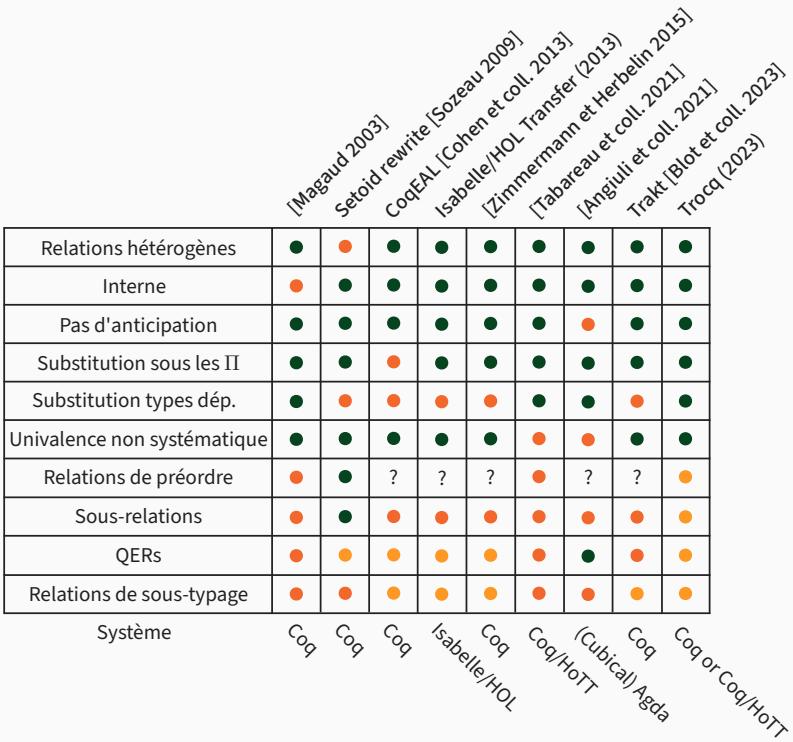


Fig. 11.1 : Comparaison d'outils de transfert de preuve

laquelle l'*anticipation* fait référence à la nécessité de définir une structure dédiée pour la signature à transporter. Les *lieurs* peuvent empêcher le transfert, de même que *les types dépendants*. Ces derniers sont récupérés en présence de l'univalence. La première publication [80] sur la traduction de la paramétricité univalente suggère que la traduction ne requiert pas l'axiome lors de la traduction des termes du fragment F^{ω}. Cependant, Trocq peut s'en débarrasser pour une classe de termes strictement plus large. Enfin, le tableau indique les approches qui peuvent traiter les *relations de quasi-équivalence* (QER) et les relations de sous-typage (explicite).

[80] : Tabareau *et coll.* (2018), « Equivalences for free : univalent parametricity for effective transport »

Dans son état actuel, le greffon Trocq peut déjà s'attaquer à la bureaucratie du transfert de preuves formelles de pointe, en mathématiques abstraites, en vérification de programmes, ou les deux [81]. Nous espérons que notre travail, une fois mis en production, permettra d'appliquer le même lemme à une grande variété de types différents : types isomorphes, sous-types et types quotients. De plus, ce cadre ouvre la voie à une plus large gamme d'extensions, par exemple en réalisant une unification modulo à la fois la réécriture généralisée et les relations de transfert hétérogènes, résolvant potentiellement les problèmes parfois appelés *alignement de concept*. Nous concluons par deux problèmes concrets et difficiles à résoudre dans les assistants de preuve interactifs, que de telles extensions pourraient aider à résoudre. Le premier est l'identification canonique d'objets dans les types par des entiers naturels, par exemple $\{x : \mathbb{R} \mid \exists n : \mathbb{N}, x = \iota(n)\}$, *etc.* Le second est l'identification de différentes constructions paramétriques, qui *coïncident* pour certaines classes spécifiques de paramètres, par exemple l'anneau $\mathbb{Z}/q\mathbb{Z}$, défini pour tous les entiers $q > 0$, et le champ de Galois $\mathbb{F}_q$, défini lorsque $q = p^k$, se trouvent être canoniquement isomorphes si et seulement si q est premier.

[81] : Allamigeon *et coll.* (2023), « A Formal Disproof of Hirsch Conjecture »

Implémentation d'outils de pré-traitement en Coq-Elpi

Introduction

Dans les deux parties précédentes, nous avons présenté deux solutions pour traiter sous un angle particulier le problème général du transfert de preuve. La première, Trakt, a été conçue sur la base de `zify`, un outil de pré-traitement *ad hoc* ciblant la tactique `lia`. La seconde, Trocq, est une relation de paramétricité, une méthode plus générale avec une approche plus théorique. Dans cette partie, nous nous intéressons à l'implémentation de ces outils, c'est-à-dire tout le travail effectué pour passer d'un algorithme de pré-traitement théorique à un greffon utilisable dans des preuves Coq concrètes. Dans ce premier chapitre, nous nous plaçons à l'échelle de l'architecture logicielle mise en place pour bâtir ces greffons. Dans le chapitre suivant, nous détaillons en particulier les problèmes qui surviennent lors de l'implémentation d'un greffon de paramétricité comme Trocq.

Architecture logicielle d'un greffon de pré-traitement

12

12.1 Base de connaissances utilisateur 109
12.1.1 Utilisation des bases de données Coq-Elpi 109
12.1.2 Stockage des termes Coq . . 110
12.2 Traversée du but initial . . 110
12.2.1 Une tactique de traduction en Coq-Elpi 111
12.2.2 Trakt : leçons d'une première tentative 112

Bien que différents dans leur stratégie de pré-traitement des buts, les deux outils conçus dans cette thèse s'attaquent à un problème similaire et ont des besoins similaires. Nous avons donc imaginé une architecture logicielle commune aux deux greffons, illustrée en Figure 12.1[1] et présentée dans ce chapitre. Premièrement, l'outil de pré-traitement a besoin d'être informé des modifications qu'il doit appliquer au but initial. L'utilisateur doit donc lui communiquer des informations avant de passer en mode preuve. Pour ce faire, notre architecture intègre une base de connaissances à l'outil, avec des commandes Coq pour y ajouter des données (§ 12.1). Deuxièmement, il faut que le greffon implémente un algorithme de pré-traitement accessible depuis le mode preuve de Coq. L'architecture comporte donc une tactique qui implémente une traduction prenant en entrée le but initial et éventuellement des paramètres, pour construire, en utilisant la base de connaissances, le but associé après pré-traitement accompagné d'un terme de preuve qui justifie la substitution (§ 12.2). Après le passage de l'outil de pré-traitement, l'utilisateur n'a plus qu'à prouver le but associé, idéalement avec un autre outil de preuve automatique.

1 : Les parties en orange correspondent à la partie du scénario de preuve à implémenter dans le greffon de pré-traitement.

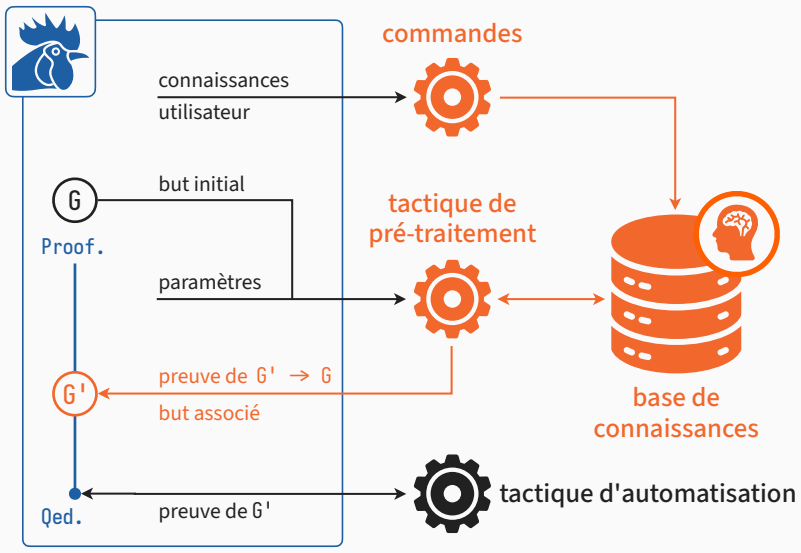


Fig. 12.1 : Architecture globale des greffons de pré-traitement développés pendant cette thèse

Par exemple, pour pré-traiter le but initial suivant, qu'il s'agisse de Trakt ou de Trocq, il faut que l'utilisateur indique comment il souhaite traduire le type `int`, la constante `0`, l'opération `-`, ainsi que l'égalité :

```
forall (x y : int), x - y = 0
```

Ces informations sont les connaissances utilisateur, données *via* des commandes et stockées dans la base de connaissances. Ensuite, l'utilisateur exécute la tactique en lui fournissant des paramètres propres à chaque outil. La tactique traverse le but initial et construit le but associé ainsi qu'une preuve d'implication entre les deux permettant de changer le contexte de preuve pour ne plus avoir que le but associé à prouver. Un but associé plausible pourrait être le suivant, par exemple avec `lia` comme tactique d'automatisation, en associant `int` à `Z`,

l'égalité à elle-même et les différentes valeurs dans `int` à leurs équivalents dans `Z` :

```
forall (x y : Z), x - y = 0
```

Puisque les deux greffons ont été développés en Coq-Elpi, nous prenons des exemples alternativement dans Trakt et dans Trocq pour illustrer nos propos. Ce choix technique a été fait pour plusieurs raisons. D'abord, ce méta-langage offre un haut niveau d'abstraction vis-à-vis de la syntaxe des termes Coq. En effet, l'encodage des termes en HOAS se marie très bien avec les constantes universelles du langage Elpi, offrant la possibilité de manipuler des λ-termes contenant des lieurs sans jamais avoir à maintenir des indices de De Bruijn corrects. De plus, l'utilisation des variables du méta-langage pour représenter les variables d'unification de Coq permet de manipuler facilement des termes à trous, situation fréquente lors de la réification de termes écrits dans la syntaxe de surface de Coq, voire de forger des termes à trous dans les méta-programmes et déléguer une partie du travail à Coq en exploitant son élaborateur et son vérificateur de typage. Ensuite, l'enrobage que constitue Coq-Elpi autour du langage Elpi fournit une véritable boîte à outils de méta-programmation en Coq, avec diverses API utiles : création de commandes et tactiques, contrôle fin sur le comportement de l'unification et de la conversion, déclarations de termes et de types, manipulation des univers, *etc.* Enfin, le langage Elpi est construit autour du paradigme de la programmation logique, qui se prête très bien à l'implémentation d'algorithmes récursifs parcourant des arbres de syntaxe, tels que ceux développés pendant cette thèse. *A fortiori*, lorsqu'ils sont basés sur des règles d'inférence, comme l'algorithme au cœur de Trocq, on obtient une correspondance directe entre le code et la description de l'algorithme sur papier.

12.1 Base de connaissances utilisateur

Les deux greffons présentés dans cette thèse ont l'avantage d'être extensibles, c'est-à-dire qu'ils permettent à l'utilisateur de personnaliser le pré-traitement grâce à l'ajout de données supplémentaires avant la traduction du but. Dans le cas de Trakt, il s'agit des plongements entre types, relations, symboles, *etc.* Dans le cas de Trocq, ce sont les témoins de paramétricité. Cette section présente l'utilisation des bases de données Coq-Elpi comme base de connaissances utilisateur (§ 12.1.1), ainsi que les aspects techniques intéressants lors du stockage de termes Coq dans une base de données au niveau méta (§ 12.1.2).

12.1.1 Utilisation des bases de données Coq-Elpi

Les greffons Trakt et Trocq utilisent plusieurs bases de données pour organiser les informations à stocker. Comme expliqué en § 4.2.1, en Coq-Elpi, une base de données est une série d'instances de faits, c'est-à-dire de prédicats toujours vrais dont les arguments sont les données que l'on souhaite stocker. Ainsi, Trakt enregistre par exemple les plongements de symboles comme des instances d'un prédicat `symbol` prenant en argument les données nécessaires, listées en § 6.1.3. De la même manière, chaque nature différente d'information à stocker est associée à un prédicat particulier dans la base de données, de sorte à structurer le code et donner des messages d'erreur informatifs lorsqu'une donnée est manquante. Dans Trocq, les axiomes ont également un prédicat dédié, afin de pouvoir utiliser le greffon sans les ajouter automatiquement, et ainsi pouvoir traduire des

buts qui ne les nécessitent pas. Des commandes sont définies pour pouvoir ajouter proprement des données. Par exemple, dans le cas d'un témoin `u` de l'axiome d'univalence, la commande à utiliser dans TROCQ est la suivante :

```
Param Register Univalence u.
```

12.1.2 Stockage des termes COQ

La plupart des outils d'automatisation des preuves travaillent avec une base de connaissances contenant des constantes sélectionnées dans le contexte. Ainsi, la tactique `ring` utilise une base d'instances d'une structure algébrique d'anneau et la tactique `auto` effectue une recherche de preuve à partir d'une banque de lemmes choisis par l'utilisateur. La conception de ces bases de connaissances est difficile car le stockage des termes COQ au niveau méta pose des problèmes subtils. En effet, la syntaxe de surface de COQ masque différentes informations à travers des notations et éléments implicites (arguments, instances d'univers, *etc.*). Or, ces trous laissés dans les termes sont en réalité des variables qui peuvent interagir avec l'état global de l'assistant de preuve, *via* les contraintes d'unification ou le graphe de contraintes d'univers par exemple, dès qu'une unification est lancée entre un terme à trous et un autre terme, ce qui arrive systématiquement lors de la lecture d'une base de données indexée par de tels termes. Le stockage de termes à trous dans une base de données n'est donc pas un choix robuste. Pourtant, cette légèreté syntaxique est cruciale du point de vue de l'utilisateur, car l'annotation manuelle de tous les termes dans leur intégralité rendrait l'usage de l'assistant de preuve très pénible. Dans les cas où ces trous laissés dans les termes sont inférables à partir du reste du terme, on peut laisser l'élaborateur de COQ se charger de ce travail et stocker le terme complet obtenu. Toutefois, l'élaborateur complète toujours un terme en se plaçant dans un certain contexte qui peut varier entre le moment où la base de connaissances est remplie et le moment où la tactique de pré-traitement s'exécute. Ainsi, deux termes en apparence unifiables peuvent ne pas l'être dans leur forme complète, si le terme recherché en base de données est syntaxiquement différent de celui qui a été enregistré auparavant en passant par l'élaborateur. Rendre systématique l'usage de l'élaboration sur les arguments d'une commande COQ destinée à l'enregistrement de termes dans une base de données au niveau méta est par conséquent une solution naïve qui apporte des erreurs difficiles à tracer sur le long terme. En pratique, un compromis intéressant, pour laisser à l'utilisateur une certaine liberté dans la syntaxe tout en stockant des termes avec une indépendance maximale vis-à-vis de l'état global de COQ, est de stocker en base de données uniquement des références globales, c'est-à-dire des termes identifiés par leur nom enregistré dans COQ.[2].

