Transformations logiques pour SMTCoq

Louise Dubois de Prisque

Transformations logiques pour SMTCoq

Rapport de stage

Soutenu le 7 Septembre 2020

Louise Dubois de Prisque

Master de Logique, Mathématiques et Fondements de l'Informatique

Encadré par Chantal Keller (Maîtresse de conférences, LRI) et

Valentin Blot (Chargé de recherche, INRIA)

Stage au LSV, Deducteam

Université Paris-Saclay

Mai-Août 2020

Résumé

Cet article s'inscrit dans le cadre de l'interaction entre deux outils permettant de prouver mécaniquement des théorèmes mathématiques : les assistants de preuve d'une part, et les prouveurs automatiques de l'autre. En particulier, j'ai contribué au développement de l'interface appelée SMTCoq dont le but est d'augmenter l'automatisation des assistants de preuve en utilisant la puissance des prouveurs automatiques. L'objectif était de fournir un cadre pour définir et certifier des transformations logiques qui permettent de passer d'une logique complexe à une logique plus simple, tout en s'assurant que le résultat de la transformation implique le terme transformé. À long terme, nous voudrions passer de la logique de l'assistant de preuve Coq à celle des prouveurs automatiques, pour augmenter le nombre de théorèmes qu'une interface prouveur/assistant de preuve permettra de démontrer. Nous verrons que ce cadre a été fourni, et qu'il existe deux manières de prouver la correction des transformations en question. Par ailleurs, j'ai travaillé sur un exemple de transformation dont la vérification est simple : la monomorphisation, dont j'ai prouvé la correction. Le prochain objectif sera d'intégrer cette transformation au code de SMTCoq.

Cet article contribue au développement de l'interface SMTCoq, qui utilise deux outils permettant la mécanisation des preuves mathématiques : les assistants de preuve et les prouveurs automatiques. SMTCoq vise à augmenter l'automatisation de l'assistant de preuve Coq en utilisant la puissance de prouveurs automatiques SMT et SAT, et certifie ces derniers. Cependant, la logique de Coq est bien plus complexe que celle des prouveurs, et il faut que l'interface puisse passer de l'une à l'autre. Pour cette raison, nous avons fourni un cadre pour définir et certifier des transformations logiques permettant de passer d'une logique complexe à une logique plus simple. Ces transformations doivent être correctes, c'est-à-dire que le résultat de la transformation doit impliquer le terme transformé. Par ailleurs, nous avons implanté et certifié une transformation particulière : une monomorphisation en théorie des types simples. Notre prochain objectif sera de l'intégrer au code de SMTCoq, pour pouvoir écrire des tactiques qui enverront des buts polymorphes aux prouveurs.

Table des Matières

1 Introduction et outils utilisés

1.1 Pourquoi vérifier des preuves par ordinateur ?

Avant qu'un article de recherche en mathématiques ne soit publié, il est soumis à relecture humaine. Cependant, il arrive que ce travail de vérification ne suffise pas et que des erreurs se glissent dans l'article, au point que le résultat prétendument démontré se révèle faux (11). Celles-ci sont parfois liées à la fastidiosité de la preuve : de nombreux sous-cas à gérer, une syntaxe verbeuse, des lemmes imbriqués... Ainsi, il pourrait être avantageux de déléguer le travail de vérification ou de preuve à une machine. Mais pourquoi la croire moins faillible qu'un être humain, qui, de plus, est spécialisé dans le domaine ?

À cela, il existe une raison : la machine ne fera pas d'erreur de calcul ni de raisonnement. Par exemple, elle ne pourra pas oublier de sous-cas dans de longues disjonctions, elle ne substituera pas un signe pour un autre, etc. Mais bien sûr, ce n'est le cas que si elle a été correctement programmée. Il s'avère donc utile de prouver la correction de notre programme de vérification. A nouveau, nous pouvons être tentés de le faire mécaniquement. Nous nous engageons alors dans une régression à l'infini, sauf si nous choisissons de "croire" certaines lignes de code spécifiques, sans vérification supérieure, qui constitueront le noyau logique auquel nous faisons confiance. C'est sur cette idée que se fondent les assistants de preuve. Au lieu de vérifier le code de notre prouveur, nous pouvons aussi croire qu'il ne présentera pas de bugs en le testant sur une grande quantité d'exemples, ce qui est le cas des prouveurs automatiques.

1.2 Assistants de preuve

Comme mentionné ci-dessus, il existe deux grandes familles de programmes permettant d'automatiser les preuves mathématiques : *les assistants de preuve et les prouveurs automatiques*. Présentons la première : les *assistants de preuve*.

Il s'agit de logiciels permettant de définir et de vérifier des preuves de théorèmes formels. Par exemple, nous avons Coq, Isabelle/HOL, Agda ou Lean. Pour vérifier ces preuves, le logiciel examine si l'arbre de preuve obtenu est bien celui d'une dérivation valide. La validité en question correspond aux règles logiques utilisées par l'assistant de preuve. Elles constituent sa *base de confiance*. Ce sont des lignes de code, parfaitement identifiées, qui ne sont pas vérifiées, auxquelles nous faisons confiance en usant du logiciel. Dans certains assistants de preuve comme HOL-light, ce *kernel* est aussi petit que possible et correspond à un ensemble de règles logiques simples, mais ce n'est pas toujours le cas. L'important, néanmoins, est que tout ce qui ne se trouve pas dans le kernel en dérive, afin que l'utilisateur sache exactement ce en quoi il croit. Par exemple, des tactiques ont été rajoutées à certains assistants de preuve (Coq ou Lean) pour en faciliter l'usage. Il s'agit de transformations opérées sur le terme de preuve, qui peuvent être très simples ou complexes. Mais ces tactiques ne font pas partie du kernel, elles se fondent dessus.

Dans tous les cas, malgré leur fiabilité, les assistants de preuve demeurent peu utilisés et sont souvent réservés aux spécialistes du domaine. En effet, ils demandent beaucoup d'interaction de la part de l'utilisateur, et un niveau de détail très rare dans les preuves papier. Un résultat qui semble évident peut demander l'utilisation de nombreuses tactiques ou une recherche fastidieuse dans les

bibliothèques des lemmes déjà prouvés. Par exemple, pour une preuve arithmétique, l'assistant n'appliquera pas de lui-même la commutativité de l'addition. Il faudra retrouver ce lemme dans une bibliothèque ou le reprouver soi-même.

1.3 Prouveurs automatiques

Les *prouveurs automatiques* sont des logiciels souvent codés dans des langages de programmation offrant des fonctionalités bas niveau, dont le but est de résoudre une classe de problèmes mathématiques le plus rapidement possible. Par exemple, zChaff, avatarSAT, CVC4 ou veriT sont des prouveurs automatiques. Ils font de la *recherche de preuve*, c'est-à-dire qu'ils vont chercher à produire une preuve démontrant ou infirmant l'énoncé en entrée. Ils demandent peu de la part de l'utilisateur : si le problème d'entrée leur est envoyé sous le bon format, ils effectueront leurs calculs automatiquement. Ce qu'on attend d'un prouveur est aussi qu'il ne se trompe jamais. S'il répond qu'un théorème est vrai, il faudrait que le théorème le soit, et si au contraire il exhibe un contre-exemple au prétendu théorème, il faudrait que ce contre-exemple en soit bien un du problème en question.

Cependant, les prouveurs automatiques ne s'appuient pas sur un kernel logique parfaitement identifié comme le font les assistants de preuve. Cela signifie qu'ils peuvent avoir des bugs, et qu'il s'avère difficile d'identifier leur base de confiance. C'est bien souvent leur code entier qu'il faut croire.

1.4 Coq

L'assistant de preuve utilisé dans ce stage est Coq. Sa logique sous-jacente s'appelle le calcul des constructions inductives (CIC) (4) (5). Son noyau, implémenté en OCaml, se fonde sur celle-ci. Remarquons ici quelques particularités de cet outil auxquelles j'ai dû m'adapter.

Il se fonde sur la théorie des types. Dans celle-ci, chaque *terme* possède un *type* (le typage est noté " :"), et tous se combinent selon des règles logiques. Les types sont vus comme des propositions et les termes sont vus comme des preuves du type correspondant. Par exemple, si j'ai une fonction $f : A \to B$ et un terme $x : A$, je peux appliquer f à x et j'obtiens $f\ x : B$. Les termes bien typés sont ceux qui ont été construits d'après les règles de typage.

Coq est un *vérificateur de types*. Étant donné un terme de preuve, il vérifie donc que le terme est bien typé, donc qu'il est une preuve de ce type.

Coq comporte des types inductifs : ceux-ci comportent des *constructeurs*, c'est-à-dire une liste de règles qui expliquent comment former des termes de ce type. Par exemple, le type Nat est tel que 0 est de ce type, et, pour tout élément de type Nat, son successeur l'est aussi. Cela s'écrit en Coq :

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

Comme Coq ne permet d'écrire que des termes bien typés, il faut parfois inclure les cas d'erreur dans les définitions. J'ai donc eu besoin, dans mes preuves, de traiter tous les cas d'erreur qui sont triviaux dans des démonstrations sur papier. Le type inductif `option` permet, par exemple, de gérer le cas où la fonc-

tion renvoie un résultat (constructeur `Some`), et celui où elle échoue (constructeur `None`).

