SMTCoq : automatisation expressive et extensible

Journées Francophones des Langages Applicatifs

Vendredi 1er février 2019

Valentin Blot

Amina Bousalem

Quentin Garchery

Chantal Keller

Vue d'ensemble

Logiciels de vérification formelle :

- Assistants de preuve (Coq, Isabelle)
  - noyau de vérification
  - expressifs
  - interactifs
- Prouveurs automatiques de type SMT
  - moins expressifs
  - autonomes

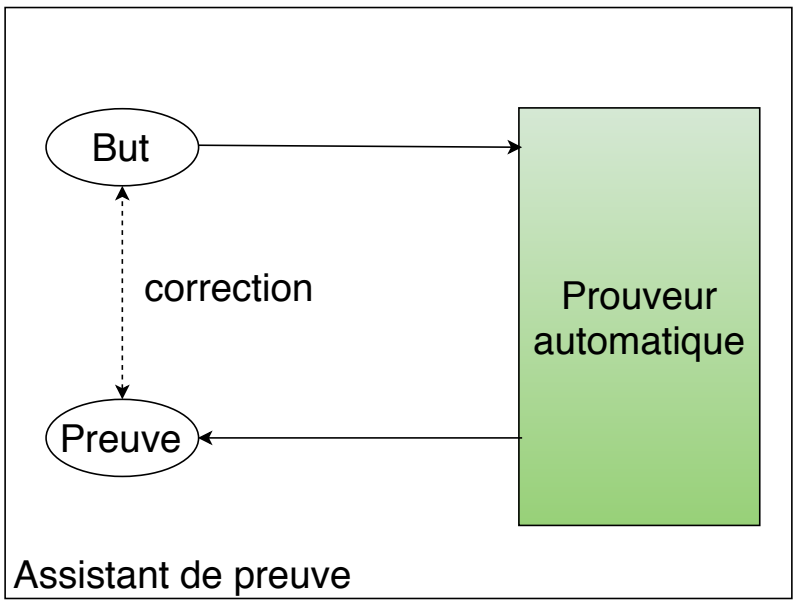
Problème : besoin d'automatisation dans Coq

Solution : communication entre ces deux types de logiciels

Automatisation dans les assistants de preuve

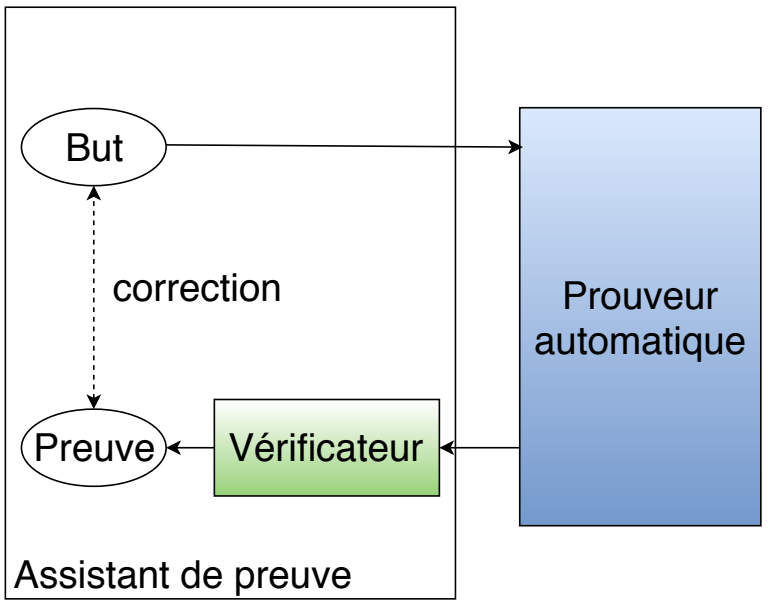
Intégration d'un prouveur dans un assistant de preuve :

