Démonstration automatique en Coq

Quentin Garchery

Mardi 4 septembre 2018

Stage au LRI, Paris-Saclay
Université Paris-Sud / CNRS
sous la direction de

Chantal Keller
Maître de Conférences
Université Paris-Sud

Valentin Blot
Post-doctorant
Université Paris-Sud

Vue d'ensemble

Les logiciels formels permettent de vérifier des propriétés mathématiques. Les méthodes formelles s'étendent à la preuve de programmes.

On considère deux types de logiciels formel :

- Assistants de preuve (Coq, Isabelle)
- Prouveurs automatiques (zChaff, veriT, CVC4)

Objet du stage : SMTCoq, une interface entre assistant de preuve et prouveurs automatiques.

Assistants de preuve

Les assistants de preuves sont des logiciels formels

- expressifs,
- interactifs,
- modulaires,
- sûrs.

```
Variable R : real_model.
Theorem four_color (m : map R) :
  simple_map m -> map_colorable 4 m.
Proof.
  exact (compactness_extension four_color_finite).
Qed.
```

Prouveurs automatiques

Les prouveurs automatiques sont des logiciels formels

- automatiques,
- à la logique plus restreinte,
- qui peuvent ne pas renvoyer de résultat,
- auxquels on accorde moins de confiance.

```
(set-logic QF_UF)                    ; logique utilisée
(declare-fun x () Bool)     ; déclaration du booléen x
(assert (and x (not x)))                ; hypothèse
(check-sat)
(exit)
```

Certificats de veriT

Si le problème n'est pas satisfiable, certains prouveurs automatiques renvoient 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.

$1 : (\mathit{input}\ (x \wedge \neg x))$; hypothèse $x \wedge \neg x$

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

$3 : (\mathit{andproj}\ (\neg x)\ 1\ 1)$; de $x \wedge \neg x$, on obtient $\neg x$

$4 : (\mathit{resolution}\ ()\ 3\ 2)$; de x et $\neg x$, on obtient $()$

Présentation de SMTCoq

SMTCoq est une interface entre Coq et différents prouveurs automatiques qui est actuellement développée par Chantal Keller en collaboration avec l'Université de l'Iowa.

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

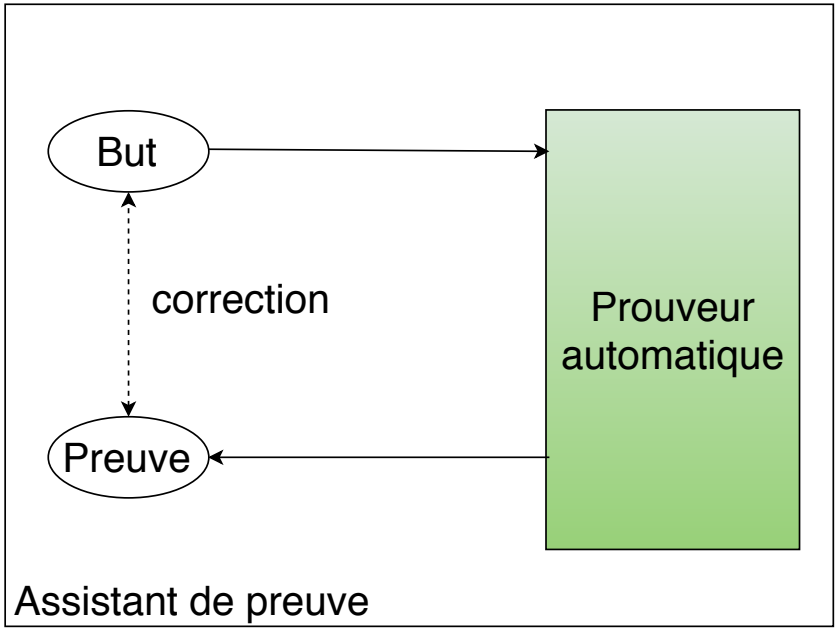
Permet de prouver automatiquement le théorème suivant :

