When Bécassine brings automation to Coq

Valentin Blot[1,2] Louise Dubois de Prisque [1,2], Chantal Keller[2], Pierre Vial[1,2]

[1]Deducteam (Inria Paris-Saclay) [2]LMF (Gif-sur-Yvette)

This work is funded by Nomadic Labs

November 4, 2021

- **Coq**: **proof assistant** based on **type theory** and the **Curry-Howard isomorphism**

  Formulas = types, proofs = programs

  - Four Colors Theorem
  - Feit-Thomson Theorem
  - `CompCert` (certified compiler)

- These successes are possible because of its design
  - Strong **type-checking** within `Coq`
  - **Rich specification** language
  - **Highly trusted** (small logical kernel)

```
Goal forall (A : Type) (l : list A) (n : nat), length l = S n → l ≠ [].
Proof. intros A l n H H'. rewrite H' in H. discriminate. Qed.
```

```
Goal forall (A : Type) (l : list A) (n : nat), length l = S n → l ≠ [].
Proof. intros A l n H H'. rewrite H' in H. discriminate. Qed.
```

Let us admit

```
Lemma search_app : forall (A: Type) (x: A) (l1 l2: list A),
                  search x (l1 ++ l2) = (search x l1) || (search x l2).
```

```
Goal forall (A : Type) (l : list A) (n : nat), length l = S n → l ≠ [].
Proof. intros A l n H H'. rewrite H' in H. discriminate. Qed.
```

Let us admit

```
Lemma search_app : forall (A: Type) (x: A) (l1 l2: list A),
            search x (l1 ++ l2) = (search x l1) || (search x l2).
```

```
Goal forall (A : Type) (x: A) (l1 l2 l3: list A),
            search x (l1 ++ l2++l3) = search x (l3 ++ l2 ++ l1).
Proof. intros A H x l1 l2 l3. rewrite !search_app.
rewrite orb_comm with (b1 := search x l3).
rewrite orb_comm with (b1 := search x l2) (b2 := search x l1).
rewrite orb_assoc. reflexivity . Qed.
```

```
Goal forall (A : Type) (l : list A) (n : nat), length l = S n → l ≠ [].
Proof. intros A l n H H'. rewrite H' in H. discriminate. Qed.
```

Let us admit

```
Lemma search_app : forall (A: Type) (x: A) (l1 l2: list A),
                   search x (l1 ++ l2) = (search x l1) || (search x l2).
```

```
Goal forall (A : Type) (x: A) (l1 l2 l3: list A),
                   search x (l1 ++ l2++l3) = search x (l3 ++ l2 ++ l1).
Proof. intros A H x l1 l2 l3. rewrite !search_app.
rewrite orb_comm with (b1 := search x l3).
rewrite orb_comm with (b1 := search x l2) (b2 := search x l1).
rewrite orb_assoc. reflexivity . Qed.
```

Coq lacks automation

- The user must be very specific
- Difficult for the beginner/non-formal method specialist
- May discourage new users (*e.g.*, maths, industry)

```
Goal forall (A : Type) (l : list A) (n : nat), length l = S n → l ≠ [].
Proof. snipe. Qed.
```

```
Lemma search_app : forall (A: Type) (x: A) (l1 l2: list A),
                   search x (l1 ++ l2) = (search x l1) || (search x l2).
Proof. induction l1 ; snipe. Qed.
```

```
Goal forall (A : Type) (x: A) (l1 l2 l3: list A),
                   search x (l1 ++ l2++l3) = search x (l3 ++ l2 ++ l1).
Proof. intros A H. snipe search_app. Qed.
```

Coq lacks automation

- The user must be very specific
- Difficult for the beginner/non-formal method specialist
- May discourage new users (*e.g.*, maths, industry)

Motivation: improving the automation of Coq

Coq (Proof assistant)	First-order provers
Very expressive logic	**Limited expressivity**
Checks proofs	**Finds proofs**
Highly trustable	**Less so**

- `Coq` **difficult to automatize**
- Even the first-order part of the proofs
- **FOL highly automated** outside `Coq`
- Line of software development:
  call external solvers to handle the first-order parts of the proofs
  (avoid **redundant code**!)
- Partial transformations from `Coq` logic to FOL

Plan

1. `Coq` *vs.* automated provers

2. `Sniper`

Why trust Coq?

