DÉMONSTRATION AUTOMATIQUE EN COQ

Quentin Garchery

sous la direction de

Chantal Keller
Maître de Conférences
Université Paris-Sud

Valentin Blot
Post-doctorant
Université Paris-Sud

Stage au LRI, Paris-Saclay
Université Paris-Sud / CNRS

Mars-Août 2018

1 Fiche de synthèse

1.1 Contexte général: méthodes formelles

Les méthodes formelles rassemblent différents logiciels formels qui permettent de formuler des propriétés mathématiques puis de les vérifier. La validité du résultat ne dépend alors que de ce logiciel de vérification. C'est dans ce cadre que G. Gonthier et B. Werner ont prouvé, en Coq, le théorème des quatre couleurs.

Les méthodes formelles s'étendent à la preuve de programme: il s'agit alors de vérifier qu'un programme correspond à sa spécification. C'est notamment le cas de Compcert [10] qui est un compilateur de code C qui a été certifié en Coq par X. Leroy. La certification d'un programme permet de s'assurer de la robustesse et de la fiabilité de celui-ci. L'importance de la méthode et de la correction des logiciels en général est mise en avant dans le cas des systèmes critiques. En effet, l'échec du premier lancement d'Ariane 5 (1996) est dû à un *bug* logiciel.

1.2 Problème étudié

Parmi ces logiciels formels, on s'intéressera aux assistants de preuves (2.1) et aux prouveurs automatiques (2.2) et plus particulièrement aux interfaces entre un assistant de preuve et différents prouveurs automatiques. De telles interfaces façonnent une preuve à partir du certificat fourni par un des prouveurs automatiques, dans l'objectif d'améliorer l'automatisation de l'assistant de preuve considéré. Ce fonctionnement est celui de Coqhammer [6] pour l'assistant de preuve Coq et de Sledgehammer [5] pour Isabelle.

Pendant mon stage, j'ai travaillé sur SMTCoq [1, 9] qui sert aussi d'interface à différents prouveurs automatiques mais qui a la particularité de reproduire fidèlement, en Coq, le certificat reçu. Cette approche permet également de vérifier les certificats fournis par les prouveurs et donc d'améliorer la confiance que l'on a dans ces outils.

Nous aimerions que le développement qui s'appuie sur cette automatisation soit adapté aux assistants de preuves: les preuves sont modulaires et peuvent reposer sur des lemmes précédemment démontrés. Nous voudrions aussi pouvoir formaliser une théorie en partant des axiomes de celle-ci puis démontrer automatiquement de nouvelles propriétés de cette théorie.

1.3 Contribution proposée

Pendant mon stage, je me suis attaché à améliorer l'expressivité de SMTCoq de ce point de vue là. J'ai donc rajouté la possibilité de transmettre à SMTCoq des axiomes ou des lemmes déjà démontrés. Cet ajout s'est traduit par une extension de la logique utilisée et se retrouve dans toutes les étapes intermédiaires de SMTCoq, étapes j'ai étendues en conséquence. Cet aspect est original: la technique d'encodage des instanciations des lemmes que j'ai utilisée (7.5) permet d'alléger la suite de la vérification. Enfin, j'ai automatisé le procédé de vérification final afin de préserver la facilité d'utilisation de SMTCoq.

1.4 Arguments en faveur de la validité de la contribution

En plus des tests présents dans SMTCoq initialement, le code final passe d'autres tests pour s'assurer du gain d'expressivité dû au rajout des lemmes et des quantificateurs. J'ai, par exemple, pu vérifier des propriétés, automatiquement et dans Coq, en théorie des groupes, dans une théorie formalisant les listes d'entiers, sur des fonctions définies récursivement, etc.
D'autre part, ma contribution respecte le principe sceptique de SMTCoq (5.1), le développement et le calcul étant faits principalement en OCaml, en dehors de l'assistant de preuve. Cet aspect donne une meilleure robustesse à SMTCoq face aux changements internes des prouveurs automatiques.

1.5 Bilan et perspectives

Les formules acceptées par SMTCoq doivent porter sur les termes concrets du type des booléens et doivent aussi être en forme prénexe. On pourrait améliorer l'expressivité en étendant les cas d'application (5.2.3) aux termes du type des propositions en Coq.
On pourrait aussi utiliser des méthodes de *machine learning* pour sélectionner les lemmes à envoyer au prouveur automatique comme c'est fait dans Coqhammer et Sledgehammer [4, 6]. L'avantage étant qu'avec des lemmes pertinents et en petit nombre le prouveur automatique trouve plus rapidement et plus souvent la preuve du théorème en question.
Un autre but que l'on souhaite poursuivre est de certifier le logiciel de vérification Why3 [7]. Puisque Why3 utilise des prouveurs automatiques, il faut alors pouvoir certifier les démonstrations faites par ces prouveurs. Ce sujet est lié à celui de ma thèse intitulée "Certification de la génération et la transformation d'obligations de preuves" et encadrée par Claude Marché, Chantal Keller et Andrei Paskevich. À cette occasion, j'utiliserai SMTCoq et je profiterai de l'amélioration de son expressivité due à mon stage. Je pourrai l'améliorer encore en étendant le format des lemmes que l'on peut transmettre à SMTCoq.

Ces améliorations de l'expressivité et de l'efficacité combinées avec la facilité d'utilisation et la robustesse de SMTCoq ont pour objectifs d'en faire un outil accessible et de généraliser son utilisation dans les projets développés en Coq.

2 Logiciels utilisés

2.1 Assistants de preuve

Les assistants de preuves sont des outils puissants qui permettent d'exprimer des théorèmes complexes puis de les vérifier de manière interactive. Ils proposent à un utilisateur de formuler son problème puis de le démontrer, le rôle principal de l'assistant de preuve étant alors de vérifier que la preuve fournie est correcte. Pour une propriété donnée, l'utilisateur doit donc construire une preuve parfaitement rigoureuse et exhaustive de la propriété ce qui peut rendre le processus de vérification long et fastidieux. La confiance accordée à ces outils dépend de la compréhension que l'on peut avoir dans son noyau, étant donné que c'est la partie sur laquelle repose la vérification. Afin de faciliter cette compréhension, les assistants de preuve favorisent une implantation dans un langage de haut niveau proche de la logique de leur noyau tel que OCaml. L'accent est mis sur la concision et la clarté du code.

Dans la suite nous utiliserons Coq comme assistant de preuve. La logique de son noyau se fonde sur le calcul des constructions inductives [13] (ou types inductifs). La sémantique du langage n'est pas donnée en détail dans ce rapport mais les aspects importants seront précisés au moment de leur utilisation. La partie 3 introduit deux techniques d'utilisation de Coq.

2.2 Prouveurs automatiques

Les prouveurs automatiques, quant à eux, ne demandent pas de preuves de la part de l'utilisateur. L'effort de certification est alors réduit à la formalisation du problème et dans certains cas le prouveur donne une trace de son exécution appelée certificat. En contrepartie, la logique d'un prouveur automatique est plus limitée et/ou la réponse en temps fini n'est pas garantie. Puisqu'un prouveur automatique doit chercher la preuve du théorème entré, l'efficacité de son implantation est capitale. Pour cette raison, les prouveurs automatiques sont écrits dans des langages de plus bas niveau tels que C ou C++ et font usage de structures mutables complexes.

La partie 4 détaille le fonctionnement des prouveurs automatiques ainsi que l'utilisation que nous en ferons.

2.3 SMTCoq

Une interface entre assistant de preuve et prouveurs automatiques telle que SMTCoq offre les avantages des deux types de logiciels formels décrits ci-dessus. Un autre avantage de SMTCoq est sa modularité: il est en effet possible de rajouter d'autres prouveurs automatiques, ceux-ci n'étant pas nécessairement du même type (prouveurs SAT/SMT) et n'ayant pas nécessairement le même format d'entrée ni le même format de sortie.

La partie 5 présente SMTCoq qui est l'objet du stage et qui est un projet actuellement développé par Chantal Keller en collaboration avec l'Université de l'Iowa. Les fichiers source de SMTCoq sont écrits en OCaml et en Coq qui sont aussi les langages utilisés pendant le stage.

Les fragments de code Coq fournis sont encadrés et sont compilables lorsqu'ils sont chargés dans l'environnement de SMTCoq. Il faut donc les préfacer par

```
Require Import SMTCoq Bool.
Open Scope Z_scope.
```

Cela requiert l'installation de SMTCoq, disponible à l'adresse

`https://github.com/smtcoq/smtcoq`

On préférera la version utilisant native-coq [2].

3 Techniques de preuves en Coq

Cette partie présente deux techniques de preuves en Coq qui peuvent être combinées: la réflexion calculatoire et la réification. Ces techniques sont au cœur du fonctionnement de SMTCoq.

3.1 Réflexion calculatoire

Pour définir de nouveaux termes en Coq, on peut utiliser un type inductif ou une définition. Nous allons voir que la réflexion calculatoire n'est possible que dans le second cas. Nous commençons par détailler les types inductifs, ce qui permettra de mettre en avant cette différence.

