Tactique automatique en Coq

Quentin Garchery

Lundi 26 novembre 2018

LRI, Paris-Saclay
Univ. Paris-Sud / Inria

Chantal Keller
LRI, Univ. Paris-Sud

Valentin Blot
IRIF, Univ. Paris 7

Vue d'ensemble

Les logiciels formels permettent de vérifier des propriétés mathématiques. On considère deux types de logiciels formels :

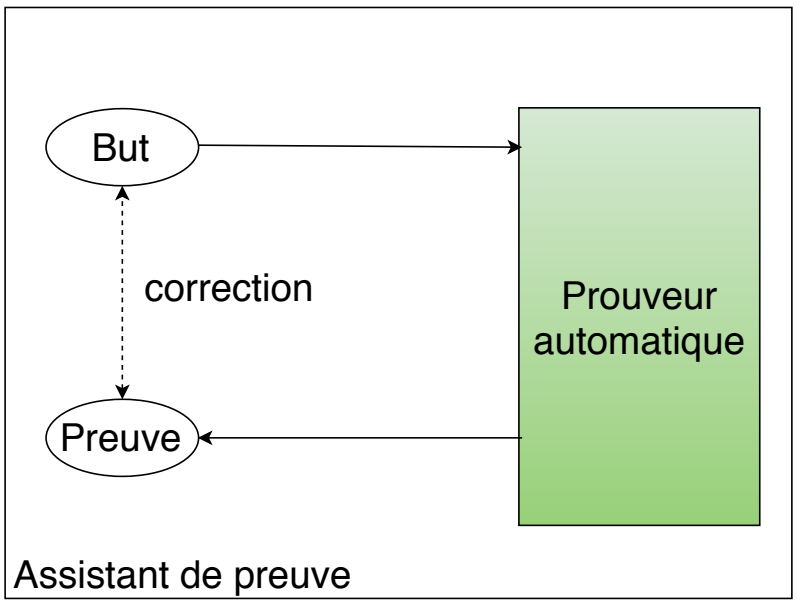
- Assistants de preuve, expressifs et interactifs
- Prouveurs automatiques de type SMT, moins expressifs mais l'effort de certification est plus restreint

Cadre : communication entre ces deux types de logiciels pour bénéficier de leurs avantages respectifs.

Automatisation dans les assistants de preuve

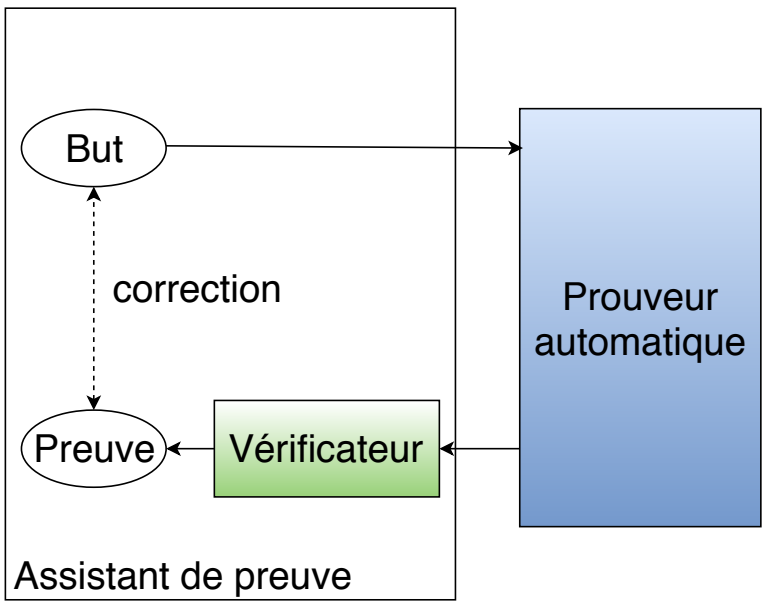
Deux approches à l'intégration d'un prouveur automatique dans un assistant de preuve :

Approche autarcique



Vérifier le code du prouveur automatique.

