Décision automatique de relations inductives pour SMTCoq

Louise Dubois de Prisque

Université Paris-Saclay, Laboratoire Méthodes Formelles, INRIA

2 Février 2024
Journée Francophone des Langages Applicatifs

Table des matières

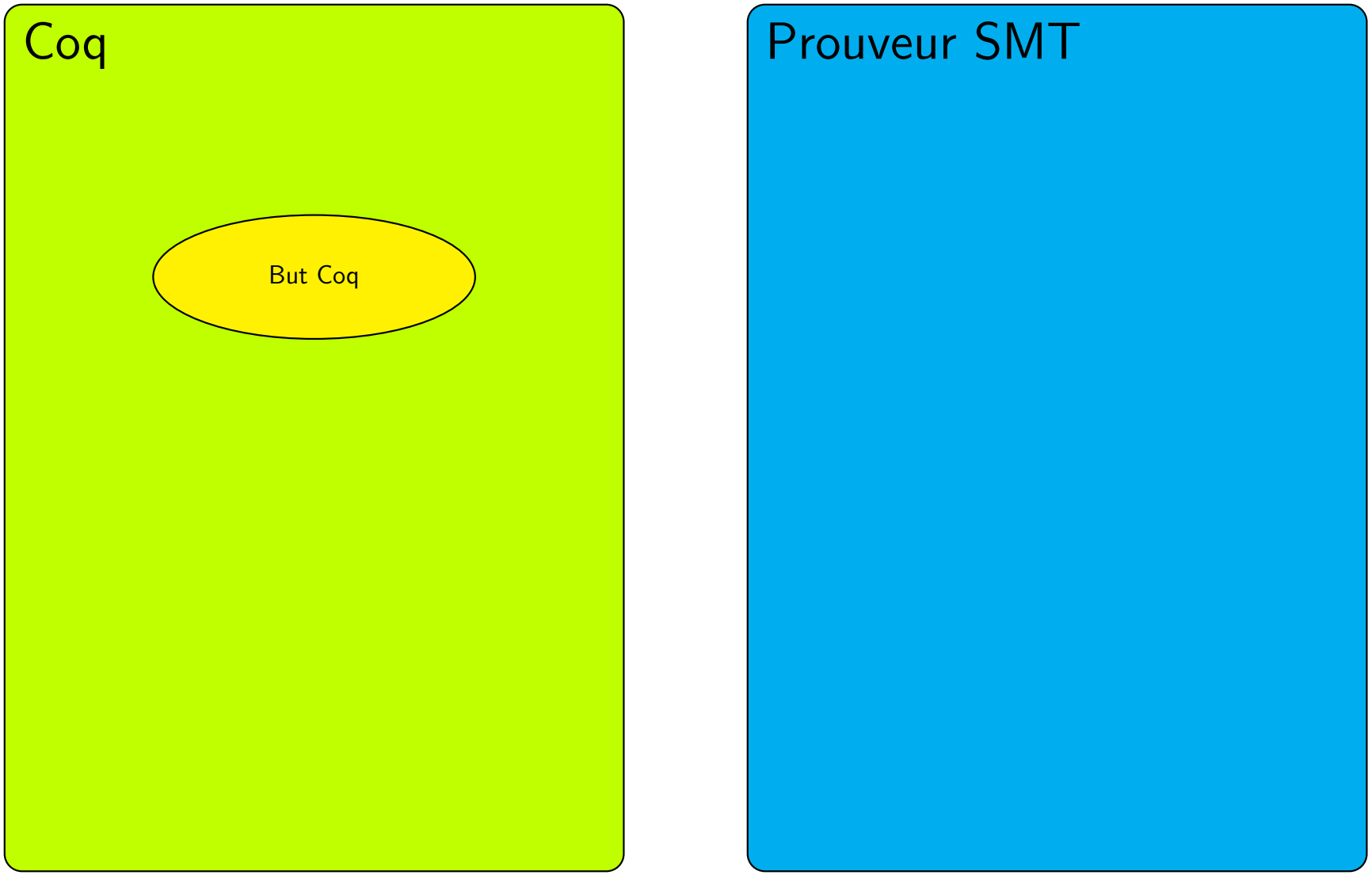
Les relations inductives

Exemple

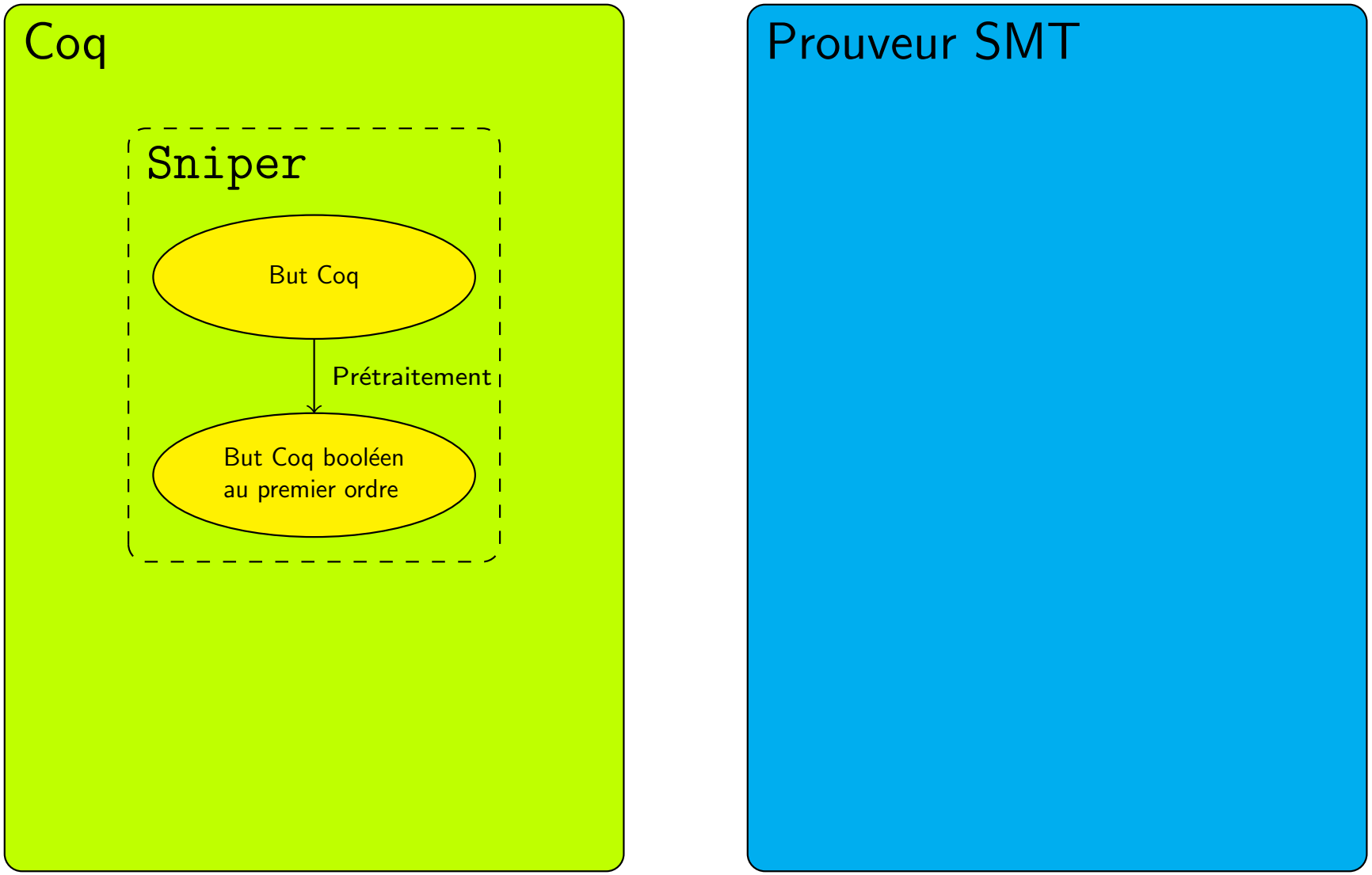
```
Inductive mem : Z → list Z → Prop :=
| MemRecur : forall (n n': Z) (l: list Z),
    mem n l → mem n (n'::l)
| MemMatch : forall (n: Z) (l: list Z),
    mem n (n::l).
```

- Écrire des spécifications en Coq
- Naturel, lisible, tactiques adaptées (`induction`, `inversion`...)
- Constructeur = règle d'inférence
- Décidables ou non

