Décision de relations inductives pour SMTCoq *

Démonstration d'un prototype

Louise Dubois de Prisque[1,2]

[1] Université Paris-Saclay, CNRS, ÉNS Paris-Saclay, Laboratoire de Méthodes Formelles
[2] Inria

Résumé

Les utilisateurs de l'assistant de preuve Coq écrivent le plus souvent les spécifications des objets sur lesquels ils prouvent des énoncés sous la forme de relations inductives. Cependant, la plupart des prouveurs automatiques externes ne savent pas les interpréter et cela restreint donc leur usage au sein de Coq. Nous présentons un plugin Coq qui permet, étant donné une relation inductive décidable dans `Prop`, de définir automatiquement son pendant booléen. Une preuve de correction est également générée dans certains cas. Notre outil est intégré à la tactique `snipe`, une extension de SMTCoq qui préprocesse des énoncés Coq puis appelle un prouveur SMT pour les prouver automatiquement. Lors de notre démonstration, nous montrerons au travers d'exemples qu'il permet à `snipe` d'automatiser des preuves au sujet de spécifications inductives décidables.

1 Un problème de décision

La manière naturelle d'écrire des spécifications (même décidables) dans l'assistant de preuve Coq est d'utiliser des relations inductives et non d'écrire des fonctions booléennes. En effet, cela permet à l'utilisateur de ne décrire que les cas où la relation est vraie, d'obtenir un code plus lisible où chaque constructeur de la relation correspond à une règle d'inférence, d'user des principes d'inversion associés au sein d'une preuve, et de ne pas s'occuper des questions de décidabilité. Considérons le prédicat `mem`, qui est vrai lorsque le premier argument appartient à la liste donnée en second argument. Sous la forme d'une relation inductive, il s'écrit :

```
Inductive mem : Z → list Z → Prop :=
| MemMatch : forall (xs : list Z) (z : Z), mem z (z :: xs)
| MemRec : forall (xs : list Z) (z z' : Z), mem z xs → mem z (z' :: xs).
```

Le problème de cette forme est qu'elle n'est pas utilisable par un outil ou une tactique d'automatisation qui ne connaît pas les relations inductives. En particulier, l'outil SMTCoq [3], qui appelle des prouveurs SMT au sein de Coq, et utilise donc une logique classique, ne permet de raisonner que sur des fonctions booléennes. Ainsi, un but tel que celui-ci :

```
Lemma mem_imp_not_nil s l : mem z l → l <> [].
```

ne peut pas être prouvé par SMTCoq. En effet, la formule ne pourra pas être traduite dans les booléens sans transformation préalable de `mem` en fonction booléenne équivalente `mem_dec`. Mais, en supposant cette fonction donnée, SMTCoq, via la tactique d'automatisation `snipe`, sera capable de prouver le but suivant :

```
Lemma mem_dec_imp_not_nil z l : mem_dec z l = true → l <> [].
```

Un fichier d'exemple est disponible à l'adresse suivante : https://github.com/smtcoq/sniper/blob/JFLA23/theories/deciderel/examples.v

2 Contribution

La fonction `mem_dec`, est générée par une nouvelle commande `decide` sous la forme d'un point fixe :

*Cette contribution est soutenue par un partenariat entre Nomadic Labs et l'Inria.

```
Fixpoint mem_dec (z : Z) (l : list Z) :=
  match l with
  | [] ⇒ false
  | x :: _ ⇒ Z.eqb z x (* une egalite decidable sur Z *)
  end || match l with
         | [] ⇒ false
         | _ :: xs ⇒ mem_dec z xs
         end
```

La commande génère automatiquement cette fonction en effectuant une disjonction booléenne (`orb`) entre deux fonctions intermédiaires qui correspondent respectivement à la règle `MemMatch` et à la règle `MemRec`. Les fonctions intermédiaires sont des filtrages par motifs qui renvoient `true` quand la conclusion d'un constructeur est applicable. Celles-ci peuvent nécessiter une linéarisation dans le type du constucteur, dans ce cas, la commande recherche s'il y a des égalités booléennes existantes sur les types donnés, mais elle ne les construit pas automatiquement.

