# Prétraitement compositionnel en Coq

Louise Dubois de Prisque

université PARIS-SACLAY

# Prétraitement compositionnel en Coq

*Compositional preprocessing in Coq*

**Thèse de doctorat de l'université Paris-Saclay**

École doctorale n°580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : ISN. Référent : ENS Paris-Saclay

Thèse préparée à l'unité de recherche Laboratoire Méthodes Formelles (Université Paris-Saclay, CNRS, ENS Paris-Saclay), sous la direction de **Gilles DOWEK**, Directeur de Recherche, le co-encadrement de **Chantal KELLER**, Maîtresse de conférences et le co-encadrement de **Valentin BLOT**, Chargé de Recherche

**Thèse soutenue à Paris-Saclay, le 10 Juillet 2024, par**

**Louise DUBOIS DE PRISQUE**

## Composition du jury

Membres du jury avec voix délibérative

| | |
|---|---|
| **Sylvie BOLDO**<br>Directrice de Recherche, INRIA | Présidente & Examinatrice |
| **Sylvain BOULMÉ**<br>Maître de Conférences, Université Grenoble-Alpes | Rapporteur & Examinateur |
| **Nicolas TABAREAU**<br>Directeur de Recherche, INRIA | Rapporteur & Examinateur |
| **Sophie TOURRET**<br>Chargée de Recherche, INRIA | Examinatrice |

THESE DE DOCTORAT
NNT : 2024UPASG040

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

**Titre :** Prétraitement compositionnel en `Coq`
**Mots clés :** Assistant de preuve, prouveur automatique, métaprogrammation

**Résumé :** Cette thèse présente une nouvelle méthodologie de prétraitement de buts de l'assistant de preuve Coq afin de les envoyer à un prouveur automatique. Cette méthodologie consiste à composer des transformations atomiques et certifiantes en fonction du but à prouver. Nous présenterons d'abord la bibliothèque de transformations que nous avons écrite, puis nous présenterons un outil d'ordonnancement pour ces transformations, qui se veut générique. Le tout a conduit à l'implémentation une nouvelle tactique « pousse-bouton » appelée « `snipe` » pour Coq.

**Title :** Compositional preprocessing in `Coq`
**Keywords :** Proof assistant, automatic solver, metaprogramming

**Abstract :** This thesis presents a new methodology for preprocessing goals in the Coq proof assistant in order to send them to an automatic solver. This methodology consists of composing atomic and certifying transformations based on the goal to be proven. We will first present the library of transformations that we have written, then we will present an orchestrating tool for these transformations, which aims to be generic. This has led to the implementation of a new "push-button" tactic called `snipe` for Coq.

# Remerciements

Je dédie cette thèse à mon père, que j'ai perdu fin 2022, au beau milieu de celle-ci. Je suis sûre que tu aurais été très heureux d'en découvrir l'aboutissement et d'assister à ma soutenance.

J'aimerais remercier en premier mes deux encadrant·es de thèse : Chantal Keller et Valentin Blot.

Chantal, tu m'as beaucoup aidée à gagner en assurance, tu es parvenue à me guider de très près lors de ma première année, notamment en me poussant à publier très tôt. Puis, lors des années qui ont suivi, la confiance et la liberté que tu m'as accordées m'ont permis d'explorer plus d'idées à ma guise. C'était souvent difficile, et je sais que sans toi je n'aurais jamais été jusqu'ici. Merci encore.

Valentin, j'ai toujours été impressionnée par ta capacité à faire des liens entre les différents domaines, ma thèse est plutôt axée sur la pratique mais grâce à toi j'ai pu comprendre dans quelles théories elle pouvait s'inscrire. Je te remercie pour ta gentillesse et ton écoute.

Je remercie aussi mon directeur de thèse Gilles Dowek, qui m'a suivie de plus loin mais qui a toujours été disponible en cas de besoin. Merci Gilles pour ta confiance et ta bienveillance.

Je remercie les membres du jury : Sylvain Boulmé et Nicolas Tabareau pour leur relecture et leurs conseils précieux, ainsi que Sylvie Boldo, Yannick Forster et Sophie Tourret pour avoir bien voulu en faire partie.

Je remercie également toute l'équipe Deducteam et le Laboratoire Méthode Formelles, en particulier je remercie Amélie, Pierre, Rébecca, Thiago, Luc, Théo, Margot, Émilie, Alexis, ainsi que Pablo du Laboratoire d'Informatique de l'école Polytechnique.

Je remercie Assia, Enzo et Denis avec lesquel·les il fut très plaisant de travailler.

Parmi mes amis, celles et ceux qui ont écouté mes plaintes quand la thèse devenait difficile, ou mes envolées lyriques quand au contraire, j'avais l'impression de faire d'immenses progrès, ainsi que ceux qui m'ont emmenée me changer les idées pour mieux revenir pleine d'enthousiasme, je remercie Armance, Nolwenn, Flo, Esteban, Raquel, Théotime, Héloïse, Pinson, Yann, Hécate, Pablo, Mel, Melvin, Tantal... J'en oublie forcément, ne m'en voulez pas trop. Écrire tous ces noms me rappelle à quel point la thèse a traversé des époques de ma vie diverses, je n'en reviens pas d'avoir fait tout ce parcours...

Je remercie toute ma famille et je souhaite mentionner en particulier Clarisse, pour son soutien sans faille, Claire, Alexis et Augustin.

Merci à mes chiens Ozone et Thémis, pour le bruit qu'ils faisaient pendant mes visios importantes.

Et enfin, *last but not least*, merci à Emma, qui m'a patiemment fait répéter certains exposés, qui m'a encouragée mille fois quand je me sentais illégitime, en bref, qui m'a toujours soutenue. Je t'en suis très reconnaissante, même si les mots me manquent ici.

# Table des matières


**1 INTRODUCTION** **9**

1.1 Cadre de la thèse : les méthodes formelles . . . 9
1.1.1 Pourquoi faire des méthodes formelles ? . . . 9
1.1.2 Méthodes formelles et automatisation . . . 10
1.2 Les assistants de preuve . . . 10
1.3 Automatiser les assistants de preuve . . . 12
1.4 Un avant-goût des contributions . . . 13

**2 CONTEXTE** **17**

2.1 L'assistant de preuve `Coq` et sa logique . . . 17
2.1.1 De la logique minimale aux systèmes de types purs . . . 17
2.1.2 La logique de `Coq` : le Calcul des Constructions Inductives . . . 26
2.1.3 L'implémentation de `Coq` . . . 32
2.2 `SMTCoq`, le backend principal . . . 38
2.2.1 Fonctionnement . . . 38
2.2.2 Les prouveurs SAT et SMT . . . 40
2.2.3 Limitations de `SMTCoq` . . . 43

**3 MÉTHODOLOGIE** **49**

3.1 Définitions préalables . . . 49
3.1.1 Qu'est-ce qu'une transformation ? . . . 49
3.1.2 Quelques exemples . . . 50
3.2 Mantra méthodologique . . . 50
3.2.1 Le `Sniper` contre (ou avec ?) le `Hammer` . . . 51
3.2.2 Certifiant ou certifié ? . . . 55
3.3 État de l'art . . . 56
3.3.1 Le plugin `Tactician` . . . 56
3.3.2 Des prouveurs automatiques dans `Coq` . . . 57
3.4 Métaprogrammer en `Coq` . . . 58
3.4.1 Choix des métalangages . . . 59
3.4.2 Le premier langage de tactique : `Ltac` . . . 59
3.4.3 Une extension plus robuste : `Ltac2` . . . 68
3.4.4 Manipuler des termes `Coq` en `Coq` : `MetaCoq` . . . 70
3.4.5 Faire fi des lieurs : `Coq-Elpi` . . . 74

**4 TRANSFORMATIONS** **79**
4.1 Interprétation des fonctions . . . 79
4.1.1 Définitions . . . 80
4.1.2 Réflexivités et dépliage . . . 81
4.1.3 Expansion . . . 82
4.1.4 Réduction des points fixes . . . 83
4.1.5 Élimination du pattern matching . . . 85
4.1.6 Retour sur l'exemple initial . . . 87
4.2 Types algébriques . . . 87
4.2.1 Trois propriétés fondamentales . . . 88
4.2.2 Le principe de génération . . . 90
4.3 Relations inductives . . . 91
4.3.1 Les relations inductives . . . 91
4.3.2 Décidabilité . . . 92
4.3.3 Présentation de la commande `Decide` . . . 94
4.3.4 Prétraitement : linéarisation . . . 95
4.3.5 Génération du point fixe . . . 98
4.3.6 Heuristiques . . . 102
4.3.7 Exemple récapitulatif . . . 102
4.3.8 Travaux connexes . . . 103
4.3.9 Principe d'inversion . . . 104
4.4 Polymorphisme prénexe . . . 105
4.4.1 Définitions . . . 105
4.4.2 Stratégies . . . 106
4.5 Ordre supérieur . . . 109
4.5.1 Fonctions anonymes . . . 109
4.5.2 Élimination de l'ordre supérieur « prénexe » . . . 110

**5 UN ORDONNANCEUR** **111**
5.1 Naissance de l'idée . . . 111
5.1.1 Première version de la tactique `snipe` . . . 111
5.1.2 Déclencher des tactiques ? . . . 115
5.1.3 Symboles à ne pas interpréter . . . 116
5.2 Le plugin `Orchestrator` : *triggers* et filtres . . . 117
5.2.1 Présentation . . . 117
5.2.2 Le cœur de l'ordonnanceur : les *triggers* . . . 117
5.2.3 Les filtres . . . 126
5.3 Implémentation de l'ordonnanceur . . . 128
5.4 Preuve de concept : la tactique `snipe` à l'œuvre . . . 129

**6 CONCLUSION ET PERSPECTIVES** **133**
6.1 Conclusion . . . 133
6.2 Amélioration des transformations actuelles . . . 133
6.2.1 Les relations inductives . . . 133
6.2.2 Les points fixes anonymes . . . 134
6.3 Nouvelles transformations . . . 135
6.3.1 Encodage des applications partielles . . . 135
6.3.2 Défonctionnalisation . . . 136
6.3.3 Types dépendants de types . . . 136

6.4 Perspectives pour l'ordonnanceur . . . . . 139
6.4.1 Ajout d'états . . . . . 139
6.4.2 Une syntaxe plus *user friendly* . . . . . 140
6.4.3 Faire une évaluation . . . . . 140
6.4.4 Diversifier les applications . . . . . 141

# Chapitre 1

# INTRODUCTION

## 1.1 Cadre de la thèse : les méthodes formelles

### 1.1.1 Pourquoi faire des méthodes formelles ?

Les *méthodes formelles* sont un domaine de l'informatique où les propriétés des programmes, représentées par des formules logiques, sont établies par des démonstrations *formelles*, qui utilisent donc un langage à la syntaxe et à la sémantique rigoureusement définies. Cette rigueur provient de la *logique mathématique*, qui prit son essor à la fin du XIX$^{\text{ème}}$ et au début du XX$^{\text{ème}}$ siècle grâce aux efforts conjoints de mathématiciens et de philosophes comme Frege [23] ou Russell et Whitehead [47], qui s'intéressaient aux propriétés qui caractérisent un raisonnement correct, et aux fondations des théories mathématiques.

Le lien entre les *méthodes formelles* et la *logique mathématique* s'établit à la fois avec l'apparition des premiers modèles de calcul (des outils mathématiques pour modéliser le comportement des programmes) tels que le lambda-calcul [12], et grâce à la notion de *preuve formelle* qui leur est commune.

Ainsi, pour prouver qu'un programme a bien le comportement attendu, il n'est pas suffisant de donner un argument informel, il faut fournir toutes les étapes du raisonnement, chacune justifiée par une règle d'inférence. Ces étapes peuvent être *vérifiées* voire *produites* automatiquement.

Afin de justifier l'usage des méthodes formelles en informatique, on cite souvent des applications dans des domaines critiques, où le bug d'un programme pourrait causer une grande perte (en vies humaines, en argent...), et où, donc, une preuve de sa correction peut être préférée à une batterie de tests, en dépit d'un coût de développement et de temps important. L'aéronautique comme la Blockchain se sont donc intéressées aux recherches menées dans cette branche et ont repris des technologies ou des techniques des méthodes formelles, comme les *assistants de preuve*[1] ou les *prouveurs automatiques* [20], que nous décrirons plus tard.

Mais, pour introduire mon travail, j'aimerais argumenter que ce qui motive beaucoup de chercheurs et chercheuses en méthodes formelles, et ce qui m'a motivée tout au long de cette thèse, est bien plutôt une raison théorique. En effet, la plupart des mathématiques sont informelles, beaucoup d'étapes de l'argumentation sont passées sous silence. Parfois, même, la démonstration n'est même pas mentionnée. Qui n'est pas tombé, en consultant un livre de mathématiques, sur cette formule toujours quelque peu irritante : « La démonstration est triviale et nous la laissons au lecteur » ?

1. Par exemple, une formalisation en `Coq` de contrats écrits en Ethereum est disponible ici : `https://github.com/sec-bit/tokenlibs-with-proofs`.

En étudiant la logique mathématique, sur laquelle s'appuient les méthodes formelles, j'ai découvert un raisonnement à grains plus fins, dépourvu de tout implicite, et la satisfaction de ne plus rien laisser au hasard. La logique mathématique se fonde sur la thèse philosophique selon laquelle *une démonstration valide peut être écrite dans un langage formel*, et donc, pas seulement dans un langage naturel. Il nous faut un métalangage mathématique lui aussi, pour traiter des mathématiques ou des langages de programmation. Comme ce métalangage et ses propriétés peuvent être implémentées dans divers logiciels, les méthodes formelles sont à la croisée des mathématiques et de l'informatique.

Je suis convaincue que c'est cette rigueur qui motive principalement tant de chercheurs et chercheuses en logique mathématique ou en méthodes formelles, et pas toujours les applications qui justifient *a posteriori* certains de leurs travaux.

### 1.1.2 Méthodes formelles et automatisation

En m'intéressant, dans cette thèse, plus particulièrement à *l'automatisation* des preuves[2] au sein des méthodes formelles, la phrase sur la trivialité d'une démonstration écrite par l'auteur paresseux au lecteur sollicité devient complètement erronée. Automatiser un raisonnement soi-disant *trivial*, et en particulier dans le *Calcul des Constructions Inductives* qui m'a accompagné tout au long de cette thèse et que nous définirons en temps venu, est au contraire une tâche hautement difficile. Cependant, il est indéniable qu'une part des méthodes formelles consiste à appliquer des lemmes bureaucratiques, à gérer parfois des dizaines et des dizaines de cas faciles.

Bénéficier d'une meilleure automatisation au sein de logiciels utilisés dans les méthodes formelles est donc une demande récurrente de la communauté des chercheurs et ingénieurs du domaine. L'avantage est que, dans un logiciel, la preuve, bien que générée automatiquement, demeure toujours accessible, au sens où il est possible d'afficher l'objet de preuve, et de le vérifier à nouveau. Ainsi, étudier précisément ce qui s'automatise ou non nous donne une meilleure notion de ce qui est *trivial*, cela nous permet aussi de mieux définir ce qui est encore le propre du raisonnement mathématique et qui résiste à l'automatisation.

Une thèse sur l'automatisation comme celle-ci, bien que produisant des résultats concrets qui, je l'espère, serviront plus tard à ceux qui souhaitent utiliser mon développement logiciel, m'a également donné une compréhension plus fine de ce qui fait la complexité d'une démonstration.

## 1.2 Les assistants de preuve

Au sein des méthodes formelles, les logiciels appelés *assistants de preuve* sont particulièrement utilisés. Ils sont l'implémentation concrète d'une logique donnée $\mathcal{L}$, avec sa syntaxe et sa sémantique. L'assistant de preuve le plus rudimentaire doit au moins pouvoir fournir une interface pour écrire des termes de $\mathcal{L}$, des preuves des termes de $\mathcal{L}$, et vérifier *automatiquement* que ces preuves sont correctes. Autrement dit, un assistant de preuve automatise entièrement la tâche de « relecture » d'une preuve. Il n'automatise cependant pas toute sa construction. Elle est le plus souvent faite pas à pas. L'état de la preuve en cours (des hypothèses et une conclusion à prouver) s'appelle le *but*.

La partie du logiciel qui définit la logique, ses règles et permet de vérifier les preuves écrites par l'utilisateur·ice est appelée le *noyau logique*.

Par exemple, considérons un assistant de preuve qui implémente la *logique minimale*. Afin de donner à tous mes éventuels lecteurs, et en particulier ceux qui ne sont pas familiers du sujet,

2. Pour être plus précise, il s'agit plutôt, comme nous le verrons, d'un prétraitement qui permet, entre autres, d'automatiser plus facilement certaines preuves.

une intuition de ce qu'est un assistant de preuve, j'ai effectivement réalisé une implémentation documentée en `OCaml` de cet assistant, `Minlog`, disponible en suivant ce lien[3]. Son noyau contient une définition de la grammaire utilisée : les formules peuvent être des variables ou des suites d'implications, et les séquents sont des paires d'une liste de formules (les hypothèses), et d'une formule qui est la conclusion. Il contient aussi une énumération des règles de cette logique : la règle axiome, la règle du *modus ponens* et la règle d'abstraction, et un algorithme pour vérifier la validité d'une preuve donnée. Les preuves sont des arbres dont chaque nœud est étiqueté par un séquent (l'état de la preuve à ce moment) et une règle (appliquée pour passer au(x) séquent(s) fils de ce nœud).

Tout le reste du logiciel ne fait pas partie du noyau : les fonctions d'affichage, l'interface pour l'utilisateur·ice et même les fonctions appelées à chaque fois qu'il applique une règle sont codées dans des fichiers à part. L'utilisateur·ice peut très bien appliquer la règle axiome sur le séquent $A \vdash B$ même si c'est incorrect. En interne, le programme construit un arbre de preuve invalide. C'est au moment de la vérification par le noyau que le programme renverra une erreur, donc quand la preuve (erronée ou non) est terminée.

Pour en savoir plus, nous vous renvoyons à ce code et à sa documentation.

Notons cependant que cet exemple simpliste n'est pas réaliste : les assistants de preuve implémentent le plus souvent une logique expressive, pour que des énoncés mathématiques complexes puissent y être formalisés. Par exemple, les mathématiques utilisent abondamment des *quantificateurs*.

Les assistants de preuve fournissent également à leurs utilisateur·ices une interface d'affichage commode pour écrire les démonstrations, et, plus important, certains peuvent fournir ce qu'on appelle des *tactiques*, c'est-à-dire des mot-clefs qui servent à transformer une preuve en cours en une nouvelle (ou de finir la démonstration), souvent pour simplifier la preuve. Utiliser les tactiques peut permettre d'éviter d'écrire le terme de preuve, qui est beaucoup moins lisible.

Encore une fois pour illustrer la notion de tactique, dans `Minlog`, nous avons distingué les tactiques basiques qui appliquent une règle de la logique minimale, de tactiques plus élaborées, qui appliquent plusieurs étapes à la fois.

Plus généralement, les assistants de preuve offrent une grande sûreté, *dans la mesure où leur fiabilité repose sur celle de leur noyau logique*. Le code du noyau est gardé suffisamment petit pour qu'un nombre de personne le plus réduit possible puisse le maintenir, et pour limiter les erreurs possibles d'implémentation. Dans l'assistant de preuve `Coq` sur lequel je travaille, le noyau fait tout de même 20 000 lignes de code `OCaml`.

Même si la communauté d'un assistant de preuve y trouve régulièrement des bugs[4], le noyau est tout de suite mis à jour et les preuves écrites dans l'assistant peuvent être retestées avec la nouvelle version de manière à s'assurer qu'elles n'exploitaient pas la faille.

L'avantage est donc que l'on délimite très précisément la partie du code à laquelle on croit (la fondation pour nos croyances, en quelque sorte), et que cette partie est scrupuleusement examinée et testée continuellement. Tout le reste du logiciel (notamment les tactiques) ne fait pas partie du noyau, mais doit toujours appeler celui-ci, qui validera ou non les étapes produites par les tactiques[5].

3. `https://github.com/louiseddp/MinITP/tree/minlog`

4. Par exemple, il était possible de prouver une contradiction dans `Coq` lorsque un inductif avait plus de 255 constructeurs.

5. Je voudrais tout de même remarquer qu'en pratique, on fait non seulement confiance au noyau mais aussi à la partie du code qui gère les entrées/sorties (et notamment l'affichage). On pourrait imaginer un code qui, en interne, fait les bonnes vérifications, mais affiche « Validé ! » à une preuve de la quadrature du cercle. Certes, cela serait un peu étrange, mais je le mentionne parce que même lorsqu'on tente d'identifier les fondations de nos croyances, on tombe toujours sur des éléments « impurs », liés à une interaction avec quelque chose d'extérieur, que ça soit les impressions sensibles pour les humains, ou les entrées/sorties pour les machines.

En terme de vérification mathématique, les assistants de preuve semblent offrir des garanties qu'aucun relecteur expert d'un domaine ne pourrait nous prodiguer : ils délimitent précisément où peuvent se trouver leurs failles (dans le noyau donc), ils sont beaucoup plus « surveillés » par leur communauté que ne peut l'être un chercheur ou une chercheuse. À la rigueur, on peut avoir plusieurs relecteurs, mais tout ce qui assure leur fiabilité est leur position académique, nous croyons ensuite les théorèmes dans les articles qu'ils ont accepté bien souvent sans autre forme d'examen. Tandis que la fiabilité du noyau des assistants de preuve repose sur des tests répétés, une maintenance continue de leur code, et parfois aussi, un examen conjoint des mêmes démonstrations par différents assistants.

Pour résumer, les assistants de preuve sont des logiciels :

— Qui implémentent un métalangage riche, permettant d'écrire des raisonnements mathématiques ou des preuves de programmes

— Dont la fiabilité repose sur un noyau logique restreint

— Qui automatisent la vérification de preuves

— Qui proposent souvent des *tactiques* et des plugins pour aider l'utilisateur·ice à construire ses preuves.

## 1.3 Automatiser les assistants de preuve

Tout débutant avec un assistant de preuve le découvre bien vite (et c'est ce que j'ai souligné aussi dès 1.1.1), un lemme simple, que l'on ne s'embarrasserait même pas à démontrer sur papier, peut demander du travail pour que le logiciel en valide la démonstration. Nous distinguerons deux types de difficultés, et nous avons choisi de ne traiter que le deuxième type dans cette thèse :

1. Parfois, il s'agit de difficultés inhérentes à l'implémentation.

   **Exemple 1** [Difficultés liées à l'implémentation] Un même objet mathématique peut être représenté par différentes structures de données : les entiers unaires et les entiers binaires représentent tous les deux l'ensemble des entiers naturels $\mathbb{N}$. L'utilisateur·ice d'un assistant de preuve devra se préoccuper du fait que ces deux représentations ne sont pas le même objet du point de vue de l'assistant, et devra donc démontrer des lemmes d'équivalence et redémontrer des lemmes similaires pour chacun de ces deux types de données.

2. Parfois, ce sont des lemmes faciles, mais qui demandent à gérer beaucoup de sous-cas, qui n'auraient pas été traités sur papier.

   **Exemple 2** [Nombreux sous-cas]
   Supposons que nous voulions montrer que si la concaténation de deux listes donne un singleton (une liste avec un seul élément), alors soit la première liste est vide et la seconde est un singleton, soit c'est l'inverse.
   Ce théorème `Coq` s'écrit :

   ```
   Theorem app_eq_unit (x y:list A) (a:A) :
   x ++ y = [a] -> x = [] /\ y = [a] \/ x = [a] /\ y = [].
   ```

   Sur papier, on dirait probablement que ce lemme est trivial. Or, si l'on tente d'en écrire la preuve formellement en `Coq`, l'utilisateur·ice devra s'embarrasser à gérer cinq sous-cas :

(a) Les deux listes sont vides : il ou elle en déduit une contradiction puisque la concaténation de deux listes vide ne peut pas donner une liste non vide.

(b) Une des deux listes contient plus qu'un singleton : la concaténation de deux listes ne pourra pas donner un singleton, on a une nouvelle contradiction.

(c) La première liste est vide, la seconde est un singleton : on a bien la formule à gauche du $\vee$ qui est vraie.

(d) La première liste est un singleton, la seconde est vide : on a bien la formule à droite du $\vee$ qui est vraie.

(e) Les deux listes sont des singletons : il ou elle en déduit une contradiction puisque la concaténation de deux singletons ne donne pas un singleton.

On voit que sur une structure de donnée plus complexe (par exemple, des arbres $n$-aires), le nombre de cas à traiter pour des lemmes similaires sera encore plus grand, et que l'utilisateur·ice perdra beaucoup de temps à écrire et démontrer ces lemmes « bureaucratiques ».

Trouver une façon de résoudre le cas n°1, c'est-à-dire automatiser au maximum la preuve de lemmes similaires pour des types de données isomorphes est une tâche difficile que je n'ai pas du tout considérée dans cette thèse. Il existe d'autres problèmes empêchant l'utilisation plus massive d'un assistant de preuve et qui sont liés au fossé entre implémentation et mathématiques sur papier, mais j'ai choisi de présenter le plus simple d'entre eux pour cette introduction. L'important ici est simplement de remarquer que chacun de ces problèmes est complexe et doit être traité de façon séparée.

Or, ce travail de thèse offre une petite avancée dans la facilitation de l'utilisation d'un assistant de preuve, non pas en traitant un problème spécifique lie au fossé implémentation/mathématiques sur papier, mais plutôt en proposant plus d'automatisation générique.

Ainsi, trouver une manière *ad hoc* d'automatiser la situation n°2, ou au moins ses sous-cas, n'est pas forcément une tâche difficile. En effet, il y a bien souvent une tactique ou une suite de tactiques *ad hoc* qui fonctionne et qui permet de réduire la taille de son code. Mais, à force d'écrire ses tactiques d'automatisation *ad hoc*, ou d'allonger son code en traitant des sous-cas faciles, l'utilisateur·ice lassé·e risque de finir par déplorer le temps passé au développement de ses preuves, voire de considérer les méthodes formelles comme trop coûteuses pour ses objectifs.

Implémenter une (ou plusieurs) tactiques génériques, permettant d'automatiser autant de situations telles que la n°2 que possible, est l'une des tâches de cette thèse, au sein de l'assistant de preuve `Coq` (l'assistant de preuve sur lequel j'ai travaillé).

Deux raisons font qu'une plus grande automatisation des assistants de preuve est cruciale :

1. Ils restent des logiciels principalement utilisés dans le domaine académique car ils ne sont pas assez accessibles. Or ils pourraient être utilisés par des étudiant·es par exemple, pour mieux leur faire comprendre la notion de démonstration, ou par des développeur·euses qui souhaitent produire un code sûr.
2. Le coût de développement des preuves est parfois trop important, à cause du temps passé à démontrer des lemmes très simples mais nécessaires.

## 1.4 Un avant-goût des contributions

Tous ces constats étant faits, il se pose la question de savoir comment automatiser ces logiciels. La richesse de leur logique fait que trouver un algorithme générique semble difficile.

De fait, on trouve des tactiques qui, étant donné un lemme à prouver d'une forme bien déterminée, sont capables de conclure sans le concours de l'utilisateur·ice. Le plus souvent, la tactique permet d'automatiser des preuves d'une logique donnée (premier ordre, propositionnelle), ou traitant d'objets donnés (entiers, *bitvectors*...). Mais, il faut un travail préalable pour transformer le but initial de manière à ce qu'il soit accepté par la tactique. Autrement dit, il faut *prétraiter* les énoncés.

**Exemple 3** Un besoin de prétraitement

Supposons que nous souhaitions prouver, grâce à une tactique recourant à un prouveur automatique du premier ordre, qu'appliquer la composée de deux fonctions sur une liste `l` revient à appliquer d'abord la première fonction sur toute la liste pour obtenir une liste intermédiaire, puis appliquer la seconde fonction sur toute cette dernière.

La fonction `map` permet d'appliquer une même fonction sur toute une liste d'éléments. Pour représenter le fait que les éléments peuvent être d'un type de donnée quelconque, on utilise des variables de type `Type`. Ainsi, en `Coq`, le lemme s'écrit :

```
Lemma map_compound : forall (A B C : Type) (f : A -> B) (g : B -> C) (
    l : list A),
map g (map f l) = map (fun x => g (f x)) l
```

Notons que `fun x => g (f x)` représente la composée de `g` et de `f`.

Même si ce lemme est très simple et semble se prouver directement après une induction sur la liste `l`, il n'est pas possible de recourir à notre tactique qui appelle un prouveur automatique du premier ordre directement. En effet, le polymorphisme (avoir des variables de type), l'ordre supérieur (pouvoir avoir une fonction qui prend en argument une autre fonction, ici `map` prend `f`, `g` et `fun x => g (f x)` en argument), et les fonctions anonymes (cette construction `fun x => g (f x)`), font partie de la logique de `Coq` mais pas de la logique du premier ordre qui est moins expressive. De manière plus cachée, la définition de `map` est définie par cas sur la liste `l`, selon qu'elle soit vide ou non. Cela s'appelle le *pattern matching* et cela ne fait pas non plus partie de la syntaxe de la logique du premier ordre. Bien sûr, je donnerai par la suite des définitions plus précises de toutes ces constructions, mais je ne voulais pas étendre plus cet exemple.

Pourtant, un prétraitement permet de retomber en logique du premier ordre. Il suffit d'introduire les variables de types `A`, `B` et `C`, de remplacer la fonction anonyme par une nouvelle fonction `h` qui serait définie comme la composée de `f` et de `g`, et de transformer `map` en 3 sous-fonctions `mapf`, `mapg`, et `maph`, définies comme `map f`, `map g`, et `map h`. Un gros travail serait aussi d'envoyer les définitions de ces fonctions dans une forme compréhensible pour un prouveur automatique (en particulier, sans polymorphisme et sans *pattern matching*). Une fois que tout ceci est fait, une induction et un appel au prouveur automatique permettent d'effectuer toute la preuve.

Il existe plusieurs techniques pour prétraiter des énoncés. Supposons que nous avons un énoncé $T$ dans une logique expressive $\mathcal{L}$, et que nous voulons l'encoder dans une logique moins expressive $\mathcal{L}'$.

— On peut décider d'avoir une machinerie spécifique qui encode d'un seul coup un maximum des énoncés de $\mathcal{L}$ vers ceux de $\mathcal{L}'$.

— On peut décider de combiner plusieurs petites transformations, chacune prouvée correcte ou générant sa preuve de correction dans le cas concret où elle est appelée, de manière à passer progressivement de $\mathcal{L}$ vers $\mathcal{L}'$.

C'est cette deuxième option de prétraitement qui est adoptée dans cette thèse. Elle présente l'avantage d'obtenir une preuve de correction de chaque petite étape, et de pouvoir s'adapter à différents backends, puisque les transformations peuvent être combinées de différentes manières.

La première tactique d'automatisation considérée dans cette thèse recourt à un prouveur externe grâce au plugin `SMTCoq`. Celui-ci appelle depuis `Coq` des prouveurs automatiques du premier ordre, très efficaces sur des théories spécifiques, nommés prouveurs SMT (*Satisfiability Modulo Theories*). Il récupère la preuve construite automatiquement par le prouveur et la transforme en objet compréhensible par le noyau de `Coq`. J'ai donc choisi la plupart des transformations que j'ai implémentées en fonction de `SMTCoq`. Le but a donc été d'écrire des transformations visant à encoder certains aspects de la logique de `Coq` en logique du premier ordre.

Je présenterai tout d'abord le cadre de cette thèse : les logiques dans lesquelles j'ai travaillé, l'assistant de preuve `Coq`, puis le fonctionnement de `SMTCoq`.

Par la suite, je détaillerai mes contributions :

1. La première est une méthodologie générique pour *prétraiter* des buts dans `Coq`. Le principe est d'appliquer au but une composition de petites transformations logiques très spécifiques, chacune prouvant sa correction vis-à-vis de celui-ci. Je montrerai en quoi cette manière de procéder s'oppose à la méthode monolithique de l'état de l'art, ainsi que ses avantages. Par ailleurs, je parlerai des différents outils de métaprogrammation utilisés au sein de mon développement, car l'avantage de développer chaque transformation séparément est de pouvoir utiliser le meilleur outil selon les circonstances, plutôt que de s'en tenir à un seul d'entre eux.
2. La deuxième est la bibliothèque des transformations que j'ai implémentées. Le code est disponible dans le plugin `Sniper` (dans une version spécifique à ce manuscrit) à l'adresse suivante : `https://github.com/louiseddp/sniper/tree/for-phd`. Nous vous recommandons d'installer `Sniper` en suivant les instructions du `README` de cette version et d'examiner les fichiers d'exemples qui sont eux aussi spécifiques à cette version.
3. La troisième contribution est un autre plugin, un *ordonnanceur* de transformations appelé `Orchestrator`. À partir d'une liste de tactiques et d'une description dans un langage se voulant *user-friendly* de ce qui peut les déclencher dans un but `Coq`, le programme *ordonnanceur* est capable de déclencher les tactiques au bon moment. Il en résulte un code plus lisible, où l'action des tactiques sur ses arguments est clairement distinguée de la façon dont les arguments sont sélectionnés, plus efficace et plus facilement extensible.
4. Une dernière contribution consiste à utiliser l'ordonnanceur pour combiner ces transformations, de manière à cibler `SMTCoq` comme backend. La tactique finale, complètement automatique, s'appelle `snipe`. Nous développerons cette dernière contribution dans le chapitre consacré à l'ordonnanceur. L'ordonnanceur est encore en développement, mais son code est disponible dans une autre branche spécifique à ce manuscrit : `https://github.com/louiseddp/sniper/tree/for-phd-orchestrator`

# Chapitre 2

# CONTEXTE

Dans ce chapitre, je présenterai la Théorie des Types et surtout sa variante nommée Calcul des Constructions Inductives, sur lequel se fonde l'assistant de preuve `Coq`. Ensuite, je parlerai du plugin d'automatisation pour `Coq`, `SMTCoq`, qui sert de backend principal à mon développement logiciel `Sniper`, et dont sont aussi issus les principes méthodologiques qui ont guidé mon travail.

## 2.1 L'assistant de preuve `Coq` et sa logique

Comme `Coq` est un assistant de preuve basé sur la Théorie des Types, il nous faut avant toute chose introduire celle-ci. Le fil directeur de notre présentation sera l'objectif suivant : *disposer d'une théorie logique pertinente et suffisamment expressive pour y écrire des démonstrations et y représenter le comportement des programmes informatiques.*

Par ailleurs, la logique de `Coq` est dite *intuitionniste* ou *calculatoire* : une preuve d'une proposition $A$ est toujours vue comme un moyen de produire $A$. Des principes comme le raisonnement par l'absurde ou le tiers exclu ne peuvent donc y être admis. La logique intuitionniste est souvent associée aux preuves de programmes parce qu'un « moyen de produire $A$ », c'est un programme qui renvoie une preuve de $A$, comme nous allons le voir.

### 2.1.1 De la logique minimale aux systèmes de types purs

#### Les caractéristiques d'une logique formelle

Une logique a pour but de nous donner les outils qui permettent de décrire nos objets d'études et d'écrire des raisonnements corrects. Elle s'appuie d'abord sur un *langage*. Pour parler de propriétés d'un programme, nous avons besoin d'un langage *formel*, c'est-à-dire un langage qui proscrit les ambiguïtés. Les langages *naturels* ne sont pas adaptés au raisonnement car ils sont truffés d'homonymes, d'implicites et sont bien trop complexes. Cependant, dans les langages formels, comme dans tout langage, nous avons besoin de symboles, de règles pour les combiner, et de donner une signification à nos combinaisons de symboles.

**Syntaxe** Ainsi, une logique comporte un ensemble de symboles définis par une *grammaire*. Cette grammaire nous dit comment combiner les symboles entre eux. On appelle cela les *règles de syntaxe* et il existe une façon standard de les représenter. Par exemple, supposons que nous disposions d'une calculatrice rudimentaire, qui ne peut faire qu'additionner ou multiplier. Nous souhaitons décrire le langage formel qu'elle comprend. (Par souci de simplicité, nous oublierons

qu'il faut appuyer sur la touche $\hookrightarrow$ pour obtenir un résultat et nous ne considérerons que les expressions arithmétiques.)

Notre grammaire sera représentée de la façon suivante :

$$\begin{array}{rcll} \mathcal{E} & ::= & (\mathcal{E}) & \textit{Parenthèses} \\ & | & n & \textit{Entiers} \\ & | & \mathcal{E} + \mathcal{E} & \textit{Addition} \\ & | & \mathcal{E} * \mathcal{E} & \textit{Multiplication} \end{array}$$

Cela signifie qu'une expression, notée $\mathcal{E}$, que peut comprendre cette calculatrice est soit un entier, soit une somme de deux sous-expressions (c'est pour cela que le symbole $\mathcal{E}$ apparaît encore), soit une multiplication. Une expression bien formée dans ce petit langage serait donc `1+(2*3)` mais pas `(1` ou `1++42` par exemple. Dans ces deux derniers cas, la calculatrice afficherait sans doute quelque chose comme : `Err`.

Maintenant que nous savons comment construire des expressions dans un langage formel, considérons une nouvelle grammaire un peu plus complexe. Il s'agit de la grammaire de la *logique minimale*, implémentée dans `Minlog`. Ici, les expressions représentent des *formules*, c'est-à-dire des énoncés que l'on peut tenter de prouver ou d'infirmer.

On se donne pour base un ensemble de *variables propositionnelles* $\mathcal{V}(\mathcal{P})$ qui représentent des énoncés atomiques, et notre grammaire permet de composer ces variables entre elles pour obtenir des énoncés plus complexes. Autrement dit, on s'abstrait du contenu du raisonnement pour n'en garder que la structure. Mais en pratique, ces variables propositionnelles peuvent être remplacées par des énoncés concrets.

Pour construire des formules, on peut soit utiliser des variables propositionnelles, soit former des implications (éventuellement parenthésées). Notons que l'implication est par défaut associative à droite, donc $A \to B \to C$ se lit $A \to (B \to C)$. La grammaire de la logique minimale est :

$$\begin{array}{rcll} P & ::= & P \to P & \\ & | & X & X \in \mathcal{V}(\mathcal{P}) \end{array}$$

Nous étendrons petit à petit cette logique et donc également sa grammaire.

**Jugements et règles de dérivation** Pour la logique minimale, il nous faut donner les *règles de dérivation* qui nous permettent de prouver des formules. Celles-ci nous permettent de donner un sens aux formules et aux symboles de la grammaire, et en particulier, en logique minimale, à l'implication. Il faut remarquer qu'avec les expressions arithmétiques, nous n'assertons rien, il n'y aurait donc pas de sens à parler de règles de dérivation.

Pour prouver des formules, il nous faut bien souvent des *hypothèses*, qui sont les conditions sous lesquelles nous pouvons effectuer cette preuve. Notre ensemble d'hypothèses à disposition s'appelle un *contexte*.

Premièrement, nous étendons donc notre syntaxe avec une nouvelle expression, les contextes. Ici, les contextes mentionnent les formules, d'où l'occurrence du $P$ pour la deuxième règle syntaxique sur les contextes.

$$\begin{array}{rcll} P & ::= & P \to P & \\ & | & X & X \in \mathcal{V}(\mathcal{P}) \\ \Gamma & ::= & \bullet & \textit{Contexte vide} \\ & | & \Gamma, P & \textit{Ajout d'une formule au contexte} \end{array}$$

Une fois que nous pouvons représenter un ensemble quelconque d'hypothèses, nous avons besoin d'une façon de symboliser qu'étant donné les hypothèses de $\Gamma$, nous affirmons la conclusion $P$. C'est ainsi que nous introduisons le symbole $\vdash$ et la notion de *jugement*, sous la forme d'un *séquent*. Et nous rajoutons les jugements à la grammaire :

$\mathcal{J} \quad ::= \quad \Gamma \vdash P$

Ainsi, $\Gamma \vdash P$ signifie : en supposant les formules de $\Gamma$, on a aussi $P$. Mais il est possible d'avoir des jugements *valides* ou *invalides*, puisque pour le moment, nous ne leur avons pas encore donné de signification, mais seulement leurs règles de syntaxe. Les jugements valides sont ceux qui peuvent être obtenus par l'application d'une ou plusieurs des règles de dérivation données ci-dessous. Les jugements invalides sont ceux qui ne peuvent pas être obtenus ainsi. Les règles de dérivation sont représentées de la façon suivante :

$$\frac{Hyp_1 \quad Hyp_2 \quad ... \quad Hyp_n}{Concl}$$

Dans un raisonnement, elles se suivent et peuvent former un objet mathématique appelé *arbre de dérivation*, comme ici :

$$\dfrac{Hyp_3 \quad \dfrac{Hyp_1}{Concl_1 = Hyp_2}}{Concl_2}$$

Maintenant, présentons les trois règles de la logique minimale.

La première règle est la règle axiome, selon laquelle si l'on a $P$ dans nos hypothèses, on peut aussi en déduire $P$. Elle s'écrit formellement :

$$\frac{P \in \Gamma}{\Gamma \vdash P}\ ax$$

La deuxième concerne l'introduction de l'implication ; on l'appelle aussi la règle d'abstraction. Elle affirme que si, parmi nos hypothèses, nous avons la proposition $P$ et que nous pouvons en déduire $Q$, alors nous pouvons retirer $P$ de nos hypothèses et déduire $P$ implique $Q$. Autrement dit, à partir du séquent $\Gamma, P \vdash Q$, on peut prouver le séquent $\Gamma \vdash P \to Q$.

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q} \to intro$$

La troisième et dernière règle, le *modus ponens* (ou l'élimination de l'implication), est connue et formalisée depuis Théophraste (un disciple d'Aristote). Elle nous permet d'utiliser l'information représentée par une implication. Si l'on peut prouver $P \to Q$ et $P$ à partir de $\Gamma$, alors, on peut aussi en déduire $\Gamma \vdash Q$.

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

Ces règles de déduction servent donc à donner une signification aux symboles de la logique, et également à utiliser ceux-ci pour construire des raisonnements.

**Sémantique dénotationnelle** Une manière de donner du sens aux expressions d'un langage formel est de donner leur sémantique dénotationnelle, autrement dit, de leur faire correspondre un objet mathématique qui serait leur *dénotation*. Pour la grammaire des expressions arithmétique, c'est assez facile. On peut faire correspondre à chaque symbole de la forme $n$ du langage un entier naturel de $\mathbb{N}$, et faire correspondre le $+$ et le $*$ du langage à l'addition et à la multiplication dans $\mathbb{N}$. L'étude de la sémantique est l'objet de la *théorie des modèles* dont ne nous parlerons pas ici. En effet, les structures mathématiques associées aux objets de la théorie des types sont bien souvent complexes et dépassent le cadre de cette thèse.

Maintenant que nous avons défini ce qu'est une *logique*, il faut expliquer comment celle-ci permet l'étude des langages de programmation. Plus précisément, en présentant les lambda-calculs pur et typé, nous allons introduire des langages de programmation théoriques très simples, et nous verrons leur lien avec la logique (notamment, bien sûr, la logique minimale). Puis nous étendrons ces langages simples jusqu'à atteindre l'expressivité logique de `Coq`.

### Le lambda-calcul pur

Pour étudier un programme, il nous faut une mathématisation adéquate de la notion de *calcul*. En effet, à partir de son entrée, un programme effectue des calculs puis produit une sortie. Formellement, cette notion de calcul est aussi appelée *réduction*.

Le lambda-calcul pur, en tant que langage de programmation le plus simple, présente déjà une notion de réduction. Nous allons voir tout cela, mais donnons d'abord sa grammaire. Les expressions du lambda-calcul sont appelées des *termes*. Syntaxiquement, nous disposons d'un ensemble de variables $\mathcal{V}(\mathcal{X})$ souvent notées $x, y, z...$, un moyen de construire des *fonctions* et de les *appliquer*. Et c'est tout !

$$\begin{array}{rcll} t & ::= & x & x \in \mathcal{V}(\mathcal{X}) \\ & | & \lambda x.t & \textit{Abstraction} \\ & | & t\ t & \textit{Application} \end{array}$$

La construction $\lambda x.t$ est appelée l'*abstraction*. C'est une notation alternative, commune en lambda-calcul et en théorie des types, pour remplacer la fonction mathématique $f : x \mapsto t(x)$. La construction $t\,u$ serait écrite $t(u)$ par les mathématiciens, car elle représente l'application du terme $t$ au terme $u$. Cette manière d'écrire souligne le fait qu'*il n'y a pas de distinction entre les fonctions et les autres termes*. En lambda-calcul, on peut syntaxiquement appliquer n'importe quel terme à n'importe quel autre. Sémantiquement, cela conduira à des bizarreries dont nous allons parler.

Maintenant, nous pouvons nous intéresser à la façon dont ce langage modélise le calcul. Comme il dispose uniquement de fonctions et d'applications, la seule chose que nous pouvons faire, c'est étudier le comportement des fonctions appliquées à leurs arguments. Ainsi, le lambda-calcul pur dispose d'une seule règle de réduction, que l'on appelle la *$\beta$-réduction*. Pour comprendre cette règle, nous avons besoin de la notion cruciale de *substitution*.

Le lambda-calcul comporte des variables que l'on peut trouver liées ou libres. Dans l'expression $\lambda x.x$, la variable $x$ est liée par le lambda, c'est-à-dire qu'on pourrait très bien remplacer cette expression par $\lambda y.y$ pour obtenir une expression sémantiquement équivalente. La variable liée est indissociable du lieur (ici, le lambda) qui l'abstrait. Les variables qui ne sont rattachées à aucun lambda sont dites libres, comme dans l'expression $x\ y$ où $x$ et $y$ sont toutes les deux libres.

La substitution est une opération dont le but est de remplacer, dans un lambda-terme, une variable libre par un autre terme. Par exemple $t[y := u]$ signifie que l'on souhaite remplacer, dans $t$, la variable $y$ par le terme $u$, et plus précisément, toutes les *occurrences libres* de la variable $y$. Elle est définie par induction selon la forme du terme $t$ :

- Si $t$ est la variable $x$, alors $t[y := u] = u$ si $x = y$, et $t[y := u] = t$ sinon. C'est logique : on veut remplacer les occurrences de $y$ par $u$, pas les occurrences d'une autre variable.
- Si $t$ est de la forme $v\ w$, on substitue dans les deux sous-termes $v$ et $w$. Autrement dit, $(v\ w)[y := u] = v[y := u]\ w[y := u]$.
- Si $t$ est de la forme $\lambda x.v$, alors $t[y := u] = t$ si $x = y$ et $t[y := u] = \lambda x.v[y := u]$ sinon. En effet, dans le premier cas, $y$ n'est pas libre donc on ne veut pas substituer cette variable par $u$, sous peine de changer la signification du terme initial. Dans le second cas, il faudra prendre garde à ce que $u$ ne mentionne aucune occurrence libre de $x$, sinon celles-ci seront

*capturées* par le lambda. Si $u$ a des occurrences libres de $x$, il suffit de renommer le lieur de $t$ par une nouvelle variable $z$ qui n'apparaît nulle part. Par exemple, $(\lambda x.y)[y := x\,x]$ ne donne pas $\lambda x.x\,x$ mais $\lambda z.x\,x$.

Nous pouvons enfin définir la $\beta$-réduction :

$$\boxed{(\lambda x.t)\,u \to^{\beta} t[x := u]}$$

Pour expliquer cette règle, revenons à l'intuition sous-jacente : nous disposons d'une fonction qui abstrait sur un argument $x$, et nous appliquons cette fonction à un argument concret $u$. Il nous faut donc substituer tous les $x$ par l'argument concret $u$ dans $t$. C'est bien ce qu'il se passe en programmation quand des fonctions sont appelées : les variables sont substituées par les arguments concrets donnés dans l'appel.

Le lambda-calcul est donc intéressant pour étudier les calculs des programmes composés de fonctions, mais il permet d'écrire des calculs qui ne terminent pas. L'exemple paradigmatique du calcul qui diverge est celui-ci : considérons le terme $\delta = \lambda x.x\,x$ et appliquons-le à lui-même (on note ce nouveau terme $\Delta$). On obtient :

$\Delta = \delta\,\delta = (\lambda x.x\,x)\,(\lambda x.x\,x)$

$\Delta \to^{\beta} \Delta$

Ce terme se réduit en lui-même, il peut donc de nouveau se réduire en lui-même, etc. Autrement dit, le calcul ne termine jamais. Ces termes servent à représenter les programmes qui divergent.

Mais nous pouvons nous intéresser à un fragment du lambda-calcul où toutes les séquences de réduction terminent. Autrement dit, nous voulons que tous les termes de ce fragment puissent se réduire vers une forme sur laquelle il n'est plus possible d'appliquer la règle de $\beta$-réduction. Par exemple, $x$ est un tel terme, on ne peut pas réduire une variable. Ces formes sont dites *normales*, et, du point de vue de la programmation, elles correspondent au moment où le programme a fini de tourner et aboutit à un résultat.

Pour ce faire, nous allons introduire le lambda-calcul simplement typé.

### Le lambda-calcul simplement typé

Nous venons d'expliquer ce qu'est le lambda-calcul (pur), il nous reste à introduire la notion de *type*, qui s'avérera fondamentale tout au long de cette thèse.

En programmation, les fonctions n'attendent pas n'importe quel argument : celui-ci doit avoir un type précis, qui correspond à la nature de la donnée qui est attendue. Et de même, les fonctions renvoient un élément d'un type donné. Par exemple, en `OCaml`, l'utilisateur·ice dispose de types de base comme les entiers, les booléens ou les flottants, mais il peut aussi définir ses propres types ou utiliser des types fonctionnels (car les fonctions ont également un type).

Les *théories des types* utilisent cette notion pour caractériser tous les objets sur lesquels elles permettent de raisonner. Chaque objet du langage se voit octroyer un type, et, selon le type en question, nous pourrons inférer que l'objet a telle ou telle propriété. Par exemple, il sera possible de démontrer que tout élément de type booléen est égal à `true` ou `false`, mais cela ne sera pas vrai pour un élément du type des entiers naturels.

Ces théories des types s'opposent aux mathématiques usuelles (celles que vous verrez enseignées dans la majorité des cours) qui utilisent la théorie des ensembles. Dans celle-ci, on raisonne plutôt sur des objets qui appartiennent à des ensembles donnés. La différence majeure est que la théorie des ensembles s'appuie sur des *axiomes* tandis que les théories des types s'appuient sur des règles de raisonnement appelées *jugements de typage*. Par exemple, $\vdash t : A$ est le jugement affirmant que $t$ a le type $A$.

Ceci étant dit, le lambda-calcul simplement typé utilise les termes du lambda-calcul pur, mais il rajoute des types aux variables et aux arguments de fonctions, ce qui permet de discriminer

les termes entre ceux qui seront bien ou mal typés. Par exemple, $(\lambda(x : A).x)\ t$ n'est bien typé que si $t$ a le type $A$.

Donnons la grammaire de ce calcul, en remarquant qu'on ajoute à la grammaire des termes et des types un contexte de typage qui octroie aux variables leur type :

| | | | |
|---|---|---|---|
| $\Gamma$ | ::= | $\bullet$ | *Contexte vide* |
| | \| | $\Gamma,\ x : A$ | *Ajout d'une variable au contexte* |
| $A$ | ::= | $\iota$ | *Type de base* |
| | \| | $A \to A$ | *Type des fonctions* |
| $t$ | ::= | $x$ | $x \in \mathcal{V}(\mathcal{X})$ |
| | \| | $\lambda(x : A).t$ | *Abstraction* |
| | \| | $t\ t$ | *Application* |

En pratique, le type de base symbolisé par $\iota$ pourra être remplacé par des types concrets (par exemple les entiers naturels, les booléens...), auxquels il faudra alors rajouter les termes correspondants (les habitants de ces types), ainsi que des expressions qui les utilisent. Mais nous verrons une autre façon d'introduire ces types de base lorsque nous nous intéresserons aux inductifs.

Les termes bien typés de notre langage sont ceux qui peuvent être obtenus après une dérivation de typage. Une dérivation de typage est un arbre de preuve qui utilise les règles d'inférences suivantes :

— La règle variable, qui assure qu'une variable est bien typée d'un type donné quand on lui a assigné ce type dans le contexte ;

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

— La règle abstraction, qui permet de typer une fonction étant donné le type de son argument et son type de retour ;

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : A \to B}\ abs$$

— La règle d'application, qui permet de typer une application étant donné le type de la fonction et celui de son argument.

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ app$$

**Exemple 4** Exemple d'une dérivation de typage

Supposons que nous souhaitions typer le terme $(\lambda\ (x :\ \iota \to \iota).\ x)\ (\lambda\ (x :\ \iota).\ x)$.

On aura la dérivation suivante :

$$\frac{\dfrac{\dfrac{\dfrac{x :\ \iota \to \iota \in \{x :\ \iota \to \iota\}}{x :\ \iota \to \iota \vdash x :\ \iota \to \iota}\ var}{\vdash \lambda\ (x :\ \iota \to \iota).\ x :\ (\iota \to \iota) \to (\iota \to \iota)}\ lam \qquad \dfrac{\dfrac{x :\ \iota \in \{x :\ \iota\}}{x :\ \iota \vdash x :\ \iota}\ var}{\vdash \lambda\ (x :\ \iota).\ x : \iota \to \iota}\ lam}{\vdash (\lambda\ (x :\ \iota \to \iota).\ x)\ (\lambda\ (x :\ \iota).\ x) :\ \iota \to \iota}\ app$$

Le lambda-calcul simplement typé a la propriété cruciale que tout terme bien typé $t$ normalise fortement, autrement dit, il n'existe pas de séquence de réduction infinie $t \to^\beta t_1 \to^\beta t_2 \to^\beta ...$ Le lambda-calcul étant par ailleurs *confluent*, cela implique que pour tout terme $t$ bien typé, il existe un unique $u$, appelé forme normale de $t$, tel que $t \to^\beta * u$ et $u$ ne se réduit pas (ici, $\to^\beta *$ est la clôture transitive de $\to^\beta$).