3.1.1 Types inductifs

Prenons l'exemple des formules conjonctives booléennes. Ces formules sont des éléments du type inductif `AndTree` qui a un constructeur `Bool` pour les feuilles et un constructeur `And` pour les nœuds:

```
Inductive AndTree :=
  Bool (b : bool)
| And (left: AndTree) (right: AndTree).
```

Définissons aussi un type inductif qui donne l'interprétation d'une formule conjonctive:

```
Inductive Interp : AndTree -> bool -> Prop :=
  InterpBool b :
    Interp (Bool b) b
| InterpAnd b1 b2 b3 t1 t2 :
    Interp t1 b1 -> Interp t2 b2 -> b1 && b2 = b3 ->
    Interp (And t1 t2) b3.
```

`Interp t b` signifie que l'arbre `t` s'interprète en un booléen `b`. Cette définition suit la définition des formules conjonctives. En effet, il y a bien deux cas, `InterpBool` pour les feuilles de l'arbre et `InterpAnd` pour les nœuds. Le cas `InterpAnd` s'explique ainsi: si `t1` s'interprète en `b1` et si `t2` s'interprète en `b2`, alors `And t1 t2` s'interprète en `b1 && b2`. On a noté `&&` la fonction Coq `andb` qui implémente la conjonction booléenne.
On peut alors faire des preuves sur les éléments de ce type inductif:

```
Definition t := And (And (Bool true) (Bool false)) (And (Bool true) (Bool true)).

Lemma Interp_t_false : Interp t false.
Proof.
  eapply InterpAnd ; [
    eapply InterpAnd ; [ apply InterpBool | apply InterpBool | reflexivity ]
  | eapply InterpAnd ; [ apply InterpBool | apply InterpBool | reflexivity ]
  | reflexivity ].
Qed.
```

3.1.2 Définitions en Coq et convertibilité

La réflexion calculatoire repose sur la convertibilité de deux termes: deux termes sont convertibles lorsqu'ils se réduisent vers un même terme. Aussi, à chaque nouvelle définition, la nouvelle constante qui est définie est convertible à sa définition.

Plutôt que d'utiliser un prédicat qui prend une formule conjonctive `t` et un booléen `b`, et qui énonce que `t` s'interprète en `b`, on peut définir une fonction récursive qui calcule l'interprétation et combiner

cette fonction avec le prédicat de l'égalité. On peut ensuite montrer que les deux formalisations sont bien équivalentes.

```
Fixpoint interp (t : AndTree) :=
  match t with
    Bool b => b
  | And t1 t2 => interp t1 && interp t2
  end.

Proposition Interp_eq_interp t b :
  Interp t b <-> interp t = b.
```

L'avantage de cette définition de l'interprétation par rapport au type inductif est que, grâce à la convertibilité de `interp` à sa définition, on a une preuve triviale du lemme précédent.

```
Lemma interp_t_false : interp t = false.
Proof.
  reflexivity.
Qed.
```

En effet, en notant `a` ≡ `b` lorsque `a` est convertible à `b`, on a:

```
interp t ≡ interp (And (Bool true) (Bool false)) &&
           interp (And (Bool true) (Bool true))
         ≡ (interp (Bool true) && interp (Bool false)) &&
           (interp (Bool true) && interp (Bool true))
         ≡ (true && false) && (true && true)
         ≡ false && (true && true)
         ≡ false
```

Ce fonctionnement peut être exploité pour construire des preuves qui reposent sur un calcul de convertibilité de termes Coq.

3.2 Réification

3.2.1 *Embeddings*

La réification est le fait de passer d'un *shallow-embedding* à un *deep-embedding*. Dans le cas du *deep-embedding*, un terme est représenté dans un nouvel AST ce qui met en évidence sa structure. À l'inverse, un *shallow-embedding* du même terme est traduit directement vers sa valeur dans le langage cible.

Reprenons l'exemple des formules conjonctives et considérons la formule $u := (b_1 \wedge b_2) \wedge (b_3 \wedge b4)$. À l'instar de la section précédente, le *deep-embedding* de u est donné dans le type `AndTree`:

```
And (And (Bool b1) (Bool b2)) (And (Bool b3) (Bool b4))
```

et son *shallow-embedding* peut être donné en utilisant la conjonction booléenne de Coq:

```
(b1 && b2) && (b3 && b4)
```

Dans la suite, on s'intéresse au problème de mettre des formules booléennes conjonctives en forme de peigne. Avec `b1`, `b2`, `b3` et `b4` des termes Coq de type `bool`, on veut, par exemple, pouvoir passer de:

```
v := (b1 && b2) && (b3 && b4)    à    v' := b1 && (b2 && (b3 && b4))
```

Pour cela, on a besoin de récupérer la structure du booléen `v`, c'est l'étape de réification. Il s'agit donc de construire, à partir de `v`, un terme du type `AndTree` qui a la même structure que `v`.

3.2.2 Méthodes

Il n'est pas possible d'écrire en Coq une fonction qui détermine la forme conjonctive d'un argument booléen. En effet, le type `bool` n'ayant que deux constructeurs (`true` et `false`), inspecter par *pattern-matching* nous donnera un de ces deux constructeurs.

Une solution est d'étudier la structure du terme en question à partir de sa représentation OCaml sous-jacente. C'est l'approche utilisée par SMTCoq.

Il est aussi possible d'utiliser des tactiques qui renvoient un terme. Dans le cas des formules conjonctives, on définit:

```
Ltac reify A := match A with
  | andb ?X ?Y => let rx := reify X in
                  let ry := reify Y in
                  constr:(And rx ry)
  | ?X => constr:(Bool X) end.
```

Le terme `u := (b1 && b2) && (b3 && b4)` réifié donne bien

```
And (And (Bool b1) (Bool b2)) (And (Bool b3) (Bool b4))
```

et on notera que l'interprétation de ce nouveau terme est convertible à `u`.

3.2.3 Intérêt et exemple d'utilisation

L'intérêt de la réification est que la structure du terme réifié est mise en évidence. Il devient alors possible de manipuler explicitement cette structure.

Par exemple, sur le type `AndTree`, il est possible de définir une fonction `peigne` qui renvoie l'arbre en argument mis sous forme de peigne. Le théorème de correction de cette fonction établit que

```
forall t : AndTree, interp (peigne t) = interp t
```

Ce théorème de correction n'est applicable que si on a un terme de la forme `interp x`, forme que l'on peut obtenir en s'assurant que pour tout terme booléen `b`, l'interprétation de la réification de `b` est convertible à `b`.

En combinant tous ces résultats, on peut définir une tactique `peignify` qui met les formules conjonctives en forme de peigne. Cette tactique permet par exemple de démontrer le lemme suivant:

```
Lemma peigne4 b1 b2 b3 b4:
  (b1 && b2) && (b3 && b4) = b1 && ((b2 && b3) && b4).
Proof.
  peignify. reflexivity.
Qed.
```

Cette preuve a un contenu calculatoire: le calcul de la fonction `peigne` sur la réification du terme `u` de type `bool`. Pour le code complet de cette partie et en particulier le code de la tactique `peignify`, voir l'annexe A, la preuve de correction est inspirée de [13], section 3.3.

4 Prouveurs automatiques

4.1 Utilisation par SMTCoq

Différents prouveurs automatiques sont mis à la disposition de l'utilisateur de SMTCoq. Parmi ceux-ci, il y a zChaff, un prouveur SAT, c'est-à-dire un prouveur qui résout des problèmes de satisfiabilité de formules booléennes. On trouve aussi des prouveurs SMT (veriT et CVC4) que nous présentons plus en détail dans cette partie.

Pour SMTCoq, les prouveurs automatiques sont vus comme des boîtes noires qui résolvent des problèmes logiques. Plus précisément, SMTCoq interagit avec un prouveur automatique en traduisant le problème dans un format reconnu par celui-ci (4.3) puis en interprétant sa réponse (4.4). Ce fonctionnement est un des intérêts de l'approche sceptique (5.1).

4.2 Prouveurs SMT

Un problème SMT est un problème de satisfiabilité de formules propositionnelles pour certaines théories prédéfinies. Expliquons succinctement le fonctionnement d'une version de l'algorithme DPLL utilisé par les prouveurs SMT [12]. Nous ne considérerons que la théorie LIA, théorie qui contient les entiers et les symboles d'addition, de soustraction et d'inégalité. On prend en exemple le problème qu'on appellera $lia5$ et qui consiste à satisfaire la formule $((x + y \leq -3 \wedge y \geq 0) \vee x \leq -3) \wedge x \geq 0$ où x et y sont des entiers.

L'algorithme commence par identifier les atomes, c'est-à-dire les sous-formules spécifiques à la théorie LIA. Dans notre exemple cela revient à poser

$$\begin{aligned} a &:= \ x + y \leq -3 \\ b &:= \ y \geq 0 \\ c &:= \ x \leq -3 \\ d &:= \ x \geq 0 \end{aligned}$$

et à chercher si la formule $((a \wedge b) \vee c) \wedge d$ est satisfiable. Cette étape peut être suivie par une transformation de Tseitin (6.2) afin d'obtenir un problème en Forme Normale Conjonctive (CNF) qui est le format de prédilection des prouveurs SAT.