De plus, Coq repose sur une logique intuitionniste. Cela signifie que $(a = b) \vee (a \neq b)$ n'est pas donné pour tous les types. J'ai donc eu besoin de prouver ce lemme pour les types algébriques sur lesquels j'ai travaillé pour utiliser les modules d'ensembles finis. Ils implémentent en Coq les ensembles finis où l'on peut toujours démontrer soit que $a = b$ soit que $a \neq b$. Par ailleurs, l'outil SMTCoq sur lequel j'ai travaillé utilise des formules booléennes. Or, les quantificateurs dans les booléens ne sont pas définissables en Coq en toute généralité à cause de sa logique intuitionniste, ce qui demande un traitement particulier pour ceux-ci.

1.5 Interaction prouveurs/assistants

Nous pouvons remarquer que le défaut des assistants de preuves (une plus faible automatisation) correspond précisément au point fort des prouveurs automatiques (une forte automatisation). De même, l'avantage des assistants de preuve (leur base de confiance précisément identifiée) est ce qui manque aux prouveurs automatiques (une base de confiance importante, pas clairement identifiée). De ce constat, l'idée de développer une interface assistant de preuve/-prouveurs automatiques a émergé. Autrement dit, nous pouvons certifier le code des prouveurs automatiques en utilisant un assistant de preuve. Une fois certifié, la base de confiance du prouveur n'est alors plus l'entiereté de son code, mais précisément le kernel logique de l'assistant employé.

Par ailleurs, les tactiques écrites en Coq sont d'abord des tactiques simples (réécriture, application ou introduction d'hypothèses...). Mais il est possible d'écrire des tactiques qui font appel à toute la puissance d'un prouveur automatique, et qui, par là même, ne demanderont à l'utilisateur que de faire appel à la tactique pour prouver le résultat.

L'interface SMTCoq, sur laquelle j'ai travaillé durant mon stage, permet ainsi d'améliorer l'automatisation de l'assistant de preuve Coq en appelant des prouveurs externes. Elle certifie également ces prouveurs. Nous en présenterons le fonctionnement dans la deuxième partie.

1.6 Prouveurs SMT

Les prouveurs automatiques utilisés dans SMTCoq sont principalement des prouveurs SMT (*Satisfiability Modulo Theories*). Il convient donc d'expliquer leur fonctionnement. Ils supportent toujours une ou plusieurs théories, c'est-à-dire un ensemble d'axiomes.

Les solveurs SMT reçoivent une instance d'un problème SMT : une formule du premier ordre avec égalité et sans quantificateur, pouvant comporter des variables libres. Ils fournissent en sortie soit une assignation des variables libres si la formule est satisfaisable, soit un certificat, c'est-à-dire une preuve qu'elle ne l'est pas. Donnons les étapes de leur fonctionnement : tout d'abord, le solveur met la formule en forme normale conjonctive (CNF). Une formule en CNF est une conjonction de clauses, où une clause est une disjonction de formules atomiques niées ou non. Par exemple, $\neg((x = y) \wedge (y = z)) \wedge (x \leqslant z)$ n'est pas en CNF mais $(\neg(x = y) \vee \neg(y = z)) \wedge (x \leqslant z)$ qui est équivalente l'est.

Ensuite, le solveur ignore les variables libres pour transformer la formule en une formule de la logique propositionnelle. Le problème devient un problème SAT. Le solveur utilise un algorithme de backtracking pour fournir une instance qui la satisfait. S'il n'y en a pas, la formule n'est pas satisfaisable. S'il y en a un, le solveur SMT compare le modèle fourni avec les règles de la théorie. Si celui-ci ne convient pas, le solveur ajoute alors de nouvelles clauses à la formule et appelle de nouveau le prouveur SAT qui lui fournit un autre modèle. L'algorithme s'arrête lorsque la formule n'est pas satisfaisable ou qu'il trouve un modèle cohérent avec la théorie.

Le prouveur SAT que j'ai mentionné prend en entrée des formules de la logique propositionnelle. En sortie, il donne une assignation des variables propositionnelles (une valeur booléenne) qui rend la formule valide, ou renvoie un certificat s'il n'y en a pas. Un certificat est une suite de règles qui permettent de reconstruire la preuve de la non-satisfaisabilité de la formule.

Le principal à retenir de cette procédure, c'est qu'un solveur SAT ou SMT a besoin d'un problème d'entrée sous une forme bien particulière. De plus, dans le cas du solveur SMT, il ne comprend que la logique du premier ordre sans quantificateur (FOL), et la logique propositionnelle pour le prouveur SAT.

Exemple 1.1 *Voici comment un prouveur SMT prouve la satisfaisabilité de la formule* $(x > 0 \vee x \leqslant 0) \wedge (x + 1 = y + 5)$. *Elle est déjà en forme normale conjonctive, il la transforme en formule propositionnelle :* $(p \vee q) \wedge r$. *Ensuite, le solveur SAT fournit une instance, par exemple :* $p =$ `true`, $q =$ `true`, $r =$ `true` *Mais ces assignations sont incompatibles avec la théorie des réels car on ne peut avoir* $x > 0 \wedge x \leqslant 0$. *Le solveur ajoute alors la nouvelle clause :* $\neg p \vee \neg q$. *Ensuite, le solveur SAT envoie une nouvelle instance :* $p =$ `true`, $q =$ `false`, $r =$ `true` *et celle-ci convient.*

1.7 Contributions

J'ai travaillé sur l'interaction prouveurs/assistants, en essayant de transformer des formules d'une logique plus complexe à une logique plus simple. À terme, il faudra encoder certaines formules de CIC pouvant contenir des inductifs ainsi que, par exemple, la relation de typage elle-même dans FOL pour que l'on puisse passer d'un but dans un assistant de preuve à une formule prouvable par un solveur SMT. Voici la liste de mes contributions :

- Encodage dans Coq deux langages typés et écriture d'une transformation permettant de passer du plus complexe au plus simple.
- Définition des règles de validité de ces deux langages.
- Preuve que cette transformation est correcte au sens que la validité du terme dans un contexte traduit implique celle du terme dans le contexte de départ.
- Écriture des fonctions d'interprétation qui transforment les termes et les types du langage en termes et types de Coq. Ces fonctions transforment un terme défini par un inductif (on appelle cela un *encodage profond* car on a accès à la structure du terme) en un terme de Coq (un *encodage superficiel*).

Le code de mes contributions se trouve à l'adresse suivante :
`https://gitlab.inria.fr/lduboisd/smt_coq_stage`

2 SMTCoq : une interface prouveur/assistant

SMTCoq est l'une de ces interfaces prouveurs/assistants dont nous avons parlé. Nous expliquerons d'abord les avantages de cette interface (d'une part vérification des prouveurs et d'autre part plus grande automatisation des assistants de preuve), puis nous détaillerons son fonctionnement, ainsi que la contribution apportée par ce stage.

2.1 Deux approches pour vérifier un prouveur

Il existe deux manières de certifier un prouveur automatique :

— L'approche *autarcique* : on certifie l'ensemble du code du prouveur au sein d'un assistant de preuve. Cette manière de procéder est sans doute la plus fiable, mais elle est fragile et très coûteuse. Il suffit en effet de modifier quelques aspects du code, et la preuve de correction (la preuve que le code programme correspond bien aux spécifications, et donc, dans le cas des prouveurs automatiques, qu'il ne fournit pas de certificats ou d'instances erronées) sera à refaire.

— L'approche *sceptique* : c'est celle choisie par l'outil SMTCoq. Nous avons dit qu'étant donné une formule non-satisfaisable, les prouveurs renvoyaient des certificats : il suffit donc de vérifier à chaque appel du prouveur que le certificat qu'il renvoie est bien une preuve d'insatisfaisabilité de la formule. Cette approche garantit la correction (car SMTCoq verifie que le certificat fourni par le prouveur implique bien que la formule n'est pas satisfaisable), mais pas la complétude. En effet, il peut y avoir des formules non satisfaisables où le certificat renvoyé est erroné, ou alors, quand SMTCoq n'arrive pas à reconstruire la preuve à partir de celui-ci. SMTCoq vérifie donc les réponses du prouveur, ce qui le rend moins dépendant du code de celui-ci, et qui lui permet ainsi d'interagir avec plusieurs prouveurs.