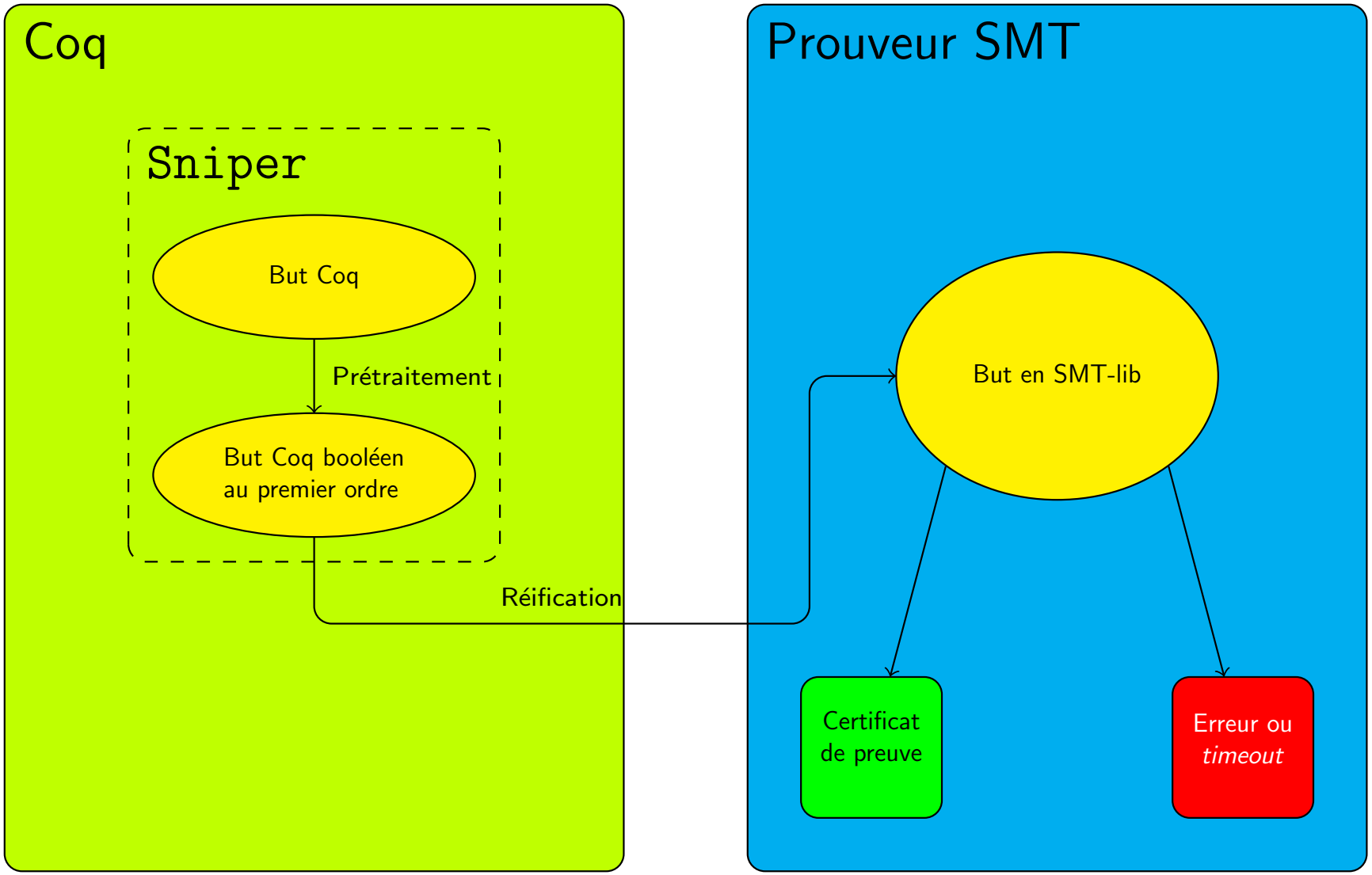
Fonctionnement de SMTCoq et de Sniper



Fonctionnement de SMTCoq et de Sniper



Fonctionnement de SMTCoq et de Sniper



Fonctionnement de SMTCoq et de Sniper

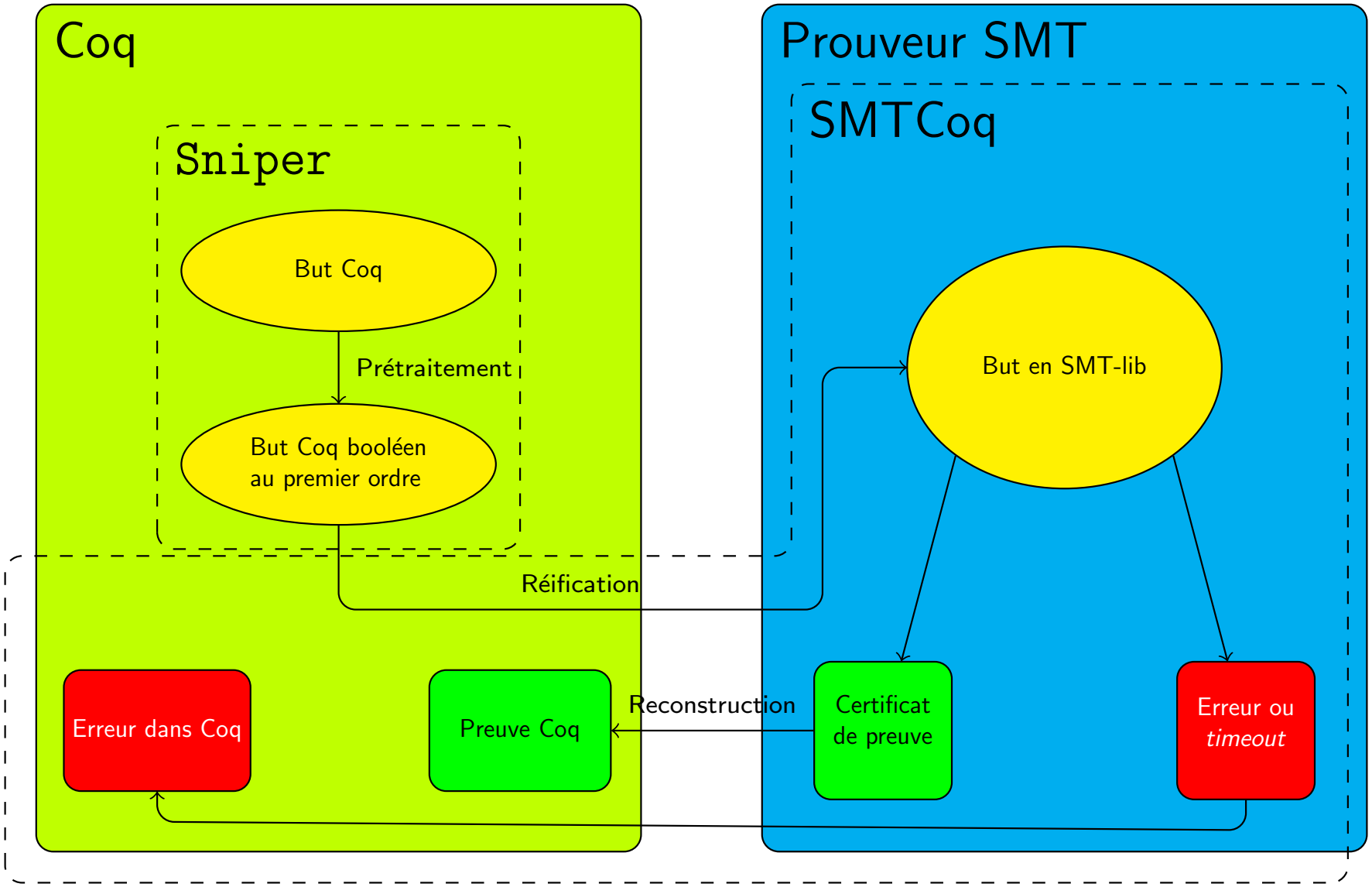


Table des matières

Deux versions d'un but Coq

- Avec une relation inductive :

```
Goal (forall (n: Z), mem n [] → False).
Fail snipe.
```

Deux versions d'un but Coq

- Avec une relation inductive :