Ensuite, l'algorithme DPLL répète les étapes suivantes:

- Appeler un prouveur SAT sur la conjonction des formules (dans l'exemple on commence avec une seule formule). Si ce n'est pas satisfiable, alors l'algorithme répond que le problème n'est pas satisfiable.
- Dans le cas contraire, le prouveur SAT donne une instanciation qui satisfait toutes les formules. La formule de l'exemple est satisfiable, une instanciation pourrait être: $a \wedge \neg b \wedge c \wedge d$, ce qui signifie que toutes les variables booléennes sont à $true$ sauf b qui est à $false$.
- Vérifier que l'instanciation est valide dans les théories. Si c'est le cas, l'algorithme renvoie que le problème est satisfiable.
- Sinon, rajouter la négation de l'instanciation à la liste des formules et recommencer. C'est le cas de notre exemple: il n'est pas possible, dans la théorie LIA, d'avoir à la fois $x \leq -3$ et $x \geq 0$. À la fin de cette étape, le problème est donc ramené à la satisfiabilité des deux formules suivantes:

$$\begin{aligned} &((a \wedge b) \vee c) \wedge d \\ &\neg a \vee b \vee \neg c \vee \neg d. \end{aligned}$$

Cet algorithme termine puisque qu'à chaque répétition des étapes ci-dessus une nouvelle instanciation des atomes n'est plus possible et qu'il y a un nombre fini de telles instanciations.

4.3 Format d'entrée, le langage SMT-LIB

Le langage SMT-LIB a vocation à être un format d'entrée commun à différents prouveurs SMT tels que veriT. Ce langage offre donc un cadre pour faire des comparaisons de prouveurs SMT. La définition du langage [3] donne des indications sur la sémantique que doivent avoir certaines constructions du langage.

Pour savoir si le problème *lia*5 de la section précédente est satisfiable, on peut appeler un prouveur SMT sur le fichier *lia*5.*smt*2 suivant:

```
(set-logic QF_LIA)
(declare-fun x () Int)
(declare-fun y () Int)
(assert (and (or (and (<= (+ x y) (- 3)) (>= y 0)) (<= x (- 3))) (>= x 0)))
(check-sat)
(exit)
```

On appelle assertion une formule contenue dans une ligne d'un fichier SMT-LIB commençant par *assert*: la formule $((x + y \leq -3 \wedge y \geq 0) \vee x \leq -3) \wedge x \geq 0$ est une assertion de *lia*5.*smt*2.

4.4 Format de sortie, les certificats de veriT

Lorsqu'on appelle un prouveur automatique sur un fichier SMT-LIB, si la conjonction des assertions est satisfiable, le prouveur renvoie *sat*. Si la conjonction des assertions n'est pas satisfiable, le prouveur renvoie *unsat* et fournit un fichier de certificat qui est un fichier qui explique pourquoi ce n'est pas satisfiable. Le format de ce fichier de certificat peut varier en fonction du prouveur SMT considéré.

Dans la suite de cette section nous nous intéressons aux certificats du prouveur SMT veriT. Ceux-ci utilisent des clauses, c'est-à-dire une disjonction d'une liste de formules, ce que nous noterons (f_1 ... f_n) pour la clause contenant les formules f_1, ..., f_n. La clause vide, notée (), représente l'absurde. Les certificats de veriT sont constitués d'une liste de règles de la forme:

$$id : (typ\ cl\ dep)$$

où *id* est un entier qui identifie la règle, *typ* est le type de la règle, *cl* est une clause qu'on appelle résultat de la règle et *dep* liste toutes les dépendances de la règle. Ainsi, montrer que le problème n'est pas satisfiable revient à obtenir une règle dont le résultat est la clause vide. Une règle peut utiliser le résultat d'une autre règle identifiée par *id*, dans ce cas sa liste de dépendance contient l'identifiant *id*.

4.4.1 Règle *input*

Les règles de type *input* sont des règles qui établissent les hypothèses du problème et correspondent à une assertion dans le fichier SMT-LIB. Par exemple, la règle

$$1 : (input\ (x \geq 0))$$

est utilisée dans le cas où on suppose que x est un entier positif. Cette règle (et les règles *input* en général) ne dépend pas du résultat d'autres règles.

4.4.2 Règle *resolution*

Pour manipuler ces clauses, veriT peut utiliser une règle *resolution*. La règle de résolution appliquée à deux clauses donne une nouvelle clause contenant toutes les formules de ces deux clauses sauf les formules qui ont leur négation dans l'autre clause. Par exemple, la résolution de $a \vee b$ et de $\neg a \vee \neg c$ donne $b \vee \neg c$. Grâce à cette règle, on peut montrer que le problème qui suppose l'existence d'un entier x tel que $x \geq 0$ et $\neg(x \geq 0)$ n'est pas satisfiable:

$$1 : (input\ (x \geq 0))$$
$$2 : (input\ (\neg(x \geq 0)))$$
$$3 : (resolution\ ()\ 1\ 2)$$

Une règle de résolution peut avoir plus de deux dépendances, son résultat est alors défini récursivement. Soit une règle de résolution ayant pour résultat c et pour liste de dépendance l et considérons la liste l' qui a une dépendance x en plus par rapport à l. Le résultat de la règle de résolution ayant pour dépendance la liste l' est l'application de la règle de résolution à c et au résultat de la règle identifiée par x.

4.4.3 Règles *not_implies*0 et *not_implies*1

La règle *not_implies*0 implémente la règle logique qui donne A à partir de $\neg(A \Rightarrow B)$ et *not_implies*1 implémente la règle logique qui donne $\neg B$ à partir de $\neg(A \Rightarrow B)$.

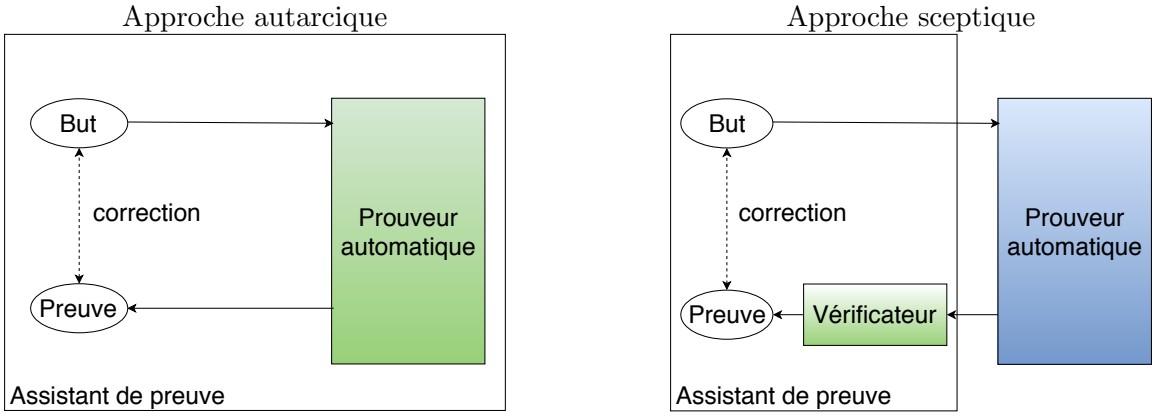
Ces règles dépendent donc du résultat d'une autre règle qui doit nécessairement être une clause de la forme $(\neg(X \Rightarrow Y))$. Les résultats des règles *not_implies*0 et *not_implies*1 sont alors respectivement (X) et $(\neg Y)$. Cette règle nous permet de montrer que la formule $\neg(A \Rightarrow A)$ n'est pas satisfiable:

$$1 : (input\ (\neg(A \Rightarrow A)))$$
$$2 : (not\_implies0\ (A)\ 1)$$
$$3 : (not\_implies1\ (\neg A)\ 1)$$
$$4 : (resolution\ ()\ 2\ 3)$$

5 Présentation de SMTCoq

5.1 SMTCoq, une interface sceptique entre Coq et les prouveurs automatiques

Pour améliorer l'automatisation de Coq et y intégrer l'utilisation de prouveurs automatiques, il y a principalement deux approches.



L'approche autarcique consiste à vérifier le code du prouveur automatique à l'intérieur de l'assistant de preuve. L'avantage de cette méthode est qu'une fois cette vérification faite, on sait que chaque appel du prouveur automatique nous renverra une preuve correcte.

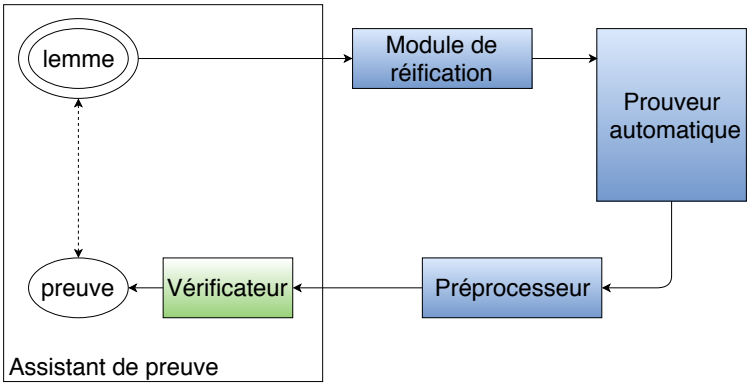
Dans l'approche sceptique, le certificat renvoyé par le prouveur automatique est vérifié à chaque appel de celui-ci. Cette approche, utilisée par SMTCoq, ne permet pas de garantir la complétude du système: certains buts valides ne sont pas démontrés, notamment lorsque le prouveur automatique renvoie un certificat erroné ou que la reconstruction de la preuve par SMTCoq n'est pas possible. En retour, cette approche ne fige pas l'implantation du prouveur automatique puisque ce n'est pas son code qui est vérifié mais sa réponse. Un autre avantage est que l'effort de certification est plus restreint: pour un certificat fixé, il faut vérifier que celui-ci correspond bien à une preuve du but.