2.2 Des nouvelles tactiques pour Coq

SMTCoq définit de nouvelles tactiques pour Coq. Elles appellent un prouveur automatique différent : zChaff (solveur SAT), CVC4 ou veriT (solveurs SMT). Ainsi, le but à prouver le sera automatiquement, et demanderont à l'utilisateur de Coq beaucoup moins d'efforts. Nous avons vu que les prouveurs automatiques envoyaient une preuve de la non-satisfaisabilité d'une formule dans le cas où aucune instance de ses variables ne la rend vraie. Pour prouver un théorème, SMTCoq envoie donc la négation du but au prouveur. Le prouveur renvoie un certificat démontrant que la négation du but est fausse. On obtient donc une preuve de la double négation de notre théorème. Mais Coq utilise deux manières de représenter les formules logiques : les formules dans `bool` et celles dans `Prop`. La première est classique, la seconde intuitionniste. Pour s'accorder avec les prouveurs, qui usent de la première logique, SMTCoq travaille sur des formules de `bool` et utilise un traitement particulier pour les quantificateurs. Comme on ne peut pas les définir dans `bool`, on utilise un lemme quantifié dans `Prop`, que l'on instancie et que l'on transforme alors en formule de `bool`. On utilise un encodage profond pour les termes de SMTCoq, sauf dans le cas des quantificateurs où un encodage superficiel est privilégié.

2.3 Schéma du fonctionnement de SMTCoq

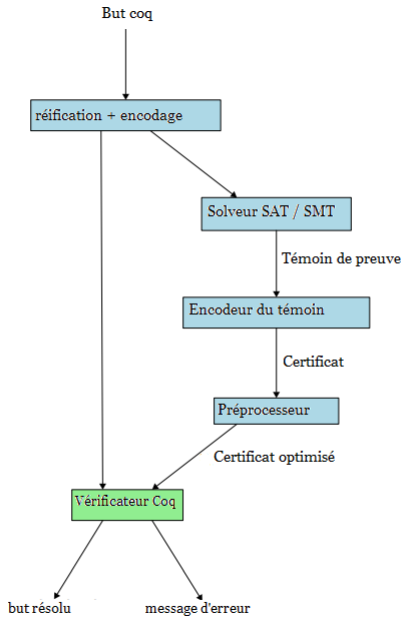

FIGURE 1 – Automatisation avec SMTCoq

Examinons plus précisément le fonctionnement de notre interface.

Tout d'abord, SMTCoq *réifie* le but Coq à démontrer. En effet, Coq n'a pas accès à la syntaxe de ses propres termes. Autrement dit, il ne fera pas la différence entre `true`$\vee$`true` ou `true`, par exemple, car ces termes sont convertibles l'un à l'autre (c'est-à-dire qu'ils se réduisent au même terme, `true`, après calcul). Réifier le but Coq lui permettra de faire cette différence, car la réification est l'opération qui donne accès à la structure du terme. Ensuite, c'est là qu'intervient *l'encodage*. Un encodage est une fonction qui à des termes d'un langage formel associe des termes d'un autre langage. Les prouveurs ne comprennent que la logique du premier ordre, il faut donc encoder une partie de la logique de Coq dans celle-ci. C'est l'aspect sur lequel j'ai travaillé pendant mon stage, détaillé au pargraphe suivant. Puis, le solveur renvoie un témoin de preuve que la formule n'est pas satisfaisable, il faut alors l'encoder dans la logique de Coq. Ensuite, le *préprocesseur* optimise le certificat en diminuant sa place en mémoire, et permet d'obtenir un format commun aux différents solveurs utilisés. Enfin, le *vérificateur Coq* s'assure que le certificat obtenu implique bien la négation de la négation du but de départ. Si c'est le cas, la formule est prouvée. Autrement, Coq renverra un message d'erreur : soit la formule que l'on souhaite prouver n'est pas valide, soit le prouveur automatique a un bug dans son code, soit on a appelé à tort le prouveur (pour une formule que l'on n'a pas pu encoder dans la logique du prouveur). Cependant, Coq affichera seulement qu'il ne peut pas prouver le but et nous ne saurons pas dans quel cas d'erreur nous nous trouvons.

2.4 De la logique de Coq vers celle des prouveurs

Actuellement, SMTCoq permet de résoudre des buts de théories que supportent les prouveurs SMT, et des buts propositionnels. Nous aimerions pouvoir augmenter la quantité de théorèmes prouvables. Pour cela, il faut encoder certains aspects de la logique de Coq dans la logique du premier ordre, afin de passer de buts exprimés dans CIC à des buts de FOL. Il s'agit alors d'appliquer des *transformations logiques*. Deux méthodes sont possibles. Soit, à l'instar de CoqHammer (12), on applique une grosse transformation logique qui encode de nombreux aspects de CIC (le typage, les produits dépendants, le polymorphisme, les types inductifs...). Le souci sera alors de prouver la correction de cette transformation. Des tentatives ont montré la complexité et donc la limite de l'approche. Soit on choisit de composer plusieurs petites transformations logiques successives qui encodent certains aspects de CIC dans une logique plus simple. L'avantage est alors que la preuve de correction de chacune de ces transformations s'avère beaucoup plus simple (mais nous verrons qu'elle demeure difficile). De plus, nous pouvons travailler indépendamment sur chaque transformation. C'est l'option choisie pour développer SMTCoq. Certaines transformations se composeront avec d'autres, elles ne seront pas nécessairement appliquées sur un but donné comme le montre le schéma ci-dessous.

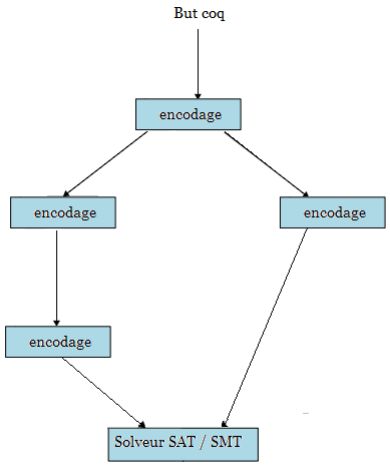

FIGURE 2 – Zoom sur l'aspect encodage de SMTCoq

J'ai travaillé sur le cadre permettant de définir et de certifier ces transformations. J'ai également validé celui-ci par l'implantation d'une transformation simple qui sera l'objet du chapitre suivant.

3 Transformations logiques

3.1 Définitions

Une transformation logique (ou traduction) est une fonction qui permet de passer d'un langage source $\mathscr{S}$ à un langage d'arrivée $\mathscr{T}$. Si le langage est typé, on doit avoir une fonction de traduction [.] pour les termes et une autre $[\![.]\!]$ pour les types. La fonction traduisant les termes peut dépendre de celle concernant les types. Le contexte est traduit point à point, c'est à dire que si Γ est une liste de types, $trad(\Gamma :: A) = trad(\Gamma) :: [\![A]\!]$, où *trad* est la fonction traduisant les listes de types. Ici, nous nous intéressons aux traductions *syntaxiques*, c'est-à-dire qu'elles sont définies inductivement sur la syntaxe des termes de notre langage.

Les "bonnes" propriétés pour une traduction donnée dépendent de l'objectif que l'on s'est assigné. Généralement, nous pouvons demander :

- La *correction* : Si pour Γ un contexte, t un terme et A un type, on a $\Gamma \vdash t : A$ alors on a aussi $trad(\Gamma) \vdash [t] : [\![A]\!]$. On dit aussi qu'elle *préserve le typage.*
- La *complétude* : Si pour Γ un contexte, t un terme et A un type, on a $trad(\Gamma) \vdash [t] : [\![A]\!]$ alors on a aussi $\Gamma \vdash t : A$
- La *préservation de la conversion* : Si deux termes ou types sont convertibles, leurs traductions doivent l'être aussi.
- La *préservation de la consistance* : Si $[\![\bot_{\mathscr{S}}]\!]$ est habité (c'est-à-dire qu'il existe un terme avec ce type), alors $\bot_{\mathscr{T}}$ aussi. Un langage avec ce type habité signifie que celui-ci est inconsistant, car avoir une preuve de $\bot$ revient à avoir une preuve de toute proposition.

Dans les transformations qui vont nous intéresser, les terminologies usuelles de "correction" et de "complétude" que nous venons de donner sont inversées. En effet, pour les traductions usuelles, ce que signifie la correction, c'est que la traduction préserve la prouvabilité. Pour les traductions dont SMTCoq a besoin, pas besoin de préserver la prouvabilité. Si on ne la préserve pas, et cela peut arriver puisque l'on passe d'une logique expressive à une logique moins expressive, il sera simplement impossible de prouver le but Coq initial en usant de cette transformation, mais rien de faux n'aura été démontré. La notion de "correction" requise pour SMTCoq s'approche donc plus de la notion usuelle de "complétude".

L'important est que la traduction implique bien le but de départ : nous voulons que le résultat des transformations sucessives, envoyé au prouveur automatique, soit bien une preuve du but initial. Ainsi, le prouveur démontrera quelque chose de plus fort que le théorème qui intéresse l'utilisateur. Il n'y arrivera donc pas toujours, mais quand il y parviendra, nous serons certains de la validité du théorème qui nous occupait. Une traduction correcte pour SMTCoq est donc une traduction qui permet de s'assurer qu'une preuve du but transformé en est une du but de départ. *Pour cette raison, quand nous traiterons des transformations de SMTCoq, nous userons du terme "correction" pour parler de ce qui s'appelle classiquement "complétude".*

3.2 Exemples de transformations logiques

Donnons désormais des exemples de transformations logiques.