```
Goal (forall (n: Z), mem n [] → False).
Fail snipe.
```

- Avec une fonction booléenne :

```
Goal (forall (n: Z), mem_decidable n [] = true
   → False).
snipe. Qed.
```

Deux versions d'un but Coq

- Avec une relation inductive :

```
Goal (forall (n: Z), mem n [] → False).
Fail snipe.
```

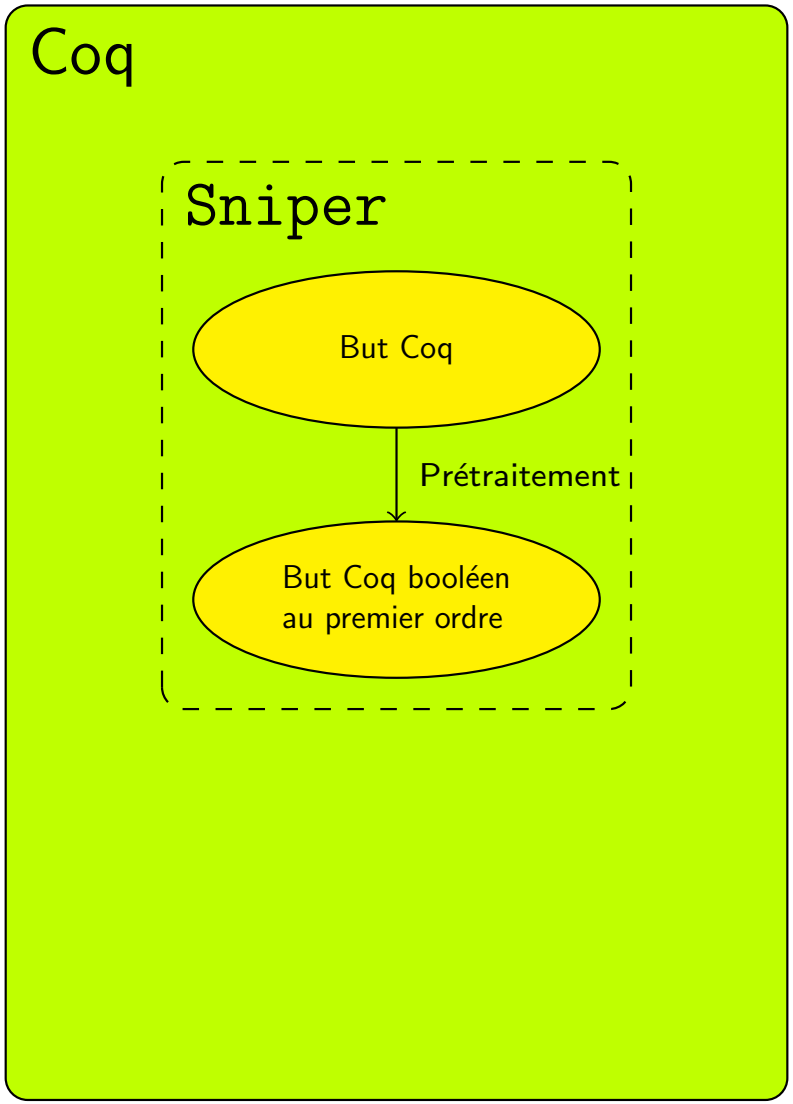
- Avec une fonction booléenne :

```
Goal (forall (n: Z), mem_decidable n [] = true
   → False).
snipe. Qed.
```