Approche autarcique



Vérifier le code du prouveur

Approche sceptique



Vérifier la réponse du prouveur

Plan

SMTCoq : approche sceptique de l'automatisation dans Coq

Présentation de SMTCoq

SMTCoq : interface sceptique à différents prouveurs automatiques

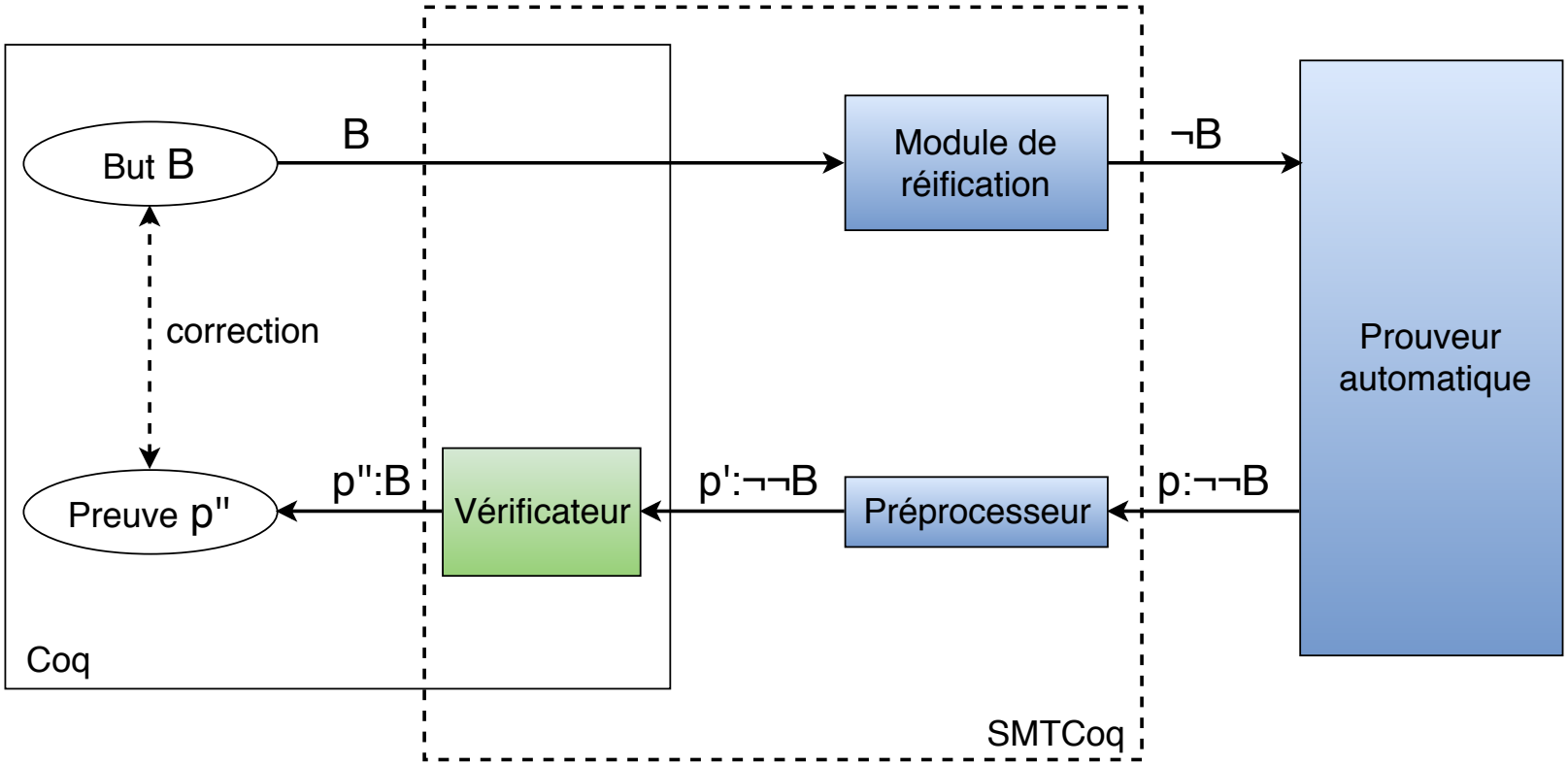
Objectifs de SMTCoq :

- amélioration de l'automatisation de Coq
- amélioration de la confiance dans les prouveurs automatiques

```
Goal :  forall f x y z, x <> y + 1 \/ f y = f (x+1).
Proof.  smt.  Qed.
```

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 but.