Approche sceptique



Vérifier la réponse du prouveur automatique.

Présentation de SMTCoq

SMTCoq est une interface entre Coq et différents prouveurs automatiques (veriT, CVC4) qui est actuellement développée par Chantal Keller et Valentin Blot en collaboration avec l'Université de l'Iowa.

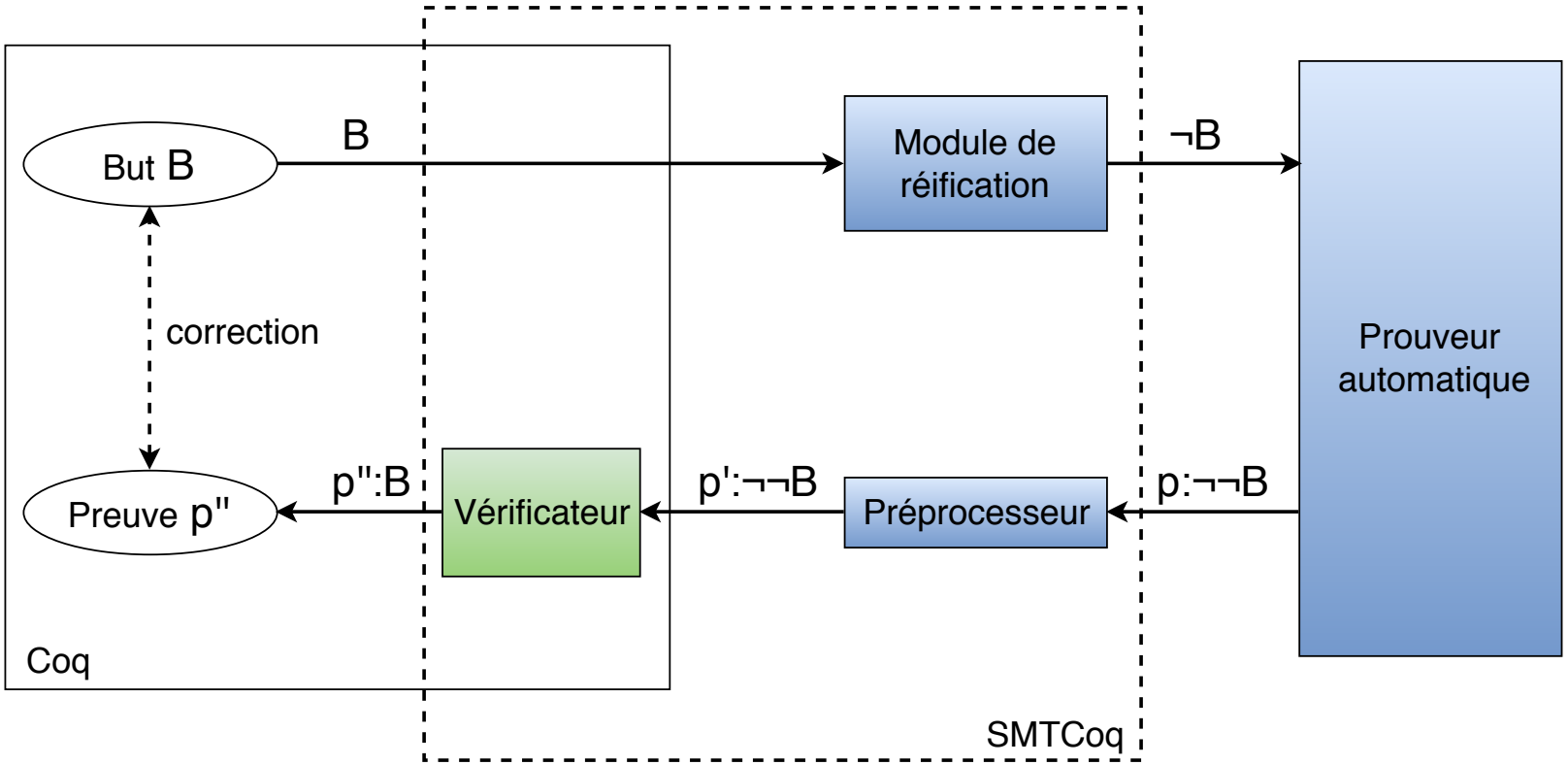
But : amélioration de l'automatisation de Coq et de la confiance dans les prouveurs automatiques.