5.2 Fonctionnement de SMTCoq

5.2.1 Amélioration de l'automatisation

Chacune des tactiques Coq définies par SMTCoq invoque un prouveur automatique différent: zChaff, CVC4 ou veriT. Ces tactiques permettent à l'utilisateur Coq de faire appel à un prouveur automatique pour résoudre le but courant et donc de profiter de l'automatisation du prouveur.

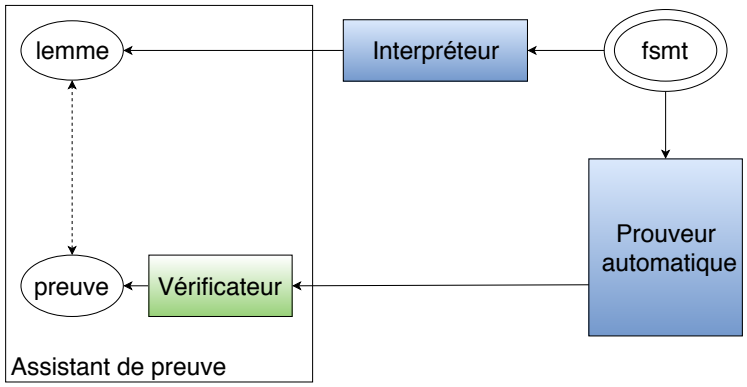
Les prouveurs automatiques fournissent un certificat uniquement dans le cas où le problème n'est pas satisfiable (4.4). Pour utiliser ce fonctionnement, SMTCoq envoie la négation du but au prouveur automatique. La preuve ne peut être reconstruite que dans le cas où la réponse est *unsat* et est accompagnée d'un fichier de certificat. Ce fichier prouve que la négation du but mène à l'absurde. Autrement dit, on obtient une preuve de la double négation du but. On fera en sorte que ce dernier soit du type des booléens. Les prédicats utilisés doivent donc être à valeurs booléennes, en particulier les domaines des variables doivent être décidables. Le but étant du type des booléens, on peut le prouver à partir de sa double négation.



La première étape est la réification, le lemme est traduit dans l'AST des formules acceptées par SMTCoq. Le prouveur automatique est appelé à partir de cet AST. En cas de succès de ce prouveur, on obtient un certificat de preuve. Il s'agit ensuite de rejouer ce certificat en Coq (voir partie 6). Il faut pour cela mettre le certificat dans une forme adaptée au vérificateur SMTCoq afin qu'il puisse l'interpréter et créer un terme de preuve. Le préprocesseur effectue une étape de *parsing* du certificat qui se souvient des sous-formules déjà rencontrées (*hash-consing*) à l'aide de tables de hachage. Il y a aussi une étape d'adaptation de ces certificats. En effet, les prouveurs automatiques peuvent parfois ne pas mentionner une étape de la preuve qu'il faut alors construire. De plus, il faut pouvoir adapter les certificats fournis par les prouveurs automatiques qui peuvent reposer sur une logique différente de celle de Coq.

5.2.2 Amélioration de la confiance

Dans la suite on s'attachera principalement à développer l'aspect automatisation de Coq mais SMTCoq propose également une commande de reconstruction d'une preuve effectuée par un prouveur automatique.



Cette commande prend en paramètre un fichier *fsmt* décrivant le lemme (typiquement écrit en SMT-LIB) et le certificat fourni par un prouveur automatique. Le lemme Coq est reconstruit à l'aide de l'interpréteur de SMTCoq et la preuve est reconstruite grâce au vérificateur. Une fois la reconstruction faite, la vérification que la preuve correspond bien au lemme est laissée à Coq.

Puisqu'un nouveau lemme Coq est créé, l'utilisateur peut vérifier que c'est bien le but qu'il voulait prouver. Ainsi, la confiance dans les prouveurs automatiques est améliorée: on peut vérifier la réponse du prouveur.

5.2.3 Cas d'application de SMTCoq

Dans les deux cas, les formules acceptées sont les formules logiques booléennes en forme prénexe. Les domaines de quantification doivent aussi être décidables. La logique est étendue avec les combinaisons des théories suivantes: arithmétique linéaire sur $\mathbb{Z}$, égalité et fonctions non-interprétées, auxquelles s'ajouteront la théorie des vecteurs de bits et la théorie des tableaux.

5.3 Utilisation de SMTCoq

5.3.1 La tactique `verit`

La nouvelle tactique `verit` définie par SMTCoq permet de résoudre automatiquement les buts dans les booléens en forme prénexe. On reprend l'exemple de la partie 4. En utilisant les fonctions booléennes de Coq, `negb` pour la négation et `>=?` et `<=?` pour les inégalités larges, le problème *lia*5 devient :

```
Lemma lia5 :
  forall x y,
    negb ( ((x+y <=? - (3)) && (y >=? 0) || (x <=? - (3))) && (x >=? 0)).
Proof.
    verit.
Qed.
```

La tactique `verit` commence par introduire les variables quantifiées universellement en tête de formule (dans l'exemple ce sont `x` et `y`) puis s'attend à ne pas avoir d'autres quantificateurs. C'est ensuite la négation de la formule qui est envoyée à veriT. La reconstruction de la preuve ne peut avoir lieu que si veriT renvoie *unsat* ainsi qu'un fichier de certificat.

5.3.2 La commande de reconstruction

La commande `Verit_Theorem` nous permet de créer un terme Coq à partir du certificat fourni par veriT appelé sur *lia*5.*smt*2. On obtient alors le terme Coq `lia5`:

```
U:---  *goals*        All (1,0)      (Coq Goals Wrap)
1 lia5
2      : negb
3          (((Smt_var_x + Smt_var_y <=? - (3)) && (Smt_var_y >=? 0)
4            || (Smt_var_x <=? - (3))) && (Smt_var_x >=? 0))
```

On notera que ce terme représente bien la négation de la formule énoncée dans le fichier SMT-LIB.

6 Transformation de certificats

Nous avons vu dans la partie précédente que c'est grâce à un procédé de réification (3.2) que l'on peut passer du but initial en Coq au prouveur automatique. Nous allons maintenant voir comment la réponse du prouveur automatique, le fichier de certificat, peut être transformée en un terme de preuve Coq.

6.1 Des certificats de toutes les couleurs

La tactique `verit` interprète le fichier de certificat *tcertif* fourni par le prouveur automatique en un terme Coq *ccertif*, ce qui permet de construire une preuve du but initial.

Format	Fichier texte	Code OCaml	Code Coq
Appellation	*tcertif*	*ocertif*	*ccertif*
Composant	*trule*	*orule*	*crule*

Cette interprétation passe par une étape intermédiaire *ocertif* écrite en OCaml. Cette étape a plusieurs avantages. En premier lieu, elle permet d'utiliser les outils de *parsing* du fichier de certificat (*ocamllex*, *ocamlyacc*). Par ailleurs, en utilisant la représentation OCaml des termes Coq, la traduction d'un *ocertif* en un *ccertif* est facilitée. Enfin, les *ocertif* sont définis dans un format facilement manipulable ce qui permet d'appliquer des adaptations (7.2.1 et 7.5), des simplifications (6.3.1) ou encore des optimisations (6.4.2).

6.2 Transformation de Tseitin et *hash-consing*

6.2.1 Motivations

La transformation de Tseitin d'une formule donne une formule équisatisfiable qui est en CNF. L'avantage de cette transformation est que sa complexité est linéaire en temps comme en espace. En comparaison, l'utilisation des lois de De Morgan pour obtenir une formule en CNF a une complexité en pire cas exponentielle. Le principe de cette transformation est d'introduire de nouvelles variables pour toutes les sous-formules, ce qui correspond au *hash-consing* qui est fait par SMTCoq [1]. Cette étape intervient au moment du *parsing*, c'est-à-dire au moment du passage d'un *tcertif* à un *ocertif*. Du point de vue de SMTCoq, ce procédé signifie un gain en espace (les sous-formules ne sont pas répétées) et un gain en temps (les comparaisons de formules deviennent des comparaisons d'entiers).

6.2.2 Fonctionnement

La transformation commence par nommer toutes les sous-formules en partant des feuilles. La nouvelle formule à satisfaire est alors la conjonction de la variable représentant toute la formule et de formules additionnelles qui garantissent que les nouvelles variables sont équivalentes aux sous-formules qu'elles représentent.

On reprend l'exemple de la partie 4. En nommant les atomes on avait obtenu la formule $((a \wedge b) \vee c) \wedge d$. On introduit également e pour la sous-formule $a \wedge b$, f pour la sous-formule $e \vee c$ et g pour la sous-formule $f \wedge d$ qui est en fait toute la formule. La formule transformée devient donc $g \wedge D_e \wedge D_f \wedge D_g$ où les D_α sont des formules qui nous assurent que la varible α est équivalente à la formule qu'elle représente. On a par exemple $D_e = (\neg a \vee \neg b \vee e) \wedge (\neg e \vee a) \wedge (\neg e \wedge b)$ qui nous assure que la varible e est équivalente à $a \wedge b$.