Par exemple, on a typé le terme $(\lambda\ (x :\ \iota \to \iota).\ x)\ (\lambda\ (x :\ \iota).\ x)$, mais, par une étape de $\beta$-réduction, il se réduit en sa forme normale $(\lambda\ (x :\ \iota).\ x)$.

L'idée sous-jacente est qu'un terme bien typé ne conduira jamais, comme le terme $\Delta\,\Delta$, à une réduction qui ne termine pas. Il n'est d'ailleurs pas possible de donner un type à $\Delta\,\Delta$. Les termes bien typés ont un bon comportement vis-à-vis du calcul, leur évaluation aboutit toujours à une forme normale.

Ce langage n'est pas encore suffisamment expressif pour servir de langage de programmation, mais il permet de caractériser certains programmes qui terminent toujours. Notons déjà que `Coq` s'appuie sur le même principe : en `Coq`, tous les programmes terminent et renvoient une valeur. Par ailleurs, les notions de preuve et de programme vont se rejoindre, comme nous allons le voir tout de suite avec la correspondance de Curry-Howard.

**La correspondance de Curry-Howard** Si nous reprenons les règles de la logique minimale, et que nous les mettons en correspondance avec les règles de typage du lambda-calcul simplement typé, nous remarquons qu'elles sont très similaires. Les types et les preuves sont obtenus par les mêmes règles, seulement, dans le cas du lambda-calcul, nous avons une information supplémentaire que sont les termes.

C'est ce constat qui fait la force des assistants de preuve fondés sur la théorie des types : toute dérivation de typage de $A$ peut être vue comme une preuve de $A$. Le terme correspondant est appelé *terme de preuve*, il contient en lui toute l'information qui a permis de dériver $A$. Mais ce terme est aussi un programme : si c'est une fonction de type $A \to B$, alors, appliqué à un argument de type $A$ (une preuve de $A$), il nous fournira un terme de type $B$ (une preuve de $B$).

Cette correspondance fut énoncée formellement en 1980 [26]. Elle donne un nouvel éclairage sur l'aspect intuitionniste (ou calculatoire) des logiques que nous examinons : une preuve de $A$ est un moyen de produire $A$, un calcul qui aboutit à un élément $a : A$, donc précisément un programme. La correspondance de Curry-Howard s'étend à des constructions plus complexes que nous allons examiner, car pour le moment, le système de preuve à notre disposition est réduit.

### Les types dépendants

Dans les mathématiques classiques, il arrive de quantifier sur des objets indexés par un entier naturel $n$. Par exemple, si l'on veut raisonner sur une matrice carrée quelconque de taille $n$, on écrira : $\forall A \in \mathscr{M}_n$, ...

En théorie des types, les matrices carrées de taille $n$ formeront un type $\mathscr{M}_n$ qui dépend de l'entier naturel $n$, *qui dépend donc d'un terme*. Les familles de types qui dépendent d'un terme donné sont appelés *types dépendants* et ces derniers confèrent beaucoup d'expressivité logique aux théories des types qui les utilisent.

Dans notre grammaire du lambda-calcul simplement typé, on remplace l'expression des types de fonctions par une nouvelle construction. Les types de base sont désormais paramétrés par des termes au type fixé :

$$\begin{array}{rcll} A & ::= & \iota(\overrightarrow{t_i : A_i}) & \textit{Types de base} \\ & | & \Pi(x : A), B & \textit{Type des fonctions dépendantes} \end{array}$$

Cette présentation est effectuée pour introduire les notions progressivement, mais, en pratique, pour exploiter les types dépendants, il faudrait également avoir des constantes typées dans les termes.

Le type $A \to B$ devient alors une notation pour $\Pi(x : A), B$ lorsque $B$ ne dépend pas de $x$. Les règles de typage associées sont les suivantes :

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A), B}\ abs$$

$$\frac{\Gamma \vdash t : \Pi(x : A), B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\ app$$

Par souci de concision, nous ne faisons que le mentionner, mais nous avons besoin de la notion de contextes et de types bien formés. Par exemple, le type $B\ x$ n'est pas bien formé dans le contexte vide parce que la variable $x$ est libre, mais, si $A$ est un type bien formé, alors $B\ x$ est un type bien formé dans le contexte $x : A$.

### Le polymorphisme

Dans le précédent système, nous admettons des types qui dépendent des termes, mais nous n'admettons pas encore des termes dépendant de types. Pourtant, il s'agit d'une construction très fréquente dans les langages de programmation fonctionnels, appelée le *polymorphisme*. Considérons une fonction identité `OCaml` dans les entiers : `let id_int (x: int)= x`. Elle aura pour type `int -> int`. En admettant les entiers parmi nos types de base, on peut représenter la fonction identité dans notre langage : $\lambda(x : int).x : int \to int$. On peut aussi avoir une fonction similaire sur les booléens : $\lambda(x : bool).x$.

Mais, dans le langage `OCaml`, on peut écrire la fonction identité *sur un type quelconque* : `let id x = x`, qui a le type `'a -> 'a`. Le `'a` représente une variable de type implicitement quantifiée universellement. Elle peut être instanciée par n'importe quel type (mettons, `int`), pour retrouver une fonction identité monomorphe (comme notre exemple avec l'identité sur les entiers).

Nous pouvons étendre notre langage de manière un peu différente que précédemment pour admettre le polymorphisme : nous avons besoin de variables de type, et d'une quantification sur les types. Nous décrivons ici le système F.

| | | | |
|---|---|---|---|
| $A$ | ::= | $X$ | *Variables de type* |
| | \| | $\forall$X. A | *Quantification sur les types* |
| | \| | $A \to A$ | *Type des fonctions* |

Du côté des termes, il faut pouvoir abstraire un terme vis-à-vis d'une variable de type et appliquer un terme à un type. On étend donc ainsi la grammaire des termes :

| | | | |
|---|---|---|---|
| $t$ | ::= | $x$ | *Variables de termes* |
| | \| | $\lambda(x : A).t$ | *Abstraction* |
| | \| | $t\ t$ | *Application* |
| | \| | $\lambda X.\ t$ | *Abstraction sur un type* |
| | \| | $t\ A$ | *Application d'un terme à un type* |

On rajoute également deux règles de typage pour ces nouvelles constructions :

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \lambda X.\ t : \forall X.\ A}\ X \notin FV(\Gamma)$$

Ici, $X \notin FV(\Gamma)$ signifie que $X$ ne doit pas apparaître libre dans $\Gamma$.

$$\frac{\Gamma \vdash\ t : \forall X.\ A}{\Gamma \vdash t\ B : A[B := X]}\ app_{type}$$

Et nous pouvons enfin typer notre fonction identité générique grâce à la dérivation suivante :

$$\frac{\dfrac{x : X \vdash x : X}{\vdash \lambda(x : X).x : X \to X}}{\vdash \lambda X.\lambda(x : X).x : \forall X.(X \to X)}$$

### Les systèmes de types purs

Cette notion de dépendance entre les termes et les types peut être généralisée.

Plus précisément, nos systèmes peuvent admettre ou non les dépendances suivantes :

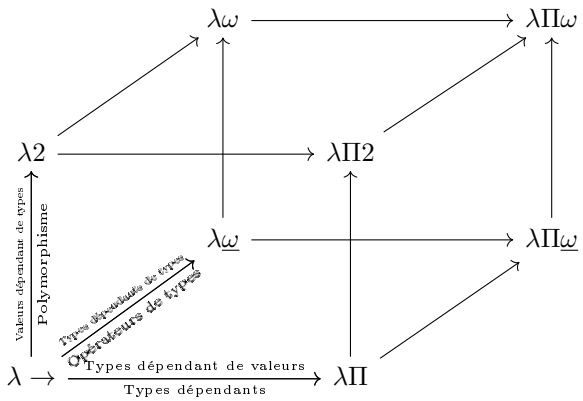
— Des termes dépendant de termes

— Des types dépendant de termes

— Des types dépendant de types

— Des termes dépendant de types

Le calcul le plus simple, le lambda-calcul simplement typé, n'admet que la première des dépendances. La logique de `Coq`, au contraire, admet *toutes* ces dépendances. On peut disposer de langages intermédiaires admettant certaines de ces constructions, mais pas toutes. Les différentes inclusions entre langages forment alors un cube appelé $\lambda$-cube, et présenté par Barendregt dans [3].




Nous ne détaillerons pas les différents systèmes, mais nous présenterons leur généralisation avec *les systèmes de types purs*.

Ceux-ci permettent de décrire tous les systèmes du cube (et d'autres encore), avec un ensemble de règles de typage paramétrées par des *sortes* et la façon dont elles dépendent les unes des autres.

Ainsi, un système de type pur (STP) est une théorie des types dans laquelle on se donne un ensemble $\mathscr{S}$ de sortes, ainsi qu'un sous-ensemble $Ax$ de $\mathscr{S}^2$ d'*axiomes* et un sous-ensemble $R$ de $\mathscr{S}^3$ de *règles*. Les axiomes servent à donner des types aux sortes de $\mathscr{S}$, et les règles servent à donner des types aux produits dépendants.

La grammaire des STP est la suivante :

$$
\begin{array}{lcll}
\Gamma & ::= & \bullet & \\
 & | & \Gamma,\ x : A & \\
A,\ t & ::= & x & \\
 & | & s & s \in \mathscr{S} \\
 & | & \Pi(x : A).B & \\
 & | & \lambda(x : A).t & \\
 & | & t\ t &
\end{array}
$$

Les règles de typage associées sont celles-ci :

$$\frac{(s_1, s_2) \in Ax}{\Gamma \vdash s_1 : s_2}$$

$$\frac{(s_1, s_2, s_3) \in R \quad \Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi(x : A).B : s_3}$$

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

$$\frac{\Gamma \vdash \Pi(x : A),\ B : s}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

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

$$\frac{A \equiv B \quad \Gamma \vdash t : A \quad \Gamma \vdash B : s \quad s \in \mathscr{S}}{\Gamma \vdash t : B}$$

La dernière règle introduit une notion que nous n'avons pas encore présentée : la conversion ($\equiv$). Elle assure que des types qui s'évaluent de la même façon (en appliquant la $\beta$-réduction) seront considérés comme identiques d'un point de vue des règles de typage. La conversion est définie comme la clôture symétrique, réflexive et transitive de la réduction. Et la réduction ($\longrightarrow$) est elle-même définie comme la clôture contextuelle de la $\beta$-réduction. Nous ne détaillerons pas toutes les règles de cette clôture contextuelle, mais, pour en donner une idée, elle permet de réduire seulement les têtes/les arguments des applications, ou de réduire sous des $\lambda$-abstractions, etc.

### 2.1.2 La logique de `Coq` : le Calcul des Constructions Inductives

La logique utilisée par l'assistant de preuve `Coq` correspond au système de type pur au sommet du lambda-cube, appelé Calcul des Constructions, auquel on a rajouté notamment la possibilité de définir des inductifs (des coinductifs aussi, mais nous n'en parlerons pas dans cette thèse) et une hiérarchie d'univers. Nous verrons d'abord la définition du calcul des constructions, puis ce qu'est un inductif ainsi que les règles de conversion associées.

#### Le Calcul des Constructions

Le Calcul des Constructions (CC) peut-être défini très simplement comme une instance de STP. On se donne deux sortes : $*$ et $\Box$, et un unique axiome dans l'ensemble $Ax$, ainsi, $Ax = \{(*, \Box)\}$.

La première sorte $*$ permet de typer les types, et donc d'avoir des jugements de la forme $\Gamma \vdash A : *$, elle est analogue à la constante *Type* mentionnée précédemment.

La seconde sorte $\Box$ permet de typer le type $*$ des types grâce à l'unique axiome de CC, on la note parfois $U$ pour *univers*.

L'ensemble $R$ des règles de CC vaut alors $\{(*, *, *); (\Box, *, *); (\Box, \Box, \Box); (*, \Box, \Box)\}$. Nous ne redonnons pas explicitement les règles de typage associées : elles se retrouvent ci-dessus en 2.1.1.

Ce calcul a été étendu pour comprendre une hiérarchie d'univers. L'idée est la suivante : puisqu'on a typé $*$, on voudrait aussi pouvoir typer $\Box$. Mais admettre $\Box : \Box$ comme règle de typage mène à un système incohérent [27]. Et introduire une troisième sorte $\triangle$ telle que $\Box : \triangle$ ne fait que repousser la question : quel est alors le type de $\triangle$ ? La solution naturelle est d'admettre une infinité d'univers $\Box_i$ avec $i \in \mathbb{N}$, et ajouter une infinité d'axiomes $(\Box_i, \Box_{i+1})$ pour $i \in \mathbb{N}$.

Autrement dit, chaque univers est indexé par un niveau $i$, et a pour type un univers dont le niveau est $i+1$. On rajoute aussi les règles $(\Box_i, \Box_{i+j}, \Box_{i+j})$ et on note $* = \Box_0$.

Par ailleurs, on peut vouloir distinguer deux sortes de base au lieu d'avoir seulement $*$, de manière à bénéficier d'une sorte *imprédicative* (c'est ce qui est fait dans `Coq`). Une sorte imprédicative $s$ est une sorte permettant de construire de nouveaux objets de $s$ en quantifiant universellement sur $s$. Concrètement, dans CC, on rajoute alors la sorte $\circ$, l'axiome $(\circ, \Box_1)$ et l'infinité de règles $(s, \circ, \circ)$ pour tout $s \in \mathscr{S}$. Ces règles permettent l'imprédicativité de $\circ$.

Une présentation de cette extension de CC a été faite par Christine Paulin [37]. Mais elle s'éloigne de la représentation de CC avec les notations pour les STP en usant des terminologies qui sont celles de l'assistant de preuve `Coq`. Ainsi, on aura que l'ensemble des sortes $\mathscr{S} = \{Prop\} \bigcup_{i=0}^{\infty} Type_i$. (Il faut remarquer qu'ici, $* = Type_0 = \Box_0$, aussi noté $Set$, et $Prop = \circ$).

Le système de `Coq` admet une notion d'univers encore plus puissante, avec la propriété de *cumulativité* et le *polymorphisme* d'univers. Comme cette thèse fait le lien entre des logiques moins puissantes et la logique de `Coq`, nous n'avons pas considéré ces aspects et nous vous renvoyons au manuel de `Coq`.

### Les inductifs

Rappelons-nous : l'objectif de `Coq` est de fournir assez d'expressivité pour pouvoir y exprimer les propriétés des programmes. Or, à ce stade, il nous manque encore un ingrédient important des programmes : les types de données.

Il faut remarquer que le Calcul des Constructions contient le système F, un système admettant du polymorphisme et donc une quantification du second ordre (une quantification sur des prédicats). Et le système F permet déjà d'encoder les types de données, donc pour commencer, on pourrait imaginer reprendre cet encodage pour `Coq`.

Prenons l'exemple des booléens pour voir comment ils pourraient être représentés. L'évaluation d'une structure conditionnelle (un `if ... else`) dans un programme qui renvoie une valeur de type $A$ correspond à donner une première valeur de type $A$ dans le cas où le booléen s'évalue vers $true$, et une seconde valeur de type $A$ si le booléen s'évalue vers $false$. Ainsi, l'encodage des booléens sera : $\forall A.A \to A \to A$. Le booléen $true$ sera représenté par la première projection :

$true = \lambda A.\lambda(x : A).\lambda(y : A).x$

Et $false$ sera représenté par la seconde projection :

$false = \lambda A.\lambda(x : A).\lambda(y : A).y$

Si $A$ est un type de données, et $b$ est de type bool, alors on aura bien que $b\ A\ x\ y \to^{\beta} * x$ si $b \equiv true$ et $b\ A\ x\ y \to^{\beta} * y$ si $b \equiv false$.

Cependant, avec ce choix de représentation, des propriétés qui semblent très naturelles ne sont pas prouvables dans CC [36]. Par exemple, $true \neq false$ n'est pas prouvable, on doit donc l'ajouter comme axiome.

De plus, on fait face à une certaine inefficacité au niveau du calcul : la manière dont les termes se réduisent est insatisfaisante. Considérons le type des données des entiers unaires : c'est un type qui peut être obtenu soit en prenant l'entier 0, soit en appliquant la fonction successeur $S$ à un entier déjà donné $n$. Sans rentrer dans les détails, on ne peut représenter de fonction qui permette de calculer le prédécesseur d'un entier $S\ n$, en un nombre constant d'étapes de réduction. Pourtant, dans un langage comme `OCaml`, c'est possible, et voici ce que cela donne :

```
let pred = function
  | 0 -> 0
  | S n -> n
```

La possibilité d'effectuer un *pattern matching* sur l'argument donné en entrée de la fonction permet de distinguer deux cas pour un entier : soit c'est zéro, soit c'est un successeur. Donc `pred (S n)` retourne `n` en une seule étape de calcul.

L'idée est donc d'introduire dans `Coq` la possibilité de définir des inductifs plus naturellement, avec leurs règles d'introduction qu'on nomme alors plutôt *constructeurs*, et leurs règles d'élimination qui correspondent au *pattern matching* (dépendant). Le langage CC devient alors CIC, pour Calcul des Constructions Inductives.

Les inductifs les plus basiques sont les *énumérations*, et l'exemple le plus typique d'énumération est le type `bool`. Nous verrons plus loin en 2.1.3 d'autres commandes possibles, mais dans la syntaxe concrète de `Coq`, les booléens peuvent être définis ainsi :

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Les règles de typage et de conversion de `Coq` vont alors être étendues pour tenir compte de ce nouvel inductif et de ses deux constructeurs, et ce de la façon suivante :

$$\frac{}{\vdash bool : Set}$$

$$\frac{}{\Gamma \vdash true : bool}$$

$$\frac{}{\Gamma \vdash false : bool}$$

La règle de l'élimination du type `bool` correspond à la structure conditionnelle que nous avons vue, autrement dit : étant donné un booléen $b$, on peut faire une analyse de cas sur $b$ et octroyer un type à celle-ci. En `OCaml`, on aurait deux expressions de même type `c` et `d` et on pourrait écrire une fonction dépendant d'un booléen ainsi : `let f = if b then c else d`. En `Coq`, `c` et `d` peuvent ne pas avoir le même type. Par ailleurs, le `if ... else` n'est que du sucre syntaxique pour l'expression plus générique `match ... with` (ou, en toute généralité, `match ... in ... as .... return`). Derrière le `match`, on écrit le terme à examiner, et derrière le `with`, on écrit les différentes valeurs que peut prendre ce terme (c'est-à-dire les différents *motifs* possibles), ainsi que la valeur que l'on souhaite renvoyer dans chaque cas.

Par exemple, on peut faire correspondre à un booléen `b` une valeur entière en écrivant :

```
match b with
| true => 1
| false => 0
end
```

au lieu d'écrire `if b then 1 else 0`.

Ainsi, la règle de typage de l'élimination d'un booléen sera :

$$\frac{\Gamma \vdash b : bool \quad \Gamma \vdash P : bool \to s \quad \Gamma \vdash c : P\ true \quad \Gamma \vdash d : P\ false \quad s \in \mathscr{S}}{\vdash \texttt{match}\ b\ \texttt{as}\ y\ \texttt{return}\ P y\ \texttt{with}\ true \Rightarrow c \mid false \Rightarrow d\ \texttt{end} : P\ b}$$

L'idée est que l'on définit un *prédicat dépendant* sur les booléens $P$. Pour dériver $P\ b$ avec $b$ un booléen quelconque, il suffit de trouver deux termes $c : P\ true$ et $d : P\ false$.

On a introduit une nouvelle construction syntaxique, le *pattern matching* (ou destructeur) `match ... as ... return ... with ...` pour la représenter. L'idée est que le premier argument derrière le `match` est le terme à examiner, le `as` permet d'abstraire le terme sur lequel on

matche en lui donnant un nom de variable (ici, $y$) et on écrit le prédicat de retour derrière le `return`, donc ici $P\,y$. Derrière le `with`, on doit donner un terme pour chacun des constructeurs de *bool*, donc ici, le terme $c$ pour la branche *true* et le terme $d$ pour la branche $false$.

Les règles de *conversion* doivent être étendues pour tenir compte du *pattern matching*. Un destructeur appliqué à un constructeur se réduit vers la branche correspondante, on appelle cela la $\iota$-réduction. Ainsi, on a :

— `match` $true$ `as` $y$ `return` $Py$ `with` $true \Rightarrow c \mid false \Rightarrow d$ `end` $\equiv c$

— `match` $false$ `as` $y$ `return` $Py$ `with` $true \Rightarrow c \mid false \Rightarrow d$ `end` $\equiv d$

Nous pouvons désormais parler d'un type inductif un peu plus compliqué, dans le sens où il permet d'utiliser la *récursion* : le type des entiers naturels. En `Coq`, on l'écrit ainsi :

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

La nouveauté est que le constructeur `S` prend déjà un `nat` en argument[1]. Cela va se retrouver dans les règles de typage, car le constructeur $S$ suppose qu'un entier soit déjà donné pour être appliqué.

$$\frac{}{\Gamma \vdash nat : Set}$$

$$\frac{}{\Gamma \vdash 0 : nat}$$

$$\frac{\Gamma \vdash n : nat}{\Gamma \vdash S\,n : nat}$$

$$\frac{\Gamma \vdash n : nat \quad \Gamma \vdash P : nat \to s \quad \Gamma \vdash a : P\,0 \quad \Gamma, m : nat \vdash b : P\,(S\,m) \quad s \in \mathscr{S}}{\vdash \texttt{match}\ n\ \texttt{as}\ y\ \texttt{return}\ Py\ \texttt{with}\ 0 \Rightarrow a \mid S\,m \Rightarrow b\ \texttt{end} : P\,n}$$

Pour écrire le *principe d'induction*, le *pattern matching* n'est pas assez puissant : il ne permet qu'une analyse de cas sur les entiers, en renvoyant un terme pour le cas où l'entier $n$ sur lequel on matche est 0 et un autre terme pour le cas où c'est le successeur d'un autre entier $m$. Mais dans le cas successeur, on voudrait bien pouvoir supposer que la propriété $P$ s'applique déjà à $m$. Pour pallier ce problème, deux solutions sont possibles.

— Le principe d'induction peut être défini comme un nouveau terme *natind* ajouté au contexte. On se donne alors $natind : \Pi(P : nat \to s).\ P\,0 \to (\Pi(n : nat).P\,n \to P\,(S\,n)) \to \Pi(n : nat).\ P\,n$. Ici, $s$ est une sorte qui peut être un $Type_i$ ou $Prop$. Le type de *natind* signifie que pour prouver une propriété $P$ sur tous les entiers, il suffit : 1) de la prouver pour 0 puis 2) de supposer qu'on dispose d'une preuve de $P\,n$ pour un entier arbitraire $n$ et de construire alors une preuve de $P(S\,n)$.

— Une autre construction appelée *point fixe*, avec ses règles de typage, est ajoutée à CIC. Combinée au *pattern matching*, elle permet de retrouver les principes d'induction. Nous en parlerons par la suite au paragraphe sur les points fixes (ci-dessous).

1. J'ai décidé de ne pas détailler ce point, mais il faut noter que pour éviter des problèmes de non-terminaison, les occurrences de l'inductif $I$ en cours de définition dans le type des constructeurs doivent être *strictement positives*. Cela interdit par exemple d'avoir un constructeur de type $(I \to I) \to I$. Pour une définition précise, la référence est le manuel de `Coq` : `https://coq.inria.fr/refman/language/core/inductive.html#strict-positivity`.

### Paramètres et annotations

La richesse de CIC permet de définir divers inductifs, notamment des inductifs polymorphes comme les listes, des prédicats comme celui de parité d'un nombre entier, ou des types de données qui ont déjà par typage des propriétés logiques, comme les listes de taille fixée $n$. Nous avons besoin de notions formelles pour différencier et caractériser ces types, notamment parce que certains peuvent être facilement définis dans des langages moins riches, tandis que la difficulté est plus grande pour d'autres.

Nous définissions d'abord *l'arité* d'un inductif par induction.

— Si $s \in \mathscr{S}$, $s$ est une arité.

— Si $a$ est une arité et $C : s$ pour une certaine sorte $s$, $\Pi(x : C).\, a$ est une arité.

Par la suite, on note $\vec{x}$ les variables $x_1, ..., x_m$ et $\vec{C}$ les types $C_1, ..., C_m$. Une arité est donc de la forme $\Pi(\vec{x} : \vec{C}).\, s$, c'est un terme dont la conclusion est une sorte.

Un inductif $I$ a toujours pour type une arité. Cependant, dans la définition de l'inductif, on va distinguer les *paramètres* des *annotations*[2]. Par exemple, soit un inductif $I$ de type $\Pi(\vec{x} : \vec{A})(\vec{y} : \vec{B}) \to s$, dont les paramètres sont les $x_i : A_i$ et les annotations les $y_i : B_i$. Les $A_i$ et les $B_i$ peuvent dépendre des $x_j$ et $y_j$ introduits précédemment.

En `Coq`, la définition de $I$ sera de la forme :

```
Inductive I (x1: A1) ... (xk: Ak) : forall (y1: B1) ... (yl: Bl), Type
:= ...
```

À la place du `Type`, nous aurions pu mettre une autre sorte (`Set` ou `Prop`). Supposons que $I$ ait $n$ constructeurs. Une fois les $\vec{x}$ fixés, chaque constructeur $c_j$ de $I$ est une fonction dont le codomaine est $I\ x_1\ ...\ x_k\ t_{1,j} ...\ t_{l,j}$. Les $\vec{x}$, soit les paramètres, ne dépendent donc pas du constructeur $c_j$, tandis que les $\vec{t_j}$, soit les annotations, peuvent dépendre du constructeur $c_j$.

Donnons deux exemples qui illustrent bien cette différence : les listes et le prédicat `even`.

```
Inductive list (A : Type) : Set :=
| nil : list A
| cons : A -> list A -> list A.


Inductive even : nat -> Prop :=
| even0 : even 0
| evenS : forall n, even n -> even (S (S n)).
```

Les deux constructeurs de `list`, `nil` et `cons`, doivent nécessairement avoir le même type de retour `list A`, *paramétré* par `A`. En revanche, le prédicat `even` est *annoté* par un entier $n$, qui peut prendre des valeurs différentes selon ses constructeurs. Les inductifs peuvent bien sûr comporter à la fois des paramètres et des annotations, comme dans le cas des listes indexées par leur taille.

```
Inductive listn (A : Type) : nat -> Set :=
| niln : listn A 0
| consn : A -> listn A n -> list A (S n).
```

2. On reprend ici la terminologie de Christine Paulin (voir [37]).

### Egalité propositionnelle

Un inductif très important est l'égalité. Nous avons déjà une notion d'égalité en CIC avec la *conversion*. On l'appelle aussi égalité calculatoire ou définitionnelle. L'idée est que les termes convertibles ne sont pas discriminables du point de vue des règles de typage (d'un point de vue syntaxique, les termes convertibles peuvent bien sûr être différents). Autrement dit, du point de vue du système logique, ce sont les mêmes termes. Une autre égalité, interne au système cette fois, est appelée *égalité propositionnelle* et c'est un inductif.

```
Inductive eq (A: Type) (x: A) : A -> Prop :=
| eq_refl : eq A x x.
```

Nous voyons que c'est un inductif avec un seul constructeur `eq_refl`, deux paramètres $A$ et $x$, et une annotation de type $A$. Il faut noter que cette égalité n'est *pas décidable* en toute généralité. Ceci entraîne que pour écrire en `Coq` des tests d'égalité, nous avons besoin de définir des égalités booléennes entre termes de certains types.

L'inductif `eq` vient avec son principe d'induction `eq_ind`, qui est de type :

$\Pi(A : Type)(x : A)(P : A \to Prop).\, P\, x \to \Pi(y : A).\, eq\, A\, x\, y \to P\, y.$

Ceci signifie que si deux termes sont égaux et qu'un prédicat vaut pour l'un, alors il vaut pour l'autre.

Grâce à ce principe d'induction, il nous est désormais possible de prouver que $true \neq false$ (rappelons que c'était une des raisons de l'introduction des inductifs dans CIC). La notation $x \neq y$ signifie simplement $x = y \to False$, avec $False$ défini comme un inductif à zéro constructeur :

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

Nous devons aussi définir `True`, qui est l'inductif au codomaine dans `Prop` avec un seul constructeur :

```
Inductive True : Prop :=
| I : True.
```

Ces inductifs ne doivent pas être confondus avec les booléens `true` et `false`.

Premièrement, définissons un prédicat par *pattern matching* sur son argument booléen $b$ :

```
Definition P (b : bool) :=
match b with
| true => True
| false => False
end.
```

Ensuite, étant donné une égalité de type $true = false$, il suffit de prouver $P\ true$ pour prouver $P\ false$. Or, $P\ true$ est trivial (car on doit prouver $True$ qui a un habitant trivial $I$). On définit donc le terme suivant :

```
Definition true_not_false (H : true = false) : False :=
match H with
| eq_refl => eq_ind true P I false H
end.
```

Ce terme a bien le type souhaité.

### Les points fixes

Comme nous l'avions mentionné au paragraphe sur les inductifs (ci-dessus), le *pattern matching* ne permet pas à lui seul d'écrire les principes d'induction des inductifs. Il nous faut ajouter

une dernière construction dans Coq appelée *point fixe*. Il s'agit d'un combinateur de point fixe noté $fix$ dont les règles de typage sont les suivantes (en notant $s$ une sorte de $\mathscr{S}$) :

$$\frac{\Gamma \vdash \Pi(\vec{x} : \vec{A}).\, T : s \quad \Gamma, f : \Pi(\vec{x} : \vec{A}).\, T \vdash t : \Pi(\vec{x} : \vec{A}).\, T \quad Guard_n(f, t)}{\Gamma \vdash fix(f : \Pi(\vec{x} : \vec{A}).\, T).\, t : \Pi(\vec{x} : \vec{A}).\, T}$$

Comme le terme $t$ peut mentionner $f$, nous avons besoin d'une *condition de garde* pour s'assurer que les appels à $f$ sont faits d'une façon qui ne mettra pas en défaut la normalisation forte des termes de Coq, ainsi que la cohérence du système (le fait de ne pas pouvoir prouver $False$).

Sans cette condition, il serait possible d'écrire des dérivations de typage correctes avec des termes de type $False$ On peut par exemple dériver le terme $fix(f : False).f$ qui a le type $False$. Mais comme l'appel récursif fait à $f$ n'y est pas strictement décroissant, ce terme ne respecte pas la condition de garde.

Ainsi, nous avons introduit la notation $Guard_n(f,\ t)$ pour signifier que l'éventuel appel à $f$ fait dans le terme $t$ sera fait sur l'argument $n$ de façon décroissante. Nous ne détaillerons pas comment cette condition de garde est vérifiée, mais, en pratique, cela signifie que l'appel récursif de $f$ est fait sur un sous-terme de l'argument $x_n : A_n$. De cette façon, on s'assure qu'il n'y aura pas d'appels infinis à $f$ : au cours du calcul, on tombera bien sur un cas de base où il n'y aura plus d'appel à $f$.

Désormais, on peut réécrire le principe d'induction de $nat$ en combinant *pattern matching* et point fixe :

$natind = \lambda\,(P : nat \to Prop)(p_0 :\ P\ 0)(p_S : \Pi(n :\ nat).P\ n \to P\ (S\ n)).$
$fix\ (F : \Pi(n : nat).P\ n).\ \lambda(n :\ nat).$ `match` $n$ `as` $n_0$ `return` $(P\ n_0)$ `with`
$|\ 0 \Rightarrow p_0$
$|\ S\ n_0 \Rightarrow p_S\ n_0\ (F\ n_0)$
`end`

Ici, on remarque que contrairement au principe de l'analyse de cas sur $nat$, la preuve $p_S$ peut mentionner la fonction $F$. Ce point fixe $F$ est néanmoins correct car, dans le corps de la fonction, $F$ est appelé sur le sous terme $n_0$ de $S\ n_0$.

Une dernière chose à préciser est que le combinateur $fix$ s'accompagne d'une règle de réduction limitée, afin d'empêcher que la règle ne s'applique à l'infini et ne brise donc la normalisation forte du système : on ne peut réduire un point fixe que lorsque l'argument sur lequel s'effectue la récursion est un constructeur. La règle de réduction qui semblerait naturelle est :

$fix\ f\ t \longrightarrow t[f := fix\ f\ t]$

Mais on pourrait appliquer cette réduction à nouveau puisque on a fait réapparaître la construction $fix$ en substituant $f$ dans $t$. La règle de réduction dans Coq est donc la suivante, étant donné l'argument $a_n$ sur lequel s'effectue la récursion, $c$ un constructeur du type $A_n$ et $b_1, ..., b_m$ ses arguments (paramètres ou annotations) :

$(fix\ f.\ t)\ a_1\ ...\ a_{n-1}\ (c\ b_1\ ...\ b_m) \longrightarrow t[f := fix\ f.\ t]\ a_1\ ...\ a_{n-1}\ (c\ b_1\ ...\ b_m)$

Le point fixe n'est pas nécessairement appliqué entièrement pour que la règle de réduction puisse être utilisée, mais il doit au moins être appliqué jusqu'à l'argument récursif $n$.

### 2.1.3 L'implémentation de Coq

L'assistant de preuve Coq ne se réduit pas à son langage logique. Celui-ci est implémenté dans le noyau de Coq, et permet de vérifier que les termes fournis par l'utilisateur·ice sont bien typés. Dans cette partie, il nous faut voir tout d'abord certains aspects de la représentation des termes dans le noyau de Coq (et d'une représentation alternative hors du noyau), et ensuite nous

verrons des caractéristiques de `Coq` qui ne sont pas présentes dans le noyau mais dont nous avons eu besoin au cours de cette thèse.

### La représentation des termes

`Coq` est implémenté dans le langage fonctionnel `OCaml`. Dans le noyau, les termes de `Coq` sont représentés par le type `constr`. A chaque constructeur de ce type correspond une expression de la syntaxe de `Coq`.

**Les variables** Tandis que sur papier, il est aisé de représenter les variables et d'en changer de nom au besoin, l'implémentation pose plus de difficultés car elle est guidée par des contraintes d'efficacité et des règles plus rigoureuses.

Dans le lambda-calcul (et dans les théories des types), les termes ne sont pas considérés en tant que tels mais sont considérés à *alpha-équivalence* près. L'alpha-équivalence permet de capturer formellement la notion de *renommage*. Ainsi, renommer des variables liées dans un terme $t$ conduit à un terme $t'$ dans la même classe d'alpha-équivalence, et on considérera que $t$ et $t'$ sont égaux.

Très souvent sur papier, le fait que les termes sont considérés modulo alpha-équivalence est mentionné, puis l'on ne s'en occupe plus. Au contraire, dans une implémentation, il faut toujours prêter attention à cette notion.

L'idée la plus intuitive pour la représentation des variables dans l'implémentation d'un assistant de preuve est d'utiliser des variables nommées, exactement comme sur papier : les variables sont alors des chaînes de caractère ou des entiers. Cependant, cette représentation est inefficace ; notamment, les fonctions de substitution doivent éviter les captures de variable ce qui rend le code plus sujet à comporter des bugs et moins performant.

Pour ces raisons, plusieurs représentations des variables ont été inventées :

— Les indices de De Bruijn

— Les niveaux de De Bruijn

— Les variables localement sans nom

— La syntaxe abstraite d'ordre supérieur

En `Coq`, le choix s'est porté sur les indices de De Bruijn. Nous allons donc parler de ceux-ci dans ce paragraphe (nous ne parlerons pas des niveaux de De Bruijn et des variables localement sans nom), puis nous viendrons à la syntaxe abstraite d'ordre supérieur lorsque nous parlerons de `Coq-Elpi` 3.4.5.

Le principe des indices de De Bruijn est de représenter les variables par des entiers qui indiqueront quand a été introduit leur lieur (produit dépendant, opérateur de point fixe, de *pattern matching* ou lambda) dans le terme. La variable $n$ indique par exemple qu'il faut traverser $n$ lieurs avant de tomber sur le lieur qui l'a introduite.

> **Exemple 5** [Indices de De Bruijn] Considérons le terme suivant (comportant une variable libre $z$) :
> $\lambda x.\ x\ (\lambda y.\ x\ y\ z)$
> Il se réécrira en utilisant les indices de De Bruijn comme $\lambda.\ 0\ (\lambda.\ 1\ 0\ 2)$. On remarque deux choses : premièrement, les variables libres sont représentées par des indices supérieurs au nombre de lieurs dans le terme. Deuxièmement, une même variable peut avoir des indices différents selon sa position dans le terme : c'est le cas de $x$ ici.

Bien que l'usage des indices de De Bruijn soit contre-intuitif pour un être humain et demande à faire de l'arithmétique sur les indices lors de l'écriture de lambda-termes, la substitution sera

aisée à implémenter. Elle va avec une opération de *lifting*, qui consiste à incrémenter les indices des autres variables lorsqu'une nouvelle variable est introduite.

**Exemple 6** [Substitution et lifting de lambda-termes] En `OCaml`, considérons les termes du lambda-calcul, représentés par des indices de De Bruijn :

```
type trm =
| Var of int (* Indices de De Bruijn *)
| Abs of trm
| App of trm*trm
```

L'opération de lifting (`k` fois, à partir de l'indice `l`) sera implémentée ainsi :

```
let rec lift k l t =
  match t with
  | Var n -> if l <= n then (Var (n+k)) else (Var n)
  | Abs t -> Abs (lift k (l+1) t)
  | App (t1, t2) -> App (lift k l t1) (lift k l t2)
```

Appliquer `lift k l` au terme `t` équivaut à introduire `k` nouvelles variables à la position `l`.

Et la substitution sera implémentée de cette façon :

```
let rec subst n u t =
  match t with
  | Var k -> if k = n then (lift k 0 u) else
    if n < k then Var (k-1) else Var k
  | Abs t -> Abs (subst (n+1) u t)
  | App (t1, t2) -> App (subst n u t1) (subst n u t2)
```

On substitue la variable `n` par le terme `u` dans le terme `t`.

Donc quand `t` est une variable, alors si c'est `n` on remplace cette variable par `u`. Mais il faut tenir compte du fait que `u` a des variables qui peuvent être capturées parce qu'on a introduit `k` variables de plus, c'est pourquoi on lifte ce terme `k` fois.

Si l'indice de De Bruijn `k` de cette variable est supérieur à `n`, il faut tenir compte du fait qu'une variable a disparu et donc décrémenter `k`. S'il est inférieur, alors comme `k` est plus haut dans la pile des nouvelles variables que `n`, la substitution de `n` n'affecte pas l'indice de `k`. Autrement dit, dans ce cas là il y a autant de variables introduites avant la variable `k` dans le terme initial que dans le terme après substitution.

Pour le cas où `t` est une abstraction, on cherche à substituer la variable `n+1` et non plus `n` une fois passé sous le lieur.

Le cas de l'application est trivial puisqu'il ne contient aucun lieur.

Cet exemple sur un calcul plus simple que CIC permet de comprendre le fonctionnement des indices de De Bruijn. Dans `Coq`, il y a beaucoup plus de lieurs différents mais tout marche sur le même principe. Notons néanmoins que les indices de De Bruijn commencent à 1 dans le noyau (ce qui n'est pas standard mais n'est qu'un choix de développement).

**Le type `Econstr`** Un autre aspect de l'implémentation de `Coq` que nous souhaiterions souligner est que le type `constr` n'est pas la seule représentation des termes en `OCaml`, même si c'est la seule dans le noyau. Notamment, `Coq` propose un type de donnée `Econstr` qui correspond aux termes « à trous ». L'idée est que `Coq` peut retrouver une partie de l'information tout seul pour reformer

ensuite des `constr`. Par exemple, ces termes peuvent être partiellement typés (par exemple, on ne précise pas toujours les paramètres des types polymorphes), ou ne pas mentionner l'univers dans lequel ils sont typés.

Les `Econstr` peuvent aussi contenir des métavariables. Ces métavariables sont appelées « variables existentielles » dans le manuel de référence de Coq. Elles représentent un terme qui n'a pas encore été explicité. Ces variables existentielles sont surtout utiles lorsque `Coq` est utilisé en mode interactif pour écrire des preuves. Initialement, quand on cherche à prouver un but `Coq`, sa preuve est sous forme d'une variable existentielle $?G$ dont on précise petit à petit la forme jusqu'à obtenir un terme sans variable existentielle. Par exemple, si le but est une conjonction, on peut indiquer à `Coq` que sa preuve est en réalité une paire. `Coq` va alors introduire deux nouvelles variables existentielles et remplacer $?G$ par $(?G_1, ?G_2)$. Ces métavariables sont aussi appelées *evar*. En interne, `Coq` gère un environnement d'*evar* qui permet de leur octroyer un type.

### Les commandes vernaculaires

Pour interagir avec l'utilisateur·ice, `Coq` propose un ensemble de commandes vernaculaires qui permettent de définir des termes (via les mot-clefs `Definition`, `Fixpoint` pour les points fixes ou `Inductive` pour les inductifs), de vérifier leur type (`Check`), d'afficher la définition d'un terme (`Print`) etc.

Nous mentionnons ces commandes parce qu'il est important de les distinguer des tactiques. Certaines commandes permettent d'ajouter des termes à l'environnement *global* de `Coq`, tandis que les tactiques n'agissent *jamais* sur cet environnement. Ces dernières s'exécutent dans l'environnement *local* à la preuve.

Certaines commandes vernaculaires permettent aussi de désactiver des vérifications faites par `Coq`. Elles peuvent alors potentiellement casser la normalisation et/ou la cohérence du système. Par exemple, `Unset Guard Checking` va désactiver toutes les vérifications faites de la condition de garde, permettant l'écriture de points fixes aux appels récursifs illimités. Une autre commande connue est `Axiom`. Le plus souvent, cette commande sert à renforcer la théorie logique de `Coq` en admettant, par exemple, le tiers exclu (ce qui rend la logique de `Coq` classique). Mais en pratique, on pourrait très bien écrire `Axiom broke_Coq_coherence : False`. Il faut donc particulièrement prêter attention à ces commandes non sûres et les utiliser à bon escient. Nous en avons fait usage dans `Sniper`, pour des fonctions sur lesquelles nous ne voulions pas avoir à nous soucier de leur preuve de terminaison (voir 3.4.4).

Nous proposons dans `Sniper` une commande pour la transformation qui gère les relations inductives : 4.3. Elle se distingue donc nettement des transformations qui sont implémentées en tant que tactiques car elle ne peut être utilisée au cours d'une preuve.

Une commande vernaculaire utile permet aussi d'ouvrir des *sections* (`Section toto` ouvre la section « toto »). Au sein de ces sections peuvent être déclarées des variables locales de section, disponibles au sein de toute la section et y compris des preuves qui y sont faites. Nous parlerons parfois de variables de section par la suite.

### Les tactiques

L'écriture d'un terme de preuve peut vite se révéler fastidieuse, et, pour les preuves complexes, le terme en lui-même devient illisible pour un humain. D'ailleurs, même dans les preuves simples, il est plus naturel de lire un raisonnement mathématique qu'un terme de preuve.

Afin de pallier ces deux problèmes, `Coq` propose un *mode interactif* qui permet de construire le lambda-terme en usant de mots-clefs de plus haut niveau que nous avions appelés *tactiques*. Historiquement, les tactiques ont été inventées dans le langage appelé *Logic for Computable Functions* [33] (La logique des fonctions calculables, LCF). L'idée est que le terme à prouver est

présenté sous la forme d'un *but*, c'est-à-dire un contexte (comprenant hypothèses et définitions locales) et une conclusion. Les tactiques prennent un but $G$ en entrée et produisent une liste de sous-buts $L$ en sortie. Quand $L$ devient vide, on peut appeler le noyau. Le plus souvent, les tactiques ont un effet de bord qui consiste à construire partiellement le terme de preuve.

En `Coq`, la commande `Definition` permet de donner directement le lambda-terme qui constitue la preuve. (Notons qu'ici, « preuve » est utilisé au sens large, c'est-à-dire qu'on utilise ce mot aussi bien pour les termes de type `Type` que ceux de type `Prop`. Parfois, nous aurons besoin de faire la différence).

Au contraire, la commande `Proof` symbolise l'ouverture du mode interactif. L'utilisateur·ice peut alors exécuter une par une les tactiques qu'il ou elle écrit, et voit l'évolution du but `Coq` au fur et à mesure que les tactiques s'appliquent.

Comparons une même preuve avec et sans tactique dans un exemple :

**Exemple 7** [Terme de preuve vs tactiques] Reprenons la preuve que $true \neq false$, mais cette fois, nous définissons le prédicat $P$ à l'aide de la syntaxe `let ... in` qui permet d'écrire des définitions locales :

```
Definition true_not_false_term :=
fun (H : true = false) =>
let P := fun (b : bool) =>
match b with
| true => True
| false => False
end in
match H with
| eq_refl => eq_ind true P I false H
end : (true = false) -> False.
```

Ce terme de preuve est bien un habitant du type que l'on souhaite prouver, mais il n'est pas lisible au premier coup d'œil (il faut connaître le type de `eq_ind` par exemple). Pourtant la propriété démontrée est très basique, cela laisse donc présager de la difficulté d'écrire des termes de preuves lorsqu'on souhaite démontrer d'importants théorèmes mathématiques. En revanche, le script de tactique qui génère le terme de preuve est beaucoup plus simple :

```
Lemma true_not_false_tactics : (H : true = false) -> False.
Proof. intro H. inversion H. Qed.
```

La première tactique, `intro H`, indique à `Coq` que le terme de preuve sera une fonction, dont l'argument est appelé `H`. En pratique, l'utilisateur·ice voit l'hypothèse `H : true = false` ajoutée à son contexte local, initialement vide, et le but passer de `(H : true = false)-> False` à simplement `False`.

La deuxième tactique utilisée est plus complexe, elle permet de générer les conditions nécessaires pour que `H` soit vraie. Mais ici, la tactique est suffisamment puissante pour se rendre compte que `H` est fausse, et donc conclure. `inversion` est capable de construire les termes de preuve qui affirment que les constructeurs d'un type inductif sont distincts. Rappelons que l'écriture de `Qed` appelle le noyau pour vérifier que le terme construit a bien le type du but : `Coq` ne fait pas confiance aux tactiques.

Les tactiques permettent donc à l'utilisateur·ice de se rapprocher du langage mathématique naturel et de lui masquer la construction du terme de preuve qui sera donné au noyau. Parmi les tactiques basiques de `Coq` nous retrouvons donc l'introduction d'hypothèse (`intro`), l'analyse de

cas (`destruct`), l'induction (`induction`). L'une d'entre elles est particulièrement importante : `reflexivity`, qui clôt un sous-but de la forme $x = y$ avec $x \equiv y$. Elle recourt à la conversion de `Coq` et utilise ensuite le constructeur `eq_refl` de l'égalité. Par ailleurs `Coq` propose des langages pour combiner ces tactiques que nous verrons au paragraphe 3.4 ainsi que des *tactiques d'automatisation* dont le but est de résoudre les buts simples d'un seul coup. Par exemple, la seule tactique `intuition` aurait résolu l'exemple (elle fait du raisonnement propositionnel et utilise quelques propriétés basiques des inductifs).

### Les classes de types

Nous aurons l'occasion de parler de classes de types (ou *typeclass*) à plusieurs reprises. Décrivons donc de quoi il s'agit. Une classe de type permet de définir des spécifications et des propriétés abstraites, qui seront par la suite instanciées par des types concrets. Cela permet de raisonner d'abord sur des propriétés génériques avant de considérer des cas particuliers. Par exemple considérons cette classe de type :

```
Class EqDec (A : Type) :=
{ eqb : A -> A -> bool ;
  eqb_leibniz : forall x y, eqb x y = true -> x = y }.
```

Les *instances* de celle-ci seront les types possédant une égalité booléenne décidable. Pour prouver qu'un type `T` est une instance de la classe de type, il faut donner une fonction `eqb_T : T -> T -> bool` et une preuve de `eqb_leibniz T`. Par exemple, `nat` est une instance de la classe de type.

```
#[refine] Instance nat_eqdec : EqDec nat :=
{ eqb x y := Nat.eqb x y}.
apply beq_nat_true_iff. (* Lemme de type: forall x y, Nat.eqb x y <-> x = y *)
Defined.
```

Le mot-clef `#refine` signifie qu'il manque un champ de la classe de type, et que celui-ci sera défini via l'utilisation de tactiques plutôt que via l'écriture d'un terme de preuve complet.

Ici, pour être membre de la classe de type `EqDec`, il faut non seulement se donner une égalité décidable, mais aussi une preuve que ladite définition est bien une égalité (champ `eqb_leibniz`). C'est le rôle du script de preuve `apply beq_nat_true_iff`.

Une classe de type peut être paramétrée par une autre (ou par elle-même). Cela permet par exemple de prouver que l'on dispose d'une instance de `EqDec T`, alors on peut construire une instance de `EqDec (list T)`. La notation `‘x: SomeTypeclass A` permet de spécifier que l'instance construite dépend d'un contexte où `x: SomeTypeclass A` apparaît (et où l'on dispose de toutes les fonctions associées à `SomeTypeclass A`). Ainsi, dans `list_eqdec`, on peut retrouver la fonction `eqb` pour le type `T` donné en paramètre de `list`.

```
#[refine] Instance list_eqdec ‘{eqt : EqDec T} : EqDec (list T) :=
{ eqb := fix f (l1 l2 : list T) :=
  match l1, l2 with
  | [], [] => true
  | x :: xs, y :: ys => eqb x y && f xs ys
  | _, _ => false
  end }. ...
```

## 2.2 SMTCoq, le backend principal

Dans ce paragraphe, nous présentons en détail SMTCoq [2], le backend principal de notre outil. La raison pour laquelle nous avons choisi de distinguer ce backend des autres que nous avons pu tester avec Sniper, en lui consacrant une sous-partie entière, est que la grande majorité des transformations concrètes de Sniper ont été écrites pour pallier avant tout les limitations de SMTCoq.

Autrement dit, Sniper sert d'abord à améliorer SMTCoq, même si, au fur et à mesure de son développement, nous avons vu que ses transformations pouvaient permettre à d'autres tactiques de réussir à prouver certains buts Coq.

Rappelons cependant que la méthodologie de Sniper se veut générique et applicable à divers outils, même si en pratique, cette thèse se centre sur l'amélioration de SMTCoq.

### 2.2.1 Fonctionnement

Le plugin SMTCoq fournit des tactiques et des commandes qui permettent d'appeler des prouveurs automatiques au sein de Coq, à savoir des prouveurs SAT ou SMT. Nous définirons la logique qu'ils utilisent et leur fonctionnement en 2.2.2.

Un objectif de SMTCoq est de *ne pas étendre la base de confiance de Coq*. On aurait pu imaginer une tactique qui encode un but Coq $G$ au format d'entrée d'un prouveur automatique, puis qui demande à celui-ci de résoudre le but. Si le prouveur renvoie une erreur ou « no » alors Coq considérerait que le but n'est pas prouvé et si le prouveur renvoie « yes », le but serait considéré comme prouvé, bien que Coq ne dispose pas de la preuve. Ce mode de fonctionnement signifierait que l'utilisateur·ice prend pour base de confiance deux outils : tout d'abord Coq et plus précisément son noyau, et ensuite le prouveur automatique utilisé. Mais le problème est que les prouveurs automatiques n'ont pas de noyau logique clairement identifié. Leur base de confiance est l'entiereté du code, code souvent très complexe, difficile à certifier, et écrit dans un langage de programmation impur, comme C ou C++ par exemple. Cela ne veut pas dire que les prouveurs automatiques ne sont pas fiables : ils sont testés sur des milliers de buts et la confiance que l'on a dans ces outils repose donc sur leur capacité à répondre juste (c'est-à-dire à trouver si un ensemble de formule est satisfiable ou non) à tous ces tests. Mais d'un point de vue théorique ou mathématique, on peut considérer qu'une croyance empirique en la fiabilité d'un outil, bien que très solide, n'est pas suffisante. Par ailleurs, si Coq étend sa base de confiance à celle de l'outil, on perd le terme de preuve, qui devient un simple axiome. Or les axiomes n'ont pas de contenu calculatoire. Pour toutes ces raisons, SMTCoq utilise une autre approche.

Dans SMTCoq, une fois que le but Coq $G$ est envoyé au prouveur et résolu par celui-ci, SMTCoq récupère le certificat de la preuve de $G$ produite par le prouveur et le reconstruit en Coq. Le fait de récupérer un certificat de preuve plutôt que d'analyser le code du prouveur signifie également que SMTCoq adopte une méthode *sceptique*. Quand on utilise un prouveur automatique que l'on certifie dans un assistant de preuve, deux options sont possibles :

1. On peut certifier tout le code du prouveur dans l'assistant de preuve. Ceci revient à réécrire le code du prouveur dans l'assistant de preuve et à démontrer qu'il est correct. C'est la méthode *autarcique*. Mais prouver des procédures de décision ou des techniques d'instanciation de quantificateurs fondées sur des heuristiques complexes est une tâche très difficile. De plus, le moindre changement dans le code du prouveur cassera la preuve de correction et il faudra donc la réparer à chaque mise à jour. Le code des preuves est donc peu maintenable, on risque d'avoir un outil qui devient vite obsolète.
2. On peut *reconstruire dans le langage de l'assistant* la preuve fournie par l'outil automatique. C'est la méthode *sceptique*. Elle a l'avantage de conduire à des preuves beaucoup plus faciles

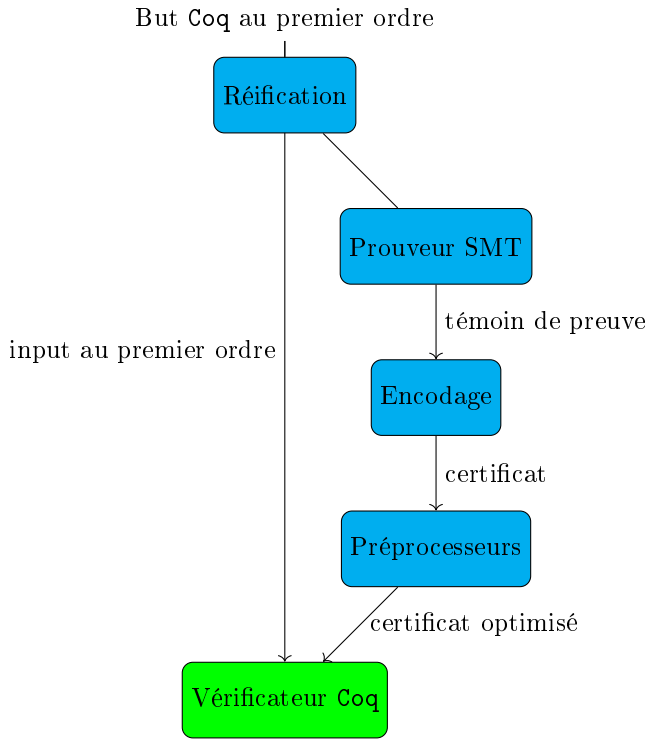



FIGURE 2.1 – Le fonctionnement de SMTCoq

à écrire puisqu'on raisonne seulement sur les certificats de preuve (la sortie du prouveur) et non plus sur le prouveur entier. Par ailleurs, un changement dans le code du prouveur n'affecte pas nécessairement les preuves Coq liées à la forme des certificats (à moins que le prouveur ne change de format de certificat bien sûr). Un autre avantage est que l'on peut choisir un même type pour représenter les certificats venant de prouveurs différents. On est donc plus modulaire, dans le sens où pour brancher un nouveau prouveur, il n'y a pas besoin de se relancer dans des preuves. Il faudra encoder le format de ses certificats dans le terme correspondant de l'assistant de preuve, et l'étendre (par exemple en rajoutant des constructeurs) le cas échéant.