— L'une des plus connues est la transformation de *Kuroda*, qui permet de traduire un séquent de LK (Logique classique) dans LJ (Logique intuitionniste). Elle est à la fois complète et correcte (10).
— *L'interprétation* est une transformation particulière, qui envoie des termes d'un langage donné dans des termes de la logique de Coq. Nous la verrons plus en détail dans la partie suivante.
— *La monomorphisation* est une transformation qui prend un langage avec des types polymorphes, c'est-à-dire des types de la forme $\forall X(...)$, où X est une variable de type, et ôte les quantificateurs sur les variables de type (par exemple, en instanciant chaque variable de type par un ou plusieurs types donnés).
— *La skolémisation* permet de passer de la logique du premier ordre à une logique du premier ordre sans quantificateurs existentiels. Elle traduit des formules sous formes prénexe en remplaçant chaque occurence du quantificateur existentiel par un nouveau symbole de fonction appliqué à toutes les variables précédant ce quantificateur. Intuitivement, cette fonction renvoie un témoin pour chacune de ses entrées. Par exemple, la formule $\forall x \forall y \exists z\, T(x, y, z)$ est skolemisée en $\forall x \forall y\, T(x, y, f(x, y))$
— Une transformation permet aussi de passer d'un langage $\mathscr{S}$ avec des applications partielles à un langage $\mathscr{T}$ qui n'en contient pas. Par exemple, si f est une fonction à deux arguments, $f\ x$ existe dans $\mathscr{S}$, mais pas dans $\mathscr{T}$. A la place, on utilise un prédicat binaire spécial, @, qui prend en premier paramètre la fonction et en second son argument. Ainsi, dans $\mathscr{T}$, on introduit un nom pour $f\ x$, par exemple f_x. Ensuite, on traduit $f\ x\ y$ par $@(f_x, y)$ et on ajoute un axiome dans $\mathscr{T}$: $\forall y, @(f_x, y) = f\ x\ y$

3.3 La transformation étudiée : une monomorphisation

Lors de ce stage, je n'ai pas traduit de formules de la logique de Coq vers la logique des prouveurs. J'ai choisi une logique de départ plus simple et une logique d'arrivée plus riche, afin de réduire la complexité de la transformation pour m'intéresser avant tout au cadre permettant de la certifier. Le langage $\mathscr{S}$ duquel je suis partie est celui de la théorie des types simples (3) avec polymorphisme prénexe. Le langage $\mathscr{T}$ est un fragment de $\mathscr{S}$ sans variable de type.

Voici la grammaire de $\mathscr{S}$ pour les types : $A, B := X \mid A \to B \mid \text{Nat} \mid \text{Bool}$. On suppose que X est une variable de type implicitement quantifiée en tête de formule.

Exemple 3.1 *Considérons un exemple parmi les types de $\mathscr{S}$, $X \to Y$. La quantification prénexe implicite signifie que ce type est en fait : $\forall X\ \forall Y,\ X \to Y$. Comme la quantification est toujours universelle et prénexe, on peut permuter les quantificateurs, ce type est tout aussi bien défini par : $\forall Y\ \forall X,\ X \to Y$*

Les termes de $\mathscr{S}$ sont : $t, u := x \mid \lambda x : A.u \mid \text{true} \mid \text{false} \mid u \wedge v \mid \neg u \mid u\ v$. Comme nous sommes dans la théorie des types simples, les *formules* de $\mathscr{S}$ sont les termes de type Bool. Tout terme bien construit est typé, avec les règles de typage du lambda-calcul simplement typé usuel.

La grammaire des types de $\mathscr{T}$ est $A, B := A \to B \mid \text{Nat} \mid \text{Bool}$. On n'a donc plus de variables de types (quantifiées). Les termes sont identiques à ceux de $\mathscr{S}$. Les traductions $[\![.]\!]$ et $[.]$ sont définies ainsi :

$[\![\text{Nat}]\!] := \text{Nat}$
$[\![X]\!] := \text{Nat}$
$[\![\text{Bool}]\!] := \text{Bool}$
$[\![A \to B]\!] := [\![A]\!] \to [\![B]\!]$
$[x] := x$
$[\lambda x : A.u] := \lambda x : [\![A]\!].\,[u]$
$[\neg u] := \neg\,[u]$
$[u\ v] := [u]\ [v]$
$[u \wedge v] := [u] \wedge [v]$

En Coq, j'ai choisi de séparer les termes de $\mathscr{S}$ et de $\mathscr{T}$ en les définissant par deux inductifs différents. J'ai donc distingué les `PTerm` et les `PType` (pour le langage polymorphe) des `MTerm` et des `MType`(pour le langage monomorphe). Voici le code de ces deux types algébriques :

```
Inductive PType : Type :=
| PBool : PType
| PNat : PType
| PVar : nat → PType (* les variables sont
    representees par des entiers *)
| Pa : PType → PType → PType.

Inductive PTerm : Type :=
| PTermVar : nat → PTerm
| PTrue : PTerm
| PFalse : PTerm
| PNeg : PTerm → PTerm
| PAnd : PTerm → PTerm → PTerm
| PLam : PType → PTerm → PTerm
| PApp : PTerm → PTerm → PTerm.

Inductive MType : Type :=
| MBool : MType
| MNat : MType
| Ma : MType → MType → MType.

Inductive MTerm : Type :=
| MTermVar : nat → MTerm
| MTrue : MTerm
| MFalse : MTerm
| MNeg : MTerm → MTerm
| MAnd : MTerm → MTerm → MTerm
| MLam : MType → MTerm → MTerm
| MApp : MTerm → MTerm → MTerm.
```

Nous allons désormais voir comment prouver en Coq que cette traduction est correcte. Pour cela, nous aurons besoin d'exprimer la notion de correction en Coq, comme l'explique la prochaine partie.

4 Preuves de correction : deux méthodes

4.1 Présentation des deux méthodes et comparaison

Pour prouver la correction de notre traduction, deux méthodes sont possibles, nous en donnons d'abord le principe :

— Nous pouvons utiliser des *prédicats inductifs* qui donnent les règles de validité des termes de $\mathscr{S}$ et de $\mathscr{T}$. Il faudra définir ces prédicats dans Coq, et montrer alors que la validité d'un terme dans le langage d'arrivée sous un certain contexte traduit implique que ce terme était déjà valide dans le langage de départ, dans ce même contexte non traduit. L'avantage de cette méthode est que définir les prédicats inductifs s'avère plutôt simple. Cependant, introduire de nouvelles règles sémantiques dans Coq, plutôt qu'utiliser la sémantique de Coq elle-même (comme le fait la seconde méthode), introduit un "détour" dans notre preuve. Il faut revenir au schéma du fonctionnement de SMTCoq pour le comprendre : notre but final est de prouver un but Coq, pas de prouver quelque chose dans un nouveau langage. Nous devrons donc montrer, en plus, qu'un terme valide selon le prédicat inductif que nous avons défini implique que sa traduction en Coq est vraie.

— La deuxième méthode consiste à *interpréter* les types et les termes de $\mathscr{S}$ et de $\mathscr{T}$ par, respectivement, des types de Coq et des termes du type Coq en question. Pour cela, nous avons besoin d'une *fonction d'interprétation* qui est une fonction permettant de passer d'un langage à des objets de Coq. La preuve de correction consistera alors à montrer que si l'interprétation de $[t]$ est une preuve a d'un type A, alors l'interprétation de t est la preuve b d'un type B, tel que $[\![B]\!] = A$. Cette méthode a l'avantage d'être directe, contrairement à la première, mais écrire la fonction d'interprétation s'avère complexe en Coq.

4.2 Méthode prédicat inductif

Les règles de validité choisies sont reprises de l'article de Harrison (8) et sont celles de la théorie des types simples. Rappelons que les seuls contextes admissibles de cette théorie sont les termes de type Bool. En Coq, nous avons donc d'abord besoin d'un prédicat inductif qui vérifie que le contexte est valide. Il prend un ensemble fini en paramètre et vérifie que chaque terme de cet ensemble est bien de type Bool. Les fonctions `infer_typeP` et `infer_typeM` calculent le type d'un terme de $\mathscr{S}$ et de $\mathscr{T}$ respectivement. Voici le code de `infer_typeP` :

```
Fixpoint infer_typeP (c: context_typeP) (t: PTerm):
   option PType :=
match t with
| PTermVar i ⇒ nth_error c i
| PTrue ⇒ Some PBool
| PFalse ⇒ Some PBool
| PNeg x ⇒ match infer_typeP c x with
              | Some PBool ⇒ Some PBool
              | _ ⇒ None
              end
```

```
| PAnd x y ⇒ match infer_typeP c x with
              | Some PBool ⇒  match infer_typeP c y
                 with
                          | Some PBool ⇒ Some PBool
                          | _ ⇒ None
                           end
              | _ ⇒ None
              end
| PLam A v ⇒ match infer_typeP (A::c) v with
          |Some B ⇒ Some (Pa A B)
          | _ ⇒ None
      end
| PApp u v ⇒ match infer_typeP c u, infer_typeP c v
   with
| Some (Pa A B), Some C ⇒
if syn_eq_typeP A C then Some B else None
| _, _ ⇒ None
end
end.
```

La fonction `nth_error` prend deux arguments : une liste et un entier n. Elle renvoie un type option : `Some` avec le n-ième élément de la liste en argument s'il existe, et `None` s'il n'existe pas de tel élément.