Dans SMTCoq, au lieu de rajouter les formules D_α, les sous-formules sont enregistrées dans un tableau: une nouvelle sous-formule peut faire référence à une sous-formule qui est à un indice précédent dans le tableau.

6.3 Le vérificateur

Le vérificateur de SMTCoq contient un type inductif *crule* qui représente les *trule* ainsi qu'une fonction `checker`. Nous allons définir ces termes Coq, voir comment ils implémentent la sémantique des certificats de veriT (4.4) et comment ils sont utilisés pour produire le terme de preuve.

6.3.1 Le type inductif *crule*

Chaque constructeur du type inductif *crule* peut représenter alternativement une *trule* ou une autre d'un ensemble de *trule*. Le constructeur `Res` ne représente que la règle *resolution* mais d'autres constructeurs peuvent représenter différentes règles en fonction de leurs paramètres. Par exemple, le constructeur `ImmBuildProj` regroupe les règles *not_implies*0 et *not_implies*1 (4.4.3) et contient aussi un paramètre entier qui vaut `0` ou `1` et qui indique quelle est la *trule* représentée. Ce regroupement de règles au fonctionnement similaire permet, dans la suite, de simplifier le traitement de ces règles.

On appellera *ccertif* la liste de toutes les *crule* du certificat.

6.3.2 La fonction récursive `checker`

Pour enregistrer le résultat des règles précédentes on utilise un tableau de clauses appelé état. On rappelle qu'une clause représente la disjonction d'une liste de formules et est notée entre parenthèses. La fonction `checker` implémente l'application des règles du certificat et modifie donc l'état à chaque nouvelle *crule* rencontrée. Plus précisément, `checker` est une fonction Coq qui est définie récursivement sur son paramètre de type *ccertif* et qui a aussi un paramètre état. Le résultat des *crule* suit le fonctionnement des *trule*. Par exemple le résultat de `ImmBuildProj 0 1` est (X) si le premier emplacement du tableau d'état est de la forme $(\neg(X \Rightarrow Y))$. À chaque appel de `checker`, une nouvelle *crule* est consommée, son résultat est utilisé pour modifier l'emplacement courant du tableau d'état.

Il n'y a pas de *crule* correspondant à la règle *input*. À la place, le tableau d'état est initialisé avec le résultat de la règle *input*. Ce fonctionnement n'est possible que lorsqu'il n'y a qu'une seule règle *input*, ce qui est le cas de la tactique `verit` (la valeur d'initialisation est la négation du but). La taille du tableau est égale au nombre de règles du *tcertif*.

Lorsque toute la liste *ccertif* a été consommée, `checker` renvoie `true` si le dernier emplacement modifié du tableau d'état contient la clause vide et `false` sinon.

6.3.3 Théorème de correction

Sur le type des formules, SMTCoq définit une fonction `interp` qui inverse la réification (3.2.3). Le vérificateur repose sur le théorème de correction qui nous assure que pour tout *ccertif* `cc` et toute formule `f`, si le tableau d'état `t` est initialisé comme décrit précédemment, alors:

```
checker cc t = true -> negb (interp f)
```

Une fois celui-ci démontré, la preuve est obtenue en l'appliquant à la réflexivité de l'égalité. En effet, si `checker cc t` est convertible à `true`, alors `checker cc t = true` est convertible à `true = true`.

6.3.4 Preuve du théorème de correction

On dit qu'une clause est valide lorsque son interprétation est prouvable. La preuve du théorème de correction repose sur le lemme `step_checker_correct` qui s'énonce ainsi: si toutes les clauses du tableau d'état sont valides alors une étape de `checker` modifie ce tableau en un tableau dont toutes les clauses sont valides. Il suffit en fait de vérifier que la nouvelle clause générée par `checker` est valide.

À partir de ce lemme on obtient une preuve du théorème de correction. Supposons que le lemme initial (la négation du but dans le cas de la tactique `verit`) soit prouvable. Initialement, le tableau d'état ne contient donc que des clauses valides et, d'après `step_checker_correct`, cette propriété est conservée par la fonction `checker`. Si de plus `checker` renvoie `true` cela veut dire qu'il y a la clause vide dans le dernier tableau d'état. L'interprétation de la clause vide nous conduit à une contradiction.

6.3.5 Exemple de l'identité

Considérons la proposition Coq suivante où `implb` est l'implication booléenne:

```
Proposition identity A : implb A A.
```

L'emplacement courant, c'est-à-dire l'emplacement du tableau d'état à modifier lors d'un appel de `checker`, initialisé à 2 et est incrémenté à chaque appel de `checker`. Ici encore on retrouve le même fonctionnement que les certificats de veriT.

Lorsqu'on applique `verit`, puisque c'est la négation du but qui est envoyée, le prouveur automatique nous renvoie le *tcertif* de 4.4.3. Celui-ci est traduit par SMTCoq en un *ccertif* `cc` et l'état est initialisé en un tableau `t` avec `cc := [ImmBuildProj 0 1; ImmBuildProj 1 1; Res 2 3]` et `t := [|nid; nid; nid; nid|]` où `nid` est la clause $(\neg(A \Rightarrow A))$.

$$
\begin{aligned}
\texttt{checker cc t} &\equiv \texttt{checker [ImmBuildProj 1 1; Res 2 3] [|nid; } (A)\texttt{; nid; nid|]} \\
&\equiv \texttt{checker [Res 2 3] [|nid; } (A)\texttt{; } (\neg A)\texttt{; nid|]} \\
&\equiv \texttt{checker [] [|nid; } (A)\texttt{; } (\neg A)\texttt{; ()|]} \\
&\equiv \texttt{true}
\end{aligned}
$$

En appliquant le théorème de correction, on obtient `negb (interp (`$\neg(A \Rightarrow A)$`))` qui est convertible à `negb (negb (implb A A))`. Cette proposition est bien équivalente à `implb A A`.

6.4 Le préprocesseur

Les avantages d'une étape intermédiaire de compilation des certificats sont multiples (6.1). Nous détaillons le format utilisé puis nous donnons une optimisation rendue possible par cette étape.

6.4.1 Le type OCaml *orule*

Les *orule* sont des enregistrements constitués, en plus du code OCaml d'une *crule*, de méta-données qui permettent leur modification. En particulier, les *orule* contiennent un champ *prev* et un champ *next*. Ainsi, un *ocertif* est une liste doublement chaînée de *orule*. Les *orule* contiennent aussi un champ *used* de type *int* qui est utilisé par la fonction *alloc*.

6.4.2 Allocation dans le tableau d'état

On remarque que dans l'exemple précédent, le tableau d'état n'a pas besoin d'être aussi grand: 2 emplacements suffisent pour ce certificat. Pour réduire l'espace mémoire utilisé, il faut calculer le nombre maximum de clauses à retenir à chaque étape de `checker`, c'est ce que fait la fonction *alloc*. La taille du tableau d'état est fixée à cette valeur. Cette fonction assigne aussi une position à chaque règle qui l'enregistre à l'aide d'un nouveau paramètre. Ce paramètre de position est utilisé pour déterminer l'emplacement courant et est noté au début des *crule*, de sorte que lorsque `checker` rencontre `ImmBuildProj 1 0 2`, le premier emplacement de l'état est modifié en (X) si le deuxième emplacement de l'état contient $\neg(X \Rightarrow Y)$. Puisque les emplacements des clauses sont modifiés, il faut aussi modifier les dépendances des règles. Enfin, `checker` prend un paramètre qui indique quelle est la position de la dernière règle, le résultat de la dernière règle n'étant pas nécessairement au dernier emplacement du tableau d'état comme précédemment. En initialisant `t` à `[|nid; nid|]` et en posant `cc := [ImmBuildProj 1 0 2; ImmBuildProj 2 1 2; Res 1 1 2]`, on a :

$$\begin{aligned}
\texttt{checker cc t 1} &\equiv \texttt{checker [ImmBuildProj 2 1 2; Res 1 1 2] [|}(A)\texttt{; nid|] 1} \\
&\equiv \texttt{checker [Res 1 1 2] [|}(A)\texttt{; }(\neg A)\texttt{|] 1} \\
&\equiv \texttt{checker [] [|}()\texttt{; }(\neg A)\texttt{|] 1} \\
&\equiv \texttt{true}
\end{aligned}$$

7 Préprocesseur pour les lemmes quantifiés

Dans cette partie on commence par donner la forme générale des certificats de veriT dans le cas des lemmes quantifiés universellement. On explique ensuite comment un certificat peut être modifié pour faciliter son encodage en un *ccertif* (partie 8).