Nous pouvons résumer le fonctionnement de SMTCoq par la figure qui va suivre : 2.1. (Nous retrouverons cette figure lorsque nous verrons l'apport de Sniper à SMTCoq : 3.1.)

Commentons les différentes étapes de ce schéma :

1. Le but Coq d'entrée, envoyé aux prouveurs, doit être dans la logique des prouveurs, c'est-à-dire la logique du premier ordre pour les prouveurs SMT ou la logique propositionnelle classique pour les prouveurs SAT. Autrement, l'utilisateur·ice recevra un message qui dira que la réification a échoué.
2. Ce but est *réifié*, c'est-à-dire qu'on récupère l'arbre syntaxique des termes Coq qui représentent le but et les éventuelles hypothèses, de façon à ensuite le ou les traduire dans le format d'entrée des prouveurs (SMT-LIB).

3. Le prouveur va tenter de résoudre ce but : plus précisément, il essaie de dériver une contradiction à partir des hypothèses et de la négation du but. S'il trouve une preuve, on passe à l'étape suivante.
4. Un témoin de preuve est produit. Ce témoin (ou certificat) dépend du prouveur utilisé, il faut donc l'encoder dans un format commun à tous les prouveurs.
5. Bien souvent, les certificats ne sont pas assez précis. Il manque des étapes de preuves, notamment parce que le prouveur fait des simplifications silencieuses sur les formules. Il faut reconstruire les morceaux de preuve manquants. Une optimisation importante est faite aussi : comme les certificats sont souvent très gros, les formules sont représentées sous la forme de tableaux persistants en `Coq`.
6. Le vérificateur `Coq` est alors appelé. Dans le certificat, une dérivation d'une contradiction a été faite à partir du but `Coq` initial, ce vérificateur s'occupe donc de valider chacune des étapes du certificat. La validation de chaque étape distincte est effectuée par un vérificateur dédié (par exemple, il y en a un dédié à l'arithmétique, un dédié au raisonnement propositionnel, etc.). Le vérificateur final n'utilise donc que la combinaison des vérificateurs atomiques. On a alors deux possibilités : soit le certificat produit par le prouveur est bien une preuve que la négation du but conjointe aux hypothèses est contradictoire et le but est prouvé, ou on a un message d'erreur.

### 2.2.2 Les prouveurs SAT et SMT

Comme la logique des prouveurs SMT est la logique cible, nous devons les présenter plus précisément. Un prouveur SMT est divisé en deux parties : la partie qui gère la logique propositionnelle et qui contient un prouveur SAT, et la partie qui gère les théories, contenant un module par théorie. Présentons donc d'abord brièvement les prouveurs SAT.

#### Les prouveurs SAT

Un prouveur SAT est un logiciel où est implémenté un algorithme (une variante de l'algorithme Davis-Putnam-Logemann-Lovelan [34]) qui décide un problème SAT. Un problème SAT répond à la question suivante : étant donné un ensemble de *clauses* $\mathcal{C}$ de la logique propositionnelle classique, existe-t-il une *valuation* $\rho$ pour laquelle toutes ces clauses sont satisfaites ?

— Les clauses sont des formules de la forme $l_1 \vee ... \vee l_n$, avec les $l_i$ des littéraux, c'est à dire des formules atomiques ou leur négation. La conjonction est donc implicite dans les problèmes SAT : une formule $p \wedge \neg q$ sera considérée comme deux clauses $p$ et $\neg q$.

— Une valuation assigne $\top$ ou $\bot$ à toutes les variables propositionnelles présentes dans les clauses, elle permet de définir trivialement l'interprétation d'une formule $P$ comportant des connecteurs (qu'on notera $\mathcal{I}(P)$).

Ainsi, un témoin de preuve pour un problème SAT consiste en une valuation $\rho$ pour laquelle $\forall P \in \mathcal{C}$, $\mathcal{I}(P) = \top$. Quand le prouveur répond UNSAT, le témoin de preuve sera une dérivation de la clause vide, (c'est à dire la formule atomique $\bot$ qui est fausse quelle que soit la valuation), en appliquant à plusieurs reprises *la règle de résolution*.

La règle de *résolution* s'énonce ainsi :

$$\frac{p \vee l_1 \vee ... \vee l_n \quad \neg p \vee m_1 \vee ... \vee m_k}{l_1 \vee ... \vee l_n \vee m_1 ... \vee m_k}$$

Voici par ailleurs un exemple de problème SAT pour lequel le solveur répondrait UNSAT, ainsi qu'un certificat qu'il pourrait fournir. Comme précédemment, $\mathcal{C}$ désigne l'ensemble de clauses initial. $\mathcal{C} = \{x \vee y, \neg z, z \vee \neg y, \neg x \vee y\}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{z \vee \neg y \qquad \neg z}{\neg y} \qquad x \vee y}{x} \qquad \neg x \vee y}{y} \qquad \neg y}{\bot}$$

Dans cette dérivation, le $\neg y$ est une clause intermédiaire qui est ajoutée à l'ensemble $\mathcal{C}$, de manière à servir plus tard pendant la dérivation (pour déduire la contradiction).

Les prouveurs SAT ne permettent de raisonner que sur une logique très limitée. Mais, dès que l'on passe à la logique du premier ordre, le problème de satisfiabilité d'un ensemble de formules n'est plus décidable.

### Les prouveurs SMT

Les prouveurs SMT sont des prouveurs dont le langage est la logique du premier ordre, spécialisés à des *théories décidables*, qu'ils peuvent d'ailleurs combiner, et sur lesquels ils sont particulièrement performants. Il faudrait plutôt dire *les* logiques du premier ordre puisque, outre les formules atomiques ($x, y, ..., \top, \bot$), et les connecteurs logiques ($\rightarrow, \vee, \wedge, \neg, \forall, \exists$), toute logique du premier ordre s'accompagne d'une signature $\Sigma$ dans laquelle sont définis les symboles de fonction et les prédicats.

Il faut distinguer les prouveurs SMT des prouveurs automatiques dits « du premier ordre » qui sont généralement plus efficaces sur des problèmes quelconques de logique du premier ordre, mais bien moins performants sur des théories données. Ces derniers ne sont pas utilisés dans `SMTCoq` mais dans d'autres outils comme `hammer` (voir 3.2.1).

Étant donnés une signature $\Sigma$ contenant des types simples (c'est-à-dire des types de données de base ou des constantes désignant un type abstrait), des fonctions et des symboles de prédicats munis de leur arité (typée), une théorie $\mathscr{T}$ est un ensemble de formules booléennes (les axiomes) exprimées grâce à cette signature et les symboles/variables usuels de la logique du premier ordre.

Les prouveurs SMT répondent à la question : un ensemble de formules $\mathscr{F}$ donné est-il satisfiable dans la théorie $\mathscr{T}$ ? Autrement dit, $\mathscr{F}$ est-il $\mathscr{T}$-satisfiable ?

Les prouveurs SMT sont composés de différents modules : un prouveur SAT qui gère la partie propositionnelle du problème, et un module pour chaque théorie. Les deux communiquent entre eux jusqu'à trouver un modèle de $\mathscr{F} \cup \mathscr{T}$ ou jusqu'à dériver la clause vide. Le solveur SAT considère chaque atome comme une variable propositionnelle et propose un modèle $M$ qu'il envoie aux solveurs de théories. Ceux-ci vérifient alors indépendamment si $M$ est compatible avec les axiomes de la théorie dont ils se chargent. Si c'est le cas, alors le prouveur répond `SAT`. Si ce n'est pas le cas, alors le module de théorie qui a trouvé la contradiction rajoute une clause $C$ dans l'ensemble des clauses telle que $M$ n'est plus un modèle de $\mathscr{F} \cup C$. Le choix de $C$ est fait de telle sorte que $C$ soit une « explication » de l'incohérence de $M$. Ensuite, ce module de théorie rappelle le module SAT sur ce nouvel ensemble de clauses, jusqu'à aboutir sur $\bot$ ou jusqu'à trouver un modèle compatible avec $\mathscr{T}$.

Pour mieux illustrer comment les modules de théorie et le prouveur SAT opèrent, considérons l'exemple suivant :

— Nous considérons une seule théorie $\mathscr{T}$, la théorie dite EUF (*Equality Logic With Uninterpreted Functions*). La signature $\Sigma$ contient un symbole de prédicat binaire $\cong$ qui vérifie les trois axiomes suivants (réflexivité, symétrie, transitivité) :

— $\forall x,\ x \cong x$

— $\forall x\ \forall y,\ x \cong y \to y \cong x$

— $\forall x\ \forall y\ \forall z,\ x \cong y \wedge y \cong z \to x \cong z$

Pour tout symbole de fonction $f$ d'arité $n$ de $\Sigma$, on ajoute l'axiome de congruence :

— $\forall \vec{x}\ \forall \vec{y},\ \bigwedge_{i=1}^{n} x_i \cong y_i \to f(\vec{x}) \cong f(\vec{y})$

De même, pour tout symbole de prédicat $P$ d'arité $n$ de $\Sigma$, on ajoute un axiome de congruence :

— $\forall \vec{x}\ \forall \vec{y},\ \bigwedge_{i=1}^{n} x_i \cong y_i \to P(\vec{x}) \to P(\vec{y})$

— L'ensemble des clauses $\mathscr{C}$ est : $\{b \cong c \vee f(b) \cong c,\ a \ncong g(b),\ g(f(c)) \cong a\}$.

— Le module SAT considère chaque atome comme des variables propositionnelles et transforme ainsi $\mathscr{C}$ en $\mathscr{C}' = \{x \vee y,\ \neg z,\ w\}$. Il trouve un modèle $M$ qui assigne la valeur de vérité $\top$ à toutes les variables sauf à $z$ qui reçoit la valeur $\bot$.

— Le module de théorie détecte que ce modèle est incompatible avec la théorie $\mathscr{T}$, car $b \cong c \wedge f(b) \cong c \wedge g(f(c)) \cong a \models a \cong g(b)$. Il ajoute la clause $C = b \ncong c \vee f(b) \ncong c \vee g(f(c)) \ncong a \vee a \ncong g(b)$ à l'ensemble des clauses.

— Le module SAT est rappelé et raisonne cette fois-ci sur l'ensemble $\{x \vee y,\ \neg z,\ w,\ \neg x \vee \neg y \vee \neg z \vee \neg w\}$. En assignant $\top$ à $y$ et $w$ et $\bot$ à $x$ et $z$, on obtient un autre modèle.

— Cette fois-ci, le modèle n'est pas rendu incohérent par la théorie et le solveur peut répondre `SAT`.

**Mise en forme normale** Un aspect très important de l'utilisation de ces prouveurs est qu'ils ont besoin de prétraiter la formule qu'ils prennent en entrée. En particulier, ils la mettent sous forme *prénexe* (avec les quantificateurs en tête de formule), la skolémisent (éliminent les quantificateurs existentiels pour obtenir une formule équisatisfiable), et la mettent en *forme normale conjonctive* (CNF). Dans cette forme, les implications sont éliminées (grâce à la règle classique $A \to B \equiv \neg A \vee B$), et la formule est transformée en une conjonction de disjonction de littéraux grâce à la méthode de Tseitin qui utilise des règles de la logique classique.

Ceci implique en particulier que la logique de `SMTCoq` est elle aussi *classique*. Le préprocessing consistant à mettre les formules d'entrée en forme prénexe et en CNF utilise certaines règles classiques. Ainsi, même s'il existait une preuve constructive de certains buts, les preuves fournies par le plugin seront classiques et ne pourront pas être reconstruites en preuves intuitionnistes sans revoir l'architecture de `SMTCoq`.

Le plugin n'utilise cependant pas d'axiome classique : pour obtenir des preuves compatibles avec la logique intuitionniste, les hypothèses et le but doivent être exprimés sous la forme d'égalités entre booléens. En effet, en Coq la logique classique est valide sur les booléens. Cela demande de prouver ou de supposer la décidabilité d'égalités sur certains types (voir 2.2.3).

**Prouveurs et théories utilisées dans `SMTCoq`** Les prouveurs SMT utilisés dans `SMTCoq` sont `cvc4` (pour la théorie des tableaux et celle des vecteurs de bit) et `veriT` (pour la précision des certificats qu'il produit au sujet des quantificateurs). `SMTCoq` utilise également le prouveur SAT `zChaff`. Cependant, dans `Sniper`, comme notre objectif est de traiter un maximum de la logique de `Coq`, nous avons utilisé la tactique fournie par `SMTCoq` qui fonctionnait avec la logique la plus expressive possible. Or, la seule tactique supportant les quantificateurs (du premier ordre) est la tactique appelant le prouveur `veriT`. Notons que le support des lemmes quantifiés qui peuvent être transmis au prouveur fut l'objet d'un travail ultérieur aux premières versions du plugin [7].

### 2.2.3 Limitations de SMTCoq

Avec SMTCoq, nous sommes *a priori* limité·es dans notre automatisation car les buts supportés sont booléens et au premier ordre. Par ailleurs, SMTCoq n'encode pas les types et les fonctions définis par les utilisateur·ices, il traduit seulement les types interprétés par les prouveurs.

Nous allons décrire les principales limites de SMTCoq à travers la présentation d'une classe de type restreignant les types sur lesquels des preuves automatiques sont possibles et à travers les premières extensions du plugin qui ont permis d'automatiser un plus grand nombre de buts.

#### La classe de type CompDec

Examinons d'abord la classe de type CompDec et ses propriétés, car sa présence dans le code de SMTCoq a eu une grande influence sur l'implémentation de Sniper. En particulier, elle a fortement contraint la forme des buts que l'outil pouvait prouver automatiquement, mais en contrepartie j'ai pu également me servir des propriétés de cette classe de type dans certaines des transformations que j'ai écrites.

**Pourquoi a-t-on besoin de cette classe de type ?** Dans SMTCoq, les termes, les types et les formules qui seront envoyés aux prouveurs sont représentés par des types inductifs. Ainsi, par exemple, pour le type `form` des formules, on retrouve un constructeur pour les atomes, un pour le « ou », un pour le « et », etc. Il s'agit donc d'un *encodage profond* de la logique des prouveurs en Coq. Toutefois, les preuves de correction de SMTCoq reposent sur une *interprétation* de ces inductifs, c'est-à-dire une fonction qui permet de passer d'un encodage profond à un *encodage superficiel* (d'inductifs représentant la logique cible aux termes Coq concrets correspondant). Les fonctions d'interprétation sont difficiles à typer. Le type des formules est interprété par le type `bool`, le type `Ty: type` d'un terme est interprété par un type de Coq : `interp_type Ty`, et les termes sont interprétés par une paire dépendante entre un `type` et un élément de l'interprétation de ce type. Autrement dit, pour `t: term` un terme de SMTCoq :

```
interp_term t: {Ty: type & interp_type Ty}
```

Nous simplifions volontairement ici cette description, mais il ne s'agit pas de l'implémentation exacte dans SMTCoq.

Par ailleurs, l'encodage profond des types de SMTCoq correspond à l'inductif suivant :

```
Inductive type :=
| TFArray : type -> type -> type (* tableaux *)
| Tindex : index -> type (* variables de type *)
| TZ : type (* entiers relatifs *)
| Tbool : type (* booleens *)
| Tpositive : type
(* entiers strictement positifs binaires,
utiles seulement parce que les
constructeurs de Z prennent des "positive" en argument *)
| TBV : N -> type. (* bitvectors *)
```

En particulier, le type des tableaux est présent car le prouveur cvc4 permet de travailler avec leur théorie. L'encodage du type des tableaux `TFArray` est interprété par un type des tableaux `FArray` formalisé dans SMTCoq, qui vérifie les axiomes de la théorie des tableaux de SMT-LIB. En particulier, une valeur par défaut doit exister pour les cas où l'on tente d'accéder à une case du tableau plus grande que sa taille, donc le type des éléments du tableau doit être habité, et le type des clefs permettant de faire des recherches dans le tableau doit être totalement ordonné et muni

d'une égalité décidable. Le type `FArray` prend donc en argument deux types et les propriétés requises sur ceux-ci.

Ainsi, la classe de type `CompDec` regroupe toutes ces propriétés.

```
Class CompDec T := {
  ty := T;
  Eqb :> EqbType ty;
  Ordered :> OrdType ty;
  Comp :> @Comparable ty Ordered;
  Inh :> Inhabited ty
}.
```

Comme `CompDec` requiert notamment une égalité décidable, il existe une égalité booléenne `eqb_of_compdec` dans cette classe de type, telle que `eqb_of_compdec : forall (A: Type), CompDec A -> A -> A -> bool`, que je mentionne car elle sera utilisée en 4.3.4.

Les contraintes spécifiques au type des tableaux se retrouvent dans la fonction d'interprétation de tous les types. On a donc :

```
interp_type (t:type) : {ty: Type & CompDec ty}
```

Ici, on remarque que le type de retour de `interp_type` a été étendu en une paire dépendante contenant `CompDec`.

Ainsi, dans le cas `interp_type (TFArray t1 t2)`, on peut interpréter récursivement `t1` et `t2`. On obtient deux paires dépendantes dont on récupère les premières projections `interpt1` et `interpt2` puis les preuves `H: CompDec interpt1` et `H1: CompDec interpt2`. Enfin, on peut utiliser ces résultats pour renvoyer la bonne interprétation `FArray interpt1 H interpt2 H1`. Pour tous les autres constructeurs, cette classe de type n'est pas utilisée dans `interp_type`, mais comme elle est nécessaire dans le cas des tableaux, elles apparaît dans le type de retour. L'égalité décidable seule est utilisée pour l'interprétation des atomes, car nous sommes en logique classique et ne pouvons donc admettre que des prédicats décidables.

La présence de `CompDec` sur tous les types de `SMTCoq` est une contrainte forte parce qu'elle ne permettra ensuite que des raisonnements sur des types vérifiant tous les champs de la *typeclass*. Dans `Sniper`, les buts sur un type abstrait `A` devront donc présenter cette hypothèse supplémentaire `CompDec A`, même si nous n'utilisons pas la théorie des tableaux.

Cependant, des instances de `CompDec` sont prouvées pour des types usuels, par exemple `nat` ou `bool`. De même, des preuves paramétriques de `CompDec` existent, comme par exemple la preuve de : `forall (A : Type), CompDec A -> CompDec (list A)`.

Le code définissant cette classe de type est disponible au lien suivant : https://github.com/smtcoq/smtcoq/blob/SMTCoq-2.2+8.13/src/classes/SMT_classes.v.

Les preuves `Coq` pour certains types usuels sont disponibles ici : https://github.com/smtcoq/smtcoq/blob/SMTCoq-2.2+8.13/src/classes/SMT_classes_instances.v

### Traductions entre représentations de structures de données et entre logiques

En `Coq`, plusieurs représentations d'une même structure de donnée coexistent, pour des raisons de performance, de facilité à effectuer des preuves sur une structure plutôt que sur l'autre etc. Par exemple, dans la bibliothèque standard, on trouve le type des entiers naturels unaires (`nat`) et celui des entiers binaires (`N`), tout comme le type des entiers strictement positifs (`positive`), qui servent à construire les entiers relatifs (`Z`). Il existe des isomorphismes entre ces structures (ou des sous-ensembles de celles-ci).

Or, comme nous l'avons vu au paragraphe précédent, les types interprétés pour le prouveur SMT ne concernent que les entiers de `Z`. Toutes les autres représentations sont considérées comme

des types non interprétés pour le prouveur, qui ne peut alors utiliser ses lemmes de théories à leur sujet.

Un premier travail [7] tente donc de traduire certains types de données vers `Z`, à l'aide d'une tactique appelée `nat_convert`, en ajoutant éventuellement des conditions quand on dispose d'un plongement mais non d'un isomorphisme. Le but est de disposer d'une traduction qui passe sous les symboles de fonctions non interprétés : en effet, le prouveur SMT doit combiner des théories (et en particulier EUF, la théorie de l'égalité et des fonctions non interprétées, avec celle de l'arithmérique linéaire).

**Exemple 8** [Traduction entre `nat` et `Z`]

Dans le code qui suit, on voudrait obtenir un but au sujet de `Z` :

```
Lemma uninterpreted_symbol_nat :
forall (f : nat -> nat) (n: nat),
Nat.eqb (f (n + 0)) (f n) = true.
```

L'appel à `nat_convert` permet de transformer le but comme on le souhaite :

```
forall (f' : Z -> Z) (z: Z),
((0 <=? z) --->
(0 <=? f' z) --->
(0 <=? f' (z + 0)) ---> (f' (z + 0) =? f' z)) = true
```

Ici, `<=?` est l'inégalité dans `bool` sur `Z`, et `=?` l'égalité. Les conditions associées au plongement sont écrites avec une implication booléenne (`--->`) parce que `SMTCoq` effectue ses preuves sur des booléens.

Une limite importante de `nat_convert` est que la tactique ne se comporte pas bien sur les buts qui ne sont pas déjà exprimés comme une égalité entre deux booléens.

Or, les utilisateur·ices de `Coq` ne souhaitent pas nécessairement exprimer leurs buts, même parlant d'égalités décidables, directement dans `bool`. Une autre tactique de `SMTCoq` appelée `prop2bool` permet, quand cela est possible, de passer d'un but `G` exprimé dans `Prop` à un but de la forme `b=true`, où bien sûr `b : bool` est l'équivalent booléen de `G`.

Malheureusement, `prop2bool` et `nat_convert` n'interagissaient pas bien ensemble, ce qui signifie qu'un but tel que : `forall (f : nat -> nat)(n: nat), f (n + 0)= f n` ne peut pas être résolu par `SMTCoq`, à cause de la présence de l'égalité en tant qu'inductif `Coq` *et* d'un type de donnée d'entiers qui n'est pas celui des prouveurs automatiques.

Cependant, ces deux premières tactiques de préprocessing intégrées dans le code de `SMTCoq` peuvent être considérées comme des transformations logiques, au même titre que celles du code de `Sniper`.

Entre temps, un plugin `Coq` plus puissant appelé `Trakt` qui subsume la tactique `nat_convert` *et* la tactique `prop2bool` été implémenté par Enzo Crance, Denis Cousineau et Assia Mahboubi [14]. À l'aide d'une base de données que l'utilisateur·ice alimente en lemmes de plongement ou d'isomorphisme entre ses types de données favoris, la tactique `trakt` est capable de convertir un type de données en un autre, dans une logique cible (`bool` ou `Prop`) fixée par laquelle elle est paramétrée.

**Exemple 9** [Plongement de `nat` dans `Z` avec passage de `Prop` à `bool`]

L'objectif de l'utilisateur·ice est de plonger `nat` dans `Z`, de manière à ce que tout prédicat (égalité ou inégalité dans ce cas-ci) sur les entiers avec un codomaine dans `Prop` soit transformé en un prédicat avec un codomaine dans `bool`.

Il ou elle doit d'abord prouver des lemmes de plongement sur les fonctions `Z.of_nat` et `Z.to_nat` qui sont des inverses partiels :

— `nat_Z_id: forall (n : nat), n = Z.to_nat (Z.of_nat n)`

— `nat_Z_bool_cond_id: forall (z : Z), (0 <=? z)= true -> Z.of_nat (Z .to_nat z)= z`

— `nat_Z_bool_cond_true: forall (n : nat), (0 <=? Z.of_nat n)= true`

Ensuite, il ajoute à la base de données de Trakt ces lemmes et leurs preuves. Ces dernières seront utilisées lors du préprocessing du but : Trakt génère à la volée une preuve que le but transformé est équivalent au but initial.

```
Trakt Add Embedding nat Z Z.of_nat Z.to_nat nat_Z_id nat_Z_bool_cond_id
 nat_Z_bool_cond_true.
```

L'utilisateur·ice gère aussi les symboles, par exemple le +, en démontrant :

```
add_embedding: forall (n m : nat), Z.of_nat (n + m)= ((Z.of_nat n)+ (Z.
of_nat m)).
```

Il ou elle ajoute les lemmes sur ce symbole à la base de données :

```
Trakt Add Symbol (Nat.add)(Z.add)(add_embedding)
```

Il ou elle procède de même pour le symbole d'arité nulle 0, il ajoute aussi des lemmes sur l'égalité, et finalement, il ou elle peut faire appel à Trakt :

```
Lemma uninterpreted_symbol_nat :
forall (f : nat -> nat) (n: nat),
(f (n + 0)) (f n) = true.
Proof. trakt Z bool. (* le type canonique cible est Z, la logique
    cible est bool* )
(* Trakt donne: forall (f' : Z -> Z) (z: Z), ((0 <=? z) ->
(0 <=? f' z) ->
(0 <=? f' (z + 0)) ->
Z.eqb (f' (z + 0)) (f' z) = true) *)
```

Par défaut, Trakt ne convertit pas les flèches de Prop vers celles de bool, mais une petite tactique de préprocessing supplémentaire permet de le faire, pour obtenir enfin un but entièrement dans bool.

On part donc d'un but non prouvable par SMTCoq et non traitable par la combinaison de nat_convert et prop2bool, pour aboutir à un but préprocessé qui peut être prouvé automatiquement.

Dans l'état actuel des choses, deux branches de SMTCoq coexistent :

— La branche principale de SMTCoq utilise prop2bool et nat_convert.

— Une branche de SMTCoq utilise Trakt. Cependant, malgré la plus grande puissance et la plus grande modularité de cette tactique, des tests unitaires échouent encore, ce qui empêche la fusion de cette branche avec la branche principale. La raison est que SMTCoq utilise le type `bitvector N` et trakt essaie de convertir le paramètre N, ce que nous souhaitons éviter.

*Sniper utilise les deux branches et le code variera légèrement selon les branches. Certaines transformations* ne peuvent pas fonctionner *sans trakt, mais il faut en même temps que Sniper puisse être compatible avec la branche master de son backend.*

Il faut également retenir qu'en pratique, même si le cœur de SMTCoq ne supporte *que* des buts booléens au premier ordre et ne pouvant parler que d'un seul type d'entiers, et qu'ainsi les tactiques primitives d'appel aux prouveurs présentent ces limitations, *nous utilisons dans Sniper une version non primitive de l'appel au prouveur, qui utilise soit Trakt soit nat_convert et prop2bool*. Par ailleurs, le *seul* prouveur utilisé comme backend dans Sniper est veriT, muni

de sa tactique (non primitive) `verit`. C'est le seul prouveur de `SMTCoq` qui peut prendre en paramètre des lemmes quantifiés et qui supporte les quantificateurs au premier ordre.

En fin de compte, `SMTCoq` sans `trakt` (ou sans `nat_convert` et `prop2bool`) supporte un fragment booléen de `Coq`, avec quantificateurs universels prénexes et sans quantificateurs existentiels, et certaines théories comme l'arithmétique (de `Z`). Grâce à `trakt` (ou avec `nat_convert` et `prop2bool`), `SMTCoq` supporte un fragment de `Coq` où les types doivent être `CompDec`, avec quantificateurs universels prénexes et sans quantificateurs existentiels, et certaines théories comme l'arithmétique (de `Z`, `nat`, `BinInt` ou `int`).

# Chapitre 3

# MÉTHODOLOGIE

Les prérequis ayant été présentés dans les précédents chapitres, mon travail de thèse commencera à être exposé dans celui-ci, selon deux axes :

— **La méthodologie abstraite** : L'idée est de présenter une méthodologie générique dont l'utilité pour l'automatisation des assistants de preuves en théorie des types commence à être visible, grâce à son application lors de l'implémentation issue de mon travail de thèse, prenant la forme d'un plugin `Coq` appelé `Sniper`. Par ailleurs, l'implémentation a permis d'éclaircir et d'affiner cette méthodologie.

— **La méthodologie concrète** : Le but sera de présenter et de comparer les outils qui ont été nécessaire à mon implémentation concrète. Il s'agit des outils de métaprogrammation en `Coq`. J'espère que cette partie du chapitre pourra résumer l'expérience et le recul que j'ai acquis dans ce domaine et servir aux futur·es chercheur·ses qui souhaiteraient utiliser l'un ou plusieurs d'entre eux. Cette recherche du bon outil de métaprogrammation reste encore une question ouverte et je ne prétendrai pas ici répondre avec assurance à celle-ci, mais j'apporterai des exemples concrets qui pourront être utiles.

## 3.1 Définitions préalables

### 3.1.1 Qu'est-ce qu'une transformation ?

Le plugin `Sniper` implémente des *transformations logiques*, ou *traductions*.

Formellement, une transformation logique est une fonction dont le domaine est l'ensemble des formules d'un langage source $\mathcal{S}$ et le codomaine celui des formules d'un langage cible $\mathcal{T}$. On note cette fonction $[\![.]\!]$. Le contexte d'un séquent est traduit de manière naturelle.

Les transformations logiques doivent souvent vérifier des propriétés qui dépendent de l'objectif que l'on s'est assigné. Comme propriétés importantes, nous pouvons lister :

— La *correction* : Soient $\Gamma$ un contexte et $A$ une formule, cet énoncé dit que s'il existe une preuve de $[\![\Gamma]\!] \vdash_{\mathcal{T}} [\![A]\!]$, alors il existe aussi une preuve de $\Gamma \vdash_{\mathcal{S}} A$. Autrement dit, s'il existe une preuve du séquent transformé, il existe bien une preuve du séquent initial.

— La *complétude* : Soient $\Gamma$ un contexte et $A$ une formule, cet énoncé dit que s'il existe une preuve de $\Gamma \vdash_{\mathcal{S}} A$, alors il existe une preuve de $[\![\Gamma]\!] \vdash_{\mathcal{T}} [\![A]\!]$. Autrement dit, s'il existe une preuve du séquent initial, il existe bien une preuve du séquent transformé.

— Une autre notion de correction se place dans le cadre des langages typés. La fonction $[\![.]\!]$ devient une traduction sur les types et on a une nouvelle fonction de traduction $[.]$ pour les termes. Cette notion de correction s'énonce ainsi : $\Gamma \vdash_{\mathcal{S}} t : A \iff [\![\Gamma]\!] \vdash_{\mathcal{T}} [t] : [\![A]\!]$.

Les deux propriétés qui nous ont intéressé·es dans ce travail de thèse sont la correction et la complétude. Précisons toutefois que correction et complétude sont des termes assez ambigus et sont utilisés dans différentes acceptions (parfois réciproques de la présentation ci-dessus). La correction est souvent la propriété clef que l'on cherche à démontrer ou à garantir, et la complétude est sa réciproque, mais le nom « correction » dépendra du contexte et des objectifs.

La correction (au premier sens ci-dessus) est la propriété clef pour les transformations de `Sniper` car l'objectif est de garantir que si un prouveur automatique arrive à démontrer le séquent transformé, il est possible de retrouver une preuve du séquent initial. Nous nous intéressons donc à la correction du point de vue de l'interaction de la prouvabilité avec les transformations logiques. Beaucoup de transformations de `Sniper` ont la même logique source et cible, et consistent à ajouter des hypothèses dérivables à partir du séquent initial (mettons, une hypothèse $u : B$) Toutes ces transformations sont correctes car si on a $\Gamma \vdash u : B$, alors $\Gamma \vdash t : A \Leftrightarrow \Gamma,\ u :\ B \vdash t : A$.

La complétude est quelque chose que l'on souhaite approcher. Voyons cela par un exemple extrême. Notons $\bot_{FOL}$ la formule atomique toujours fausse en logique du premier ordre. Traduire tous les termes de CIC par $\bot_{FOL}$ est parfaitement correct (toujours au premier sens ci-dessus), mais on ne parviendra jamais à prouver $\vdash_{FOL} \bot_{FOL}$. Le but est donc que la traduction maintienne la prouvabilité du plus grand nombre possible de séquents de départ.

### 3.1.2 Quelques exemples

— La *monomorphisation* est une transformation logique qui permet de passer d'un langage $\mathcal{P}$ avec types polymorphes (c'est-à-dire comportant des variables de type $Type$), à un langage qui n'en comporte pas. En stage de Master, j'ai travaillé sur une transformation de monomorphisation prouvée correcte en `Coq`, nous y reviendrons au paragraphe 3.2.2. Par ailleurs, `Sniper` propose une transformation de monomorphisation, comme décrit ici : 4.4.

— La *défonctionnalisation* est une transformation qui permet de passer d'un langage source avec applications partielles à un langage cible qui n'en contient pas. Elle introduit un prédicat binaire @ et des axiomes dans le langage cible, de telle sorte que si $f : A \to B \to C$, le terme $f\ x$ est considéré comme une fonction $f_x$ avec l'axiome $\forall y, @(f_x,\ y) = f\ x\ y$.

## 3.2 Mantra méthodologique

Ce mantra méthodologie est le cœur de mon travail. Il a guidé toute l'implémentation de `Sniper`, aussi imparfaite une implémentation soit-elle en comparaison de son idéal.

Le voici :

`Sniper` doit être générique et compositionnel.

Les deux notions vont ensemble, comme nous allons le voir.

— **Générique** : Le problème que `Sniper` se propose de résoudre est de permettre l'utilisation de *backends* dans `Coq` (en particulier, des prouveurs automatiques), qui n'ont pas pour logique CIC. Le plugin procède en préprocessant des énoncés de CIC de façon à ce qu'ils soient ensuite exprimés dans la logique cible du *backend*, mais surtout sans figer une implémentation qui serait spécifique à un seul de ces *backends*, ou à une seule logique cible. Il y a déjà eu des travaux nécéssitant un préprocessing de CIC vers la logique du premier ordre, mais le préprocessing en question a été pensé spécfiquement pour un seul type de *backend*, ce qui en limite la portée, comme nous le verrons en 3.2.1.

Ne pas utiliser de préprocessing générique risque aussi de dupliquer le travail et le code résultant, puisque certaines étapes de préprocessing peuvent être communes à différents *backends*. S'approcher d'une forme de généricité, en identifiant un peu plus clairement ce que pourrait être une étape de préprocessing/une transformation logique, est l'un des objectifs de mon travail. Par ailleurs, pour éviter de figer une implémentation trop spécifique et donc atteindre cette généricité, l'idée est d'utiliser la *compositionnalité*.

— **Compositionnel** : Au lieu d'écrire une transformation logique de préprocessing en un bloc, qui supporterait un fragment $CIC_0 \subset CIC$, et qui transformerait tout terme de $CIC_0$ en terme d'une logique cible $\mathcal{L}$, l'idée est de disposer de petites transformations qui peuvent se composer et qui seraient réutilisables pour différents préprocessings (par exemple, pour une logique cible $\mathcal{L}'$ ou un fragment $CIC_1$). Nous avons identifié et implémenté différentes étapes cruciales, les plus atomiques possible, qui, nous l'espérons, pourront servir à différents plugins pour `Coq`. Le but est également de fournir un cadre qui permet à un·e utilisateur·ice d'implémenter, s'il ou elle le souhaite, sa propre transformation et de pouvoir le plus facilement possible l'insérer parmi les transformations déjà disponibles dans `Sniper` (dont il ou elle sélectionnera le sous-ensemble dont il ou elle a besoin).

Plus concrètement, nous pouvons illustrer le fonctionnement du plugin `Sniper` avec son *backend* principal (que nous souhaitons ne pas être le seul !), afin d'avoir une vision plus schématique de comment s'articuleraient les différentes transformations, grâce au schéma suivant : 3.1.

De nouveau, nous avons inclus la partie `SMTCoq` du schéma, qui permet de mieux visualiser où se place le préprocessing de `Sniper`. Mais il faut imaginer que les transformations de `Sniper` pourraient être utilisées pour un autre *backend* que `SMTCoq` et qu'on pourrait donc avoir une sortie différente après l'application des différentes transformations, ainsi qu'un agencement des transformations différent.

Un des avantages de la compositionnalité est aussi la facilité des preuves des transformations. Des tentatives ont été faites pour démontrer la correction d'un très gros préprocessing d'énoncés, mais n'ont jamais abouti car la tâche s'est avérée trop ardue [1]. Ce travail a été fait au sujet du langage de programmation F*, qui permet d'exprimer des spécifications sur son programme. Des conditions de vérification de ces spécifications sont ensuite générées par le vérificateur de types du langage. Elles sont exprimées dans un système de type d'ordre supérieur, et sont ensuite encodées dans la logique du premier ordre pour permettre la résolution de ces conditions par un prouveur SMT. Dans le travail cité, il s'agissait de prouver formellement, en `Coq`, la correction de ce préprocessing du langage typé d'ordre supérieur de F* vers la logique du premier ordre, « d'un seul bloc » (car cet encodage fait partie de la base de confiance de F*). Mais la difficulté était trop importante et le travail n'a pas pu aboutir.

Même démontrer la correction d'une étape atomique de préprocessing, en toute généralité, peut s'avérer longue et fastidieuse, voir ci-dessous au paragraphe 3.2.2 pour une discussion de ce point. Mais à défaut d'être courte et aisée, elle paraît plus envisageable, justement parce que l'étape est suffisamment atomique.

### 3.2.1 Le `Sniper` contre (ou avec ?) le `Hammer`

#### Qu'est-ce que `Hammer` ?

La méthode des *hammers* provient d'un assistant de preuve appelé Isabelle/HOL, qui repose sur une théorie légèrement moins expressive que celle de `Coq` mais basée sur un paradigme différent, dans laquelle la logique est classique. Un outil, `Sledgehammer` [6], a été développé avec succès pour Isabelle/HOL. Un essai pour `Coq` a été fait avec le développement de `CoqHammer` [17]. Mais,

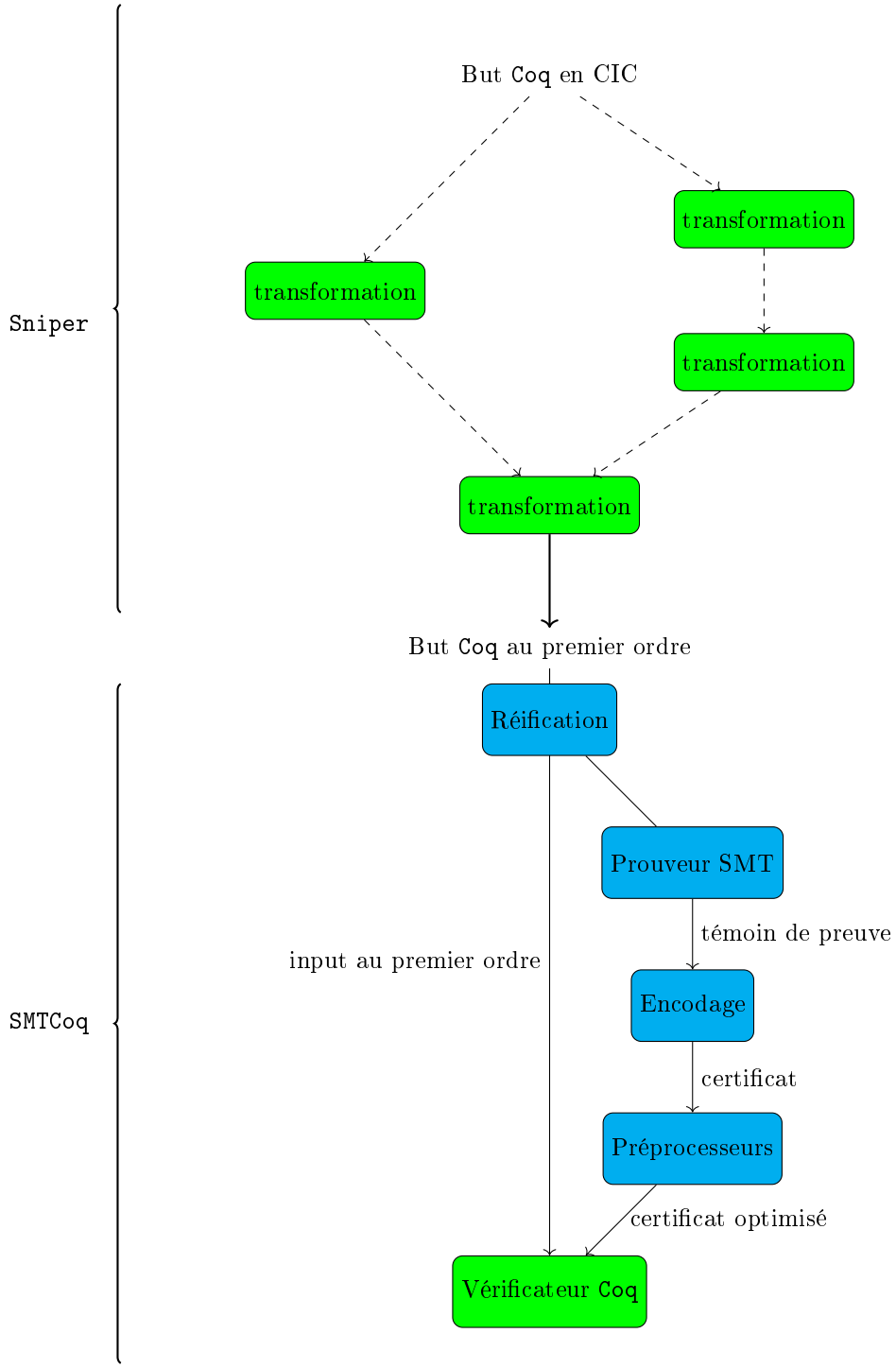



FIGURE 3.1 – La méthodologie de `Sniper`

contrairement à `Sledgehammer`, `CoqHammer` n'est pas beaucoup utilisé dans des développements `Coq`. C'est de ce constat que sont nées les réflexions qui ont conduit à imaginer `Sniper`.

Ainsi, l'idée d'avoir de petites tactiques `Coq` de prétraitement atomiques est née des difficultés rencontrées par l'approche complètement opposée de `CoqHammer`. Les *hammers* sont une famille de tactiques pour des assistants de preuve, dont l'objectif est de fournir une automatisation complète pour l'utilisateur·ice. Autrement dit, celui-ci ou celle-ci n'a qu'à appuyer sur un bouton ou appeler la tactique `hammer` pour que de nombreux buts considérés comme simples mais parfois fastidieux à démontrer, soient prouvés. Les *hammers* reposent sur quatre étapes :

1. La sélection de lemmes : un algorithme d'intelligence artificielle permet, en analysant la forme du but à montrer, de sélectionner une centaine de lemmes qui seront probablement pertinents pour résoudre le but.
2. Le préprocessing : à partir d'un but exprimé dans le langage souvent riche de l'assistant de preuve (présentant de l'ordre supérieur, du polymorphisme, des types dépendants de termes...), le but est encodé dans la logique d'un ou des prouveurs automatiques cibles (typiquement, la logique du premier ordre).
3. Le prouveur automatique résout alors automatiquement le but préprocessé, ou renvoie un message d'erreur.
4. À partir de la preuve générée automatiquement par le prouveur, le *hammer* doit *reconstruire* la preuve dans l'assistant de preuve. Nous verrons comment cela est fait dans `CoqHammer`.

Le plugin `CoqHammer` adapte cette méthode pour le Calcul des Constructions Inductives. L'étape qui nous intéresse le plus ici est l'étape (2), car `Sniper` proposera une méthode alternative. Dans `CoqHammer`, la traduction de CIC vers la logique du premier ordre se fait en trois grosses étapes : tout d'abord, les termes de CIC sont traduits dans une logique appelée $\text{CIC}_0$. Cette première traduction n'est ni correcte, ni complète, et la logique de $\text{CIC}_0$ n'est pas cohérente (on a $Type : Type$). Mais, en pratique, comme les prouveurs automatiques n'ont pas une logique suffisamment riche pour exploiter ce paradoxe, cette incohérence ne pose pas de problème. Une fois le but exprimé dans $\text{CIC}_0$, une traduction en une étape sera faite entre cette logique encore très puissante (car présentant toujours ordre supérieur, point fixes, types dépendants, etc.) et la logique du premier ordre. Il s'agit donc d'une *transformation logique*, mais, elle n'est ni correcte ni complète, comme l'expliquent eux-mêmes les auteurs dans leur article. La dernière étape du préprocessing consiste à faire quelques optimisations à partir des formules et des termes obtenus au premier ordre.

Comme cette traduction n'est pas correcte, il n'est pas possible de reconstruire la preuve d'un but initial $G$ à partir du but transformé $G'$ qui a été démontré par les prouveurs automatiques. La *reconstruction* de la preuve dans `CoqHammer` est en réalité plutôt une *construction*. L'idée est de ne récupérer que, dans les certificats produits par les prouveurs, les lemmes utilisés par ceux-ci. Ensuite, ces lemmes seront des arguments pour une tactique automatique auto-suffisante appelée `sauto` [16], qui construit une preuve en CIC (ou à des variantes de `sauto`). La preuve des prouveurs est donc analysée simplement pour reprendre les lemmes utilisés comme des *hints*, mais le reste de la preuve n'est pas conservé. Il faut bien noter que `sauto` fonctionne indépendamment des prouveurs automatiques et produira une preuve très différente de celle qu'ils ont produit. Ceci est à contraster avec `SMTCoq`, où la preuve `Coq` suit pas à pas celle du prouveur SMT.

Il faut bien noter qu'à ce jour, `hammer`, malgré son manque d'utilisateur·ices, reste encore l'une des tactiques achevant l'automatisation la plus poussée en `Coq`. Cependant, nous considérons que cette approche peut être améliorée, en abordant l'étape de préprocessing très différemment.

Par ailleurs, en Isabelle/HOL, la logique des prouveurs est plus proche de celle de l'assistant de preuve qu'en `Coq`. Ceci peut expliquer pourquoi `Sledgehammer` rencontre plus de succès que `CoqHammer` : le préprocessing est plus aisé en Isabelle/HOL, le but prouvé par les prouveurs

automatiques est plus proche du but initial, ce qui permet notamment que la construction de preuve échoue moins souvent.

### Les limites du `Hammer`

Tout d'abord, notons que la méthode de préprocessing choisie pour `CoqHammer` n'est pas certifiée. Ainsi, elle ne peut pas être reprise telle quelle pour `SMTCoq`. Puisque ce dernier plugin reprend toutes les étapes de la preuve produite par le prouveur externe, la preuve `Coq` obtenue est une preuve du but transformé, que nous avions noté $G'$. Or, le passage de $G$ à $G'$ n'est pas prouvé, ni même prouvable en général, puisque la logique $\mathrm{CIC}_0$ est incohérente.

En traduisant un langage plus faible que $\mathrm{CIC}_0$, on parvient à obtenir une transformation correcte, mais cette démonstration de correction a été faite sur papier [15] et elle doit être faite *dans l'assistant de preuve* pour être acceptée par celui-ci. Or, si nous souhaitons prouver un encodage aussi complexe, nous revenons au problème mentionné ci-dessus, avec la tentative faite dans F*.

De manière plus générale, tout plugin qui utiliserait des prouveurs automatiques entièrement certifiés en `Coq` (ou qui reprendrait leur certificats), aurait besoin d'un préprocessing certifié *dans l'assistant de preuve*. Pour tous ces cas, le préprocessing offert par `CoqHammer` n'est pas adapté.

Remarquons rapidement que l'étape de sélection de lemmes qui recourt à des techniques de *machine learning* pourrait cependant être réutilisée pour d'autres tactiques que `hammer`.

Un second point à remarquer est que cette transformation « d'un seul bloc » de $\mathrm{CIC}_0$ à la logique du premier ordre consiste en un code peu modulaire et difficilement modifiable. Par exemple, serait-il facile de l'adapter pour utiliser des prouveurs SMT au lieu de prouveurs du premier ordre, de façon à bénéficier de leurs modules de théories ? Devra-t-on réécrire un nouvel encodage alors même qu'il y aurait beaucoup de similitudes entre le nouvel encodage et l'ancien ?

Un troisième point, soulevé par les auteurs eux-mêmes, est l'imprédictibilité de la tactique `hammer`. Elle repose sur l'appel de prouveurs automatiques avec une limite de temps. En essayant nous-mêmes, nous avons remarqué que, selon le prouveur qui trouvait la preuve en premier, la tactique de reconstruction pouvait ou non échouer par la suite.

Pour pallier ce problème, l'appel à `hammer` qui est suivi d'un succès propose à l'utilisateur·ice de remplacer `hammer` par la tactique de reconstruction qui a réussi, suivie de ses arguments. Retirer tous les appels à `hammer` permet en effet de ne plus avoir un script de preuve non-déterministe.

Or, si nous disposions d'un préprocessing tel que $G' \implies G$, de telle sorte que l'étape de reconstruction se soucie seulement de la preuve de $G'$, la reconstruction serait plus aisée, et nous perdrions en imprédictibilité.

Un quatrième et dernier point que nous souhaiterions remarquer est que la tactique `hammer` est difficilement extensible. Il faut revoir tout le code du préprocessing pour y ajouter un support pour une partie plus importante de CIC. Or, nous souhaiterions pouvoir étendre et/ou adapter le préprocessing sans revoir tout son code à chaque fois.

C'est ici que `Sniper` entre en jeu. La tactique `hammer` porte ce nom car elle peut être vue comme un gros marteau qui « écrase » le but à démontrer d'un seul coup. Elle évoque la puissance, mais aussi un travail à gros grains, car, pour chaque but à démontrer, on donne le même coup de marteau. Le plugin `Sniper` a été nommé ainsi pour contraster avec les *hammers*. Le sniper est là lui aussi pour résoudre en une seule fois le but d'un assistant de preuve, mais s'y ajoute l'idée de précision, d'ajustement.

Et en effet, le mantra méthodologique de `Sniper` est né de cette limitation principale des *hammers* et en particulier des *hammers* dans les assistants de preuves où le fossé entre la logique de l'assistant et celle des prouveurs est grande (comme `Coq`).

Au lieu de proposer un préprocessing d'un seul bloc que l'on sera incapable de certifier, le préprocessing est décomposé en plusieurs petites étapes, qui seront chacune prouvées dans l'assistant de preuve. Comme ces étapes sont simples, nous serons en meilleure mesure de savoir si un but pourra être prouvé ou non, et nous pourrons prouver la correction de chacune d'entre elles.

#### Le `Sniper` avec le `Hammer`

Nous en touchons déjà un mot ici même si nous développerons cette idée dans nos perspectives, mais, en fin de compte, nous voyons l'approche de `Sniper` et celle de `Hammer` plus complémentaires qu'antagonistes. En effet, la puissance de `Hammer` repose surtout sur la partie « sélection de prémisses », qui est un module indépendant, et sur la tactique de ~~re~~construction de preuve `sauto`. Cette tactique généralise la tactique de recherche de preuve en logique intuitionniste `firstorder` [13].

Or, disposer d'un préprocessing plus robuste, et donc d'user de la méthodologie de `Sniper` dans (une partie de) `Hammer` pourrait permettre d'obtenir un outil plus prévisible et plus extensible.

### 3.2.2 Certifiant ou certifié ?

#### Deux méthodes

Nous souhaitons que la *correction* des transformations logiques soit garantie : pour ce faire, deux méthodes sont possibles.

- **Écrire une transformation certifiée** : nous pouvons démontrer une fois pour toutes la correction de la transformation. Pour cela, nous avons besoin d'exprimer la syntaxe du langage source et cible dans un métalangage. En effet, comme les transformations sont définies inductivement sur la structure des termes, il faut pouvoir accéder à leur syntaxe. Ensuite, à nouveau, deux possibilités s'offrent à nous :
  - Soit nous exprimons dans notre assistant de preuve la sémantique des deux langages $\mathcal{S}$ et $\mathcal{T}$ avec une relation inductive et on démontre que si la traduction d'un séquent est valide dans la sémantique de $\mathcal{T}$, le séquent initial est également valide. Par ailleurs, il faudra démontrer que les deux sémantiques sont correctes vis-à-vis de la sémantique de l'assistant de preuve.
  - Soit nous interprétons les termes et les types de $\mathcal{S}$ et $\mathcal{T}$ dans notre assistant de preuve, grâce à des fonctions permettant de passer des termes de nos langages objets $\mathcal{S}$ et $\mathcal{T}$ à ceux de notre métalangage (celui de l'assistant de preuve). Autrement dit, il nous faut quatre fonctions $\mathcal{I}_{Ty,\mathcal{S}}$, $\mathcal{I}_{term,\mathcal{S}}$, $\mathcal{I}_{Ty,\mathcal{T}}$ et $\mathcal{I}_{term,\mathcal{T}}$ telles que les $\mathcal{I}_{term}$ retournent des termes de l'assistant de preuve et les $\mathcal{I}_{Ty}$ des types de celui-ci. Ensuite, il faudra démontrer un théorème qui dit que si l'interprétation d'un terme $t$ traduit (c'est-à-dire $[t]$) est une paire dépendante entre un type $A$ et un élément $a : A$, alors, on peut calculer l'interprétation de $\mathcal{I}_{term,\mathcal{S}}(t)$ qui sera une paire dépendante entre un type $B$ et un terme $b : B$, tel que $[\![B]\!] = A$.
- **Écrire une transformation certifiante** : cette seconde approche demande moins de travail, mais elle ne garantit pas que la transformation soit correcte une fois pour toutes. À chaque appel de la transformation logique sur un terme concret, celle-ci génère sa preuve de correction (pour ce cas précis). L'approche certifiante consiste à disposer de la transformation logique qui traduit les termes et les types, et d'avoir une fonction (et non un

théorème) qui prend en entrée le but initial et qui génère une preuve que le but transformé l'implique bien.