2 : Il s'agit des constantes, des types inductifs et des constructeurs.

12.2 Traversée du but initial

Après la conception de la base de connaissances et son remplissage par l'utilisateur, l'outil de pré-traitement effectue une traduction du but initial. Cette traduction prend la forme d'un algorithme récursif défini par induction sur la syntaxe. Cette section présente la structure générale d'une tactique qui implémente une telle traduction (§ 12.2.1), ainsi que les leçons à tirer de l'implémentation de TRAKT (§ 12.2.2).

12.2.1 Une tactique de traduction en Coq-Elpi

Le rôle de la tactique de traduction est de traverser le but initial afin de construire à la fois le but associé et une preuve que substituer ce dernier au but initial est une opération valide. Ensuite, elle doit appliquer cette preuve, pour laisser à l'utilisateur un contexte de preuve avec le but associé, sans aucune obligation de preuve supplémentaire.

Structure de la tactique Dans Trakt comme dans Trocq, la tactique principale de traduction lit le but d'entrée et appelle le prédicat principal de traversée du but, qui produit en sortie le but associé ainsi qu'une preuve de pré-traitement. On se sert ensuite de l'opération `refine`, qui effectue essentiellement la même action que la tactique Coq du même nom, c'est-à-dire appliquer un terme de preuve à trous, les trous représentant les nouvelles obligations de preuve. Dans notre cas, on s'attend à un unique trou, ayant le type du but associé et étant idéalement prouvable grâce à une tactique de preuve automatique. L'appel est donc le suivant :

```
refine {{ lp:Proof (_ : lp:EndGoalTy) }} InitialGoal NewGoals
```

La variable `Proof` contient la preuve d'implication et `EndGoalTy` est le but associé, tous deux générés par la traduction.

Prédicat récursif de traduction L'implémentation de l'algorithme de traduction constitue le point central d'une tactique de pré-traitement. Il s'agit d'un prédicat récursif qui prend en entrée au moins un terme qui sera lors du premier appel le but initial, et rend en sortie au moins deux termes, le but associé et la preuve de validité de la substitution.

Les prédicats Elpi peuvent être définis en plusieurs instances, l'unification à la Prolog de ce méta-langage permettant de sélectionner l'instance correspondant au cas que l'on doit traiter. Si les arguments de la tête de l'instance ne s'unifient pas avec les arguments de l'appel courant au prédicat, on passe à l'instance suivante, et ainsi de suite jusqu'à l'échec.[3] Dans le cas d'un prédicat défini par induction sur la syntaxe d'un terme Coq, on écrit au moins une instance par construction disponible dans le langage. Plusieurs sous-cas pour une même construction peuvent être distingués en exécutant des prédicats de test au début du corps de chaque instance. En effet, si l'un de ces prédicats échoue, l'instance suivante sera sélectionnée. Pour contraindre le programme à n'explorer qu'une seule instance, on peut ajouter une *coupure* (avec le caractère `!`) après ces prédicats de test. Tout cela permet de mieux organiser le code et de définir l'ordre de priorité dans l'essai des différents cas disponibles.

3 : Une bonne pratique est de toujours définir une dernière instance qui attrape tous les cas restants et échoue avec un message d'erreur.

Par ailleurs, pour appliquer dans Coq un terme de preuve le plus complet possible lors de l'appel à `refine`, il est nécessaire de faire passer la preuve par un prédicat de typage et/ou d'élaboration afin de boucher les derniers trous qui ne représentent pas une preuve à venir mais une annotation de type laissée implicite dans le prédicat de traduction. Ceci permet de déléguer une partie du travail à Coq et d'écrire les preuves d'une façon plus naturelle lors du développement du greffon.

12.2.2 Trakt : leçons d'une première tentative

Le premier prototype implémenté en suivant l'architecture logicielle présentée plus tôt est Trakt, présenté dans cette thèse en partie I. Le greffon améliore le pré-traitement par canonisation incarné précédemment par la tactique `zify`, en effectuant une traduction similaire mais étendue à des buts de la famille SMT. L'implémentation de Trakt a permis d'identifier des patrons de conception intéressants à garder à l'esprit pour de futurs projets en Coq-Elpi. Elle présente quelques imperfections, mais atteint globalement son objectif, comme le montre son intégration à la bibliothèque SMTCoq avec succès. Cette sous-section en effectue un bilan.

Encodage des termes en HOAS Une première fonctionnalité utile de Coq-Elpi est l'encodage des termes en HOAS. En effet, grâce à cet encodage, on peut exprimer les contextes comme des fonctions dans le méta-langage. Par exemple, un contexte $C[\cdot]$ est représenté par une variable `C` de type `term → term`. Compléter le contexte avec un terme `x` revient alors à effectuer une application fonctionnelle `C x` ; le mettre à jour en passant sous un nouveau nœud, par exemple une fonction `f` à un argument, se fait en créant une nouvelle méta-fonction `C'` :[4]

```
pi x\ C' x = C (app [f, x])
```

Dans cet encodage, les lieurs sont l'association d'un terme représentant le type de la variable liée et d'une méta-fonction représentant le corps de la fonction dans le cas d'une abstraction ou bien le codomaine dans le cas d'un produit dépendant. Traverser un tel lieur se fait en créant une variable locale fraîche par le même procédé que ci-dessus. Tous les termes calculés à partir de l'application de la méta-fonction sont exprimés en fonction de cette variable, permettant de maintenir un terme clos tout au long du processus. Ces termes étant également des méta-fonctions, il suffit de rajouter un constructeur de lieur pour obtenir un terme Coq bien formé. Par exemple, voici un extrait de l'instance du prédicat principal de Trakt chargée de pré-traiter les produits dépendants :[5]

```
preprocess (prod N T F) /* ... */ :- !,
  @pi-decl N T x\ preprocess (F x) /* ... */ (F' x) (PF x),
  % ...
```

En une ligne de code, on introduit une variable fraîche `x` dans le domaine `T` pour effectuer un appel récursif sur le codomaine `F x` et récupérer le codomaine associé `F' x` ainsi qu'une preuve `PF x`. Le reste du prédicat retravaille ces termes pour obtenir le produit dépendant associé et la preuve de pré-traitement.

Enfin, l'abstraction d'un sous-terme t dans un terme u en une fonction f telle que $u \equiv f\,t$ est une tâche simple en Coq-Elpi. En effet, il existe un prédicat `copy` dans la bibliothèque standard, initialement défini comme une identité profonde : il traverse le terme dans toute sa structure et applique l'identité aux feuilles de l'arbre. En ajoutant localement une instance de `copy`, il est possible d'effectuer des substitutions : avec l'instance supplémentaire `copy X Y`, toute occurrence de `X` rencontrée pendant le parcours est remplacée par `Y` dans le terme de sortie du prédicat. En représentant t par une variable `T` et u par `U`, on peut abstraire t dans u en une seule ligne :

```
pi x\ copy T x ⇒ copy U (F x).
```

4 : Dans ce code, `x` est une constante universelle représentant une variable liée, permettant d'accéder au corps de la méta-fonction `C` pour définir `C'`. Du fait de son statut en Elpi, il est obligatoire de la rendre explicite comme argument de `C'`, afin qu'elle n'échappe pas à sa portée locale.

5 : Dans Coq-Elpi, les produits dépendants sont représentés par des termes `prod N T F`, où `N` est le nom d'affichage de la variable liée dans Coq, `T` est le domaine, et `F` est une méta-fonction contenant le codomaine.

On remplace toutes les occurrences de `T` dans `U` par une variable fraîche `x`, donnant une méta-fonction `F` qui représente l'abstraction souhaitée. On peut ensuite utiliser cette méta-fonction directement ou bien en faire une véritable fonction Coq en ajoutant un lieur par-dessus.

Personnalisation des termes Coq Une autre fonctionnalité cruciale de Coq-Elpi est de permettre au développeur de définir de nouveaux types et d'ajouter des constantes du type de son choix. Il est donc possible d'émuler le comportement des types algébriques que l'on trouve dans les langages fonctionnels plus classiques, en déclarant un nouveau type ainsi que différentes constantes pour représenter ses constructeurs. Voici un exemple de définition des entiers naturels en Elpi :

```
kind nat type.
type zero nat.
type succ nat → nat.
```

Ces valeurs n'étant pas des types algébriques à proprement parler, l'ensemble des constructeurs n'est ici pas fermé et le développeur ou l'utilisateur d'une base de code Elpi peut très bien ajouter de nouveaux constructeurs à un des types qui y sont définis. Ainsi, l'API de Coq-Elpi permettant de manipuler les termes Coq expose une représentation des termes *extensible*, ce qui est une excellente nouvelle pour la méta-programmation. En effet, il est possible de créer de nouveaux nœuds dans l'AST des termes Coq pour représenter diverses informations utiles lors de la manipulation de termes au niveau méta. Cette technique est utilisée dans Trakt, où deux nouveaux constructeurs sont ajoutés, dans un objectif d'annotation des termes Coq :

```
type prod2 name → term → term → (term → term) → term.
type cast term → term.
```

Le constructeur `prod2` est similaire au constructeur d'origine `prod` représentant le produit dépendant. Il diffère de ce dernier en ce qu'il contient un argument supplémentaire de type `term`. Grâce à cet argument, il devient possible de construire un produit dépendant contenant le domaine avant et après traduction. Cette information permet de savoir si le type d'une variable liée a changé pendant la traduction, et ainsi d'adapter la preuve en conséquence. Le constructeur `cast` est utilisé pour distinguer les fonctions de plongement ajoutées par l'algorithme de traduction des éventuelles fonctions de plongement déjà présentes dans le but initial avant traduction. Dans les deux cas, une procédure de nettoyage doit être exécutée après exploitation de ces informations, afin que le terme rendu à Coq puisse être traduit de nouveau dans la syntaxe native de l'assistant de preuve. En effet, pour mettre à disposition les termes Coq dans le méta-langage, le type Elpi utilisé pour les représenter est en bijection avec le type natif de Coq, et l'ajout de ces nouveaux constructeurs empêche à Coq-Elpi de jongler entre les deux représentations. Dans le cas de Trakt, cette procédure de nettoyage est très simple puisqu'il suffit d'effacer les annotations `cast` pour ne laisser que le terme sous-jacent et d'oublier l'un des deux domaines des nœuds `prod2` en les remplaçant par des nœuds `prod` pour obtenir de nouveau un terme Coq bien formé.

Réification des preuves Un schéma utile en méta-programmation identifié pendant l'implémentation de Trakt est la réification des preuves, soit la conception d'un type algébrique dans le méta-langage pour représenter les preuves. Cela permet une certaine abstraction par rapport aux termes Coq bruts représentant les

différents pas de preuve à effectuer pour passer du but initial au but associé. De plus, l'usage de la réification aide à la compréhension des fragments de preuve construits par le méta-programme lors de la traduction, et permet d'externaliser la génération du terme de preuve Coq final vers un autre prédicat que le principal, améliorant ainsi la lisibilité du code du greffon. Par exemple, les preuves au sein de Trakt sont d'abord générées dans un type Elpi `proof` dont les constructeurs représentent les différents pas de preuve possibles lors de la traduction, puis elles passent par une fonction qui reconstruit les fragments de preuve Coq correspondants avant de rendre le terme final à l'assistant de preuve.

Traversée du terme et maintien du contexte Malgré tous les points positifs identifiés, Trakt souffre de quelques lourdeurs, en particulier dans son prédicat principal de traduction. En effet, comme expliqué en § 6.2, la traduction doit distinguer les positions covariantes des positions contravariantes, pré-traiter différemment les termes selon que leur type est plongeable ou non, et garder en mémoire le contexte du terme courant afin de générer des preuves de réécriture s'appliquant à l'entièreté d'un atome logique, pour la composition de preuves. Pour toutes ces raisons, le prédicat possède des paramètres supplémentaires pour savoir dans quel cas il se trouve à chaque nœud du terme à traduire. Chaque ajout de fonctionnalité est une liste de nouveaux cas particuliers à traiter, ce qui ajoute davantage d'arguments et de tests au prédicat.[6] De plus, les différentes disjonctions de cas sont nombreuses et parfois profondes, ce qui les rend difficiles à exprimer en ajoutant simplement des instances au prédicat. Elles sont donc implémentées avec des branchements conditionnels, qui ne sont pas une construction native en Elpi. Les branchements sont effectués *via* un prédicat de test `if` explicite qui nuit légèrement à la lisibilité.

6 : Ici, le problème n'est pas spécialement dû au méta-langage, mais plutôt à l'approche *ad hoc* de conception du greffon, ciblant une utilité pratique à court terme pour l'utilisateur.

Trakt est un succès en pratique, mais son implémentation reflète son modèle *bottom-up*, centré d'abord sur un problème concret à résoudre, avec une certaine abstraction apportant au greffon son extensibilité et sa flexibilité, détaillée en § 6. Les quelques limites mentionnées font partie des raisons qui ont mené à la réflexion autour d'une solution plus générale, une conception *top-down* partant de la théorie vers la pratique. Les greffons de paramétricité développés dans les années 2010 incarnent la promesse d'un regroupement de tous les cas particuliers de la traduction dans un cadre général, avec une implémentation certes plus difficile mais plus élégante. Trocq pousse cette approche jusqu'à l'unification des différentes traductions de paramétricité elles-mêmes, apportant certaines subtilités dans l'implémentation, présentées au chapitre suivant.

13 Implémentation d'un greffon de paramétricité

Le second prototype conçu pendant cette thèse, Trocq, vise à résoudre le même problème de pré-traitement que l'outil Trakt présenté plus tôt, tout en étant capable de traduire n'importe quel terme de CC_ω. Trocq est en particulier un raffinement de la traduction de paramétricité univalente permettant de se passer de l'axiome d'univalence partout où un traitement manuel ne l'utilisant pas est possible. Ce raffinement passe par la conception d'une hiérarchie de Σ-types représentant des types de témoins de paramétricité plus ou moins riches, allant du témoin de paramétricité brute au témoin univalent. Les niveaux des témoins, appelés classes de paramétricité, sont alors contraints pendant la traversée du but afin de n'autoriser que les témoins suffisamment riches pour construire la preuve de pré-traitement du but initial. Les classes finales sont fixées après cette traversée par une procédure distincte, afin d'obtenir une traduction unique. Les contraintes ajoutées pendant le processus nécessitent un moyen syntaxique de représenter et manipuler les classes de paramétricité ; c'est pourquoi nous utilisons une théorie des types annotée où les univers sont accompagnés d'une classe de paramétricité. Enfin, dans un contexte d'utilisation concrète, la traduction fonctionne à partir d'une base de connaissances contenant des témoins de paramétricité prouvés par l'utilisateur sur différentes constantes pouvant apparaître dans les buts à traduire.