Notons que nous avons besoin d'un contexte pour les variables de termes (une liste de types). Nous verrons en détail l'implémentation des variables par la suite. Par ailleurs, pour que le terme soit bien typé, nous avons besoin d'une fonction `syn_eq_typeP` qui vérifie que le premier argument d'une application a bien un type flèche A et que son deuxième a bien le type A. Désormais que les deux fonctions qui, d'un terme de nos langages, infère son type, sont codées, nous pouvons définir le prédicat `context_okP` comme suit :

```
Definition context_okP (X : contextP) :=
(forall (x : PTerm), Se.In x X → (infer_typeP [] x
   = Some PBool)).
```

Le prédicat `Se.In` est défini pour les modules d'ensembles finis (ici les ensembles de `PTerms`) qui est vrai lorsqu'un élement donné se trouve bien dans cet ensemble. Le prédicat `context_okM` se définit de la même manière.

Nous pouvons maintenant donner la sémantique de notre langage source (les règles de conversions et les règles de validité) :

```
Inductive ConvP : PTerm → PTerm → Prop :=
| ConvPRefl : forall (u: PTerm), ConvP u u
| ConvPSym : forall (u v : PTerm), ConvP u v →
   ConvP v u
| ConvPTrans : forall (u v w : PTerm), ConvP u v →
                ConvP v w → ConvP u w
| ConvPNeg1 : ConvP (PNeg (PTrue)) PFalse
| ConvPNeg2 : ConvP (PNeg PFalse) PTrue
| ConvPAnd1 : forall (x : PTerm), ConvP (PAnd x
   PTrue) x
```

```
| ConvPAnd2 : forall (y : PTerm), ConvP (PAnd PTrue
   y) y
| ConvPAnd3 : forall (x: PTerm), ConvP (PAnd x
   PFalse) PFalse
| ConvPAnd4 : forall (y: PTerm), ConvP (PAnd PFalse
   y) PFalse
| ConvPCongLam : forall (u v : PTerm), ConvP u v →
                 (forall (A : PType), ConvP (PLam A
                    u) (PLam A v))
| ConvPCongApp : forall (u u' v v': PTerm), ConvP u
   u' → ConvP v v' →
                    ConvP (PApp u v) (PApp u' v')
| ConvPBeta : forall  (A: PType) (t u: PTerm),
       ConvP (PApp (PLam A t) u) (PSubst 0 u t)
       (* substitution, voir partie sur les
          variables *).

Inductive validP : contextP → PTerm → Prop :=
| SemPTrue : forall (c : contextP), context_okP c →
   validP c PTrue
| SemPConv : forall c (t u : PTerm), context_okP c →
       ConvP t u → validP c t → validP c u
| SemPAnd : forall  c (t u : PTerm), context_okP c →
       validP c t → validP c u → validP c (PAnd t
          u)
| SemPAssume: forall c (t u : PTerm), context_okP c
   →
       Se.In t c → validP c t
| SemPSubst : forall c (t: PTerm) (i: nat) (s:
   PType), context_okP c →
            validP c t → validP c
               (PSubst_Type_in_term i s t).
```

Les règles de conversion donne les paires de termes qui sont équivalents d'un point de vue logique : c'est donc une relation d'équivalence (réflexive, symétrique et transitive). On se place dans un cadre classique, donc la double négation de `true` est convertible à `true` et c'est la même chose pour `false`. Les règles `ConvPAnd` donnent la signification du connecteur $\wedge$. Les règles de congruence permettent de construire deux applications ou deux lambda-abstractions convertibles à partir de termes convertibles. La règle de Bêta-réduction correspond à celle du lambda-calcul simplement typé.

Les règles de validité dans le langage $\mathscr{S}$ sont

— Le terme `true` est valide dans tout contexte.

— Si un terme est valide dans un contexte et convertible en un autre, alors ce deuxième terme sera valide dans le même contexte.

— Si deux termes sont valides dans un contexte, leur conjonction le sera aussi.

— Si un terme est présent dans un contexte, il est valide par hypothèse.

— Si un terme est valide dans un contexte, on peut instancier l'une de ses variables de types et le terme résultant restera valide dans ce contexte.

Quant à la sémantique du langage d'arrivée, elle est identique à l'exception de la dernière règle, dont nous n'avons plus besoin. Cette règle du langage source assure que si un terme t est valide dans un contexte donné, alors il est toujours valide en substituant ses variables de type par n'importe quel type. En effet, nous avons dit qu'il y avait implicitement un quantificateur prénexe devant ces variables, il est donc toujours possible de les instancier. Comme il n'y a pas de variables de type dans $\mathscr{T}$, nous n'avons plus besoin de cette règle.

Désormais, nous pouvons énoncer le théorème à prouver. Il prouve que la transformation est correcte car la validité d'un terme dans un contexte traduit implique la validité de ce terme dans le contexte non traduit.

```
Theorem trad_is_sound: forall c t, context_okP
   c →
validM (trad_set c) t → validP c (inj t).
```

Ici, `inj` désigne simplement l'injection des termes de $\mathscr{T}$ dans $\mathscr{S}$ et `trad_set` traduit le contexte du langage source en un contexte du langage d'arrivée en appliquant sur chacun des termes la fonction de monomorphisation.

La preuve procède par induction sur les quatre règles de validité dans $\mathscr{T}$. Les trois premiers cas sont triviaux car nous pouvons appliquer `SemPTrue`, `SemPConv` et `SemPAnd` respectivement. Mais il n'est pas possible d'appliquer directement `SemPAssume` car si un terme est présent dans un contexte traduit, il ne l'est pas forcément dans le contexte de départ.

Considérons par exemple le terme $\lambda x : \forall X, X.\text{true}$. On a $[\lambda x : \forall X, X.\ \text{true}] = \lambda x : \text{Nat}.\ \text{true}$. On a bien sûr que $\lambda x : \text{Nat}.\ \text{true} \vdash_{\mathscr{T}} \lambda x : \text{Nat}.\ \text{true}$, en vertu de la règle "assume" mais pas $\lambda x : \forall X, X.\ \text{true} \vdash_{\mathscr{S}} \lambda x : \text{Nat}.\ \text{true}$, en vertu de cette même règle. En revanche, dans ce cas, on constate que c'est la règle d'instanciation qui a été appliquée (et elle peut l'être un nombre fini arbitraire de fois). L'essence de la preuve en Coq relie donc la règle "assume" du langage d'arrivée à la règle d'instanciation du langage de départ. Informellement, il dit que si un terme est valide dans un contexte, on peut instancier ses n premières variables de types par Nat et le terme restera valide. Par ailleurs, il faut montrer que c'est bien ce que fait la traduction, et c'est le sens du lemme suivant :

```
Lemma reverse_inj : forall x t, trad_term x = t
   → exists n,
(compo_n n (fun i u ⇒ PSubst_Type_in_term i
   PNat u) x) = inj t.
```

La fonction `compo_n` est une fonction qui en itère une autre (ici, la substitution d'une variable de type par Nat), avec un entier en paramètre. Cet entier correspond à la variable de type instanciée à chaque itération, qui doit changer jusqu'à ce qu'il n'y ait plus de variables à instancier.

Reconstituons le raisonnement : d'une part, on a que t est valide dans un contexte Γ traduit en vertu de la règle `SemPAssume`. Donc t est la traduction d'un terme x du contexte Γ. Grâce à `reverse_inj`, on déduit qu'il existe n tel que inj t est issu de l'instantiation de n variables de types par Nat. Or, comme x est valide en vertu de la règle "assume", inj t est valide en vertu de $n + 1$ applications de la règle `SemPSubst` .

Nous ne pourrons détailler l'ensemble du raisonnement car le code Coq comporte une cinquantaine de lemmes, mais nous en avons donné les lignes principales. Il faut aussi que la fonction de composition commute avec la substitution

des variables de types par Nat et, ainsi, des lemmes de commutation pour chaque constructeur de nos termes ont été démontrés. En revanche, ce travail nous a permis de dégager des caractéristiques générales sur la formalisation des transformations logiques. Lorsque nous travaillons sur la preuve de leur correction, nous aurons une partie syntaxique et une partie sémantique propre à la traduction. Ici, la partie syntaxique concerne les lemmes de commutation, et la partie sémantique est traitée par le lemme `reverse_inj`.

4.3 La gestion des variables

Nous avions laissé de côté la question de la gestion des variables mais il est temps d'y revenir. Comme, dans nos langages, il en existe un nombre dénombrable d'entre elles, elles sont distinguées par des entiers. Mais coder la substitution devient difficile. Sur papier, lorsque l'on rencontre un terme de la forme $\lambda x.u$ et qu'on veut substituer la variable y dans u par un terme t, on prend garde à ce que x ne soit pas libre dans t en renommant les variables à la volée pour éviter une capture. Dans Coq, ce renommage est difficile à implémenter. Nous pourrions penser à écrire un prédicat inductif qui corresponde à l'α-équivalence, mais faire des preuves modulo une relation d'équivalence s'avère fastidieux.