**Exemple**

— Les tactiques `Coq` sont *certifiantes*. Autrement dit, elles produisent une justification lorsqu'elles sont appliquées (même si cette justification peut éventuellement être refusée au moment du `Qed`, s'il s'avère que la tactique comporte un bug ou ne vérifie pas si l'utilisateur·ice l'a appliquée à raison). Au contraire, utiliser `Admitted` est un exemple de non-certification, puisqu'on produit un énoncé que l'on ne justifie pas et que l'on ajoute à la base de confiance de `Coq`.

**Raisons du choix de l'approche certifiante pour `Sniper`**

Avant d'entamer cette thèse, j'avais déjà commencé un stage sur le même sujet [22]. Or, lors de ce stage, nous avions souhaité écrire une transformation certifiée (pour l'aspect correction) de monomorphisation. Nous avons réussi cette certification (dans une logique plus simple que celle de `Coq`) en utilisant la première des méthodes mentionnées ci-dessus pour écrire des transformations certifiantes, mais, alors qu'il s'agissait seulement d'instancier *un* quantificateur, il fallait déjà une cinquantaine de lemmes et un millier de lignes de code `Coq` pour prouver la correction de cette transformation.

En comparaison, `specialize` fait le même travail, en version certifiante.

Elle transforme un terme $H : \forall(x : A),\ B$, en terme $H : B\ t$ étant donné $t : A$. Ainsi, la tactique `specialize` prend en entrée un terme $H\ t$ et va transformer la variable existentielle $?G$ de la preuve du but courant en $let\ H' := H\ t\ in\ ?G'$. On obtient un morceau de certificat qui permettra au vérificateur de types de `Coq` de vérifier la preuve finale.

Si nous avions souhaité certifier cette transformation, il aurait fallu quantifier sur tous les contextes $C$ possibles pour $?G$ et vérifier que le passage de $C[let\ H' := H\ t\ in\ ?G']$ à $C[?G]$ est correct.

Par ailleurs, la robustesse du développement effectué pendant ce stage n'est pas du tout garantie : nous avons émis l'hypothèse qu'un petit changement dans la sémantique du langage source ou cible pouvait casser toutes les preuves, et ce de manière non triviale.

L'approche certifiante nous a parue beaucoup plus adaptée à nos objectifs, et nous avons donc choisi d'abandonner ce travail.

Cela n'exclut pas d'avoir des transformations certifiées un jour dans le plugin `Sniper`. Outre la satisfaction de s'assurer que la transformation ne présente aucun bug, cela permettrait de spécifier précisément le sous-ensemble de buts `Coq` que notre plugin est capable de prétraiter.

## 3.3 État de l'art

Dans cette section, nous allons mentionner d'autres outils d'automatisation pour `Coq` que `CoqHammer`, et expliciter si nécessaire les différences avec notre outil `Sniper`.

### 3.3.1 Le plugin `Tactician`

Le plugin `Tactician` [5] est une adaptation du plugin `TacticToe` [24] (écrit pour l'assistant de preuve HOL4), mais pour `Coq`.

L'idée de ces deux plugins est de se baser sur des algorithmes de *machine learning* pour deviner la prochaine tactique à appliquer et la suggérer à l'utilisateur·ice, étant donné un contexte local de preuve.

En compilant un fichier `Coq` avec `Tactician`, le plugin va constituer une base de données de paires ($\mathcal{S}_{before}$, *tac*), où $\mathcal{S}_{before}$ désigne l'état de la preuve avant l'application de *tac*. Cette base de données sert à l'apprentissage de l'algorithme. Il est également possible de nourrir cette base de données au fur et à mesure, et de commencer avec une base vide. Le plugin, une fois importé, tourne en arrière-plan d'un IDE pour `Coq`. Il va alors enregistrer les paires au fur et à mesure de l'exécution des tactiques, qui initialement doivent être écrites par l'utilisateur·ice.

Comme l'objectif est de prédire la tactique la plus pertinente à partir d'un nouveau but $\mathcal{S}_{new}$, encore jamais vu par l'algorithme, des caractéristiques (c'est-à-dire des sous-termes) vont être extraites du contexte de preuve $\mathcal{S}_{new}$, et comparées aux caractéristiques des états déjà présents dans la base de donnée. Les états les plus similaires, c'est-à-dire ceux dont la distance selon une mesure définie par les auteurs est moindre, sont associés à une tactique *tac* correspondante, qui sera suggérée par le plugin. La méthode utilisée est appelée « algorithme des $k$ plus proches voisins », et c'est avec celle-ci que les meilleurs résultats ont été obtenus par le plugin (d'autres algorithmes ont été essayés mais permettaient d'automatiser moins de preuves).

Le cœur du plugin `Tactician` propose deux tactiques cruciales :

— La tactique `suggest` qui suggère à l'utilisateur·ice la prochaine tactique à appliquer.

— La tactique `synth` qui essaie de compléter la preuve automatiquement (et qui, si elle réussit, affiche le script de preuve à l'utilisateur·ice).

Une évaluation de `Tactician` a été effectuée sur la bibliothèque standard, montrant que le plugin est capable, grâce à la tactique `synth`, de prouver automatiquement environ 40% des lemmes de la bibliothèque standard.

Nous n'avons pas effectué d'évaluation pour `Sniper`, mais nous sommes certain·es que le résultat sera moindre que `Tactician`. En effet, `suggest` ou `synth` peuvent très bien suggérer l'application de lemmes intermédiaires, là où `Sniper` va requérir de l'utilisateur·ice qu'il ou elle sélectionne ou prouve les lemmes nécessaires à sa preuve.

En revanche, les résultats obtenus par les tactiques de `Tactician` sont plus imprévisibles, puisque fondés sur des algorithmes de *machine learning*. Notre plugin `Sniper` vise une application moins large, mais une plus grande prédictibilité, ce qui, nous le pensons, est tout aussi important pour l'adoption d'une tactique par des utilisateur·ices.

### 3.3.2 Des prouveurs automatiques dans `Coq`

#### Le projet `Ergo`

Le prouveur SMT `Alt-Ergo` a été prouvé correct en `Coq` [29] (la partie SAT du prouveur, ainsi qu'une procédure de décision permettant de combiner la théorie de l'égalité avec une autre théorie arbitraire). Au lieu d'appeler un prouveur externe et de croire le résultat qu'il produit, ou de reconstruire la preuve à partir des certificats comme le fait `SMTCoq`, l'idée est de disposer d'un prouveur entièrement écrit et certifié en `Coq`. Une partie du code `OCaml` de `Alt-Ergo` a de ce fait été réécrite en `Coq` et vérifiée.

De plus, une tactique d'automatisation, `ergo`, qui combine théorie de l'égalité sur les symboles non interprétés avec la théorie de l'arithmétique linéaire et qui fait appel à ce prouveur certifié, a été implémentée.

À notre connaissance, ce code n'est malheureusement plus maintenu (le dernier *commit* du dépôt date d'il y a six ans). Par ailleurs, comme `SMTCoq` propose une automatisation similaire (avec un support pour plus de théories comme celle des vecteurs de bits ou des tableaux même

si nous n'en avons pas eu l'usage) et demeure compatible avec la dernière version de Coq, nous avons choisi SMTCoq et non ergo comme *backend*.

#### La tactique itauto

Une autre tactique, itauto, proposant d'appeler un prouveur SAT vérifié en Coq cette fois, a été implémentée plus récemment [4]. Grâce à ce travail, des techniques d'optimisation issues de l'implémentation de prouveurs SAT récents ont été formalisées et vérifiées en Coq, ce qui permet d'extraire un prouveur efficace.

Nous pensons que itauto pourrait être un *backend* intéressant pour Sniper, mais comme la tactique ne supporte que la partie propositionnelle du raisonnement (et donc pas les quantificateurs), cela nécessiterait d'implémenter des transformations d'instanciation des quantificateurs. Nous avons donc préféré nous concentrer sur un *backend* comme SMTCoq qui proposait déjà un support des quantificateurs.

## 3.4 Métaprogrammer en Coq

Pour écrire des transformations logiques, nous avons besoin d'accéder à la syntaxe des termes de Coq. En effet, au moins sur papier, les transformations sont définies récursivement sur la syntaxe des termes Coq.

Par exemple, si nous souhaitons éliminer les termes présentant du *pattern matching*, il nous faudra *a minima* une fonction capable de scanner un terme Coq et de répondre *true* s'il comporte la construction `match ... with ...` et *false* sinon.

Or, il n'est pas possible d'accéder directement dans Coq à la syntaxe des termes de Coq. Il nous faut un *métalangage* (qui peut être Coq lui-même). Ce métalangage nous donne accès à des fonctions qui manipulent les termes de Coq, comme des fonctions sur un type de donnée particulier. Autrement dit, les termes de Coq deviennent un type de donnée du métalangage.

Il existe deux fonctions fondamentales en métaprogrammation : la fonction de *réification*, qui prend en entrée un terme de Coq exprimé dans le langage-objet, donc en Coq, avec la syntaxe de l'utilisateur·ice, et qui renvoie l'arbre de syntaxe abstrait correspondant à ce terme dans le métalangage, et la *déréification*, qui fait l'opération inverse.

**Exemple 10** Logique propositionnelle et Coq

Par exemple, supposons que notre langage-objet soit la logique propositionnelle et que notre métalangage soit Coq.

En Coq, nous disposerons du type suivant, représentant les formules de la logique propositionnelle :

```
Inductive form : Set :=
  | atom : string -> form
  | and : form -> form -> form
  | or : form -> form -> form
  | not : form -> form
  | impl : form -> form -> form
  | top : form.
```

La réification de $((A \vee B) \wedge C) \to \neg\top$ sera le terme `Coq impl (and (or (atom "A")(atom "B"))(atom "C"))(not top)`.

La déréification du terme `or (atom "A")(impl top (not (atom "B")))` sera la formule $A \vee (\top \to \neg B)$

Le langage-objet, dans Sniper, est toujours Coq, et les métalangages seront respectivement : Ltac, Ltac2, MetaCoq (donc Coq), et elpi.

### 3.4.1 Choix des métalangages

Le choix des métalangages utilisés pour cette thèse repose sur une part de contingence et pas uniquement sur des raisons scientifiques (connaissance préalable des langages, maintenance, réactivité de la communauté, etc.). Par ailleurs, l'objectif était aussi l'expérimentation avec différents langages, même si certains n'ont pas été retenus.

Notamment, le langage Mtac2 [28] n'a pas été utilisé car :

— Sa communauté est plus réduite que celle des autres métalangages que nous avons utilisés, ce qui rend l'apprentissage du langage beaucoup plus ardue.

— Il ne propose pas de fonctions de réification / de déréification, ce qui rend difficile la construction de termes précis.

— Certaines tentatives d'installation via opam ont été faites, mais le plugin semble ne pas être accessible sur toutes les versions de Coq (en particulier, coq-8.17 sur lequel j'ai beaucoup travaillé). Or, au vu de toutes les dépendances de Sniper, nous avons besoin d'outils accessibles sur plusieurs versions de Coq.

Le langage Rtac [31] n'est simplement plus développé et a donc été écarté.

Le langage OCaml nécessite beaucoup plus de travail de port et de maintenance, car il utilise directement le noyau de Coq. Les métalangages récents permettent de passer facilement d'une version de Coq à une autre, et sont mieux documentés. Nous ne l'avons donc pas utilisé non plus.

Nous espérons avoir plus de stabilité en utilisant des métalangages plus « haut niveau ». En pratique, le passage de Sniper, initialement écrit dans la version 8.13 de Coq, a pu être effectué assez facilement jusqu'à 8.17 (les ports jusqu'à 8.19 ne devraient pas être plus difficiles mais nous n'avons pas pu nous en occuper avant le rendu de ce manuscrit).

### 3.4.2 Le premier langage de tactique : Ltac

#### Présentation

Le premier langage de tactiques pour Coq est appelé Ltac [18]. L'écriture de tactiques d'automatisation en OCaml s'avère coûteux en temps de développement, et nécessite d'utiliser une autre interface que celle de Coq. Le but de Ltac est donc d'offrir un langage avec suffisamment de puissance pour écrire des tactiques potentiellement non sûres, sans être limité par le vérificateur de types de Coq, mais avec un coût d'entrée bien moins grand et une maintenabilité plus aisée que lors de l'écriture de tactiques directement en OCaml. D'emblée il faut remarquer que Ltac est non typé, ce qui peut faire que beaucoup de bugs ne sont détectables qu'à l'exécution d'une tactique.

Ltac étend les combinateurs de tactiques de Coq, appelés *tacticals*, (par exemple le point virgule : tac1; tac2 applique tac1 puis tac2 à tous les sous-buts générés par tac2). Ainsi, il dispose d'un *pattern matching* superficiel, qui utilise donc la syntaxe de Coq, pour reconnaître des sous-termes dans le but les hypothèses, ou dans un terme Coq donné.

**Exemple 11** [Une tactique traitant les conjonctions et les disjonctions]
Un exemple est la tactique ci-dessous pouvant résoudre tous les buts Coq ne contenant que des conjonctions ou les disjonctions de propositions atomiques, dans le but ou les hypothèses.

```
Ltac auto_conj_disj :=
match goal with
| |- ?A /\ ?B => split ; auto_conj_disj
| |- ?A \/ ?B => left ; solve auto_conj_disj
| |- ?A \/ ?B => right ; auto_conj_disj
| H : ?A /\ ?B |- _ => destruct H ; auto_conj_disj
| H : ?A \/ ?B |- _ => destruct H ; auto_conj_disj
| _ => assumption
end.
```

On remarque que l'on peut matcher soit sur le but (à droite de `|-`, soit sur les hypothèses (à gauche)). Par ailleurs, la syntaxe `?A/\?B` permet d'utiliser à la fois directement le langage de `Coq` (et non pas sa syntaxe réifiée) et des métavariables existentielles, sur lesquelles on peut appliquer de nouvelles tactiques.

Par ailleurs, le mot-clef « `solve` » permet de forcer une tactique à résoudre le but. Autrement dit, `solve tac` échoue si `tac` ne résout pas le but, même si `tac` seule aurait réussi.

Une autre caractéristique importante de `Ltac` est le *backtracking*. Il est implicite dans l'exemple : on a utilisé un `match`, mot-clef qui le permet. En effet, si une branche du `match` échoue, les autres seront essayées. Cela peut être le cas dans l'exemple si on essaie `left` quand le but est une disjonction mais que la tactique ne parvient pas à prouver l'argument de gauche de cette disjonction. La tactique va alors revenir en arrière et essayer de prouver l'argument de droite.

`Ltac` dispose aussi d'un `lazymatch` qui échoue et n'essaie pas d'autres branches si le motif filtré correspond à une branche mais que les tactiques de la branche échouent.

L'échec permet donc dans certains cas de déclencher le *backtracking* ou est utilisé à la place d'une structure conditionnelle.

On utilise également la notion de succès/échec quand on utilise la construction `repeat`. La tactique donnée en argument de `repeat` sera appliquée en boucle, y compris aux sous buts générés, jusqu'à échouer ou ne faire aucun progrès. J'ai parfois eu recours au code suivant :

```
repeat match goal with
| H : _ |- _ => tac
end
```

Il permet de traiter successivement chaque hypothèse, car `Ltac` tentera d'appliquer `tac` à chacune d'entre elles et ciblera une nouvelle hypothèse en cas d'échec ou de but demeuré inchangé. En effet, il n'est pas possible d'écrire directement une fonction similaire à `List.map` sur les hypothèses.

Précisons également que ces `match` sont syntaxiques et non modulo conversion. Heureusement, `Ltac` admet des définitions locales, dans lesquelles on peut par exemple évaluer un terme `Coq` avec une stratégie au choix avant d'utiliser du *pattern matching*. Ainsi, `let x := eval cbv (f 2)in tac x` évalue `f 2` grâce à la stratégie *call-by-value* avant de lui appliquer la tactique `tac`.

### Le côté obscur de `Ltac`

Le nom de ce paragraphe est tiré d'un cours de Bruno Barras[1]. Il souligne bien le fait que la sémantique de `Ltac` est parfois ambiguë, à cause de stratégies d'évaluation complexes et de l'absence de typage. Ceci m'a causé beaucoup de difficultés lors d'implémentations de tactiques

1. https://coq.inria.fr/files/adt-30jun09-bruno-ltac.pdf

élaborées, mais m'a conduit aussi à écrire le répertoire d'astuces classiques qui, je l'espère, pourra servir à quiconque se lance dans la métaprogrammation avec ce langage.

Certes, `Ltac` est très rapide à prendre en main (pour ses fonctionnalités haut niveau) comparé à d'autres métalangages, c'est le métalangage le plus fréquemment utilisé quand on veut rajouter un peu d'automatisation *ad hoc* dans son code, mais il devient beaucoup moins adapté dans des cas plus complexes.

En particulier, la *combinaison* de tactiques y est très difficile, et faire des opérations bas niveau sur des termes de `Coq` réifiés y est complètement impossible. Pour le second point, nous avons d'autres métalangages qui permettent une « chirurgie » sur les termes qui se combineront très bien avec `Ltac` pour le travail de plus haut niveau.

Mais pourquoi la combinaison de tactiques est-elle si difficile ?

Pour reprendre le cours de Bruno Barras mentionné ci-dessus, « une tactique ne peut pas à la fois retourner une valeur et agir sur le but ». Or, dans `Sniper`, nous voulons faire précisément cela : nous avons besoin de transformer ou de rajouter des hypothèses mais aussi de savoir quelles hypothèses nous avons rajoutées, donc de les renvoyer pour appeler une nouvelle tactique.

Un autre problème majeur qui y est lié est *l'ordre d'évaluation* de `Ltac`. Il faut ainsi différencier dans ce langage les termes qui seront considérés comme des *expressions* et les termes considérés comme des *tactiques* proprement dites, car l'évaluation des premiers se fera toujours avant celle des deuxièmes.

**Exemple 12** [Retourner un terme nouvellement créé]

Considérons la tactique suivante :

```
Ltac intro_return_id := let x := fresh in intro x ; x.
```

A priori, cette tactique crée un identifiant frais, l'introduit (donc agit sur le but) et le renvoie.

Mais tentons de l'utiliser sur un but trivial :

```
Goal True -> True.
let y := intro_return_id in exact y.
(* message d'erreur: variable y should be bound to a term *)
```

C'est peu clair, tentons d'afficher ce que vaut `y` *via* la tactique `idtac`. Nous obtenons alors le message suivant : `<tactic closure>`.

Il faut savoir que lorsque ce message apparaît, c'est qu'il y a un souci d'ordre d'évaluation de `Ltac`. Ici, la définition locale `let x := fresh in intro x` n'est pas entièrement évaluée dans la première phase car elle contient un terme considéré comme une tactique (`intro x`), dont l'évaluation (et l'exécution) doit se faire dans un second temps. Elle prend donc la valeur indéterminée `<tactic closure>` le temps d'atteindre la seconde phase d'évaluation. Mais la deuxième définition locale `let y := intro_return_id` s'évalue dans la première phase aussi : `y` vaudra donc également `<tactic closure>`, d'où la présence d'un message d'erreur lorsque l'on tente de recourir à la tactique `exact`.

Ainsi, lorsque c'est possible, il vaut mieux distinguer les tactiques dont le but est de produire une *expression*, et les tactiques qui font un *effet de bord* (autrement dit, qui agissent sur le ou les buts `Coq` en cours de preuve), et être conscient qu'une tactique faisant les deux à la fois causera des difficultés d'implémentation.

Une autre difficulté de `Ltac` est liée aux quantificateurs. Nous allons illustrer ceci grâce à la syntaxe `match ... context`. Elle permet de reconnaître un sous-terme au sein d'un terme donné. Voici un exemple tiré du code de `Sniper` :

```
match goal with
| |- context C[match ?expr with _ => _ end] => ...
```

Ici, quand le but est de la forme $\forall(\vec{x} : \vec{A}), C[\texttt{match } f\ \vec{x} \texttt{ with } \ldots]$, le `match` ne sera pas détecté, parce que ce n'est pas un sous-terme clos. De même, dans le code, la variable `expr` peut aussi ne pas être détectée pour la même raison (il s'agit rarement d'un sous-terme clos dans les cas qui nous intéressent).

Si l'on tente une récursion naïve pour passer sous les quantificateurs, le code produira une erreur :

**Exemple 13** [Passage sous les quantificateurs]

```
Ltac detect_match_naive :=
let rec loop G :=
match G with
| context C[match ?expr with _ => _ end] => idtac expr
| forall (A : _), ?G' => loop G'
end
in
match goal with
| |- ?G => loop G
end.

Goal (forall (n : nat), pred n = match n with 0 => 0 | S n' => n' end)
    .
detect_match_naive.
(* message d'erreur : Must evaluate to a closed term
offending expression:
G
this is an object of type
constr_under_binders *)
```

Nous verrons qu'une série de techniques existent pour pallier ce problème également.

Un tout dernier problème est qu'il n'est pas facile de transmettre une information d'un sous-but à un autre. Nous verrons comment le problème se pose et est résolu dans le sous-paragraphe suivant.

### Astuces classiques

Je tiens à recueillir ces astuces de métaprogrammation en `Ltac` parce qu'elles sont, pour la plupart d'entre elles, très peu intuitives, très peu documentées (à moins de faire les bonnes recherches sur certains forums `Coq` ou notes de blogs), et que je pense pourtant qu'elles sont absolument cruciales à quiconque souhaite se servir des fonctionnalités avancées de ce langage.

Par ailleurs, celles-ci sont utilisées dans le code de `Sniper` et ont donc été indispensables lors de l'implémentation de ce plugin.

**L'usage de continuations** La première astuce pour contourner le problème des tactiques qui ne peuvent pas agir sur le but et renvoyer une valeur est d'écrire les tactiques avec des *continuations*, c'est-à-dire rajouter à toutes les tactiques qui produisent une expression un nouvel argument qui sera une autre tactique prenant en entrée le résultat de la tactique précédente. On appelle cette manière de coder le style CPS (*continuation passing style*).

**Exemple 14** [Suite de l'exemple en CPS]
Réécrivons la tactique en CPS :

```
Ltac intro_return_id_cps := fun k => let x := fresh in intro x ; k x.
```

Cette fois, le but peut être résolu :

```
Goal True -> True.
intro_return_id_cps ltac:(fun x => exact x). Qed.
```

Le code devient vite assez lourd : pour aider l'analyseur grammatical de `Ltac`, il faut préciser la nature du terme donné en argument à `intro_return_id_cps`, et ne pas oublier d'$\eta$-expanser la tactique `exact` en introduisant un lieur.

Quand les continuations commencent à se chaîner, ou que la continuation en question dépend de la branche d'un `match` qui s'est exécuté par exemple, la lisibilité du code en est vraiment affectée, si bien que d'autres solutions ont été imaginées.

**Forcer l'évaluation des tactiques** Une autre manière de traiter le problème, et qui rend la syntaxe plus lisible de mon point de vue est de contraindre les tactiques à être évaluées dans la première phase, comme les expressions.

Or, une construction ambiguë en `Ltac`, qui peut à la fois retourner un terme `Coq` ou exécuter une tactique est le `match ... with`. Encapsuler le `match ... with` dans un `let ... in` fait que son contenu sera interprété comme une *expression*, et les tactiques qui devraient normalement s'exécuter dans la deuxième phase d'évaluation sont forcées de s'évaluer dans la première phase[2].

**Exemple 15** [Suite de l'exemple en forçant l'évaluation]

```
Ltac intro_return_id_force :=
let x := fresh in
let _ := match goal with _ => intro x end in
x.
```

Ici, l'exécution de `intro x` n'est pas retardée dans la phase d'évaluation des tactiques, car nous avons forcé son exécution dans la phase d'évaluation des expressions.

Le but peut être résolu grâce à cette réécriture de la tactique :

```
Goal True -> True.
let x := intro_return_id_force in exact x. Qed.
```

Attention tout de même, si jamais, dans le code suivant : `let _ := match goal with _ => tac`, la tactique `tac` crée des sous-buts, la tactique principale va s'appliquer à tous les sous-buts.

Par exemple :

```
Ltac split_return_id_force :=
let x := fresh in
let _ := match goal with _ => intro x ; split end in
x.

Goal True -> (True /\ (True -> True)).
let x := split_return_id_force in ltac:(exact x).
```

2. Une description du fonctionnement des phases d'évaluation en `Ltac` est décrite dans les commentaires d'une *issue* du dépôt de `Coq` : https://github.com/coq/coq/issues/4732#issuecomment-337542429.

```
(* message d'erreur : In environment
H : True
The term "H" has type "True"
while it is expected to have type
 "True -> True". *)
```

Il vaut mieux donc réserver cette astuce pour forcer l'exécution de tactiques qui ne s'appliquent qu'à un seul but et n'en créent pas de nouveaux.

**L'encodage ou l'ajout dans le but** Cette astuce permet de résoudre à la fois le problème des tactiques produisant des valeurs et faisant des effets de bord, et le passage sous des quantificateurs.

Pour retourner une valeur, il suffit de la poser dans le but en cours. Ensuite, du *pattern matching* sur le but permet de retrouver l'hypothèse posée (c'est la première qui sera examinée).

**Exemple 16** [Suite de l'exemple : ajout dans le but]

```
Ltac return_id_in_goal :=
let x := fresh in intro x ;
let H := fresh in pose (H := x).

Goal True -> True.
return_id_in_goal;
match goal with
| H := ?x : _ |- _ => exact x
end. Qed.
```

L'important est de remarquer que les hypothèses sont filtrées de la dernière introduite à la première, ce qui permet de retrouver facilement la dernière introduite par une tactique (et donc la valeur d'intérêt). Cependant, cette technique alourdit le but.

Pour passer sous les quantificateurs d'un terme donné, l'idée est d'avoir un but qui est précisément le terme à considérer. Mais comme ce but n'est pas à prouver, il faut créer un but trivial permettant tout de même l'analyse du terme. Ainsi, si le terme est `T`, le sous-but créé sera de type `False -> T`. Après une introduction du faux (mettons, `Hfalse: False`), il est possible d'écrire des tactiques examinant `T` en introduisant récursivement les quantificateurs. Il n'y a donc plus de problème de variables libres. Une fois l'exécution des tactiques terminée, `destruct Hfalse` permet de résoudre ce sous-but. Cependant, comme `Ltac` ne permet pas de communiquer d'un but à l'autre, il faut utiliser une nouvelle astuce pour pouvoir transmettre le terme ou l'information obtenue depuis le sous-but trivial aux autres buts.

Pour éclaircir nos propos, nous allons continuer l'exemple 13 de la détection d'un `match`.

**Exemple 17** [Passage sous les quantificateurs]

Réécrivons la fonction `detect_match` avec cette nouvelle technique :

```
Ltac detect_match_less_naive :=
let rec loop :=
match goal with
| |- context C[match ?expr with _ => _ end] => let T :=
type of expr in idtac expr T
| |- forall (A : _), _ => let H := fresh in intro H ; loop
end in
lazymatch goal with
```

```
| |- ?G => let H := fresh in assert (H : False -> G) by
(let Hfalse := fresh in intro Hfalse;
loop ; elim Hfalse)
end.

Goal (forall (n : nat), pred n = match n with 0 => 0 | S n' => n' end)
    .
detect_match_less_naive.
(* affiche: H nat *)
```

Comme nous voulons détecter sur quelle expression s'applique le `match`, mais sans altérer le but, nous ne voulons pas introduire les quantificateurs dans le but principal. Un sous-but est donc créé, sur lequel on s'applique la sous-tactique `loop`. Á la fin de l'exécution de celle-ci, ce sous-but est prouvé.

Cependant, ici, nous ne faisons qu'*afficher* les informations. Comment faire si nous souhaitons, par exemple, renvoyer le type `T` de l'expression `expr` filtrée au but principal ? En effet, la portée de `T` dans ce code ne peut aller au-delà des parenthèses après le `by` du `assert by`. Et si on décide de plutôt poser `T` dans le but, il ne le sera que sans le sous-but trivial et demeurera inaccessible au but principal.

**Les *evars*** En `Coq`, la tactique `epose` permet de rajouter des variables existentielles au contexte local. Par exemple, `epose (x := ?[x_evar] : ?[T])` pose une variable notée `x` dont le corps est pour le moment représenté par la variable existentielle `x_evar` et dont le type est également une variable existentielle `T`. Il est possible de préciser un type avec `epose`, au lieu de le laisser indéterminé. Exécuter ensuite `instantiate (x_evar := 1)` permet de remplacer `x_evar` par `1` et le type `nat` est automatiquement inféré (et remplace `T`).

Maintenant, comment utiliser ces *evars* pour transmettre une information d'un but à un autre ? La solution est de créer une *evar* dans le but principal, et de l'instancier par une valeur calculée dans le sous-but trivial (qui ne sert, pour rappel, qu'à passer sous des quantificateurs dans notre exemple).

Le code final, qui renvoie bien le type de la variable filtrée au but initial, est le suivant :

**Exemple 18** [Passage sous les quantificateurs et retour de valeur au but principal] Précisons que l'on profite de cette version de la tactique pour nettoyer le but d'hypothèses inutiles.

```
Ltac detect_match :=
let e_evar := fresh in
let rec loop :=
match goal with
| |- context C[match ?expr with _ => _ end] =>
  let T := type of expr in
  instantiate (e_evar := T)
| |- forall (A : _), _ =>
  let H := fresh in intro H ; loop
end
in
let e := fresh in
epose (e := ?[e_evar] : Type) ;
lazymatch goal with
```

```
| |- ?G => let H := fresh in assert (H : False -> G) by
(let Hfalse := fresh in intro Hfalse;
loop ; elim Hfalse) ; clear H ;
let e' := eval unfold e in e in idtac e'
end.

Goal (forall (n : nat), pred n = match n with 0 => 0 | S n' => n' end)
    .
detect_match.
(* affiche nat *)
```

Dans `Sniper`, la combinaison de la technique de l'encodage dans le but et de l'utilisation d'*evars* pour faire communiquer les buts entre eux a été utilisée en 4.1.5.

**La fonction de contexte** Une dernière méthode, qui n'a pas été utilisée dans `Sniper` mais que je tiens à mentionner par souci d'exhaustivité, peut remplacer l'usage des deux dernières astuces. Elle est expliquée par Adam Chlipala au chapitre 15, paragraphe 5, de son livre sur `Coq` [11].

En `Ltac`, le code suivant fera que la variable `?z` ne matche que d'éventuels sous-termes qui ne seront pas introduits par le filtrage.

```
match x with
| fun (y: nat) => ?z
...
```

Autrement dit, `?z` ne peut pas mentionner `y`. Mais on peut utiliser une construction plus générale, `?@z`, qui permet de mentionner des variables nouvellement introduites.

Ainsi, on peut réécrire le code de `detect_match` avec l'aide de deux fonctions auxiliaires permettant respectivement de retourner le tuple des variables nouvellement introduites et de requantifier sur un tuple de variables présentes dans le contexte local :

```
Ltac intro_and_return_tuple n t :=
match n with
| 0 => t
| S ?n' =>
  let H := fresh in
  let _ := match goal with _ => intro H end in
  intro_and_return_tuple n' (H, t)
end.

Ltac revert_tuple t :=
match t with
| (?x, ?y) => revert x ; revert_tuple y
| _ => idtac
end.
```

L'idée de la fonction principale `detect_match_context_fun` est de compter le nombre de quantificateurs (universels) à introduire. La fonction `body` utilise les variables précédemment abstraites, ainsi que la variable sous le dernier `forall`. On fait la récursion sur une fonction qui prend en entrée un $n$-uplet $g$ (les $n$ variables rencontrées jusqu'ici), la première projection de $g$ contenant les $n-1$ premières variables et la seconde projection contenant la dernière variable introduite.

Malheureusement, cette astuce seule ne suffit pas, car le `match` de `Ltac` ne permet pas de combiner le motif `fun` et la syntaxe `context`. Pour détecter le *pattern matching*, nous avons besoin d'introduire les quantificateurs. Autrement, nous aurions simplement pu arrêter la récursion sous les lieurs lorsque la forme `fun g => context [match ?expr with _ => _ end]` est détectée, et afficher le type de `expr` comme voulu. Mais ici, nous devons d'abord retourner le nombre de quantificateurs à introduire, les introduire effectivement, et ensuite chercher le `match` et le type de `expr`, pour finalement requantifier sur les variables introduites afin de laisser le but inchangé.

```
Ltac detect_match_context_fun :=
let rec loop G n :=
lazymatch G with
| fun g => forall x, @?body g x => loop ltac:(eval cbv beta in (fun g => body (
    fst g) (snd g))) (S n)
| _ => n
end
in match goal with
|- ?G =>
  let n := loop (fun _ : unit => G) 0 in
  let tp := intro_and_return_tuple n unit in
  match goal with
  | |- context [match ?expr with _ => _ end] => let T := type of expr in idtac
      T
  end ; revert_tuple tp
end.

Goal (forall (n : nat), pred n = match n with 0 => 0 | S n' => n' end).
detect_match_context_fun.
(* affiche: nat *)
```

Une autre version plus concise n'utilise pas la syntaxe `context`, mais il faut alors gérer différents cas, selon où peut se trouver le *pattern matching* dans le sous-terme qui nous intéresse, à la main :

```
Ltac detect_match_context_fun' :=
let rec loop G :=
lazymatch G with
| fun g => forall x, @?body g x => loop ltac:(eval cbv beta in (fun g => body (
    fst g) (snd g)))
| fun (g : ?T) => ?f match @?expr g with _ => _ end =>
let e := fresh in
let e_evar := fresh in epose (e := ?[e_evar ] : T) ; let t := type of (expr e)
in idtac t
| fun (g : ?T) => match @?expr g with _ => _ end =>
let e := fresh in
let e_evar := fresh in epose (e := ?[e_evar ] : T) ; let t := type of (expr e)
in idtac t ; clear e
end
in match goal with
|- ?G => loop (fun _ : unit => G)
end.
```

Nous avons fini de présenter Ltac et les différentes astuces que nous avons utilisées. Ce travail de regroupement d'exemples et de documentation a pour objectif d'expliquer à la fois le code de Sniper, et de fournir grâce à des exemples que nous espérons clairs les limitations apparentes de Ltac et leurs éventuels contournements.

Comme ce métalangage n'est pas suffisant en raison de ses limitations et de sa difficulté d'utilisation, d'autres sont apparus, et nous présenterons d'abord celui qui ressemble le plus à Ltac.

### 3.4.3 Une extension plus robuste : Ltac2

Le langage Ltac2 [38] a pour but de remplacer Ltac et donc de pallier ses faiblesses, tout en maintenant la rétrocompatibilité avec les développements en Coq utilisant du Ltac.

C'est un langage à la ML (*Meta Language*), comprenant : la possibilité de définir ses propres types algébriques, du polymorphisme prénexe, avec du typage statique (c'est-à-dire, du typage à la compilation et non à l'exécution comme en Ltac). Le typage statique permet de distinguer clairement les tactiques Ltac2 produisant des effets de bords de celles produisant des valeurs, et de facilement mélanger les deux, ce qui était source de nombreuses difficultés en Ltac. Tout simplement, les tactiques ne faisant que des effets de bords (qui affichent un message, modifient le contexte de la preuve, etc.) sont de type `unit -> unit`. Faute de traduction satisfaisante, toutes les fonctions prenant un argument de type `unit` seront appelées des *thunks*.

L'utilisateur·ice peut également étendre le type des exceptions Ltac2 de manière à définir les siennes. Ceci permet d'avoir un mécanisme de *backtracking* plus prévisible que celui de Ltac.

Deux fonctions cruciales sont disponibles en Ltac2, dans le module `Control` dédié à la gestion du contexte de preuve : `hyps : unit ->(ident * constr option * constr)list` et `goal : unit -> constr`. La première renvoie les hypothèses, sous la forme d'un triplet : identifiant (= nom de l'hypothèse), corps dans le cas où l'hypothèse est une définition locale, et type. Par exemple, `n: nat` sera représenté par `(ident:(n), None, constr:(nat))` et `f := @id nat : nat -> nat` par `(ident:(f), Some constr:(@id nat), constr:(nat -> nat))`. Nous avons introduit les annotations pour l'analyseur syntaxique `constr:(...)` et `ident:(...)` qui sont souvent utilisées en Ltac2.

La fonction `hyps` va de pair avec la fonction `hyp : ident -> constr` qui permet de renvoyer le terme Coq correspondant à un identifiant donné, au sein du but courant. Ainsi, `hyp ident:(n)` renvoie le `constr` correspondant, `n`.

Par ailleurs, la fonction `goal` renvoie le but à prouver. Ces fonctions, pourtant fondamentales, n'existent pas en Ltac. Pour récupérer toutes les hypothèses et ensuite y appliquer la même fonction, l'utilisateur·ice de Ltac doit user d'astuce, en marquant les hypothèses déjà traitées avec un tag (la fonction `id` par exemple).

**Exemple 19** Comparaison Ltac et Ltac2

Supposons que nous souhaitions dupliquer toutes nos hypothèses. Nous allons développer un code Ltac et un code Ltac2 permettant de le faire, afin de souligner l'avantage net de Ltac2 dans ce cas précis, grâce à la primitive `hyps`.

```
Ltac duplicate_hypotheses_aux :=
  repeat match goal with
    | H : ?T |- _ =>
      lazymatch T with
        | id _ => fail
        | _ =>
          let H0 := fresh in
```

```
        pose proof (H0 := H : id T) ;
        change T with (id T) in H
    end
  end.

Ltac remove_id :=
  repeat match goal with
    | H : id ?T |- _ => change (id T) with T in H
  end.

Ltac duplicate_hypotheses :=
  duplicate_hypotheses_aux; remove_id.

Ltac2 duplicate_hypotheses () :=
  let hs := Control.hyps () in
  let rec aux hs :=
    match hs with
    | (id, _, _) :: hs' =>
      ltac1:(H |- let H0 := fresh in pose proof (H0 := H)) (Ltac1.
          of_constr (hyp id)) ; aux hs'
    | _ => ()
    end
  in aux hs.
```

Commentons ce code : en `Ltac`, la construction `repeat match` cherche le premier motif de filtrage qui correspond à une des branches possibles, et ne va tenter un autre motif qu'en cas d'échec ou en cas de non-progrès (c'est-à-dire si le contexte local ne change pas à la fin de l'exécution de la branche).

Rappelons bien aussi que le `match` de `Ltac` essaie toutes les branches du *pattern matching*, tandis que le `lazymatch` ne tentera que la première branche où le motif correspond au terme filtré.

Ainsi, pour ne pas dupliquer à l'infini la première hypothèse, il faut trouver une condition d'arrêt qui soit compatible avec la sémantique du `repeat`. C'est ici qu'intervient l'astuce. L'hypothèse `T` dupliquée est changée en une hypothèse convertible, `id T`. Or, quand une hypothèse de la forme `id T` est filtrée, la tactique échoue, provoquant un *backtracking* permettant de passer à la prochaine hypothèse.

Trois remarques maintenant : (1) ce n'est absolument pas naturel d'implémenter les choses ainsi. On voudrait disposer d'une primitive dans le langage qui permet de collecter les hypothèses, et de pouvoir itérer dessus sans avoir à imaginer des astuces, qui, de surcroît, (2) ne permettent pas d'être complet·e. En effet, cette astuce ne permet pas de dupliquer les hypothèses commençant par `id`. Enfin, (3) l'implémentation fait deux passes sur les hypothèses (parce qu'il faut enlever les tags `id`), alors qu'on aurait voulu en faire qu'une seule.

Or, `Ltac2` résout *tous* ces problèmes !

L'implémentation de la fonction qui duplique les hypothèses, dans ce langage est assez proche de ce qu'on aurait pu écrire en `OCaml` : nous y retrouvons l'usage de deux des primitives susmentionnées et une fonction qui itère sur les hypothèses.

Comme certaines des tactiques de `Ltac1` ne sont pas disponibles en `Ltac2`, comme `pose proof`, nous avons recours a la notation `ltac1:(x1 ... xn |- )`. Les `xi` sont

les arguments de la tactique `Ltac`. Pour l'appeler via `Ltac2`, il faut lui donner des arguments de type `Ltac1.t`. C'est pour cette raison que nous convertissons le `constr` obtenu via la fonction `hyp` en arguments (dynamiquement typés) pour les tactiques `Ltac1`. (C'est aussi un prétexte pour montrer cette syntaxe extrêmement utile en `Ltac2`, qui permet d'assurer une forte rétrocompatibilité.)

Pour résumer, le tableau suivant montre toute l'utilité de `Ltac2` par rapport à `Ltac` dans cet exemple :

| Langage | Efficacité | Complétude | Facilité d'implémentation |
|---|---|---|---|
| `Ltac` | ☹ | ☹ | ☹ |
| `Ltac2` | ☺ | ☺ | ☺ |

En `Ltac2`, certaines caractéristiques plaisantes de `Ltac` sont toujours disponibles : pouvoir faire du *pattern matching* sur des termes `Coq` en usant d'une représentation superficielle en particulier. Mais, dans le même temps, `Ltac2` expose le type `constr`, qui donne accès à la représentation des termes `Coq` dans le noyau, et qui permet donc une manipulation plus bas-niveau.

### 3.4.4 Manipuler des termes `Coq` en `Coq` : `MetaCoq`

Le projet `MetaCoq` [39] est né de la volonté de produire un environnement de métaprogrammation *certifié* pour `Coq`. En effet, puisque `MetaCoq` décrit la syntaxe et la sémantique des termes de `Coq` au sein même de `Coq`, il est possible d'écrire des fonctions `MetaCoq` prouvées correctes. Ainsi, `MetaCoq` implémente un type inductif `term` dont les constructeurs correspondent à ceux du type `OCaml` représentant les termes de `Coq`.

Par ailleurs, `MetaCoq` propose de nouvelles tactiques et commandes permettant d'exécuter des opérations (des effets de bords) décrites dans un type inductif appelé `TemplateMonad`. Ces opérations sont celles permettant de réifier un terme ou de le déréifier, d'afficher une valeur, de renvoyer une erreur... Pour effectuer celles-ci dans un script de preuve `Coq`, il faut utiliser la tactique `run_template_program` suivie de l'opération désirée et d'une continuation qui prendra en argument la valeur de retour de cette opération. Par exemple, pour réifier un terme `I` et l'environnement associé (voir 3.4.4 pour une description de l'environnement) et lui appliquer `tac`, on utilise `run_template_program (tmQuoteRec I)tac`.

De surcroît, dans `MetaCoq` sont démontrées de nombreuses propriétés métathéoriques de `Coq`, mais c'est un immense aspect de ce plugin dont nous n'avons pas eu besoin, le but final étant de certifier l'extraction de fonctions `Coq` en `OCaml`. Ainsi, plus précisément, `MetaCoq` se divise en PCUIC (*Predicative Calculus of Inductive Constructions*) où sont démontrées des propriétés métathéoriques de `Coq` (par exemple la preuve de correction du vérificateur de types de `Coq` [40]), et en `Template-Coq`, utilisé pour la métaprogrammation. Notre outil `Sniper` ne dépend que de `Template-Coq`.

Nous nous concentrerons donc dans cette modeste présentation de `MetaCoq` sur son aspect « métaprogrammation », c'est-à-dire sur le type `term` des termes `Coq` et sur l'utilisation de la `TemplateMonad` pour la réalisation de plugins.

Dans `Sniper`, nous avions d'abord envisagé d'utiliser `MetaCoq` pour écrire des transformations *certifiées* : c'est en effet l'outil parfait pour cet objectif. Mais, ayant abandonné cette idée (du moins, pour cette thèse), nous avons toutefois continué de travailler avec `MetaCoq` car ce n'est pas son seul avantage.

Tout d'abord, de façon très pragmatique, nous avions fini par acquérir une certaine familiarité avec cet outil, ce qui rendait nos implémentations bien plus aisées. Ensuite, utiliser `MetaCoq`

permet de rester dans Coq pour développer nos transformations, ce qui évite d'apprendre un nouveau langage ou de changer d'environnement de développement. Enfin, rien ne dit qu'à l'avenir, Sniper ne puisse pas compter des transformations certifiées, et MetaCoq deviendra alors quasiment indispensable (il nous faudra nécessairement une représentation Coq des termes Coq, sans cela, aucune preuve formelle ne pourra être écrite).

Enfin, MetaCoq donne facilement accès à la représentation interne des termes de Coq. Le type term de Template-Coq est l'exacte contrepartie du type constr en OCaml. Nous avons trouvé que la manipulation des termes était plus aisée en MetaCoq que dans les autres métalangages que nous avons utilisés :

1. Ltac ne donne accès qu'à une représentation superficielle des termes de Coq, et nous avons vu combien il était difficile d'écrire des programmes Ltac traversant des lieurs.
2. Ltac2 donne certes accès au type kind qui se calque sur le type OCaml des termes de Coq, mais, dans les versions anciennes de Coq que j'ai utilisées au début de ma thèse (jusqu'à 8.16), les lieurs ne peuvent pas mentionner de variables libres, ce qui contraint un peu plus le développement de plugins.
3. Le langage Elpi que nous présenterons ci-après utilise une représentation complètement différente des variables, ce qui est parfois un avantage, mais ce qui peut également complexifier l'implémentation.

### Les rudiments de MetaCoq

Présentons une partie du type term de template-coq :

```
Inductive term : Type :=
    tRel : nat -> term (* indices de De Bruijn *)
  | tVar : ident -> term (* variables de section, de contexte... *)
  | tProd : aname -> term -> term -> term (* produits *)
  | tSort : Universe.t -> term (* univers *)
  | tLambda : aname -> term -> term -> term (* lambda abstractions *)
  | tApp : term -> list term -> term (* applications *)
  | tInd : inductive -> Instance.t -> term (* inductifs *)
  | tConstruct : inductive -> nat -> Instance.t -> term (* constructeurs *)
  | tCase : case_info ->
          predicate term -> term -> list (branch term) -> term (* pattern
              matching *)
  ...
.
```

Il existe un constructeur du type term pour chaque forme syntaxique possible de termes de Coq. Les indices de De Bruijn ne prennent qu'un entier en argument (le nombre de lieurs à traverser pour retrouver celui qui les concerne). Dans le noyau de Coq, les indices commencent à 1, mais comme cette convention est peu courante, en MetaCoq, les indices commencent à 0.

Le constructeur tVar concerne les variables nommées (variables de section ou variables locales). Le constructeur tProd est celui des produits dépendants, il prend en argument une annotation pour le lieur, le type de la variable qu'il lie et un terme. Le constructeur tLambda est similaire, mais pour les fonctions. Le constructeur tSort concerne les univers que nous ne détaillerons pas. Notons simplement qu'il existe un terme de type Universe.t pour *Prop* et un pour *Set*. L'application, tApp, prend en premier argument la fonction et en second argument la liste de termes à laquelle elle est appliquée. Avoir des applications $n$-aires et non binaires permet de simplifier grandement l'écriture des plugins. Les constructeurs tInd et tConstruct ont pour

premier argument un `inductive` (qui contient un chemin absolu vers la déclaration d'inductif correspondante dans l'environnement global, et un entier qui est le numéro de l'inductif dans un bloc d'inductifs mutuels). Le terme de type `Instance.t` permet de gérer le polymorphisme d'univers et nous n'en parlerons pas. Le terme `tConstruct` attend aussi un entier naturel en argument, qui est le numéro du constructeur considéré. Le terme `tCase` est un *pattern matching*, qui prend en argument, entre autres, la variable sur laquelle on matche, un type de retour (éventuellement dépendant), et une fonction par branche (représentée par le type `branch term`).

**Exemple 20** Fonction identité dans `nat`

Dans ce type inductif `term`, la fonction identité dans les entiers naturels réifiée s'écrirait ainsi :

```
tLambda
{| binder_name := nNamed "x"; binder_relevance := Relevant |}
(tInd {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"] "nat")
    ;
         inductive_ind := 0 |} []) (tRel 0)
```

Le constructeur de tête est bien celui des fonctions, `tLambda`. Son premier argument de type `aname` comporte une annotation pour le *pretty-printer*, qui permet d'afficher le nom des variables libres dans le corps de la fonction. Le champ noté `binder_relevance` concerne un type particulier, appelé *SProp* [25], qui ne nous intéresse pas dans `Sniper`.

Tous nos lieurs seront toujours annotés comme `Relevant`. Remarquons qu'il serait tout à fait possible de se passer de ces annotations sur les lieurs, et donc de tous les arguments de type `aname` parmi les constructeurs du type `term`. En effet, ce qui compte en `MetaCoq`, c'est l'indice de De Bruijn de la variable liée. Nous avons parlé de ceux-ci plus haut, pour rappel, voir 2.1.3.

Le deuxième argument de `tLambda` est le type de l'argument de la fonction, donc ici, l'inductif `nat`. Nous avons effectivement un constructeur `tInd` appliqué au chemin absolu du fichier définissant `nat`, et aux instances d'univers. Elles sont vides, d'où la présence d'une liste vide comme argument de `tInd`, puisque `nat` n'utilise aucune variable de type.

Enfin, le troisième et dernier argument de `tLambda` est la variable d'indice de De Bruijn 0, c'est-à-dire la dernière variable liée.

Nous pouvons remarquer que le constructeur `tInd`, outre les informations d'univers sur lesquelles nous ne nous attarderons pas, ne permettent pas de savoir les caractéristiques de l'inductif qu'il représente. En effet, ce informations sont stockées dans l'environnement global.

Celui-ci est constitué de déclarations d'univers et de contraintes le cas échéant, mais aussi des constantes (comme les définitions de fonctions) et des inductifs qui ont été déclarés dans `Coq`.

Ainsi, il est possible de réifier l'environnement global suffisant à définir `foo`, grâce à la commande `MetaCoq Quote Recursively foo`, et `MetaCoq` le fait intelligemment, de façon à ne pas afficher les inductifs ou les constantes qui ne sont pas mentionnés par `foo`.

**Exemple 21** Réification de l'environnement global pour `nat`

Après la réification de `nat` et de son environnement global, toutes les informations sur cet inductif sont accessibles dans `MetaCoq`. Notamment, nous trouvons la déclaration de l'inductif `nat` (heureusement !).

Une déclaration d'inductif, dans l'environnement global, est représenté comme un enregistrement donc chaque champ contient des informations dont le ou la métraprogrammeur·euse aura besoin.

Pour présenter l'environnement et les déclarations d'inductifs avons décidé de retirer des champs qui complexifieraient nos explications car ils ne servent pas à cette thèse. Voici la représentation réifiée de la déclaration de l'inductif `nat`.

```
({|
universes := ([Level.lzero], ConstraintSet.empty);
declarations :=
  [(MPfile ["Datatypes"; "Init"; "Coq"], "nat",
  InductiveDecl
    {|
    ind_finite := Finite; (* L'inductif est-il un point fixe ou un
        co-point fixe ? *)
    ind_npars := 0; (* le nombre de paramètres *)
    ind_params := []; (* leur nom et leur type *)
    ind_bodies := (* la liste de tous les blocs inductifs mutuels *)
      [{|
       ind_name := "nat"; (* le nom de l'inductif *)
       ind_indices := []; (* la liste de toutes ses annotations *)
       ind_type := tSort (Universe.of_levels (inr Level.lzero)); (*
           le type de l'inductif *)
       ind_ctors := (* la liste de ses constructeurs *)
         [{|
          cstr_name := "O";
          cstr_args := [];
          cstr_indices := [];
          cstr_type := tRel 0;
          cstr_arity := 0
         |};
         {|
          cstr_name := "S";
          cstr_args :=
          [{|
             decl_name := {| binder_name := nAnon; binder_relevance
                 := Relevant |};
             decl_body := None;
             decl_type := tRel 0
           |}];
          cstr_indices := [];
          cstr_type :=
          tProd {| binder_name := nAnon; binder_relevance := Relevant
               |}
            (tRel 0) (tRel 1);
          cstr_arity := 1
         |}];
        |}];
    |})];
|}
```

Nous avons déjà indiqué en commentaire quelques détails sur cet enregistrement quand ils étaient suffisamment concis pour ne pas encombrer le code, mais voici

quelques indications supplémentaires :

— Le champ `universes` précise que cet inductif vit dans *Set* (qui est aussi $Type_0$, d'où la mention de `Level.lzero`). Il n'y a aucune contrainte d'univers, puisque celles-ci concernent le polymorphisme d'univers.

— Le champ `declarations` liste toutes les déclarations d'inductifs, il peut y en avoir plusieurs si celui-ci est un inductif mutuel. Au sein de ce champ, nous trouvons une seule déclaration qui correspond à `nat`.

— Le champ `cstr_indices` est vide pour tous les constructeurs car il n'a des éléments que dans le cas de la réification d'inductifs avec annotations (nous les avons définis en 2.1.2).

— Dans le type du constructeur, le type inductif lui-même est représenté par une variable libre, de même que les éventuels paramètres. La variable la plus récente est celle du dernier paramètre introduit (donc l'indice de De Bruijn le plus faible), et la moins récente est celle qui représente l'inductif. Par exemple, le type du constructeur `S` étant `forall _ : nat, nat`, sa représentation en `MetaCoq` sera `tProd ... (tRel 0)(tRel 1)`.

Notons que, de façon similaire, les constantes réifiées récursivement disposent d'un environnement global muni de la déclaration de la constante en question (où l'on peut trouver, bien sûr, le corps de cette constante).

#### Métaprogrammer en `MetaCoq`

La plupart des fonctions `MetaCoq` s'écrivent en inspectant, par *pattern matching*, la forme du terme que l'on cherche à transformer. Le premier constructeur, `tRel`, est celui des variables représentées par des indices de De Bruijn. Une grosse partie de l'implémentation en `MetaCoq` consiste donc à calculer le bon indice lors de la construction de termes particuliers.