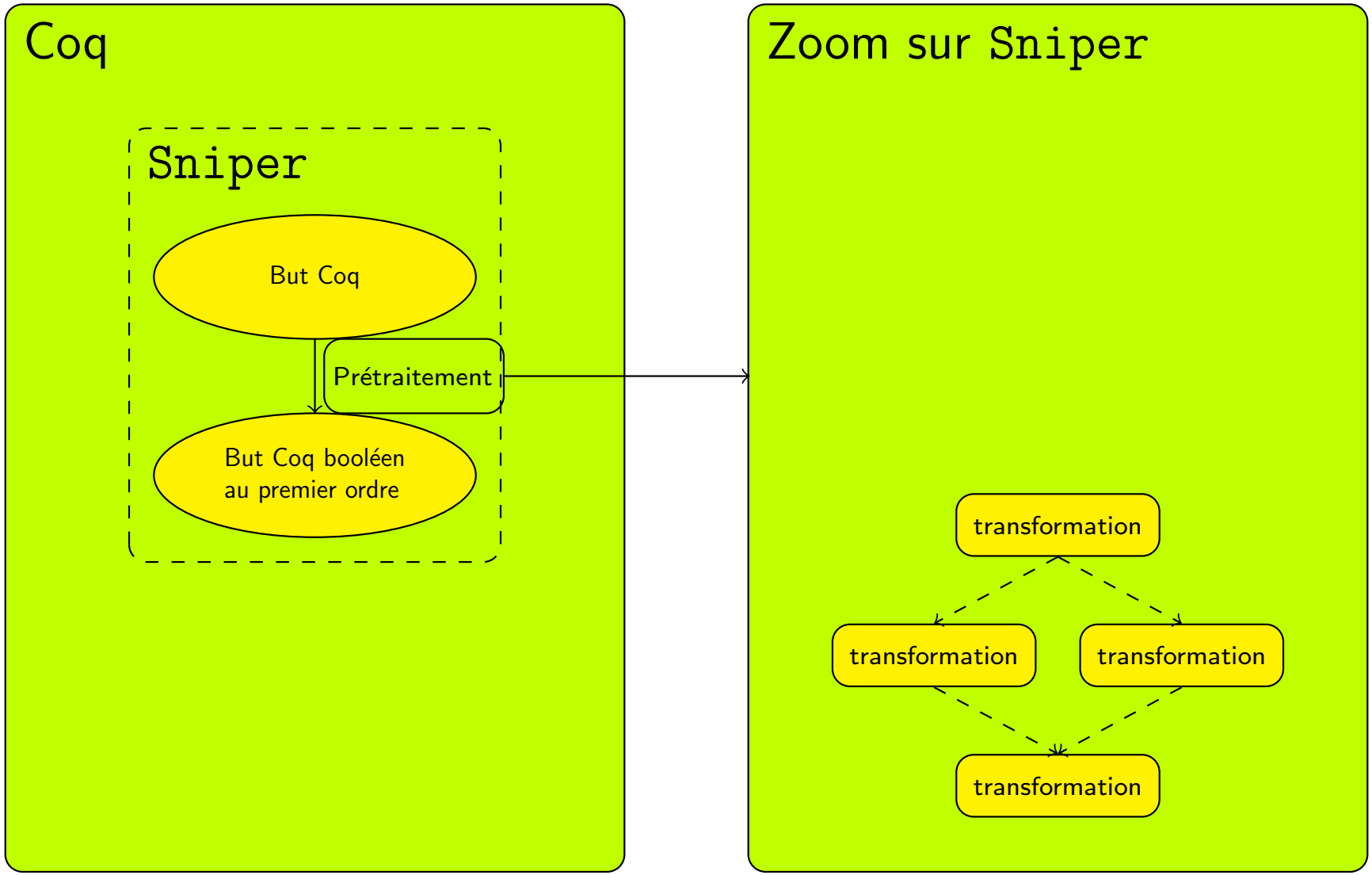
- Pas de traduction de `Prop` à `bool` dans `Sniper` pour les prédicats définis par l'utilisateur...