La solution consiste à utiliser les *indices de De Brujin* : le numéro de la variable désigne le nombre de lambdas à traverser pour atteindre son lieu. C'est pour cette raison que le constructeur correspondant à la lambda-abstraction dans nos deux langages ne prend que le type de la variable et un terme en paramètre. Par exemple, le terme $\lambda x.\lambda y.\lambda z.(y\ (\lambda v.\ y)v\ u)$ peut être représenté par $\lambda.\lambda.\lambda.(var\ 1\ (\lambda.\ var2\ var\ 0)\ var\ 23)$. Les variables où tous les lambdas sont traversés correspondent aux variables libres. On voit que selon la position dans le terme, on désignera la même variable par un indice ou un autre, toujours selon le nombre de lambdas à traverser. Cela complique le codage de la substitution. Pour la faire fonctionner, il y a besoin d'une fonction `Lift`. Celle-ci augmente tous les indices des variables à partir d'un certain rang. Quand on passe sous un lambda, comme la variable n est désormais désignée par l'indice $n+1$, `Lift` augmente le rang de 1. En voici le code Coq :

```
Fixpoint MLift (n: nat) (t: MTerm) := match t with
| MTrue ⇒ MTrue
| MFalse ⇒ MFalse
| MNeg u ⇒ MNeg (MLift n u)
| MAnd u v ⇒ MAnd (MLift n u) (MLift n v)
| MVar i ⇒ if i <? n then MVar i else MVar (S i)
| MLam A u ⇒ MLam A (MLift (S n) u)
| MApp u v ⇒ MApp (MLift n u) (MLift n v)
end.
```

Pour coder la substitution, on utilise cette fonction `Lift`. La substitution prend en paramètre l'indice de la variable qu'on veut remplacer, le terme par lequel on substitue et le terme dans lequel on substitue. Les deux cas intéressants de cette fonction sont le cas variable et le cas lambda.

Pour ce deuxième cas, nous savons que la variable désignée en-dehors de ce lambda par l'indice n devient d'indice $n+1$. Si notre but est de substituer celle-ci par un autre terme, il nous faut donc non plus envoyer n en paramètre

de `Subst` mais $n+1$. Par ailleurs, le terme par lequel on substitue doit être lifté, autrement, s'il comporte des *var* 0, celles-ci seront liées par le lambda sous lequel nous sommes passés.

Pour le cas variable, on remplace la variable par le terme par lequel on substitue si elle a le même indice que celui en paramètre de la substitution. Si elle a un indice plus petit, on ne change rien. Si son indice est strictement plus grand, il doit être diminué de 1 puisque dans la Bêta-réduction, on a fait disparaître le lambda qui lie la variable qu'on substitue, et que cela décale les indices supérieurs. Voilà donc le code de la substitution.

```
Fixpoint MSubst (n: nat) (s t: MTerm) : MTerm :=
match t with
| MTrue ⇒ MTrue
| MFalse ⇒ MFalse
| MNeg u ⇒ MNeg (MSubst n s u)
| MAnd u v ⇒ MAnd (MSubst n s u) (MSubst n s v)
| MLam A u ⇒ MLam A (MSubst (S n) (MLift 0 s) u)
| MApp u v ⇒ MApp (MSubst n s u) (MSubst n s v)
| MVar i ⇒ match i?= n with
        | Lt ⇒ MVar i
        | Eq ⇒ s
        | Gt ⇒ MVar (i-1)
end
end.
```

Cela permet de définir la Bêta-réduction comme dans le constructeur `ConvBeta` ci-dessus.

4.4 Méthode interprétation dans Coq : théorie

Passons à la seconde méthode. Elle est présentée par Chantal Keller dans sa thèse pour le lambda calcul simplement typé (9) et elle est reprise de l'article de Garillot et de Werner (7). Je l'ai adaptée à la théorie des types simples. Il s'agit de définir, pour le langage $\mathscr{S}$ et le langage $\mathscr{T}$, deux fonctions. L'une associe à tout type un type Coq, et l'autre associe à tout terme bien typé un terme de ce type, et échoue sinon. Comme le langage $\mathscr{S}$ contient des types polymorphes, c'est celui pour lequel les fonctions d'interprétation de types et de termes seront les plus difficiles à écrire. Nous nous concentrerons donc sur celles-ci. Notons l'interprétation des types $|\bullet|_{Type}$ et celle des termes $|\bullet|_t$. Informellement, nous devons avoir :

$|\text{PNat}|_{Type} = \text{Nat}$ (le Nat de $\mathscr{S}$ va sur le Nat de Coq)
$|\text{PBool}|_{Type} = \text{bool}$ (le bool de $\mathscr{S}$ va sur le bool de Coq)
$|A \to B|_{Type} = |A|_{Type} \to |B|_{Type}$

Pour le cas variable de type, on suppose que l'interprétation des types prend en paramètre une fonction $f : \text{Nat} \to \text{Type}$ qui à tout entier (et donc tout indice de variable), associe un type. Le code de la fonction devient donc :

```
Fixpoint interp_typeP (f : nat → Type ) (T : PType)
    :=
match T with
```

```
| PNat ⇒  nat
| PBool ⇒ bool
| Pa A B ⇒ interp_typeP f A → interp_typeP f B
| PVar i ⇒ f i
end.
```

L'interprétation des termes est plus complexe car le type de retour de la fonction dépend du type du terme pris en argument (en lambda-calcul simplement typé). Informellement, pour w un terme, $|w|_t$ doit renvoyer une paire dépendante (A, u) où A correspond à l'interprétation du type de w et u est un élément de type A. La paire dépendante se note `existT A u` en Coq. Cependant, la spécificité de Coq fait qu'il faut prévoir un constructeur pour le cas où w est mal typé, c'est pourquoi la fonction renvoie la paire dépendante dans un type option. La seconde projection donne la "vraie" interprétation du terme. Nous voulons aussi que :

$|\text{PTrue}|_{t,\mathscr{I}} = \text{true}$

$|\text{PFalse}|_{t,\mathscr{I}} = \text{false}$

$|\text{u} \wedge \text{v}|_{t,\mathscr{I}} = |\text{u}|_{t,\mathscr{I}} \wedge |\text{v}|_{t,\mathscr{I}}$ (le premier $\wedge$ est le "et" des termes du langage et le second est le "et" de Coq).

$|\neg \text{u}|_{t,\mathscr{I}} = \neg |\text{u}|_{t,\mathscr{I}}$

$|\text{u v}|_{t,\mathscr{I}} = |\text{u}|_{t,\mathscr{I}}\ |\text{v}|_{t,\mathscr{I}}$

$|x : A|_{t,\mathscr{I}} = \mathscr{I}(x : A)$, où $\mathscr{I}$ est une fonction interprétant le contexte des variables, et x est une variable.

$|\lambda x.\text{u}|_{t,\mathscr{I}} = y \mapsto |\text{u}|_{t,\mathscr{I}(x:A \leftarrow y)}$

Cette dernière équation signifie que l'environnement des variables de termes s'enrichit récursivement à chaque fois que la fonction d'interprétation rencontre une lambda-abstraction.

4.5 Méthode interprétation dans Coq : mise en oeuvre

Maintenant que nous savons les conditions que doit remplir la fonction d'interprétation, nous pouvons l'implémenter dans Coq. Il nous faut déjà définir la fonction qui interprète le contexte : à partir d'une liste de types du langage source, elle assigne à la variable d'indice n l'interprétation du n-ième terme de la liste. Si la variable n'est pas dans la liste, elle est interprétée par unit.

```
Definition interp_contextP (f : nat → Type ) (g:
    contextP)  := forall (n: nat),
match nth_error g n with
| Some A ⇒ interp_typeP f A
| _ ⇒ unit
end.
```

Ensuite, il faut deux fonctions, une qui ajoute un élément au contexte, et l'autre qui en ôte un. Elles décalent les interprétations pour les autres indices en conséquence :

```
Definition interp_tailP A (h: nat → Type) g (f:
    interp_contextP h (A::g)) :
interp_contextP h g := fun n ⇒ f (S n).
```

```
Definition interp_consP A (h: nat → Type) g (f:
    interp_contextP h g)
(a: interp_typeP h A) : interp_contextP h (A::g) :=
fun n ⇒ match n with
| O ⇒ a
| S n ⇒ f n
end.
```

Nous avons ensuite besoin d'une fonction d'interprétation des variables dépendant de leur indice de De Brujin :

```
Fixpoint interp_dbrP (h: nat → Type) (g: contextP)
    (n:nat) {struct n} :
option {A: PType & interp_contextP h g →
    interp_typeP h A} :=
match g,n return
option {A: PType & interp_contextP h g →
    interp_typeP h A} with
| nil, _ ⇒ None
| A::g, O ⇒ Some (existT A (fun f ⇒ f O))
| A::g, S n ⇒
match interp_dbrP h g n with
| Some (existT B b) ⇒
Some (existT B (fun f ⇒ b (interp_tailP _ _ _ f)))
| _ ⇒ None
end
end.
```

Enfin, il est possible d'écrire le code de la fonction d'interprétation auxiliaire (pour un contexte non vide). Comme cette fonction est longue, nous en laissons le code en annexe. Notons ici que le cas variable recourt à la fonction juste au-dessus, le cas lambda renvoie la paire entre un type flèche et une fonction qui, étant donné une interprétation du contexte, l'étend avec `interp_consP`. Le cas application est complexe : nous avons besoin d'une fonction `CastP` qui applique une coercion de types (nous expliquerons cela au prochain paragraphe). Les cas `PTrue`, `PFalse`, `PNeg` et `PAnd` sont verbeux mais ils font simplement correspondre le `PTrue` de notre langage $\mathscr{S}$ avec le `true` de Coq, le `PFalse` avec le `false` de Coq, etc, à condition que les termes en arguments des connecteurs aient bien le type booléen.