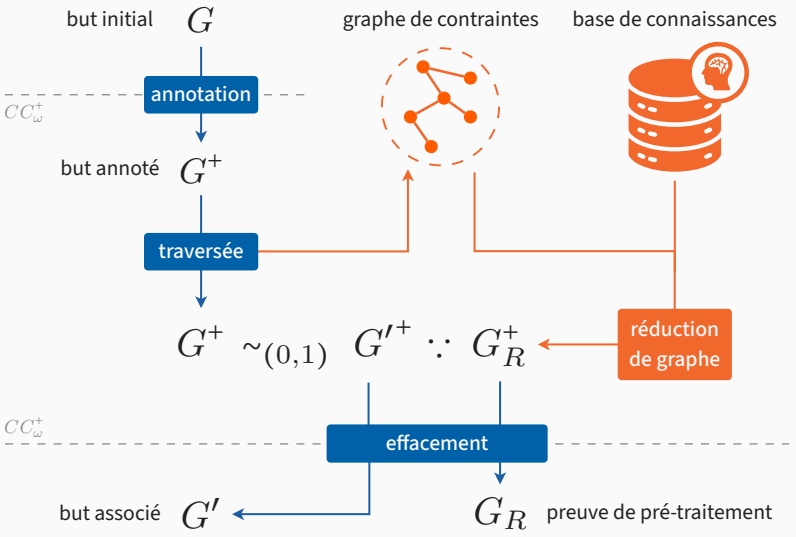


Fig. 13.1 : Fonctionnement de Trocq

Ainsi, le fonctionnement de l'implémentation de Trocq peut être résumé par la Figure 13.1. D'abord, le but initial G est annoté en ajoutant des variables fraîches sur tous les univers, soit des classes de paramétricité non contraintes, pour donner un but G^+ qui sera le but d'entrée à traverser. Cette traversée applique des règles structurelles et génère un nouveau but G'^+ ainsi qu'un témoin de paramétricité G^+_R liant les deux buts au niveau $(0,1)$, le plus petit niveau permettant par la suite d'extraire du témoin une preuve de $G'^+ \to G^+$. La traversée ajoute diverses contraintes sur les classes de paramétricité que l'on représente par un graphe de contraintes. Après la traversée, le graphe est réduit et les classes de paramétricité finales sont fixées. L'affectation des variables de classes de pa-

13.1 Générer et habiter la hiérarchie de paramétricité . . . 116
13.1.1 Génération de la hiérarchie et mise en place du greffon . . 116
13.1.2 Flexibilité des témoins de paramétricité 118
13.2 Implémentation de la relation de paramétricité . . . 119
13.2.1 Des règles d'inférence au programme logique 120
13.2.2 Fonctionnalités utiles de Coq-Elpi 122
13.3 Inférence de classes de paramétricité 123
13.3.1 Définition du problème . . . 123
13.3.2 Solution retenue dans Trocq 126
13.3.3 Implémentation 127
13.3.4 Affaiblissement et sous-typage 129
13.4 Polymorphisme d'univers . 129
13.4.1 Levée de l'ambiguïté typique 130
13.4.2 Univers algébriques et univers liés 132

ramétricité déclenche des requêtes dans la base de connaissances pour récupérer les éventuels témoins fournis par l'utilisateur, au niveau requis pour que la preuve globale soit bien typée. Enfin, une fois que tous les termes sont complets, on effectue un effacement pour récupérer un but associé G' et une preuve de pré-traitement à extraire de G_R, deux termes Coq valides.

Ce chapitre soulève différents points d'intérêt techniques dans le travail d'implémentation de Trocq. Tout d'abord, nous nous intéressons à la génération de la hiérarchie de paramétricité, cadre sur lequel repose tout le reste du greffon (§ 13.1). Ensuite, nous expliquons en quoi le paradigme de Coq-Elpi rapproche l'implémentation de ce cadre de sa description relationnelle (§ 13.2), puis nous décrivons l'implémentation de l'inférence de classes de paramétricité (§ 13.3). Enfin, nous identifions les limites de l'implémentation actuelle du polymorphisme d'univers dans Coq, une fonctionnalité cruciale dans l'implémentation de Trocq (§ 13.4).

13.1 Générer et habiter la hiérarchie de paramétricité

L'implémentation de Trocq utilise la méta-programmation dans l'écriture de la procédure de traversée du but, mais également dans la mise en place du greffon auparavant, pour automatiser la combinatoire engendrée par la hiérarchie de paramétricité. En effet, il existe 6 niveaux dans la hiérarchie et une classe de paramétricité est l'association d'un niveau covariant et d'un niveau contravariant. Il y a donc en tout 36 classes de paramétricité possibles et par conséquent autant de variantes pour chaque définition indexée par une classe : types des témoins de paramétricité, lemmes de paramétricité, fonctions d'affaiblissement, *etc.* Cette multiplicité est telle qu'il ne serait pas raisonnable d'écrire toutes les définitions manuellement. La formulation symétrique des témoins de paramétricité réduit cet effort manuel à 6 variantes, une pour chaque niveau de la hiérarchie. Le reste des termes peut alors être généré en combinant les 6 variantes de base. Dans cette section, nous présentons comment cette génération faite à l'aide de Coq-Elpi aide le développeur dans la mise en place du greffon mais aussi l'utilisateur dans la prise en charge des témoins de paramétricité ajoutés en base de connaissances.

13.1.1 Génération de la hiérarchie et mise en place du greffon

La présentation théorique de Trocq définit une hiérarchie de types de témoin de paramétricité pour lier deux types A et B, allant du type de témoin brut $A \to B \to \square$ au type de témoin univalent $\mathrm{Param}^{\top}\ A\ B$ du Théorème 9.1.1. L'implémentation doit définir un type de témoin pour chaque niveau possible de la hiérarchie et déclarer autant de versions des lemmes de paramétricité.

Types des témoins de paramétricité Les types de témoin $\mathrm{Param}^{(\alpha,\beta)}$ de la Définition 9.1.6 sont indexés par une classe de paramétricité (α, β) et définis comme la combinaison d'une relation avec deux témoins unilatéraux $\mathrm{M}_{\alpha}\ R$ et $\mathrm{M}_{\beta}\ R^{-1}$, chacun portant sur un sens de la relation R.

En Coq, les paires dépendantes peuvent être représentées d'une manière équivalente par des *enregistrements*. En effet, les enregistrements facilitent la manipulation de structures car ils sont plats[1] et il est possible de nommer les projections

1 : Tous les termes au même niveau dans la structure peuvent être obtenus avec le même nombre de projections.

qu'on leur applique.[2] L'implémentation de Trocq utilise donc l'enregistrement suivant comme représentation concrète de $\mathrm{Param}^{(\alpha,\beta)}$:

2 : Ce sont les *champs* de l'enregistrement.

```
Record Param^(α,β)@{i} (A B : Type@{i}) := {
  R : A → B → Type@{i};
  covariant : Map_α R;
  contravariant : Map_β (sym_rel R)
}.
```

Dans cette famille d'enregistrements, l'implémentation des types de témoins unilatéraux M est fidèle à la description théorique, en passant simplement d'une paire dépendante à un enregistrement et en nommant les champs. Ainsi, le type de témoins unilatéraux univalents M_4 est implémenté par l'enregistrement suivant :

```
Record Map_4@{i} {A B : Type@{i}} (R : A → B → Type@{i}) := {
  map : A → B;
  map_in_R : forall (a : A) (b : B), map a = b → R a b;
  R_in_map : forall (a : A) (b : B), R a b → map a = b;
  R_in_mapK : forall (a : A) (b : B) (r : R a b),
                (map_in_R a b (R_in_map a b r)) = r
}.
```

La fonction `map` est bien une application de `A` dans `B` ; le champ `map_in_R` décrit la propriété pour le graphe de cette application d'être inclus dans la relation `R` ; le champ `R_in_map` décrit la propriété inverse, c'est-à-dire que la relation `R` est incluse dans le graphe de `map` ; enfin, le champ `R_in_mapK` indique que les deux champs précédents s'annulent.[3] Pour obtenir les autres types de témoins unilatéraux, il suffit alors de retirer des champs à cet enregistrement. Par exemple, le type d'enregistrement sans le dernier champ correspond au témoin unilatéral M_3 et le type d'enregistrement vide correspond à M_0. L'association d'un témoin unilatéral au niveau α sur la relation `R` et d'un témoin unilatéral au niveau β sur la relation inverse `sym_rel R` correspond effectivement à un témoin de paramétricité $\mathrm{Param}^{(\alpha,\beta)}$.

3 : Le nommage est inspiré de la bibliothèque MathComp, dans laquelle une propriété d'annulation se nomme avec un suffixe `K` pour *cancel*.

Génération de termes en Coq-Elpi La génération de tous ces enregistrements est possible *via* une commande Coq-Elpi. On écrit alors un prédicat prenant en paramètre la classe de paramétricité de l'enregistrement souhaité. Il ouvre un module comme espace de noms dédié à cette classe de paramétricité et y déclare l'enregistrement. Voici la manière d'effectuer par exemple la déclaration du module associé à $\mathrm{Param}^{(4,3)}$ en Coq-Elpi :[4]

4 : Pour des raisons de lisibilité, on ne mentionne pas les univers à ce stade.

```
coq.env.begin-module "Param43" none,
RelDecl =
  parameter "A" _ {{ Type }} (a\
  parameter "B" _ {{ Type }} (b\
    record "Rel" {{ Type }} "BuildRel" (
      field [] "R" {{ lp:a → lp:b → Type }} (r\
      field [] "covariant" {{ Map4.Has lp:r }} (_\
      field [] "contravariant" {{ Map3.Has (sym_rel lp:r) }} (_\
    end-record))))),
coq.env.add-indt RelDecl _,
coq.env.end-module _.
```

Les valeurs `parameter`, `record`, `field` et `end-record` sont des constructeurs d'un type Coq-Elpi utilisé pour représenter les déclarations de types inductifs Coq.

L'API `coq.env.*` contient toutes les fonctions permettant d'interagir avec l'environnement Coq, et en particulier d'effectuer de nouvelles déclarations.

Lemmes de paramétricité Parmi tous les cas possibles de la relation de paramétricité de Trocq, certains relèvent uniquement du λ-calcul (comme l'application ou l'abstraction) et la construction du témoin consiste essentiellement à utiliser un combinateur pour joindre les résultats d'appels récursifs ; d'autres requièrent des preuves définies indépendamment de la procédure de traversée du but. Dans les cas de l'univers et du produit dépendant, il s'agit des preuves $p_{\square}$ et p_{Π} que nous avons appelées *lemmes de paramétricité*. Ces lemmes sont représentés en Coq par une famille de termes ayant les types suivants :[5]

5 : Dans le cas du produit dépendant, les classes de paramétricité retenues sont

$$(\alpha, \beta) = \mathcal{D}_{\Pi}(\gamma)$$

```
Definition Param□^γ : Param^γ Type Type.

Definition ParamΠ^γ
  (A A' : Type) (A_R : Param^α A A')
  (B : A → Type) (B' : A' → Type)
  (B_R : forall a a' a_R, Param^β (B a) (B' a')) :
    Param^γ (forall (a : A), B a) (forall (a' : A'), B' a').
```

Ces déclarations sont effectuées de la même manière que pour les enregistrements, en faisant d'abord les preuves manuellement pour les 6 niveaux de la hiérarchie, donnant 6 témoins unilatéraux à combiner entre eux pour obtenir les preuves finales. Une analyse de cas est faite sur la classe de paramétricité γ pour déterminer si le principe d'univalence (pour l'univers) ou d'extensionnalité des fonctions (pour le produit dépendant) est nécessaire pour effectuer la preuve. La différence avec les types de témoins de paramétricité est que les déclarations concernent ici des constantes et non des types inductifs. Le contenu d'une déclaration est donc un terme Coq dans l'encodage de Coq-Elpi et le prédicat utilisé pour faire la déclaration est `coq.env.add-const`.

13.1.2 Flexibilité des témoins de paramétricité

Afin d'implémenter la relation de paramétricité de Trocq, il est nécessaire de déclarer tous les types de témoins de paramétricité possibles, mais également d'assurer une certaine compatibilité entre ces types et une flexibilité dans leur utilisation. En effet, pour le développeur comme pour l'utilisateur, le contenu des témoins de paramétricité doit être transparent et la hiérarchie ne doit causer de lourdeur ni dans le code, ni dans les déclarations utilisateur. Ainsi, il faut pouvoir extraire facilement une information d'un enregistrement qui a le niveau suffisant pour la contenir, et il faut pouvoir accepter un témoin fourni par l'utilisateur partout où un témoin plus faible est attendu.

Fonctions d'affaiblissement Les témoins de paramétricité de Trocq peuvent être *affaiblis* à l'aide d'une fonction décrite en Figure 10.6. Cet affaiblissement intervient lorsque le témoin disponible est plus riche que le type de témoin attendu, afin que le témoin de paramétricité global reste bien typé. Ceci permet à l'utilisateur de ne déclarer qu'un seul témoin de paramétricité pour lier deux constantes entre elles, à la classe de paramétricité la plus élevée pour laquelle la preuve est possible. Ainsi, partout où un témoin est requis sur ces constantes à un niveau atteignable par affaiblissement à partir de celui du témoin fourni, l'affaiblissement est ajouté automatiquement par le greffon.

Une fonction d'affaiblissement peut être définie entre un type annoté source et un type annoté cible si le dernier est sous-type du premier. Dans le cas de base où le témoin est un enregistrement, l'affaiblissement correspond à l'oubli de champs et la recomposition des champs restants dans un nouvel enregistrement plus faible. La génération des différents affaiblissements se fait à partir de fonctions d'oubli atomiques déclarées à la main, supprimant le champ de niveau maximal dans un témoin de paramétricité unilatéral. Par exemple, voici les types de la fonction d'oubli du niveau $\mathbf{3}$ vers le niveau $\mathbf{2}_{\mathrm{a}}$ et de la fonction d'oubli de la classe $(\mathbf{4},\mathbf{3})$ vers la classe $(\mathbf{4},\mathbf{2}_{\mathrm{a}})$ générée avec Coq-Elpi à partir de cette dernière :

```
Definition forgetMap^3_{2_a} {A B : Type} {R : A → B → Type} :
  Map_3 R → Map_{2_a} R.

Definition forget^{(4,3)}_{(4,2_a)} {A B : Type} :
  Param^{(4,3)} A B → Param^{(4,2_a)} A B.
```

Projections Dans la définition précédente de l'enregistrement $\texttt{Map}_{\mathbf{4}}$, les champs ont été nommés pour rendre plus lisible l'extraction de données. Or, ces champs sont ceux d'un enregistrement unilatéral ensuite inclus dans un autre enregistrement, et chaque enregistrement existe dans un espace de noms séparé. En l'état, la récupération d'un champ est syntaxiquement lourde et dépendante de la classe de paramétricité du témoin en question. Cependant, les fonctions d'oubli peuvent être déclarées comme des *coercions*, permettant ainsi au vérificateur de typage de Coq d'oublier un nombre arbitraire de champs et donc de vérifier qu'un témoin est suffisamment riche. Grâce à cela, on déclare une seule fois une fonction de projection pour chaque champ, et cette fonction peut s'appliquer à tous les témoins de paramétricité qui contiennent ce champ en préservant le typage. Par exemple, voici la projection du champ `map_in_R` correspondant au niveau $\mathbf{2}_{\mathrm{a}}$ dans la hiérarchie :

```
Definition map_in_R {A B : Type} :
  Param^{(2_a,0)} A B → forall (a : A) (b : B), map R a = b → R a b.
```

Cette projection concerne en réalité le témoin unilatéral de gauche à droite, puisqu'elle s'applique à un témoin de classe $(\mathbf{2}_{\mathrm{a}},\mathbf{0})$, mais cette méthode permet également de nommer différemment les champs du témoin unilatéral de droite à gauche. Ainsi, Trocq contient aussi des déclarations pour les projection symétriques. La projection symétrique de `map_in_R` est nommée `comap_in_R` et a le type suivant :[6]

```
Definition comap_in_R {A B : Type} :
  Param^{(0,2_a)} A B → forall (b : B) (a : A), comap R b = a → R a b.
```

6 : La valeur `comap` est le symétrique de `map` et a pour type `A → B`.