SMTCoq en détail : approche sceptique



Conversion initiale du but dans les booléens

De la négation du but, on obtient un certificat prouvant faux

Formules de SMTCoq

Type algébrique simplifié des formules de SMTCoq :

```
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
 | ...
```

Vérificateur : les règles et les certificats

Règles logiques permettant de déduire de nouvelles formules :

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

`Andproj prem i` projette sur `i` la conjonction donnée en `prem`

Exemple : si `prem` $\rightsquigarrow$ `And A B` alors `Andproj prem 2` justifie `B`

```
Definition certif := list rule.
```

Certificat vérifié $\Rightarrow$ problème non satisfiable

Vérificateur : la fonction *checker*

Type de `checker` : `certif -> formula -> bool`
La fonction `checker` maintient une liste de formules appelée *état*

Posons :

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

Vérificateur : la fonction *checker*

Type de `checker` : `certif -> formula -> bool`
La fonction `checker` maintient une liste de formules appelée *état*

Posons :

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

Calcul de `checker cert in`

L'état est initialisé à `[in]`

Vérificateur : la fonction *checker*

Type de `checker` : `certif -> formula -> bool`
La fonction `checker` maintient une liste de formules appelée *état*

Posons :

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

Calcul de `checker cert in`

L'état est initialisé à `[in]`

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

Vérificateur : la fonction *checker*

Type de `checker` : `certif -> formula -> bool`
La fonction `checker` maintient une liste de formules appelée *état*

Posons :

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

Calcul de `checker cert in`

L'état est 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*

Type de `checker` : `certif -> formula -> bool`
La fonction `checker` maintient une liste de formules appelée *état*

Posons :