La fonction d'interprétation finale est simplement celle dans un contexte vide.

4.6 La coercion de types

Une des spécificités de Coq est que pour que les fonctions qu'on y écrit typent bien, il ne suffit pas que deux types ou objets soient égaux : nous avons aussi besoin de convertir l'un dans l'autre. Ainsi, pour que la fonction d'interprétation fonctionne dans le cas "application", il nous faut vérifier comme toujours que le premier argument est un type flèche (par exemple A) et que le second, que nous noterons C, est tel que $A = C$. Autrement dit, pour un terme `PApp u v`, on doit avoir que `infer_typeP u = Some A →B` et `infer_typeP v = C` avec `A=C`.

Mais il faut aussi écrire une fonction qui convertit C en A pour que l'on puisse effectivement effectuer l'application. C'est le rôle des différentes fonctions `cast` que j'ai écrites. Elles renvoient une fonction de conversion (selon le principe de l'égalité de Leibniz qui dit que deux éléments égaux ont les mêmes propriétés) quand deux objets sont égaux, et renvoie `NoCast` sinon, qui est un message d'erreur. Pour cela, nous avons besoin de l'inductif suivant :

```
Inductive cast_result (A: Type) (n m: A) : Type :=
| Cast (k: forall P, P n → P m)
| NoCast.
```

Ensuite, la fonction `castP`, qui s'appuie sur `castNat` (pour les entiers naturels), et convertit deux types du langage source s'ils sont égaux, s'écrit ainsi :

```
Fixpoint castP (A B : PType) : cast_result PType A
   B :=
match A as x, B as y return cast_result PType x y
   with
| PBool, PBool ⇒ Cast (fun (P : PType → Type) (x :
   P PBool) ⇒ x)
(* renvoie l'identite *)
| PNat, PNat ⇒  Cast (fun (P : PType → Type) (x : P
   PNat) ⇒ x)
(* renvoie l'identite *)
| PVar i, PVar j ⇒ match castNat i j with
          | Cast k' ⇒
let k P := let Pa y := P (PVar y) in fun x ⇒ k' Pa
   x in Cast k
          | _ ⇒ NoCast
          end
(* echoue quand les entiers sont distincts,
et s'appuie sur le cast des entiers sinon pour
   caster i en j *)
| Pa C D, Pa E F ⇒
match castP C E, castP D F with
| Cast k1, Cast k2 ⇒
let k P :=
let Pb G := P (Pa G D) in let Pc G := P (Pa E G) in
fun x ⇒ k2 Pc (k1 Pb x)
in Cast k
| _, _ ⇒ NoCast
end
| _, _ ⇒ NoCast
end.
```

Ainsi, pour la version interprétation, nous voyons que la sémantique du langage est directement donnée par la fonction d'interprétation (en annexe). Pour le moment, seules les fonctions sont écrites et il faudra ensuite prouver que si la traduction de l'interprétation d'un terme vaut `true`, ce terme vaut aussi `true`.

5 Travaux connexes

5.1 Why3 : un vérificateur de programmes

Why3 (6) est un vérificateur de programmes qui se fonde, comme SMTCoq, sur l'approche sceptique. Quand on souhaite vérifier un programme avec cet outil, on lui envoie un contrat, c'est-à-dire un ensemble de conditions que doit remplir le programme. A partir de celui-ci, Why3 génère des obligations de preuve : des formules logiques qui doivent être prouvées pour que le contrat soit valide. Celles-ci peuvent être démontrées soit par un assistant de preuve, soit par un prouveur automatique. Ces tâches de preuve sont transformées pour être comprises par un prouveur automatique. Pour cela, des transformations ont été écrites pour Why3, à laquelle l'utilisateur peut faire appel. Ces transformations composables sont semblables à celles que nous voudrions coder pour SMTCoq. Ici, le code entier de la transformation n'est pas prouvé (comme j'ai pu le faire pour la monomorphisation), puisque nous suivons l'approche sceptique. En revanche, les transformations codées fournissent un certificat qui, lui, sera vérifié. Il faut prouver que la transformation, qui s'applique à un but x et produit un certificat c et un but y, est telle que c garantit que le résultat y implique x.

5.2 F* : un langage de programmation fonctionnel

F* est un langage de programmation fonctionnel dont le but est également la certification de programmes. Il utilise un prouveur automatique qui est le prouveur SMT Z3. L'utilisateur peut exprimer des conditions pour son programme, et le vérificateur de type du langage génère des conditions de vérification dans un système de type d'ordre supérieur. Pour les prouver, il les encode ensuite dans la logique du premier ordre en une seule étape et les envoie au prouveur automatique (SMT). Cependant, la preuve de la correction de cet encodage n'a pas encore été effectuée à cause de sa complexité malgré une tentative (1). Il faut donc pour l'instant faire confiance à cet encodage. Pour cette raison, utiliser des transformations plus simples, prouver séparément leur correction et les composer entre elles (solution envisagée pour SMTCoq et commencée durant mon stage) est une alternative intéressante.

5.3 CoqHammer : une autre interface prouveur/Coq

Nous avions aussi cité CoqHammer : un outil permettant d'augmenter l'automatisation dans Coq en appelant une tactique puissante appellée Hammer (12). Il s'inspire de SledgeHammer pour l'assistant de preuve Isabelle\HOL (2). L'idée est d'envoyer aux prouveurs SMT des buts Coq traduits dans la logique du premier ordre par une grande transformation logique non certifiée, qui opère sur un fragment du Calcul des Constructions Inductives. Elle traduit à la fois les inductifs, le polymorphisme, la relation de typage, les points fixes, le produit dépendant... Mais, encore une fois, la complexité de la transformation fait qu'aucune preuve de correction n'existe à ce jour. Cependant, la correction de CoqHammer repose sur le fait que la preuve est reconstruite par rapport au but de départ. Ainsi, même sans prouver la correction de la transformation, la tactique "hammer" reste correcte.

6 Conclusion et perspectives

6.1 Rappel du résultat principal

Rappelons le résultat principal : il existe deux méthodes pour certifier une transformation logique.

1. Soit on définit un prédicat de validité dans le langage source et dans celui d'arrivée, et on montre que si la traduction est valide, le terme initial l'est aussi.
2. Soit on interprète le langage source et le langage d'arrivée par des termes de Coq, on montre que si la traduction d'un terme a une interprétation, alors ce terme en a un aussi.

Ces deux méthodes sont liées, car la validité d'un terme dans un des deux langages implique que son interprétation est valide. Durant mon stage, j'ai prouvé le premier d'entre eux pour deux langages simples et une monomorphisation particulière, et j'ai défini les fonctions d'interprétation qui permettront de montrer le second.

6.2 Généralisation du résultat

Pour généraliser le résultat, il faudrait pouvoir ne plus appliquer une instanciation précise (l'instanciation par Nat), mais une instanciation par plusieurs types différents par variable. On aurait alors une fonction de traduction du contexte qui prendrait en paramètre une liste de types arbitraires Γ, et qui instancierait la variable de type d'indice n de chaque terme du contexte par le n-ième élement de Γ. Cela complexifie le code pour deux raisons :

— Tout d'abord, il n'est plus possible de séparer les deux langages, la transformation doit se faire au sein du même langage. En effet, le type du terme résultant peut encore comporter des variables. De plus, le type arbitraire par lequel on instancie peut très bien toujours comporter des variables de type polymorphes. Cela n'a pas d'intérêt pour construire un terme à envoyer au prouveur mais il nous faut gérer ce cas pour avoir une preuve de correction générale, ou alors définir un prédicat `NoVariable` qui vérifie que le type par lequel on instancie n'a pas de variable de type.

— Ensuite et surtout, quand on instancie successivement des variables de type (par exemple, on instancie la variable 0 puis la variable 1), il faut faire attention aux captures de variables. Si le type par lequel on instancie la variable 0 contient la variable 1, instanicer 0 puis 1 ne sera pas équivalent à instancier 1 puis 0. Ma preuve originelle contient de nombreux lemmes de commutation qui ne seront plus valides et qu'il me faudra retravailler. Sinon, je devrais utiliser le prédicat `NoVariable` sur les types par lesquels j'instancie, car, pour ceux-ci, les lemmes de commutation fonctionneront.