Permet de prouver automatiquement le théorème suivant :

$$\forall x \forall y \forall f . x \neq y + 1 \vee f(y) = f(x - 1)$$

Fragment supporté : logique propositionnelle, arithmétique linéaire sur $\mathbb{Z}$, égalité et fonctions non-interprétées, théorie des tableaux, théorie des vecteurs de bit, variables quantifiées universellement en tête de formule.

SMTCoq en détail : approche sceptique



En transmettant la négation du but au prouveur automatique, on obtient un certificat prouvant l'absurde. Dans les cas d'application de SMTCoq, la double négation du but implique le but.

Définitions et preuves en Coq

La structure de données des formules de SMTCoq est définie par un type inductif :

```
Inductive formula :=
| Bool (b : bool)
| And (f1 : formula) (f2 : formula).
```

L'interprétation calcule la valeur d'une formule de SMTCoq :

```
Fixpoint interp t := match t with
| Bool b => b
| And f1 f2 => interp f1 && interp f2.
```

Un lemme prouvable :

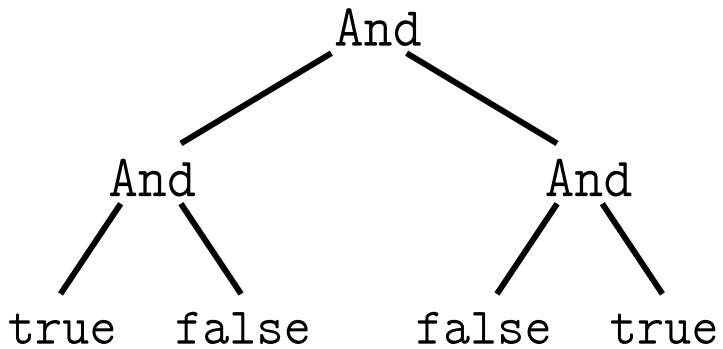
```
Lemma andproj1 f1 f2 :
 interp (And f1 f2) = true -> interp f1 = true.
```

Réification

La réification du terme `(true && false) && (false && true)` est donnée par

```
And (And (Bool true) (Bool false))
    (And (Bool false) (Bool true))
```