Pour faire au mieux de l'arithmétique des indices de De Bruijn sans se tromper, j'ai trouvé que la bonne façon de raisonner était de se demander à chaque fois « combien de variables ai-je introduit *après* l'indice que je considère ? », car cela donnera le nombre de *lifts* à effectuer.

Un deuxième aspect des difficultés d'implémentation est le fait que nous sommes dans `Coq`. Nous devons donc écrire des fonctions qui terminent, et qui ne peuvent pas utiliser d'effet de bord. En particulier, les fonctions doivent renvoyer *toutes* les valeurs qu'elles modifient, et donc, des $n$-uplets ou des enregistrements, ce qui rend vite le code moins lisible. Elles doivent aussi prendre en argument toutes les valeurs dont elles ont besoin, conduisant parfois à écrire des fonctions prenant en entrée des dizaines d'arguments ou des enregistrements dont il faut ensuite retrouver les champs (ce qui conduit à un problème similaire).

Par ailleurs, puisque les fonctions doivent terminer, il faudra leur rajouter un paramètre de *fuel* (un entier qui diminue de 1 à chaque appel de la fonction), ou prouver leur terminaison, ou, tout simplement, puisque ces fonctions `MetaCoq` ne servent jamais dans les preuves, désactiver la condition de garde grâce à la commande `Unset Guard Checking`. Nous avons choisi de ne jamais prouver la terminaison, puisque nous ne l'aurions pas fait non plus dans un autre métalangage.

### 3.4.5 Faire fi des lieurs : `Coq-Elpi`

#### Présentation du langage `Elpi`

Le dernier métalangage que nous avons utilisé dans cette thèse est `Coq-Elpi` [42]. Il permet de faire de la métaprogrammation pour `Coq` dans un langage appelé `Elpi`, qui utilise *la syntaxe abstraite d'ordre supérieur* et la *programmation logique*.

Le travail fastidieux consistant à travailler sur de l'arithmétique des indices de De Bruijn peut finir par convaincre le ou la métaprogrammeur·euse que cette représentation des variables, qui est certes efficace pour les machines, est trop contraignante pour les humains.

Une autre manière de représenter les variables a vu le jour, grâce à la syntaxe abstraite d'ordre supérieur (SAOS). L'idée est de représenter les fonctions du langage-objet par des fonctions du métalangage. En SAOS, il n'y a jamais de variable libre : si jamais nous devons passer sous un lieur dans le langage objet, une variable est introduite dans le métalangage, et elle demeure donc liée.

**Exemple 22** Le lambda calcul en SAOS

Pour représenter les termes du lambda-calcul en `Coq`, la version avec des indices de De Bruijn est la suivante :

```
Inductive term : Set :=
  | Var : nat -> term
  | Lam : term -> term
  | App : term -> term -> term
```

La première projection $\lambda x.\ \lambda y.\ x$ sera donc le terme `Lam (Lam (Var 1))`. En SAOH, l'inductif correspondant (en désactivant la condition de positivité pour éviter l'occurrence négative de `term`) sera :

```
Inductive term : Set :=
  | Lam : (term -> term) -> term
  | App : term -> term -> term
```

Il n'y a plus de constructeur pour les variables, puisqu'il n'est pas possible de parler de variables libres. Le terme correspondant à la première projection sera : `Lam (fun x => Lam (fun y => x))`.

Par ailleurs, `Elpi` utilise la *programmation logique* (Prolog). Au lieu d'écrire des fonctions, on écrit des prédicats, avec des arguments considérés comme des entrées et d'autres comme des sorties. Chaque fois que la relation est vraie, on écrit une clause, avec, à gauche, la conclusion de la clause, et à droite, la condition pour qu'elle soit vraie (permettant d'éventuels appels récursifs). De plus, les clauses sont résolues par `Elpi` en utilisant un algorithme d'*unification*.

**Exemple 23** Concaténation de deux listes

Dans le langage `Elpi`, la concaténation de deux listes (polymorphes) s'écrit ainsi :

```
pred append i:list A, i:list A, o:list A.
  append [] L L .
  append [X|XS] L [X|L1] :- append XS L L1 .
```

La première ligne déclare le prédicat, le type de ses arguments et s'ils sont des entrées (`i`) ou des sorties (`o`). La deuxième ligne est le cas de base : quand la première liste est vide, le troisième argument de la relation (la sortie), doit s'unifier avec la deuxième liste `L` donnée en entrée. La troisième ligne est un cas d'appel récursif : si on a `append XS L L1`, alors on doit avoir `append [X|XS] L [X|L1]`.

Pour écrire des clauses qui passent sous des lieurs, on doit utiliser la notation `pi`. Cet opérateur permet d'introduire une constante fraîche au sein du programme, et d'éventuelles hypothèses à son sujet.

**Exemple 24** Le lambda-calcul simplement typé en `Elpi`

En supposant donné un type `ty` pour les types, on a :

```
type arr ty -> ty -> ty. % constructeur du type flèche

type lam (term -> term) -> term. % constructeur pour les lambda-
    abstractions
type app term -> term -> term. % constructeur pour les applications

of (app F X) T :- of F (arr S T), of X S. % typage de l'application
of (lam F) (arr S T) :- pi x\ of x S => of (F x) T. %typage de la
    lambda abstraction
```

Le cas intéressant est celui du typage de la lambda-abstraction : à droite, on introduit une variable `x` dont on suppose qu'elle a le type `S`, et dans ce cas `F x` doit avoir le type `T`. La variable doit toujours être introduite explicitement par un `pi` avec des conditions sur celle-ci écrites avant l'opérateur `=>`. Cela permet d'avoir un argument générique à donner à `F` et à faire un appel récursif sur un argument de type `term`, car rappelons que `F` est une fonction.

### Les motivations de `Coq-Elpi`

Le plugin `Coq-Elpi` propose une interface pour métaprogrammer en `Coq` tout en utilisant le langage `Elpi`. L'idée de `Coq-Elpi` est double : c'est à la fois de proposer un nouveau métalangage pour `Coq` qui utilise la SAOS, et, à plus long terme, de proposer un nouvel élaborateur pour `Coq` [42].

Pour le premier aspect, nous avons déjà remarqué que calculer le bon indice de De Bruijn peut devenir rapidement difficile et source de bugs. La SAOS permet de traverser récursivement les lieurs du terme et propose une représentation plus proche de celle que l'on aurait pu avoir sur papier. Les termes avec des variables libres n'ont pas de sémantique associée, ils existent seulement pour des manipulations syntaxiques. Souhaiter travailler seulement dans des contextes clos pour éviter ces problèmes de variables libres est donc naturel.

Pour le second aspect, nous avons mentionné que `Coq` utilise un « double » du type `constr`, appelé `Econstr`, qui comprend des métavariables existentielles, c'est à dire des trous qui attendent d'être instanciés par un terme concret. Trouver ces instanciations, via un algorithme d'unification d'ordre supérieur, est la tâche de l'élaborateur. Or, pour des raisons qui dépassent le cadre de cette thèse, les développeur·euses de `Coq-Elpi` considèrent que leur langage est particulièrement adapté pour pallier les défauts de l'élaborateur actuel de `Coq` (notamment, la duplication de code entre les `constr` et leurs « jumeaux existentiels » les `Econstr`). En effet, le langage `Elpi` est fondé précisément sur l'unification d'ordre supérieur. Par ailleurs, au lieu d'avoir un constructeur `OCaml` correspondant aux métavariables `Coq` dans le type `Econstr` comme c'est le cas actuellement, l'idée serait d'utiliser directement les métavariables de `Elpi`, au sein d'un unique type `constr` (sans plus de jumeau, donc).

### Métaprogrammer en `Coq-Elpi`

À l'instar de `Ltac2`, `Coq-Elpi` propose toutes les API qui permettent de manipuler l'environnement global de `Coq`. Ce métalangage permet aussi d'appeler du code `Ltac` au sein d'un programme écrit en `Elpi`, ce qui permet de mieux mêler les métalangages au besoin. Il dispose bien sûr d'un type `term` représentant les termes de `Coq`, où les constructeurs présentant des lieurs utilisent la SAOS. Il est aisé de passer de la réification à la déréification, de la syntaxe de `Coq` à celle de `Coq-Elpi`. Par exemple, le terme `coq:{{ lp:A -> lp:B}}` permet d'utiliser les métavariables de `Elpi` à l'intérieur d'une flèche de `Coq`.

J'ai trouvé toutes les fonctions qui traversent l'arbre de syntaxe abstraite d'un terme très aisées à écrire en `Coq-Elpi`, une fois habituée à la syntaxe de la programmation logique. Elles se révèlent également plus concises. Des transformations dont la lisibilité est compromise par la manipulation des indices de De Bruijn peuvent se résumer en quelques lignes de `Coq-Elpi`. Un exemple est la transformation qui élimine les égalités d'ordre supérieur dans `Sniper`, qui a été implémentée en `MetaCoq` et en `Ltac`. Une nouvelle version a été implémentée en `Coq-Elpi` (avec la relecture et l'aide précieuse de Enrico Tassi)[3]. Notons que le cœur de la transformation tient en 7 lignes de code, et elle est débarrassée de l'appel à des fonctions `lift` qui entravaient la lisibilité du code dans d'autres métalangages comme `MetaCoq` ou `Ltac2`.

En revanche, dès que la structure du terme généré n'a que peu de rapport avec le terme initial, la gestion des lieurs en SAOS devient beaucoup plus difficile. Parfois, on introduit une variable `x: T` dans le contexte du programme, mais, la même variable `x` devra avoir un type différent, mettons, `T'`, en sortie, ce qui est cause de difficultés.

Par ailleurs, la gestion du contexte de preuve est difficile. Le métalangage `Coq-Elpi` permet d'écrire des tactiques. La majorité des prédicats `Elpi` agissant sur un but `Coq` prennent en argument un but « ouvert » et ont pour sortie une liste de buts « scellés ».

Un but « ouvert » a introduit toutes les hypothèses et les définitions du contexte de la preuve `Coq` dans le contexte du programme `Elpi`. Autrement dit, il y a des variables de type `term` dans le contexte du programme qui correspondent à chacune des hypothèses ou des définitions dans le but `Coq`. Par exemple, si dans le but nous avons `H: nat`, alors dans le contexte du programme `Elpi`, on a une clause qui nous assure que la variable `H` vérifie `of H nat`. En revanche, les buts en sortie de ces prédicats sont « scellés », donc toutes les variables du contexte sont liées (par des lieurs spécifiques appelés `nabla`). Cela implique que si l'on mentionne la version réifiée d'une hypothèse (mettons qu'il s'agisse de la variable `x`) dans le but initial $G$, elle n'existera plus dans les buts $G_1$, ..., $G_n$ produits par l'application d'une tactique. Ceci est très souvent source d'erreur. Par exemple, en `Ltac`, si le but est de la forme `A\/B` et présente dans le contexte de preuve `x : A`, on peut simplement écrire `left; exact x` pour résoudre le but. En `Elpi`, `x` n'existera plus après l'application de `left`. Cette difficulté du passage de variables d'un but à l'autre peut être résolue en étendant le type `term` avec un nouveau constructeur, `pos_ctx`, qui prend en argument un entier. On remplace, dans les termes sur lesquels on travaille, chaque occurrence d'une variable du contexte local par `pos_ctx n`, où `n` correspond à la position de cette variable au sein du contexte. (Notons qu'ici, ceci revient à utiliser des niveaux de De Brujin (la variable la plus récente a l'indice le plus élevé) pour les variables du contexte local). Dans le nouveau but, on pourra remplacer `pos_ctx n` par la variable effectivement à cette position dans le contexte. Malheureusement, cette astuce fait perdre en efficacité, car elle introduit plusieurs passes sur le contexte de preuve.

Pour résumer, `Coq-Elpi` est un métalangage très puissant quand il s'agit d'écrire du code lisible et concis qui agit sur des *termes*. Il est beaucoup plus difficile à prendre en main quand on l'utilise pour écrire des *tactiques*. En revanche, grâce au travail de ses développeur·euses, la documentation est extrêmement fournie et permet une meilleure prise en main que tous les autres métalangages.

3. https://github.com/louiseddp/metaprogramming-rosetta-stone/blob/main/ho_equalities/elpi/ho_equalities.v

# Chapitre 4

# TRANSFORMATIONS

Ce chapitre présente le cœur du code de `Sniper` : la bibliothèque des transformations atomiques, leur spécification informelle et parfois leur code.

Deux manières de classifier ces transformations coexistent : par l'*utilité* des transformations (ce *pour quoi* elles ont été implémentées) et par la *façon* dont elles agissent (*comment* elles se déclenchent et *comment* elles modifient le but `Coq` qui leur est donné).

J'ai opté pour la première des classifications mais nous verrons que la seconde est très utile quand il s'agit d'*ordonnancer* les transformations, au chapitre 5.

## 4.1 Interprétation des fonctions

Cette première série de transformations, bien qu'utilisables indépendamment les unes des autres, ont été pensées et conçues dans un but précis, qui nécessite leurs appels successifs dans l'ordre où je vais les présenter.

Ce but est de donner la sémantique des fonctions définies dans l'environnement global de `Coq` dans un langage supporté par un prouveur SMT directement dans le contexte local. En effet, les prouveurs externes de `SMTCoq` n'ont pas accès à l'environnement global de `Coq`. De plus, même si une fonctionnalité était ajoutée pour chercher la définition des constantes de `Coq`, celle-ci serait obtenue dans un langage qui n'est pas celui des prouveurs.

**Exemple 25** [Fonctions non interprétées]

Supposons que nous souhaitons prouver le but `Coq` suivant automatiquement, en ayant à notre disposition dans le contexte *local* les variables de section `H1` et `H2`, ainsi que la définition de `length_nat` :

```
Section example_transfo_functions.

Fixpoint length_nat (l : list nat) :=
  match l with
    | [] => 0
    | x :: xs => S (length_nat xs)
  end.

Variable (H1: length_nat [] = 0).

Variable (H2: forall x xs,
```

```
length_nat (x::xs) = S (length_nat xs)).

Lemma length_nat_cons_not_nil x xs :
  length_nat [] <> length_nat (x::xs).
  Proof. verit. Qed.

End example_transfo_functions.
```

Si l'on décide de mettre en commentaire les deux ou l'une des deux hypothèses `H1` et `H2`, alors la tactique `verit` renvoie le message d'erreur suivant : `veriT returns 'unknown'`. Ceci nous montre bien que `H1` et `H2` vont permettre à `veriT` de résoudre le but. En effet, ce prouveur ne connaît pas la définition de `lenght_nat`. Même en ajoutant une variable de section `H3` qui fournit la définition souhaitée à la place de `H1` et `H2` :

```
Variable (H3: length_nat =
fix length_nat (l : list nat) : nat :=
  match l with
  | [] => 0
  | _ :: xs => S (length_nat xs)
  end).
```

la tactique `verit` fournit un message d'erreur un peu plus cryptique (`Anomaly " Uncaught exception Invalid_argument("index out of bounds").`") qui indique en réalité que la réification de l'hypothèse `H3` a échoué (d'où une recherche dans un tableau des types réifiés qui échoue). L'hypothèse n'est tout simplement pas en logique du premier ordre.

J'ai donc écrit quatre transformations pour pallier ce problème. Le but est de fournir les définitions des fonctions `Coq` sous forme équationnelle à `veriT` (en logique polymorphe du premier ordre).

Une autre manière de voir ces transformations est qu'elles font l'encodage réciproque de l'une des transformations fournies par le plugin `Coq` appelé `Equations` [41].

En effet, en fournissant les équations qui définissent la fonction `length_nat`, ce plugin permet de reconstituer la définition sous forme de point fixe. Au contraire, j'ai écrit des transformations qui partent du point fixe pour finalement obtenir les équations.

### 4.1.1 Définitions

La première tactique de `Sniper` est très simple : elle prend en entrée un terme `Coq` et si celui-ci est une constante, elle réifie l'égalité définitionnelle en égalité propositionnelle et l'ajoute au contexte local.

Par exemple, considérons la relation `in_int`, définie dans la librairie standard de `Coq` dans le ficher `Arith`. Elle prend en argument trois entiers et affirme que le troisième est dans l'intervalle défini par le premier (inclus) et le deuxième (exclu). En appelant `get_def in_int`, le contexte local sera étendu par une nouvelle hypothèse (prouvée) : `in_int_def : in_int = fun p q r => p <= r /\ r < q`.

Le code de la tactique[1] (atomique) est le suivant :

```
Ltac get_def x :=
```

1. Voir aussi : https://github.com/louiseddp/sniper/blob/for-phd/theories/definitions.v#L89

```
let x' := eval unfold x in x in
let H := fresh x "_def" in assert (H : x = x') by reflexivity.
```

Comme on peut le remarquer, la preuve est également très simple. Puisque la constante et sa version dépliée sont $\delta$-convertibles, la réflexivité de l'égalité suffit à conclure.

La partie difficile de cette transformation est que nous avons de nombreux termes dont nous ne voulons *pas* déplier la définition, sous peine de générer des informations inutiles et donc d'encombrer le contexte local de définitions déjà connues par le backend, ou de générer des hypothèses qui ne pourront de toute façon pas être traitées par la suite. Nous en parlerons dans la section 5.1 où est présenté l'agencement des transformations.

Des exemples d'utilisation sont répertoriés ici : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_definitions.v.

### 4.1.2 Réflexivités et dépliage

Une première version de la tactique `snipe` a recours à la transformation des définitions. Mais, à cause des problématiques que nous soulèverons en 5.1, cette transformation a été divisée en trois encore plus atomiques :

— La transformation `assert_reflexivity` qui, pour toute constante `c` génère et prouve l'hypothèse de type `c=c`.

— La transformation `unfold_reflexivity` qui déplie à droite toute égalité de la forme `c=c`.

— La transformation `unfold_in` qui prend en argument une hypothèse `H` et un terme `c`, et qui déplie la définition de `c` dans `H`.

Notons que combiner `assert_reflexivity` et `unfold_reflexivity` permet déjà de revenir à ce qu'accomplit la transformation des définitions, à condition de rappeler récursivement la transformation `assert_reflexivity` sur d'éventuelles constantes apparues suite à l'exécution de `unfold_reflexivity`.

Par exemple, considérons la fonction `nth_default`.

Appliquer `assert_reflexivity` à `nth_default` va générer et prouver une hypothèse :

`H : nth_default= nth_default`

Puis, appeler `unfold_reflexivity H` va permettre de déplier la définition de `nth_default` comme l'aurait fait directement la transformation des définitions `get_def`. On obtient alors :

```
H : nth_default =
  fun (A : Type) (default : A)
  (l : list A) (n : nat) =>
  match nth_error l n with
  | Some x => x
  | None => default
  end
```

Ici, on voit que `nth_default` fait référence à `nth_error`, on aura donc besoin de rappeler `assert_reflexivity` sur `nth_error`, puis de nouveau utiliser `unfold_reflexivity` pour obtenir la définition de cette nouvelle fonction.

L'utilité de ces transformations, et notamment de la transformation `unfold_in` dont nous n'avons pas parlé dans l'exemple, sera expliquée quand la combinaison des transformations sera décrite, et notamment, en 5.2.3.

Dans l'exemple récapitulatif de cette section, nous n'utiliserons pas ces transformations, mais nous les utiliserons en 5.4.

### 4.1.3 Expansion

Le code de la transformation est disponible dans ce fichier : https://github.com/louiseddp/sniper/blob/for-phd/theories/expand.v.

Cette transformation ne s'applique qu'à *toplevel*, à chaque fois qu'une hypothèse $H$ a un type de la forme $f = g$ avec $f$ et $g$ de type $\forall (x_1 : A_1) ... (x_n : A_n),\ B$.

La tactique va alors transformer cette hypothèse de manière à éliminer l'égalité d'ordre supérieur, en quantifiant sur les arguments des deux fonctions. Ceci revient à faire du $\lambda$-lifting : les variables locales à la fonction sont remplacées par des arguments sur lesquels on quantifie hors de la fonction. Ainsi, $H$ deviendra :

$H' : \forall (x_1 : A_1) ... (x_n : A_n),\ f\ x_1 ... x_n = g\ x_1 ... x_n$

Il faut bien noter que dans cette transformation, nous n'avons *pas* besoin de l'extensionnalité des fonctions. Au contraire, cette transformation utilise la *réciproque* de l'axiome.

Nous perdons en prouvabilité en transformant $H$ en $H'$ car effectivement, prouver que $H' \rightarrow H$ nécessiterait l'extensionnalité des fonctions, mais nous avions déjà précisé que les transformations de `Sniper` n'ont pas vocation à être complètes.

Par exemple, supposons qu'on dispose dans le contexte local de l'hypothèse :

```
H: hd_error =
fun (A : Type) (l :list A) =>
match l with
  | [] => None
  | x :: _ => Some x
end
```

L'appel à la transformation `expand H` génère et et prouve :

```
H': forall (A: Type) (l: list A),
@hd_error A l =
match l with
  | [] => None
  | x :: _ => Some x
end
```

Nous utilisons les tactiques de `Coq` pour générer ce terme de preuve : il s'agit d'une combinaison de `rewrite` et de `reflexivity`.

**Implémentation** Cette tactique combine deux langages de métaprogrammation : `Ltac` et `MetaCoq`. Le *pattern matching* superficiel est effectué en `Ltac`. Ici, on matche sur une hypothèse pour vérifier qu'elle est bien une égalité à `toplevel`. Puis, le type de l'égalité est réifié en `MetaCoq` de façon à s'assurer que l'égalité est bien d'ordre supérieur. Si tel est le cas, on crée autant d'arguments que le terme contient de produits dépendants, on réifie les deux termes qui sont égaux et on génère la nouvelle égalité. La construction de celle-ci s'appuie sur la fonction auxiliaire `gen_eq` (le code est simplifié afin de ne pas s'occuper de la génération de noms pour les arguments de la fonction).

```
Fixpoint gen_eq
(l : list term) (* types des arguments des fonctions *)
(B : term) (* codomaine des fonctions *)
(t : term) (* fonction 1 *)
(u : term) (* fonction 2 *)
{struct l} :=
```

```
match l with
  | [] => mkEq B t u
  | A :: l' => mkProdName "x" A
    (gen_eq l' B (tApp (lift 1 0 t) [tRel 0]) (tApp (lift 1 0 u) [tRel 0]))
end.
```

Le cas de base génère l'égalité dans le type `B` entre les termes `t` et `u` qui auront été appliqués à des variables libres. Dans le cas récursif, on applique les termes `t` et `u` à une variable fraîche qui a pour type `A`, et qui sera liée par la fonction `mkProdName` (celle-ci construit un produit dépendant). On rappelle `gen_eq` sur les deux termes partiellement appliqués et le reste de la liste des types des arguments des fonctions. Lorsque le terme généré $T$ est dé-réifié, il est transparent qu'on ait écrit $(...(f\ x_1)\ x_2)...\ x_n)$ au lieu de $f\ x_1\ ...\ x_n$ car `MetaCoq` « aplatit » les applications automatiquement. Enfin, la nouvelle hypothèse $H : T$ est assertée puis prouvée par réflexivité en `Ltac`, et l'hypothèse initiale dont on n'aura plus besoin est effacée grâce à la tactique `clear`.

Des exemples d'utilisation sont répertoriés ici : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_expansion.v.

### 4.1.4 Réduction des points fixes

Le code de la transformation est disponible dans ce fichier : https://github.com/louiseddp/sniper/blob/for-phd/theories/elimination_fixpoints.v.

Cette transformation remplace un point fixe anonyme par sa définition (une constante `Coq` $c$) dans le corps du point fixe. Une autre manière de le voir est que l'on effectue une étape de réduction du point fixe dans un terme, en remplaçant la variable $f$ qui définit le point fixe par la construction $fix\ f\ ...$ (comme vu au paragraphe 2.1.2), et qu'ensuite on *replie* cette définition en la remplaçant par la constante $c$.

Par exemple :

```
forall (A: Type)(l : list A), @length A l =
(fix length_anon (l : list A) : nat :=
  match l with
  | [] => 0
  | _ :: l' => S (length_anon l')
  end) l
```

est transformé en :

```
forall (A: Type)(l : list A), @length A l =
  match l with
  | [] => 0
  | _ :: l' => S (length l')
  end
```

Une condition nécessaire pour utiliser la réduction du point fixe est que celui-ci soit appliqué à au moins $n$ termes, $n$ étant la position de l'argument récursif. Ainsi, la transformation ne fonctionnera que si le point fixe est suffisamment appliqué.

**Implémentation** Cette transformation est implémentée en `Coq-Elpi`. Une première version moins générale était implémentée dans un mélange de `Ltac` et `MetaCoq` : elle ne traitait que le cas où l'hypothèse était de la forme :

$\forall \vec{x},\ t\ \vec{x} = fun\ ...\ \Rightarrow fix\ ...\ \vec{x}$.

et une tentative d'implémentation a été faite en `Ltac2`, mais seul `Coq-Elpi` nous a paru suffisamment souple pour répondre aux besoins de cette transformation. En effet, la manipulation des lieurs est trop contraignante en `Ltac` ou même en `Ltac2` (au moment où l'implémentation a été tentée, car, depuis les fonctionnalités manquantes en `Ltac2` ont été rajoutées).

Par exemple, dans l'implémentation en `Coq-Elpi`, le point fixe anonyme peut se présenter en position négative ou comme argument d'une autre fonction. Ainsi, si l'on dispose d'une variable `toto : nat -> nat`, l'hypothèse :

```
H : forall A l, toto (length l) =
toto (fix length (l : list A) : nat :=
match l with
| [] => 0
| _ :: l' => S (length l')
end l) -> True
```

deviendra :

```
H' : forall (A : Type) (l : list A), toto (length l) =
toto (match l with
| [] => 0
| _ :: l' => S (length l')
end) -> True
```

L'idée de l'implémentation est d'abord d'identifier tous les points fixes anonymes présents dans l'hypothèse $H$ initiale. Quand leur définition utilise des variables libres, il faut abstraire celles-ci. Un cas où cette abstraction est nécessaire est par exemple le point fixe anonyme dans le corps de la fonction `length`, qui est paramétré par une variable de type `A` :

```
fix length (l : list A) : nat :=
  match l with
  | nil => 0
  | (_ :: l') => S (length l')
  end
```

Ce terme deviendra, une fois abstrait :

```
fun (A: Type) =>
fix length (l : list A) : nat :=
  match l with
  | nil => 0
  | (_ :: l') => S (length l')
  end
```

Ensuite, on cherche dans le contexte local les constantes qui se réduisent dans les points fixes (abstraits) de la liste

$[\lambda\ (\vec{x_1} : \vec{A_1})\ (fix\ f_1\ \lambda(\vec{y_1} : \vec{B_1})\ t_1),\ ...,\ \lambda\ (\vec{x_n} : \vec{A_n})\ (fix\ f_n\ \lambda(\vec{y_n} : \vec{B_n})\ t_n)]$.

Si on ne trouve pas de telle constante pour l'un des points fixes, la transformation échouera. Si par contre on trouve une liste de constantes $[c_1,\ ...,\ c_n]$, on crée un nouvel énoncé $H_i$ par paire de constante et de son point fixe associé :

$$\forall(\vec{a_i} : \vec{A_i})(\vec{b_i} :\ \vec{B_i}),\ (\lambda\vec{x_i}, (fix\ f_i\ \lambda(\vec{y_i} : \vec{B_i})\ t_i))\ \vec{a_i}\ \vec{b_i} = t_i[f_i := c_i\ \vec{a_i}].$$

La preuve de chaque $H_i$ consiste à introduire les arguments du point fixe jusqu'à avoir dans son contexte local l'argument récursif $k$. Cet argument $y_k : B_k$ est ensuite détruit via la tactique `destruct`, de façon à ce que l'argument en position $k$ dans chaque sous-but commence par un constructeur du type inductif $B_k$. Ceci permet d'appliquer la réduction du point fixe et donc de conclure par `reflexivity`.

Une fois les $H_i$ prouvées, on réécrit chacune d'entre elles dans l'hypothèse $H$ initiale. Comme la réécriture peut être faite sous des lieurs, on utilise `setoid_rewrite` au lieu de `rewrite`. Comme les $H_i$ ne servent qu'à modifier l'hypothèse d'entrée, ils sont ensuite effacés afin de ne pas encombrer le contexte local.

Des exemples d'utilisation (notamment quand le point fixe n'est pas à *toplevel*) sont répertoriés ici : `https://github.com/louiseddp/sniper/blob/for-phd/examples/example_fixpoint.v`

### 4.1.5 Élimination du pattern matching

Le code de la transformation est disponible dans ce fichier : `https://github.com/louiseddp/sniper/blob/for-phd/theories/elimination_pattern_matching.v`.

Le *pattern matching* sur un terme n'est pas présent dans la syntaxe de la logique du premier ordre. Il est vrai que nous pouvons considérer des extensions de celle-ci qui l'admettent, mais les prouveurs automatiques de `SMTCoq` en particulier ne le gèrent pas.

Mais le *pattern matching* peut se représenter par une *liste d'équations*, chacune d'entre elles correspondant à un constructeur du type inductif dont fait partie le terme filtré. C'est donc l'encodage que nous avons choisi pour cette transformation.

Cette transformation peut traiter une hypothèse de la forme :
$\forall(\vec{x} : \vec{A})$, $C[$`match` $f\ \vec{x}$ `with` $c_i\ \vec{y_i} \Rightarrow\ g_i\ \vec{y_i}$ `end`$]$.

Un cas particulier très fréquent est lorsque lorsque $f\ \vec{x}$ est l'une des variables quantifiées $\vec{x}$, et où le *pattern matching* s'effectue alors sur une variable. Un exemple est la formule en 4.1.4, où la fonction `length` est définie par récursion sur la variable `l`.

Un autre cas limite géré par la transformation est lorsque le `match` est à *toplevel* et pas derrière des quantificateurs universels.

Pour en revenir au cas général, après l'appel de la transformation, l'hypothèse initiale est remplacée par $n$ nouvelles hypothèses, $n$ correspondant au *nombre de constructeurs* du type inductif $I$ de $f\ \vec{x}$.

Ces hypothèses, notées les $H_i$, seront les suivantes :

$$H_i : \forall(\vec{x} : \vec{A})(\vec{y_i} : \vec{B_i}), f\ \vec{x} = c_i\ \vec{y_i} \rightarrow C[g_i\ \vec{y_i}]$$

**Exemple 26** [Cas général]

Prenons comme exemple une définition de fonction par pattern matching sur une expression qui n'est pas une variable. L'hypothèse d'entrée `H` a pu être obtenue par le prétraitement des transformations que nous avons déjà présentées.

```
H: forall A d l n, @nth_default A d l n =
match nth_error l n with
| Some x => x
| None => d
end
```

Apprès l'appel de `eliminate_dependent_pattern_matching H`, on obtient `H1` et `H2` :

```
H1: forall A d l n x,
  @nth_error A l n = Some x -> @nth_default A d l n = x.
```

```
H2: forall A d l n,
  @nth_error A l n = None -> @nth_default A d l n = d.
```

La preuve des nouveaux énoncés $H_i$ se présente sous la forme d'un script de tactiques. Celui-ci consiste à introduire les variables quantifiées universellement (les $\vec{x}$ et les $\vec{y_i}$) ainsi que l'éventuelle hypothèse d'égalité $H_{eq}$. L'hypothèse initiale $H$ sera spécialisée aux variables $\vec{x}$, et on y réécrira $H_{eq}$ si besoin. Après cette opération, $H$ et le but seront convertibles, ce qui permet de conclure avec `assumption`.

**Implémentation** La tactique principale est implémentée en une cinquantaine de lignes en `Ltac`. Son écriture m'a confrontée aux limitations très contraignantes de ce langage et consiste en une suite d'astuces permettant de les contourner. Elle présente donc un cas d'application intéressant des techniques présentées en 3.4.2, notamment :

1. Comme `Ltac` ne permet pas de travailler facilement sous des lieurs, il faut avoir un but qui ait précisément le type de l'hypothèse que la transformation traite, pour pouvoir introduire les termes liés dans le contexte local. Ceci ne fonctionne évidemment que dans le cas de quantification *prénexe*.
2. Le seul moyen, à ma connaissance, de transmettre une information d'un sous-but à un autre est d'utiliser des *evar*, qui seront instanciées dans un sous-but $G_1$, but ayant permis de faire apparaître un terme utile à un autre sous-but $G_2$.

L'idée est de créer deux *evar*, `n_evar` et `T_evar` l'une pouvant contenir un nombre entier et l'autre un type quelconque. Le terme `n_evar` sera instancié par le nombre de quantificateurs (universels) à introduire avant que `Ltac` ne puisse détecter la présence d'un sous-terme du but comportant un `match` à *toplevel*. `Ltac` ne le détectera en effet que si celui-ci ne mentionne pas de variables libres.

La variable `T_evar` est ensuite instanciée par le type (inductif) $I$ du terme sur lequel on effectue le *pattern matching*. Cette information est nécessaire pour savoir combien d'hypothèses $H_i$ seront générées à partir de $H$. Une sous-tactique va récupérer le nombre de constructeurs de $I$ (*via* une réification de celui-ci en `MetaCoq`) et créer une *evar* pour chaque constructeur. Cela permettra d'instancier les *evar* ainsi générées par nos nouveaux buts.

Une fois que le nombre de variables à introduire est connu et que les *evar* sont déclarées, on crée un but $False \to T$ avec $T$ le type de l'hypothèse initiale. On introduit l'hypothèse absurde et tous les quantificateurs universels, en prenant soin de conserver les termes introduits dans un tuple (pour des raisons de typage, on ne peut pas utiliser une liste en `Ltac`).

Quand `Ltac` peut de nouveau détecter le `match` et l'expression $e$ filtrée, on peut créer les sous-buts qui auront la forme requise pour les $H_i$. Selon si $e$ est une variable ou une expression plus complexe, on utilisera la tactique `case e` ou `case_eq e`.

La tactique `case` applique l'éliminateur du type $I$ : ainsi, si le but est $C[e]$, il sera changé en $n$ sous-buts (autant que de constructeurs de $I$), de la forme $\forall(\vec{y_i} : \vec{B_i}), C[g_i\ \vec{y_i}]$ (nous avons précisé au-dessus à quoi correspondaient ces noms de variables). La tactique `case_eq` fait la même chose mais introduit l'égalité supplémentaire $f\ \vec{x} = c_i\ \vec{y_i}$, qui n'est pas nécessaire lorsqu'on matche sur une variable (car elle sera triviale). Il reste à utiliser `revert` sur les variables introduites pour retrouver les quantificateurs universels et avoir $n$ buts qui sont précisément les $H_i$. Les *evar* seront instanciées par ces buts, mais pas encore prouvées : il faut le faire dans le but principal et non pas dans le but $False \to T$ pour ne pas perdre l'information. Ce but $False \to T$ est prouvé trivialement et il ne reste qu'à démontrer les $H_i$.

La preuve est faite en introduisant les $\vec{x}$ et en spécialisant l'hypothèse initiale $H$ à ceux-ci. Ensuite, on cherche l'éventuelle égalité $f\ \vec{x} = c_i\ \vec{y_i}$ qu'on réécrit dans $H$, et on peut conclure par

assumption. Dans le cas où $H$ est déjà une égalité (quantifiée), on peut accélérer la preuve et simplement utiliser le script de tactiques intros; rewrite H ; reflexivity.

Des exemples d'application de la transformation sont disponibles ici : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_pattern_matching.v

### 4.1.6 Retour sur l'exemple initial

Maintenant que j'ai présenté le groupe de transformations permettant d'obtenir (entre autres) les équations de fonctions, il est possible de revenir sur l'exemple problématique de 4.1.

L'idée est de récupérer d'abord la définition de length_nat et de l'ajouter au contexte local grâce à la transformation des définitions. On obtient :

```
length_nat_def : length_nat =
fix length_nat (l : list nat) : nat :=
  match l with
  | [] => 0
  | _ :: xs => S (length_nat xs)
  end
```

On peut appeler la transformation d'expansion sur cette hypothèse, et on obtient :

```
length_nat_def_expansed : forall (l : list nat), length_nat l =
  (fix length_nat (l : list nat) : nat :=
    match l with
    | [] => 0
    | _ :: xs => S (length_nat xs)
    end) l
```

Puis on peut éliminer le point fixe anonyme en utilisant la troisième transformation sur cette nouvelle hypothèse :

```
length_nat_def_expansed_fix : forall (l : list nat), length_nat l =
  match l with
  | [] => 0
  | _ :: xs => S (length_nat xs)
  end
```

Et enfin, on utilise l'élimination du *pattern matching* et on obtient deux hypothèses dans le contexte local qui sont précisément les variables de section que l'on avait rajoutées à la main et dont avait besoin le prouveur veriT :

```
H_nil : length_nat [] = 0
H_cons : forall (xs : list nat) (x : nat),
  length_nat (x :: xs) = S (length_nat xs)
```

Ainsi, la combinaison de ces quatre transformations de Sniper permet déjà d'étendre le nombre de buts que la tactique verit est capable de prouver.

## 4.2 Types algébriques

Dans cette thèse, nous appelons « type algébrique » un type pouvant s'obtenir par combinaison de types de base, au moyen d'opérateurs de type : les sommes (union disjointe) et les produits ($n$-uplets), et de la récursion.

Les types algébriques permettent de représenter des données structurées, l'exemple paradigmatique étant les listes. Par ailleurs, ceci exclut les types dépendants de termes[2] et les relations inductives (voir 4.3).

Les prouveurs SMT de `SMTCoq` ne supportent pas les types algébriques. Cela signifie que, dès lors qu'ils en rencontrent un, mettons, `list Z`, ils vont le considérer comme un type non interprété. Voyons ceci dans un exemple.

**Exemple 27** Types algébriques

Nous disposons de cinq variables de section, deux donnant la définition de la fonction de concaténation entre deux listes (comme vu au paragraphe précédent), et trois qui vont nous intéresser ici, donnant la sémantique du type algébrique `list Z`.

```
Section example_transfo_algtypes.

Variable app_nil : forall (l : list Z), l ++ [] = l.
Variable app_cons :
  forall (x : Z) (l xs : list Z),
  l ++ (x :: xs) = x :: (l ++ xs).

Variable noconf_principle : forall (x : Z)(xs : list Z),
[] = x :: xs -> False.
Variable inj_cons : forall (x x' : Z) (xs xs' : list Z),
x :: xs = x' :: xs' -> x = x' /\ xs = xs'.
Variable gen_principle: forall (l : list Z),
l = [] \/ l = hd 0%Z l :: tl l.


Lemma app_eq_unit_Z :
  forall (l l' : list Z) (x: Z), l ++ l'= [x] ->
    l = [x] /\ l' = [] \/ l = [] /\ l' = [x].
verit. Qed.

End example_transfo_algtypes.
```

Grâce aux trois dernières variables de section, `veriT` est capable de trouver la preuve de la propriété que nous souhaitons prouver. Mais en mettre une (ou plusieurs) en commentaire conduira soit le prouveur à chercher une preuve indéfiniment, soit à renvoyer le message `veriT returns 'unknown'`. Nous avons besoin de ces trois propriétés : la liste vide est différente de toute liste obtenue en utilisant `::`, le constructeur `::` est injectif et toute liste est soit vide, soit la combinaison d'une tête de liste et d'une queue de liste.

### 4.2.1 Trois propriétés fondamentales

Plus généralement, à partir de la définition d'un type algébrique, trois propriétés fondamentales sont dérivables.

Supposons que la définition d'un type `I`, à $m$ constructeurs, soit :

```
Inductive I (A1: Type) ... (An : Type) : Type :=
```

2. Précisons que tous les types dépendants de termes (qui ne sont pas des types) sont exclus, mais on autorise les types dépendants de types, et plus précisément le polymorphisme prénexe.

```
...
| ci : Bi1 -> ... -> Bik -> I A1 ... An
...
.
```

Les trois propriétés en question seront :

1. Le principe de non-confusion : deux termes obtenus à partir de l'application de deux constructeurs différents de $I$ ne peuvent être égaux.
   $$\bigwedge_{i,j=1,i\neq j}^{m} (\forall(\vec{A} : Type)(\vec{y_i} : \vec{B_i})(\vec{z_j} : \vec{B_j}),\ c_i\ \vec{A}\ y_{i_1}\ ...\ y_{i_k} \neq c_j\ \vec{A}\ z_{j_1}\ ...\ z_{j_k})$$
2. L'injectivité des constructeurs :
   $$\bigwedge_{i=1}^{m} (\forall(\vec{A} : Type)(\vec{y_i} : \vec{B_i})(\vec{z_i} : \vec{B_i}),\ c_i\ \vec{A}\ y_{i_1}\ ...\ y_{i_k} = c_i\ \vec{A}\ z_{i_1}\ ...\ z_{i_k} \rightarrow \bigwedge_{j=1}^{k} y_{i_j} = z_{i_j})$$
3. Le principe de génération : tout terme de type $I\ \vec{A}$ a été obtenu par l'application d'un constructeur de $I$.
   $$\forall(\vec{A} : Type)(t :\ I\ \vec{A}),\ \bigvee_{i=1}^{m} \exists(\vec{y_i} : \vec{B_i}), t = c_i\ \vec{A}\ y_{i_1}\ ...\ y_{i_k}$$

Ces propriétés donnent une partie de la sémantique au premier ordre des types algébriques considérés. Il manque bien sûr le principe d'induction associé, mais nous n'avons pas cherché à automatiser l'induction. De plus, contrairement à ces énoncés, le principe d'induction est automatiquement dérivé et enregistré comme une nouvelle constante à la déclaration d'un inductif.

La première transformation concernant les types algébriques implémentée dans `Sniper` prend en entrée un type algébrique `I`, puis génère dans le contexte local et prouve les deux premiers énoncés.

Le code de celle-ci est disponible dans ce fichier : https://github.com/louiseddp/sniper/blob/for-phd/theories/interpretation_algebraic_types.v.

Des exemples sont disponibles au code suivant :https://github.com/louiseddp/sniper/blob/for-phd/examples/example_algtypes.v.

Le principe de génération sera décrit dans le paragraphe suivant 4.2.2. En effet, tel qu'il est énoncé ci-dessus, il comporte des quantificateurs existentiels, non supportés dans `SMTCoq`. Nous avons dû en changer la forme de façon à ne pas utiliser ces quantificateurs.

Par exemple, `interp_alg_types list` va générer et prouver les énoncés correspondant aux variables de section `noconf_principle` et `inj_cons` dans l'exemple 27.

**Implémentation** Les fonctions principales de cette transformation ont été implémentées en `MetaCoq`, avec du `Ltac` pour les lier entre elles et les appeler.

En effet, pour rappel, `MetaCoq` ne propose pas de tactique agissant sur le contexte de preuve dans son ensemble, et ne permet que de travailler sur des termes individuellement. Par contre, `MetaCoq` permet facilement de construire des nouveaux termes, avec des variables libres.

Nous aurions pu utiliser aussi `Coq-Elpi`, mais ce dernier métalangage est plus adapté lors de traversées de l'arbre syntaxique du terme d'entrée pour produire un nouveau terme. Toutefois, ici, nous souhaitons générer un tout nouveau terme dont la structure ne dépend pas de celle du terme d'entrée, mais plutôt d'informations à son sujet.

La transformation utilise la tactique `run_template_program` avec l'argument `tmQuoteRec I` (voir 3.4.4 pour rappel) afin d'obtenir toutes les informations nécessaires sur l'inductif `I` qui sont contenues dans l'environnement global. En particulier, nous avons besoin : du nombre $n$ et du type de ses paramètres, ainsi que du nombre $m$ et du type de ses constructeurs. Les fonctions

qui créent les énoncés réifiés dans le type `term` de `MetaCoq` ne seront pas détaillées ici : elles consistent en une manipulation fastidieuse d'indices de De Bruijn.

Nous allons cependant examiner le script `Ltac` de preuve des énoncés de non confusion et d'injectivité des constructeurs.

Pour le principe de non-confusion, c'est très simple. Une fois que l'on a sélectionné des constructeurs $c_i$ et $c_j$, montrer qu'ils sont disjoints consiste à introduire leurs arguments ainsi que l'hypothèse d'égalité absurde `H` via `intros`. La tactique `discriminate` permet de conclure, car elle a précisément la capacité de prouver n'importe quel but quand une hypothèse impossible se trouve dans le contexte. Bien sûr, nous aurions pu construire un terme de preuve, similaire à celui que nous avions écrit dans la preuve que $true \neq false$ ici en 2.1.2. Mais en faisant cela, on se prive de tactiques déjà implémentées, déjà utilisées dans de nombreuses preuves `Coq` et donc très certainement dépourvues de bugs, là où une implémentation de notre part aurait été fastidieuse et faillible.

Pour prouver l'injectivité des constructeurs : pour chaque énoncé, on introduit tous les termes de celui-ci et l'hypothèse d'égalité de la forme $c_i\ \vec{A}\ y_{i_1}\ ...\ y_{i_k} = c_i\ \vec{A}\ z_{i_1}\ ...\ z_{i_k}$ via `intros`. Ensuite, grâce à la tactique `inversion` appliquée sur l'hypothèse d'égalité, on peut récupérer automatiquement les égalités de la forme $y_{i_j} = z_{i_j}$ que l'on souhaite prouver. Notons qu'on aurait pu également utiliser `injection` suivie de `intros`.

Il suffit ensuite de prouver séparément chaque terme de la conjonction, grâce à `split` qui décompose un but de la forme `A /\ B` en deux sous-buts `A` et `B`, puis de substituer dans les égalités, qui deviennent ainsi triviales.

### 4.2.2 Le principe de génération

Ce dernier énoncé a été traité à part pour éliminer les quantificateurs existentiels apparaissant dans la manière la plus naturelle de l'écrire. Le code de la transformation (sans quantificateurs existentiels) est disponible dans ce fichier : https://github.com/louiseddp/sniper/blob/for-phd/theories/case_analysis.v.

Le code de la transformation (avec quantificateurs existentiels) est disponible dans ce fichier : https://github.com/louiseddp/sniper/blob/for-phd/theories/case_analysis_existentials.v. Des exemples pour la version de la transformation générant un énoncé avec des quantificateurs existentiels sont disponibles au code suivant : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_case_analysis_existentials.v.

Pour éliminer ces quantificateurs, comme nous sommes en logique intuitionniste, nous n'avons pas le choix : il nous faut fournir des témoins.

Pour rappel, voici l'énoncé initial :

$\forall(\vec{A} : Type)(t :\ I\ \vec{A}), \bigvee_{i=1}^{m} \exists(\vec{y_i} : \vec{B_i}), t = c_i\ \vec{A}\ y_{i_1}\ ...\ y_{i_k}$

Or, pour chaque terme $t$ d'un type inductif $I$, étant donné le constructeur numéro $i$ et son argument numéro $j$, de type $B_{i_j}$, il est possible de définir une fonction de projection que nous appellerons $p_{i_j}$. Celle-ci renvoie soit l'argument $j$ du constructeur $i$ si jamais $t$ est de la forme $c_i\ y_{i_1}\ ...\ y_{i_k}$, soit une valeur par défaut sinon.

En `Coq`, nous écrivons cette définition ainsi :

```
Definition pij :=
  fun (A1: Type) ... (An: Type) (default: Bij) (x : I A1 ... An)=>
  match x with
  | c1 _ ... _ => default
  ...
  | ci _ ... yij ... _ => yij
```

```
...
| cm _ ... _ => default
end.
```

Les fonctions de projection seront construites d'abord en `MetaCoq` puis dé-réifiées pour les ajouter au contexte local (en tant que définitions locales).

Grâce à ces fonctions de projection, l'énoncé de génération *avec témoins* qui sera produit par la transformation est le suivant :

$\forall(\vec{A} : Type)(t :\ I\ \vec{A}), \bigvee_{i=1}^{m} t = c_i\ \vec{A}\ (p_{i_1}\ t)\ ...\ (p_{i_k}\ t)$

Pour les listes (pour l'exemple nous avons choisi les listes d'entiers relatifs), l'énoncé obtenu sera similaire à la variable de section `gen_principle` définie à l'exemple 27, modulo le fait qu'au lieu d'utiliser `tl`, nous aurons `tl_default` qui renvoie une valeur par défaut donnée en paramètre au lieu de la liste vide dans le cas `nil`.

L'énoncé est donc :

$\forall(A :\ Type)(l' :\ list\ A)(d :\ A)(l : list\ A),\ l = [] \vee l = hd\ d\ l\ ::\ tl_{default}\ l'\ l$

Des exemples sont disponibles au code suivant :`https://github.com/louiseddp/sniper/blob/for-phd/examples/example_case_analysis.v`.

Par ailleurs, un post-traitement particulier est effectué après la production de cet énoncé. En effet, dans l'état actuel de `Sniper`, cet énoncé est monomorphisé indépendamment de la transformation de monomorphisation (voir 4.4), de façon à pouvoir immédiatement instancier les quantificateurs suivants par des valeurs par défaut des types monomorphes.

Comme la classe de type `CompDec` possède un habitant canonique (voir 2.2.3), cet habitant sera la valeur par défaut recherchée.

Si l'instanciation par des valeurs par défaut n'est pas faite, nous avons remarqué que le prouveur automatique peut ne pas réussir à utiliser l'énoncé et donc ne résout pas certains buts, car il ne parvient pas à instancier correctement les quantificateurs.

Ainsi, l'énoncé après post-traitement, si on suppose donné une instance $B : Type$ et une valeur par défaut $b : B$, deviendra :

$\forall(l : list\ B),\ l = [] \vee l = hd\ b\ l\ ::\ tl_{default}\ []\ l$

Cependant, effectuer ce post-traitement très spécifique va à l'encontre de notre méthodologie qui se fonde sur des transformations atomiques. C'est pour cette raison que nous souhaiterions améliorer la manière dont l'énoncé de génération est post-traité, comme expliqué dans les perspectives en 6.4.1.

## 4.3 Relations inductives

### 4.3.1 Les relations inductives

Les relations inductives sont des types inductifs dont le codomaine est `Prop`. Elles permettent de définir des propriétés sur des termes via leurs constructeurs : à chacun d'entre eux correspond *une règle d'inférence*. Nous parlerons aussi de « prédicat inductif » pour des relations inductives à un seul argument.

Dans les relations inductives que nous considérerons, les arguments de type $P :\ Prop$ d'un constructeur seront appelées prémisses du constructeur en question, et le codomaine de son type de retour sera appelé conclusion du constructeur (ce codomaine a également le type $Prop$).

**Exemple 28** [Parité d'un entier naturel unaire]

Si nous voulions définir le prédicat `even` sur papier, nous aurions besoin de deux règles :

$$\frac{}{\Gamma \vdash even\ 0}\ \texttt{ev0}$$

$$\frac{\Gamma \vdash n : \texttt{nat} \quad \Gamma \vdash even\ n}{\Gamma \vdash even\ (S\ (S\ n))}\ \texttt{evRec}$$

La première dit que l'entier zéro est pair, la seconde assure que si un entier $n$ est pair, le successeur de son successeur $(S\ (S\ n))$ l'est aussi.

En `Coq`, ceci correspond au prédicat inductif ci-dessous :

```
Inductive even : nat -> Prop :=
| ev0 : even 0
| evRec : forall (n: nat), even n -> even (S (S n)).
```

Le premier constructeur implémente bien la règle `ev0` et le second la règle `evRec`. Implicitement, ces deux écritures signifient également qu'il n'existe pas d'autres manières d'obtenir un nombre pair que par l'application d'une de ces deux règles.

Le premier constructeur n'a pas de prémisse et a pour conclusion `even 0`.

Le second constructeur a pour prémisse `even n` et pour conclusion `even (S (S n))`.

Les relations inductives sont omniprésentes dans les développements `Coq`. Elles permettent à l'utilisateur·ice d'exprimer de façon claire et dans une syntaxe agréable des propriétés sur les objets sur lesquels il ou elle travaille. Des tactiques très utiles fournies par `Coq` (telles que `inversion`, `constructor`, `induction`...) permettent d'exploiter facilement les informations contenues par ces relations inductives lors des preuves. Utiliser des relations inductives est sans doute la manière la plus naturelle d'écrire des spécifications en `Coq`. Pour ces raisons, il était très important qu'une partie du code de `Sniper` concerne ces relations inductives.

### 4.3.2 Décidabilité

Il existe des relations inductives bien plus complexes que `even`. Notamment, nous pouvons définir en `Coq` les termes du Système F (une logique du second ordre avec du polymorphisme et de l'imprédicativité), ainsi que ses règles de typage. Or, celles-ci s'expriment naturellement par une relation inductive ternaire[3]. Notons cette relation `type_assignment` : ainsi, `type_assignment Γ T t` représente la proposition $\Gamma \vdash t : T$ dans le Système F (à la Curry, c'est-à-dire sans annotation de type pour les variables).

Contrairement au prédicat `even`, où nous pourrons, pour tout entier $n$, obtenir une preuve de sa parité ou de la négation de celle-ci, nous ne pourrons pas, étant donné un terme $t$, un type $T$ et un environnement $\Gamma$ quelconque du Système F à la Curry, obtenir un habitant de `type_assignment Γ T t` ou de `type_assignment Γ T t -> False`. En d'autres mots, la relation `even` est *décidable* et la relation `type_assignment` est *indécidable* [46].

Le fait d'être décidable en `Coq` signifie qu'il existe une procédure de décision, c'est-à-dire un algorithme (qu'on peut écrire avec une fonction `Coq`) qui, étant donné un entier $n$, fournit un terme de preuve de sa parité ou de son imparité. Contrairement à la disjonction « \/ » qui n'est *pas informative* (car à partir de celle-ci nous ne pourrons pas construire un terme du type de son premier argument ou un terme du type de son deuxième), le « ou » dont nous parlons est une disjonction *informative*. En `Coq`, elle se note « + » (ou `sumbool`), c'est un inductif à deux

3. Pour en savoir plus, voir le fichier `Coq` : https://github.com/uds-psl/coq-library-undecidability/blob/coq-8.18/theories/SystemF/SysF.v qui fait partie d'une librairie de problèmes indécidables.

constructeurs `left` et `right`. Ainsi, dire que `even` est décidable signifie que pour tout `n`, il existe une preuve de `{even n} + {even n -> False}`.