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

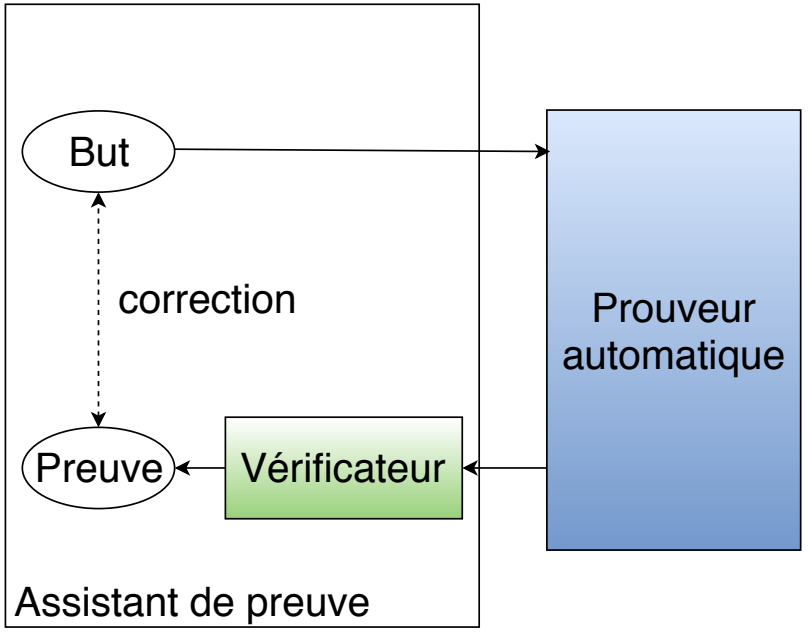
Fragment supporté : logique propositionnelle, arithmétique linéaire sur $\mathbb{Z}$, égalité et fonctions non-interprétées, variables quantifiées universellement en tête de formule.

Automatisation dans les assistants de preuve

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

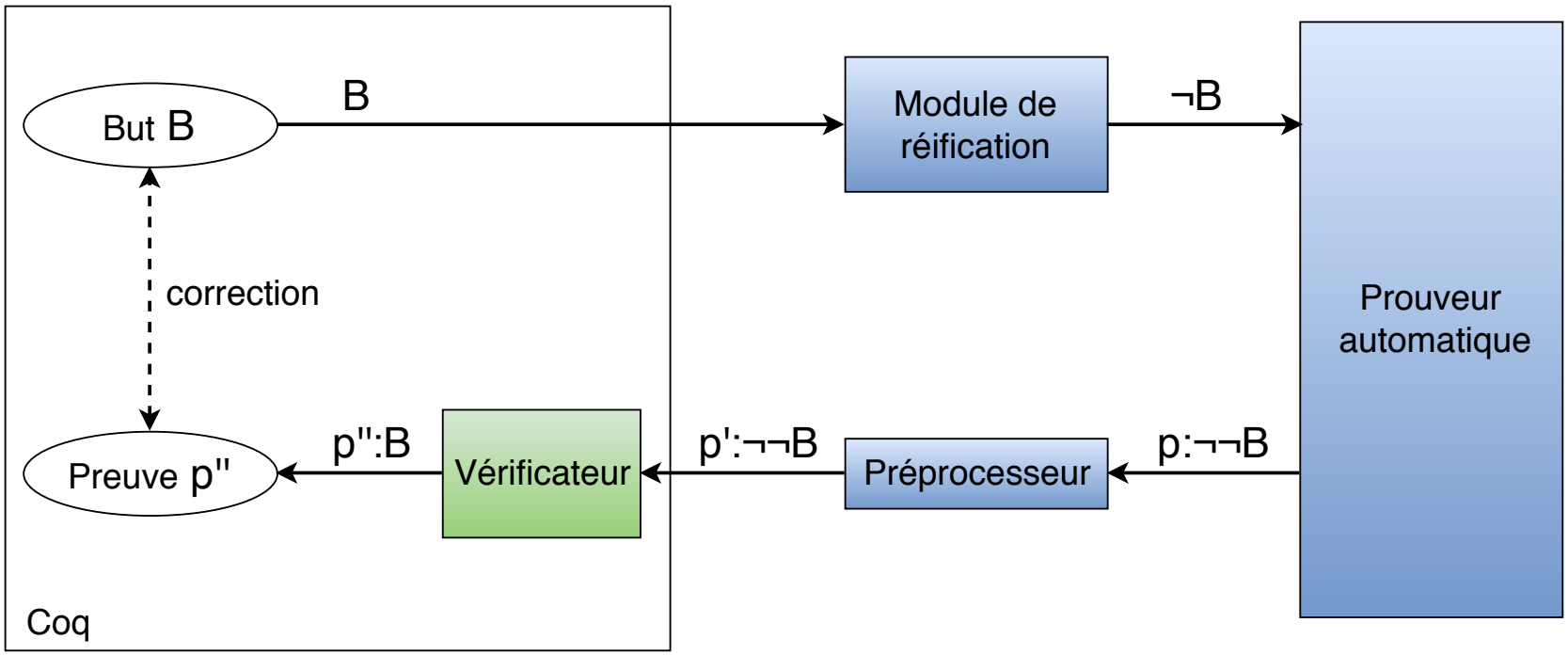


Approche autarcique : vérifier le code du prouveur automatique.



Approche sceptique : vérifier la réponse du prouveur automatique à chaque appel de celui-ci.

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.