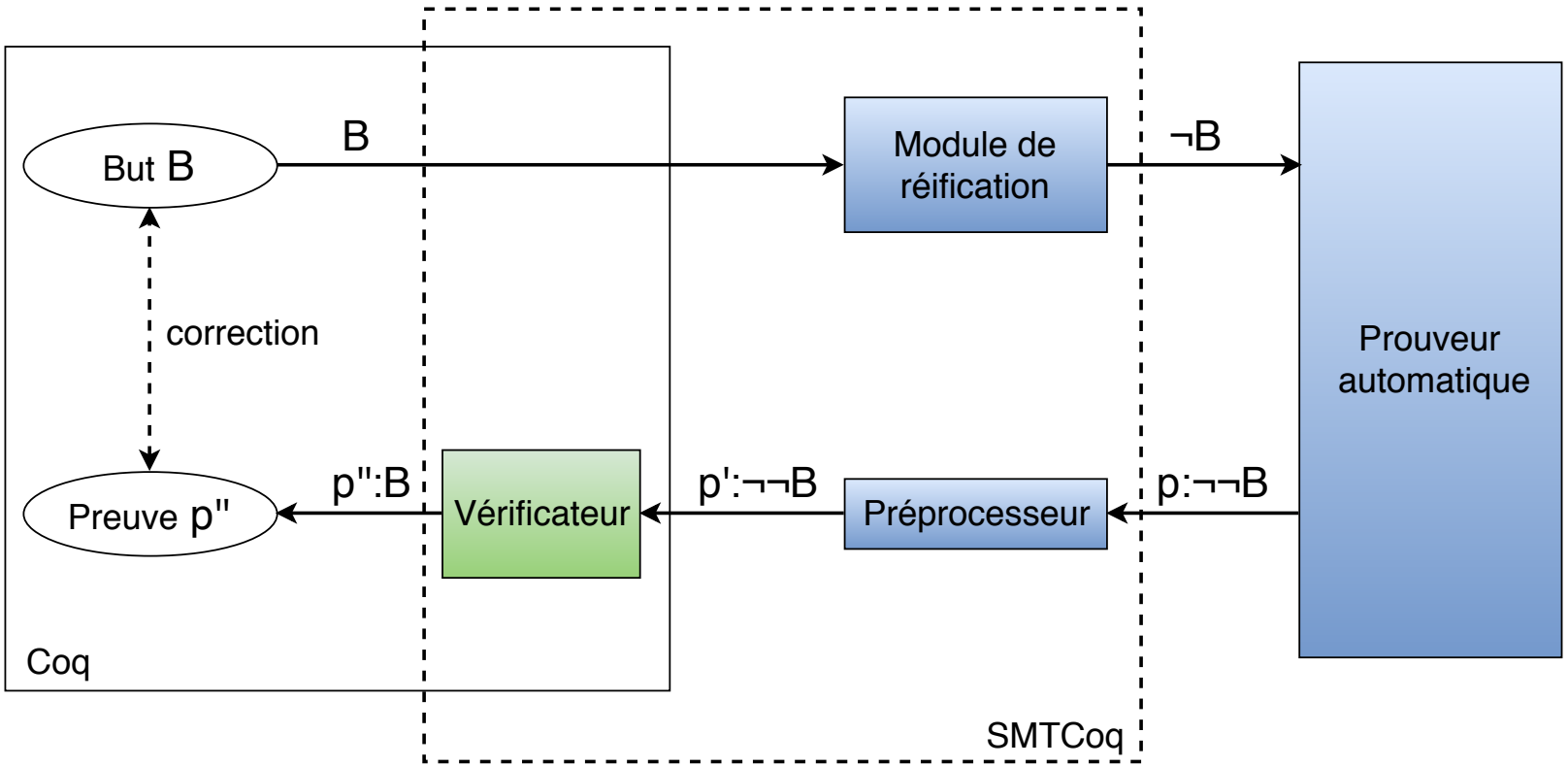
qui représente l'arbre suivant :



Réification : rendre explicite la structure d'un terme en l'exprimant dans une structure de données.

Dans le cas de SMTCoq, cela permet entre autres l'écriture du problème dans un fichier servant d'entrée aux prouveurs.