Toutefois, pouvoir instancier par un type arbitraire serait très utile pour SMTCoq. A partir d'un but polymorphe et d'une procédure de décision qui permettrait de trouver le "bon" type par lequel instancier, il sera alors possible d'envoyer un but monomorphe au prouveur. Cette procédure de décision n'aura pas à être certifiée car, quelle que soit l'instanciation choisie, nous l'aurons prouvée correcte. J'espère pouvoir poursuivre ce travail durant ma thèse.

6.3 Intégration au code de SMTCoq

Pour intégrer la transformation que j'ai codée dans SMTCoq, il faudra coder une procédure de décision qui, étant donné un but Coq, applique ou non la monomorphisation (et de préférence, la monomorphisation généralisée). De plus, il reste à prouver que la validité telle que je l'ai définie avec un prédicat inductif implique bien la validité de l'interprétation de nos termes. Autrement dit, il faut montrer le théorème suivant :

```
Theorem valid_implies_interp : Valid nil t →
   Interp t = true
```

D'ailleurs, il peut se généraliser à une liste non vide (si le terme t est valide dans un contexte Γ, alors son interprétation est valide quand la conjonction de tous les termes de Γ est vraie).

6.4 De nouvelles transformations

Dans notre paragraphe sur les transformations logiques, nous avions cité d'autres transformations possibles à intégrer à SMTCoq. Nous pouvons ici lister plusieurs possibilités :

- L'encodage des types inductifs.
- La skolémisation.
- L'encodage de l'ordre supérieur avec le symbole d'application syntaxique dont nous avions parlé au paragraphe 3.
- La défonctionnalisation, pour éliminer le problème des fonctions prenant en paramètre d'autres fonctions (ce qui n'est pas possible en logique du premier ordre). Par exemple, pour une liste l d'entiers et pour encoder $map\ (x \mapsto x + 1)\ l$, on considère que map est de type $A \to list\ \text{Nat} \to list\ \text{Nat}$, on considère f_0 de type A et on définit la fonction $apply$ telle que pour tout entier i, $apply\ f_0\ i = i + 1$.
- L'encodage de la relation de typage, avec un prédicat de typage ajouté à notre logique du premier ordre.

Pendant ma thèse, j'espère pouvoir coder et certifier plusieurs de ces transformations dans le but de fournir une tactique automatique en Coq très efficace.

Bilan du stage : compétences annexes

Durant ce stage, j'ai développé mes compétences en Coq. Notamment, j'ai découvert les modules (en travaillant sur les ensembles finis), de nouvelles tactiques comme exact, now, intuition, remember, et les sections qui permettent d'abstraire une partie du code avant de l'implémenter.
Par ailleurs, j'ai découvert l'outil de travail GitHub, qui permet de mettre à jour son travail, de le poster en ligne, et de conserver les modifications de son code.
J'ai aussi pu user de nouvelles fonctionnalités en LaTeX (manipulation d'images, de tableaux...).
Le télétravail m'a conduite à utiliser mon ordinateur personnel. Comme mon système d'exploitation n'est pas Linux, j'ai également découvert le terminal Cygwin.
Ma formation étant originellement mathématique, j'ai donc pu me familiariser avec les outils du chercheur en informatique.

Remerciements

Mes remerciements vont tout d'abord à mes encadrants de stage : madame Chantal Keller et monsieur Valentin Blot. Leur grande disponibilité, leurs conseils et nos discussions m'ont toujours permis d'avancer lorsque je rencontrais des difficultés. Ils m'ont aussi permis de découvrir le monde de l'automatisation et m'auront beaucoup enthousiasmée. Je serai très heureuse de continuer à travailler avec eux en thèse et de poursuivre le développement de mes recherches.

Je remercie également Pierre Vial, post-doctorant à Paris-Saclay, pour ses explications, ses suggestions et nos discussions informelles qui m'ont communiqué une bonne vision de l'articulation de différents domaines de recherche entre eux.

Mes remerciements vont aussi à mes professeurs du Master de Logique Mathématique et des Fondements de l'Informatique qui m'ont permis d'effectuer ce stage malgré le confinement.

Pour finir, je remercie toute l'équipe de la Deducteam pour son excellent accueil malgré le télétravail, et les conseils que ses membres m'ont prodigués.

Appendice : le code de la fonction d'interprétation auxiliaire

```
Fixpoint interp_auxP (h: nat → Type) (g: contextP) (t: PTerm) :
option {A: PType & interp_contextP h g → interp_typeP h A} :=
 match t with
| PTermVar i ⇒ interp_dbrP h g i
| PLam A u ⇒
match interp_auxP  h (A::g) u with
| Some (existT U d) ⇒
Some (Pa A U) (fun f z ⇒ d ((interp_consP _ _ _ f z)))
| _ ⇒ None
end
| PApp u v ⇒
match interp_auxP h g u, interp_auxP h g v with
| Some (existT (Pa A B) b), Some (existT C c) ⇒
match castP C A with
| Cast k ⇒ Some (existT B (fun f ⇒ (b f) (k _ (c f))) )
| _ ⇒ None
end
| _, _ ⇒ None
 end
| PTrue ⇒ Some
    (existT PBool (fun f ⇒ true))
| PFalse ⇒ Some
    (existT PBool (fun f ⇒ false))
| PNeg x ⇒ match interp_auxP h g x with
          | Some (existT PBool y) ⇒
            Some (existT PBool
                                (fun w ⇒ (match (y w) with
                                                                  | true ⇒
                                                                      false
                                                                  | false ⇒
                                                                      true
                                                                end)))
          | _ ⇒ None
end
| PAnd x y ⇒ match interp_auxP h g x with
          |  Some (existT PBool u) ⇒ match interp_auxP h g y with
                |  Some (existT PBool v) ⇒
                Some (existT PBool
                                        (fun w ⇒ (match (u w, v w) with
                                                | (true, true) ⇒ true
                                                 | (_, _) ⇒ false
                                                end)))
                                    | _ ⇒ None
end
          | _ ⇒ None
end
end.
```

Cette fonction prend en paramètre la fonction qui interprète les variables de type, le contexte (une liste de types pour les variables de terme) et le terme à interpréter. Elle renvoie une paire dépendante : sa première projection est l'interprétation du type de ce terme et sa deuxième projection est une fonction qui, étant donné une interprétation du contexte, renvoie un élément de l'interprétation du type. Si le terme est mal typé, elle renvoie `None`.

Elle effectue un *pattern matching* sur le terme t :

— Si c'est une variable, elle utilise la fonction `interp_dbrP` (voir paragraphe 4).

— Si c'est une lambda-abstraction, on ajoute au contexte le type de la variable abstraite et on regarde l'interprétation du terme sous le lambda dans ce contexte étendu.

— Si c'est une application `PApp u v`, on vérifie qu'elle est bien typée et on applique la fonction de coercion entre le domaine du type de u et le type de v comme expliquée au paragraphe 4.

— Si c'est `PTrue`, alors l'interprétation du type de t est `bool` et, quelle que soit l'interprétation du contexte, le terme t est interprété par le `true` booléen de Coq.
— Si c'est `PFalse`, alors l'interprétation du type de t est `bool` et, quelle que soit l'interprétation du contexte, le terme t est interprété par le `false` booléen de Coq.
— Si c'est une négation, il faut vérifier que l'argument du connecteur a bien pour type `PBool` et dans ce cas, on regarde si son interprétation vaut `true` ou `false`. Si elle vaut `true`, alors la négation la change en `false`, et vice-versa.
— Si c'est une conjonction, il faut vérifier que ses deux arguments sont bien de type `PBool` et dans ce cas, l'interprétation de la conjonction reproduit les règles de sémantique de $\wedge$ en logique classique.

Références

[1] A. Aguirre : Towards a provably correct encoding from F* to SMT. Inria Internship Report, août 2016.

[2] J. Blanchette : Hammering away : A user's guide to sledgehammer for Isabelle/HOL, 2020.

[3] A. Church : A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[4] T. Coquand et G. Huet : Constructions : a higher order proof system for mechanizing mathematics. *European Conference on Computer Algebra*, p. 151–184, 1985.

[5] T. Coquand et C. Paulin : Inductively defined types. *Lecture Notes in Computer Science*, 417, 1990.

[6] Q. Garchery, C. Keller, C. Marché et A. Paskevich : Des transformations logiques passent leur certificat. *Journées Francophones des Langages Applicatifs*, 2020.

[7] F. Garillot et B. Werner : Simple types in type theory : deep and shallow encodings. *Theorem Proving in Higher Order Logics, 20th International Conference*, 2007.

[8] J. Harrison : Towards self-verification of HOL light. *Lecture Notes in Computer Science*, (4130), 2006.

[9] C. Keller : *A Matter of Trust : Skeptical Communication Between Coq and External Provers*. Thèse de doctorat, Ecole Polytechnique, Palaiseau, France, 2013.

[10] O. Laurent : Théorie de la démonstration, 2008.

[11] M. Lecat : *Erreurs de mathématiciens des origines à nos Jours*. Castaigne, Bruxelles, 1935.

[12] Łukasz Czajka et C. Kaliszyk : Hammer for coq : Automation for dependent type theory. *Journal for Automated Reasoning*, (61):423–453, 2018.