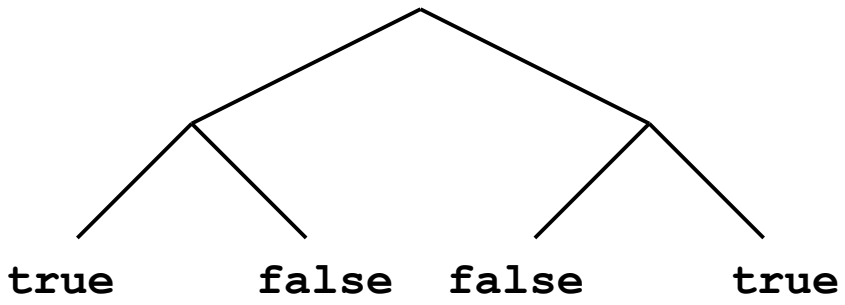
Réification

Réification : rendre explicite la structure d'un terme.

Par exemple, en notant || la fonction Coq qui implémente la disjonction booléenne, la structure du terme

`(true || false) || (false || true)`

est donnée par l'arbre suivant :



Cet arbre peut être représenté dans une structure de données.

La structure du terme réifié est alors directement manipulable, on peut l'écrire dans un format adapté aux prouveurs automatiques.

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

Vérificateur : la fonction *checker*

On reprend le certificat de veriT donné en exemple :

$$1 : (\mathit{input}\ (x \wedge \neg x))$$

En notant $\mathit{inp} := x \wedge \neg x$, l'état est initialisé à $[\mathit{inp}; \mathit{inp}; \mathit{inp}; \mathit{inp}]$.

Vérificateur : la fonction *checker*

On reprend le certificat de veriT donné en exemple :

$$1 : (\mathit{input}\ (x \wedge \neg x))$$
$$2 : (\mathit{andproj}\ (x)\ 1\ 0)$$

En notant $\mathit{inp} := x \wedge \neg x$, l'état est initialisé à $[\mathit{inp}; \mathit{inp}; \mathit{inp}; \mathit{inp}]$.

La règle 2 tranforme l'état en $[\mathit{inp}; x; \mathit{inp}; \mathit{inp}]$.

Vérificateur : la fonction *checker*

On reprend le certificat de veriT donné en exemple :

$$1 : (\mathit{input}\ (x \wedge \neg x))$$
$$2 : (\mathit{andproj}\ (x)\ 1\ 0)$$
$$3 : (\mathit{andproj}\ (\neg x)\ 1\ 1)$$

En notant $\mathit{inp} := x \wedge \neg x$, l'état est initialisé à $[\mathit{inp}; \mathit{inp}; \mathit{inp}; \mathit{inp}]$.

La règle 2 tranforme l'état en $[\mathit{inp}; x; \mathit{inp}; \mathit{inp}]$.

La règle 3 tranforme l'état en $[\mathit{inp}; x; \neg x; \mathit{inp}]$.

Vérificateur : la fonction *checker*

On reprend le certificat de veriT donné en exemple :

$$1 : (\mathit{input}\ (x \wedge \neg x))$$
$$2 : (\mathit{andproj}\ (x)\ 1\ 0)$$
$$3 : (\mathit{andproj}\ (\neg x)\ 1\ 1)$$
$$4 : (\mathit{resolution}\ ()\ 3\ 2)$$

En notant $\mathit{inp} := x \wedge \neg x$, l'état est initialisé à $[\mathit{inp}; \mathit{inp}; \mathit{inp}; \mathit{inp}]$.

La règle 2 tranforme l'état en $[\mathit{inp}; x; \mathit{inp}; \mathit{inp}]$.

La règle 3 tranforme l'état en $[\mathit{inp}; x; \neg x; \mathit{inp}]$.

La règle 4 tranforme l'état en $[\mathit{inp}; x; \neg x; ()]$.

Le résultat de *checker* sur ce certificat et avec l'état initialisé comme ci-dessus est donc *true*.

Vérificateur : le théorème de correction

Théorème de correction

Si `checker certif = true` où `certif` est un certificat ayant pour `input` la formule f alors f est fausse.

Point clé de la preuve : le lemme de correction d'une étape.
Ce lemme nous assure que chaque règle modifie un emplacement de l'état en une 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.