Le type inductif `+` a pour codomaine `Type`, mais il est très facile de faire correspondre ses habitants à des booléens, comme le montre la fonction suivante.

```
Definition sumbool_to_bool (A B : Prop) (x : {A} + {B}) : bool :=
match x with
| left _ => true
| right _ => false
end.
```

Cette fonction efface la preuve de `A` ou de `B` contenue dans le terme `x` et ne garde que l'information de s'il s'agissait d'une preuve de `A` ou de `B`. Nous pouvons faire de même pour le prédicat `even`, et choisir de décider ce prédicat par une fonction à valeurs dans dans `bool` plutôt que dans `sumbool`.

Voici donc la fonction en question, dont on peut prouver (avec un principe d'induction forte) qu'elle est équivalente au prédicat `even`.

```
Fixpoint even_dec (n : nat) : bool :=
  match n with
    | 0 => true
    | 1 => false
    | S (S n') => even_dec n'
  end.
```

Le lemme de correction sera le suivant :

```
Lemma even_decidable : forall n, even n <-> even_dec n = true
```

Ce lemme signifie qu'à chaque fois que `even n` est vrai, sa version booléenne l'est aussi (on dira que `even_dec` est *complète* relativement à `even`), et à chaque fois que `even_dec n` est vraie, `even n` est vrai (on dira que `even_dec` est *sûre* relativement à `even`).

Pourquoi avons-nous parlé de décidabilité ? Tout simplement parce que, avec `SMTCoq`, qui utilise une logique booléenne, nous ne pouvons guère espérer prouver quoi que ce soit au sujet de relations qui ne sont pas décidables sans ajouter l'axiome du tiers exclu (`forall (A: Prop) , A \/ ~ A`), qui rendrait la logique de `Coq` classique.

Or, nous ne souhaitons pas réduire *a priori* le nombre d'utilisateur·ices potentiels de `SMTCoq` parce que le plugin admettrait des axiomes qui seraient inutilement trop forts, voire incompatibles avec d'autres axiomes requis par certain·es d'entre elles et eux. Par ailleurs, ajouter des axiomes dans ses fichiers `Coq` conduit à bloquer les calculs : il n'est pas possible d'extraire des informations de ceux-ci.

Au lieu d'utiliser des axiomes, je me suis donc concentrée sur les relations inductives décidables. Il s'agit de relations que l'utilisateur·ice a défini, pour des raisons de convenance, dans `Prop`, mais qu'il ou elle aurait tout à fait pu définir par une fonction à valeurs booléennes équivalente.

Même avec ces relations inductives décidables, `SMTCoq` est inutilisable sans un gros prétraitement préalable. Des buts complètement triviaux résistent à l'outil.

> **Exemple 29** [Zéro est pair] La preuve `Coq` de la parité de zéro, `even 0`, n'utilise qu'une seule tactique : `Proof. constructor. Qed.`
>
> En effet, le constructeur `ev0` a bien le type recherché `even 0`. Mais, en essayant de substituer la tactique `verit` à `constructor`, nous obtenons un message d'erreur :

`Failure("Verit.tactic: can only deal with equality over bool")`. Avec `SMTCoq`, nous ne pouvons rien prouver qui n'ait pas la forme d'une égalité entre booléens.

Mais si l'utilisateur·ice avait choisi de définir la notion de parité d'un entier naturel positif plutôt avec `even_dec`, `verit` aurait réussi à prouver `even_dec 0` (à condition d'utiliser les tactiques de `Sniper` sur les fonctions décrites en 4.1).

### 4.3.3 Présentation de la commande `Decide`

Afin de résoudre ce problème, `Sniper` propose une commande vernaculaire `Decide`, qui permet, dans des cas simples, de générer automatiquement la fonction booléenne équivalente à une relation inductive décidable donnée. Des heuristiques écrites en `Ltac2` permettent parfois de générer la preuve de correction nécessaire, mais, si celles-ci échouent, la preuve sera demandée à l'utilisateur·ice.

Le code de la commande `Decide` est disponible dans le sous-dossier suivant : `https://github.com/louiseddp/sniper/blob/for-phd/theories/deciderel`

Contrairement à ce que nous avons fait jusque là, nous avons préféré implémenter cette transformation sous la forme d'une commande plutôt que d'une tactique parce que nous sommes parti·es du principe que l'utilisateur·ice qui a écrit une relation inductive `R` souhaitera ajouter une fois pour toutes à l'environnement global de `Coq` son équivalent booléen, puisqu'il ou elle va très certainement écrire plusieurs preuves à son sujet. L'autre option aurait été d'utiliser, à chaque preuve, une tactique qui poserait une définition locale du point fixe décidant `R` et ajouterait au contexte le lemme d'équivalence, mais cela aurait été beaucoup plus inefficace. Un travail futur pour améliorer `Sniper` serait de proposer les deux possibilités, car disposer d'une version « tactique » de `Decide` donnerait à l'utilisateur·ice moins de travail : il ou elle n'aurait même pas à écrire la commande `Decide` ! La traduction des relations inductives serait faite de manière complètement transparente au sein de la tactique `snipe`.

La commande `Decide` fonctionne en plusieurs étapes :

1. Les relations inductives subissent un *prétraitement* de manière à *linéariser* la conclusion de leurs constructeurs. Autrement dit, le codomaine du type de chacun des constructeurs ne doit pas mentionner deux fois la même variable. Ce prétraitement facilite ensuite la génération du point fixe, car, celui-ci utilisant du *pattern matching*, les motifs sont soumis à une condition de linéarité.

2. Le point fixe est généré, il s'agit d'une analyse de cas constructeur par constructeur. Nous verrons surtout comment j'ai géré les indices de De Bruijn en `MetaCoq`. En effet, l'idée de la transformation est simple mais sa réalisation ne l'est pas, à cause de la gestion des variables. Par ailleurs, un gros travail a été de se familiariser avec la `TemplateMonad` de `MetaCoq`, indispensable pour l'écriture de commandes dans ce métalangage.

3. Nous parlerons enfin de la preuve d'équivalence entre l'inductif initial et le point fixe à valeurs booléennes qui le décide. Nous avons choisi d'écrire des heuristiques en `Ltac2` car cette solution, bien qu'incomplète, a le mérite d'être beaucoup plus rapide à implémenter et beaucoup plus simple que la génération d'un terme de preuve.

Une alternative à la commande `Decide`, pensée pour les cas où nous aurions besoin d'un prétraitement pour un backend intuitionniste, est de générer les principes d'inversion pour ces relations inductives, nous verrons donc que `Sniper` propose cette petite transformation additionnelle.

### 4.3.4 Prétraitement : linéarisation

Afin d'éclairer les raisons pour lesquelles nous avons besoin de linéariser les relations inductives avant d'essayer de les transformer en point fixe, considérons un exemple.

**Exemple 30** Appartenance d'un entier à une liste

```
Inductive mem : nat -> list nat -> Prop :=
| MemMatch : forall (ns : list nat) (n : nat), mem n (n :: ns)
| MemRec : forall (ns : list nat) (n n' : nat), mem n ns -> mem n (n'
    :: ns).
```

La relation binaire `mem n l` est vraie lorsque `n` appartient à la liste `l`. Nous voudrions pouvoir décider la relation, et pour cela nous allons procéder à une analyse de cas sur la liste, et vérifier qu'elle a une forme qui correspond à l'un des constructeurs de la relation inductive. La première fonction (incorrecte), que l'on voudrait générer automatiquement, serait celle-ci :

```
Fail Fixpoint mem_dec (n : nat) (l : list nat) :=
match l with
| [] => false
| n :: _ => true
| n' :: ns => mem_dec n ns
end.
(* The command has indeed failed with message:
Pattern "n' :: ns" is redundant in this clause. *)
```

`Coq` pense que nous avons écrit un motif redondant, parce que nous avons deux cas où la liste n'est pas vide. La variable `n` dans le motif `n :: _` est en effet une nouvelle variable locale qui masque la variable donnée en argument de la fonction. Ce n'est pas ce que l'on souhaiterait, nous voudrions que le motif capture la première variable `n`. Or, ce n'est pas possible en `Coq` : toutes les variables dans les motifs sont fraîches. Dans le noyau de `Coq`, les branches des *pattern matchings* sont en effet des fonctions qui lient chacun des arguments des constructeurs, la seule possibilité est donc de représenter la tête de la liste par une variable fraîche, dont on testera par la suite si elle est égale à `n`.

Cet exemple montre que nous avons besoin que les conclusions des constructeurs (c'est à dire le codomaine du type de ceux-ci) soient *linéaires*. Nous n'avons pas le même souci pour les prémisses parce que nous n'avons pas besoin de faire du *pattern matching* sur celles-ci.

Si nous avions, à la place de `mem`, le prédicat linéaire équivalent suivant, nous pourrions effectivement générer le point fixe :

```
Inductive mem_linear : nat -> list nat -> Prop :=
  | MemMatch_linear : forall (ns : list nat) (n n' : nat), Nat.eqb n n' = true
      -> mem_linear n (n' :: ns)
  | MemRec_linear : forall (ns : list nat) (n n' : nat), mem_linear n ns ->
      mem_linear n (n' :: ns).
```

Et le point fixe serait alors :

```
Fixpoint mem_dec (n : nat) (l : list nat) :=
match l with
| [] => false
```

```
| n' :: ns => Nat.eqb n n' || mem_dec n ns
end.
```

Nous voyons cependant que cette linéarisation ne fonctionnerait pas si nous avions une relation inductive polymorphe `mem_poly` définie ainsi :

```
Inductive mem_poly (A: Type): A -> list A -> Prop :=
| MemMatch_poly : forall (xs : list A) (x : A), mem_poly x (x :: xs)
| MemRec_poly : forall (xs : list A) (x x' : A), mem_poly x xs -> mem_poly x
    (x' :: xs).
```

La raison est que les égalités entre types ne sont pas toutes décidables. Nous n'avons aucun espoir de trouver une version booléenne de l'égalité entre deux termes de types arbitraire. Le prédicat `mem_poly (nat -> nat)` n'est pas décidable pour cette raison. Mais puisque pour utiliser `SMTCoq`, nous devons de toute manière rajouter ou inférer une condition de décidabilité sur les types (plus précisément, les types doivent être `CompDec` comme vu en 2.2.3), nos preuves sur des types polymorphes sont déjà limitées par cette condition. J'ai donc décidé de proposer une gestion des relations inductives admettant des variables de type `Type` comme premiers paramètres, vérifiant la propriété `CompDec`.

Ainsi, cette transformation de *linéarisation* procède en trois étapes. Pour la description de ces étapes, on note $Cd$ le prédicat `CompDec`, et on notera $t \doteq_{A,HA} u$ lorsque $t$ et $u$ sont égaux selon l'égalité définie dans la classe de type $Cd\ A$, avec $HA : Cd\ A$.

1. Comme nous souhaitons pouvoir gérer des relations inductives avec du polymorphisme paramétrique, il faut générer, à partir de relations inductives polymorphes, une relation inductive avec un prédicat (arbitraire) sur les variables de type. Ce prédicat sera instancié par la suite.

   Ainsi, à partir de $R : \forall (A_1 : Type),\ ...(A_n : Type),\ u$, la première étape permet de déclarer dans l'environnement global, étant donné un prédicat $P : Type \to Type$, l'inductif $R_P : \forall (A_1 : Type)(HA_1 : P\ A_1)\ ...(A_n : Type)(HA_n : P\ A_n),\ u$.

   Les constructeurs de $R_P$ sont obtenus en remplaçant chaque occurrence de $R$ par $R_P$ et en lui ajoutant les $HA_i$ attendus en argument.

2. Ce prédicat $P$ est ensuite instancié par `CompDec`. Pour tous les types concrets qui apparaissent dans la relation inductive, une preuve qu'ils sont bien une instance de la classe de type `CompDec` est recherchée. En effet, pour rappel, `SMTCoq` propose déjà quelques preuves que des types de base sont `CompDec`. Par exemple, `SMTCoq` dispose d'un terme `compdec_list` de type `forall (A : Type), CompDec A -> CompDec (list A)`.

3. Enfin, l'étape de linéarisation proprement dite est effectuée. Introduisons des notations. Soit une relation inductive :

   $R_{Cd} : \forall (A_1 : Type)(HA_1 : Cd\ A_1)\ ...(A_n : Type)(HA_n : Cd\ A_n)(p_1 : B_1)\ ...\ (p_m : B_m), I_1 \to ... \to I_k \to Prop$

   Les $p_i$ sont des *paramètres* de type les $B_i$, qui peuvent être soit des $A_i$ soit des types inductifs (de type $Set$), pouvant avoir pour paramètres certains des $A_i$. Par ailleurs, on se donne des preuves $h_{B_1},\ ...,\ h_{B_m}$ de type $Cd\ B_1,\ ...Cd\ B_m$ respectivement, dans le contexte $\Gamma := A_1 : Type,\ A_1 : Cd\ A_1,\ ...,\ A_n : Type,\ HA_n : Cd\ A_n$.

   De même, les éléments de type les $I_i$ sont des annotations, au sens de 2.1.2, qui sont nécessairement des types inductifs de type $Set$ pouvant avoir pour paramètres certains des $A_i$. On se donne aussi des preuves $h_{I_1},\ ...,\ h_{I_k}$ de type $Cd\ I_1,\ ...Cd\ I_k$ dans le contexte $\Gamma$.

   On suppose que la relation $R_{Cd}$ a $j$ constructeurs, de la forme :

   $c_i : \forall (x_{i_1} : J_1),\ ...,\ (x_{i_l} : J_l), pr_1 \to ... \to pr_r \to R_{Cd}\ A_1\ HA_1 ...\ A_n\ HA_n\ p_1\ ...\ p_m\ t_{c_{i_1}}\ ...\ t_{c_{i_k}}$

Chaque $t_j$ est d'un type algébrique complètement appliqué et qui fait partie de la classe de type `CompDec`, tout comme les $x_{i_j}$. De plus, les $t_j$ ne peuvent contenir que des variables parmi les $x_{i_j}$, des paramètres de la relation $R_{Cd}$, des termes clos, des constructeurs des $I_j$ ou des $J_j$ ou des symboles de fonctions (autres que des constructeurs) à arguments et valeurs dans les $I_j$ et les $J_j$.

Dans l'étape de linéarisation, on considère chaque $t_{c_{i_1}}$, ... $t_{c_{i_k}}$ de la conclusion du constructeur $c_i$. On raisonnera ici sur $t_{c_{i_q}}$ avec $q \in \{1, ..., k\}$. On commence l'algorithme en se donnant l'ensemble $E$ de variables, parmi les $x_{i_j}$, déjà rencontrées dans les autres arguments de la conclusion (les autres $t_{c_{i_j}}$), et en considérant le symbole de tête de $t_{c_{i_q}}$. Nous avons cinq cas selon le terme examiné :

— Soit c'est un constructeur $c_{I_v}$ de $I_q$, dans ce cas on reprend l'algorithme pour chaque argument du constructeur $c_{I_v}$ qui n'est pas en position de paramètre pour celui-ci.

— Soit c'est un paramètre $p$ de $R_{Cd}$ parmi les $p_j : B_j$. Dans ce cas on substitue à $p$ une variable fraîche $x : B_j$ dans $t_{c_{i_q}}$ et on ajoute aux prémisses de notre constructeur l'égalité $x \doteq_{B_j,HB_j} p$.

— Soit c'est une variable $x$ *parmi les* $x_{i_j} : J_j$. Si $x \notin E$ on reprend l'algorithme soit avec $t_{c_{i_{q+1}}}$ s'il ne reste plus d'autres termes à traiter, soit avec les termes restants (notamment si on examinait les différents arguments d'une application) et avec $E \cup \{x\}$. Si $x \in E$ alors on substitue à $x$ une variable fraîche $x' : B_j$ dans $t_{c_{i_q}}$ et on ajoute aux prémisses de notre constructeur $c_i$ l'égalité $x' \doteq_{J_j,HJ_j} x$ avant de poursuivre l'algorithme.

— Soit c'est une fonction $f$ à $v$ arguments, dans ce cas elle est appliquée à des termes $t_{f_1}$, ..., $t_{f_v}$. Tout d'abord, on substitue à $f\ t_{f_1}\ ...\ t_{f_v} : C_f$ (avec $C_f$ un $I_j$ ou un $J_j$) une variable fraîche $x' : C_f$, et on ajoute dans les prémisses de notre constructeur $c_i$ l'égalité $x' \doteq_{C_f,HC_f} f\ t_{f_1}\ ...\ t_{f_v}$, puis on recommence à appliquer l'algorithme sur chacun des $t_{f_1}$, ..., $t_{f_v}$.

— Soit c'est un terme clos et dans ce cas on ne fait rien, on passe au terme suivant.

Il faut appliquer le même algorithme sur chaque constructeur séparément, pour obtenir une relation inductive avec une conclusion linéaire.

Prenons un exemple un peu plus complexe que `mem` pour illustrer comment s'applique l'algorithme de linéarisation.

**Exemple 31** La relation inductive `Add`

Soit la relation inductive suivante :

```
Inductive Add (A: Type) (a: A) : list A -> list A -> Prop :=
| Add_head l : Add A a l (a::l)
| Add_cons x x' l l' : Add A a l l' -> Add A a (x::l) (x::l').
```

Intuitivement, `Add A a l l'` est vraie lorsque la liste `l'` est exactement la liste `l`, avec l'élément `a` en plus quelque part.

La première étape permet d'obtenir l'inductif suivant, étant donné `P: Type -> Type`.

```
Inductive Add_P (A: Type) (HA: P A) (a: A) : list A -> list A ->
    Prop :=
| Add_head_P l : Add_P A HA a l (a::l)
| Add_cons_P x x' l l' : Add_P A HA a l l' -> Add_P A HA a (x::l) (x
    ::l').
```

La deuxième étape remplace `P` par `CompDec`, et on obtient :

```
Inductive Add_CompDec (A: Type) (HA: CompDec A) (a: A) :
  list A -> list A -> Prop :=
| Add_head_CompDec l : Add_CompDec A HA a l (a::l)
| Add_cons_CompDec x x' l l' : Add_CompDec A HA a l l' ->
  Add_CompDec A HA a (x::l) (x'::l').
```

Pour la troisième et principale étape, considérons d'abord le second constructeur `Add_cons_CompDec` de `AddCompDec`. On remarque que, parmi les arguments de sa conclusion qui ne sont pas des paramètres (`x::l` et `x'::l'`), les variables `x`, `x'`, `l` et `l'` ne sont jamais utilisées deux fois. Quand on examine les paramètres `A HA` et `a`, on trouve que `A` est présent de manière implicite en tant que paramètre de `_::_`. En effet, `x::l` peut s'écrire `@cons A x l`. Mais comme `A` est en position de paramètre pour `cons`, il n'y a rien à faire de particulier. En résumé, après analyse, ce constructeur n'est pas modifié.

Considérons ensuite le constructeur `Add_head_CompDec`. Le premier argument de sa conclusion qui n'est pas un paramètre est `l`. On ajoute `l` à l'ensemble des variables déjà vues. Le second argument de sa conclusion qui n'est pas un paramètre est `a::l`, ou encore `@cons A a l`. Parmi les arguments de `cons`, le premier est un paramètre donc on ne le regarde pas, le deuxième est `a`, un paramètre de `AddCompDec`, mais pas un paramètre de `cons`. On rajoute donc une variable `a'` et l'égalité `eqb_of_compdec A HA a a'` (voir 2.2.3), et on remplace `a` par `a'`. Le troisième argument de `@cons A a l` est `l`, variable que l'on a déjà rencontrée. On rajoute donc une variable `l'` et l'égalité `eqb_of_compdec (list A)(compdec_list HA)l l'`, et on remplace `l` par `l'`.

Finalement, on obtient l'inductif suivant :

```
Inductive Add_CompDec_linear (A: Type) (HA: CompDec A) (a: A) : list
     A -> list A -> Prop :=
| Add_head_CompDec_linear a' l l' : eqb_of_compdec A HA a a' ->
  eqb_of_compdec (list A) (compdec_list HA) l l'->
  Add_CompDec_linear A HA a l (a'::l')
| Add_cons_CompDec_linear x x' l l' : Add_CompDec_linear A HA a l l'
     ->
  Add_CompDec_linear A HA a (x::l) (x'::l').
```

Après tous ces prétraitements, nous sommes en mesure de générer le point fixe.

### 4.3.5 Génération du point fixe

Dans ce paragraphe, nous ajoutons une nouvelle condition aux relations inductives que nous traitons. En conservant les mêmes notations que ci-dessus, remarquons que les $t_{c_{i_1}}$ ... $t_{c_{i_k}}$ de la conclusion du constructeur $c_i$ contiennent parmi leur sous-termes certaines des variables $x_{i_1}$, ..., $x_{i_l}$. Ces variables forment donc un ensemble noté $Sub(concl_{c_i})$, tel que : $Sub(concl_{c_i}) \subseteq \{x_{i_1}, ..., x_{i_l}\}$,

On peut définir un ensemble similaire pour chaque prémisse $pr_{i_j}$, qu'on note alors $Sub(pr_{i_j})$.

Nous ne pourrons traiter que les relations inductives linéarisées qui vérifient la condition : $\forall i_j,\ Sub(pr_{i_j}) \subseteq Sub(concl_{c_i})$.

Plus concrètement, cette condition signifie que les prémisses d'un constructeur ne peuvent pas utiliser un argument qui n'est pas mentionné dans la conclusion de celui-ci.

Le problème si nous omettons cette condition sera expliqué grâce à l'exemple suivant. Notons que celui-ci est tiré d'un article qui extrait des procédures de *semi-décision* à partir de relations inductives [35], dont nous reparlerons en 4.3.8.

**Exemple 32** Le lambda-calcul simplement typé

Définissons les types, les termes et l'environnement de typage d'un lambda-calcul simplement typé en `Coq`, utilisant les indices de De Bruijn pour représenter les variables.

```
Inductive type : Set :=
| Iota : type
| Arr : type -> type -> type.

Inductive term : Set :=
| Var : nat -> term
| Lam : type -> term -> term
| App : term -> term -> term.

Definition env := list type.

Inductive typ : env -> term -> type -> Prop :=
| typ_var : forall (e: env) (n: nat) (A: type), nth_error e n =
    Some A -> typ e (Var n) A
| typ_lam : forall (e: env) (t: term) (A B: type), typ (A::e) t B
    -> typ e (Lam A t) (Arr A B)
| typ_app : forall (e: env) (t u: term) (A B: type), typ e t (Arr
    A B) ->
  typ e u A -> typ e (App t u) B.
```

Par souci de concision, nous supposons que nous avons déjà démontré `Hterm : CompDec term` et `Htype : CompDec type`, et que nous avons également `Hopt : CompDec (option term)` qui découle directement de `Hterm`. La transformation de la relation inductive en point fixe se fait naturellement (après linéarisation pour le cas `typ_lam`) pour les deux constructeurs `typ_var` et `typ_lam`.

```
Fixpoint typ_dec e t A {struct t} :=
match t with
| Var n => eqb_of_compdec (option term) Hopt (nth_error e n) A
| Lam A' t' =>
  match A with
  | Arr A1 A2 =>
    eqb_of_compdec type Htype A1 A' && typ_dec (A1::e) t' A2
  | _ => false
  end
| App t1 t2 => ???
end.
```

Mais le cas de l'application pose problème : puisque la variable `A` n'apparaît que dans les prémisses, elle n'est pas encore liée dans le corps du point fixe et il n'est donc pas possible d'écrire : `typ_dec e t1 (Arr A B)&& typ e t2 A`. Nous aurions besoin d'*inférer* le type de `t2` pour ensuite *vérifier* le type de `App t1 t2`.

Or, ce problème dépasse le cadre de notre transformation, même si nous avons ici une piste d'amélioration (en s'inspirant du typage bidirectionnel). On pourrait en effet

imaginer créer, à partir de la relation initiale, des relations inductives mutuelles de manière à passer du mode *vérification* au mode *génération* quand nécessaire, pour ensuite obtenir un point fixe mutuel, qui décide la relation ou produit l'un de ses arguments.

Ceci étant précisé, nous pouvons passer à l'algorithme de génération du point fixe. Il se décompose en deux étapes principales :

1. La génération du point fixe proprement dite, sans s'assurer de la terminaison.
2. La recherche d'un argument sur lequel la fonction générée effectue une récursion structurelle.

Commençons par la seconde étape, parce que c'est la plus simple. Nous avons choisi une heuristique pour trouver automatiquement l'argument qui décroît au sein des appels successifs du point fixe. Dans le cas où cette heuristique ne fonctionne pas, la commande `Decide` demande à l'utilisateur de fournir lui-même la position de cet argument. L'idée est de scanner tous les constructeurs de la relation inductive initiale, et de repérer ceux où la relation apparaît dans les prémisses. Nous les appellerons « constructeurs récursifs ». En effet, ils correspondront aux appels récursifs dans le point fixe.

Ensuite, parmi les arguments (qui ne sont pas des paramètres et que nous avions numérotés de 1 à $k$ dans l'item 3 de l'algorithme de linéarisation en 4.3.4) donnés à la relation en conclusion des « constructeurs récursifs », nous en cherchons un (mettons, le $j$-ème) qui est de la forme $C\ t$, avec $C$ un constructeur du type inductif $I_j$, pour tous les « constructeurs récursifs ».

**Exemple 33** Retour de la relation `Add`

Reprenons notre relation `Add` linéarisée comme à l'exemple 31. Le seul constructeur pour lequel nous aurons un appel récursif est le second, `Add_cons_CompDec_linear`. Or, dans ce cas-là, la conclusion est : `Add_CompDec_linear A HA a (x::l)(x'::l')`. Le premier et le second argument, une fois les paramètres `A`, `HA` et `a` exclus, sont tous les deux de la forme `cons _ _`, donc sous un constructeur. Cela signifie que la récursion structurelle peut se faire soit sur l'un, soit sur l'autre.

Commençons la génération du point fixe en nous donnant une variable libre $f$ pour représenter celui-ci, des variables $x_1$, ..., $x_{2n+m}$ pour les paramètres ($n$ paramètres de type $Type$, $n$ hypothèses de décidabilité et $m$ paramètres de types algébriques ou de type une variable parmi les $\{x_1, ..., x_n\}$, pour rappel). Nous nous donnons également des variables $y_1$, ..., $y_k$ pour représenter les annotations de la relation inductive. Ce sont ces dernières qui vont nous intéresser le plus, les autres seront simplement liftées au fil de l'exécution de notre programme. Au départ, nous considérons donc un terme de la forme $f\ x_1\ ...\ x_{2n+m}\ y_1\ ...\ y_k$. À chaque position d'argument $j$, nous associons un motif (le motif « variable ») qui peut s'unifier avec n'importe quelle autre variable.

Pour savoir si un des constructeurs de la relation inductive peut s'appliquer, et donc, pour savoir si le point fixe doit renvoyer *true*, nous allons examiner la forme de la conclusion des constructeurs de la relation inductive. Si jamais nous trouvons un $t_{i_j}$ dont le symbole de tête est un constructeur du type $I_j$, nous savons qu'il nous faut effectuer un *pattern matching* sur la variable $y_j$. Le motif « variable » associé à la position $j$ sera donc remplacé par un nouveau motif, dépendant de la branche dans laquelle nous nous trouvons. Par exemple, si à la position $j$, la relation attend un terme de type *nat*, le motif « variable » sera remplacé par le motif $O$ dans la première branche du *pattern matching*, et par le motif $S\ var$, où $var$ est un motif « variable », dans la seconde branche.

Après ce premier *pattern matching*, soit :

1. le nouveau motif correspond bien, c'est-à-dire *s'unifie*, avec le motif de $t_{i_j}$ et dans ce cas il nous faut tester les autres arguments de la conclusion du même constructeur (les $t_{i_{j'}}$ pour $j' \neq j$) pour voir s'ils correspondent aussi au bon motif associé, ou s'il faut également effectuer un *pattern matching* sur une ou plusieurs variables $y_{j'}$ ;
2. soit le motif ne s'unifie pas et dans ce cas, nous savons que le constructeur $i$ en question ne pourra pas s'appliquer dans cette branche du *pattern matching* ;
3. soit il faut faire un nouveau *pattern matching*, sur une des variables fraîches introduites dans la branche dans laquelle nous sommes.

Une fois qu'il existe un constructeur $i$ tel que, pour tout $j$, les motifs $j$ créés par des *pattern matchings* successifs s'unifient avec $t_{i_j}$, on sait que la conclusion du constructeur $i$ peut s'appliquer. On demande alors à ce que la conjonction booléenne des prémisse de ce constructeur soit vraie (ou alors, on renvoie *true* s'il n'y a pas de prémisse). Il faut bien sûr s'être assuré au préalable que toutes les prémisses peuvent être elles aussi traduites par un équivalent booléen.

L'unification nous a permis de faire correspondre les indices de De Bruijn des variables présentes dans le point fixe que nous souhaiterions générer aux indices de De Bruijn des variables du constructeur en question dans la relation inductive, il faut donc utiliser cette correspondance pour que les prémisses mentionnent les bons indices de De Bruijn.

En même temps, quand une variable $y_j$ est matchée, même si le motif créé s'unifie avec un certain $t_{i_j}$, il faut toujours tester s'il ne s'unifie pas également avec un certain $t_{i'_j}$ pour $i' \neq i$. En effet, si c'est le cas, la conclusion de ce constructeur $i'$ peut elle aussi s'appliquer dans cette branche du *pattern matching*. Si deux constructeurs de la relation inductive ou plus peuvent s'appliquer dans une même branche, cette branche prendra la forme d'une disjonction booléenne.

**Exemple 34** Un exemple sans paramètre

Pour faire simple, nous expliquerons le code permettant la génération du point fixe avec une relation inductive sans paramètre.

Prenons la relation inductive `smaller: list nat -> list nat -> Prop`, qui est vraie lorsque la première liste est plus petite que la seconde.

```
Inductive smallernat : list nat -> list nat -> Prop :=
| cons1 : forall l', smallernat [] l'
| cons2 : forall l l' x x', smallernat l l' -> smallernat (x :: l) (
    x' :: l').
```

Pour commencer, nous avons deux variables $y_1$ et $y_2$ et deux motifs « variable » associés. On a aussi que : $t_{1_1} = @nil\ nat$, $t_{1_2} = l'$, $t_{2_1} = x :: l$, $t_{2_2} = x' :: l'$.

Notre algorithme détecte que, quel que soit le constructeur choisi de `smallernat`, le premier argument de la relation n'est jamais une variable. En effet, $t_{1_1} = @nil\ nat$ et $t_{2_1} = x :: l$. Le premier *pattern matching* du point fixe sera donc sur la variable $y_1$. Ensuite, les motifs sont mis à jour : à la position 2, le motif ne change pas, mais à la position 1, on a une branche où le motif est $@nil\ nat$ et une seconde branche où le motif est $@cons\ nat\ var\ var$.

Dans la branche $@nil$, seul le premier constructeur peut s'appliquer. On vérifie alors le motif pour la position 2, qui est un motif « variable ». Comme $t_{1_2}$ est aussi une variable, l'unification est terminée. Les prémisses de `cons1` doivent être vraies, mais il n'y en a pas, donc cette branche vaut simplement *true*.

Dans la branche $@cons$, on est dans un cas où le second constructeur peut s'appliquer, mais plus le premier. Cependant, le second motif (juste une variable) ne s'unifie pas encore avec $t_{2_2} = x' :: l'$. On effectue donc un *pattern matching* sur $y_2$.

Dans la branche @*nil nat*, on renvoie *false*.

Dans la branche @*cons nat* $var_1$ $var_0$, notons que, deux nouvelles variables ayant été introduites, il faut mettre à jour les indices de De Bruijn des variables (bien que nous n'en ayons pas fait mention jusqu'ici par souci de clarté).

Le motif @*cons nat* $var_1$ $var_0$ à la position 1 devient :

@*cons nat* $var_3$ $var_2$.

Par ailleurs, les prémisses du constructeur `cons2` doivent être vraies pour qu'il s'applique. Grâce aux correspondances calculés lors de l'unification, on remplace les variables `l` et `l'` de `smallernat l l'` par celles qui correspondent dans le point fixe. Ainsi, le second argument non paramétrique de @*cons nat* $var_3$ $var_2$ pour `l` soit la variable d'indice de De Bruijn 2 et le second argument de @*cons nat* $var_1$ $var_0$, soit la variable d'indice de De Bruijn 0, pour `l'`.

Finalement, on obtient bien la fonction souhaitée :

```
Fixpoint smallernat_dec (l l' : list nat) :=
match l with
| [] => true
| x :: xs =>
  match l' with
  | [] => false
  | x :: xs' => smallernat xs xs'
  end
end.
```

### 4.3.6 Heuristiques

Nous avons pu écrire un programme `MetaCoq` qui dans certains cas, génère automatiquement un point fixe qui décide une relation inductive donnée. Cependant, il nous reste à démontrer que ce point fixe est effectivement *correct* vis-à-vis de la relation inductive initiale. Par correct, nous entendons deux propriétés :

— La complétude : à chaque fois que la relation inductive est vraie pour des arguments donnés, le point fixe doit valoir *true* étant donné ces mêmes arguments.

— La sûreté (*soundness*) : quand le point fixe vaut *true* sur certains arguments, la relation inductive doit également être vraie.

La conjonction de ces deux propriétés s'appelle donc la *correction*.

Nous n'avons pas cherché à être exhaustifs : l'écriture de termes de preuve manuellement est fastidieuse et longue. Par souci d'efficacité, et par volonté de se focaliser sur d'autres aspects dans cette thèse, nous avons préféré écrire un script de preuve en `Ltac2` qui ne fonctionne pas dans tous les cas. Quand le script ne fonctionne pas, l'utilisateur·ice peut prouver à la main les deux propriétés, ou une seule des deux si jamais la tactique associée à l'une des deux propriétés a réussi.

### 4.3.7 Exemple récapitulatif

Reprenons la relation inductive suivante :

```
Inductive mem : nat -> list nat -> Prop :=
| MemMatch : forall (ns : list nat) (n : nat), mem n (n :: ns)
| MemRec : forall (ns : list nat) (n n' : nat), mem n ns -> mem n (n' :: ns).
```

La commande à écrire dans `Sniper` pour décider cette relation sera :
`MetaCoq Run (Decide mem).`

L'inductif intermédiaire (la version linéarisée déjà écrite en 4.3.4) sera généré, ainsi que le point fixe `mem_linear_decidable`. Dans ce cas, notre heuristique de preuve `Ltac2` parvient à générer la preuve d'équivalence `forall n l, mem n l <-> mem_linear_decidable n l = true`. Comme `MetaCoq` ne propose pas de mécanisme d'exception dans la `TemplateMonad`, nous ne disposions pas d'une construction `try ... catch`, qui aurait permis d'essayer de générer la preuve d'équivalence et de l'introduire dans l'environnement global en cas de succès ou de demander à l'utilisateur·ice de l'écrire en cas d'échec. Notre solution est donc de demander à l'utilisateur·ice la preuve à chaque fois, mais d'afficher le nom du terme de preuve dans le cas où la preuve en `Ltac2` a réussi. Ici, nous supposerons que ce terme de preuve s'appelle `decidability_proof`.

Suite à la première commande, il nous faut donc écrire :

```
Next Obligation. exact decidabililty_proof. Qed.
```

Enfin, il faut que la tactique `trakt`, utilisée dans notre plugin (voir 2.2.3), connaisse cette nouvelle équivalence entre la relation (d'arité 2, d'où le 2 qui intervient dans la commande) et la fonction. La dernière commande à entrer sera :

```
Trakt Add Relation 2 mem mem_linear_decidable decidability_proof.
```

Désormais, un but mentionnant `mem` peut-être converti en un but mentionnant le point fixe équivalent.

Par exemple, la tactique `snipe` qui combine la plupart des transformations de ce chapitre sera capable de prouver le but suivant :

```
Lemma mem_imp_not_nil s l : mem s l -> l <> [].
```

Sans notre commande `Decide`, `snipe` renvoyait un message d'erreur signifiant que le prouveur automatique ne raisonne pas dans `Prop`. Désormais, un appel à `trakt` à l'intérieur de `snipe` permet de transformer ce but en `mem_linear_decidable s l = true -> l <> []`, et la suite des transformations prend le relais pour déplier la définition de `mem_linear_decidable`, interpréter le type des listes, etc, avant l'appel du prouveur automatique.

Un fichier d'exemples est consultable à l'adresse suivante : `https://github.com/louiseddp/sniper/blob/for-phd/theories/deciderel/examples.v`

### 4.3.8 Travaux connexes

Il n'est pas possible de décrire mon travail sur les relations inductives sans parler de deux travaux aux objectifs similaires, et de voir par quels aspects je les rejoins ou m'en distingue.

1. La thèse de Pierre-Nicolas Tollitte [44] : publiée en 2013, elle a conduit à l'implémentation d'un plugin, écrit en `OCaml`, qui extrait du code exécutable (donc des fonctions), à partir de relations inductives. Ce nouveau plugin a aussi donné lieu à un article avec Catherine Dubois et David Delahaye [43].

   Une notion cruciale dans cette thèse est celle de *mode* : étant donnée une relation inductive $R$ à $k$ arguments, nous pouvons choisir lesquels seront donnés en *entrée* à la fonction et lesquels seront des valeurs de *sortie* qui devront être calculées par cette fonction. Un mode est dit *complet* quand tous les arguments sont des entrées, et la valeur de retour de la fonction est donc un booléen. Ainsi, nous avons travaillé seulement sur le mode complet. L'avantage est que dans ce mode, la question du *déterminisme* de la fonction ne se pose pas. En mode non complet, la fonction extraite d'une relation inductive peut avoir plusieurs

valeurs de retour possible. En effet, si la relation `toto` a un constructeur dont la conclusion est `toto 0 1` et un autre dont la conclusion est `toto 0 2`, la fonction associée, mettons, `toto_fun` qui prendrait le premier argument en entrée et aurait pour sortie le second serait indéterministe. Que renverrait `toto_fun 0` ? 1 ou 2 ?

Au contraire, en mode complet, la seule sortie possible est *true*, que l'une ou plusieurs conclusions des constructeurs aient les mêmes arguments.

À partir d'un travail fondateur [19], Pierre-Nicolas Tollitte étend les relations inductives pouvant être transformées en code exécutable, et il est le premier à générer les fonctions en `Coq` (plutôt qu'en `OCaml`). Mais sa généralisation du premier travail sur le sujet s'applique surtout aux fonctions extraites en mode incomplet. Il a remarqué que le critère de l'article fondateur pour déceler si une relation inductive donnerait lieu à une fonction déterministe ou non était assez restrictif et il en a trouvé un plus fin. Or, comme nous l'avons souligné, le problème du déterminisme ne se pose pas en mode complet, et nous ne nous sommes intéressé·es qu'à ce mode.

Notre apport concerne la gestion partielle du polymorphisme prénexe (avec ajout d'hypothèses d'égalités décidables), la recherche automatique de l'argument sur lequel faire la récursion structurelle et plus pragmatiquement, nous proposons une version bien plus récente de l'outil. En effet, le port du plugin pour l'utiliser dans `Sniper` n'a pas été envisagé car les plugins pour `Coq` écrits en `OCaml` demandent de gros efforts de portage lorsque la version de `Coq` change. Tout réécrire dans un langage plus « robuste » (du point de vue des versions successives) nous a paru plus pertinent.

2. L'article de Zoe Paraskevopoulou, Aaron Eline et Leonidas Lampropoulos [35], qui est paru en même temps que nous écrivions notre propre plugin. Il propose des procédures de semi-décision pour des relations inductives. La force de leur article est d'exhiber la forte similarité entre les relations inductives extraites en mode complet et en mode incomplet (les vérificateurs et les générateurs), et d'utiliser les vérificateurs au sein des générateurs et vice-versa. Le problème est que leurs fonctions extraites utilisent des opérations monadiques (et donc de l'ordre supérieur), qui n'est géré que très partiellement par `Sniper`. Le corps de la fonction ne serait pas traduisible par le préprocessing offert par les autres transformations que nous avons écrites.

   Par ailleurs, leurs procédures étant des procédures de semi-décision, la question de la terminaison des fonctions n'est pas problématique pour ces auteur·ices : toutes leurs fonctions font une récursion structurelle sur un entier naturel de *fuel*. En conséquence, l'énoncé de correction produit est de la forme (en notant $R$ la relation, $f$ la fonction qui la semi-décide en mode complet, et $\vec{x}$ les arguments de la relation) :

   $\forall \vec{x},\ R\ \vec{x} \longleftrightarrow \exists (fuel :\ nat),\ f\ fuel\ \vec{x} = true$

   Or, `SMTCoq` ne gère pas les quantificateurs existentiels. Un tel énoncé serait inexploitable pour l'outil. Pour toutes ces raisons, continuer à écrire notre propre algorithme de décision nous a paru nécessaire.

### 4.3.9 Principe d'inversion

J'ai également écrit en `MetaCoq` une transformation moins complexe, générant et prouvant automatiquement le principe d'inversion d'une relation inductive. Le code est disponible ici : `https://github.com/louiseddp/sniper/blob/for-phd/theories/indrel.v`.

L'idée est que cette transformation peut être utile à un *backend* utilisant la logique intuitionniste, comme par exemple la tactique `firstorder`, qui recherche un terme de preuve en logique intuitionniste du premier ordre.

Étant donnée une relation inductive :

```
Inductive R (A1: T1) ... (An: Tn) : B1 -> ... -> Bk -> Prop :=
| c1 : forall (x11: U11) ... (x1l: U1l), p11 -> ... p1r -> R A1 ... An t11
    ... t1k
...
| cm : forall (xm1: Um1) ... (xml: Uml), pm1 -> ... pmr -> R A1 ... An tm1
    ... tmk
.
```

la transformation crée le terme suivant :
$\forall (A_1 : T_1) ... (A_n : T_n)(x_1 : B_1) ..., (x_k : B_k)$,
$R\ A_1\ ...\ A_n\ x_1\ ...\ x_k \longleftrightarrow \bigvee_{i=1}^{k} \exists\ x_{i_1}\ ...\ x_{i_l}, \bigwedge_{j=1}^{r} p_{i_j} \bigwedge_{j=1}^{k} x_i = t_{i_j}(x_{i_1}, ..., x_{i_l})$
Ce terme est ensuite prouvé par double implication :

$\rightarrow$ : on introduit dans le contexte local toutes les variables et l'hypothèse `H: R (A1: T1) ... (An: Tn)x1 ... xk` via `intros`. Puis, on effectue une inversion sur `H`, ce qui nous donnera autant de sous-cas que de constructeurs de `R`. Pour chaque sous cas, nous disposons d'hypothèses dans le contexte local qui permettent de prouver l'un des énoncés existentiel.

$\leftarrow$ : on introduit dans le contexte local toutes les variables et la grande disjonction. Puis on raisonne par cas : dans chacun des sous cas, une fois les quantificateurs existentiels éliminés et les témoins correspondants obtenus via la tactique `destruct`, on peut remplacer les `xi` par les `tij` dans le but, grâce aux égalités de la forme `xi=tij` présentes dans les hypothèses. Cela permet d'utiliser la tactique `constructor`, car le but aura alors exactement la forme de la conclusion du constructeur `j` de `R`. Il restera à utiliser la tactique `assumption` autant que nécessaire (pour chaque prémisse du constructeur `j`, déjà présente dans nos hypothèses).

**Exemple 35** La relation inductive `Add`
Reprenons l'exemple de la relation inductive `Add` définie à l'exemple 31. La transformation génère et prouve le principe d'inversion suivant :
$\forall (A : Type)(a : A)(l\ l' : list\ A),\ Add\ A\ a\ l\ l' \longleftrightarrow \exists l_1,\ l_1 = l \wedge l_1 = l' \vee \exists a_1\ a_2\ l_1\ l_2,\ l = a_1 :: l_1\ \wedge l' = a_2 :: l_2 \wedge Add\ A\ a\ l_1\ l_2$

Un fichier d'exemples est disponible à l'adresse suivante : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_indrel.v

## 4.4 Polymorphisme prénexe

### 4.4.1 Définitions

Lors des précédents chapitres, nous avons rencontré beaucoup d'énoncés qui présentent du *polymorphisme prénexe*. Il s'agit d'une quantification universelle sur des variables de type $Type$ en tête de formule ou de terme.

**Exemple 36** Un énoncé admettant du polymorphisme prénexe
Le *principe de non confusion* (voir 4.2.2), avec pour les listes, est généré par une de nos transformations et comporte du polymorphisme prénexe. $\forall (A : Type)(x : A)(l : list\ A), [] = x :: xs \rightarrow False$

**Exemple 37** Un énoncé admettant du polymorphisme non prénexe
Cet énoncé comporte du polymorphisme *non prénexe*. En effet, il n'est pas possible de trouver, dans la logique de `Coq`, un terme convertible dont tous les quantificateurs universels seraient en tête de formule.

$$(\forall\,(A : Type),\ A \to A) \to (\forall\,(A : Type),\ A \to A)$$

### 4.4.2 Stratégies

Nous avons écrit deux transformations correspondant à l'instanciation des variables de type selon deux stratégies distinctes. Cette instanciation s'appelle aussi *monomorphisation*.

La monomorphisation complète est indécidable [10] : étant donné un ensemble de formules $F$ en logique du premier ordre avec polymorphisme prénexe, il n'existe pas d'algorithme calculant un ensemble $F'$ de formules monomorphes équisatisfiables. Mais l'article que nous venons de citer présente un encodage correct et complet du polymorphisme dans une logique multi-sortée monomorphe.

Au lieu d'implémenter cet encodage qui risque de casser la structure des termes, et de rendre le but après monomorphisation éloigné du but de départ, nous avons préféré suivre notre méthodologie qui consiste à disposer de transformations les plus simples possible. Nous choisissons donc de monomorphiser nos termes, avec des types algébriques choisis dans le but `Coq` courant. Pour pallier l'incomplétude inhérente à cette méthode, nous proposons deux stratégies différentes.

Avant de présenter les deux transformations, il nous faut remarquer que lorsque l'utilisateur·ice écrit $Type$ sans plus de précision, `Coq` introduit une variable fraîche d'univers quelconque. Par défaut, donc, $Type$ ne sera pas convertible au plus petit des types selon la relation de sous-typage, $Set$. Il faut préciser explicitement le niveau d'univers de $Type$ avec un niveau noté $Set$, pour que la conversion soit possible. Ainsi, on aura :

```
Goal (Type=Set). Fail reflexivity.
(* The command has indeed failed with message:
Unable to unify "Set" with "Type@{filename.n}". *)
Abort.

Goal (Type@{Set}=Set). reflexivity. Qed.
```

Cette précision est importante, parce que dans les deux transformations de monomorphisation, je chercherai souvent des termes de type $Type$, mais c'est un abus de langage, car en réalité je cherche à la fois des termes de type $Type$ et de type $Set$.

**La stratégie « grossière »** Le code de cette stratégie est disponible ici : `https://github.com/louiseddp/sniper/blob/for-phd/theories/instantiate_type.v`. Elle a été implémentée en `Ltac`.

La stratégie « grossière » consiste à analyser le but `Coq` à prouver, dans son contexte, et à y détecter tous les termes ayant le type $Type$.

Pour chaque formule polymorphe, chaque variable de type sera instanciée par chacun des termes détectés.

**Exemple 38** Détection des variables de type $Type$ Supposons que le but `Coq` et son contexte aient la forme suivante :

```
H : forall (A B : Type) (x1 x2 : A) (y1 y2 : B),
    (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2

  ============================================
(forall (x1 x2 : bool) (y1 y2 : nat),
 (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2) /\
(forall (x1 x2 : nat) (y1 y2 : bool),
 (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2) /\
(forall x1 x2 y1 y2 : bool, (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2)
```

Les termes de type *Type* du but sont :

— `bool`

— `nat`

— `bool*nat`

— `nat*bool`

— `bool*bool`

L'hypothèse `H` quantifie sur deux variables de type *Type*, nous aurons donc généré $5^2 = 25$ instances de la formule une fois la transformation appliquée, comme nous pouvons le voir ci-dessous.

```
  H1 : forall x1 x2 y1 y2 : bool,
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H2 : forall (x1 x2 : bool) (y1 y2 : bool * bool),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H3 : forall (x1 x2 : bool) (y1 y2 : nat),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H4 : forall (x1 x2 : bool) (y1 y2 : nat * bool),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H5 : forall (x1 x2 : bool) (y1 y2 : bool * nat),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H6 : forall (x1 x2 : bool * bool) (y1 y2 : bool),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H7 : forall x1 x2 y1 y2 : bool * bool,
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H8 : forall (x1 x2 : bool * bool) (y1 y2 : nat),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H9 : forall (x1 x2 : bool * bool) (y1 y2 : nat * bool),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H10 : forall (x1 x2 : bool * bool) (y1 y2 : bool * nat),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H11 : forall (x1 x2 : nat) (y1 y2 : bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H12 : forall (x1 x2 : nat) (y1 y2 : bool * bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H13 : forall x1 x2 y1 y2 : nat,
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H14 : forall (x1 x2 : nat) (y1 y2 : nat * bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H15 : forall (x1 x2 : nat) (y1 y2 : bool * nat),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H16 : forall (x1 x2 : nat * bool) (y1 y2 : bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H17 : forall (x1 x2 : nat * bool) (y1 y2 : bool * bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H18 : forall (x1 x2 : nat * bool) (y1 y2 : nat),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H19 : forall x1 x2 y1 y2 : nat * bool,
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
```

```
H20 : forall (x1 x2 : nat * bool) (y1 y2 : bool * nat),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H21 : forall (x1 x2 : bool * nat) (y1 y2 : bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H22 : forall (x1 x2 : bool * nat) (y1 y2 : bool * bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H23 : forall (x1 x2 : bool * nat) (y1 y2 : nat),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H24 : forall (x1 x2 : bool * nat) (y1 y2 : nat * bool),
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H25 : forall x1 x2 y1 y2 : bool * nat,
   (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
------------------------------------------------
(forall (x1 x2 : bool) (y1 y2 : nat),
(x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2) /\
(forall (x1 x2 : nat) (y1 y2 : bool),
(x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2) /\
(forall x1 x2 y1 y2 : bool, (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2)
```

Cette stratégie a le (gros) défaut d'être exponentielle en le nombre de variables de type $Type$ en tête de formule. Suite à cette première implémentation, une stratégie de monomorphisation plus efficace mais moins exhaustive a donc été considérée.

**La stratégie « paramétrique »** Le code de cette stratégie est disponible ici : https://github.com/louiseddp/sniper/blob/for-phd/theories/instantiate_inductive_pars.v. Elle a été implémentée en `Coq-Elpi`.

L'idée, cette fois-ci, est de regarder les paramètres clos des inductifs ou de leurs constructeurs. Cette stratégie n'est évidemment pas complète, par exemple, elle passe à côté des instances de la forme $A \to B$, mais elle permet de résoudre les buts où le polymorphisme prénexe intervient car les formules mentionnent un inductif avec des paramètres de type $Type$.

Soit $I$ un inductif et $T_1$, ..., $T_k$ les types de ses paramètres (nous ne considérons pas ses annotations).

Nous appellerons *contexte d'instanciation* un terme de la forme :

$\lambda x_i,\ I\ \_\ ...\ x_i\ ...\ \_$, avec les _ des « espaces réservés » qui peuvent désigner n'importe quels termes, ou $\lambda x_i,\ c\ \_\ ...\ x_i\ ...\ \_$ avec $c$ un constructeur de $I$. Notons que si le type de $x_i$ et des paramètres de $I$ le permet, $x_i$ peut apparaître plusieurs fois dans le contexte d'instanciation.

Nous noterons ces contextes $C[x_i]$.

**Exemple 39** Contexte d'instanciation

$A * \_$ est un contexte d'instanciation, où le terme $A$ apparaît à gauche d'un type produit.

La stratégie paramétrique consiste ensuite à faire correspondre les contextes d'instanciation des variables de type $Type$ d'une formule, aux contextes d'instanciation déjà présents dans le but `Coq` et son contexte.

**Exemple 40** Instanciation d'une formule

Par exemple, soit la formule `H : forall (A B : Type)(x1 x2 : A)(y1 y2 : B), (x1, y1)= (x2, y2)-> x1 = x2 /\ y1 = y2.`

Elle présente le contexte $A * \_$ avec $A$ une variable de type.

Le but de l'exemple 38 présente le contexte $nat * \_$, donc $A$ doit être instanciée par $nat$.

**Exemple 41** Instanciation de tout le but

Finalement, pour reprendre le but de l'exemple 38, le but résultant sera :

```
H1 : forall x1 x2 y1 y2 : bool,
(x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H2 : forall (x1 x2 : bool) (y1 y2 : nat),
(x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
H3 : forall (x1 x2 : nat) (y1 y2 : bool),
 (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
------------------------------------------------
(forall (x1 x2 : bool) (y1 y2 : nat),
(x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2) /\
(forall (x1 x2 : nat) (y1 y2 : bool),
(x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2) /\
(forall x1 x2 y1 y2 : bool, (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2)
```

Enfin, cette transformation est correcte parce que chaque instanciation d'une variable de type est une règle d'application en CIC.

Des exemples pour les deux transformations sont disponibles ici : https://github.com/louiseddp/sniper/tree/for-phd/examples/example_monomorphization.v.

## 4.5 Ordre supérieur

### 4.5.1 Fonctions anonymes

Le code de cette transformation implémentée en Coq-Elpi est disponible ici : https://github.com/louiseddp/sniper/blob/for-phd/theories/anonymous_functions.v.

Dans Coq, une fonction anonyme est un terme de la forme `fun x1 ... xn => ...`.

Les prouveurs du premier ordre ne connaissent pas cette représentation des fonctions, ils ont besoin d'une fonction concrète.