Trusting Coq:

- Typing system

  strong normalization/consistency

- Implementation of the typing rules

Why trust Coq?

Trusting Coq:

- Typing system

  strong normalization/consistency

- Implementation of the typing rules

Coq

Kernel

Type checker

$\sim$ *6000 loc*

Why trust Coq?

Trusting Coq:

- Typing system

  strong normalization/consistency

- Implementation of the typing rules

Coq

Kernel

Type checker

$\sim$ *6000 loc*

- IDEs
- Tactics (automation), *e.g.*, Ltac
- Plugins (incl. `SMTCoq` and `MetaCoq`)
- Machine learning

Trusting Coq:

- Typing system

  strong normalization/consistency

- Implementation of the typing rules

Coq

Kernel

Type checker

∼ *6000 loc*

- IDEs
- Tactics (automation), *e.g.*, Ltac
- Plugins (incl. `SMTCoq` and `MetaCoq`)
- Machine learning

This part does not need to be trusted

Why trust Coq?

Trusting Coq:

- Typing system

  strong normalization/consistency

- Implementation of the typing rules

≠ First-order provers

whole code has to be trusted

(autom., search, optim.)

Coq

Kernel

Type checker

$\sim$ *6000 loc*

- IDEs
- Tactics (automation), *e.g.*, Ltac
- Plugins (incl. `SMTCoq` and `MetaCoq`)
- Machine learning

This part does not need to be trusted

Coq
based on the *Calculus of Inductive Constructions (CIC)*

First-order logic (FOL)
- functions and relations
- basic datatypes (**bool**, **int**, **float**)
- boolean equality
- quantification over objects

incl. linear integer arithmetics, etc

In CIC but not in FOL:

- Higher-order computation (functions are first-class objects):

```
map f [x1 ; ... ; xn] := [f x1 ; ... ; f xn ]
```
⇝ `map f` is a function on lists

- Higher-order quantification

```
forall (A B C : Type) (f : A -> B) (g : B -> C), (map g) o (map f) = map (g o f)
```

- Dependent types, *e.g.*, `Vec A n` is definable

the type of lists of length `n` whose elements have type `A`

Coq Logic *vs.* First-Order Logic

Coq

based on the *Calculus of Inductive Constructions (CIC)*

First-order logic (FOL)

- functions and relations
- basic datatypes (**bool**, **int**, **float**)
- boolean equality
- quantification over objects

incl. linear integer arithmetics, etc

Coq Logic *vs.* First-Order Logic

Coq
based on the *Calculus of Inductive Constructions (CIC)*

First-order logic (FOL)
- functions and relations
- basic datatypes (**bool**, **int**, **float**)
- boolean equality
- quantification over objects

incl. linear integer arithmetics, etc

Zoom on Coq inductives

- Inductive types

```
Inductive list (A : Type) : Type :=
[ ] : list A | _ :: _ : → list A → list A
```

- Fixpoints and pattern-matching:

```
Fixpoint length { A : Type } (l: list A) := match l with
  [ ] ⇒ 0 | a :: l0 ⇒ 1 + length l0
```

- Generic (non-boolean) Leibniz equality on any type

Leibniz equality is a dependent type

- When we make two programs interact, we need an interface

```
Theorem destruct_list : forall l : list A, {x:A & {tl:list A | l = x::tl}}+{l = nil}.
Proof.
  induction l as [|a tl].
  right; reflexivity.
  left; exists a; exists tl; reflexivity.
Qed.
```