Nous étudierons l'exemple suivant où `=?` est l'égalité à valeurs booléennes sur les entiers en Coq:

```
Lemma instance_2 f :
  (forall x, f (x+1) =? f x + 7) ->
  f 3 =? f 2 + 7.
```

7.1 Instanciation d'un lemme par veriT: la règle *forall_inst*

Lorsqu'un lemme en forme prénexe est donné par une règle *input*, veriT peut instancier ce lemme avec la règle *forall_inst*. Donnons le début du certificat simplifié obtenu en appelant `verit` sur l'exemple:

$$\begin{aligned}
&1 : (\mathit{input}\ (\forall\ x,\ f\ (x+1) = f\ x + 7)) \\
&2 : (\mathit{forall_inst}\ (\neg(\forall\ x,\ f\ (x+1) = f\ x + 7)\ (f\ (2+1) = f\ 2 + 7))) \\
&3 : (\mathit{resolution}\ (f\ (2+1) = f\ 2 + 7)\ 2\ 1)
\end{aligned}$$

On remarquera que la règle *forall_inst* ne dépend d'aucune autre règle et que l'utilisation de son résultat passe une règle *resolution*.

7.2 Lier une instanciation à un lemme

7.2.1 Lien entre une règle *forall_inst* et une règle *input*

Dans l'exemple, la dépendance de la règle *forall_inst* au lemme donné dans la règle *input* est évidente pour plusieurs raisons: le certificat est de petite taille, il y a égalité syntaxique entre le lemme et une sous-formule du résultat de la règle *forall_inst*, la règle de résolution qui utilise la règle *forall_inst* est

située juste après celle-ci et a exactement 2 dépendances. Cependant, dans le cas général, aucune de ces raisons ne reste valide. En particulier, veriT fait un renommage des variables liées qui apparaissent dans les lemmes (le certificat complet est donnée dans l'annexe B). Retrouver la dépendance demande donc, a priori, d'unifier à α-équivalence près des formules contenues dans une règle $forall\_inst$ et dans les règles *input*. Heureusement, veriT fait un *hash-consing* des formules qui apparaissent dans les *tcertif*. Cela nous permet d'enregistrer la dépendance au lemme dans la règle $forall\_inst$ au moment du *parsing* des certificats de veriT. La règle 2 devient:

$$2 : (forall\_inst\ (\neg(\forall\ x,\ f\ (x+1) = f\ x + 7)\ (f\ (2+1) = f\ 2 + 7))\ 1)$$

7.2.2 Lien entre une formule et un lemme Coq

On cherche à établir un lien entre une formule et un lemme rajouté par l'utilisateur de SMTCoq. La difficulté vient du fait que les lemmes additionnels peuvent apparaître modifiés dans les certificats de veriT. Rétablir la forme initiale est une solution trop coûteuse car il faudrait modifier la structure de toutes les formules suivantes qui en dépendent, directement ou indirectement. Pour résoudre ce problème, on utilise deux tables de hachage distinctes: *tbl* et *tbl′*.

La table *tbl* sert, comme précédemment, pour le *hash-consing* des formules. On a vu que ces formules sont ensuite interprétées pour construire le terme de preuve, il faut donc que celles-ci soient traduites fidèlement. D'autre part, les variables qui apparaissent dans les lemmes et qui sont liées par leurs quantificateurs sont à traiter séparément des autres variables. En effet, elles sont quantifiées universellement alors que les autres variables sont implicitement quantifiées existenciellement (c'est un problème de satisfiabilité). Par ailleurs ces variables n'ont pas de sens en dehors du lemme dans lequel elles sont quantifiées. Puisque ces variables ne sont pas utilisées par la suite, il serait inutile d'enregistrer un terme qui contient une de ces variables. Pour résoudre ce problème, au moment du *parsing*, on maintient, en plus de la formule qui est en train d'être traitée, une valeur booléenne qui indique si cette formule contient une variable quantifiée universellement. Lorsque ce booléen est à *true*, on n'enregistre pas la sous-formule. Par exemple dans la formule $\forall x, (f\ (x+1) = f\ x \wedge f\ 0 = 0)$, la sous-formule $f\ 0 = 0$ est enregistrée dans la table *tbl*, mais ni $f\ (x{+}1) = f\ x$ ni $f\ (x{+}1) = f\ x \wedge f\ 0 = 0$ ne le sont.

La table *tbl′* nous sert pour reconnaître des formules aux modifications de veriT près. Par exemple, une égalité peut apparaître inversée dans la règle *input* d'un lemme additionnel. À chaque fois qu'il faut enregistrer une nouvelle sous-formule de la forme $a = b$, en plus de vérifier si cette formule est déjà contenue dans la table *tbl′*, on regarde aussi si la formule $b = a$ est dans cette même table. Ainsi, on peut reconnaître efficacement des formules modulo symétrie de l'égalité.

7.3 Logique de veriT et de Coq

On a vu que veriT utilise des clauses de la forme:

$$(\neg(\forall\, x,\ P\ x)\ (P\ n))$$

Une telle clause est une tautologie pour tout prédicat P et toute valeur n en logique classique.

Cependant, dans la logique intuitionniste de Coq, ce n'est plus vrai. Une solution serait de remplacer cette clause par la clause suivante:

$$((\forall\, x,\ P\ x) \Rightarrow P\ n)$$

mais cela demande de profonds changements des *ccertif* et de leur utilisation par le vérificateur:

- il faut créer une nouvelle *crule* pour pouvoir prendre en compte toutes les règles *input* et pas seulement celle correspondant au but comme c'est fait au moment de l'initialisation du tableau d'état (6.3.2)

- il faut modifier le format des *crule* pour accepter les formules quantifiées, ce qui demande ensuite de raisonner dans Coq sur des termes à α-équivalence près.

7.4 La règle *same*

Il arrive que, à la suite de modifications des certificats, une règle devient inutile car son résultat est le même que celui d'une règle précédente. Pour la supprimer, il faut modifier tous les paramètres des règles suivantes pour qu'ils fassent référence à la première règle qui a ce même résultat. Cependant, faire cette modification à chaque fois que l'on veut supprimer une règle a une complexité quadratique en la taille des certificats. Pour remédier à ce problème, on introduit une nouvelle *orule same* qui est une règle qui dépend d'une seule autre règle. Dans un premier temps les règles à supprimer sont remplacées par des règles *same* et un dictionnaire est créé afin d'associer à chaque règle *same* la règle non-*same* lui correspondant (associer une règle *same* à sa dépendance retombe dans un comportement quadratique). Ensuite les règles *same* sont supprimées et les paramètres des règles restantes sont modifiés en suivant le dictionnaire.

7.5 Modifier le résultat de la règle $forall\_inst$

Pour ces raisons, il est préférable de modifier les règles de la forme:

$$id : (forall\_inst\ (\neg lemma\ lemma\_inst)\ id\_lemma)$$

où *lemma* est un des lemmes rajoutés par l'utilisateur et *lemma_inst* est une instance de ce même lemme en une règle:

$$id : (forall\_inst\ (lemma\_inst)\ id\_lemma)$$

Il faut aussi modifier les règles suivantes qui dépendent du résultat de cette règle. On fait l'hypothèse supplémentaire qu'une règle $forall\_inst$ dépendant d'un lemme l ne sera utilisée dans la suite du certificat que dans une règle de résolution ayant aussi une dépendance à l. On se ramène donc à modifier seulement les règles de résolution, ce que l'on fait ainsi: si une règle de résolution a pour liste de dépendance *dep*, on trouve toutes les règles $forall\_inst$ de cette liste et on enlève leurs dépendances de *dep*. Cette modification a un complexité en pire cas linéaire dans la taille des certificats. Dans le cas où il ne reste plus qu'une seule dépendance, la règle *resolution* devient une règle *same*. En reprenant l'exemple de la section précédente, on obtient:

$$\begin{aligned}
&1 : (input\ (\forall\ x,\ f\ (x+1) = f\ x+7))\\
&2 : (forall_inst\ (f\ (2+1) = f\ 2+7)\ 1)\\
&3 : (same\ (f\ (2+1) = f\ 2+7)\ 2)
\end{aligned}$$

7.6 Une application: les règles *input*

Cette modification est suffisamment générale pour permettre le traitement des règles *input*. On est effectivement amené à traiter ces règles lorsque l'utilisateur transmet des lemmes non-quantifiés à la tactique `verit`. Le résultat d'une règle *input* d'un lemme non-quantifié peut être directement utilisé par une des règles suivantes, ces lemmes n'ayant pas à être instanciés. Ainsi, les règles *input* qui ne sont ni des lemmes quantifiés ni la négation du but initial sont transformées en des règles *forall_inst* sans nécessiter de modifications des règles suivantes.

8 Vérificateur pour les lemmes quantifiés

Pour traiter le cas des lemmes quantifiés du point de vue de Coq, on a besoin de rajouter un constructeur au type inductif *crule*. On verra comment modifier le vérificateur en conséquence afin de rétablir la preuve de correction et de préserver l'automatisation de SMTCoq.

8.1 La *crule* `Forallinst`

Dans cette section, on se propose d'encoder les règles d'instanciation en laissant le langage des termes Coq utilisés dans les *crule* inchangé. Cet encodage ne s'applique qu'aux lemmes en forme prénexe; dans le cas général il faudrait nécessairement étendre le langage des termes. En particulier, on ne rajoutera pas de quantificateurs au langage, ce qui nous obligerait à étendre la fonction d'interprétation et à adapter les preuves la concernant. La légèreté de cet encodage est rendue possible par les modifications de la règle d'instanciation de la partie précédente. On rajoute tout de même un constructeur au type *crule* mais cela nous demande seulement de compléter la preuve de `step_checker_correct` (6.3.4) correspondant à cette nouvelle règle.