SMTCoq en détail

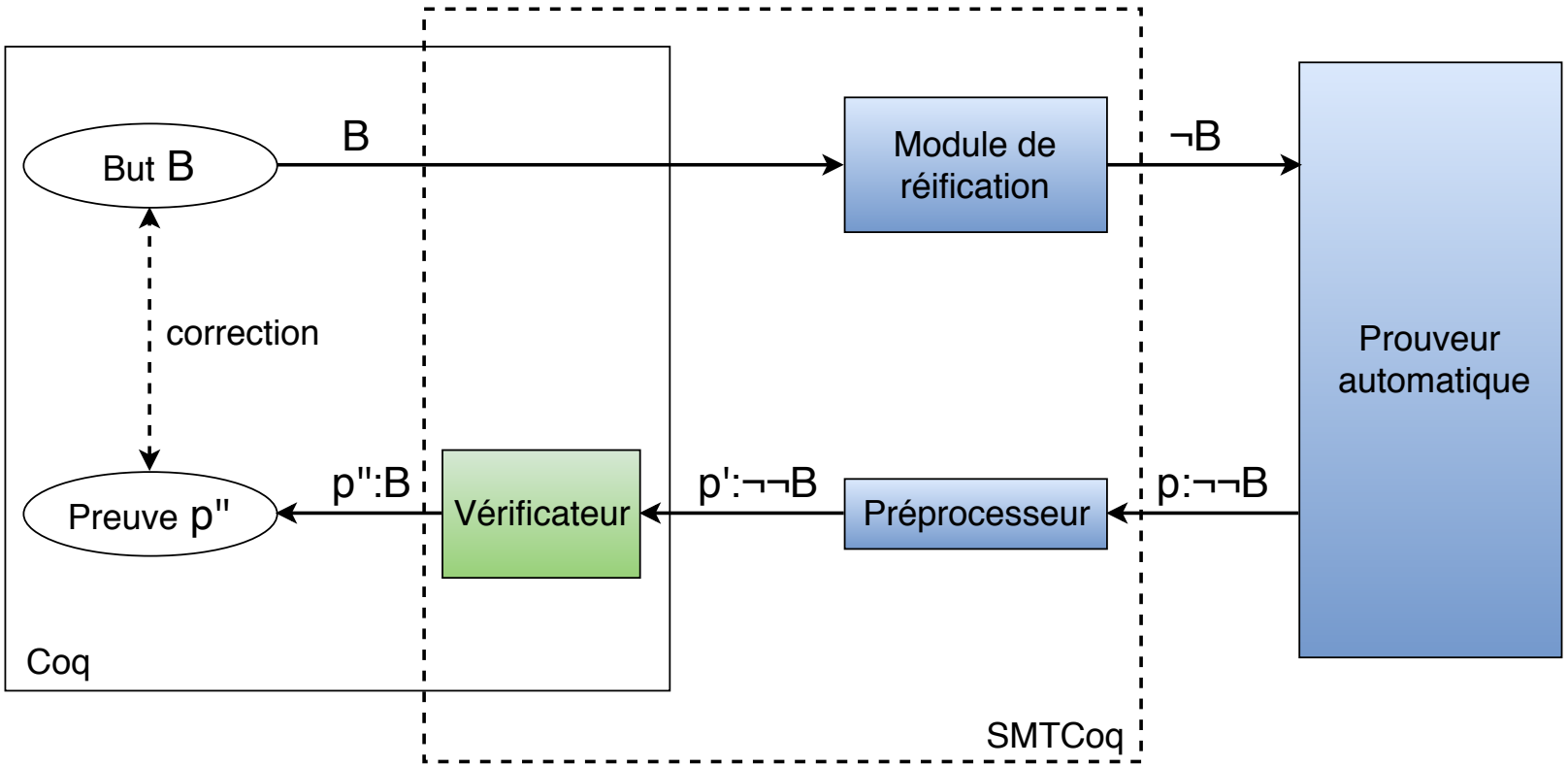


Préprocesseur

Le préprocesseur de SMTCoq remplit plusieurs objectifs :

- **parsing** des fichiers de certificats en un type de données OCaml
- **adaptations** des certificats, celles-ci sont par exemple nécessaires lorsque la logique du prouveur automatique diffère de celle de Coq
- **simplifications** en regroupant les règles du certificat au fonctionnement similaire
- **optimisations** des certificats, par exemple en élaguant les règles redondantes

SMTCoq en détail



Vérificateur : les règles et les certificats

Le certificat Coq est obtenu à partir du fichier de certificat fourni par les prouveurs automatiques.

```
Inductive rule :=
| AndProj (pos_prem : int) (ind_proj : int)
| Resolution (pos_param : int list).
```

Une règle `Andproj prem i` permet de projeter la conjonction en indice `prem` sur sa composante `i`.

Vérificateur : la fonction *checker*

La fonction `checker` prend en argument un certificat et une formule et maintient une liste de formules appelé *état*.

Posons :

```
certif := [Andproj 0 2; Andproj 0 1; Resolution [2;1]]
in := And (Bool x) (Neg (Bool x))
```

L'état est alors initialisé à `[in]`.

Après la première règle : `[in; Neg (Bool x)]`.