A Coq Theorem and its proof

```
1:(input (#1:(= op_3 #2:(op_1 op_4 op_5))))
2:(input (#3:(forall ( (RelName10 Tindex_1) (RelName11 Tindex_2) ) #4:(=> #5:(= op_3 #6:(op_1 RelName11 R
3:(tmp_betared (#7:(forall ( (@vr10 Tindex_1) (@vr11 Tindex_2) ) #8:(=> #9:(= op_3 #10:(op_1 @vr11 @vr10)
4:(tmp_qnt_tidy (#11:(forall ( (@vr14 Tindex_1) (@vr16 Tindex_2) ) #12:(=> #13:(= op_3 #14:(op_1 @vr16 @v
5:(forall_inst (#15:(or (not #11) #16:(=> #1 false))))
6:(false ((not false)))
7:(implies_pos ((not #16) (not #1) false))
```

Excerpt of an smt2 certificate

The problem of reification

- When we make two programs interact, we need an interface

```
Theorem destruct_list : forall l : list A, {x:A & {tl:list A | l = x::tl}}+{l = nil}.
Proof.
  induction l as [|a tl].
  right; reflexivity.
  left; exists a; exists tl; reflexivity.
Qed.
```

A Coq Theorem and its proof

- When we make two programs interact, we need an interface

```
Theorem destruct_list : forall l : list A, {x:A & {tl:list A | l = x::tl}}+{l = nil}.
Proof.
  induction l as [|a tl].
  right; reflexivity.
  left; exists a; exists tl; reflexivity.
Qed.
```

A `Coq` Theorem and its proof

Need for **reification** (or quoting)

⇝ translating programs of a language $\mathcal{L}$ into another language $\mathcal{L}'$.

The problem of reification

- When we make two programs interact, we need an interface

```
Theorem destruct_list : forall l : list A, {x:A & {tl:list A | l = x::tl}}+{l = nil}.
Proof.
  induction l as [|a tl].
  right; reflexivity.
  left; exists a; exists tl; reflexivity.
Qed.
```

A `Coq` Theorem and its proof

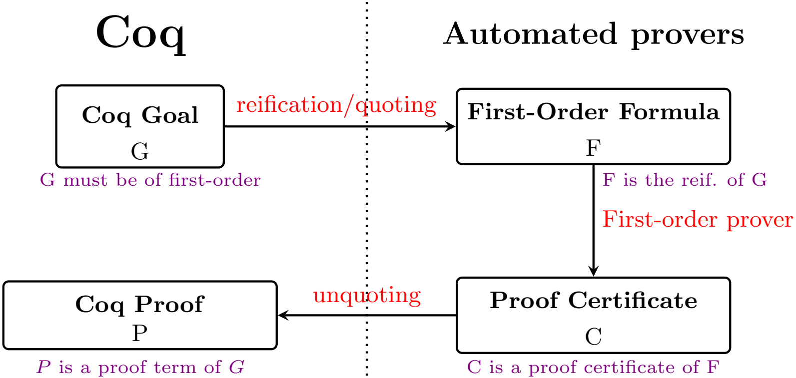
> Need for **reification** (or quoting)
> $\rightsquigarrow$ translating programs of a language $\mathcal{L}$ into another language $\mathcal{L}'$.

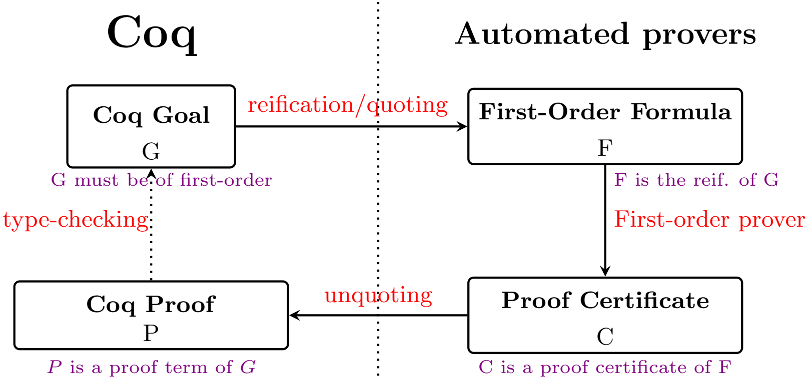
e.g., `forall (A : Set), A → A` (type)