Le nouveau constructeur s'écrit:

```
Forallinst p lemma plemma lemma_inst pinstanc
```

où `p` est le paramètre de position (6.4.2), `lemma` est le lemme dont on a identifié la dépendance (7.2), `plemma` est la preuve de ce lemme, `lemma_inst` est l'instance et `pinstanc` est un élément du type `lemma -> interp lemma_inst`, c'est-à-dire une preuve que le lemme implique l'instance.

Une étape de la fonction `checker` correspondant à une telle *crule* modifie la clause `p` de l'état en la clause `lemma_inst`. On peut facilement prouver que `lemma_inst` est valide, et même indépendemment des clauses contenues par l'état. En effet, il suffit d'appliquer `pinstanc` à `plemma`. On a donc rétabli la preuve du théorème de correction.

Le problème est en fait déplacé puisqu'il faut maintenant, pour construire une *crule*, donner un élément du type `lemma -> interp lemma_inst` qu'on appellera preuve d'instanciation. Cette preuve est donnée à l'aide d'une coupure (tactique `assert` en Coq), ce qui nous permet de donner les preuves d'instanciation dans un deuxième temps.

8.2 Preuves d'instanciation

La structure d'une instance peut différer de celle du lemme pour plusieurs raisons. Pour chacune de ces raisons nous verrons comment obtenir tout de même une preuve d'instanciation. Grâce à ces résultats on peut définir une tactique qui permet de trouver automatiquement n'importe quelle preuve d'instanciation (voir annexe C).

8.2.1 Preuve automatique d'une instanciation

Le lemme Coq suivant correspond directement à l'instanciation du lemme donné:

```
Lemma instance_c P (c : Z):
  (forall x, P x) ->
  P c.
```

De tels but peuvent être prouvés automatiquement par la tactique `auto`. Cependant ce n'est plus le cas lorsque le but est légèrement modifié. Par exemple, le lemme suivant ne peut pas être prouvé directement par `auto`:

```
Lemma instance_3 f (c : Z):
  (forall x, f x =? f c) ->
  f c =? f 3.
```

Il faut donc remettre le but dans une forme qui correspond à celle du lemme. Dans le dernier cas il s'agit de réécrire le lemme `Z.eqb_sym` avant d'appliquer la tactique `auto`.

8.2.2 Différence liée à la symétrie de l'égalité

Les lemmes qui apparaissent dans les certificats de veriT peuvent avoir subi des modifications (7.2.2), ces modifications se retrouvent dans les règles d'instanciation des lemmes. À partir du lemme Coq `forall x, f x =? f c` où `c` est une constante entière, le certificat de veriT peut contenir la règle:

$$id1 : (input\ (\forall\ x,\ f\ c = f\ x))$$

Une règle instance correspondant à ce lemme peut donc être:

$$id2 : (forall\_inst\ (\neg(\forall\ x,\ f\ c = f\ x)\ (f\ c = f\ 3)))$$

On a vu qu'on peut reconnaître que cette instance correspond au lemme donné par l'utilisateur (7.2). On est donc ramené au problème `instance_3`. Pour résoudre ce problème, on commence par remarquer que la plupart du temps, si une des égalités est inversée, alors toutes les égalités sont inversées. On écrit une tactique qui inverse toutes les égalités et pour chaque cas on essaye de résoudre le but en utilisant cette tactique et en ne l'utilisant pas.

8.2.3 Différence *impl_split*

Lorsque qu'un lemme est de la forme:

$$id1 : (input\ (\forall\ x,\ f\ x \Rightarrow b))$$

la *trule* $forall\_inst$ peut être:

$$id2 : (forall\_inst\ (\neg(\forall\ x,\ f\ x \Rightarrow b)\ (\neg(f\ c) \vee b)))$$

au lieu de la *trule* attendue:

$$id2 : (forall\_inst\ (\neg(\forall\ x,\ f\ x \Rightarrow b)\ (f\ c \Rightarrow b)))$$

Cette différence se retrouve directement dans la preuve d'instanciation associée et peut être résolue en ajoutant un lemme à la base de lemmes `Resolve`. La tactique `auto`, qui utilise cette base de lemme, trouvera automatiquement la preuve pour ce type de différence entre lemme et instance.

```
Lemma impl_split a b:
  implb a b -> orb (negb a) b.
Proof. intro. destruct a; destruct b; trivial. Qed.

Hint Resolve impl_split.
```

9 Utilisation de la tactique `verit` avec des lemmes

On définit une nouvelle tactique : `verit_base`. Cette tactique peut prendre en argument des preuves de lemmes à rajouter. Par exemple

```
Lemma fSS3 f k:
  (forall x, f (x + 1) =? f x + k) ->
  forall x, f (x + 2) =? f x + 2 * k.
Proof. intro f_k_linear. verit_base f_k_linear.
```

Cette tactique laisse à l'utilisateur les 2 preuves d'instanciation. On peut terminer la preuve grâce à la tactique `vauto` définie d'après les résultats de la partie précédente :

```
Proof. intro f_k_linear. verit_base f_k_linear. vauto. vauto. Qed
```

Plus généralement, lorsqu'on veut rajouter les lemmes prouvés par `H1`, ..., `Hn`, on utilisera :

```
verit_base H1 ... Hn; vauto.
```

On définit la tactique `verit` comme `verit_base; vauto` ce qui nous permet de conserver le comportement de cette tactique par rapport à la version antérieure de SMTCoq lorsque l'on n'utilise pas de lemmes supplémentaires. On définit également une liste qui sera toujours ajoutée à `verit_base`. Initialement cette liste est vide mais la commande `Add_lemmas H1 ... Hn` la concatène avec `H1`, ...,`Hn`. Pour finir la commande `Clear_lemmas` réinitialise la liste. D'autres exemples sont donnés dans les fichiers *Example.v* et *Tests_verit.v* du dépôt Git `github.com/smtcoq/smtcoq`.

10 Travaux connexes et conclusion

Ce stage s'inscrit dans une approche automatique de la démonstration dans un assistant de preuve et porte plus précisément sur SMTCoq.
D'autres outils sont disponibles dans ce cadre, à commencer par Coqhammer [6] qui est aussi un *plugin* pour Coq qui utilise des prouveurs automatiques. À la différence de SMTCoq, lors de la reconstruction de la preuve, Coqhammer liste les lemmes qui apparaissent dans le certificat et n'utilise rien d'autre que cette liste du certificat. Ainsi, il y a une recherche qui est faite par des tactiques en Coq et qui permet de retrouver la preuve. Cette méthode est plus robuste que celle de SMTCoq vis-à-vis des prouveurs automatiques mais demande de chercher à nouveau la preuve et est donc plus coûteuse. De la même manière, pour trouver les preuves d'instanciation (8.2), la recherche a été faite en Coq. Cette méthode s'est avérée utile pour passer outre certaines différences entre instances et lemmes. Un autre point de divergence est que SMTCoq propose aussi une commande de vérification, ce qui permet d'améliorer la confiance accordée aux prouveurs automatiques (5.2.2).
Un autre outil qui a une interface sceptique avec des prouveurs automatiques est Sledgehammer [5] qui est développé pour l'assistant de preuve Isabelle. L'idée d'un sélectionneur de lemmes qui apprend quels lemmes il est judicieux d'envoyer aux prouveurs automatiques [4] vient de Sledgehammer.
On peut aussi mentionner des approches autarciques de la démonstration automatique comme la tactique `unsat` [11] qui utilise la réflexion calculatoire ou encore la tactique `metis` en Isabelle.

En résumé, ce stage a permis d'améliorer l'expressivité de SMTCoq tout en restant dans un cadre qui assure la correction de la méthode. Cette amélioration de l'expressivité a été confirmée par de nouveaux tests. Enfin, ce stage ouvre de nouvelles pistes de travail (1.5) comme l'amélioration de l'efficacité, la mise en place d'un sélectionneur de lemmes, l'extension des cas d'application ou encore l'utilisation de SMTCoq pour la certification Why3.

Remerciements

À la suite de mon stage au LRI d'Orsay, j'adresse tous mes remerciements

À mes professeurs du Master Parisien de Recherche en Informatique qui m'ont donné la possibilité de trouver un stage correspondant tout à fait à mes attentes,

À mes maîtres de stage, madame Chantal Keller, maître de Conférences de l'Université Paris-Sud et monsieur Valentin Blot, post-doctorant de l'Université Paris-Sud, pour leur disponibilité et leur enthousiasme,
À madame Chantal Keller qui a bien voulu m'initier au sujet de sa recherche et m'a donné de précieux conseils à différents niveaux,
À monsieur Valentin Blot de son aide fréquente, si utile pour la bonne compréhension du sujet, et des discussions passionnantes que nous avons eues qui m'ont ouvert d'autres perspectives,

À toute l'équipe VALS qui m'a accueilli et en particulier à monsieur Claude Marché, madame Chantal Keller et monsieur Andrei Paskevich, que je serai très heureux de retrouver en tant qu'encadrants de ma thèse.