13.2 Implémentation de la relation de paramétricité

Le cadre de paramétricité de Trocq a été conçu dans l'objectif d'être implémenté dans Coq par la suite. Sa présentation relationnelle, détaillée en Figure 10.5, est plus lourde que la présentation traditionnelle des traductions de paramétricité, mais a l'avantage de rendre explicites des détails importants au moment de l'implémentation, à savoir les étapes de manipulation des variables liées ainsi que l'origine des termes présents dans la conclusion des règles d'inférence. Pour implémenter Trocq, Coq-Elpi est un choix naturel, son paradigme logique étant entièrement en phase avec cette présentation relationnelle. Ces différents choix

à la conception et à l'implémentation permettent d'atteindre une forte similitude entre la présentation théorique de TrocQ et le code de la relation. Cette section met en lumière cette lisibilité dans l'implémentation de TrocQ.

13.2.1 Des règles d'inférence au programme logique

Tout d'abord, on peut remarquer que la présentation déductive d'un algorithme se calque très bien en programmation logique. En effet, l'algorithme peut être implémenté par un prédicat Elpi, où chaque cas de l'algorithme correspond à une instance du prédicat, et pour chaque cas, la tête de l'instance correspond à la conclusion et le corps de l'instance correspond aux prémisses. Ainsi, comme on construirait sur papier un arbre en empilant les différentes règles de l'algorithme, en commençant par la conclusion à obtenir à la racine de l'arbre, l'implémentation Coq-Elpi consiste en un appel au prédicat de paramétricité, chaque appel récursif représentant une nouvelle règle à ajouter à une branche de l'arbre.

Toutefois, la présentation théorique garde un certain niveau d'abstraction à travers la présence de la règle TrocqConv. En effet, cette règle concentre toute la flexibilité de la paramétricité de TrocQ, en ce qu'elle peut être ajoutée n'importe où dans l'arbre de dérivation pour rendre valide un témoin de paramétricité plus fort que nécessaire. Bien que cette règle élégante permette d'éviter d'ajouter des affaiblissements dans toutes les autres règles, elle rend la traversée du terme non déterministe car il existe deux règles possibles pour chaque construction du langage. Il est du rôle de l'implémentation de choisir quand faire appel à cette règle TrocqConv. Passer des règles d'inférence au programme logique se traduit par une déterminisation de l'algorithme de traversée du terme, suivie d'une association de chaque élément[7] apparaissant dans les règles à un fragment de code atomique correspondant en Coq-Elpi.

7 : Valeurs, opérations, types de prémisses, *etc.*

Déterminisation de l'algorithme Dans TrocQ, l'implémentation de la relation de paramétricité fait systématiquement appel à la règle d'affaiblissement TrocqConv dans les cas de base, c'est-à-dire sur les variables et les constantes. Il s'agit des cas dans lesquels une prémisse pourrait ne pas être prouvable en l'absence d'affaiblissement. En effet, dans le cas des variables (règle TrocqVar), la prémisse consulte directement le contexte de paramétricité Ξ pour trouver un témoin associé. Or, ce témoin n'a pas forcément un type annoté identique à celui auquel on traite cette variable. Dans le cas où le type de la variable contient des classes de paramétricité, la règle d'affaiblissement est même un outil crucial pour poser des contraintes sur ces classes pendant la traversée du but. Le cas des constantes est analogue : la base de connaissances contient un nombre fini d'associations possibles pour une même constante, avec des types annotés représentant précisément les dépendances nécessaires pour créer les témoins de paramétricité, et une constante n'est pas toujours exactement traitée à l'un de ces types. L'affaiblissement permet d'utiliser un témoin potentiellement plus riche déclaré en base de données si le type annoté désiré pendant la traversée du but n'y existe pas exactement. Utiliser l'affaiblissement dans les autres cas du prédicat peut conduire au calcul d'un témoin de paramétricité plus riche que nécessaire, ce qui est contraire à l'objectif de TrocQ.

Un autre élément non déterministe est visible dans le cas de l'application (règle TrocqApp), seul cas dans lequel il existe un terme dont l'origine n'est pas fixée. En effet, il s'agit du domaine A de la fonction en tête de l'application. Plutôt que de

créer une variable fraîche à cet endroit et potentiellement causer un affaiblissement superflu dans l'appel récursif sur f, l'implémentation récupère le type de f dans le contexte pour lire la bonne valeur de A. Ainsi, l'affaiblissement potentiel effectué pendant le traitement de f porte sur le type B.

Correspondance entre les règles et le code Pour étudier le lien entre les règles et le code, prenons dans un premier temps le cas des variables liées, qui fait appel aux règles TrocqVar et TrocqConv. Voici la combinaison de ces deux règles qui est implémentée dans Trocq :

$$\frac{(x, T, x', x_R) \in \Delta \qquad \gamma(\Delta) \vdash_+ T \preccurlyeq T'}{\Delta \vdash_t x \mathbin{@} T' \sim x' \mathbin{\because} \Downarrow_{T'}^{T} x_R}$$

Voici à présent l'instance correspondante du prédicat de la relation de paramétricité dans Trocq :[8]

```
1 param X T' X' (W XR) :- name X, !,
2   param.store X T X' XR,
3   annot.sub-type T T',
4   weakening T T' (wfun W).
```

8 : Dans les prochains blocs de code, les prédicats d'affichage ont été retirés car ils ne sont pas utiles dans la présentation et n'enlèvent rien à notre argumentation.

Le prédicat `param` à quatre arguments est le prédicat de paramétricité principal. La tête de l'instance correspond à la conclusion de la règle, où `W` est la fonction d'affaiblissement générée en ligne 4. La condition `name X` permet de vérifier que `X` est bien une variable et de faire échouer cette instance sur tous les autres termes, afin d'exécuter à la place l'instance qui leur est dédiée. Les lignes 2 et 3 sont les prémisses. Le prédicat `param.store` est utilisé pour représenter le contexte de paramétricité Δ. Il apparaît donc que l'association entre les règles d'inférence et l'instance du prédicat est assez transparente.

Intéressons-nous à présent à un cas plus complexe, celui du type flèche, qui fait intervenir des contraintes entre classes de paramétricité ainsi que des appels récursifs à `param` :[9]

$$\frac{\begin{array}{c} C = (M, N) \qquad (C_A, C_B) = \mathcal{D}_{\to}(C) \\ C_A = (M_A, N_A) \qquad \Delta \vdash_t A \mathbin{@} \square^{(M_A, N_A)} \sim A' \mathbin{\because} A_R \\ C_B = (M_B, N_B) \qquad \Delta \vdash_t B \mathbin{@} \square^{(M_B, N_B)} \sim B' \mathbin{\because} B_R \end{array}}{\Delta \vdash_t A \to B \mathbin{@} \square^{(M,N)} \sim A' \to B' \mathbin{\because} p_{\to}^{C}\, A_R\, B_R}$$

9 : La règle d'inférence est une version volontairement réorganisée de la règle TrocqArrow, dans laquelle certaines classes de paramétricité sont explicitement nommées et où les contraintes portant sur les univers n'apparaissent pas. En effet, étant déléguées à Coq, elles n'apparaissent pas dans le code et peuvent ici être ignorées.

```
1  param
2    (prod _ A (_\ B)) (app [pglobal (const PType) _, M, N])
3    (prod `_` A' (_\ B')) (app [pglobal (const ParamArrow) UI|Args]) :-
4      param.db.ptype PType, !, std.do! [
5        cstr.univ-link C M N,
6        cstr.dep-arrow C CA CB,
7        cstr.univ-link CA MA NA,
8        param A (app [pglobal (const PType) _, MA, NA]) A' AR,
9        cstr.univ-link CB MB NB,
10       param B (app [pglobal (const PType) _, MB, NB]) B' BR,
11       param.db.param-arrow C ParamArrow,
12       prune UI [],
13       util.if-suspend C (param-class.requires-axiom C) (
14         coq.univ-instance UI0 [],
15         Args = [
```

```
16            pglobal (const {param.db.funext}) UI0, A, A', AR, B, B', BR
17          ]
18        ) (
19          Args = [A, A', AR, B, B', BR]
20        )
21      ].
```

Les six prémisses sont représentées dans le code par les lignes 5 à 10, dans le même ordre que dans la règle d'inférence. Le code restant récupère la preuve $p^C_{\to}$ présente dans la conclusion (variable `ParamArrow`) et lui applique les bons arguments, pour partie implicites dans la présentation sur papier.

13.2.2 Fonctionnalités utiles de Coq-Elpi

Comme le montrent les règles d'inférence la décrivant (Figure 10.5), la relation de paramétricité de Trocq fait intervenir des opérations de natures différentes : appels récursifs, contraintes sur des classes de paramétricité, construction de preuves à partir de lemmes de paramétricité ou de témoins ajoutés par l'utilisateur. La mise à disposition de toutes ces opérations se traduit par une certaine infrastructure logicielle, cependant masquée dans l'implémentation de Trocq par la division du code en plusieurs fichiers et certaines fonctionnalités de Coq-Elpi, telles que la *suspension de buts* ou les *règles de gestion de contraintes* [51] (CHR).

[51] : Frühwirth (1994), « Constraint Handling Rules »

Premièrement, la division du code permet une meilleure lisibilité. C'est pourquoi, dans le code de Trocq, toute la logique de contraintes sur les classes de paramétricité passe par une API de prédicats `cstr.*`, pour ne pas exposer cette portion de l'implémentation dans la définition du prédicat principal de paramétricité, et ainsi rester le plus fidèle possible aux règles d'inférence.

Deuxièmement, l'usage des CHR de Coq-Elpi permet de maintenir tout au long de la traversée du but un état global contenant les différentes contraintes de classes de paramétricité. Ceci présente l'avantage que la structure de données stockant les contraintes est invisible dans le prédicat `param`,[10] mais également qu'aucun encodage réifié des classes de paramétricité n'est nécessaire dans les termes annotés. En effet, une fonctionnalité centrale dans l'utilisation des CHR est la réification des variables dans la tête des règles. Contrairement au code classique Elpi calqué sur Prolog, où la notion de comparaison de termes repose sur l'unification, dans le contexte des CHR, la première phase de l'exécution des règles, le filtrage par motif, s'exécute au niveau méta. À ce niveau, deux variables syntaxiquement différentes dans les différents arguments des buts suspendus ne peuvent par conséquent pas être unifiées. Ceci permet entre autres d'indexer des structures de données sur des variables, donc de laisser les variables de classes de paramétricité telles quelles dans les termes annotés.[11]

10 : Étant un état global, il est possible de ne jamais le nommer ni le mentionner dans le code.

Troisièmement, la suspension de buts permet d'implémenter l'algorithme de la même manière quel que soit le but initial. En effet, lors de l'annotation d'un but avant traversée, de nombreuses variables fraîches sont créées pour représenter les classes de paramétricité encore inconnues. Dans l'appel initial au prédicat `param`, en général, la classe de paramétricité à laquelle on souhaite traiter le but est connue : on souhaite la classe $(0, 1)$ afin d'extraire une fonction du nouveau but vers ce but initial. Dans le code de la relation pour le type flèche, les variables `M` et `N` sont donc définies, ce qui détermine ensuite `C` et les autres variables du programme qui en dépendent. Cependant, au fil des appels récursifs, il est possible que certaines variables de classes de paramétricité restent indéfinies et

11 : En interne, elles sont tout de même associées à un entier, car la structure de données stockant les contraintes nécessite de pouvoir comparer les clés, une tâche plus simple sur des entiers que sur des variables Elpi.

n'aient pas une valeur concrète avant un travail ultérieur de détermination des classes de paramétricité.[12] Or, le code peut avoir à calculer des résultats à partir de telles variables.[13] La fonctionnalité de suspension de buts d'Elpi est donc la bienvenue dans l'implémentation de Trocq, puisqu'elle permet de bloquer un calcul sur une variable tant que cette variable est indéfinie, puis de le réveiller dès qu'une classe de paramétricité concrète est affectée à la variable.

12 : Le problème associé, la solution retenue et son implémentation sont le sujet de la section suivante.

13 : Par exemple, récupérer une valeur dans la base de connaissances (ligne 11 dans le dernier bloc de code), ou tester si une classe requiert un axiome (ligne 13).

13.3 Inférence de classes de paramétricité

Une particularité de la relation de paramétricité de Trocq est qu'elle peut relier un même terme à plusieurs termes associés valides. En effet, la relation part d'un terme dans CC^{+}_{ω} contenant des univers annotés par des classes de paramétricité. Or, dans notre cas, le terme d'entrée est obtenu par une annotation automatique du but initial qui crée une variable fraîche pour chaque classe de paramétricité. De plus, les règles de Trocq ne contraignent pas explicitement ces classes à être égales à une valeur en particulier, mais simplement à être au-dessus ou en-dessous d'une valeur. Ainsi, la traversée du but ne *fixe* pas les différentes classes de paramétricité trouvées dans les types annotés, mais les *contraint* seulement. À la fin du processus, certaines classes de paramétricité peuvent alors avoir plusieurs solutions valides.

Par exemple, lors du traitement de $\Pi A : \square. A \to A$ au niveau $(0, 1)$, un témoin de paramétricité de niveau $(2_a, 0)$ *minimum* est requis sur le sous-terme $\square$. Cela signifie que toutes les entrées de la relation pour $\square$ qui sont au-dessus de ce niveau sont également acceptables, car elles possèdent au moins la quantité d'information nécessaire. De la même manière, le traitement de $\mathbb{N} \to \mathbb{N}$ au niveau $(1, 0)$ demandera deux témoins de paramétricité sur $\mathbb{N}$, l'un au niveau $(0, 1)$ *minimum* et l'autre au niveau $(1, 0)$ *minimum*. Tous les témoins de niveau supérieur donnent une preuve finale bien typée.