$\rightsquigarrow$ `Prod (name "A") Set_reif (Prod unnamed A (dB 0) (dB 1))`

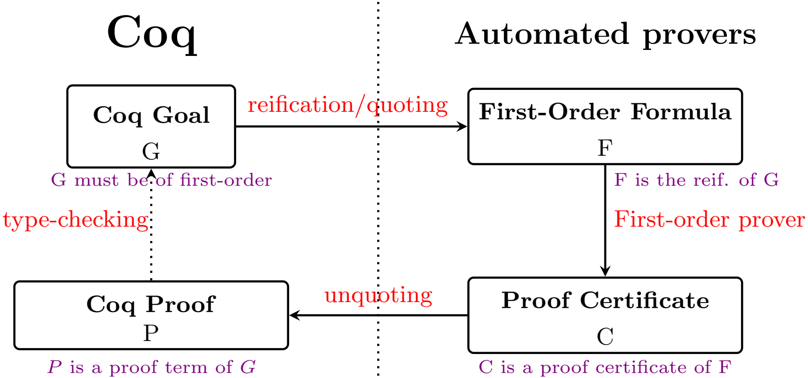
=reif. with de Bruijn indexes

Plugging in provers: autarkic approach

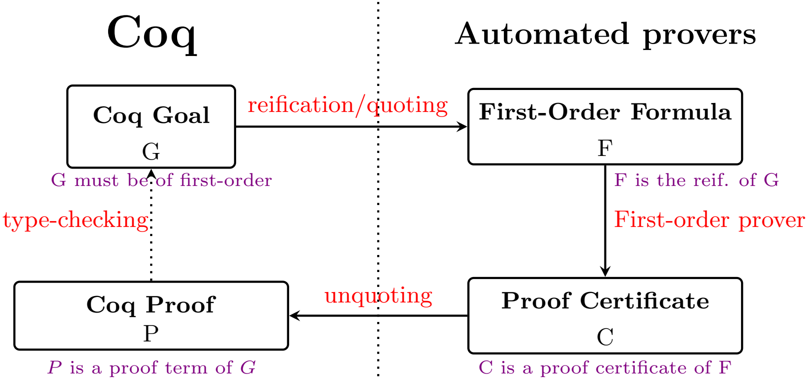



Coq
Automated provers
Coq Goal
G
reification/quoting
First-Order Formula
F
G must be of first-order
F is the reif. of G
type-checking
First-order prover
Coq Proof
P
unquoting
Proof Certificate
C
P is a proof term of G
C is a proof certificate of F