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

Calcul de `checker cert in`

L'état est 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]`

Vérificateur : le théorème de correction

Théorème de correction

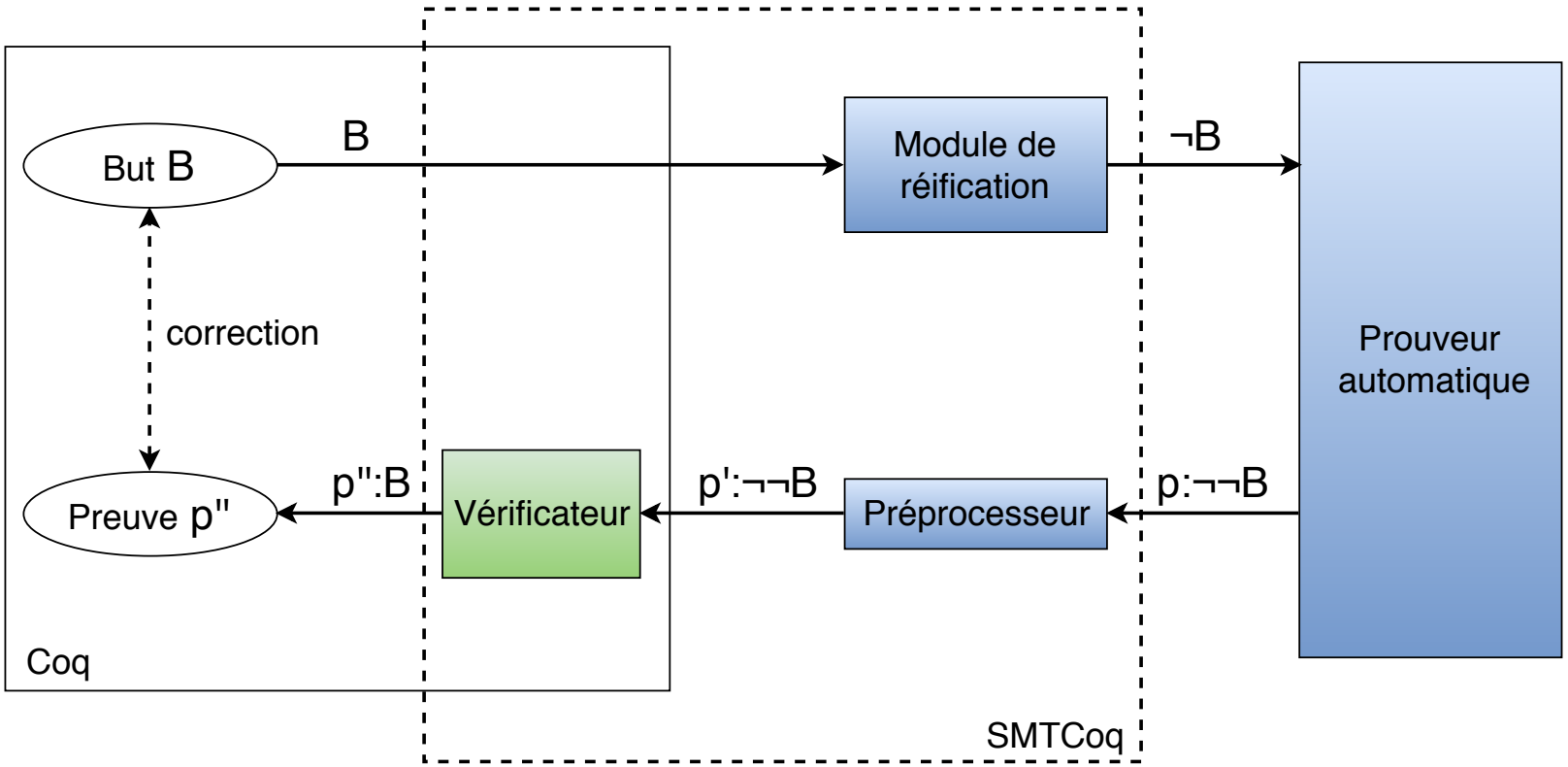
Si `checker cert in = true` pour un certificat `cert` et une formule d'entrée `in` alors `in` n'est pas satisfiable.

Point clé de la preuve : lemme de correction d'une étape
$\rightarrow$ Modularité de la preuve de correction

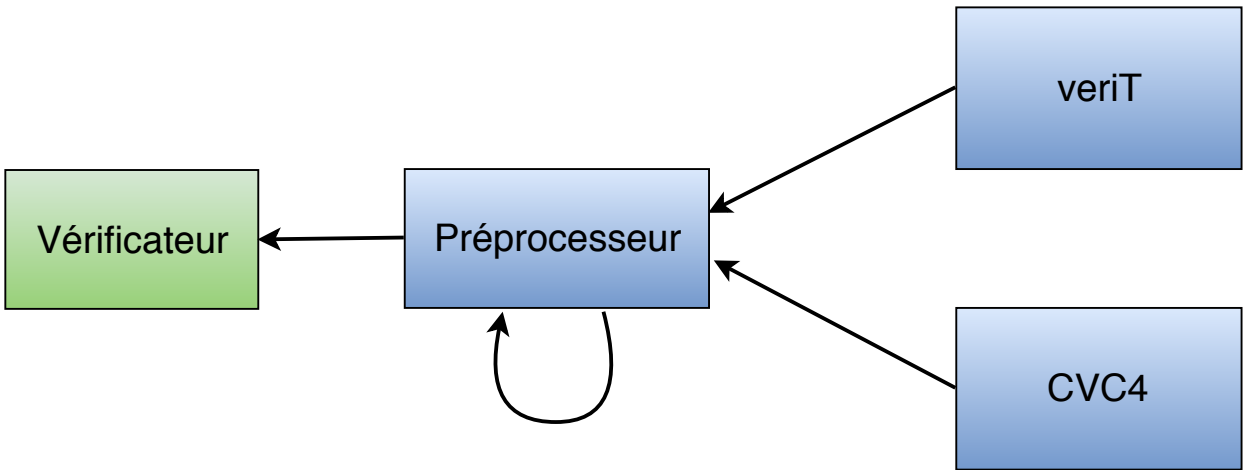
$$\dfrac{\dfrac{\dfrac{}{\texttt{true = true}}\,\mathit{refl}}{\texttt{checker cert in = true}}\,\mathit{conv} \qquad \dfrac{\mathit{correction\ d'une\ étape}}{\mathit{correction}}}{\texttt{interp in = false}}\,\mathit{MP}$$