Table des matières

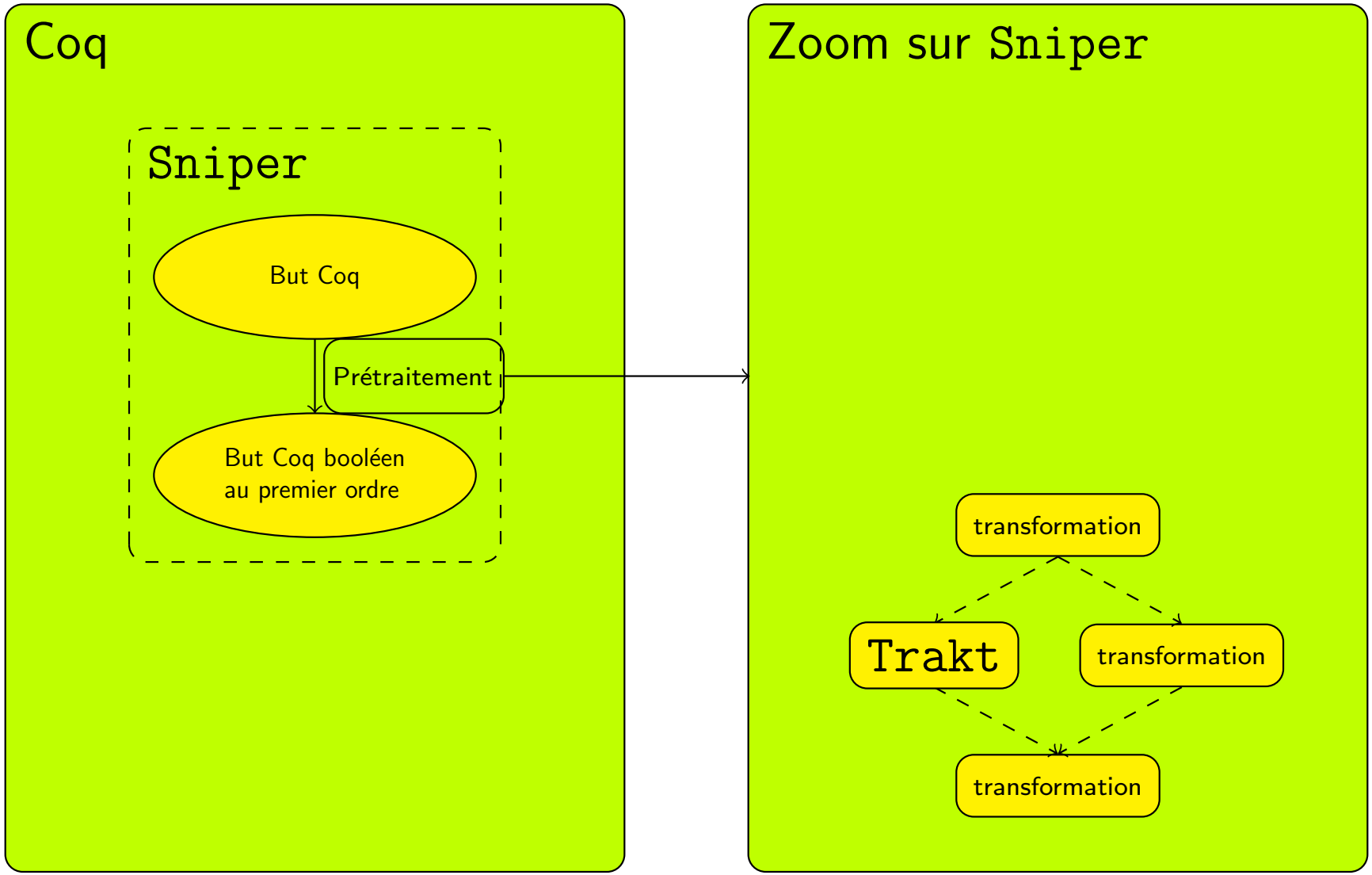
Notre contribution



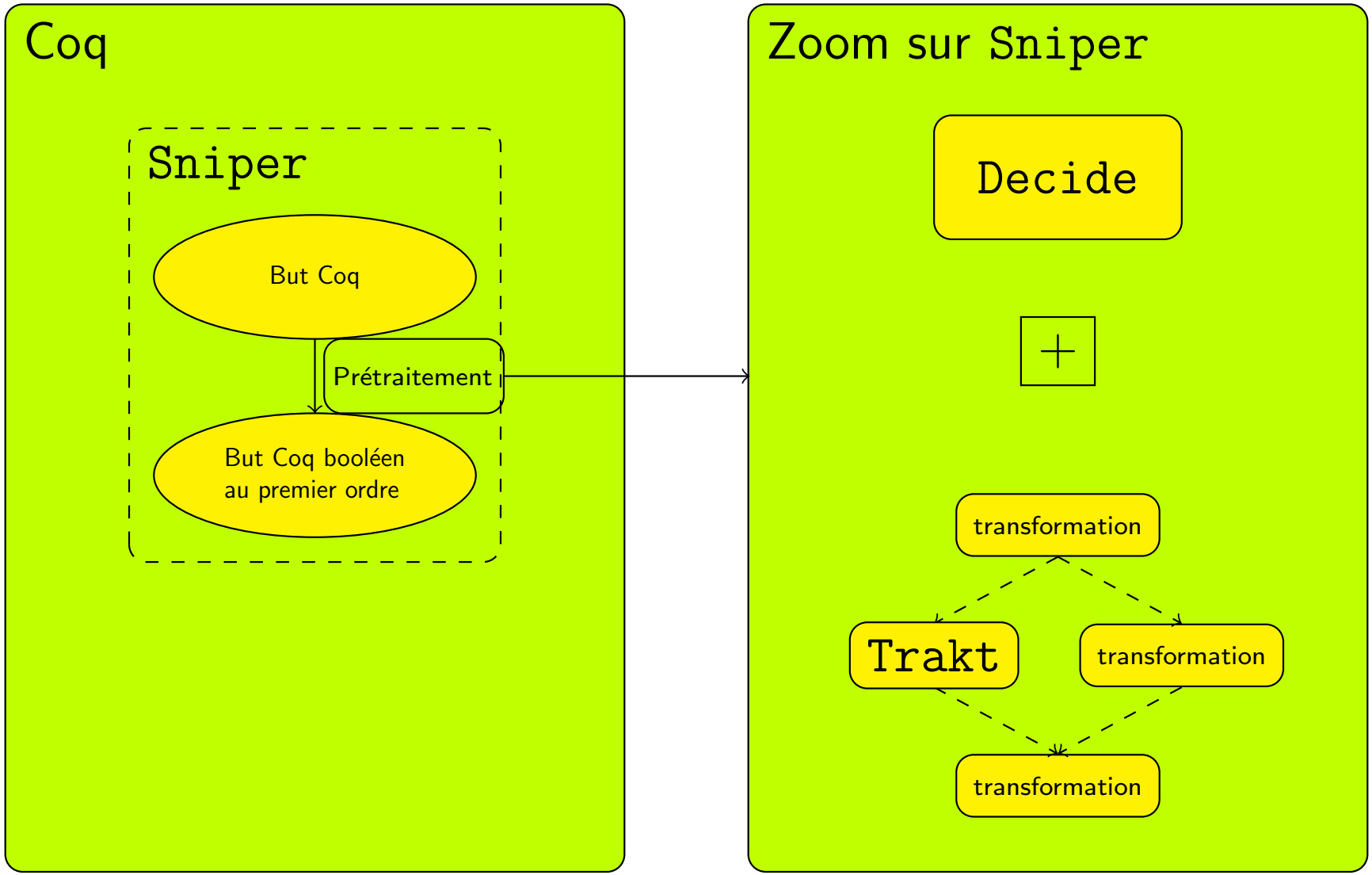
Notre contribution



Notre contribution



Notre contribution



Notre contribution

- Une commande :

```
MetaCoqRun (decide mem).
```

Notre contribution

- Une commande :

```
MetaCoqRun (decide mem).
```

- Génère le point fixe :

```
mem_decidable =
fix mem_decidable (n : Z) (l : list Z)
    {struct l} : bool :=
match l with
| [] ⇒ false
| x :: xs ⇒ Nat.eqb n x || mem_decidable
    n xs
end
```

Notre contribution

- Une commande :

```
MetaCoqRun (decide mem).
```

- Génère le point fixe :

```
mem_decidable =
fix mem_decidable (n : Z) (l : list Z)
    {struct l} : bool :=
match l with
| [] ⇒ false
| x :: xs ⇒ Nat.eqb n x || mem_decidable
    n xs
end
```

- Génère la preuve d'équivalence dont le type est :

```
forall n l, mem n l ↔ mem_decidable n l = true
```

Table des matières

Table des matières

Exemple détaillé

Relation initiale

```
Inductive mem : Z → list Z → Prop :=
| MemRecur : forall (n n': Z) (l: list Z),
 mem n l → mem n (n'::l)
| MemMatch : forall (n: Z) (l: list Z),