Plugging in provers: autarkic approach

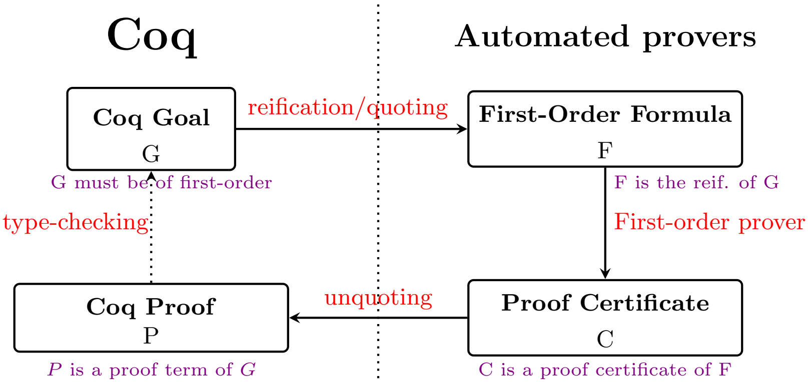


- Horizontal arrows: some `OCaml`



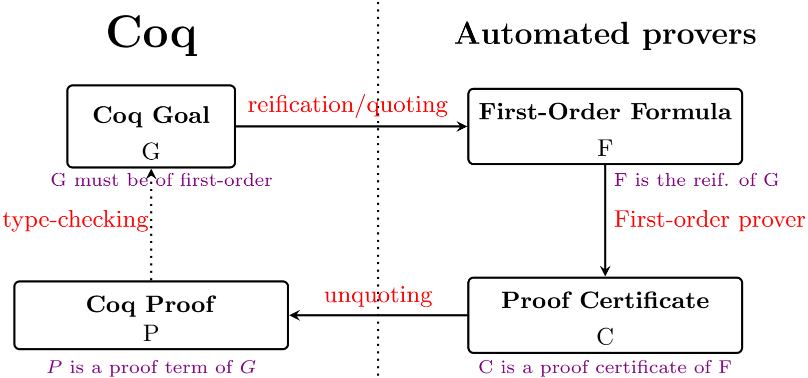
- Horizontal arrows: some `OCaml`
- Any arrow may fail (reification, solving. . .)

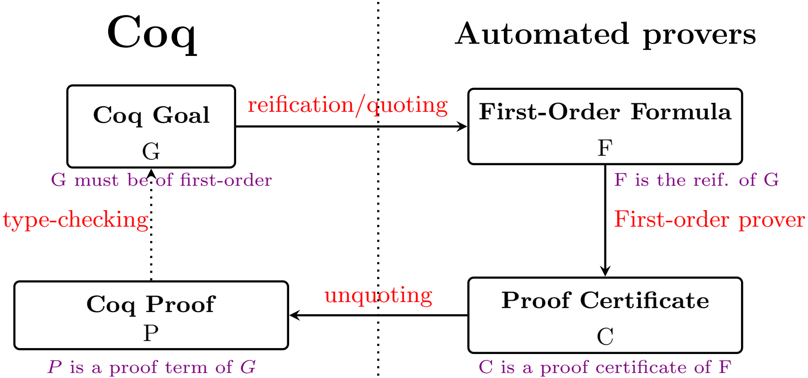
Plugging in provers: autarkic approach



- Horizontal arrows: some `OCaml`
- Any arrow may fail (reification, solving. . .)
- **Autarkic approach**: each certificate is checked on the run

Plugging in provers: autarkic approach

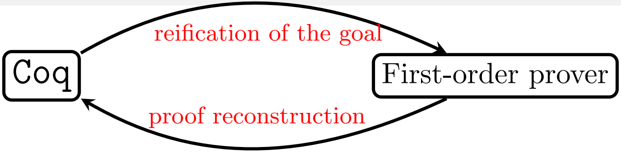




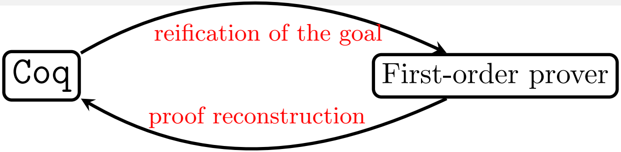
In our case:

- Plugin = `SMTCoq`
- Automated provers = SMT solvers, *e.g.*, veriT
- Under the carpet: casting Leibniz equality into boolean eq. (decidable types only)

Bécassine comes into play



- **Question.** Why aren't we happy with this?

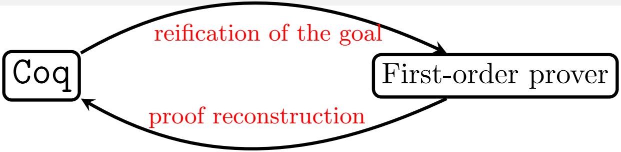


- **Question.** Why aren't we happy with this?

Problem 1
Avoid harmless polymorphism and higher-order

```
- forall (A : Type) (l1 l2 : list A),
   length (l1 ++ l2) = length l1 + length l2
```

- `f = g` with `f,g: nat -> nat`
  instead of $\forall$`(x : nat), f x = g x`



- **Question.** Why aren't we happy with this?