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

SMTCoq en détail



Préprocesseur



Le préprocesseur de SMTCoq modifie les certificats :

- parsing
- adaptations
- simplifications
- optimisations

Amélioration de l'expressivité : quantificateurs

À partir des lemmes suivants :

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

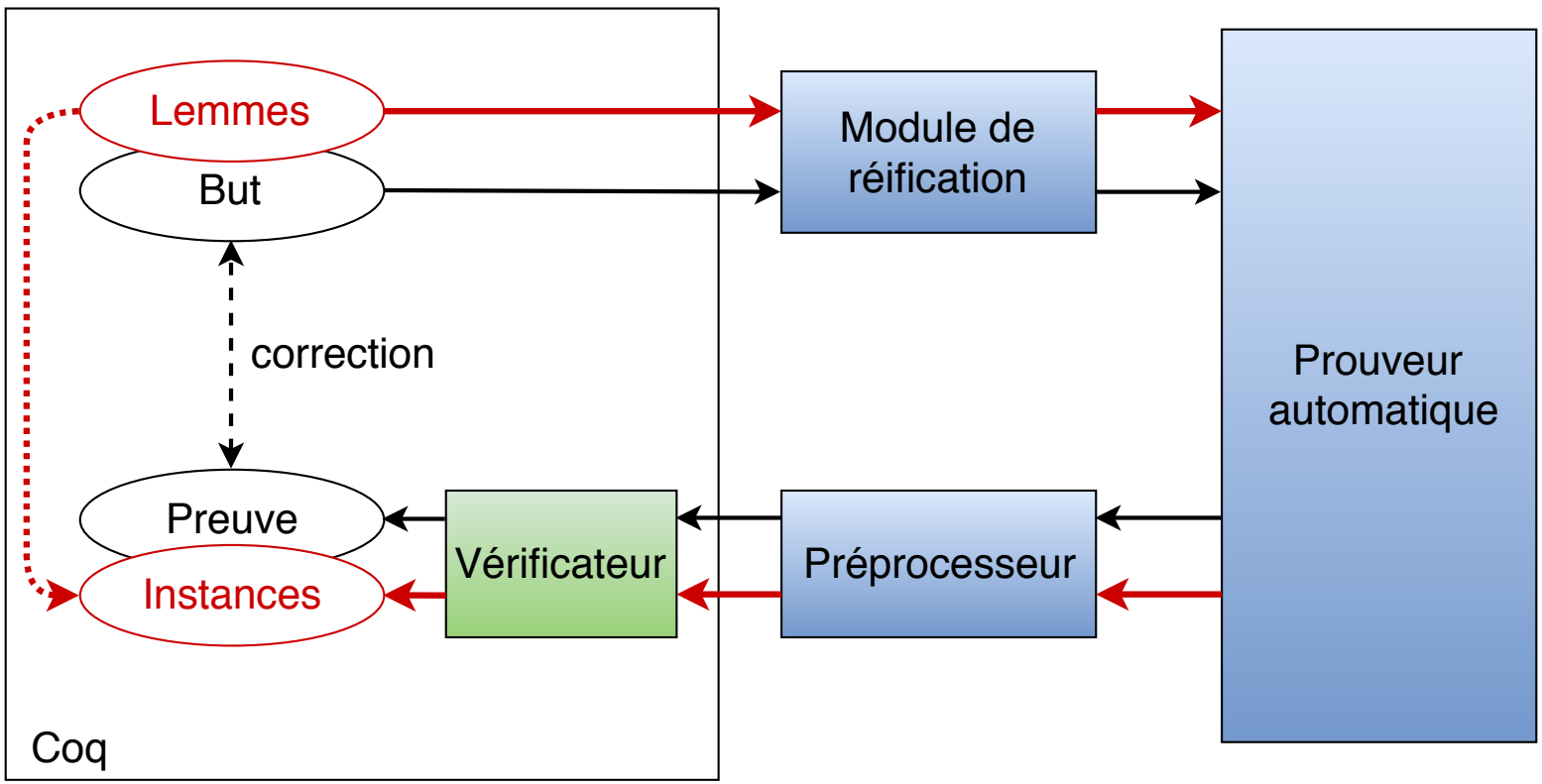
$$homme(Socrate)$$

SMTCoq ne savait pas montrer que :

$$mortel(Socrate)$$

Objectif : permettre l'ajout de lemmes quantifiés au contexte

Ajout de lemmes au contexte



La règle Inst

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

Champ `inst` : déduit par cette nouvelle règle

Avantage : type des formules inchangé
$\rightarrow$ modularité de la preuve de correction

L'instance ne doit pas contenir de quantificateurs.
$\rightarrow$ quantificateurs uniquement en tête de formule dans les lemmes

Préprocesseur pour veriT : fichiers de certificat