Afin de permettre à SMTCoq de prouver automatiquement des buts mentionnant des prédicats décidables, nous présentons donc un prototype de commande `decide` générant une fonction booléenne équivalente et son intégration au sein du plugin `Sniper` [1].

Elle s'applique sur les inductifs `I` au codomaine dans `Prop`, avec des arguments de type `Set`, ne présentant pas d'ordre supérieur ou de dépendance, et dont les types des constructeurs sont de la forme :

$\forall\,(x_1 :\, A_1),\,\ldots,\,(x_n :\, A_n),\, P_1\, x_1\, \ldots x_n \to \ldots \to\, P_k\, x_1\, \ldots x_n \to\, I\, t_1 \ldots t_l$

Les A_i sont des types inductifs de `Set`, chaque P_i est un prédicat sur tout ou partie des x_i, et les t_i sont des termes qui peuvent contenir des constructeurs appliqués aux x_i. Notons qu'une restriction est que tous les x_i doivent apparaître dans la conclusion $I\, t_1 \ldots t_l$. En générant le point fixe booléen qui décide la relation, elle utilise une heuristique simple pour trouver l'argument qui décroît.

3 Intégration au plugin `Sniper`

La commande `decide` fait partie des transformations logiques du plugin `Sniper`. L'idée de ce plugin est de combiner des transformations correctes de pré-processing sur un but Coq G, pour obtenir un but G', prouvable par une tactique d'automatisation et tel que $G' \to G$. La transformation `decide`, appliquée sur une relation R, génère dans l'environnement global une fonction booléenne R_{dec} et une obligation de preuve que R_{dec} décide R (ou le terme de preuve quand la tactique fournie avec le plugin a réussi). Ensuite, une autre transformation appelée `trakt` [2], permet de remplacer toutes les occurences de R dans le but par des occurences de R_{dec}. A la fin de l'étape de pré-processing, le but transformé est envoyé au prouveur SMT veriT et prouvé automatiquement par celui-ci. Un certificat de preuve est renvoyé et vérifié dans Coq.

4 Travaux connexes

Pierre-Nicolas Tollitte et *al.* ont également travaillé sur la décision de relations inductives en Coq [5]. Leur approche est différente car leur objectif n'était pas seulement d'obtenir une fonction booléenne à partir d'une relation inductive, mais aussi d'obtenir le k-ième argument d'une relation étant donné les autres arguments. Cependant, leur développement ne gère pas les inductifs avec des paramètres, et les problèmes de terminaison concernant les point fixes sont gérés par du *fuel*. Or, utiliser du *fuel* fait que l'on prouve un énoncé de correction avec un quantificateur existentiel sur la variable de *fuel*. Ce genre d'énoncé n'est pas exploitable par SMTCoq, qui ne supporte pas leur utilisation. Nous avons préféré cibler un fragment des relations inductives qui supporte les paramètres et obtenir des preuves de correction sans *fuel*. Par ailleurs, Zoe Paraskevopoulou et *al.* ont unifié plusieurs manières d'extraire du contenu calculatoire à partir de relations inductives [4], mais leurs fonctions ne sont pas totales et donc restent inexploitables pour un outil tel que SMTCoq.

Références

[1] Valentin Blot, Louise Dubois de Prisque, Chantal Keller, and Pierre Vial. General automation in Coq through modular transformations. In Chantal Keller and Mathias Fleury, editors, *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021*, volume 336 of *EPTCS*, pages 24–39, 2021.

[2] Denis Cousineau, Enzo Crance, and Assia Mahboubi. Trakt, uniformiser les types pour automatiser les preuves. In *JFLA'22 : Journées Francophones sur les Langages Applicatifs*, Annual Workshop. INRIA, INRIA, 2022.

[3] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq : A plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[4] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22 : 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 966–980. ACM, 2022.

[5] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing certified functional code from inductive specifications. In Chris Hawblitzel and Dale 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.