Problem 1
Avoid harmless polymorphism and higher-order

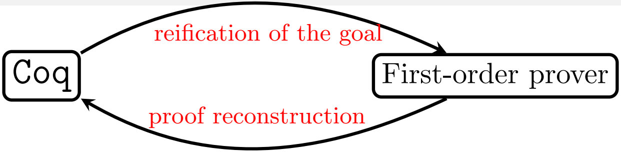
- `forall (A : Type) (l1 l2 : list A),`
  `length (l1 ++ l2) = length l1 + length l2`
- `f = g` with `f,g: nat -> nat`
  instead of $\forall$`(x : nat), f x = g x`

Problem 2
Some info. is lost during goal reification

type constructors uninterpreted *e.g.*,

- `S` $n =$ `S` $n' \rightarrow n = n'$ is forgotten
- nothing known about `List.length`

Bécassine comes into play



- **Question.** Why aren't we happy with this?

Problem 1
Avoid harmless polymorphism and higher-order

- `forall (A : Type) (l1 l2 : list A),`
  `length (l1 ++ l2) = length l1 + length l2`
- `f = g` with `f,g: nat -> nat`
  instead of $\forall$`(x : nat), f x = g x`

Problem 2
Some info. is lost during goal reification

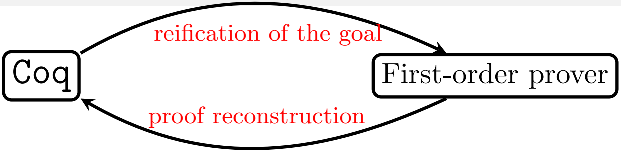
type constructors uninterpreted *e.g.*,

- `S` n = `S` $n' \rightarrow n = n'$ is forgotten
- nothing known about `List.length`

⇝ `Sniper`

- eliminates harmless polymorphism
- helps the first-order provers interpret symbols (constructors and functions)

BÉCASSINE COMES INTO PLAY



- **Question.** Why aren't we happy with this?

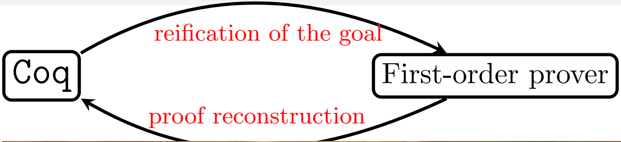
Problem 1
Avoid harmless polymorphism and higher-order

- `forall (A : Type) (l1 l2 : list A),`
  `length (l1 ++ l2) = length l1 + length l2`
- `f = g` with `f,g: nat -> nat`
  instead of $\forall$`(x : nat), f x = g x`

Problem 2
Some info. is lost during goal reification

type constructors uninterpreted *e.g.*,
- `S` $n =$ `S` $n' \to n = n'$ is forgotten
- nothing known about `List.length`

$\rightsquigarrow$ Sniper
- eliminates harmless polymorphism
- helps the first-order provers interpret symbols (constructors and functions)



- **Questio**

Problem 1
Avoid harmle
and higher-or

```
- forall (A : T
  length (l1 +
- f = g with f,g
    instead o
```

ion of the goal

First-order prover

econstruction

ppy with this?

Problem 2
Some info. is lost during goal reification

type constructors uninterpreted *e.g.*,

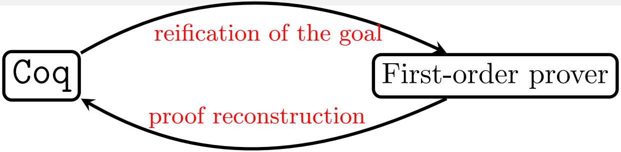
gth 12

- $\texttt{S}\ n = \texttt{S}\ n' \to n = n'$ is forgotten
- nothing known about `List.length`

ss polymorphism

er provers interpret symbols
(constructors and functions)

Bécassine comes into play



- **Question.** Why aren't we happy with this?