Vérificateur : la fonction *checker*

La fonction `checker` prend en argument un certificat et une formule et maintient une liste de formules appelé *état*.

Posons :

```
certif := [Andproj 0 2; Andproj 0 1; Resolution [2;1]]
in := And (Bool x) (Neg (Bool x))
```

L'état est alors initialisé à `[in]`.

Après la première règle : `[in; Neg (Bool x)]`.

Après la deuxième règle : `[in; Neg (Bool x); Bool x]`.

Vérificateur : la fonction *checker*

La fonction `checker` prend en argument un certificat et une formule et maintient une liste de formules appelé *état*.

Posons :

```
certif := [Andproj 0 2; Andproj 0 1; Resolution [2;1]]
in := And (Bool x) (Neg (Bool x))
```

L'état est alors initialisé à `[in]`.

Après la première règle : `[in; Neg (Bool x)]`.

Après la deuxième règle : `[in; Neg (Bool x); Bool x]`.

Enfin, la troisième règle transforme l'état en :

```
[in; Neg (Bool x); Bool x; Bool false]
```

L'état final contient `Bool false`, on a donc :

```
checker certif in = true
```

Vérificateur : le théorème de correction

Théorème de correction

Si `checker certif in = true` pour un certificat `certif` et une formule d'entrée `in` alors `in` n'est pas satisfiable.

Point clé de la preuve : le lemme de correction d'une étape.
Ce lemme nous assure que chaque règle ajoute à l'état une nouvelle formule vraie dans l'état courant.

Pour rétablir la preuve de correction lorsqu'on ajoute un nouveau type de règle, il suffit de compléter ce lemme pour ce nouveau type.

$$\dfrac{\dfrac{\dfrac{}{\texttt{true = true}}\,\textit{refl}}{\texttt{checker certif in = true}}\,\textit{conv} \qquad \dfrac{\textit{correction d'une étape}}{\textit{correction}}}{\texttt{interp in = false}}\,\textit{MP}$$

L'hypothèse du théorème de correction est obtenue en lançant le calcul de la fonction `checker`, c'est la réflexion calculatoire.

Amélioration de l'expressivité

SMTCoq ne sait pas montrer que :