Certificat veriT du syllogisme de Socrate :

$0 : (\mathit{input}\ (\forall x.\mathit{homme}\, x \Rightarrow \mathit{mortel}\, x))$ `; premier lemme`

$1 : (\mathit{input}\ (\mathit{homme}\ \mathit{Socrate}))$ `; deuxième lemme`

$2 : (\mathit{input}\ (\neg(\mathit{mortel}\ \mathit{Socrate}))$ `; négation du but`

$3 : (\mathit{forall\_inst}\ (Q))$ `; instanciation`

$4 : (\mathit{resolution}\ ()\ 3\ 0\ 1\ 2)$ `; résolution`

avec : $Q \approx (\forall x.\mathit{homme}\, x \Rightarrow \mathit{mortel}\, x) \Rightarrow (\mathit{homme}\ \mathit{Socrate} \Rightarrow \mathit{mortel}\ \mathit{Socrate})$.

Préprocesseur pour veriT : traduction de certificats

Le préprocesseur transforme les règles

$$3 : (\mathit{forall\_inst}\ (Q))$$
$$4 : (\mathit{resolution}\ ()\ 3\ 0\ 1\ 2)$$

en :

```
Inst ? ? (inst := ins) ?
Res [3; 1; 2]
```

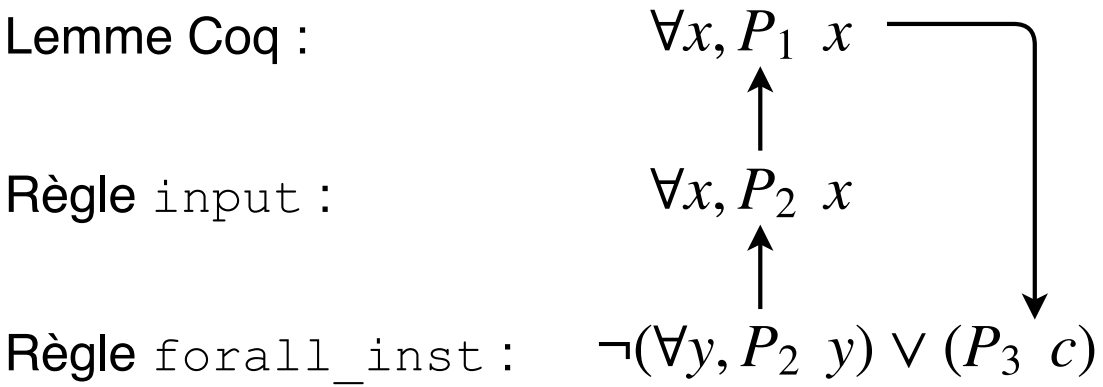
avec `ins` $\approx$ `Impl (homme Socrate) (mortel Socrate)`
et où les `?` sont à compléter