 mem n (n::l).
```

Point fixe

```
Fixpoint mem n l :=
match ?? with
| ...
end.
```

Exemple détaillé

Relation initiale

```
Inductive mem : Z → list Z → Prop :=
| MemRecur : forall (n n': Z) (l: list Z),
 mem n l → mem n (n'::l)
| MemMatch : forall (n: Z) (l: list Z),
 mem n (n::l).
```

Point fixe : premier constructeur

```
Fixpoint mem n l :=
 match l with
  | [] ⇒ false
  | n :: l ⇒ mem n l || ??
 end.
```

Exemple détaillé

Relation initiale

```
Inductive mem : Z → list Z → Prop :=
| MemRecur : forall (n n': Z) (l: list Z),
 mem n l → mem n (n'::l)
| MemMatch : forall (n: Z) (l: list Z),
 mem n (n::l).
```

Point fixe : deuxième constructeur

```
Fixpoint mem n l :=
 match l with
  | [] ⇒ false
  | n :: l ⇒ mem n l || true
 end. (* Incorrect !!! *)
```

Exemple détaillé

Relation initiale

```
Inductive mem : Z → list Z → Prop :=
| MemRecur : forall (n n': Z) (l: list Z),
 mem n l → mem n (n'::l)
| MemMatch : forall (n: Z) (l: list Z),
 mem n (n::l).
```

Point fixe : deuxième constructeur

```
Fixpoint mem n l :=
 match l with
  | [] ⇒ false
    n'                          n'?==n
  | n:: l ⇒ mem n l || true
 end.
```