Problem 1
Avoid harmless polymorphism and higher-order

- `forall (A : Type) (l1 l2 : list A),`
  `length (l1 ++ l2) = length l1 + length l2`
- `f = g` with `f,g: nat -> nat`
  instead of $\forall$`(x : nat), f x = g x`

Problem 2
Some info. is lost during goal reification

type constructors uninterpreted *e.g.*,
- `S` n = `S` $n' \rightarrow n = n'$ is forgotten
- nothing known about `List.length`

⇝ **Sniper**
- eliminates harmless polymorphism
- helps the first-order provers interpret symbols (constructors and functions)

Plan

1. Coq *vs.* automated provers

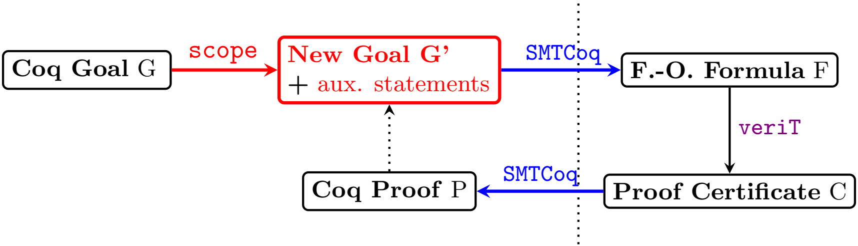
2. Sniper

Sniper is a two-fold tactic.

First Step. The tactic `scope`

- Eliminates harmless higher-order and polymorphism in the goal if needed
- Produces and proves first-order auxiliary statements in the local context (currently 6 transformations,)

Second step. The transformed goal and the auxiliary statement are sent to the SMT solver `veriT`, *via* `SMTCoq`.



Example (scope): facts about inductives datatypes

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

Example (scope): facts about inductives datatypes

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

- Constructors are pairwise disjoints ($D_{\mathtt{T}}$)

  `∀x l, [] ≠ x::l`

Example (scope): facts about inductives datatypes

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

- Constructors are pairwise disjoints ($D_{\mathtt{T}}$)
  ∀x l, [] ≠ x::l
- Constructors are injective ($I_{\mathtt{T}}$)
  ∀x y l l', x::l = y::l' → x=y ∧ l=l'

Example (scope): facts about inductives datatypes

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

- Constructors are pairwise disjoints ($D_{\texttt{T}}$)
  `∀x l, [] ≠ x::l`
- Constructors are injective ($I_{\texttt{T}}$)
  `∀x y l l', x::l = y::l' → x=y ∧ l=l'`
- Every term of this type is generated by one of the constructors ($G_{\texttt{T}}$)
  `∀(l:list A), ∃x l', l = x :: l' ∨ l = []`

Example (scope): facts about inductives datatypes

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

- Constructors are pairwise disjoints ($D_{\mathtt{T}}$)
  `∀x l, [] ≠ x::l`
- Constructors are injective ($I_{\mathtt{T}}$)
  `∀x y l l', x::l = y::l' → x=y ∧ l=l'`
- Every term of this type is generated by one of the constructors ($G_{\mathtt{T}}$)
  `∀(l:list A), ∃x l', l = x :: l' ∨ l = []`

Problem.
$T \mapsto D_T, I_T, G_T$ cannot be defined in `Coq` or in `Ltac`

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

- Constructors are pairwise disjoints ($D_{\mathtt{T}}$)
  `∀x l, [] ≠ x::l`
- Constructors are injective ($I_{\mathtt{T}}$)
  `∀x y l l', x::l = y::l' → x=y ∧ l=l'`
- Every term of this type is generated by one of the constructors ($G_{\mathtt{T}}$)
  `∀(l:list A), ∃x l', l = x :: l' ∨ l = []`

Problem.
$T \mapsto D_T, I_T, G_T$ cannot be defined in `Coq` or in `Ltac`

Solution. Gain direct access to the syntax of `Coq` terms

Example (scope): facts about inductives datatypes

Question. What should a first-order prover know about an inductive datatype `T`?

```
Inductive list (A : Type) : Type :=
| nil : list A (* [] *)
| cons : A → list A → list A. (* _ :: _ *)
```