Lemme quantifié (position 0) en argument de `Inst`
$\rightarrow$ changer les règles suivantes

Préprocesseur pour veriT : construction de `Inst`

Champs de la règle `Inst` à remplir :

- `lemma`, reconnaître à quel lemme correspond une instance
- `pinst`, montrer que le lemme reconnu implique l'instance



Instanciation Socrate : $Q \triangleq \neg(\forall y.\mathit{homme}\ y \Rightarrow \mathit{mortel}\ y) \vee (\neg(\mathit{homme}\ \mathit{Socrate}) \vee \mathit{mortel}\ \mathit{Socrate})$

Contribution : ajout de lemmes quantifiés

Règle d'une autre nature : elle utilise à la fois un encodage profond et un encodage superficiel des formules.

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

Amélioration de l'expressivité confirmée par des exemples en théorie des groupes, sur une théorie formalisant les listes d'entiers ...
À partir des axiomes suivants :

$$\forall a\, b\, c. a \times (b \times c) = (a \times b) \times c \qquad \text{(associativité)}$$

$$\forall a. e \times a = a \qquad \text{(neutre à gauche)}$$

$$\forall a. (inv\ a) \times a = e \qquad \text{(inverse à gauche)}$$

on montre automatiquement, à l'aide de SMTCoq :

$$(\forall a. e' \times a = a) \Rightarrow e' = e \qquad \text{(unique neutre à gauche)}$$

Amélioration de l'expressivité : types isomorphes

SMTCoq ne savait pas montrer que :

```
f (x +n 3) =n f (3 +n x)
```

Problème : plusieurs représentations d'une même structure

Exemples :

- `nat`, entiers naturels unaires
- `N`, entiers naturels binaires
- `Z`, entiers relatifs binaires (avec hypothèse de positivité)
- ...

Objectif : convertir les types d'une même structure

Principe

On propose une technique *peu intrusive* vis-à-vis de SMTCoq.

Difficulté: les termes de SMTCoq combinent plusieurs théories

Par exemple :

`f (x +n 3) =n 2` exprimé dans `nat`

↓

`f' (x' +z 3) =z 2` exprimé dans `Z`

Contexte de ce dernier terme :

```
x':  Z                          x' >= 0
f':  Z -> Z          f' (x' +z 3) >= 0
2 :  Z
3 :  Z
```

1ère étape : double conversion

On demande une injection `n2Z` de `nat` dans `Z` :

```
forall x : nat, x = Z2n (n2Z x)
```

Double conversion de certains sous-termes de type `nat`.

Le but `f (x +n 3) =n 2` est transformé en :

```
f (          x  +n          3)    =n          2
```

1ère étape : double conversion

On demande une injection `n2Z` de `nat` dans `Z` :

```
forall x : nat, x = Z2n (n2Z x)
```

Double conversion de certains sous-termes de type `nat`.