Ainsi, nous avons implémenté une transformation ajoutant la définition d'une fonction anonyme au contexte local. Toutes les fonctions anonymes présentes comme sous termes (à l'exception des branches des *pattern matching* qui contiennent une fonction anonyme par branche), vont être posées comme définition locales.

Par exemple, si `fun x => f (g x)` apparaît dans le but, la transformation va créer une nouvelle définition locale `composefg := fun x => f (g x)`. Par ailleurs, chaque occurrence de la version dépliée de `composefg` sera repliée.

La définition de `composefg` sera récupérée grâce aux successions de transformations que nous avons déjà présentées et qui donnent les équations des fonctions.

Par ailleurs, cette transformation est correcte car il est toujours correct d'ajouter une définition locale bien typée dans le contexte d'un but Coq, et parce que la fonction anonyme et sa définition repliée sont convertibles.

Un fichier d'exemple est disponible ici : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_anonymous_functions.v

### 4.5.2 Élimination de l'ordre supérieur « prénexe »

Le code de cette transformation implémentée en `Coq-Elpi` est disponible ici : `https://github.com/louiseddp/sniper/blob/for-phd/theories/higher_order.v`.

On dit qu'une fonction contient une quantification d'ordre supérieur prénexe si son type est de la forme : $\forall(\vec{A_i} : Type)(f : T \to U), B$

Ainsi, cette quantification sur cette variable de fonction ne peut se trouver qu'en tête du type (éventuellement après une quantification sur des variables de type).

La quantification d'ordre supérieur prénexe peut être instanciée : à chaque fois que le contexte mentionne une telle fonction d'ordre supérieur appliquée à un argument concret, une instance peut être créée. Par exemple, si le contexte mentionne une fonction $f : nat \to nat$, la fonction $map : \forall(A\ B : Type)(f : A \to B),\ list\ A \to list\ B$ peut être instanciée par $f$.

La transformation qui élimine les quantifications d'ordre supérieur prénexe consiste à poser une définition locale pour chaque instance d'une fonction d'ordre supérieur. Ainsi, avec l'exemple précédent, la transformation pose : $map_f := map\ f$, qui, dans ce cas, n'est plus d'ordre supérieur. Ensuite, chaque occurrence de $map\ f$ dans le contexte de preuve sera repliée pour que l'on obtienne $map_f$ à la place.

À nouveau, cette transformation est correcte car elle ne fait que poser une définition locale et replier la définition des constantes.

Un fichier d'exemple est disponible ici :`https://github.com/louiseddp/sniper/blob/for-phd/examples/example_higher_order.v`

# Chapitre 5

# UN ORDONNANCEUR

Les transformations du plugin `Sniper` sont combinées dans la tactique `snipe`. Celle-ci appelle toutes les transformations présentées précédemment, à l'exception des transformations sur les relations inductives (car l'une n'est pas une tactique, et l'autre génère des énoncés qui ne sont pas exploitables par un prouveur SMT). L'étape de combinaison des tactiques de préprocessing est aussi appelée `scope`. Une fois toutes les transformations appelées, `snipe` lance la tactique `verit` de SMTCoq, qui appelle le prouveur SMT `veriT` sur le but transformé. La tactique `snipe` existe en deux versions :

— Une version *ad hoc*, que j'appelle « l'ancienne version », dont l'implémentation n'est pas satisfaisante mais qui nous a permis de vérifier qu'il était possible d'obtenir une tactique de préprocessing combinant plusieurs transformations logiques. Les résultats préliminaires ainsi que les problèmes causés par cette implémentation seront présentés en 5.1. Le code de cette version est disponible dans `https://github.com/louiseddp/sniper/blob/for-phd`.

— Une version utilisant un nouveau plugin `Coq`, appelé `Orchestrator`, qui prend en entrée différentes tactiques et des informations sur celles-ci et dont la charge est ensuite d'appliquer les tactiques dans le bon ordre. Toute la suite du chapitre est consacrée à cette version de la tactique. Notons bien que le plugin `Orchestrator` est indépendant du plugin `Sniper`, et peut éventuellement être utilisé pour d'autres préprocessings. Pour le moment, les deux plugins sont encore mélangés, mais, à terme, nous souhaitons créer un paquet `opam` séparé. Le code de l'ordonnanceur est disponible dans une branche différente de `Sniper` : `https://github.com/louiseddp/sniper/tree/for-phd-orchestrator`.

## 5.1 Naissance de l'idée

Dans cette section, nous verrons pourquoi les outils existants en `Coq` n'étaient pas satisfaisants pour combiner les transformations logiques présentées aux chapitres précédents, et les problèmes auxquels nous nous sommes heurté·es.

### 5.1.1 Première version de la tactique `snipe`

#### Présentation de la première version

La première version de la tactique `snipe` a fait l'objet de deux publications à PxTP 2021 [9] et à CPP 2023 [8]. Son code est disponible ici : `https://github.com/louiseddp/sniper/blob/`

for-phd/theories/Sniper.v.

En effet, cette première version de snipe permet déjà à SMTCoq de résoudre des buts que l'outil ne pouvait pas résoudre car il manquait la signification de certains symboles (types ou fonctions) au prouveur, ou parce que le but présentait du polymorphisme ou de l'ordre supérieur prénexe.

**Description de la première version de la tactique et des transformations utilisées** La tactique snipe étant divisée en deux tactiques : scope (pour le préprocessing) et verit (pour l'appel au backend), nous décrirons ici le code de scope puisque notre travail a porté sur cette tactique.

La tactique scope commence par remettre dans le but toutes les hypothèses via une tactique auxiliaire revert_all. Puis, scope effectue un appel à trakt bool. Il est nécessaire d'utiliser revert sur toutes les hypothèses parce que trakt n'agit que sur le but.

L'appel à trakt est essentiel ici pour traduire d'éventuels prédicats inductifs en points fixes (si l'utilisateur·ice a décidé ces prédicats via la commande Decide, ou s'il ou elle l'a fait manuellement et les a ajoutés à la base de données de trakt). Nous avons besoin d'appeler trakt en premier, parce qu'il faut ensuite donner la définition de ces points fixes.

Ensuite, scope introduit toutes les variables et les hypothèses via intros.

Puis, scope élimine d'éventuelles hypothèses contenant du *pattern matching*. D'autres seront éventuellement générées plus tard et la transformation sera donc rappelée.

Pour tous les types algébriques présents dans les hypothèses et dans le but, scope génère le principe de non-confusion et l'injectivité des constructeurs.

Ensuite, scope élimine les fonctions anonymes et l'ordre supérieur « prénexe ».

Une fois cela fait, scope va générer et prouver les équations de toutes les constantes dans le contexte, via la succession de la transformation des définitions, de l'élimination des égalités d'ordre supérieur, des points fixes et du *pattern matching* à nouveau. Ce chaînage de transformations est décrit un peu plus loin à l'exemple 42.

Puis, scope va ajouter au contexte local le principe de génération pour chaque type algébrique $I$, tel qu'il existe une variable $t : I$ dans le contexte local. On crée donc, en général, moins d'énoncés de ce principe que pour les principes de non-confusion et d'injectivité des constructeurs. En effet, ce principe comporte une disjonction et ajoute des fonctions de projection au contexte. Il alourdit donc beaucoup le contexte local. Comme son utilité est d'aider le prouveur à faire une analyse de cas, nous l'avons limité aux types des variables du contexte local, puisque c'est souvent sur ces variables que l'analyse est effectuée.

Enfin, scope appelle l'une des transformations de monomorphisation. Il existe deux versions de scope, appelant l'une ou l'autre des transformations.

**Quelques exemples** Comme nous l'avons dit, cette tactique permet déjà de résoudre des buts que la tactique verit ne pouvait pas résoudre seule. Nous proposons de consulter un fichier d'exemples à l'adresse suivante : https://github.com/louiseddp/sniper/blob/for-phd/examples/example_snipe.v pour voir la première version de snipe à l'œuvre.

### Combiner des tactiques ?

Contrairement au préprocessing traditionnel qui propose une tactique unique (comme avec Trakt et sa tactique trakt), ou une seule étape écrite en OCaml, comme le préprocessing de CoqHammer, Sniper propose plusieurs tactiques. Il est donc nécessaire que ces tactiques puissent se transmettre de l'information.

Par exemple, si la tactique $tac_1$ a généré l'énoncé $H$, on voudrait pouvoir appeler directement $tac_2$ sur cet énoncé, et donc, transmettre $H$ à $tac_2$.

Or, nous avions vu dans le chapitre sur `Ltac` (voir 3.4.2) qu'écrire des tactiques qui à la fois faisaient un effet de bord (comme ici $tac_1$ qui génère un énoncé), et renvoyaient une valeur (l'énoncé $H$), rendait vite le code peu lisible car `Ltac` n'avait simplement pas été pensé pour.

Utiliser d'autres métalangages n'était pas non plus une solution idéale :

— `MetaCoq` ne permet pas de manipuler le contexte de preuve (sauf via `Ltac`), donc générer des énoncés implique de toute manière de passer par `Ltac`.

— La manipulation du contexte de preuve en `Coq-Elpi` est extrêmement difficile à cause de la syntaxe abstraite d'ordre supérieur et conduit à de nombreux bugs donc nous n'avions pas trouvé ce langage adapté non plus.

— Le langage `Ltac2` pouvait paraître adapté, mais nous avions déjà écrit des transformations dans d'autres métalangages, et, en raison du manque de documentation à son sujet, nous ne l'avons pas adopté tout de suite. Par ailleurs, il manquait certaines fonctionnalités jusqu'à des versions récentes de `Coq`, ce qui a empêché pendant longtemps sa pleine adoption.

Ainsi, dans la première implémentation de `snipe`, certaines transformations utilisaient des continuations, de manière à pouvoir les chaîner avec une autre transformation (puisqu'elles ne peuvent pas renvoyer leur résultats), et d'autres, dont l'application ne dépend pas d'un énoncé précis précédemment généré, n'en utilisaient pas.

**Exemple 42** Chaînage des transformations donnant les équations d'une fonction

Pour obtenir les équations d'une fonction, on utilisait la tactique intermédiaire (simplifiée) suivante :

```
Ltac get_equations f :=
  get_def f ltac:(fun H => expand_hyp H ltac:(fun H =>
      eliminate_fix_hyp H) ltac:(fun H => eliminate_pattern_matching
       H))
```

À cet exemple, nous voyons deux problèmes qui sont reliés, et qui font que cette méthode n'est pas satisfaisante :

1. Les transformations ne sont plus *atomiques*. L'un des principes de `Sniper` est d'avoir des transformations indépendantes, or, en quelque sorte, dans leur code, il est déjà écrit qu'elles dépendent d'autres transformations si elles utilisent des continuations. Bien sûr, il est toujours possible d'obtenir la version indépendante `tac H ltac:(fun H => idtac)`, mais elle contraint tout·e programmeur·euse qui voudrait intégrer sa propre transformation à `snipe` à utiliser des continuations dans ses tactiques. Contraindre à un style d'implémentation à cause de limitations (surmontables) d'un métalangage (`Ltac`) paraît discutable.
2. Cet aspect est le plus problématique : la manière dont les transformations sont chaînées est rigide et ne dépend pas du but `Coq` initial. Par exemple, dans cette tactique `get_equations` est codé en dur que la transformation qui élimine les égalités d'ordre supérieur est suivie par l'élimination des points fixes, et ce, même si aucun point fixe n'apparaîtra jamais dans le but. La tactique qui élimine les points fixes devra vérifier si elle s'applique réellement à l'hypothèse `H` ou non, ce qui va à l'encontre de notre souhait d'obtenir des transformations les plus atomiques possibles (nous voudrions gérer en amont si la transformation s'applique ou non). L'un de nos objectifs est de pouvoir facilement combiner les transformations dans des ordres différents, et cette implémentation semble aller à l'encontre de celui-ci.

Un autre problème majeur consiste en des interactions négatives, parfois difficiles à anticiper, entre certaines tactiques. Un exemple typique est la combinaison de la transformation des définitions (voir 4.1.1) avec la transformation qui traite l'ordre supérieur « prénexe » (voir 4.5.2).

**Exemple 43** Ordre supérieur et définitions

Supposons que nous ayons dans notre contexte local la fonction d'ordre supérieur appliquée `map f`, qui applique la même fonction `f` à une liste.

Appliquée en premier, la fonction qui déplie les définitions va prouver et ajouter dans le contexte local l'hypothèse :

```
H : map = fun (A B : Type)(g : A -> B)=> fix l ...
```

Ensuite, le reste des transformations chargées d'obtenir les équations des fonctions vont s'appliquer, et nous aurons finalement les équations de `map`.

```
H1 : forall (A B : Type) (g : A -> B), map g [] = []
H2 : forall (A B : Type ) (g : A -> B) (x : A) (xs : list A),
  map g (x::xs) = f x :: (map g xs)
```

Si ces équations apparaissent dans le contexte local, la tactique `verit` échoue à cause de la présence d'ordre supérieur (le symbole `g` en argument de `map`).

Nous pourrions nous dire naïvement qu'il suffit d'appliquer la tactique qui élimine l'ordre supérieur en premier. En effet, elle va définir `map_f := map f` et faire disparaître les occurrences de `map f` dans le but pour les remplacer par la version repliée `map_f`.

Mais, de nouveau, quand la transformation des définitions intervient, elle génère d'abord :

```
H' : map_f = map f
```

Puis, elle sera rappelée sur `map` car il nous faut donner la définition de ce nouveau symbole, régénérant alors l'hypothèse problématique :

```
H : map = fun (A B : Type)(g : A -> B)=> fix l ...
```

Nous pouvons empêcher la tactique qui déplie les définitions de s'appliquer sur des termes d'ordre supérieur. Nous reviendrons sur ce point et en quoi il pose problème. Mais, quoi qu'il en soit, ce n'est pas suffisant. Si nous procédons ainsi, nous sommes coincé·s avec une égalité inexploitable pour le prouveur (qui de surcroît comporte de l'ordre supérieur elle aussi) :

```
H' : map_f = map f
```

Il nous faut précisément déplier `map` *au sein de cette égalité, sans en créer de nouvelle.* De cette manière, nous obtiendrons :

```
H' : map_f = fix l ...
```

Et finalement, nous obtiendrons les équations de `map_f` et non celles de `map`.

Dans la première version de `snipe`, le problème était traité en interdisant de déplier des définitions de termes d'ordre supérieur (et des définitions locales), de telle sorte que la tactique qui traite l'ordre supérieur « prénexe » ne faisait pas seulement l'étape décrite en 4.5.2. Elle appelait aussi d'autres transformations, de manière à produire directement les équations de `map_f`.

Des problèmes similaires conduisent à encore plus casser l'atomicité des transformations, en intégrant de nombreuses solutions *ad hoc* pour empêcher les interactions négatives. Ainsi, la première version de la tactique `snipe`, qui combine certaines des transformations que nous avons vues dans le chapitre précédent, avait une implémentation peu lisible. Dès qu'une transformation était ajoutée dans la tactique, il fallait la retravailler en conséquence pour vérifier qu'elle

s'enchaînait bien avec les transformations précédentes, et parfois, nous étions contraint·es à user de transformations beaucoup moins atomiques que ce que nous aurions souhaité.

### 5.1.2 Déclencher des tactiques ?

Un autre problème était lié à une grande perte d'efficacité lors de la combinaison des transformations, perte d'efficacité que nous estimions évitable. En effet, jusqu'ici, nous avons supposé que les transformations s'appliquaient déjà sur les bons arguments. Mais comment trouvent-elles les bons arguments à l'intérieur du but et du contexte de preuve ? Nous allons ici nous concentrer sur les métalangages `Ltac` et `Ltac2`, qui permettent tous les deux d'utiliser directement la syntaxe de `Coq` pour analyser le contexte.

Prenons également comme exemple la transformation qui interprète les types algébriques, et plus, précisément, celles qui produit les énoncés d'injectivité des constructeurs et du fait qu'ils sont disjoints (voir 4.2).

Nous souhaiterions déclencher cette transformation dans quatre cas :

1. Quand un type algébrique sur lequel la transformation ne s'est pas déjà appliquée est présent dans les hypothèses
2. Quand un type algébrique sur lequel la transformation ne s'est pas déjà appliquée est présent dans le but
3. Quand un constructeur ou une variable d'un type algébrique sur lequel la transformation ne s'est pas déjà appliquée est présent dans les hypothèses
4. Quand un constructeur ou une variable d'un type algébrique sur lequel la transformation ne s'est pas déjà appliquée est présent dans le but

Ainsi, la solution consiste à effectuer un *pattern matching* sur le contexte et sur le but, et à analyser chaque sous-terme, grâce à la construction `Ltac` :

```
match goal with
  | |- context C[?y] => (* instructions concernant le but *)
  | _ : context C[?y] |- _ => (* instructions concernant les hypothèses *)
```

Ajouté à ce `match goal with`, il faut que la transformation s'appelle récursivement et échoue si elle tombe sur un sous-terme `y` qu'elle a déjà analysé. La transformation doit donc également garder en mémoire les sous-termes qu'elle a traités. Comme échouer déclenche le *backtracking* en `Ltac`, cela permet de tester de nouveaux termes liés par la variable `y`.

Plusieurs remarques maintenant :

— Cette solution repose sur une sémantique imprévisible, très *ad hoc*, difficile à comprendre et qui, en pratique, a conduit à beaucoup d'essais-erreurs : la sémantique du *backtracking* de `Ltac`. En pratique, le `match goal with` avec *backtracking* semble à exclure d'une implémentation claire.

— Cette solution semble inefficace : chaque transformation scanne de son côté le contexte pour vérifier si et comment elle s'applique, et ce plusieurs fois. Mais, si nous avions souhaité déléguer les analyses du contexte à la tactique `snipe` qui combine les transformations, pour diminuer le nombre de scans du contexte, cela aurait signifié que les `match goal with` seraient la tâche d'une tactique encore plus complexe. Ce n'était pas envisageable, nous nous serions heurté·es à trop de bugs extrêmement difficiles à interpréter (étant donné que les messages d'erreurs d'aucun métalangage ne semblent satisfaisants pour l'instant).

— Est-ce vraiment le rôle de la transformation elle-même de gérer le contexte global ? Dans la première version de `Sniper`, beaucoup de transformations ont fini par se décomposer en deux versions : la version atomique et locale (car s'appliquant à des termes précis) que nous avons présentée au chapitre 4, et la version effectivement utilisée, globale, qui scanne les contextes à la recherche des arguments à donner à la version locale. Dans ce cadre, il était difficile d'imaginer comment un·e utilisateur·ice externe pouvait facilement intégrer sa transformation dans l'outil. Cela lui demanderait de penser à la façon dont sa transformation interagirait avec le contexte de preuve, et donc, en pratique, cela l'éloignerait de la transformation proprement dite.

En conséquence, nous avons observé que la tactique `snipe` exécutait rapidement chaque transformation atomique, mais qu'elle devenait très lente à cause de ces analyses répétées du contexte de la preuve `Coq`. Le préprocessing classique, en une étape, ne présente pas ce problème, puisqu'il n'a besoin que d'un seul scan. Nous avions donc besoin d'une solution adaptée à notre méthodologie spécifique.

### 5.1.3 Symboles à ne pas interpréter

Nous avons vu un cas à l'exemple 43 où, si une transformation donnée s'exécutait, elle faisait échouer le prouveur automatique car elle réintroduisait des termes d'une logique d'ordre supérieur.

Il existe une autre raison pour laquelle il est souhaitable qu'une transformation ne s'applique pas. Cette raison est spécifique aux backends qui seraient déjà efficaces sur certains symboles.

Par exemple, la tactique `lia` n'a pas besoin d'un préprocessing qui donnerait l'interprétation du type algébrique `Z`, puisqu'elle permet déjà de résoudre des lemmes arithmétiques sur ce type de donnée.

De façon similaire, les prouveurs SMT disposent d'une automatisation spécialisée sur certains types de données (les entiers relatifs (`Z`), les *bitvectors*, les tableaux...). Donner la signification de ces symboles (et des fonctions associées comme l'addition dans `Z`) ne ferait qu'encombrer le contexte de preuve d'énoncés dont le prouveur n'a pas besoin. Comme nous disposons dans le plugin `Sniper` de la transformation `trakt`, qui, pour rappel, permet de convertir un type de donnée en un autre (voir 2.2.3), nous ne désirons pas non plus traduire `nat`, `Nat.add`, etc.

Ainsi, la première version de `snipe` prenait en paramètre plusieurs tuples de termes, qui étaient ensuite transmis aux transformations adéquates. Pour donner une idée, récapitulons dans le tableau (non exhaustif) suivant.

| **Nom de la transformation** | **Exemples de termes sur lesquels ne pas la déclencher** |
|---|---|
| Définitions | `Z.add, Nat.add, Z.le, Nat.le, @BITVECTOR_LIST.bv_eq, is_true` |
| Types algébriques : injectivité des constructeurs et principe de non confusion | `Z, bool, @eq, nat` |
| Types algébriques : principe de génération | `Z, bool, @eq, nat` |

L'idée d'empêcher les transformations de s'appliquer sur certains arguments a été utile pour l'implémentation de l'ordonnanceur, comme nous allons le voir ci-dessous.

## 5.2 Le plugin `Orchestrator` : *triggers* et filtres

### 5.2.1 Présentation

Nous avons vu que la combinaison des transformations posait plusieurs difficultés. Pour régler celles-ci, nous avons conçu un nouveau plugin spécialisé dans la combinaison de tactiques `Coq`, appelé `Orchestrator`.

Étant donné une liste de tactiques `Coq` en entrée (écrites ou converties dans le langage `Ltac`, et ne prenant en argument que des termes `Coq`, donc en particulier pas d'autres tactiques), une spécification de ce qui *déclenche* ces tactiques et de *filtres* (nous expliquerons ce terme par la suite), une tactique d'ordonnancement appelée `orchestrator` sera capable de sélectionner les bons arguments pour ces tactiques et de les appliquer autant de fois que nécessaire. L'ordonnanceur attend donc une liste de triplets tactiques, *triggers* (déclencheurs) et *filtres*.

Si deux tactiques peuvent s'appliquer au même moment, l'ordre d'application est simplement déterminé par l'ordre dans lequel les transformations ont été fournies dans la liste. Le but principal de l'ordonnanceur n'est donc pas de choisir lui-même des stratégies d'automatisation, mais de choisir les bons arguments pour les tactiques et de permettre de les combiner facilement. Cela signifie que si l'utilisateur·ice a demandé à l'ordonnanceur de commencer par une tactique qui peut transformer un but $G$, prouvable, en un but $G'$, non prouvable, celui-ci ne pourra pas détecter qu'il vaut mieux appliquer cette tactique après d'autres tactiques qui seraient, elles, réversibles. De même, l'ordonnanceur, dans sa version actuelle ne fait pas de *backtracking*.

L'idée de l'ordonnanceur est aussi de pouvoir proposer facilement différentes stratégies de préprocessing, en fonction du backend de son choix. L'illustration de ce principe est donnée par le schéma 5.1.

### 5.2.2 Le cœur de l'ordonnanceur : les *triggers*

Les *triggers* forment un type de donnée défini en `Ltac2` et permettant de décrire ce qui déclenche une tactique. Nous vous invitons à consulter le fichier les décrivant : `https://github.com/louiseddp/sniper/blob/for-phd-orchestrator/theories/triggers.v`.

L'idée est de fournir un langage plus simple pour cette description que celle offerte par les métalangages pour `Coq`. Un·e utilisateur·ice doit pouvoir écrire un *trigger* sans connaître aucun métalangage. Cela peut être le cas, par exemple, s'il ou elle souhaite intégrer dans l'ordonnanceur une tactique *tac* dans un cas qu'il ou elle décrira via un *trigger* mais sans savoir comment *tac* est implémentée.

De plus, utiliser le langage des *triggers* permet de séparer ce qui relève du déclenchement de la tactique de ce que fait la tactique proprement dit. Cet aspect va avec notre souhait de modularité : nous pensons que la séparation de ces deux questions d'implémentation permet d'obtenir un code plus lisible et donc plus facilement modifiable et débuggable (et nous considérons d'ailleurs que la version de `snipe` qui utilise l'ordonnanceur correspond effectivement à ces critères).

À la syntaxe des *triggers* est associée une fonction d'interprétation. Cette fonction, que nous appellerons `interp_trigger`, prend en entrée un *trigger* et un but `Coq`, et renvoie soit `[]` si la tactique associée au *trigger* n'est pas déclenchée dans le but `Coq` en question, soit une liste de listes de termes `l`. Chaque élément de `l` est une liste d'arguments possibles pour la tactique, qui a donc été déclenchée.

Par exemple, si l'interprétation d'un *trigger* de `tac` est `[[list nat]; [nat]]`, alors les applications `tac (list nat)` et `tac nat` sont toutes les deux valides. L'ordonnanceur commencera par exécuter `tac (list nat)` (et, selon la façon dont le but évolue ensuite, il pourra exécuter ou non `tac nat`).

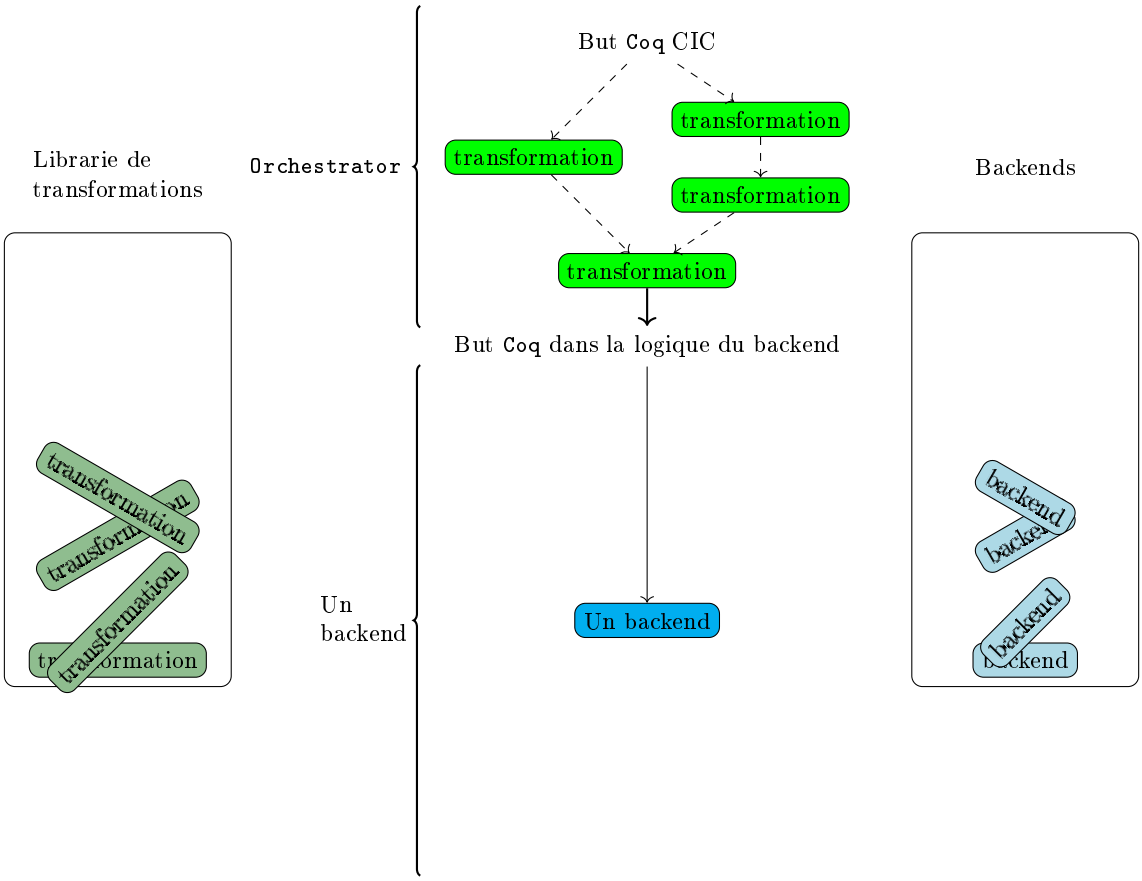



FIGURE 5.1 – Le principe de l'ordonnanceur

Notons qu'une tactique ne prenant pas d'arguments qui a été déclenchée sera associée au résultat `[[]]`.

Avant de s'intéresser à l'interprétation des *triggers*, nous allons introduire leur syntaxe, d'abord via un exemple puis en général.

**Exemple 44** Une introduction aux *triggers* via la logique propositionnelle

Ce travail a été l'objet d'un stage effectué par Alexis Carré et encadré par Chantal Keller et moi-même, qui nous a permis de trouver les idées fondamentales concernant les *triggers*. L'idée était de constituer un ordonnanceur de tactiques pour la logique propositionnelle.

Considérons des buts `Coq` de logique propositionnelle, qui peuvent donc contenir des conjonctions, des disjonctions, des formules atomiques, des négations, des implications, ou les formules $\top$ et $\bot$ (`True` et `False` en `Coq`).

À partir du but et de ce que contient le contexte local de la preuve `Coq`, nous pouvons définir des *triggers* pour des tactiques de raisonnement propositionnel.

Par exemple, le *trigger* pour la tactique `assumption` serait : le but est égal à une hypothèse. (Notons d'emblée que même si `assumption` est capable d'égalité modulo conversion, les *triggers* fonctionnent avec la notion d'égalité syntaxique.)

Le trigger (simplifié) serait donc : `TIs TGoal TSomeHyp`

En notation infixe, le *trigger* est précisément lisible comme : « le but est (syntaxiquement) égal à une hypothèse ».

Considérons maintenant le *trigger* que pourrait être celui de la tactique `split`.

En réalité `split` est une tactique assez puissante, capable d'agir sur tout but qui est un type inductif avec un seul constructeur, mais pour l'exemple, nous supposerons qu'elle agit seulement sur les conjonctions.

Ce trigger serait : « le but est une conjonction ».

On pourrait donc l'écrire : `TIs TGoal (TAnd tDiscard tDiscard)`.

Le terme `tDiscard` est une notation que nous développerons plus loin, pour l'instant, contentons-nous de dire qu'il intervient parce que `split` n'a pas besoin de connaître les deux arguments de la conjonction.

Nous avions dit que pour l'exemple, nous supposions d'abord que `split` agissait seulement sur les conjonctions. Mais cessons de faire cette supposition et rappelons qu'en réalité `split` peut aussi résoudre un but tel que `True`. Un meilleur *trigger* pour `split` serait donc :

```
TDisj (TIs TGoal (TAnd tDiscard tDiscard))(TIs TGoal (TConstr True))
```

Les *triggers* disposent en effet d'un constructeur `TConstr`, qui attend en argument n'importe quel terme `Coq`, ce qui permet ici de tester l'égalité syntaxique entre le but et le terme `True`.

Les constructeurs des *triggers* attendent parfois des arguments de type `flag`, qui permettent de préciser à la fonction d'interprétation si la tactique qui correspond au *trigger* va prendre en argument certains paramètres ou non. Dans la suite, nous allons parfois ignorer ces *flags* par souci de concision, quand nous considérerons que leur présence entrave la lecture et la compréhension d'un *trigger*.

Les deux *flags* sont : `NotArg` qui précise que la tactique ne prend pas en argument le terme `t` associé au *flag*, ou `Arg : (constr -> constr)-> flag`, tel que `Arg f` signifie que la tactique prendra en argument `f t` (rappelons qu'en `Ltac2`, `constr` désigne le type des termes `Coq`). Ainsi, `tDiscard` est une notation pour `TAny (NotArg )` (`TAny` désigne n'importe quel terme `Coq`, et ce terme ne sera pas donné à la tactique).

Pour reprendre l'exemple de la conjonction, supposons que nous souhaitions définir un *trigger* pour une tactique moins puissante que `split`, et qui attend des arguments. La tactique `fun A B => apply (conj A B)`, qui applique l'unique constructeur de `and A B`, est une telle tactique. Cette fois, le *trigger* serait :

`TIs TGoal (TAnd (TAny (Arg id))((TAny (Arg id))))`

Pour en finir avec les exemples introductifs de logique propositionnelle, présentons une construction plus complexe. Une condition suffisante pour appliquer la tactique `left` tout en maintenant le but prouvable est que le but est une disjonction, dont le premier argument est égal à une hypothèse du contexte local.

La condition, exprimée dans le langage des *triggers*, nécessite une définition locale. Les *triggers* en comportent, grâce au constructeur `TMetaLetIn : trigger -> string list -> trigger`.

L'idée est de lier certains arguments du premier *trigger* et de leur donner des noms (d'où le second argument de type `string list`). Ensuite, le second *trigger* pourra mentionner ces variables, à l'aide du constructeur `TNamed: string`. Voici le *trigger* que nous voulons écrire :

```
TMetaLetIn
  (TIs TGoal (TOr (TArg id) tDiscard)) (* le but est un "ou" dont on
      ne considère que le premier argument *)
  ["A"] (* on nomme cet argument "A" *)
  (TIs TSomeHyp (TNamed "A")) (* une hypothèse est égale à "A" *)
```

Maintenant que nous avons introduit quelques exemples de *triggers*, voyons leur syntaxe en détail.

Tout d'abord, on retrouve les *flags*, qui seront utiles à la fonction d'interprétation des *triggers*, afin de savoir si certains termes seront ou non liés par un `TMetaLetIn`, ou donnés en arguments à la tactique dont on analyse le *trigger*.

```
Ltac2 Type flag :=
  [ Arg (constr -> constr) | NotArg ].
```

Ensuite, nous avons les *variables de triggers*, qui permettent de désigner des hypothèses du contexte local (toutes ou seulement celles de sorte *Prop*), le but, les définitions locales, ou les termes liés par les `TMetaLetIn`.

```
Ltac2 Type trigger_var :=
  [ TGoal (* désigne le but *)
  | TSomeHyp (* désigne une hypothèse ou une définition locale *)
  | TSomeDef (* désigne une définition locale *)
  | TSomeHypProp (* désigne une hypothèse de type Prop *)
  | TNamed (string) (* désigne une variable locale *)
  ].
```

Puis nous avons le type inductif `trigger_term`, que la fonction d'interprétation fera correspondre à un terme de `Coq`. Rappelons notre deuxième exemple de *trigger* : `TIs TGoal (TAnd tDiscard tDiscard)`. Ici, `TAnd tDiscard tDiscard` est de type `trigger_term`. La fonction d'interprétation fait correspondre le but `Coq` courant (qui est bien un terme de `Coq`), à une conjonction. Elle renverra `[]` si le but n'est pas une conjonction (la tactique n'est pas déclenchée), et `[[]]` sinon (la tactique est déclenchée mais ne prend pas d'arguments).

Voici tout le type de donnée `trigger_term` :

```
Ltac2 Type trigger_sort :=
[ TProp (* pour la sorte Prop *)
| TSet (* pour la sorte Set *)
| TBigType]. (* pour toutes les sortes autres que Set et Prop *)

Ltac2 Type rec trigger_term := [

(* suit le type constr de OCaml *)
| TRel (int, flag) (*indices de Bruijn *)
| TVar (trigger_local_var, flag) (* variable locale ou de section *)
| TSort (trigger_sort, flag) (* sortes (simplifiées) *)
| TProd (trigger_term, trigger_term, flag) (* produits *)
| TLambda (trigger_term, trigger_term, flag) (* fonctions *)
| TLetIn (trigger_term, trigger_term, trigger_term, flag) (* définitions
    locales *)
| TApp (trigger_term, trigger_term, flag) (* applications *)
| TConstant (string option, flag) (* constantes globales *)
| TInd (string option, flag) (* inductifs *)
| TConstructor (string option, flag) (* constructeurs *)
| TCase (trigger_term, trigger_term, trigger_term list option, flag) (*
    pattern matching *)
| TFix (trigger_term, trigger_term, flag) (* point fixe *)
| TCoFix (trigger_term, trigger_term, flag) (* co-point fixe *)

(* structure à grains moins fins pour simplifier l'écriture de certains
    triggers *)
| TEq (trigger_term, trigger_term, trigger_term, flag)
| TAnd (trigger_term, trigger_term, flag)
| TOr (trigger_term, trigger_term, flag)


(* spécifique aux triggers *)
| TType (constr, flag)
| TTerm (constr, flag)
| TTrigVar (trigger_var, flag)
| TAny (flag)
].
```

Nous retrouvons dans le type `trigger_term` presque toute la structure du type `constr` des termes `Coq` en `OCaml`. En effet, il nous faut pouvoir exprimer des *triggers* de la forme « une hypothèse est un *pattern matching* », « le but est une application », etc. Par souci de simplicité, nous avons exclu certaines constructions de `Coq` qui pourront être rajoutées si nécessaire (les niveaux d'univers précis, les entiers natifs, les métavariables existentielles, les projections primitives, etc.). En effet, notre ordonnanceur reste un prototype, et nous avons pensé que dans un premier temps, ne pas considérer des tactiques qui seraient, par exemple, déclenchées par des conditions sur des niveaux d'univers ne serait pas très restrictif. Néanmoins, nous distinguons *Set*, *Prop* et *Type* dans le type `trigger_sort`.

À ces *triggers* s'ajoutent des constructions fréquentes mais non primitives de `Coq`, comme le `trigger_term` appelé `TEq` pour l'égalité, ou `TAnd` et `TOr` que nous avons déjà vus. Il est en effet plus commode d'écrire `TEq _ _ _` que `TApp (TApp (TApp TEq _)_)_`.

Tous ces constructeurs attendent des *flags* puisque nous avons toujours besoin de savoir si l'interprétation du *trigger* renverra ou non les termes considérés.

Par ailleurs, le type `trigger_term` comporte aussi des constructeurs spécifiques aux *triggers*, que nous allons détailler. Les voici :

Le premier, `TType`, signifie qu'un certain terme doit avoir le type donné en premier argument au constructeur `TType`. Par exemple, le *trigger* : « une hypothèse contient un terme de type *Set* que la tactique prendra en argument » est : `TContains TSomeHyp (TType Set (Arg id))`.

Le second `TTerm`, a déjà été vu en exemple. Il permet de désigner un terme précis de `Coq`. En outre, il permet de pallier le fait que les *triggers* ne sont pas une contrepartie exacte du type `constr`, et permet donc de décrire des termes complexes si nécessaire.

Le troisième `TTrigVar` intervient quand nous souhaitons désigner une variable de *trigger*. Au-dessus, nous avions volontairement simplifié le *trigger* pour `assumption`. Pour dire « le but est une hypothèse », nous ne pouvons pas dire `TIs TGoal TSomeHyp` puisque cela serait mal typé. Le terme `TIs` est de type `trigger_var -> trigger_term -> trigger`. Le constructeur `TTrigVar` permet donc de désigner des variables de *trigger*, donc des hypothèses, le but, des définitions locales etc, dans le type `trigger_term`. Le *trigger* (presque) correct pour `assumption` est donc : `TIs TGoal (TTrigVar TSomeHyp NotArg)`.

Enfin, nous l'avions déjà mentionné, mais `TAny` intervient quand nous n'avons pas besoin de préciser le terme dans le *trigger*. Cependant, ce terme peut quand même être donné en argument à la tactique dont nous interprétons le *trigger*, donc `TAny` attend un *flag*.

Nous pouvons passer au type `trigger` en lui-même.

```
Ltac2 Type rec trigger := [
  | TIs ((trigger_var*flag), trigger_term)
  | TPred ((trigger_var*flag), constr -> bool) (* prédicat défini par l'
      utilisateur/ice *)
  | TContains ((trigger_var*flag), trigger_term)
  | TConj (trigger, trigger) (* conjonction de deux triggers *)
  | TDisj (trigger, trigger) (* disjonction de deux triggers *)
  | TNot (trigger) (* négation d'un trigger *)
  | TMetaLetIn (trigger, string list, trigger) (* définition locale *)
  | TAlways (* permet de déclencher tout le temps la tactique *)
  | TNever (* ne déclenche jamais la tactique *)].
```

Nous avons déjà vu à quoi servait `TIs` via des exemples. Cependant, précisons qu'il attend un *flag* dont nous n'avons pas parlé jusqu'ici, tout simplement parce que la variable de *trigger* attendue en argument peut elle aussi être un argument pour la tactique considérée.

Le *trigger* `TPred` est réservé aux utilisateur·ices qui auraient besoin d'écrire un prédicat `Ltac2` arbitrairement compliqué au sein d'un *trigger*. La variable de *trigger* considérée (donc une hypothèse, le but, etc.) devra donc vérifier ce prédicat pour que la tactique associée au *trigger* soit déclenchée.

Le *trigger* `TContains` permet de considérer les *sous-termes* d'une hypothèse, du but etc. Nous l'avons déjà vu au travers d'exemples.

Pour les *triggers* `TConj`, `TDisj`, `TNot`, `TAlways` et `TNever`, nous vous renvoyons au commentaire fait dans l'extrait de code. Une petite précision néanmoins pour `TNever`. À quoi bon écrire un *trigger* qui dit qu'une tactique n'est jamais déclenchée ? Nous vous renvoyons à nos perspectives en 6.4.1 : comme nous aimerions que certaines tactiques en déclenchent d'autres, nous pouvons avoir une tactique dans notre ordonnanceur qui n'est jamais déclenchée initialement (donc son *trigger* est `TNever`), mais qui pourrait l'être via l'action d'une autre tactique.

Enfin, le *trigger* `TMetaLetIn` permet d'avoir des définitions locales dans les *triggers*. Il change aussi le comportement de la fonction d'interprétation, comme nous allons le voir.

### Fonction d'interprétation

L'interprétation d'un *trigger* donne toutes les listes d'arguments possibles pour une tactique, étant donné un but `Coq` précis.

Par exemple, l'interprétation d'un *trigger* pour la transformation qui interprète les types algébriques dans `Sniper` doit renvoyer une liste de singletons, chaque singleton étant un type algébrique du contexte local. En effet, la transformation ne prend qu'un argument qui est le type algébrique en question, et donc, la liste des arguments possibles est une liste de singletons.

Tout d'abord, il faut interpréter les variables de *triggers* :

— Le constructeur `TGoal` est interprété par un singleton : le but `Coq` courant.

— Le constructeur `TSomeHyp` est interprété par la liste des hypothèses *et* des définitions locales, c'est-à-dire leur nom, leur éventuel corps pour les définitions locales, et leur type.

— Le constructeur `TSomeDef` est interprété par la liste des définitions locales, c'est-à-dire leur nom, leur type, et leur corps.

— Le constructeur `TSomeHypProp` est interprété par la liste des hypothèses de type *Prop* de façon similaire. Il permet d'éliminer des hypothèses dont on sait d'emblée qu'elles ne seront pas pertinentes pour l'interprétation d'un *trigger*, et donc, d'être plus efficace.

— Le constructeur `TNamed` est interprété soit par un singleton : un terme qui a été lié précédemment par une définition locale de *trigger*, soit par la liste vide si un tel terme n'existe pas.

Ensuite, les éléments de type `trigger_term` vont être interprétés par rapport à un terme de type `constr`. L'idée est de vérifier s'ils correspondent à un `constr` donné ou non. Il s'agit du rôle rempli par la fonction intermédiaire `interpret_trigger_term_with_constr`, prenant en premier paramètre le terme Coq et en deuxième paramètre le `trigger_term`.

Par exemple, `interpret_trigger_term_with_constr (fun (x : nat)=> x)(TLambda (TTerm nat, Arg type)tDiscard)` renvoie `Some [Set]`. Pour rappel, `TLambda` est le `trigger_term` correspondant aux termes de `Coq` qui sont des fonctions.

En revanche, `interpret_trigger_term_with_constr (fun (x : nat)=> x)(TLambda (TTerm bool, Arg type)tDiscard)` renvoie `None`.

La fonction `type` ici est une fonction `Ltac2` du module `Constr` qui prend en entrée un terme et renvoie son type.

La fonction d'interprétation principale, `interpret_trigger`, permet de faire correspondre l'interprétation des variables de *trigger* (donc des `constr`), à l'interprétation des `trigger_term` qui précisent la forme que doivent avoir les hypothèses, le but, etc.

Soit le *trigger* que l'on note `t`, qui est `TContains (TSomeHyp, Arg id)(TLambda (TTerm nat, Arg type)tDiscard)`, et `G` le but `Coq` :

```
H: (fun (x: nat) => x) 1 = 1
H1: (fun (x: nat) => x) 3 = 3
----------------------------
True
```

L'interprétation de `t` vis-à-vis de `G` fonctionne de la façon suivante :

— On collecte toutes les hypothèses : la variable de *trigger* `TSomeHyp` est interprétée par `[(H, None, (fun (x: nat)=> x)1 = 1); (H1, None, (fun (x: nat)=> x)3 = 3)]`.

— On regarde le constructeur de tête du *trigger* : comme ici, il s'agit de `TContains`, il faut examiner tous les sous-termes des hypothèses, et appliquer la fonction `interpret_trigger_term_with_constr` à chacun d'entre eux. À chaque hypothèse correspond donc sa liste de sous-termes (les sous-termes ouverts sont acceptés mais ne peuvent jamais être utilisés en tant qu'arguments d'une tactique).

— Pour chaque sous-terme `st` de `H` et `H1`, on calcule `interpret_trigger_term_with_constr st t`. On obtiendra donc `None` pour tous les sous-termes sauf pour `fun (x: nat)=> x`, où le résultat sera `Some [Set]`.

— Comme le *trigger* indique que l'hypothèse dans laquelle on a trouvé un sous-terme adéquat doit être donnée en argument de la tactique, le résultat final est : `[[H; Set]; [H1; Set]]`

Pour un interpréter un *trigger* qui est une disjonction de deux *triggers* `t1` et `t2`, on interprète le premier *trigger*. Si le résultat est `[]`, on interprète `t2` et on renvoie le résultat de l'interprétation de `t2`. Sinon, on renvoie le résultat de l'interprétation de `t1`.

Pour interpréter une tactique dont le *trigger* est une conjonction de deux *triggers* `t1` et `t2`, on interprète les deux *triggers* séparément. Mettons que l'on obtienne `l1` et `l2`. Dans ce cas, il faut savoir comment obtenir les arguments à donner à la tactique à partir des deux listes `l1` et `l2`. L'idée est de faire le produit cartésien des deux listes de listes.

Par exemple, le produit cartésien de `[[1; 2; 3] ; [4; 5; 6]]` et `[[7; 8; 9] ; [10; 11; 12]]` est `[[1; 2; 3; 7; 8; 9]; [1; 2; 3; 10; 11; 12]; [4; 5; 6; 7; 8; 9]; [4; 5; 6; 10; 11; 12]]`.

Remarquons que le produit cartésien d'une liste vide avec une liste non vide est vide, ce qui est cohérent avec la sémantique de la conjonction.

Enfin, le cas difficile est celui de l'interprétation de la définition locale de *trigger*, le `TMetaLetIn : trigger -> string list -> trigger`. Supposons que nous voulions interpréter `TMetaLetIn t1 ["x1"; "x2"] t2`. L'idée est d'interpréter le *trigger* `t1` qui forme le corps de la définition locale. Nous obtenons soit `[]`, et dans ce cas l'interprétation complète vaudra `[]`, soit une liste de listes de `constr` non vide (mettons que l'interprétation vaut `l1`). Cette fois, chaque liste parmi la liste de listes `l1` ne formera pas les arguments de la tactique, mais des arguments auxquels nous allons faire correspondre des variables nommées (ici, `"x1"` et `"x2"`). Par exemple, si `l1 = [[nat; false]; [bool; true]]`, alors on enregistre dans un environnement `e` la paire `("x1", nat)` et la paire `("x2", false)`. Si jamais les listes de `l1` avaient plus que deux éléments, cela signifierait que `t1` a mal été écrit. En pratique, cela reviendrait à oublier les éléments supplémentaires.

Le *trigger* `t2` est interprété avec cet environnement `e`. Supposons que son interprétation renvoie `l2`, on a encore une manière de lier les variables `"x1"` et `"x2"`, puisque l'interprétation de `t1` donnait deux listes, donc deux façons possibles de lier les variables. On peut donc interpréter `t2` avec le nouvel environnement `e'`. Si cette interprétation dans `e'` vaut `l2'`, le résultat de l'interprétation complète sera `l2++l2'`.

Pour résumer, `interp_trigger (TMetaLetIn t1 lnames t2)` interprète le *trigger* `t1` et renvoie toutes les manières de lier les variables par les noms de `lnames`. Ensuite, on concatène toutes les interprétations possibles de `t2`, qui dépendent de la façon dont on a lié les variables.

**Exemple 45** Poser une définition locale

Considérons une transformation dont le rôle est de poser une définition locale de la forme $f := \lambda x,\ \ldots$ quand on lui donne une fonction anonyme en argument (et qui replie ensuite cette définition $f$ dans les autres hypothèses et le but).

Celle-ci semble à première vue avoir un *trigger* simple : il suffit qu'une $\lambda$-abstraction apparaisse dans le but ou dans une hypothèse (nous allons nous concentrer sur les hypothèses).

Le *trigger* en question est donc :

```
TContains (TSomeHyp, Arg id)(TLambda tDiscard tDiscard, Arg id)
```

Le deuxième `Arg id` indique, bien sûr, que la tactique doit prendre en argument le corps de la fonction dont elle veut poser une définition locale.

Mais on remarque que ce *trigger* conduit à donner des noms aux fonctions dans les branches des constructions `match`, ce qu'on ne souhaite pas faire.

Un *trigger* plus adapté est donc :

```
TMetaLetIn
  (TContains (TSomeHyp, Arg type) (TLambda tDiscard tDiscard (Arg id
     )))
  ["H"; "f"]
  (TConj
    (TNot
      (TMetaLetIn
        (TContains (TNamed "H", NotArg) (TCase tDiscard tDiscard None
             (Arg id)))
        ["c"]
        (TContains (TNamed "c", NotArg) (TTrigVar (TNamed "f") (Arg
           id)))))
    (TIs (TNamed "f", Arg id) tDiscard))
```

Comment lire ce *trigger* ? Tout d'abord, on détecte une hypothèse contenant une fonction anonyme, on nomme cette hypothèse $H$ et la fonction $f$ (premier *trigger* du premier `TMetaLetIn`).

Ensuite (second *trigger*), on se trouve face à une conjonction de deux *triggers*. On vérifie que cette hypothèse $H$ ne contient pas de *pattern matching*. Si elle n'en contient pas, on passe au second *trigger* de la conjonction. Si elle en contient, on nomme ce *pattern matching* $c$, et on vérifie que $c$ ne contient pas $f$.

Le second *trigger* de la conjonction est juste une astuce pour récupérer le corps de la fonction anonyme. En effet, le *trigger* `TNot` transforme `[]` en `[[]]`. Il n'a pas accès aux arguments potentiels de la tactique. Pour récupérer la fonction anonyme, nous avons donc introduit un *trigger* trivial dont c'est le seul rôle.

**Tableau des *triggers* pour les transformations de `Sniper`** Finissons avec des exemples de *triggers* pour les transformations qui font partie de la tactique `snipe` (nous n'indiquerons pas tout ici). Dans le tableau, on notera `TDisj` en notation infixe `\/` et `TDiscard` sera noté `_`. Ce tableau n'est pas exhaustif, il s'agit d'exemples.

Le lecteur ou la lectrice remarquera que certaines transformations comme l'élimination du polymorphisme prénexe sont considérées comme globales, elles ne prennent alors aucun argument et ne se déclenchent qu'une fois. Nous aimerions les rendre plus atomiques, mais cette tâche fait partie de nos perspectives. Les *triggers* sont donc parfois `TAlways`, alors que nous aimerions obtenir, à terme, des *triggers* plus précis.

| Transformation | Description informelle du *trigger* | *Trigger* |
|---|---|---|
| Réflexivité des constantes (voir 4.1.2) | Se déclenche sur toutes les constantes dans les hypothèses ou dans le but et sur les définitions locales | `TIs (TSomeDef, (Arg id))` `\/` `TContains (TSomeHyp, NotArg) (TConstant None (Arg id))` `\/` `TContains (TGoal, NotArg) (TConstant None (Arg id))` |
| Dépliage des réflexivités (voir 4.1.2) | Se déclenche sur les égalités (ici, on choisit une sur-approximation du *trigger*) | `TIs (TSomeHyp, Arg id) (TEq _ _ _ NotArg)` |
| Égalités d'ordre supérieur (voir 4.1.3) | Se déclenche sur les égalités d'ordre supérieur | `TIs (TSomeHyp, Arg id) (TEq (TProd _ _ NotArg) _ _ NotArg)` |
| Points fixes anonymes (voir 4.1.4) | Se déclenche sur les points fixes | `TContains (TSomeHyp, Arg id) (TFix _ _ NotArg)` |
| Types algébriques (sauf principe de génération) (voir 4.2) | Se déclenche sur les inductifs dans les hypothèses ou le but | `TContains (TGoal, NotArg) (TInd None (Arg id))` `\/` `TContains (TSomeHyp, NotArg) (TInd None (Arg id))` |
| Polymorphisme prénexe (voir 4.4) | Se déclenche une seule fois | `TAlways` |

**Une remarque sur `auto`**

Dans `Coq`, `auto` est une tactique paramétrable par des `Hints`, qui sont soit des lemmes à appliquer, soit des tactiques. Notamment, la commande `Hint Extern pat => tac` prend en entrée le *pattern* `pat` et déclenche la tactique `tac` si le but correspond bien au *pattern* donné.

Ce mécanisme est assez similaire à nos *triggers*, mais il ne permet pas à `pat` d'être un sous-terme du but (autrement dit, `pat` ne peut pas se trouver dans un contexte), et il faut recourir à du code `Ltac` pour utiliser ce même mécanisme sur les hypothèses (ce que nous souhaitions éviter). Ainsi, les possibilités sont moindres avec cet usage de `Hint Extern` qu'avec nos *triggers*.

### 5.2.3 Les filtres

**Présentation des filtres** Rappelons qu'en 5.1.3, nous avions vu que certaines transformations ne devaient pas se déclencher sur certains arguments. Dans l'ordonnanceur, cela se traduit par un nouveau type de donnée `Ltac2`, appelé un *filtre*, qui permet de raffiner un *trigger*.