Pourtant, il est important de déterminer une valeur finale pour ces classes, car ces valeurs ont un impact potentiel sur la quantité d'informations demandée à l'utilisateur, voire sur les axiomes requis pour effectuer le pré-traitement. Pour éviter au maximum l'usage d'axiomes et demander le moins d'informations possible à l'utilisateur, il semble intéressant de chercher à minimiser toutes les classes de paramétricité dans le témoin de sortie. Or, ce problème de minimisation n'a pas toujours de solution, en particulier lorsque le but contient des constantes utilisateur. L'implémentation de Trocq utilise une heuristique permettant dans de nombreux cas d'obtenir une solution satisfaisante.

Dans cette section, nous définissons le problème d'inférence en l'illustrant par un exemple. Ensuite, nous explorons les différentes solutions envisagées et justifions les choix techniques effectués dans Trocq. Enfin, nous donnons des détails sur l'implémentation de l'affaiblissement et le sous-typage, car ce sont des points-clés pour le bon fonctionnement de l'inférence de classes de paramétricité.

13.3.1 Définition du problème

Pour illustrer le problème de l'inférence de classes de paramétricité, prenons un exemple. On considère le traitement du but initial suivant, fraîchement annoté

automatiquement par Trocq :

$$\Pi F : \square^{\alpha} \to \square^{\beta}. \Pi A : \square^{\gamma}. F\ A \to F\ A$$

Il contient trois variables fraîches α, β et γ, une pour chaque univers. La trace de la traversée de ce terme effectuée par Trocq est la suivante :

Traitement du but initial au type $\square^{(0,1)}$.
Application de TrocqPi.

- Traitement du domaine $\square^{\alpha} \to \square^{\beta}$ au type $\square^{(2_a,0)}$.
  Application de TrocqArrow.
  - Traitement de $\square^{\alpha}$ au type $\square^{(0,2_b)}$.
    Application de TrocqSort.
    Ajout de la contrainte $(\alpha, (0, 2_b)) \in \mathcal{D}_{\square}$.
  - Traitement de $\square^{\beta}$ au type $\square^{(2_a,0)}$.
    Application de TrocqSort.
    Ajout de la contrainte $(\beta, (2_a, 0)) \in \mathcal{D}_{\square}$.
- Traitement de $\Pi A : \square^{\gamma}. F\ A \to F\ A$ au type $\square^{(0,1)}$.
  Application de TrocqPi.
  - Traitement de $\square^{\gamma}$ au type $\square^{(2_a,0)}$.
    Application de TrocqSort.
    Ajout de la contrainte $(\gamma, (2_a, 0)) \in \mathcal{D}_{\square}$.
  - Traitement de $F\ A \to F\ A$ au type $\square^{(0,1)}$.
    Application de TrocqArrow.
    - ⋆ Traitement de $F\ A$ au type $\square^{(1,0)}$.
      Application de TrocqApp en prenant $\square^{\alpha}$ comme domaine de F.
      - · Traitement de F au type $\square^{\alpha} \to \square^{(1,0)}$.
        Application de TrocqConv puis TrocqVar.
        Ajout de la contrainte $\beta \geq (1, 0)$.
      - · Traitement de A au type $\square^{\alpha}$.
        Application de TrocqConv puis TrocqVar.
        Ajout de la contrainte $\gamma \geq \alpha$.
    - ⋆ Traitement de $F\ A$ au type $\square^{(0,1)}$.
      Application de TrocqApp en prenant $\square^{\alpha}$ comme domaine de F.
      - · Traitement de F au type $\square^{\alpha} \to \square^{(0,1)}$.
        Application de TrocqConv puis TrocqVar.
        Ajout de la contrainte $\beta \geq (0, 1)$.
      - · Traitement de A au type $\square^{\alpha}$.
        Application de TrocqConv puis TrocqVar.
        Ajout de la contrainte $\gamma \geq \alpha$ (doublon).

On a donc un ensemble de contraintes à la fin de la traversée, décrivant des ensembles de valeurs admissibles pour α, β et γ, ainsi que des relations entre ces variables à toujours respecter, mais aucune valeur n'est définie concrètement. Le problème est donc de trouver une valuation admissible pour ces variables, à partir des contraintes présentes dans les règles d'inférence de Trocq.

Solution univalente Une solution au problème existe toujours : il suffit de fixer toutes les classes à $(4, 4)$, ce qui correspond à une traduction de paramétricité

univalente. Expliquons pourquoi une telle solution est toujours valide. Premièrement, les différentes contraintes possibles dans la traversée peuvent toujours se réduire à des contraintes d'ordre entre les variables impliquées ou d'égalité à $(4,4)$. Une contrainte de forme $(\alpha,\beta) \in \mathcal{D}_{\square}$ peut se réduire lorsque la valeur de β est connue, en distinguant deux cas : si β appartient à $\{0,1,2_a\}^2$, alors α n'est pas contrainte; sinon, $\alpha = (4,4)$. La procédure qui fixe toutes les classes à $(4,4)$ est donc compatible avec cette contrainte : si l'on fixe β à $(4,4)$, la contrainte impliquée sur α est compatible puisque l'on fixe également α à $(4,4)$. Une contrainte de forme $(\alpha,\beta) = \mathcal{D}_\star(\gamma)$ où $\star \in \{\to,\Pi\}$ peut se réduire à deux contraintes d'ordre lorsque la valeur de γ est connue. Si l'on fixe γ à $(4,4)$, alors les tableaux de dépendance en Figure 9.2 donnent $\alpha = \beta = (4,4)$, ce qui est compatible puisque l'on fixe également ces classes à $(4,4)$. Enfin, les contraintes d'ordre entre les classes ne sont jamais strictes, donc deux classes α et β fixées à $(4,4)$ respectent toujours une contrainte de forme $\alpha \geq \beta$. Par conséquent, la solution que l'on peut qualifier d'*univalente* est toujours valide.

Recherche d'une solution minimale Cependant, la solution univalente n'est pas satisfaisante puisqu'elle demande toujours la quantité maximale d'informations à l'utilisateur et au moins un axiome dès lors qu'un univers ou un produit dépendant est présent dans le but initial. On cherche donc une autre solution, avec l'objectif opposé : moins on demande d'informations en général, plus on peut traiter de buts à partir d'une base de connaissances utilisateur donnée, et moins on a de chances d'avoir besoin d'un axiome. Il semble alors naturel de vouloir minimiser toutes les classes de paramétricité. Toutefois, il n'est pas certain qu'une telle solution puisse être construite, ni qu'elle atteigne réellement l'objectif.

D'une part, il faut déterminer un ordre de minimisation des classes de paramétricité présentes dans les contraintes. Or, l'ajout de contraintes pendant la traversée n'est pas totalement structurel. En effet, les contraintes portant sur un nœud du but initial ne sont pas toutes déterminées lors de la traversée de ce nœud. Par exemple, toute occurrence d'une variable liée peut induire un ajout de contrainte sur les classes de paramétricité présentes dans son type, bien que ce dernier ait déjà été traité plus tôt, lors de la traversée du lieur qui introduit la variable liée. Il n'est donc pas évident que minimiser les variables dans l'ordre dans lequel on les rencontre soit la meilleure solution. De plus, en l'absence d'une étude plus poussée du comportement des contraintes en fonction de la structure du but, on ne peut écarter la possibilité que dans certains buts, minimiser une classe de paramétricité se fasse au détriment d'une autre classe de paramétricité. Dans un tel cas, en minimisant une classe avant l'autre, on ferait alors augmenter la seconde, impliquant une utilisation d'un axiome qui aurait pu être évitée en changeant l'ordre de minimisation des classes. Ce cas semble plus susceptible de se produire en présence de constantes dans le but initial, car les valuations valides des classes de paramétricité dans le type d'une constante sont propres à cette constante et ne respectent *a priori* aucune propriété commune à toutes les constantes. L'ordre de minimisation semble donc difficile à définir de façon analytique.

D'autre part, toujours dans le cas des constantes, il se peut que la minimisation ne soit pas une solution optimale. En effet, les tableaux de dépendance en Figure 9.2 suggèrent que les classes requises sur le domaine et le codomaine du type d'une fonction augmentent avec la classe requise sur ce type. Or, bien que cela soit intuitif dans le cas des types fonctionnels, sans une étude plus poussée, on ne peut affirmer que cette propriété que l'on pourrait qualifier de *monotonie* se généralise

à toutes les constantes. Dans le cas d'une constante non monotone, tenter de minimiser toutes les classes pourrait alors se montrer contre-productif. Par conséquent, le fait même de chercher à minimiser toutes les classes de paramétricité présentes dans les contraintes pourrait n'être qu'une heuristique imparfaite.

13.3.2 Solution retenue dans TROCQ

Dans le cadre de l'implémentation de TROCQ, on propose une solution empirique dont l'optimalité n'est pas garantie en théorie. Dans cette solution, on fait l'hypothèse que tout le calcul est monotone, donc la minimisation des classes de paramétricité est une bonne heuristique. Il reste à déterminer l'ordre de minimisation.

Contraintes complexes Une remarque centrale concernant l'ordre de minimisation est la suivante : lors de la minimisation d'une classe de paramétricité en particulier, afin de sélectionner la classe minimale possible, toutes les contraintes d'ordre liées à cette classe doivent être connues. En effet, les contraintes ajoutées pendant la traversée du but sont soit des contraintes d'ordre, soit des contraintes *complexes*, impliquant un tableau de dépendances à l'instar des tableaux de la Figure 9.2. Lorsque la construction concernée est un constructeur de types, elle vit dans un univers muni d'une classe de paramétricité que l'on appellera par la suite classe de *sortie*. Le tableau de dépendances a alors une ligne par classe de paramétricité, dans laquelle on peut lire les différentes dépendances sur les autres classes de paramétricité liées à cette construction. Ces dépendances sont obtenues en déterminant d'abord la classe de paramétricité de sortie. Une fois que l'on connaît ces dépendances, on peut remplacer la contrainte complexe par une liste de contraintes d'ordre à partir de chacune des classes présentes dans la ligne du tableau. Pour chaque contrainte complexe, on sait donc que la variable correspondant à la classe de sortie doit être instanciée avant les autres, afin de *réduire* la contrainte complexe en une ou plusieurs contraintes d'ordre.

Par exemple, le but illustrant le problème d'inférence de classes en § 13.3.1 est un produit dépendant. La règle TROCQPI permet de déterminer à quels types on doit traiter le domaine $\square^{\alpha} \to \square^{\beta}$ et le codomaine $\Pi A : \square^{\gamma}. F\ A \to F\ A$, à l'aide d'une contrainte $(\alpha, \beta) = \mathcal{D}_{\Pi}(\gamma)$. La traversée du but démarre avec un type annoté $\square^{(0,1)}$, c'est-à-dire qu'elle fixe la classe de sortie à $(0, 1)$. On a donc $\gamma = (0, 1)$, et ce n'est qu'avec cette information que l'on peut trouver la bonne ligne dans le tableau de la Figure 9.2 permettant de déterminer α et β.

Contraintes d'ordre Dans une contraintes d'ordre, une variable peut être en position supérieure ou inférieure. Pour trouver la classe minimale pour une variable, on considère toutes les contraintes d'ordre dans lesquelles cette variable est en position supérieure, car ces contraintes donnent des bornes inférieures pour cette variable. La classe minimale est alors la plus grande des bornes inférieures. Dans le cas où deux variables sont liées par une contrainte d'ordre $\alpha \geq \beta$, β constitue une borne inférieure pour α. Par conséquent, pour déterminer la classe minimale pour α, il faut d'abord connaître la classe constante retenue pour β. Cette contrainte d'ordre pose donc également un ordre dans la minimisation des deux variables.

L'ordre de priorité global est donc déterminé à l'aide des règles suivantes :

- $(\alpha, \beta) \in \mathcal{D}_{\square}$ se réduit en une contrainte d'égalité sur α ou bien aucune contrainte, en fonction de la valeur de β : elle force donc la variable β à être instanciée avant α ;
- $\mathcal{D}_{\Pi}(\gamma) = (\alpha, \beta)$ ou $\mathcal{D}_{\rightarrow}(\gamma) = (\alpha, \beta)$ force γ à être instanciée avant α et β ;
- $\alpha \geq \beta$ force β à être instanciée avant α ;
- $\mathcal{D}_{K}(\gamma, T)$ force γ à être instanciée avant les classes de paramétricité présentes dans le type annoté T.

13.3.3 Implémentation

Le rôle de l'algorithme d'inférence est donc de faire émerger l'ordre de minimisation à partir des différentes contraintes qui apparaissent pendant la traversée, puis de sélectionner pour chaque variable la classe minimale possible, chaque affectation permettant de réduire des contraintes complexes en des contraintes d'ordre sur d'autres variables encore non affectées. Cet algorithme peut être implémenté de plusieurs façons.

Résolution de contraintes sur domaine fini Une première idée est de remarquer que le problème de l'inférence de classes de paramétricité ressemble beaucoup à un problème de résolution de contraintes sur domaine fini. Un tel problème peut être résolu de manière élégante dans le style de la programmation logique par contraintes [82], idiomatique dans les langages basés sur Prolog tels qu'Elpi. En effet, on peut représenter chaque classe de paramétricité inconnue dans le but initial comme une variable avec un domaine initial allant de $(0, 0)$ à $(4, 4)$, muni d'un ordre partiel. Les différentes contraintes d'ordre ajoutées sur les variables réduisent leur domaine, ne laissant que les solutions valides. Les contraintes complexes sont mises en attente tant que leur classe de sortie n'est pas connue, et lorsque c'est le cas, elles sont réduites automatiquement et remplacées par de nouvelles contraintes d'ordre. Chaque contrainte ajoutée constitue également une contrainte sur l'ordre de minimisation. Ainsi, à la fin de la traversée, l'ordre de minimisation émerge naturellement et il suffit de calculer la plus grande borne inférieure de chaque variable en suivant cet ordre pour obtenir la solution voulue.

[82] : Jaffar *et coll.* (1987), « Constraint Logic Programming »

Le domaine des variables n'étant pas classique (il s'agit de classes de paramétricité et non d'entiers), il faut implémenter un solveur de contraintes *ad hoc*. Les règles de gestion de contraintes (CHR), disponibles dans Elpi et présentées en § 4.1.1, sont l'une des méthodes les plus connues pour exprimer des algorithmes complexes impliquant la génération et la gestion de contraintes avec des règles. Le langage des CHR est adapté à la conception de solveurs de contraintes, car les ingrédients centraux de ces solveurs, la propagation des contraintes et la vérification de la cohérence,[14] peuvent être implémentés sous la forme de règles. Le solveur de contraintes combine ces règles avec une procédure de recherche qui teste les valeurs restantes dans les domaines après la propagation. Dans notre cas, les règles sont utilisées pour simplifier les contraintes complexes et les réduire à des contraintes d'ordre sur les classes de paramétricité.

14 : La propriété de cohérence est la compatibilité à tout instant entre les domaines des variables et les différentes contraintes qui portent sur ces variables.