Exemple détaillé

Linéarisation

```
Inductive mem_l : Z → list Z → Prop :=
| MemRecur_l : forall (n n': Z) (l: list Z),
 mem_l n l → mem_l n (n'::l)
| MemMatch_l : forall (n n': Z) (l: list Z),
n'?==n → mem_l n (n'::l).
```

Point fixe : deuxième constructeur

```
Fixpoint mem n l :=
 match l with
  | [] ⇒ false
    n'                          n'?==n
  | n:: l ⇒ mem n l || true
 end.
```

Table des matières

Récapitulatif

- Recherche des lemmes d'égalité décidables

Récapitulatif

- Recherche des lemmes d'égalité décidables
- Linéarisation de la relation inductive

Récapitulatif

- Recherche des lemmes d'égalité décidables
- Linéarisation de la relation inductive
- Génération du point fixe

Récapitulatif

- Recherche des lemmes d'égalité décidables
- Linéarisation de la relation inductive
- Génération du point fixe
- Génération et preuve du lemme d'équivalence :
  - Heuristiques (script `Ltac2`)
  - Preuve manuelle si échec

Table des matières

Explication des étapes

- Décision de la relation inductive via la commande `Decide`

Explication des étapes

- Décision de la relation inductive via la commande `Decide`
- Ajout du lemme d'équivalence dans la base de données de `Trakt`

Explication des étapes

- Décision de la relation inductive via la commande `Decide`
- Ajout du lemme d'équivalence dans la base de données de `Trakt`
- La tactique « pousse-bouton » `snipe` connaît désormais la relation !

Table des matières

Conclusion et perspectives

- Conclusion :
  - Un nouveau module `Decide` dans `Sniper`
  - Code : `https://github.com/smtcoq/sniper/tree/coq-8.17-with-trakt`
- Perspectives :
  - Une version « tactique »
  - Étendre les cas gérés : pas besoin que la conclusion d'un constructeur mentionne tous ses arguments
  - Terme de preuve suivant la structure du point fixe

Conclusion bis : Le choix du métalangage

- Manipulation de termes ouverts : `MetaCoq` (`coq-elpi` avec HOAS plus adapté pour traverser des AST)

Conclusion bis : Le choix du métalangage

- Manipulation de termes ouverts : `MetaCoq` (`coq-elpi` avec HOAS plus adapté pour traverser des AST)
- Point faibles : tactiques dans la `TemplateMonad`, messages d'erreur... (points faibles de Coq en tant que langage de programmation)

Conclusion bis : Le choix du métalangage

- Manipulation de termes ouverts : `MetaCoq` (`coq-elpi` avec HOAS plus adapté pour traverser des AST)
- Point faibles : tactiques dans la `TemplateMonad`, messages d'erreur... (points faibles de Coq en tant que langage de programmation)
- Tactiques agissant sur le contexte : `Ltac2`

La question de la terminaison

- On cherche un potentiel appel récursif dans chaque constructeur de la relation inductive

La question de la terminaison

- On cherche un potentiel appel récursif dans chaque constructeur de la relation inductive
- Condition : l'un des arguments (le même pour chaque constructeur) doit toujours avoir pour symbole de tête un constructeur

La question de la terminaison

- On cherche un potentiel appel récursif dans chaque constructeur de la relation inductive
- Condition : l'un des arguments (le même pour chaque constructeur) doit toujours avoir pour symbole de tête un constructeur
- Si cela ne fonctionne pas, possibilité de donner l'argument manuellement

Travaux connexes

Nom du plugin	Compatible avec les versions récentes de Coq	Compatible avec SMTCoq	Plusieurs modes d'extraction
`RelationExtraction`	☹	☹	☺
`Dec-deriving`	☺	☹	☺
`Decide`	☺	☺	☹

La question du polymorphisme

- Classe de types `CompDec` dans SMTCoq

La question du polymorphisme

- Classe de types `CompDec` dans SMTCoq
- Bibliothèque de preuves : `CompDec Z`,
  `forall A, CompDec A → CompDec (list A)`, etc.

La question du polymorphisme

- Classe de types `CompDec` dans SMTCoq
- Bibliothèque de preuves : `CompDec Z`,
  `forall A, CompDec A → CompDec (list A)`, etc.
- Ajout d'hypothèses sur les variables de types polymorphes

La question du polymorphisme

- Classe de types `CompDec` dans SMTCoq
- Bibliothèque de preuves : `CompDec Z`,
  `forall A, CompDec A → CompDec (list A)`, etc.
- Ajout d'hypothèses sur les variables de types polymorphes

```
Inductive mem' (A: Type) : A → list A → Prop :=
| MemMatch' : forall a l, mem' A a (a::l)
| MemRecur' : forall a a' (l: list A), mem' A a
    l → mem' A a (a'::l).
Inductive mem'' (A: Type) (HA : CompDec A) : A →
    list A → Prop :=
| MemMatch'' : forall a l, mem'' A HA a (a::l)
| MemRecur'' : forall a a' (l: list A), mem'' A
    HA a l → mem'' A HA a (a'::l).
```