Si une transformation est déclenchée par un *trigger*, ses arguments doivent aussi passer le filtre, c'est-à-dire que ces derniers ne doivent pas être égaux à certains termes et/ou ne doivent pas vérifier un certain prédicat `p: constr -> bool`, écrit par l'utilisateur·ice en `Ltac2`.

Le type de donnée des filtres est beaucoup plus simple que celui des *triggers*, le voici :

```
Ltac2 Type rec filter := [
  | FConstr (constr list)
  | FPred (constr -> bool)
  | FConstrList (constr list list)
  | FPredList (constr list -> bool)
  | FConj (filter, filter)
```

```
| FTrivial ].
```

À quoi servent chacun de ces filtres ?

— Le filtre `FConstr` empêche une tactique de prendre en argument l'un des termes qui fait partie de la liste donnée à `FConstr`.

— Le filtre `FPred` ne vaut que pour les tactiques n'attendant qu'un argument. La tactique ne sera pas déclenchée sur un argument `t`, si son filtre est `FPred p` et si `p t = true`.

— Le filtre `FConstrList` appliqué à une liste de liste `ll` vérifie que la liste d'arguments donnée à la tactique n'est pas égale à l'une des listes de `ll`.

— Le filtre `FPredList` vérifie que la liste d'arguments `l` ne vérifie pas le prédicat de type `constr list -> bool` donné en argument.

— Le filtre `FConj` permet de combiner deux filtres : les arguments de la tactique dont le filtre est `FConj f1 f2` doivent passer les deux filtres `f1` et `f2`.

— Le filtre `FTrivial` est un filtre que tous les arguments passent. Ainsi, dès lors qu'un *trigger* active une tactique au filtre `FTrivial`, elle est forcément déclenchée, quels que soient les arguments renvoyés par l'interprétation du *trigger* en question.

**Exemples de filtres** Nous avons vu à l'exemple 43 que la transformation qui déplie les définitions et celle qui élimine l'ordre supérieur « prénexe » interagissaient mal, et la solution déployée dans la première version de `snipe`.

Une autre solution utilisant les filtres a été choisie dans l'ordonnanceur. Au lieu d'utiliser la transformation qui déplie les définitions (voir 4.1.2 pour rappel), trois autres transformations très simples sont utilisées.

La première est celle de la réflexivité des constantes et des définitions locales : on ne prouve pas les réflexivités des symboles qui sont interprétés par le prouveur SMT, ni les réflexivités des constantes d'ordre supérieur. Le filtre de cette transformation sera donc : `FConj (FConstr [Z.add; Nat.add; ...])(FPred higher_order)`.

La deuxième transformation déplie le terme à droite dans toute égalité qui est une réflexivité (en essayant la tactique `unfold` qui fait de la $\delta$-réduction). Comme son *trigger* est toute hypothèse qui est une égalité, son filtre va l'empêcher de se déclencher sur les hypothèses qui ne sont pas des réflexivités.

La troisième transformation déplie une constante précise à droite d'une égalité, et, grâce au filtre, nous le faisons seulement dans les égalités de la forme $c = ...$ où $c$ est une variable locale ou de section, et où la constante dépliée $d$ est d'ordre supérieur. Le filtre en question empêche toute paire d'arguments ne vérifiant pas ces prédicats, il s'agit donc de :

```
Ltac2 filter_unfold_in () :=
FPredList (fun l => match l with | [x; y] =>
  Bool.or
  (let t := type x in
  match! t with
    | @eq ?a ?u ?v => Bool.neg (Constr.is_var u)
  end) (Bool.neg (higher_order y)) | _ => true end)
```

Bien sûr, cet exemple peut paraître compliqué et les filtres en jeu ne sont pas triviaux. Mais, le résultat est que nous n'avons plus à nous poser le problème des constantes à déplier ou non au sein même des transformations, et à coder en dur dans les transformations comment elles se comportent. Si jamais d'autres manières de déplier les définitions s'avèrent plus pertinentes pour un autre backend, ceci peut être intégré dans les filtres des transformations (et/ou dans leurs

*triggers*). Nous avons abouti à cette solution avec trois transformations aux filtres complexes afin d'avoir des transformations épurées de la question des arguments auxquels elles s'appliquent. Les deux transformations de dépliage remplacent la transformation des définitions, ce qui permet d'avoir deux modes de dépliage, selon si de l'ordre supérieur est présent ou non.

Voyons comment les trois tactiques interagissent avec celle qui gère l'ordre supérieur « prénexe ». Supposons que dans le but `Coq` apparaisse le terme $map\ f$.

La tactique en charge de l'ordre supérieur génère la définion locale $map_f := map\ f$. La réflexivité des constantes intervient ensuite pour prouver l'énoncé $H : map_f = map_f$ et l'ajouter au contexte local. La tactique qui déplie à droite les égalités réflexives transorme $H$ de façon à obtenir $H : map_f = map\ f$. La réflexivité de la constante $map$ n'est jamais rajoutée au contexte local puisque $map$ est d'ordre supérieur : le filtre bloque la tactique. Mais la dernière tactique dont le filtre fait qu'elle ne déplie que les constantes d'ordre supérieur permet que $map$ ait sa définition dépliée au sein de l'égalité $H$. Nous obtenons alors $H : map_f = (fix\ map...)\ f$ qui est exactement ce que nous souhaitions.

## 5.3 Implémentation de l'ordonnanceur

Pour rappel, la tactique `Ltac2 orchestrator` prend en entrée une liste de triplets : les tactiques, avec leur *trigger* et leur filtre, et se charge de les appliquer automatiquement avec les bons arguments.

Pour ce faire, `orchestrator` interprète le *trigger* de la première tactique, `tac`, de la liste. Si `None` est obtenu, il passe à la deuxième tactique. Si `l::ll` est obtenu, `orchestrator` prend le premier élément `l` de `l::ll`, qui est, on le rappelle, une liste de termes qui sont les arguments de la tactique. Par ailleurs, il vérifie que la tactique n'a pas déjà été appelée sur ces arguments `l`. En effet, par défaut, une tactique ne peut jamais être appelée deux fois avec les mêmes termes. En particulier, une tactique ne prenant pas d'argument (c'est-à-dire, prenant la liste vide en argument) ne peut se déclencher qu'une fois. L'utilisateur·ice peut désactiver ce comportement par défaut.

Si, parmi la liste `l::ll`, il n'existe aucune liste `l'` d'arguments sur laquelle la tactique n'a pas été appelée, l'ordonnanceur passe à la tactique suivante. Mais si on trouve bien une telle liste `l'`, l'ordonnanceur appelle une fonction auxiliaire `run tac l'` qui permet d'exécuter la tactique `tac` sur les arguments `l'`. Une précision que nous n'avons pas donnée est que les tactiques peuvent créer plusieurs sous-buts. Ainsi, l'ordonnanceur prend aussi en entrée des précisions sur les tactiques : le numéro des sous-buts sur lesquels l'ordonnanceur doit continuer de s'appliquer si jamais la tactique crée plusieurs sous-buts. Si jamais les numéros ont mal été renseignés par l'utilisateur·ice, par défaut, l'ordonnanceur s'applique sur *tous* les sous-buts créés par la tactique.

Une fois que la tactique `tac` a été exécutée, il faut calculer *ce qui a changé* dans le but. L'ordonnanceur va s'exécuter dans un but restreint, où ne sont considérées que les hypothèses qui sont apparues ou qui ont changé (et où la tactique qui vient de s'exécuter reste dans les arguments, parce qu'elle sera potentiellement déclenchée à nouveau). Cela permet de limiter les calculs qui sont faits : rappelons que dans la première version de `snipe`, certaines tactiques scannaient systématiquement *tout* le contexte local pour savoir où s'appliquer, alors même que l'exécution de la tactique qui les précédaient n'avait modifié qu'une seule hypothèse.

Une fois que l'ordonnanceur a exécuté toutes les tactiques possibles, c'est-à-dire que l'interprétation de chaque *trigger* renvoie `[]` (ou que les tactiques ont déjà été appelées sur les arguments qui auraient pu les déclencher), alors nous avons deux possibilités : (1) soit l'exécution s'arrête, car tout le but était examiné (2) soit l'ordonnanceur reprend son exécution avec toutes les tactiques initiales, mais en considérant tout le but, et plus sur la fraction du but qui a changé suite

à l'exécution d'une tactique.

Notons que l'implémentation de l'ordonnanceur a été facilitée par la présence de *références locales* en `Ltac2`. Autrement dit, les tactiques `Ltac2`, contrairement aux fonctions `MetaCoq` par exemple, n'ont pas besoin de renvoyer tous les arguments qu'elles modifient. Elles peuvent aussi changer des valeurs dans les références locales. L'état de l'ordonnanceur, c'est-à-dire la fraction du but changée par l'exécution de la tactique précédente, les sous-termes des hypothèses (pour éviter de les calculer plusieurs fois), les arguments sur lesquels les tactiques ont déjà été appelées, etc, est une référence locale.

Une dernière remarque à faire est que l'ordonnanceur ne déclenche jamais les tactiques considérées comme globales (parce qu'elles n'attendent aucun argument) dans un état local, c'est-à-dire un état où tout le contexte de preuve n'est pas considéré. Par exemple, l'élimination du polymorphisme est globale (parce qu'elle a besoin de tout le contexte de preuve pour calculer les instances), et ne s'active donc jamais que sur *tout* le contexte de preuve et pas sur une hypothèse en particulier.

Pour des exemples d'exécution de `orchestrator`, nous vous renvoyons à la section suivante 5.4.

## 5.4 Preuve de concept : la tactique `snipe` à l'œuvre

Des exemples pour le fonctionnement de l'ordonnanceur sont disponibles ici : https://github.com/louiseddp/sniper/blob/for-phd-orchestrator/examples/examples.v.

Reprenons d'abord le théorème de l'introduction, dans l'exemple 2. On suppose que le type `A` a été introduit par une variable de section, et qu'une autre variable de section `HA: CompDec A` est présente. En effet, pour rappel, `SMTCoq` ne permet de prouver des buts qu'au sujet de types appartenant à la classe de type `CompDec`.

```
Theorem app_eq_unit (x y:list A) (a:A) :
x ++ y = [a] -> x = [] /\ y = [a] \/ x = [a] /\ y = [].
```

La tactique `snipe` est capable de prouver cet énoncé complètement automatiquement.

Comment l'ordonnanceur (paramétré pour SMTCoq) préprocesse ce but `Coq`, de manière à ce que la tactique `verit` puisse le prouver ?

Tout d'abord, avant d'activer l'ordonnanceur, nous avons décidé de toujours commencer par `intros`. Ici, `intros` ne fait rien, mais il s'avère parfois utile. Par ailleurs, on a dans le contexte local les deux listes, l'élément `a`, ainsi que nos deux variables de section `A` et `HA`.

Ensuite, l'ordonnanceur commence par la première transformation à sa disposition, qui ajoute au contexte local la réflexivité d'une constante donnée. Cette transformation se déclenche sur la fonction `app` (cachée par la notation `++`). Dans le contexte local apparaît l'hypothèse `H : app= app`.

L'ordonnanceur s'applique ensuite dans un contexte restreint en ne considérant que `H`. La première transformation se déclenche à nouveau sur `app`, mais l'ordonnanceur ne la ré-applique pas, car elle a déjà été appelée avec l'argument `app`. La deuxième transformation, qui déplie les réflexivités, est activée. L'hypothèse `H` est modifiée et devient :

```
H : app = fun A : Type =>
fix app (l m : list A) {struct l} : list A :=
match l with
  | nil => m
  | a :: l1 => a :: app l1 m
end
```

L'ordonnanceur se relance sur le contexte restreint ne contenant à nouveau que `H`. Cette fois, la transformation qui est déclenchée élimine les égalités d'ordre supérieur. Nous obtenons :

```
H : forall A l m, @app A l m =
  (fix app (l m : list A) {struct l} : list A :=
  match l with
    | nil => m
    | a :: l1 => a :: app l1 m
  end) l m
```

Puis, c'est au tour de l'élimination des points fixes d'opérer, nous obtenons :

```
H : forall A l m, @app A l m =
match l with
  | nil => m
  | a :: l1 => a :: app l1 m
end
```

La prochaine transformation dont le *trigger* permet le déclenchement sur `H` est l'élimination du *pattern matching*. Nous obtenons les deux équations de `app` :

```
H1 : forall A m, @app A [] m = m
H2 : forall A m a l1, @app A (a::l1) m = a :: app l1 m
```

L'ordonnanceur continue son exécution, et la prochaine transformation déclenchée est l'interprétation des types algébriques, donnant deux énoncés sur les listes.

```
H3 : forall (A: Type) (a : A)(xs : list A),
[] = x :: xs -> False
H4 : forall (A: Type) (x x' : A) (xs xs' : list A),
x :: xs = x' :: xs' -> x = x' /\ xs = xs'
```

La transformation donnant le principe de génération n'est pas déclenchée. En effet, comme les énoncés qu'elle génère nécessitent un traitement particulier du fait que nous avons besoin de valeurs par défaut (voir 6.4.1), cette transformation est considérée comme globale.

Puisqu'aucune autre transformation n'est déclenchée (la monomorphisation est aussi une tactique globale qui ne sera jamais activée dans un état où tout le but n'est pas examiné par l'ordonnanceur), l'ordonnanceur reprend son calcul avec toutes les transformations, et sur l'entiereté du but `Coq`.

La nouvelle transformation qui s'active est le principe de génération (qui, dans l'état actuel, génère directement un énoncé instancié, avec des habitants).

Nous obtenons une nouvelle hypothèse et deux variables (les fonctions de projection, dont nous avons fait en sorte de supprimer le corps de leur définition via la tactique `clearbody` pour ne pas activer inutilement les tactiques qui déplient les définitions) :

```
proj_list1 : forall (A: Type), A -> list A -> A
proj_list2 : forall (A: Type), list A -> list A -> list A
H5: forall (l : list A), l = [] \/ @proj_list1 A a l :: @proj_list2 A [] l
```

Dans l'état local de l'ordonnanceur ne considérant que les projections et `H5`, rien n'est déclenché. L'état considéré par l'ordonnanceur est de nouveau tout le contexte local, et l'élimination du polymorphisme est déclenché, nous permettant d'instancier toutes les hypothèses par `A`.

Le contexte local final est :

```
A : Type
HA : CompDec A
x : list A
y : list A
a : A
H1 : forall m, @app A [] m = m
H2 : forall m a l1, @app A (a::l1) m = a :: @app A l1 m
H3: forall (a : A)(xs : list A),
[] = x :: xs -> False
H4: forall (x x' : A) (xs xs' : list A),
x :: xs = x' :: xs' -> x = x' /\ xs = xs'
proj_list1 : forall (A: Type), A -> list A -> A
proj_list2 : forall (A: Type), list A -> list A -> list A
H5: forall (l : list A), l = [] \/ @proj_list1 A a l :: @proj_list2 A [] l
------------------------------------
x ++ y = [a] -> x = [] /\ y = [a] \/ x = [a] /\ y = []
```

Enfin, l'ordonnanceur ne trouve plus de *trigger* dont l'interprétation ne serait pas `[]` ou ne donnerait pas des arguments déjà utilisés par la tactique. C'est donc la fin de l'étape de préprocessing : le prouveur automatique `veriT` est appelé par la tactique `verit`, et le but est résolu automatiquement.

Maintenant, reprenons l'exemple 3. Le voici à nouveau (avec cette fois, `A`, `B` et `C` vues comme des variables de section qui appartiennent à la classe de type `CompDec`) :

```
Lemma map_compound : forall (f : A -> B) (g : B -> C) (l : list A),
map g (map f l) = map (fun x => g (f x)) l
```

Cet exemple utilise de l'ordre supérieur et des fonctions anonymes. Il nécessite une induction qui doit être faite manuellement, nous devons donc commencer par `intros f g l ; induction l`.

Dans le code actuel de l'ordonnanceur, il faut noter que les transformations gérant les fonctions anonymes et l'ordre supérieur sont globales et ne s'activent qu'une fois. Leur *trigger* est `TAlways`, ce qui signifie qu'elles se déclencheront sur tout le contexte de preuve et sans argument, et traiteront respectivement toutes les fonctions anonymes et toutes les fonctions d'ordre supérieur d'un seul coup.

Afin d'obtenir un code plus simple pour ces transformations, nous souhaiterions rendre ces transformations plus atomiques, et permettre leur activation plusieurs fois sur des arguments précis. Nous laissons la simplification de ces transformations à nos perspectives.

La première transformation à s'activer concerne les fonctions anonymes et permet d'ajouter dans le contexte local :

`f0 := fun x => g (f x)`

Par ailleurs, la définition de `f0` est repliée dans le but, faisant disparaître la fonction anonyme `fun x => g (f x)`.

La seconde transformation déclenchée est la transformation qui gère l'ordre supérieur. Elle va créer trois définitions locales, et replier ces trois définitions :

```
f1 := map f0
f2 := map f
f3 := map g
```

La réflexivité des constantes s'active sur `f0` et nous obtenons : `H0: f0 = f0`, qui deviendra `H0 : f0 = fun x => g (f x)` puis `H0 : forall x, f0 x = f (g x)` après l'appel du dépliage des réflexivités et de l'élimination des égalités d'ordre supérieur sur `H0`.

Puis, la réflexivité des constantes s'active sur `f1` et nous obtenons : `H: f1 = f1`. La transformation qui déplie les réflexivités transforme `H` en `H: f1 = map f0`. La transformation qui déplie certaines constantes dans une hypothèse, et dont le filtre ne lui permet ne s'appliquer que sur des constantes d'ordre supérieur, va déplier `map` dans `H`, qui devient :

```
H: f1 =
(fun A B => fix map (l : list A) : list B :=
  match l with
  | nil => nil
  | a :: t => f a :: map t
  end) A B f0
```

Nous avons déjà vu comment la suite des transformations (égalités d'ordre supérieur, point fixe etc) s'appliquaient. À la fin de cette étape, les équations de `f1` sont prouvées et apparaissent dans le contexte local.

```
H1 : f1 [] = []
H2 : forall x xs, f1 (x::xs) = f0 x :: f1 xs
```

Par ailleurs, un traitement similaire est fait pour `f2` et `f3`, donc nous n'allons pas le détailler. Ceci fait, l'ordonnanceur appelle l'interprétation du type algébrique `list` qui apparaît dans cet exemple aussi. Enfin, il déclenche l'élimination du polymorphisme, et tous les énoncés polymorphes (donc les énoncés sur le type `list`) vont être instanciés par `A`, `B` et `C`.

Le contexte de preuve dépend du cas dans lequel nous nous trouvons (puisque nous avons commencé la preuve par une induction), il ne sera pas détaillé pour plus de concision. Toutefois, à la fin de ce prétraitement, la tactique `verit` est capable de résoudre les deux sous-buts automatiquement.

# Chapitre 6

# CONCLUSION ET PERSPECTIVES

## 6.1 Conclusion

Cette thèse propose une nouvelle méthodologie d'automatisation pour l'assistant de preuve `Coq`. Plus particulièrement, elle offre la possibilité d'un préprocessing compositionnel, où plusieurs petites transformations logiques (sous la forme de tactiques `Coq`), sont combinées, jusqu'à atteindre la logique cible d'un prouveur externe.

Nous avons implémenté une bibliothèque de transformations logiques et un ordonnanceur de transformations, capable, à partir de la description de ce qui les déclenche, de les combiner automatiquement. Grâce à l'ordonnanceur, il est aisé de rajouter une nouvelle transformation à l'ensemble.

Cette thèse montre que cette méthodologie est viable grâce à l'implémentation d'une tactique « pousse-bouton » appelée `snipe` qui combine certaines des transformations que nous avons implémentées.

Cependant, comme dans la seconde partie de la thèse, nous nous sommes concentré·es sur les moyens de combiner les transformations déjà écrites et pas sur l'implémentation de toutes les transformations nécessaires, la tactique `snipe` ne peut pas encore rivaliser avec des tactiques d'automatisation telles que `hammer` ou `synth` du plugin `Tactician` [5].

En revanche, comme la méthodologie de l'ordonnanceur est générique, elle devrait s'adapter à plusieurs backends pour `Coq` voire pour `OCaml`, et dans d'autres domaines que l'automatisation. Ce sont les axes majeurs que nous souhaiterions poursuivre.

## 6.2 Amélioration des transformations actuelles

### 6.2.1 Les relations inductives

Pour l'instant, nous ne décidons automatiquement qu'un petit fragment des relations inductives (voir 4.3). Les raisons en sont multiples, et chacune d'entre elles est un axe d'amélioration pour cette transformation. Nous les listons donc ici et expliquons chaque point d'amélioration indépendamment.

— **La conclusion d'un constructeurs doit mentionner tous les arguments de ce dernier :** cette limitation importante sera peut-être surmontée en étudiant les prédicats

bidirectionnels et en poursuivant le travail qui a été fait dans `QuickChick`. Nous avions déjà mentionné cette perspective dans l'exemple 32, nous vous y renvoyons.

— **La preuve est générée par un script `Ltac2` :** il est sans doute possible de générer directement un lambda-terme qui suit la structure du point fixe et qui serait la preuve d'équivalence entre la relation inductive et le point fixe généré. Cela serait plus robuste qu'un script de preuve `Ltac2`, mais cela serait aussi beaucoup plus long à implémenter.

— **L'heuristique pour vérifier si la fonction termine est trop simple :** En utilisant le plugin `Equations` [41] qui permet de faire de la récurrence bien fondée, on pourrait générer des points fixes dont la terminaison n'est pas évidente. Cela demanderait une réécriture du code qui utiliserait les fonctionnalités de ce plugin pour écrire des points fixes plus complexes. Utiliser `Equations` permettrait aussi de simplifier notre code : nous générons actuellement des *pattern matchings* imbriqués et nous n'aurions plus à le faire.

— **L'implémentation en `MetaCoq` n'est pas satisfaisante :** Comme le langage de `MetaCoq` est tout simplement `Coq`, nous n'avons pas d'effet de bord. En particulier, nous n'avons pas de référence : les fonctions doivent renvoyer tous les arguments qu'elles modifient, et prendre en entrée tous les arguments dont elles ont besoin. Cela conduit à des fonctions prenant un nombre très grand d'arguments et renvoyant des enregistrements ou des tuples, ce qui compromet la lisibilité du code. Je pense que `MetaCoq` n'était pas le métalangage le plus adapté et qu'une réécriture en `Ltac2` serait souhaitable.

### 6.2.2 Les points fixes anonymes

La transformation qui gère les points fixes anonymes (voir 4.1.4) pose encore quelques difficultés dans le cas d'une réduction de point fixe avec de l'ordre supérieur.

Pour le voir, supposons que l'on ait dans notre contexte des types, une fonction, une définition locale et une hypothèse telles que :

```
A : Type
B : Type
f : A -> B
f0 := map f
H : forall (l : list A),
  f0 = (fix map (l : list A) :=
  match l with
    | [] => []
    | x :: xs => f x :: map xs) l
```

En interne l'élimination des points fixes anonymes va générer et prouver une hypothèse intermédiaire :

```
H1 : forall (A B : Type)(g : A -> B) (l : list A),
(fun A B g =>
fix map l := ...) A B g =
match l with
  | [] => []
  | x :: xs => g x :: map g xs
end
```

Or, en réécrivant `H1` dans `H`, on fait réapparaître de l'ordre supérieur, car au lieu d'avoir le terme `f0 xs`, on a le terme `map f xs` qui apparaît dans la deuxième branche du *pattern matching*. De manière *ad hoc*, la transformation replie la définition de `f0` dans ce cas précis.

Une façon plus générique de procéder serait de réécrire la transformation et de lui faire poser une définition locale ayant pour corps précisément le point fixe à replier, et de n'accepter l'élimination des points fixes qu'à *toplevel*.

Ici, la définition locale serait donc :

```
f_fix :=
(fix map (l : list A) :=
  match l with
    | [] => []
    | x :: xs => f x :: map xs)
```

Par ailleurs, on replierait la définition de `f_fix` dans `H` :

```
H : forall (l : list A),
f0 = f_fix l
```

D'autres transformations de `Sniper` permettraient d'aboutir à l'hypothèse suivante concernant `f_fix` :

```
H2 : forall l, f_fix l =
  (fix map (l : list A) :=
    match l with
      | [] => []
      | x :: xs => f x :: map xs) l
```

Comme cette fois-ci, le point fixe est à *toplevel*, l'élimination du point fixe s'activerait, et `H2` deviendrait :

```
H2 : forall l, f_fix l =
  match l with
    | [] => []
    | x :: xs => f x :: f_fix xs
  end
```

Nous n'avons pas eu le temps d'implémenter cette solution, c'est pourquoi nous la laissons dans nos perspectives.

## 6.3 Nouvelles transformations

### 6.3.1 Encodage des applications partielles

Il existe une transformation très classique qui permet d'éliminer les applications partielles, que nous avons choisi de ne pas implémenter dans `Sniper`, mais qui pourrait l'être. Il s'agit de l'encodage applicatif, dont nous allons exposer le principe.

À chaque occurrence d'application partielle, mettons, une fonction $f$ d'arité $n > 1$, appliquée à des termes $t_1$, ..., $t_k$ avec $k < n$, on ajoute un symbole $App_{f\ t_1 \ldots\ t_k}$, d'arité $n - k$ et l'axiome suivant : $\forall x_{k+1} \ldots x_n,\ App_{f\ t_1 \ldots t_k}\ x_{k+1} \ldots\ x_n = f\ t_1 \ldots\ t_k\ x_{k+1} \ldots\ x_n$

**Exemple 46** Applications partielles

Supposons qu'un but `Coq` contienne le terme `fst 1` avec : `fst := fun (x: nat)(y : nat)=> x`. La transformation doit poser : `App_fst_1 := fst 1` et replier toutes les occurrences de `fst 1`. Enfin, on ajoute l'hypothèse `H: forall (y: nat), App_fst_1 y = fst 1 y`.

### 6.3.2 Défonctionnalisation

De manière très similaire, nous aurions pu implémenter la *défonctionnalisation*.

À chaque fois qu'une fonction $f : A \to B$ est utilisée comme un argument d'un autre terme d'ordre supérieur, on remplace ce dernier par un terme du premier ordre.

Par exemple, si on traite le terme $@map\ A\ B\ f\ l$, on introduit un nouveau type $T_f$ pour la fonction $f$, un symbole $f_{defun} : T_f$ et on remplace $@map\ A\ B : (A \to B) \to list\ A \to list\ B$ par $@map' : T_f \to list\ A \to list\ B$. Ainsi, le terme $@map\ A\ B\ f\ l$ deviendra $@map'\ f_0\ l$. Enfin, on introduit un symbole applicatif @ et l'axiome : $\forall x, @\ f_{defun}\ x = f\ x$.

Ces deux transformations concernant l'ordre supérieur permettent de traiter plus de cas que la transformation implémentée dans `Sniper` (voir 4.5.2), mais elles sont beaucoup plus longues à implémenter, et alourdissent beaucoup le but initial. Comme notre objectif était de tester la composition de transformations simples, et pas d'implémenter des transformations déjà existantes, nous avons préféré consacrer notre temps à l'articulation de transformations diverses plutôt qu'à l'implémentation d'*une* transformation particulière, qui, de surcroît, n'apporte rien de nouveau à la recherche. Ces deux transformations seront cependant cruciales si `Sniper` doit être étendu.

### 6.3.3 Types dépendants de types

Nous avons traité le polymorphisme prénexe dans `Sniper`. Dans le cas des inductifs dépendant de types avec polymorphisme prénexe, les arguments polymorphes sont des *paramètres* de l'inductif considéré. Pour rappel, cela signifie que le terme $A$ de type $Type$ est fixé pour tout constructeur.

Un travail non achevé dans `Sniper` consiste à s'intéresser aux inductifs présentant des *annotations* et non des *paramètres*. Cet intérêt vient de discussions avec les développeur·euses de `FreeSpec` [30], un plugin pour certifier des calculs impurs en `Coq`, et qui cherchaient à automatiser une partie de leurs preuves `Coq`.

Présentons ce travail par un exemple. Dans `FreeSpec` se trouve un inductif `DOORS` qui modélise un sas. Le premier constructeur sert à savoir si l'une des portes du sas est ouverte, et le second représente un effet de bord, à savoir une action dans le monde physique (la fermeture d'une porte).

```
Inductive door : Type := Left | Right.

Inductive DOORS : Type -> Type :=
| IsOpen : door -> DOORS bool
| Toggle : door -> DOORS unit.
```

L'argument de type `Type` de `DOORS` dépend du constructeur considéré. Une fonction qui prend un terme de type `DOORS x` peut effectuer un *pattern matching* dépendant et renvoyer, selon les branches, des termes de types différents (construits à partir de `bool` ou de `unit`).

Il s'agit d'un encodage en `Coq` d'une construction appelée GADT (*generalized abstract data types*). Cette construction est présente en `OCaml` et permet d'avoir un type polymorphe avec des constructeurs qui instancient différemment la variable de type.

**Exemple 47** Un exemple de GADT en `OCaml`

Pour définir une syntaxe très basique en `OCaml`, avec des termes ne comportant que des entiers, des additions et des applications de ces additions, on peut écrire :

```
type term =
  | Int : int -> term
  | Add : (term -> term -> term) -> term
```

```
  | App : term -> term
```

Mais on aura des termes mal typés dans la syntaxe, comme par exemple `App (Int 3)(Int 2)`. Une solution est d'utiliser les GADTs :

```
type _ term =
  | Int : int -> int term
  | Add : (int -> int -> int) term
  | App : ('b -> 'a) term * 'b term -> 'a term
```

Ici, comme l'application attend forcément une fonction, `App (Int 3)(Int 2)` ne type plus. On note que le `_` signifie que, selon chaque constructeur, ce type polymorphe sera instancié différemment.

L'encodage des GADTs choisi dans `FreeSpec` repose sur le fait d'avoir une variable de type *Type* dans les annotations du constructeur. Considérons un type avec annotation (encore plus simple que l'exemple en `OCaml`, pour éviter les types des fonctions) en `Coq`, et une fonction sur ce type :

```
Inductive trm : Type -> Type :=
  | N : nat -> trm nat
  | B : bool -> trm bool.

Inductive trm_le : forall (A: Type), trm A -> A -> Prop :=
  | le_nat x y : le x y -> trm_le nat (N x) y
  | le_bool x y : Bool.le x y -> trm_le bool (B x) y.
```

Ici, les termes sont soit des entiers, soit des booléens, mais « boxés » par les constructeurs `N` et `B` et nous disposons d'une relation `trm_le` qui permet de comparer ces termes avec un terme du type non boxé.

Dans `Sniper`, nous avons implémenté une première transformation qui efface les dépendances dans ces types. Autrement dit, `DOORS` et `trm` sont transformés en :

```
Inductive DOORS' : Type :=
  | IsOpen' : door -> DOORS'
  | Toggle' : door -> DOORS'.
```

et

```
Inductive trm' : Type :=
  | N' : nat -> trm'
  | B' : bool -> trm'.
```

L'étape suivante est de transformer toute relation inductive (ou fonction) qui utilise des arguments avec le type inductif initial en relation inductive (ou fonction) équivalente mais aux arguments dans le type transformé. Les exemples de `FreeSpec` nous ont inspiré·es, mais nous allons garder l'exemple de `trm_le` ici car il est plus concis.

L'idée est de définir un nouveau prédicat, comme ci-dessous :

```
Inductive trm_le' : trm' -> {nat + bool} -> Prop :=
  | le_nat' x y : le x y -> trm_le' (N' x) (inl y)
  | le_bool' x y : Bool.le x y -> trm_le' (B' x) (inr y).
```

Comme nous n'avons plus d'argument de type *Type* dans `trm'`, nous utilisons un argument supplémentaire d'un type somme qui est la disjonction de tous les types possibles dans les

constructeurs (donc ici, `nat` et `bool`). On choisira le constructeur adéquat du type somme en fonction du type de l'argument qui est attendu dans l'inductif initial.

Par ailleurs, ils nous faut deux fonctions de traduction :

```
Definition trm_to_trm' (A : Type) (t : trm A) :=
  match t with
    | N n => N' n
    | B b => B' b
  end.

Definition elim_sum {A B : Type} (x : A + B) :=
  match x in _ + _ return match x with
          | inl _ => A
          | inr _ => B
          end
with
    | inl y => y
    | inr z => z
end.
```

Ensuite, nous pouvons énoncer un lemme d'équivalence entre `trm_le` et `trm_le'`. Nous avons besoin de traduire les types **et** les termes.

```
Lemma trm_le_trm_le' :
  forall (A: Type) (t: trm A) (x : A) (y : nat + bool),
    JMeq x (elim_sum y) ->
    trm_le A t x <-> trm_le' (trm_to_trm' A t) y.
```

Le lemme assure que les deux prédicats sont équivalents, à condition d'avoir traduit le terme de type $A$ du prédicat initial en terme de type $nat + bool$. Comme ces deux termes ne sont pas du même type, on utilise l'égalité John Major inventée par Conor McBride [32], `JMeq`, aussi appelée égalité hétérogène. Il s'agit d'une relation inductive permettant de comparer deux termes de type différents, ici, un terme de type $A$ et un terme de type $nat + bool$. Plus précisément, voici la définition de `JMeq` :

```
Inductive JMeq (A:Type) (x:A) : forall B:Type, B -> Prop :=
    JMeq_refl : JMeq x x.
```

Pour prouver ce lemme `trm_le_trm_le'`, nous avons besoin d'éliminer les égalités hétérogènes, car ne nous pourrons pas substituer les `x` par les `elim_sum y` comme dans une égalité homogène. Or, un axiome, appelé l'axiome K, est équivalent au théorème suivant, qui permet précisément de remplacer les égalités hétérogènes par des égalités :

```
JMeq_eq : forall (A:Type)(x y:A), JMeq x y -> x = y.
```

L'axiome K nous dit que toutes les preuves d'égalité sont égales à la réflexivité. Cependant, `Coq` ne permet pas de prouver l'axiome K ou la propriété équivalente `JMeq_eq`. La théorie de `Coq` n'a pas été étendue avec ces axiomes car ils seraient incompatibles avec l'axiome d'univalence, un principe fondamental pour la théorie homotopique des types.

Notre transformation ne serait donc utile que pour des utilisateur·ices qui accepteraient l'axiome K. Cependant, comme la propriété décrite par l'axiome K est dérivable sur les types avec une égalité décidable, il serait possible en particulier de l'utiliser pour tous les types satisfaisant `CompDec`.

Dans les cas où l'on obtient des égalités de la forme : `JMeq x y`, on peut utiliser `JMeq_eq` pour conclure `x = y`, et poursuivre la preuve.

Nous avons aussi des cas impossibles, avec `JMeq x y` et `x: bool`, `y: nat`, dont nous pouvons nous sortir sans axiome. Mais il nous faudra démontrer `bool=nat -> False`. Démontrer des égalités ou des inégalités entre types est difficile en toute généralité, et encore plus à automatiser. Notre transformation sera donc réduite à des cas où cette preuve est possible.

Cette transformation a partiellement été implémentée (les inductifs sans annotations et les prédicats équivalents sont générés automatiquement). Pour poursuivre ce travail, il faudrait encore définir automatiquement les fonctions de traduction, les lemmes d'équivalence utilisant les égalités hétérogènes, et les prouver.

Nous avons décidé de ne pas poursuivre ce travail, mais nous le considérons comme une première approche pour une traduction de certains types dépendants dans `Sniper`.

## 6.4 Perspectives pour l'ordonnanceur

### 6.4.1 Ajout d'états

Un élément important à rajouter à l'ordonnanceur mais que nous n'avons pas pu traiter est l'ajout d'états.

#### État global

Nous voudrions parfois que certaines tactiques, après leur application, en déclenchent d'autres, par exemple, sur les énoncés qu'elles viennent de produire.

L'ordonnanceur dispose déjà d'un état global, dans lequel il enregistre les hypothèses et leurs sous-termes, ainsi que les arguments sur lesquels ont déjà été déclenchées certaines tactiques. Mais nous voudrions que les tactiques puisses rajouter une marque sur certains énoncés, qui puisse servir de *trigger* pour d'autres tactiques. Cela permettrait à des tactiques données de modifier l'ordre d'application des tactiques prodigué initialement par l'utilisateur, et/ou à appliquer un traitement spécifique à certains énoncés.

Par exemple, dans `Sniper`, le principe de génération pour les listes sans quantificateur existentiel, s'énonce ainsi (voir 4.2.2) :

$\forall (A : Type)(l' : list\ A)(d : A)(l : list\ A),\ l = [] \vee l = hd\ l\ d\ ::\ tl_{default}\ l\ l'$

Or, pour *cet énoncé particulier*, nous aimerons rechercher des valeurs par défaut, c'est-à-dire des habitants du type $A$ et du type $list\ A$, de manière à simplifier les énoncés. Pour ce faire, il faut d'abord que l'énoncé soit monomorphisé, pour ensuite chercher des habitants de type concrets. Nous aimerions donc que la tactique `gen_principle` puisse inscrire dans l'état global qu'elle déclenche la monomorphisation, puis qu'elle déclenche une transformation qui cherche à instancier les quantificateurs sur $l'$ et sur $d$ par n'importe quelle valeur du bon type.

Dans l'implémentation actuelle de l'ordonnanceur, la transformation `gen_principle` fait toutes ces opérations d'un seul coup. Nous aimerions la rendre plus atomique et la décorréler de la transformation de recherche d'habitants.

Par ailleurs, la monomorphisation s'intercale entre les deux transformations, rendant le code plus monolithique encore. Dans l'implémentation actuelle, nous sommes contraint·es d'appeler, depuis le code de la tactique `gen_principle`, la tactique de monomorphisation, puis la tactique de recherche d'habitant, alors que cela devrait être la tâche de l'ordonnanceur.

Ainsi, nous rajouterions dans l'ordonnanceur la tactique `instantiate_with_any` qui cherche à instancier un quantificateur avec n'importe quelle valeur par défaut, avec un *trigger* dont l'interprétation vaut toujours `[]`. C'est l'intérêt du trigger `TNever` que nous avions introduit au paragraphe 5.2.2. La tactique du principe de génération, une fois déclenchée, écrirait dans l'état global

de l'ordonnanceur qu'il faut appeler la monomorphisation suivie de `instantiate_with_any` sur l'énoncé qu'elle a généré (autant de fois que le nombre de fonctions de projection qu'elle a créé).

#### État local

Nous souhaiterions rajouter également un état interne à chaque tactique. De cette manière, au fur et à mesure de l'évolution du contexte de preuve, une tactique peut rajouter de l'information dans son état et effectuer des opérations en conséquence.

Par exemple, dans l'implémentation actuelle de `snipe`, la tactique de monomorphisation ne prend aucun argument. Elle est globale, elle s'applique sur tout le contexte de preuve d'un seul coup. Elle calcule une fois pour toutes les instances possibles pour des hypothèses données, puis effectue ces instanciations. Cet état de fait n'est pas satisfaisant car, à l'apparition d'une nouvelle hypothèse polymorphe, ou d'une nouvelle instance potentielle, il faut effectuer à nouveau toutes les opérations.

Munie d'un état local, cette transformation de monomorphisation pourrait fonctionner ainsi :

— À l'apparition d'un nouveau terme de type $Type$, la tactique de monomorphisation inscrit dans son état local le contexte dans lequel il apparaît. Par exemple, si $list\ nat$ apparaît, alors on enregistre un contexte identité pour $list\ nat$ et le contexte $\lambda\, T.\ list\ T$ pour $nat$.

— À l'apparition d'une nouvelle hypothèse polymorphe, la tactique de monomorphisation inscrit dans son état local les contextes dans lesquels apparaissent les variables de type. Par exemple, si $H : \forall (A : Type)(l : list\ A), l = [] \to length\ l = 0$ apparaît, la tactique enregistre le contexte $\lambda\, T.\ list\ T$ pour $H$.

— À chaque fois qu'un contexte pour un type concret $A$ et un contexte pour une hypothèse $A$ sont égaux, et que la transformation n'a pas déjà été appelée sur ces arguments précis, on génère l'instance $H\ A$.

Ainsi, à chaque fois que la transformation est déclenchée par son *trigger*, elle peut : (1) mettre à jour son état, (2) créer une nouvelle instance ou (3) les deux.

Avoir un état local pourrait être utile à d'autres tactiques similaires d'instanciation, par exemple, si on cherche à instancier des quantificateurs quelconques et non simplement des hypothèses polymorphes.

### 6.4.2 Une syntaxe plus *user friendly*

Dans la version actuelle de l'ordonnanceur, la syntaxe des *triggers* reste encore assez lourde. Nous avons commencé à intégrer quelques notations grâce au système de notation de `Ltac2`, mais nous n'avons pas encore beaucoup expérimenté avec. Ces notations permettraient d'écrire un *trigger* facilement, avec le moins de connaissances possibles au sujet de subtilités en `Ltac2`, puisque l'objectif est que l'utilisateur·ice puisse ajouter sa propre transformation à l'ordonnanceur.

### 6.4.3 Faire une évaluation

Pour l'instant, la tactique `snipe`, qu'il s'agisse de la version avec ou sans ordonnanceur, n'a pas été évaluée sur de gros développements. Les raisons sont diverses :

— La tactique ne sélectionne pas de prémisses comme le fait `hammer`, elle échouera donc nécessairement sur tous les buts qui requièrent l'utilisation d'un lemme intermédiaire pour être résolus. Une solution peut être de remplacer la tactique de reconstruction de `hammer` par `snipe`, pour voir comment cette dernière tactique se comporte avec les lemmes qui ont été sélectionnés par `hammer`.

— Les types considérés doivent être membres de la classe de type `CompDec`. Pour évaluer `snipe`, il faudrait rajouter cette hypothèse à chaque fois que nécessaire sur les variables de types. Nous n'aurions donc pas une vraie évaluation de `snipe` sur les fichiers `Coq` choisis, car nous devrons faire des modifications sur ceux-ci.

— La première version de `snipe` est trop lente, mais il manque encore quelques fonctionnalités à l'ordonnanceur pour qu'il soit exploitable. Il faudrait d'abord que l'ordonnanceur puisse remplacer la première version de `snipe`.

Cependant, nous avons entamé un travail qui permet d'utiliser `snipe` comme reconstruction pour `hammer`, et nous sommes proches de pouvoir remplacer l'ancienne version de `snipe` par l'ordonnanceur. Une évaluation pourra donc bientôt être faite.

### 6.4.4 Diversifier les applications

L'ordonnanceur (tel quel, implémenté en `Ltac2`, ou la méthodologie de combiner des transformations comprenant *triggers*, filtres et états) pourrait servir à des préprocessings différents, pour des *backends* divers, comme nous l'avions déjà dit. En effet, la logique-cible n'est pas nécessairement celle des prouveurs SMT.

Plus généralement, nous voyons deux applications :

— Les transformations logiques, où l'on applique de petites transformations d'une logique source vers une logique cible.

— Les transformations de programme, où l'on applique de petites transformations pour passer d'un programme initial à un autre, écrits dans le même langage (par exemple, comme les passes d'optimisation de code en compilation).

#### En `Coq`

**D'autres tactiques d'automatisation** Il existe d'autres tactiques d'automatisation que la tactique `verit` de SMTCoq qui ont besoin de prétraitement. Nous pourrions combiner certaines des transformations actuelles de `Sniper` et d'éventuelles nouvelles transformation de manière à cibler les logiques que ces tactiques sont capables d'automatiser. Nous pensons par exemple à `firstorder` [13] qui fait de la recherche de preuve en logique du premier ordre. En particulier, la transformation qui génère le principe d'inversion des relations inductives n'est pas utile à SMTCoq mais l'est à `firstorder`. L'ordonnanceur pour `firstorder` intégrerait, par exemple, cette transformation, mais pas celles qui donnent les équations des fonctions. Une autre tactique qui pourrait être intéressante est `itauto` [4], un prouveur SAT en logique intuitionniste. Il faudrait rajouter dans notre bibliothèque une transformation qui instancie les quantificateurs, mais, une fois cette transformation rajoutée, nous pensons que l'ordonnanceur pourrait être utile à `itauto`.

**Une tactique de preuve graphique** Une tactique appelée `actema` [21] permet d'appeler, depuis `Coq`, une interface graphique qui permet de faire des preuves en logique intuitionniste du premier ordre, preuves qui seront ensuite recompilées vers des tactiques `Coq`. Or, `actema` ne gère pas le *pattern matching*, le polymorphisme ou l'ordre supérieur. Appliquer certaines transformations de `Sniper` pour préprocesser des énoncés pour `actema` fait partie de nos perspectives.

Pour le moment, nous sommes bloqué·es parce que `actema` fonctionne avec `coq-8.15` alors que l'ordonnanceur fonctionne avec `coq-8.17`, et celui-ci utilise de façon cruciale des fonctionnalités `Ltac2` qui n'existent pas dans les versions précédentes de `Coq`.

### En `OCaml`

**Un minimiseur de bug** La méthodologie de l'ordonnanceur pourrait sans doute être appliquée dès que l'on combine des petites transformations. C'est le cas du minimiseur de bugs écrit en `OCaml` et pour `OCaml` appelé `Chamelon` [45].

Des petites transformations de programmes successives (qui éliminent du code, qui remplacent certains types par des types triviaux, etc.) sont appliquées sur un code `OCaml` qui ne compile pas. À la suite de chaque application, on vérifie si l'erreur de compilation est toujours présente ou non. Si l'erreur est toujours présente, c'est que la minimisation était possible, et on continue à appliquer les transformations. Sinon, il faut essayer une autre transformation de minimisation.

Nous pensons qu'utiliser des *triggers* et des filtres pour les transformations de `Chamelon` pourrait permettre d'obtenir un minimiseur plus compositionnel, avec un code plus lisible, et peut-être même de gagner en efficacité.

**Les passes de compilation** En compilation, beaucoup de passes d'optimisation de code sont faites. À chaque passe, une transformation de programme spécifique est appliquée : élimination de code mort, simplifications de certains calculs, propagation de constantes, etc. Bénéficier d'un ordonnanceur de passes de compilation pourrait permettre d'expérimenter plus facilement plusieurs ordres pour les passes, et de rajouter facilement une nouvelle passe au besoin.

Dans LLVM, il y a un *legacy pass manager* qui réalise à peu près cette fonctionnalité d'ordonnancement. Son but est d'optimiser le temps et la mémoire utilisée par la compilation, en fonction des transformations sélectionnées par l'utilisateur·ice via la ligne de commande et la configuration par défaut (voir `https://llvm.org/docs/WritingAnLLVMPass.html#what-passmanager-does`), et de contraintes sur ces transformations.

Cependant, ce *legacy pass manager* ne fonctionne pas sur le mécanisme des *triggers*, qui présente l'avantage de séparer la partie d'analyse vérifiant si la passe de compilation s'applique de celle de l'exécution de la passe de compilation proprement dite.

Le *new pass manager*[1], en revanche, utilise ce mécanisme d'analyse permettant de vérifier si une passe de compilation s'applique ou non. Ces analyses sont enregistrées via un système de cache. Nous pensons que (1) nous inspirer de ce qui est fait en compilation pour, nous aussi, utiliser un système de *caching* de *triggers* pourrait être très intéressant pour optimiser notre ordonnanceur, (2) poursuivre nos recherches dans le domaine de la compilation beaucoup plus en détail pour mieux voir en quoi nos idées avec les *triggers* et les filtres se rapprochent ou diffèrent de ce mécanisme d'analyse. Bien sûr, ces perspectives dépassent largement le cadre de cette thèse.

1. Voir le post de blog `https://blog.llvm.org/posts/2021-03-26-the-new-pass-manager/`

# Bibliographie


[1] A. Aguirre. Towards a provably correct encoding from f* to smt. *Master's thesis, Université Paris*, 7, 2016.

[2] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[3] H. Barendregt, S. Abramsky, D. Gabbay, T. Maibaum, and H. H. Barendregt. Lambda calculi with types. 10 2000.

[4] F. Besson. Itauto : An Extensible Intuitionistic SAT Solver. In *ITP 2021 - 12th International Conference on Interactive Theorem Proving*, pages 1–18, Roma, Italy, June 2021.

[5] L. Blaauwbroek, J. Urban, and H. Geuvers. *The Tactician : A Seamless, Interactive Tactic Learner and Prover for Coq*, page 271–277. Springer International Publishing, 2020.

[6] J. Blanchette and L. Paulson. Hammering away : A user's guide to sledgehammer for isabelle/hol, 2018.

[7] V. Blot, A. Bousalem, Q. Garchery, and C. Keller. Smtcoq : automatisation expressive et extensible dans coq. In *JFLA 2019-Journées Francophones des Langages Applicatifs*, 2019.

[8] V. Blot, D. Cousineau, E. Crance, L. D. De Prisque, C. Keller, A. Mahboubi, and P. Vial. Compositional pre-processing for automated reasoning in dependent type theory. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 63–77, 2023.

[9] V. Blot, L. D. de Prisque, C. Keller, and P. Vial. General automation in coq through modular transformations. *arXiv preprint arXiv :2107.02353*, 2021.

[10] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In *International Symposium on Frontiers of Combining Systems*, pages 87–102. Springer, 2011.

[11] A. Chlipala. *Certified programming with dependent types : a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[12] A. Church. *The Calculi of Lambda-Conversion (AM-6), Volume 6*. Princeton University Press, Princeton, 1985.

[13] P. Corbineau. First-order reasoning in the calculus of inductive constructions. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, volume 3085 of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2003.

[14] D. Cousineau, E. Crance, and A. Mahboubi. Trakt : Uniformiser les types pour automatiser les preuves (démonstration). In *JFLA 2022-33èmes Journées Francophones des Langages Applicatifs*, pages 261–263, 2022.

[15] L. Czajka. A shallow embedding of pure type systems into first-order logic. In S. Ghilezan, H. Geuvers, and J. Ivetic, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016, May 23-26, 2016, Novi Sad, Serbia*, volume 97 of *LIPIcs*, pages 9 :1–9 :39. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[16] L. Czajka. Practical proof search for coq by type inhabitation. In *Automated Reasoning : 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II 10*, pages 28–57. Springer, 2020.

[17] L. Czajka and C. Kaliszyk. Hammer for coq : Automation for dependent type theory. *Journal of automated reasoning*, 61 :423–453, 2018.

[18] D. Delahaye. A tactic language for the system coq. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[19] D. Delahaye, C. Dubois, and J.-F. Étienne. Extracting purely functional contents from logical inductive types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 70–85. Springer, 2007.

[20] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. volume 3097, pages 198–212, 07 2004.

[21] P. Donato, P.-Y. Strub, and B. Werner. A drag-and-drop proof tactic. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 197–209, 2022.

[22] L. Dubois de Prisque. Transformations logiques pour SMTCoq. Master's thesis, Université Paris-Cité, Sept. 2020.

[23] G. Frege. *Grundlagen der Arithmetik : Studienausgabe MIT Dem Text der Centenarausgabe.* Breslau : Wilhelm Koebner Verlag, 1884.

[24] T. Gauthier, C. Kaliszyk, J. Urban, R. Kumar, and M. Norrish. Tactictoe : Learning to prove with tactics. *Journal of Automated Reasoning*, 65(2) :257–286, Aug. 2020.

[25] G. Gilbert, J. Cockx, M. Sozeau, and N. Tabareau. Definitional proof-irrelevance without k. *Proceedings of the ACM on Programming Languages*, 3(POPL) :1–28, 2019.

[26] W. A. Howard. The formulae-as-types notion of construction. In H. Curry, H. B., S. J. Roger, and P. Jonathan, editors, *To H. B. Curry : Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[27] A. J. Hurkens. A simplification of girard's paradox. In *Typed Lambda Calculi and Applications : Second International Conference on Typed Lambda Calculi and Applications, TLCA'95 Edinburgh, United Kingdom, April 10–12, 1995 Proceedings 2*, pages 266–278. Springer, 1995.

[28] J.-O. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer. Mtac2 : typed tactics for backward reasoning in coq. *Proceedings of the ACM on Programming Languages*, 2(ICFP) :1–31, 2018.

[29] S. Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq.* Theses, Université Paris Sud - Paris XI, Jan. 2011.

[30] T. Letan and Y. Régis-Gianas. Freespec : specifying, verifying, and executing impure computations in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 32–46, 2020.

[31] G. Malecha and J. Bengtson. Extensible and efficient automation through reflective tactics. In *Programming Languages and Systems : 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings 25*, pages 532–559. Springer, 2016.

[32] C. McBride. Dependently typed functional programs and their proofs. 2000.

[33] R. Milner. *Logic for computable functions description of a machine implementation.* Computer Science Department, Stanford University, 1972.

[34] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories : From an abstract davis–putnam–logemann–loveland procedure to dpll(. *J. ACM*, 53 :937–977, 01 2006.

[35] Z. Paraskevopoulou, A. Eline, and L. Lampropoulos. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 966–980, 2022.

[36] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[37] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In B. W. Paleo and D. Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, Jan. 2015.

[38] P.-M. Pédrot. Ltac2 : tactical warfare. In *The Fifth International Workshop on Coq for Programming Languages, CoqPL*, pages 13–19, 2019.

[39] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. The metacoq project. *J. Autom. Reason.*, 64(5) :947–999, 2020.

[40] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq coq correct ! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[41] M. Sozeau and C. Mangin. Equations reloaded : high-level dependently-typed functional programming and proving in coq. *Proc. ACM Program. Lang.*, 3(ICFP) :86 :1–86 :29, 2019.

[42] E. Tassi. Elpi : an extension language for coq (metaprogramming coq in the elpi λprolog dialect). 2018.

[43] P. Tollitte, D. Delahaye, and C. Dubois. Producing certified functional code from inductive specifications. In C. Hawblitzel and D. Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2012.

[44] P.-N. Tollitte. *Extraction de code fonctionnel certifié à partir de spécifications inductives.* PhD thesis, Paris, CNAM, 2013.

[45] M. Valnet, N. Courant, G. Bury, P. Chambart, and V. Laviron. Chamelon : un minimiseur pour et en ocaml. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, 2024.

[46] J. Wells. Typability and type checking in the second-order /spl lambda/-calculus are equivalent and undecidable. In *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 176–185, 1994.

[47] A. N. Whitehead and B. A. W. Russell. *Principia mathematica ; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.