Bibliographie

[1] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of sat/smt solvers to coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to SAT verification. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 83–98, 2010.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[4] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for isabelle/hol. *J. Autom. Reasoning*, 57(3):219–244, 2016.

[5] Jasmin Christian Blanchette and Lawrence C. Paulson. *Hammering Away, A User's Guide to Sledgehammer for Isabelle/HOL*, 2017.

[6] Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 125–128, 2013.

[8] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2016.

[9] Chantal Keller. *A Matter of Trust: Skeptical Communication Between Coq and External Provers. (Question de confiance : communication sceptique entre Coq et des prouveurs externes)*. PhD thesis, École Polytechnique, Palaiseau, France, 2013.

[10] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[11] Stéphane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. PhD thesis, University of Paris-Sud, Orsay, France, 2011.

[12] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[13] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

Annexe A: Formules booléennes conjonctives

On donne ici deux définitions, la preuve de leur équivalence, la mise en forme de peigne des formules conjonctives, la correction de cette fonction, la réification d'un booléen et une tactique qui utilise la réflexion calculatoire.

```
Require Import Bool.

Inductive AndTree :=
  Bool (b : bool)
| And (left: AndTree) (right: AndTree).

Inductive Interp : AndTree -> bool -> Prop :=
  InterpBool b :
    Interp (Bool b) b
| InterpAnd b1 b2 b3 t1 t2 :
    Interp t1 b1 -> Interp t2 b2 -> b1 && b2 = b3 ->
    Interp (And t1 t2) b3.

Definition t :=
  And (And (Bool true) (Bool false)) (And (Bool true) (Bool true)).

Lemma Interp_t_false : Interp t false.
Proof.
  eapply InterpAnd ; [
    eapply InterpAnd ; [ apply InterpBool | apply InterpBool | reflexivity ]
  | eapply InterpAnd ; [ apply InterpBool | apply InterpBool | reflexivity ]
  | reflexivity ].
Qed.

Fixpoint interp (t : AndTree) :=
  match t with
    Bool n => n
  | And t1 t2 => interp t1 && interp t2
  end.

Proposition Interp_eq_interp t b :
  Interp t b <-> interp t = b.
Proof.
  revert b. induction t as [a | t1 IHt1 t2 IHt2]; simpl; intro b.
  -split; intro H.
   +now inversion H.
   +rewrite H. apply InterpBool.
  -split; intro H.
   +inversion H.
    apply IHt1 in H2; rewrite H2.
    now apply IHt2 in H3; rewrite H3.
   +eapply InterpAnd. now apply IHt1.
    now apply IHt2. assumption.
Qed.

Lemma interp_t_false : interp t = false.
Proof.
  reflexivity.
Qed.
```

```
Fixpoint append t1 t2 :=
  match t1 with
    Bool n => And t1 t2
  | And t11 t12 => And t11 (append t12 t2)
  end.

Fixpoint peigne (t : AndTree) :=
  match t with
    Bool n => t
  | And t1 t2 => let pt1 := peigne t1 in
                 let pt2 := peigne t2 in
                 append pt1 pt2
  end.

Inductive eqt : AndTree -> AndTree -> Prop :=
  refl t: eqt t t
| sym t1 t2: eqt t1 t2 -> eqt t2 t1
| assoc t1 t2 t3: eqt (And t1 (And t2 t3)) (And (And t1 t2) t3)
| congru ta1 ta2 tb1 tb2: eqt ta1 tb1 -> eqt ta2 tb2 ->
                          eqt (And ta1 ta2) (And tb1 tb2)
| trans t1 t2 t3: eqt t1 t2 -> eqt t2 t3 -> eqt t1 t3.

Lemma eqt_correct t1 t2:
  eqt t1 t2 -> interp t1 = interp t2.
Proof.
  intro eq12. induction eq12; simpl.
  -reflexivity.
  -auto.
  -apply andb_assoc.
  -now rewrite IHeq12_1, IHeq12_2.
  -now rewrite IHeq12_1.
Qed.

Lemma append_eqt t1 t2:
  eqt (append t1 t2) (And t1 t2).
Proof.
  revert t2. induction t1; intro t2; simpl.
  -apply refl.
  -eapply trans. eapply congru. apply refl. apply IHt1_2.
   apply assoc.
Qed.

Lemma peigne_eqt t:
  eqt (peigne t) t.
Proof.
  induction t; simpl.
  -apply refl.
  -eapply trans. apply append_eqt. now apply congru.
Qed.

Lemma peigne_correct t:
  interp (peigne t) = interp t.
Proof.
  apply eqt_correct. now apply peigne_eqt.
Qed.
```

```
Ltac reify A :=
  match A with
  | andb ?X ?Y => let rx := reify X in
                  let ry := reify Y in
                  constr:(And rx ry)
  | ?X => constr:(Bool X)
  end.

Ltac peignify :=
  match goal with
  | [ |- ?A = ?B] =>
    let a := reify A in
    let b := reify B in
    change A with (interp a);
    change B with (interp b);
    rewrite <- (peigne_correct a);
    rewrite <- (peigne_correct b);
    simpl
  end.

Lemma peigne4 b1 b2 b3 b4:
  (b1 && b2) && (b3 && b4) = b1 && ((b2 && b3) && b4).
Proof.
  peignify. reflexivity.
Qed.
```

Annexe B: Fichier SMT-LIB et certificat veriT d'une instanciation

Lorsqu'on appelle veriT sur le fichier SMT-LIB *simple_instance.smt2* suivant:

```
(set-logic UFLIA)
(declare-fun f (Int) Int)
(assert (not (= (f 3) (+ (f 2) 7))))
(assert (forall ( (x Int) ) (= (f (+ x 1)) (+ (f x) 7))))
(check-sat)
(exit)
```

on obtient *unsat* et la preuve dans un fichier contenant:

```
1:(input ((not #1:(= #2:(f 3) #3:(+ #4:(f 2) 7)))))
2:(input (#5:(forall ( (x Int) ) #6:(= #7:(f #8:(+ x 1)) #9:(+ #10:(f x) 7)))))
3:(tmp_betared (#11:(forall ( (@vr0 Int) ) #12:(= #13:(f #14:(+ @vr0 1)) #15:(+ #16:(f
    @vr0) 7)))) 2)
4:(tmp_qnt_tidy (#17:(forall ( (@vr1 Int) ) #18:(= #19:(f #20:(+ @vr1 1)) #21:(+ #22:(f
    @vr1) 7)))) 3)
5:(forall_inst (#23:(or (not #17) #24:(= #3 #25:(f #26:(+ 2 1))))))
6:(or ((not #17) #24) 5)
7:(resolution (#24) 6 4)
8:(eq_transitive ((not #27:(= #2 #25)) (not #24) #1))
9:(eq_congruent ((not #28:(= 3 #26)) #27))
10:(resolution ((not #24) #1 (not #28)) 8 9)
11:(resolution ((not #28)) 10 1 7)
12:(la_disequality (#29:(or #28 (not #30:(<= 3 #26)) (not #31:(<= #26 3)))))
13:(or (#28 (not #30) (not #31)) 12)
14:(resolution ((not #30) (not #31)) 13 11)
```

```
15:(la_generic (#31))
16:(resolution ((not #30)) 14 15)
17:(la_generic (#30))
18:(resolution () 17 16)
```

Annexe C: Automatisation des preuves d'instanciation

```
Require Import SMTCoq Bool.
Open Scope Z_scope.

(* verit silently transforms an <implb a b> into a <or (not a) b> when
 instantiating a quantified theorem with <implb> *)
Lemma impl_split a b:
  implb a b = true -> orb (negb a) b = true.
Proof.
  intro H.
  destruct a; destruct b; trivial.
(* alternatively we could do <now verit_base H.> but it forces us to have veriT
   installed when we compile SMTCoq. *)
Qed.

Hint Resolve impl_split.

(* verit silently transforms an <implb (a || b) c> into a <or (not a) c>
   or into a <or (not b) c> when instantiating such a quantified theorem *)
Lemma impl_or_split_right a b c:
  implb (a || b) c -> negb b || c.
Proof.
  intro H.
  destruct a; destruct c; intuition.
Qed.

Lemma impl_or_split_left a b c:
  implb (a || b) c -> negb a || c.
Proof.
  intro H.
  destruct a; destruct c; intuition.
Qed.

(* verit considers equality modulo its symmetry, so we have to recover the
   right direction in the instances of the theorems *)
Definition hidden_eq a b := a =? b.
Ltac all_rew :=
  repeat match goal with
         | [ |- context [ ?A =? ?B]] =>
           change (A =? B) with (hidden_eq A B)
         end;
  repeat match goal with
         | [ |- context [ hidden_eq ?A ?B] ] =>
           replace (hidden_eq A B) with (B =? A);
           [ | now rewrite Z.eqb_sym]
         end.

(* An automatic tactic that takes into account all those transformations *)
```

```
Ltac vauto :=
  try (let H := fresh "H" in
       intro H; try (all_rew; apply H);
       match goal with
       | [ |- is_true (negb ?A || ?B) ] =>
         try (eapply impl_or_split_right; apply H);
         eapply impl_or_split_left; apply H
       end;
       apply H);
  auto.

Ltac verit :=
  verit_base; vauto.
```