Style direct Cependant, l'élégance d'une telle solution se fait au détriment de la traçabilité du flot de contrôle dans le processus de réduction. En effet, dans les solveurs de contraintes, lorsque les contraintes sont déclarées, après une phase

initiale de propagation, elles restent en mémoire dans un état endormi, observant les variables qu'elles concernent, afin d'être réveillées chaque fois qu'un de leurs domaines est mis à jour. Ce comportement, nécessaire pour assurer à tout instant la cohérence, rend le flot de contrôle très complexe car il est difficile de savoir quand la propagation se produira simplement en lisant le code, ce qui complique les phases de débogage.

Au lieu de cela, nous présentons un algorithme d'inférence dans un style plus pragmatique, accumulant d'abord les contraintes dans un *graphe de contraintes global*, puis en le réduisant et en instanciant les variables par la suite. Dans ce graphe de contraintes, les nœuds sont des classes de paramétricité, variables ou constantes, et il existe différents types d'arêtes, un par type de contrainte pouvant être ajoutée pendant la traversée du but. Un exemple de graphe de contraintes est disponible dans la Figure 13.2. Dans ce graphe, les arêtes de X_1 vers X_2 et X_3 représentent la contrainte $(X_2, X_3) = \mathcal{D}_\Pi(X_1)$, et l'arête de X_6 vers la classe constante $(3, 2_\mathrm{b})$ représente la contrainte $(3, 2_\mathrm{b}) \geq X_6$.

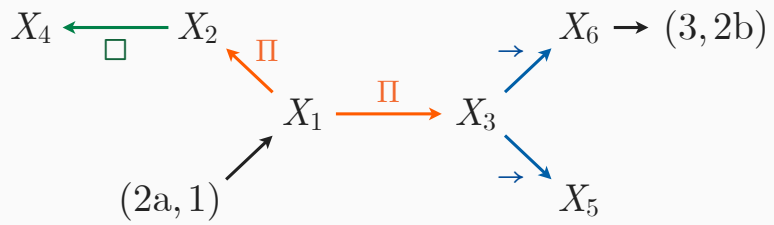


Fig. 13.2 : Exemple de graphe de contraintes

Une particularité du graphe de contraintes est que le sens des arêtes est cohérent avec l'ordre de minimisation des classes de paramétricité. Ainsi, dans le graphe ci-dessus, l'arête de X_1 à X_2 signifie qu'il y a une contrainte sur un produit dépendant mais également que la variable X_1 en est la classe de sortie, donc elle doit être instanciée avant X_2. Il en va de même pour les autres types de contraintes : X_3 doit être instanciée avant X_5 et X_6, X_2 avant X_4, *etc.* Cela signifie que le graphe de contraintes est également un *graphe de priorité*. L'ordre final d'instanciation peut alors être obtenu en effectuant un *tri topologique* sur ce graphe : on retire récursivement les nœuds d'entrée en les ajoutant à une liste, pour obtenir un ordre de minimisation admissible, tout ceci en ignorant les nœuds qui ne sont pas des variables. Par exemple, sur le graphe ci-dessus, un ordre admissible est $[X_1, X_2, X_4, X_3, X_5, X_6]$. Si l'on respecte l'ordre de minimisation, lors de l'instanciation d'une variable, toutes les arêtes pointant vers cette variable, c'est-à-dire tous les nœuds inférieurs, doivent former un ensemble de classes constantes. Chaque fois qu'une variable est instanciée, le nœud correspondant est supprimé du graphe. Ensuite, les contraintes complexes dont la variable fraîchement instanciée est la classe de sortie peuvent à présent être réduites en contraintes d'ordre, de nouveau ajoutées au graphe. Ainsi, on maintient l'invariant et la variable suivante est prête à être instanciée. Par exemple, lorsque X_1 est instanciée avec la valeur $(2_\mathrm{a}, 1)$, le nœud disparaît. On peut alors réduire la contrainte $(X_2, X_3) = \mathcal{D}_\Pi(X_1)$ en deux contraintes d'ordre $X_2 \geq (2_\mathrm{a}, 4)$ et $X_3 \geq (2_\mathrm{a}, 1)$ qui sont de nouveau ajoutées au graphe. La variable suivante dans l'ordre de minimisation est X_2, et la seule contrainte qui pointe vers le nœud associé dans le graphe est cette nouvelle contrainte d'ordre. Il n'y a donc plus de contrainte complexe et l'instanciation peut être effectuée. Le processus continue ainsi jusqu'à la suppression de tous les nœuds du graphe.

13.3.4 Affaiblissement et sous-typage

L'ajout de contraintes sur une classe de paramétricité garantit que la valeur finale qui lui est affectée après la traduction correspond aux besoins des différentes occurrences de cette variable, c'est-à-dire que le témoin de paramétricité associé possède suffisamment d'informations dans toutes ses occurrences dans le témoin global. Néanmoins, cela ne signifie pas que toutes les occurrences ont besoin du même témoin de paramétricité. Il est fort probable qu'au moins une occurrence de la variable requière moins d'informations que ce qui est contenu dans le témoin. Dans un tel cas, pour garantir le typage du terme de preuve, il faut insérer devant le témoin une fonction d'affaiblissement. Or, la nature de cette fonction ne peut être déterminée que lorsque l'on connaît la classe source et la classe cible de l'affaiblissement.

Par exemple, reprenons le but traité en § 13.3.1 :

$$\Pi F : \square^{\alpha} \to \square^{\beta}. \Pi A : \square^{\gamma}. F\ A \to F\ A$$

Dans la trace de sa traversée, on voit que le sous-terme $F\ A$ est traité deux fois, au type $\square^{(1,0)}$ puis au type $\square^{(0,1)}$. Pendant la traversée, les variables F et A se voient attribuer des variables associées F' et A'. Dans le témoin de paramétricité global, à la position de la première occurrence de $F\ A$, il est attendu une preuve de type $\mathrm{Param}^{(1,0)}\ (F\ A)\ (F'\ A')$, et à sa seconde occurrence, une preuve de type $\mathrm{Param}^{(0,1)}\ (F\ A)\ (F'\ A')$. Les deux occurrences impliquent donc des témoins de types différents, tous deux des sous-types du type final qui sera retenu après la phase d'inférence de classes. Or, ce type final n'est pas connu pendant la traversée, et il faut pourtant ajouter un affaiblissement devant le témoin F_R dans le terme de preuve au moment où les occurrences de $F\ A$ sont traversées.

Pour résoudre ce problème, l'implémentation de Trocq représente les affaiblissements dans la syntaxe par une identité avec des arguments fantômes associés aux classes de paramétricité présentes dans le graphe de contraintes :

```
Definition weaken (m₁ n₁ m₂ n₂ : map_class) {A : Type} (a : A) := a.
```

Ceci permet de maintenir un terme Coq bien formé et bien typé tout en y ajoutant des informations visibles lors du parcours du terme. Après l'affectation des variables, une procédure de complétion s'exécute, remplaçant ces marqueurs par de véritables fonctions d'affaiblissement, qui peuvent à ce stade être générées puisque le marqueur contient les deux classes de paramétricité constantes nécessaires.

On note que l'affaiblissement fait intervenir une suspension dans les cas de l'abstraction et de l'application. Cette suspension a pour utilité de gérer le cas d'un β-rédex, c'est-à-dire une application avec à sa tête une abstraction. La suspension est implémentée par un type de données Elpi pouvant contenir le corps d'une abstraction afin de retarder la création de la fonction d'affaiblissement jusqu'à la lecture des arguments fournis à cette abstraction. On exploite ici une fois de plus la représentation HOAS des termes Coq par des méta-fonctions Elpi.

13.4 Polymorphisme d'univers

Tout au long de ce document, qu'il s'agisse des parties traitant de la théorie ou de l'implémentation, nous avons présenté des λ-termes dans des calculs possé-

dant une hiérarchie d'univers où les univers sont indexés par un entier représentant leur niveau. Le système de typage permet alors de donner à chaque univers $\square_i$ un type, son successeur $\square_{i+1}$, ou n'importe quel univers de niveau strictement supérieur grâce à la règle de cumulativité. Or, les différents lemmes de paramétricité que nous avons décrits sont *univers-polymorphes*, c'est-à-dire qu'ils fonctionnent avec des types de n'importe quels univers tant que ces univers respectent les différentes contraintes posées par la définition du lemme. Ainsi, la règle TrocqPi utilise trois niveaux d'univers i, j et k qui peuvent prendre n'importe quelle valeur pourvu que la contrainte $k = \max(i, j)$ est respectée.

À l'usage des lemmes de paramétricité, les univers liés doivent donc être instanciés pour former des termes valides. Or, la propriété importante pour le typage est simplement l'*existence* d'une valeur admissible pour chaque univers lié. Par conséquent, lorsque l'on est certain qu'il existe toujours une solution au problème de contraintes sur les univers, on peut se permettre de laisser les univers *implicites*. Par exemple, une traduction de paramétricité telle que celles qui ont été présentées dans cette thèse est une traduction structurelle qui n'introduit pas d'univers ni de contrainte d'univers supplémentaires par rapport au contexte du but initial, et qui n'a pas pour objectif de raisonner sur les univers. Comme il n'y a pas de risque d'introduire des incohérences, nous pouvons nous permettre d'ignorer les univers dans nos présentations théoriques.

À l'implémentation, les univers peuvent également être laissés implicites car le noyau de Coq peut les inférer lors de la vérification du typage. Cependant, dans le cadre du polymorphisme d'univers, cette inférence est moins puissante car elle doit aussi inférer des univers liés.[15] Il devient alors nécessaire d'expliciter certains univers présents dans les termes. Or, le polymorphisme d'univers est une fonctionnalité encore expérimentale dans Coq-Elpi. De plus, certains univers dans Trocq sont *algébriques*, c'est-à-dire qu'ils s'expriment à partir d'autres univers et d'opérations arithmétiques, et l'implémentation actuelle de Coq est limitée quant à cette fonctionnalité précise. Cette section détaille donc les problèmes qui surviennent lors de l'implémentation de Trocq concernant l'ambiguïté typique et les univers algébriques.

15 : Dans le reste de cette section, on se place dans ce cadre.

13.4.1 Levée de l'ambiguïté typique

Lors de la déclaration d'une constante univers-polymorphe dans Coq, les univers présents dans le terme deviennent des univers liés. Par exemple, dans la déclaration suivante, `A` et `B` sont des types vivant chacun dans un univers frais (disons respectivement u_0 et u_1), mais ces univers sont laissés implicites grâce à l'*ambiguïté typique*, la fonctionnalité de Coq qui permet l'inférence automatique des niveaux d'univers :

```
Definition twice (A : Type) (B : Type) (f : A → A → B) : A → B :=
  fun a ⇒ f a a.
```

Quand le terme est donné à Coq pour être enregistré en tant que constante et nommé `twice`, les niveaux u_0 et u_1 sont remplacés par des univers liés, respectivement i et j. La constante `twice` est alors un terme incomplet qui attend une *instance d'univers*, c'est-à-dire un tableau de niveaux d'univers qui donne une valeur à chaque univers lié. Une nouvelle fois, grâce à l'ambiguïté typique, on peut ne pas mentionner l'instance d'univers à l'usage de la constante et Coq se charge de l'inférence, permettant de bénéficier des avantages du polymorphisme d'univers sans alourdir les termes syntaxiquement.

Cependant, l'inférence est imparfaite car elle ne minimise pas le nombre d'univers liés. En effet, il existe des cas où une constante univers-polymorphe avec au moins deux univers liés peut être instanciée avec plusieurs fois le même univers au sein d'une même instance. Or, dans un tel cas, Coq alloue une variable d'univers différente pour chaque univers lié de l'instance. Par exemple, si l'on utilise la constante `twice` déclarée ci-dessus dans un autre terme, elle apparaît sous une forme instanciée par Coq, c'est-à-dire `twice@{u₃ u₄}`, même s'il est possible de l'instancier avec deux fois u_3. Si le terme qui contient une telle constante est ensuite déclaré dans Coq, ces univers deviennent liés. Ainsi, l'instance d'univers du terme global contient plus d'univers liés que nécessaire, menant dans certains cas à des termes ultérieurs mal typés.

Pour garantir le typage des termes générés par un méta-programme, il faut déterminer si de tels cas sont possibles pour savoir si l'ambiguïté typique peut être utilisée. Dans le cas de l'implémentation de Trocq, beaucoup d'univers ont dû être rendus explicites. En effet, par une déclaration minutieuse des lemmes de paramétricité, il est possible en théorie de garantir un invariant selon lequel un témoin de paramétricité requiert autant d'univers que le but initial. Dans cet objectif, les lemmes de paramétricité doivent être déclarés en posant une contrainte de taille maximale sur leur instance d'univers, ce qui perturbe l'inférence des univers de Coq. La plupart des univers sont alors à annoter manuellement pour avoir un contrôle maximal sur le terme qui est réellement déclaré. Il a fallu pour cela ajouter de nouvelles fonctionnalités à Coq-Elpi pour permettre la manipulation de termes univers-polymorphes, d'instances d'univers, *etc.*

Par exemple, le lemme de paramétricité permettant de construire un témoin de paramétricité pour un produit dépendant, présenté en § 13.1.1, est en réalité déclaré de la façon suivante en Coq :

```
Definition Param^γ_Π@{i j k | i ⩽ k, j ⩽ k}
  (A A' : Type@{i}) (A_R : Param^α@{i} A A')
  (B : A → Type@{j}) (B' : A' → Type@{j})
  (B_R : forall a a' a_R, Param^β@{j} (B a) (B' a')) :
    Param^γ@{k} (forall (a : A), B a) (forall (a' : A'), B' a').
```

L'explicitation des univers dans le type de la déclaration revient en fait à une désactivation de l'ambiguïté typique, car Coq n'est plus autorisé à inférer plus d'univers liés ni de contraintes que ce qui est spécifié dans l'entête de cette déclaration, ce qui mène à un besoin d'annoter le terme à la main. En effet, dans le cadre du polymorphisme d'univers, l'assistant de preuve peut librement ajouter des contraintes d'univers que l'on n'a pas anticipées ou que l'on ne souhaite pas. Ces contraintes inférées sont logiques et nécessaires pour que le terme type bien, mais elles peuvent venir d'une mauvaise annotation manuelle. Si l'instance d'univers est bloquée *a priori* en explicitant l'entête de la déclaration comme ci-dessus, alors ces contraintes supplémentaires silencieuses deviennent des erreurs de typage. Par itération sur les erreurs fournies, on peut alors concevoir la déclaration correspondant aux annotations minimales souhaitées. L'équivalent de l'entête de la déclaration dans Coq-Elpi est le suivant :

```
@udecl! [I, J, K] ff [le I K, le J K] ff ⇒
  coq.env.add-const %...
```

La valeur `ff` représente le booléen *faux* indiquant que les listes d'univers et de contraintes ne peuvent être étendues par Coq. Passer l'un de ces booléens à *vrai* revient à ajouter un `+` après l'une des deux listes dans la déclaration Coq, et rend

à l'assistant de preuve la liberté d'ajouter des valeurs. Une constante `K` peut ensuite être instanciée avec `pglobal K UI`, où `UI` est l'instance d'univers obtenue à partir d'une liste de variables d'univers :

```
coq.univ-instance UI [I₀, J₀, K₀]
```

13.4.2 Univers algébriques et univers liés

En réalité, la taille des instances d'univers dans les lemmes de paramétricité pourrait encore être réduite. Le fait de créer un nouvel univers frais pour un produit dépendant et de le contraindre à être au-dessus des univers du domaine et du codomaine, comme c'est le cas dans le lemme de paramétricité présenté dans la sous-section précédente, peut être évité par l'utilisation des univers *algébriques*, c'est-à-dire créés à partir d'autres univers avec deux opérations : le maximum de deux univers et le successeur d'un univers. En effet, la règle de typage du produit dépendant ne fait intervenir que deux univers :

$$\frac{\Gamma \vdash A : \square_i \qquad \Gamma, a : A \vdash B : \square_j}{\Gamma \vdash \Pi a : A.\, B : \square_{\max(i,j)}}$$

Ainsi, le lemme de paramétricité ne nécessite en théorie que deux univers liés, et le lemme pour les univers n'en requiert qu'un seul, un univers étant toujours lié à lui-même. Par conséquent, il est possible de maintenir l'invariant selon lequel le but associé et le témoin de paramétricité utilisent autant d'univers que le but initial.