$$(\forall h.homme(h) \Rightarrow mortel(h)) \wedge$$

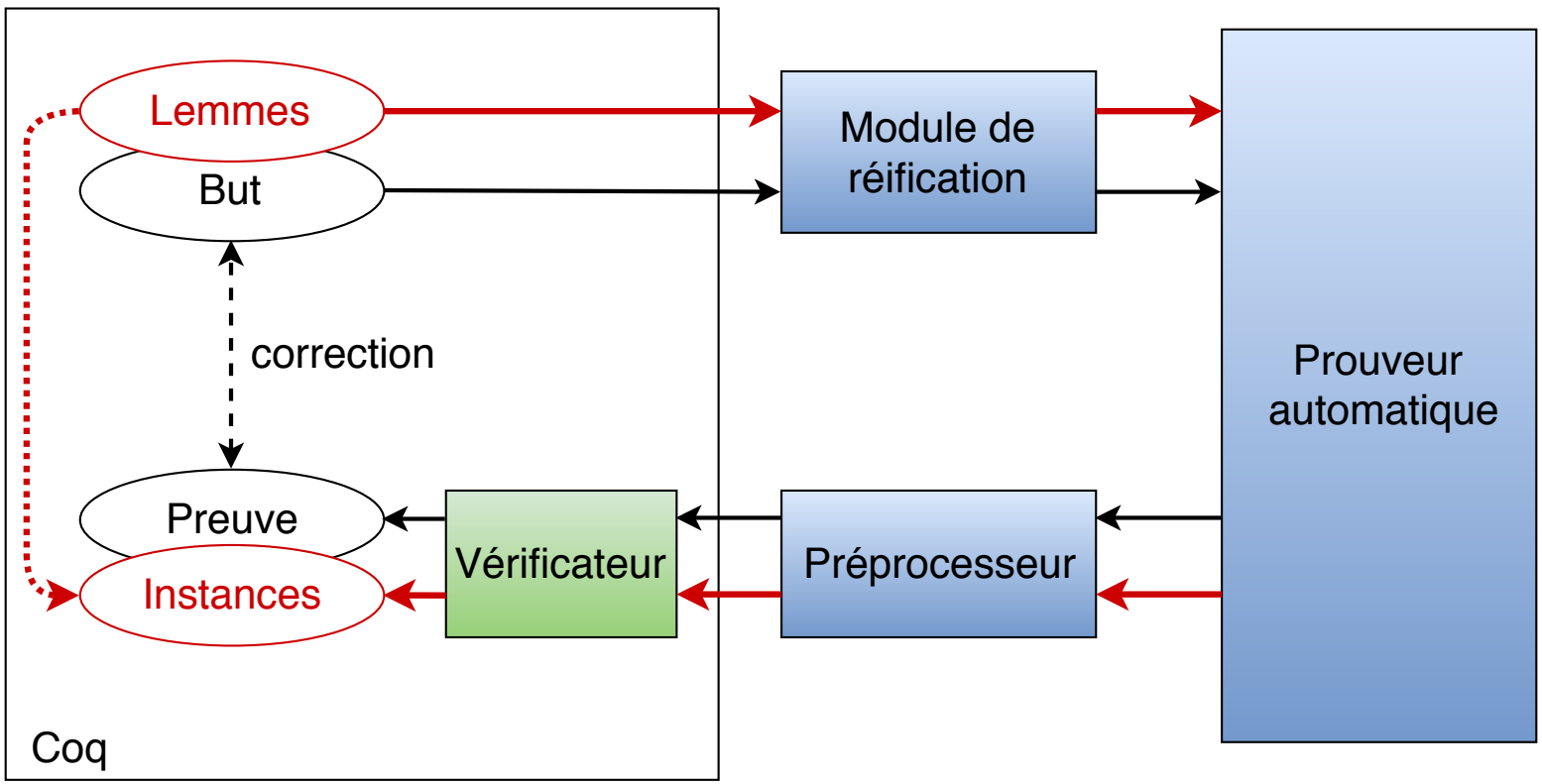
$$homme(Socrate)$$

implique :

$$mortel(Socrate)$$

Objectifs : permettre l'ajout de lemmes quantifiés au contexte et tenir compte de leurs instanciations dans le certificat. L'accent est mis sur la légèreté de cette extension.

Ajout de lemmes au contexte



La règle Inst

```
Inductive rule :=
| ...
| Inst (lemma : Prop) (plemma : lemma)
  (inst : formula) (pinst : lemma -> interp inst = true)
```

Avantage : le type des formules est inchangé. On utilise la modularité de SMTCoq lors de l'ajout de nouvelles règles.

Cas d'application : l'instance ne doit pas contenir de quantificateurs, on se restreint au cas des lemmes avec quantificateurs universels en tête de formule.

La construction d'une telle règle demande la recherche du lemme correspondant à l'instance.

Application à veriT : certificats

Si le problème n'est pas satisfiable, renvoie un fichier de certificat qui explique pourquoi c'est le cas.

Dans le certificat de veriT suivant, le résultat de la dernière règle est la clause vide () qui représente l'absurde.

$0 : (\mathit{input}\ (x \wedge \neg x))$; hypothèse $x \wedge \neg x$

$1 : (\mathit{andproj}\ (\neg x)\ 0\ 2)$; de $x \wedge \neg x$, on obtient $\neg x$

$2 : (\mathit{andproj}\ (x)\ 0\ 1)$; de $x \wedge \neg x$, on obtient x

$3 : (\mathit{resolution}\ ()\ 2\ 1)$; de x et $\neg x$, on obtient $\bot$

Application à veriT : la règle `forall_inst`

La règle `forall_inst` permet d'instancier les lemmes quantifiés donnés en en entrée dans les certificats de veriT.