Le but `f (x +n 3) =n 2` est transformé en :

```
Z2n (n2Z (f (Z2n (n2Z x) +n Z2n (n2Z 3)))) =n Z2n (n2Z 2)
```

2ème étape : réécriture

Symboles qui commutent avec la conversion :

- `a >= 0 -> b >= 0 -> Z2n a +n Z2n b = Z2n (a +z b)`
- `(Z2n a =n Z2n b) <-> (a =z b)`

Réécriture de ces égalités :

```
Z2n (n2Z (f (Z2n (n2Z x) +n Z2n (n2Z 3)))) =n Z2n (n2Z 2)
```

Pas besoin de double conversion au-dessus des symboles connus

```
Z2n (n2Z x) +n Z2n (n2Z 3)
            ↓
Z2n (n2Z x +z n2Z 3)
```

2ème étape : réécriture

Symboles qui commutent avec la conversion :

- `a >= 0 -> b >= 0 -> Z2n a +n Z2n b = Z2n (a +z b)`
- `(Z2n a =n Z2n b) <-> (a =z b)`

Réécriture de ces égalités :

```
Z2n (n2Z (f (Z2n (n2Z x +z n2Z 3)))) =n Z2n (n2Z 2)
```

Pas besoin de double conversion au-dessus des symboles connus

```
Z2n (n2Z x) +n Z2n (n2Z 3)
            ↓
Z2n (n2Z x +z n2Z 3)
```

2ème étape : réécriture

Symboles qui commutent avec la conversion :

- `a >= 0 -> b >= 0 -> Z2n a +n Z2n b = Z2n (a +z b)`
- `(Z2n a =n Z2n b) <-> (a =z b)`

Réécriture de ces égalités :

```
n2Z (f (Z2n (n2Z x +z n2Z 3))) =z n2Z 2
```

Pas besoin de double conversion au-dessus des symboles connus

```
Z2n (n2Z x) +n Z2n (n2Z 3)
```

↓

```
Z2n (n2Z x +z n2Z 3)
```

3ème étape : renommage

En posant

```
x' := n2Z x
f' := fun y -> n2Z (f (Z2n y))
```

on obtient (avec hypothèses de positivité) :

```
n2Z (f (Z2n (n2Z x +z n2Z 3))) =z n2Z 2
```

Puis le calcul de `n2Z 2` donne la constante 2 de Z.

Ne pas faire de double conversion dans les constantes :

```
S (S 0)
```

```
Z2n (n2Z (S (Z2n (n2Z (S (Z2n (n2Z 0))))))))
```

3ème étape : renommage

En posant

```
x' := n2Z x
f' := fun y -> n2Z (f (Z2n y))
```

on obtient (avec hypothèses de positivité) :

```
n2Z (f (Z2n (x' +z n2Z 3))) =z n2Z 2
```

Puis le calcul de `n2Z 2` donne la constante 2 de Z.

Ne pas faire de double conversion dans les constantes :

```
S (S 0)
```

```
Z2n (n2Z (S (Z2n (n2Z (S (Z2n (n2Z 0))))))))
```

3ème étape : renommage

En posant

```
x' := n2Z x
f' := fun y -> n2Z (f (Z2n y))
```

on obtient (avec hypothèses de positivité) :

```
f' (x' +z n2Z 3) =z n2Z 2
```

Puis le calcul de `n2Z 2` donne la constante 2 de `Z`.

Ne pas faire de double conversion dans les constantes :

```
S (S 0)
```

```
Z2n (n2Z (S (Z2n (n2Z (S (Z2n (n2Z 0)))))))
```

Contribution : conversions de type

Extension de SMTCoq aux buts avec différents types d'entiers

Foncteur dépendant d'un type `T` tel que :

- il existe une fonction `T2Z` de `T` vers `Z`
- `T2Z` est injective
- `T2Z` respecte la structure des entiers

Implémentation : définition et instanciation de ce foncteur

Possibilité de définir d'autres foncteurs pour d'autres types

Conclusion

Amélioration de l'expressivité de SMTCoq :

- ajout de quantificateurs
- conversions de types

Continuer à améliorer l'expressivité :

- gestion des quantificateurs de CVC4
- conversions d'autres types (tableaux, vecteurs de bits)
- logique du premier ordre

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

$\longrightarrow$ `smtcoq.github.io`