Cependant, l'implémentation actuelle du polymorphisme d'univers en Coq ne permet pas ce type de déclarations. En effet, le lemme de paramétricité sur les produits dépendants construit un témoin de paramétricité sous la forme d'un enregistrement, c'est-à-dire un type inductif. Une telle valeur est nécessairement obtenue par le constructeur d'enregistrements correspondant. Or, pour une raison technique dans l'implémentation actuelle de l'algorithme de vérification du graphe de contraintes d'univers dans Coq, on ne peut instancier une constante avec un univers algébrique. Il est donc nécessaire de passer par un univers frais et des contraintes supplémentaires.

Ainsi, pour chaque produit dépendant ou univers apparaissant dans le terme d'entrée, l'implémentation de la relation de paramétricité crée un univers frais. De plus, sans mécanisme de mémoire spécifique, rencontrer deux fois le même univers crée deux univers frais différents. Tout cela rend difficile le suivi des univers utilisés dans la traversée du but.

Enfin, ce problème empêche l'implémentation d'une fonctionnalité de transfert de preuve en chaînage avant au sein de Trocq. En effet, plutôt que de traduire un but initial G à prouver au sein de Coq en un but associé G', il pourrait être intéressant, en inversant les témoins de paramétricité de l'utilisateur dans la base de connaissances, de traduire un lemme p' en un lemme p utilisable dans une preuve dans le contexte sélectionné par l'utilisateur pour sa formalisation. Or, dans un tel cas, lors de la déclaration de p, l'entête contiendrait beaucoup d'univers liés inutiles, rendant difficile d'instancier le terme de façon optimale par la suite. On pourrait alors obtenir des preuves mal typées si on laisse Coq effectuer l'inférence de l'instance d'univers, et l'annotation manuelle demanderait un travail conséquent et artificiel puisque ces univers liés n'ont pas de raison d'être pertinente.

Dans le cadre de l'usage du polymorphisme d'univers en Coq-Elpi, divers prédicats peuvent être mis au point, comme un prédicat `coq.univ.super` pour obtenir le successeur d'un univers, ou bien `coq.univ.max` pour obtenir le maximum de deux univers, permettant ainsi de forger des univers algébriques arbitraires. Pourtant, malgré la possibilité de rendre accessibles ces détails bas niveau depuis le méta-langage, au moment de la traduction du terme dans Coq, l'univers algébrique deviendra inéluctablement une variable d'univers fraîche accompagnée de contraintes d'univers.

Conclusion et perspectives

Contributions

Dans ce document, nous avons présenté les deux prototypes de greffons de transfert de preuve pour l'assistant de preuve Coq développés au long de cette thèse. Le contexte général de ce travail est la recherche de solutions pour permettre à un utilisateur de se servir dans ses preuves de plusieurs formalisations d'un même concept mathématique de façon transparente, tout en gardant l'interopérabilité entre toutes les preuves effectuées sur une théorie donnée, quelle que soit la représentation choisie dans chaque preuve.

Le premier prototype, Trakt, permet d'améliorer l'automatisation des preuves d'énoncés de la famille SMT en reformulant ces énoncés et en les exprimant sous une forme canonique adaptée au format d'entrée des outils de preuve automatique disponibles dans Coq. Le projet ajoute aux outils d'automatisation existants une prise en charge de la théorie de la congruence et des fonctions non interprétées, ainsi qu'un traitement de la logique plus flexible, permettant de s'adapter aux besoins des différentes tactiques d'automatisation. La tactique `trakt` a notamment été intégrée avec succès au greffon SMTCoq *via* le projet Sniper, faisant fonctionner ensemble plusieurs outils de pré-traitement des énoncés.

Dans Trakt, les différentes représentations d'un même objet mathématique sont reliées par des isomorphismes, ou plongements partiels dans le cas d'un sous-type, et l'outil est centré sur les buts de la famille SMT, cible des prouveurs automatiques que l'on souhaite exécuter à sa suite. Ce pré-traitement permettant de faire un pont entre la preuve automatique et la preuve interactive est une instance du problème plus général du transfert de preuve, cible du second prototype, Trocq. L'ingrédient principal de cette seconde contribution est un nouveau cadre de paramétricité modulaire capable de gérer une classe de relations plus générale que les approches précédentes de la paramétricité brute et de la paramétricité univalente. La modularité de Trocq réside dans une hiérarchie de témoins de paramétricité exploitée pour effectuer du transfert de preuve en évitant le plus possible d'avoir recours à des axiomes lorsqu'ils ne sont pas nécessaires.

Perspectives

L'objectif social de la preuve formelle est d'instaurer une grande confiance entre les humains grâce à une meilleure confiance dans les résultats des chercheurs, ingénieurs, logiciens, *etc.*, c'est-à-dire des preuves et programmes informatiques certifiés sans faille. Les différentes implémentations actuelles des assistants de preuve dérivent toutes de pistes différentes dans la recherche du meilleur cadre logique pour effectuer ces preuves formelles, et au sein de chacun de ces outils, il existe un large spectre de techniques de formalisation. Cette explosion cambrienne permet d'explorer un vaste espace de possibilités, mais isole les différents travaux de formalisation les uns des autres.

Tout apport de bonnes pratiques venant des domaines du développement logiciel et de la théorie des langages de programmation est donc bienvenu pour les utilisateurs des assistants de preuve qui, de fait, en choisissant ce type de logiciel, acceptent le compromis d'un haut niveau de confiance au prix de preuves plus (bien souvent trop) manuelles. Idéalement, un utilisateur effectuant un travail de formalisation souhaite pouvoir utiliser les structures de données qu'il juge les plus pratiques, sans avoir à gérer des équivalences manuellement pour que son développement se conforme aux encodages communs au sein de la communauté utilisatrice de l'assistant de preuve qu'il a choisi. À l'inverse, si une formalisation a déjà été effectuée sur un encodage d'un concept mathématique, l'utilisateur souhaite pouvoir utiliser ce travail sur un autre encodage du même concept, s'il est avéré qu'ils sont équivalents. Les mécanismes de transfert de preuve sont un excellent intermédiaire dans ce type de situations et participent à la factorisation des efforts de preuve.

Néanmoins, il reste encore une marche à franchir pour transformer les prototypes développés pendant cette thèse en véritables outils d'assistance à la preuve. En effet, les assistants de preuve étant des objets de recherche, le développement de leur outillage se fait en même temps que le développement de leur formalisme logique lui-même ainsi que celui des techniques de méta-programmation. Malgré certaines bases déjà stables dans ces outils, comme la prise en charge du Calcul des Constructions, ou l'utilisation de la programmation logique pour effectuer des traductions syntaxiques, certaines subtilités, principalement liées aux univers, demeurent instables et retardent leur arrivée à maturité.

D'un point de vue plus pratique, ces prototypes pourraient être rendus plus utilisables par le perfectionnement des commandes servant à ajouter des informations en base de données. Dans le cas de Trocq, on pourrait imaginer une commande qui génère tous les témoins de paramétricité possibles liant un type inductif à lui-même. Ce type de preuve semble toujours possible en théorie et permettrait d'effectuer gratuitement un transfert de types de données utilisateur dans un but arbitrairement complexe. Une gestion de l'univers imprédicatif $\mathbb{P}$ permettrait de rapprocher l'outil de la version standard de Coq plutôt que de reposer sur la bibliothèque HoTT. Enfin, on peut imaginer, pour Trakt comme pour Trocq, de construire des bibliothèques de preuves à ajouter en base de données dès l'importation du greffon, afin que l'utilisateur puisse se lancer sans effort dans le transfert de preuve lorsqu'il concerne les types de données répandus dans les formalisations. Un mathématicien pourrait apprécier que Trocq soit équipé une bibliothèque de témoins de paramétricité déjà prouvés pour les types de données de la bibliothèque MathComp, afin de ne pas avoir à effectuer ces preuves lui-même, et pouvoir transformer facilement, par exemple, une matrice en liste de listes.

L'aboutissement de ces outils de transfert de preuve rendrait plus simples les raisonnements modulo équivalence au sein de l'assistant de preuve. Rapprocher la construction des preuves formelles du raisonnement intuitif que l'on peut faire sur papier attire de nouveaux utilisateurs, permettant de tendre un peu plus vers un monde sans faille logicielle, avec tous les avantages sociétaux que cela apporte.

Références

Voici les références dans leur ordre de citation.

[1] Gottlob Frege. « Begriffsschrift : Eine der Arithmetischen Nachgebildete Formelsprache des Reinen Denkens ». In : (1882) (cité à la page 3).

[2] Giuseppe Peano. « Arithmetices principia : Nova methodo exposita ». In : (1889) (cité à la page 3).

[3] Georg Cantor. « Grundlagen einer allgemeinen Mannigfaltigkeitslehre. Ein mathematisch-philosophischer Versuch in der Lehre des Unendlichen ». In : (1883) (cité à la page 3).

[4] Alain Colmerauer *et coll.* « Un système de communication homme-machine en français ». In : *Rapport préliminaire, Groupe de Recherche en Intelligence Artificielle* (1973) (cité aux pages 3, 37).

[5] Leslie Lamport. « The Temporal Logic of Actions ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), p. 872-923 (cité à la page 4).

[6] Jean-Christophe Filliâtre et Andrei Paskevich. « Why3 — Where Programs Meet Provers ». In : *Programming Languages and Systems : 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer. 2013, p. 125-128 (cité à la page 4).

[7] Niki Vazou. *Liquid Haskell : Haskell as a Theorem Prover*. University of California, San Diego, 2016 (cité à la page 4).

[8] Tobias Nipkow, Markus Wenzel et Lawrence C Paulson. *Isabelle/HOL : a proof assistant for higher-order logic*. Springer, 2002 (cité aux pages 4, 32).

[9] Ulf Norell. « Dependently Typed Programming in Agda ». In : *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Sous la dir. de Pieter W. M. Koopman, Rinus Plasmeijer et S. Doaitse Swierstra. T. 5832. Lecture Notes in Computer Science. Springer, 2008, p. 230-266. doi : 10.1007/978-3-642-04652-0_5 (cité à la page 5).

[10] Leonardo De Moura et Sebastian Ullrich. « The Lean 4 Theorem Prover and Programming Language ». In : *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*. Sous la dir. d'André Platzer et Geoff Sutcliffe. T. 12699. Lecture Notes in Computer Science. Springer, 2021, p. 625-635. doi : 10.1007/978-3-030-79876-5_37 (cité à la page 5).

[11] The Coq Development Team. *The Coq Proof Assistant*. Version 8.16. Sept. 2022. doi : 10.5281/zenodo.7313584 (cité aux pages 5, 8).

[12] Valentin Blot *et coll.* « Compositional pre-processing for automated reasoning in dependent type theory ». In : *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 2023, p. 63-77 (cité aux pages 5, 46, 56, 69, 72).

[13] John C. Reynolds. « Types, Abstraction and Parametric Polymorphism ». In : *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Sous la dir. de R. E. A. Mason. North-Holland/IFIP, 1983, p. 513-523 (cité aux pages 6, 36, 79).

[14] Nicolas Tabareau, Éric Tanter et Matthieu Sozeau. « The marriage of univalence and parametricity ». In : *Journal of the ACM (JACM)* 68.1 (2021), p. 1-44 (cité aux pages 6, 77, 82, 86, 98, 104).

[15] Cyril Cohen, Enzo Crance et Assia Mahboubi. « Trocq : Proof Transfer for Free, With or Without Univalence ». In : *European Symposium on Programming*. 2024 (cité aux pages 6, 78).

[16] Enrico Tassi. « Elpi : an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect) ». In : (2018) (cité aux pages 6, 9, 37).

[17] Frédéric Besson. « Fast Reflexive Arithmetic Tactics the Linear Case and Beyond ». In : *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*. Sous la dir. de Thorsten Altenkirch et Conor McBride. T. 4502. Lecture Notes in Computer Science. Springer, 2006, p. 48-62. doi : 10.1007/978-3-540-74464-1_4 (cité aux pages 8, 32, 47).

[18] Alonzo Church. « A set of postulates for the foundation of logic ». In : *Annals of mathematics* (1933), p. 839-864 (cité à la page 10).

[19] Alan M Turing. « Computability and λ-definability ». In : *The Journal of Symbolic Logic* 2.4 (1937), p. 153-163 (cité à la page 11).

[20] Alonzo Church. « A formulation of the simple theory of types ». In : *The journal of symbolic logic* 5.2 (1940), p. 56-68 (cité à la page 11).

[21] Henk P Barendregt. « Introduction to generalized type systems ». In : (1991) (cité à la page 13).

[22] Stefano Berardi. « Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube ». In : *Technical report, Carnegie-Mellon University (USA) and Università di Torino (Italy)* (1988) (cité à la page 13).

[23] Jan Terlouw. « Een nadere bewijstheoretische analyse van GSTT's ». In : *Manuscript* (1989) (cité à la page 13).

[24] Thierry Coquand et Gérard Huet. « The calculus of constructions ». Thèse de doct. INRIA, 1986 (cité à la page 14).

[25] Jean-Yves Girard. « Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur ». In : (1972) (cité à la page 15).

[26] Gottlob Frege. *Grundgesetze der Arithmetik*. 1903 (cité à la page 15).

[27] Matthieu Sozeau et Nicolas Tabareau. « Universe polymorphism in Coq ». In : *International Conference on Interactive Theorem Proving*. Springer. 2014, p. 499-514 (cité à la page 17).

[28] Christine Paulin-Mohring. « Définitions Inductives en Théorie des Types ». Thèse de doct. Université Claude Bernard-Lyon I, 1996 (cité à la page 17).