Forme générale de l'utilisation de la règle `forall_inst` dans les certificats de veriT :

$0 : (\mathit{input}\ (\forall x, P\ x)))$; `lemme quantifié`

$1 : (\mathit{forall\_inst}\ (\neg(\forall x, P\ x) \vee (P\ c)))$; `instanciation`

$2 : (\mathit{resolution}\ (P\ c)\ 0\ 1)$; instance

Application à veriT : traduction de certificats

Le préprocesseur transforme les règles

$0 : (\mathit{input}\ (\forall x, P\ x)))$

$1 : (\mathit{forall\_inst}\ (\neg(\forall x, P\ x) \vee (P\ c)))$

$2 : (\mathit{resolution}\ (P\ c)\ 0\ 1)$

en : `Inst _ _ (P c) _`

où les _ sont à compléter (recherche du lemme correspondant, et preuve de l'instanciation).

$\Longrightarrow$ Suppression de la forme logique de l'implication et des quantificateurs.

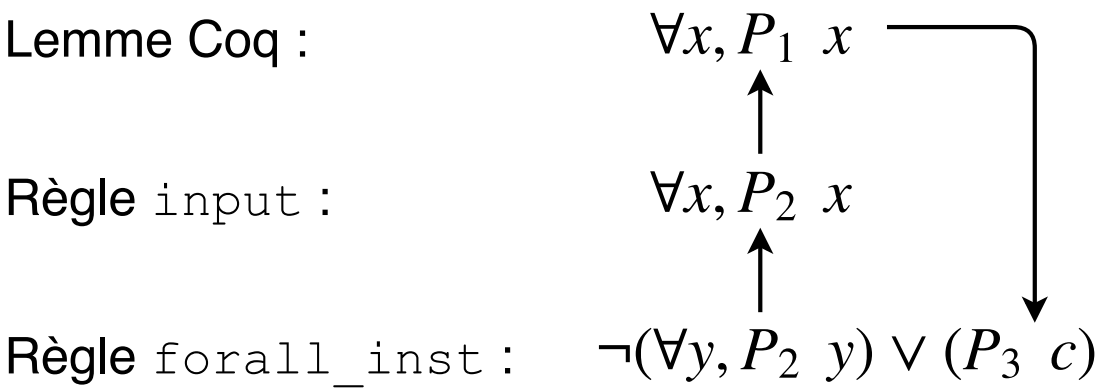
Dans l'exemple, la règle de résolution devient redondante.

Application à veriT : autres difficultés rencontrées

Parsing : il faut reconnaître les quantificateurs, problème des variables liées.

$$\forall x, (f\ (x+1) = f\ x \wedge f\ 0 = 0)$$

Reconnaître à quel lemme correspond une instance et montrer que le lemme reconnu implique l'instance.



Implémentation

La commande `Add_lemmas` ajoute les lemmes donnés en argument au contexte. Si aucun lemme n'est donné, la tactique `verit` garde le même fonctionnement que dans la version initiale de SMTCoq.

Un exemple en Coq :

```
Axiom hommes_mortels : forall h, homme h --> mortel h.
Axiom homme_Socrate : homme Socrate.
Add_lemmas hommes_mortels homme_Socrate.

Lemma mortel_Socrate : mortel Socrate.
Proof.
  verit.
Qed.
```

$\longrightarrow$ `smtcoq.github.io`

Contributions

Résultats : amélioration de l'expressivité de SMTCoq confirmée par des tests en théorie des groupes, sur une théorie formalisant les listes d'entiers ...

```
Inst (lemma : Prop) (plemma : lemma)
     (inst : formula) (pinst : lemma -> interp inst = true)
```

Nouvelle règle, d'un autre type : elle utilise à la fois un encodage profond et un encodage superficiel des formules.

Conclusion

Perspectives : étendre les cas d'application de SMTCoq.

- application à CVC4
- logique du premier ordre

Objectif : faciliter et généraliser l'utilisation de SMTCoq dans les projets développés en Coq.