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 :

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

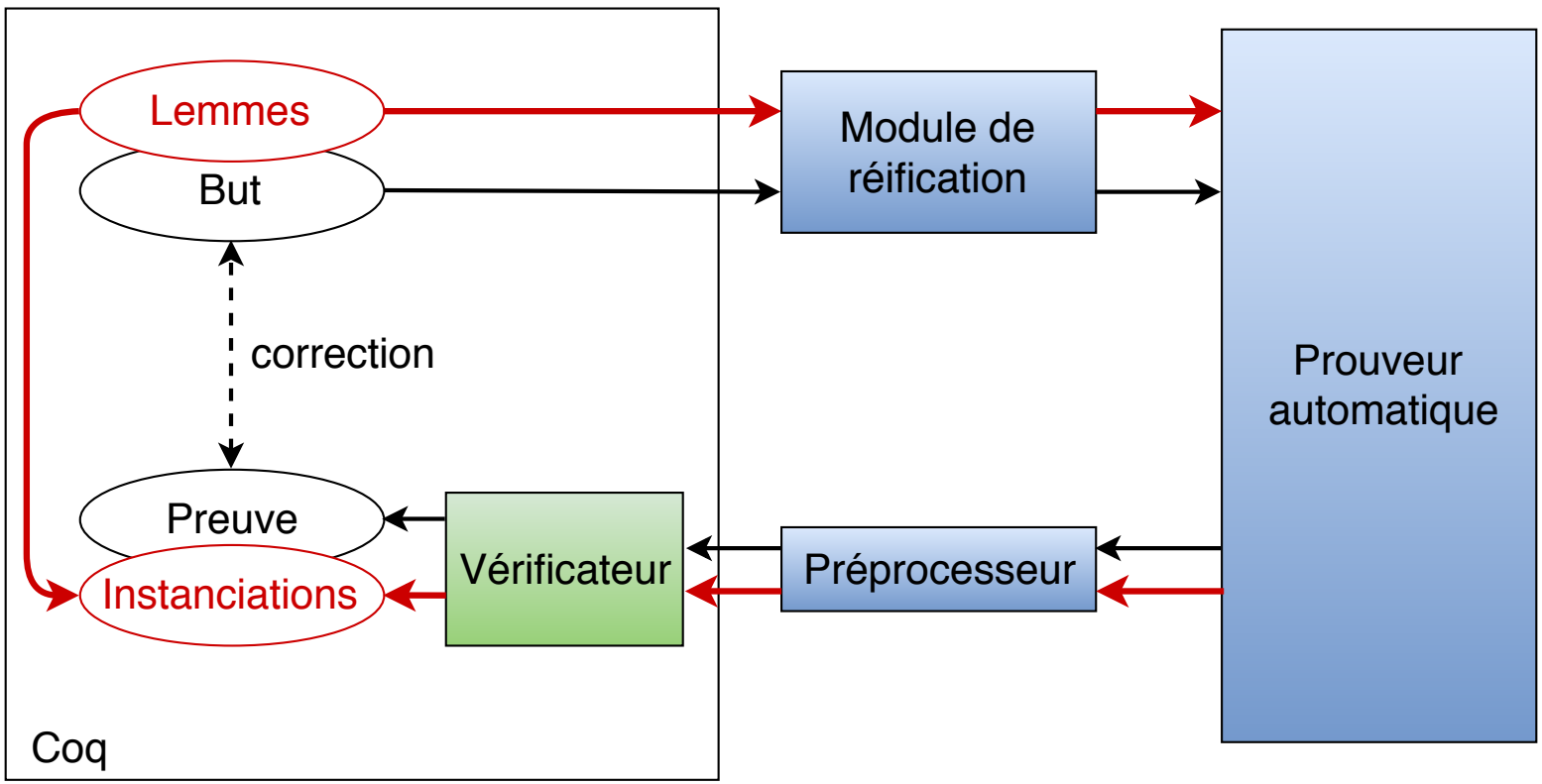
$$homme(Socrate)$$

implique :

$$mortel(Socrate)$$

Objectifs du stage : permettre l'ajout de lemmes quantifiés au contexte et tenir compte de leurs instanciations dans le certificat.

Ajout de lemmes au contexte



Règle `forall_inst`

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

Forme générale des certificats avec règle `forall_inst` :

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

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

3 : $(\mathit{resolution}\ (P\ c)\ 1\ 2)$; utilisation

Utilisation directe de la règle `forall_inst`

Une règle `forall_inst` fait intervenir des formules de la forme :

$$\neg(\forall x, P\ x) \vee (P\ c)$$

L'utilisation directe de cette règle par le vérificateur pose plusieurs problèmes :

- logique classique vs logique intuitionniste
- traitement des règle `input`
- traitement des quantificateurs dans les règles

Modifications de la règle `forall_inst`

Le préprocesseur transforme la formule

$$\neg(\forall x, P\ x) \vee (P\ c)$$

en

$$P\ c$$

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

Dans l'exemple, la règle `resolution` devient redondante.

Autres difficultés rencontrées

Parsing : il faut reconnaître les quantificateurs. Problème des variables liées et *hash consing*.

Reconnaître à quel lemme correspond une instance, veriT peut faire des modifications des instances.

Montrer que le lemme reconnu implique l'instance, ici aussi les modifications de veriT posent problème.

Résultats du stage

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.
```

⟶ `github.com/QGarchery/smtcoq-1`