[29] Philip Wadler. « Monads for functional programming ». In : *Advanced Functional Programming : First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*. Springer. 1995, p. 24-52 (cité à la page 22).

[30] Roger Hindley. « The principal type-scheme of an object in combinatory logic ». In : *Transactions of the american mathematical society* 146 (1969), p. 29-60 (cité à la page 24).

[31] Robin Milner. « A theory of type polymorphism in programming ». In : *Journal of computer and system sciences* 17.3 (1978), p. 348-375 (cité à la page 24).

[32] Cordelia V Hall *et coll.* « Type classes in Haskell ». In : *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.2 (1996), p. 109-138 (cité à la page 27).

[33] Bruno CdS Oliveira, Adriaan Moors et Martin Odersky. « Type classes as objects and implicits ». In : *ACM Sigplan Notices* 45.10 (2010), p. 341-360 (cité à la page 27).

[34] Matthieu Sozeau et Nicolas Oury. « First-class type classes ». In : *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2008, p. 278-293 (cité à la page 27).

[35] Assia Mahboubi et Enrico Tassi. *Mathematical Components*. Zenodo, jan. 2021 (cité aux pages 28, 49).

[36] Cyril Cohen, Kazuhiko Sakaguchi et Enrico Tassi. « Hierarchy Builder : algebraic hierarchies made easy in Coq with Elpi ». In : *FSCD 2020-5th International Conference on Formal Structures for Computation and Deduction*. 167. 2020, p. 34-1 (cité à la page 28).

[37] Assia Mahboubi et Enrico Tassi. « Canonical structures for the working Coq user ». In : *International Conference on Interactive Theorem Proving*. Springer. 2013, p. 19-34 (cité à la page 28).

[38] David Delahaye. « A tactic language for the system Coq ». In : *Logic for Programming and Automated Reasoning : 7th International Conference, LPAR 2000 Reunion Island, France, November 6–10, 2000 Proceedings 7*. Springer. 2000, p. 85-95 (cité à la page 29).

[39] Sascha Böhme et Tobias Nipkow. « Sledgehammer : judgement day ». In : *Automated Reasoning : 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*. Springer. 2010, p. 107-121 (cité à la page 32).

[40] Łukasz Czajka et Cezary Kaliszyk. « Hammer for Coq : Automation for dependent type theory ». In : *Journal of automated reasoning* 61 (2018), p. 423-453 (cité aux pages 32, 67).

[41] Burak Ekici *et coll.* « SMTCoq : A Plug-In for Integrating SMT Solvers into Coq ». In : *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Sous la dir. de Rupak Majumdar et Viktor Kuncak. T. 10427. Lecture Notes in Computer Science. Springer, 2017, p. 126-133. doi : 10.1007/978-3-319-63390-9_7 (cité aux pages 32, 47).

[42] Clark Barrett, Aaron Stump, Cesare Tinelli *et coll.* « The SMT-LIB Standard : Version 2.0 ». In : *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. T. 13. 2010, p. 14 (cité à la page 32).

[43] Matthieu Sozeau. « A New Look at Generalized Rewriting in Type Theory ». In : *J. Formaliz. Reason.* 2.1 (2009), p. 41-62. doi : 10.6092/issn.1972-5787/1574 (cité aux pages 35, 104).

[44] Gilles Barthe, Venanzio Capretta et Olivier Pons. « Setoids in type theory ». In : *J. Funct. Program.* 13.2 (2003), p. 261-293. doi : 10.1017/S0956796802004501 (cité à la page 35).

[45] Cyril Cohen, Maxime Dénès et Anders Mörtberg. « Refinements for Free ! » In : *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Sous la dir. de Georges Gonthier et Michael Norrish. T. 8307. Lecture Notes in Computer Science. Springer, 2013, p. 147-162. doi : 10.1007/978-3-319-03545-1_10 (cité aux pages 35, 104).

[46] Matthieu Sozeau *et coll.* « The MetaCoq Project ». In : *J. Autom. Reason.* 64.5 (2020), p. 947-999. doi : 10.1007/s10817-019-09540-0 (cité aux pages 37, 104).

[47] Pierre-Marie Pédrot. « Ltac2 : tactical warfare ». In : *The Fifth International Workshop on Coq for Programming Languages, CoqPL*. 2019, p. 13-19 (cité à la page 37).

[48] Cvetan Dunchev *et coll.* « ELPI : Fast, Embeddable, λProlog Interpreter ». In : *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer. 2015, p. 460-468 (cité à la page 37).

[49] Dale Miller et Gopalan Nadathur. *A logic programming approach to manipulating formulas and programs*. University of Pennsylvania. Moore School of Electrical Engineering …, 1987 (cité à la page 38).

[50] Frank Pfenning et Conal Elliott. « Higher-Order Abstract Syntax ». In : *ACM sigplan notices* 23.7 (1988), p. 199-208 (cité à la page 38).

[51] Thom Frühwirth. « Constraint Handling Rules ». In : *French School on Theoretical Computer Science*. Springer, 1994, p. 90-107 (cité aux pages 39, 122).

[52] Nicolaas Govert De Bruijn. « Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem ». In : *Indagationes Mathematicae (Proceedings)*. T. 75. 5. Elsevier. 1972, p. 381-392 (cité à la page 40).

[53] Frédéric Besson. « ppsimpl : a reflexive Coq tactic for canonising goals ». In : *Coq Workshop on Programming Languages*. https://popl17.sigplan.org/details/main/3/ppsimpl-a-reflexive-Coq-tactic-for-canonising-goals. 2017 (cité à la page 51).

[54] Kazuhiko Sakaguchi. *Micromega tactics for Mathematical Components*. Version 1.12. 2019–2022 (cité à la page 53).

[55] Frédéric Besson. « Itauto : An Extensible Intuitionistic SAT Solver ». In : *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Sous la dir. de Liron Cohen et Cezary Kaliszyk. T. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 9 :1-9 :18. doi : 10.4230/LIPIcs.ITP.2021.9 (cité aux pages 54, 68).

[56] Andrew W. Appel. « Verified Software Toolchain - (Invited Talk) ». In : *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Sous la dir. de Gilles Barthe. T. 6602. Lecture Notes in Computer Science. Springer, 2011, p. 1-17. doi : 10.1007/978-3-642-19718-5_1 (cité à la page 69).

[57] Andrew W. Appel *et coll. Verifiable C*. 2022 (cité à la page 69).

[58] John C. Mitchell. « Representation Independence and Data Abstraction ». In : *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*. ACM Press, 1986, p. 263-276. doi : 10.1145/512644.512669 (cité à la page 77).

[59] Simon Boulier, Pierre-Marie Pédrot et Nicolas Tabareau. « The next 700 syntactical models of type theory ». In : *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Sous la dir. d'Yves Bertot et Viktor Vafeiadis. ACM, 2017, p. 182-194. doi : 10.1145/3018610.3018620 (cité aux pages 77, 86).

[60] Philip Wadler. « Theorems for Free ! » In : *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. Sous la dir. de Joseph E. Stoy. ACM, 1989, p. 347-359. doi : 10.1145/99370.99404 (cité à la page 80).

[61] Jean-Philippe Bernardy et Marc Lasson. « Realizability and Parametricity in Pure Type Systems ». In : *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Sous la dir. de Martin Hofmann. T. 6604. Lecture Notes in Computer Science. Springer, 2011, p. 108-122. doi : 10.1007/978-3-642-19805-2_8 (cité à la page 80).

[62] Jean-Philippe Bernardy, Patrik Jansson et Ross Paterson. « Proofs for free - Parametricity for dependent types ». In : *J. Funct. Program.* 22.2 (2012), p. 107-152. doi : 10.1017/S0956796812000056 (cité à la page 80).

[63] Chantal Keller et Marc Lasson. « Parametricity in an Impredicative Sort ». In : *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*. Sous la dir. de Patrick Cégielski et Arnaud Durand. T. 16. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, p. 381-395. doi : 10.4230/LIPIcs.CSL.2012.381 (cité à la page 80).

[64] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations of Mathematics*. Institute for Advanced Study : https://homotopytypetheory.org/book, 2013 (cité aux pages 82, 83, 89).

[65] Andrej Bauer *et coll.* « The HoTT library : a formalization of homotopy type theory in Coq ». In : *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Sous la dir. d'Yves Bertot et Viktor Vafeiadis. ACM, 2017, p. 164-172. doi : 10.1145/3018610.3018615 (cité aux pages 84, 88).

[66] Talia Ringer *et coll.* « Proof repair across type equivalences ». In : *PLDI '21 : 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Sous la dir. de Stephen N. Freund et Eran Yahav. ACM, 2021, p. 112-127. doi : 10.1145/3453483.3454033 (cité à la page 85).

[67] David Aspinall et Adriana B. Compagnoni. « Subtyping dependent types ». In : *Theor. Comput. Sci.* 266.1-2 (2001), p. 273-309. doi : 10.1016/S0304-3975(00)00175-4 (cité à la page 99).

[68] Talia Ringer *et coll.* « Proof repair across type equivalences ». In : *PLDI '21 : 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Sous la dir. de Stephen N. Freund et Eran Yahav. ACM, 2021, p. 112-127. doi : 10.1145/3453483.3454033 (cité à la page 104).

[69] Gilles Barthe et Olivier Pons. « Type Isomorphisms and Proof Reuse in Dependent Type Theory ». In : *Foundations of Software Science and Computation Structures*. Sous la dir. de Furio Honsell et Marino Miculan. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001, p. 57-71 (cité à la page 104).

[70] Nicolas Magaud. « Changing Data Representation within the Coq System ». In : *TPHOLs'2003*. T. 2758. © Springer-Verlag. LNCS, Springer-Verlag, 2003 (cité à la page 104).

[71] Neelakantan R. Krishnaswami et Derek Dreyer. « Internalizing Relational Parametricity in the Extensional Calculus of Constructions ». In : *Computer Science Logic 2013 (CSL 2013)*. Sous la dir. de Simona Ronchi Della Rocca. T. 23. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany : Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, p. 432-451. doi : 10.4230/LIPIcs.CSL.2013.432 (cité à la page 104).

[72] Maxime Dénès, Anders Mörtberg et Vincent Siles. « A Refinement-Based Approach to Computational Algebra in Coq ». In : *Interactive Theorem Proving*. Sous la dir. de Lennart Beringer et Amy Felty. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, p. 83-98 (cité à la page 104).

[73] Peter Lammich. « Automatic Data Refinement ». In : *Interactive Theorem Proving*. Sous la dir. de Sandrine Blazy, Christine Paulin-Mohring et David Pichardie. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 84-99 (cité à la page 104).

[74] Florian Haftmann *et coll.* « Data Refinement in Isabelle/HOL ». In : *Interactive Theorem Proving*. Sous la dir. de Sandrine Blazy, Christine Paulin-Mohring et David Pichardie. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 100-115 (cité à la page 104).

[75] Brian Huffman et Ondřej Kunčar. « Lifting and Transfer : A Modular Design for Quotients in Isabelle/HOL ». In : *Certified Programs and Proofs*. Sous la dir. de Georges Gonthier et Michael Norrish. Cham : Springer International Publishing, 2013, p. 131-146 (cité à la page 104).

[76] Peter Lammich et Andreas Lochbihler. « Automatic Refinement to Efficient Data Structures : A Comparison of Two Approaches ». In : *J. Autom. Reason.* 63.1 (2019), p. 53-94. doi : 10.1007/s10817-018-9461-9 (cité à la page 104).

[77] Théo Zimmermann et Hugo Herbelin. « Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant ». In : *Conference on Intelligent Computer Mathematics*. Washington, D.C., United States, 2015 (cité à la page 104).

[78] Carlo Angiuli *et coll.* « Internalizing representation independence with univalence ». In : *Proc. ACM Program. Lang.* 5.POPL (2021), p. 1-30. doi : 10.1145/3434293 (cité à la page 104).

[79] Abhishek Anand et Greg Morrisett. *Revisiting Parametricity : Inductives and Uniformity of Propositions*. 2017 (cité à la page 104).

[80] Nicolas Tabareau, Éric Tanter et Matthieu Sozeau. « Equivalences for free : univalent parametricity for effective transport ». In : *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), p. 1-29 (cité à la page 105).

[81] Xavier Allamigeon, Quentin Canu et Pierre-Yves Strub. « A Formal Disproof of Hirsch Conjecture ». In : *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*. Sous la dir. de Robbert Krebbers *et coll.* ACM, 2023, p. 17-29. doi : 10.1145/3573105.3575678 (cité à la page 105).

[82] Joxan Jaffar et Jean-Louis Lassez. « Constraint Logic Programming ». In : *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1987, p. 111-119 (cité à la page 127).

COLLEGE ED MATHEMATIQUES
DOCTORAL STIC
PAYS DE LA LOIRE

Nantes Université

Titre : Méta-programmation pour le transfert de preuve en théorie des types dépendants

Mot clés : preuve formelle, automatisation des preuves, méta-programmation

Résumé : En mathématiques comme en informatique, il est d'usage de faire appel à des outils numériques de vérification pour augmenter la confiance dans les preuves et les logiciels. La pratique la plus commune est le test, mais elle est limitée. Les *assistants de preuve interactifs* sont des outils permettant d'effectuer des preuves avec une grande confiance, laissant l'humain trouver les idées des preuves tout en vérifiant méticuleusement que toutes les étapes de la preuve sont valides. Cette thèse s'inscrit dans une lignée de travaux visant à automatiser les preuves, avec l'objectif final de répandre l'usage des assistants de preuve à la place du test logiciel, partout où cela est possible et pertinent. On s'intéresse ici au partage de théorie formelle entre plusieurs représentations différentes d'un même concept mathématique, ou plusieurs implémentations d'une même spécification. Sur le plan théorique, cette étude s'appuie sur l'analyse de traductions de paramétricité pour le Calcul des Constructions, et en propose une généralisation. Ces résultats s'incarnent dans la conception de deux outils de transfert de preuve, TRAKT et TROCQ, dont on discute ici l'implémentation, à l'aide du méta-langage COQ-ELPI.

Title: Meta-Programming for Proof Transfer in Dependent Type Theory

Keywords: formal proof, proof automation, meta-programming

Abstract: In both mathematics and computer science, it is common practice to use digital verification tools to increase confidence in proofs and software. The most common practice is testing, but it is limited. *Interactive proof assistants* are tools made to perform proofs with high confidence, letting humans come up with proof ideas while meticulously checking that all proof steps are valid. This thesis is part of a line of work aimed at automating proofs, with the ultimate goal of spreading the use of proof assistants in place of software testing, wherever possible and relevant. Here, we are interested in sharing of formal theory between several different representations of the same mathematical concept, or several implementations of the same specification. From a theoretical point of view, this study is based on the analysis of parametricity translations for the Calculus of Constructions, and proposes a generalisation of them. These results are made concrete in the design of two proof transfer tools, TRAKT and TROCQ, whose implementation is discussed here, using the COQ-ELPI meta-language.