- Constructors are pairwise disjoints ($D_{\mathtt{T}}$)
  `∀x l, [] ≠ x::l`
- Constructors are injective ($I_{\mathtt{T}}$)
  `∀x y l l', x::l = y::l' → x=y ∧ l=l'`
- Every term of this type is generated by one of the constructors ($G_{\mathtt{T}}$)
  `∀(l:list A),∃x l', l = x :: l' ∨ l = []`

Problem.
$T \mapsto D_T, I_T, G_T$ cannot be defined in `Coq` or in `Ltac`

Solution. Gain direct access to the syntax of `Coq` terms

⇝ **Use MetaCoq**
reification of Coq in Coq

+ quoting/unquoting mechanisms

How does the transformations work?

How does the transformations work?

1. Generation of the reified statements in `MetaCoq` (*e.g.*, constructors are injective)
2. Unreify these statements
3. Proof of these statements with `Coq` regular tactics (`Ltac`)
4. The statements are now in the local context

How does the transformations work?

How does the transformations work?

1. Generation of the reified statements in `MetaCoq` (*e.g.*, constructors are injective)
2. Unreify these statements
3. Proof of these statements with `Coq` regular tactics (`Ltac`)
4. The statements are now in the local context

Currently implemented transformations

- Make explicit the semantics of symbols
- Eliminate higher-order equalities
- Eliminate prenex polymorphism

Example

Action of `scope` on a goal

`A : Type`

$\texttt{forall (l : list A) (n : nat), length A l} = \texttt{S n} \rightarrow \texttt{l} \neq [\,]$

Example

Action of `scope` on a goal

1. inductive datatypes

`A` : `Type`

1: `forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'`

1: `forall B (x : B) (l : list B), [ ] ≠ x :: l`

1: `forall (n n': nat), S n = S n' → n = n'`

1: `forall (n : nat), O ≠ S n`

`forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]`

Example

Action of `scope` on a goal

1. inductive datatypes
2. definitions

A : Type
1: forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'
1: forall B (x : B) (l : list B), [ ] ≠ x :: l
1: forall (n n': nat), S n = S n' → n = n'
1: forall (n : nat), 0 ≠ S n
2: length = (fun B ⇒ fix length l := match l with … end)

---

forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]

Example

Action of `scope` on a goal

1. inductive datatypes
2. definitions
3. expansion

```
A : Type
1: forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'
1: forall B (x : B) (l : list B), [ ] ≠ x :: l
1: forall (n n': nat), S n = S n' → n = n'
1: forall (n : nat), O ≠ S n
2: length = (fun B ⇒ fix length l := match l with … end)
3: forall B l, length B l = (fun B ⇒ fix length l := match l with … end) B l
```

```
forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]
```

Example

Action of `scope` on a goal

1. inductive datatypes
2. definitions
3. expansion
4. fixpoints

```
A : Type
1: forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'
1: forall B (x : B) (l : list B), [ ] ≠ x :: l
1: forall (n n': nat), S n = S n' → n = n'
1: forall (n : nat), 0 ≠ S n
2: length = (fun B ⇒ fix length l := match l with ... end)
3: forall B l, length B l = (fun B ⇒ fix length l := match l with ... end) B l
4: forall B l, length B l = match l with ... end
```

```
forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]
```

Example

Action of `scope` on a goal

1. inductive datatypes
2. definitions
3. expansion
4. fixpoints
5. elimination of pattern matching

```
A : Type
1: forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'
1: forall B (x : B) (l : list B), [ ] ≠ x :: l
1: forall (n n': nat), S n = S n' → n = n'
1: forall (n : nat), 0 ≠ S n
2: length = (fun B ⇒ fix length l := match l with … end)
3: forall B l, length B l = (fun B ⇒ fix length l := match l with … end) B l
4: forall B l, length B l = match l with … end
5: forall B, length B [ ] = 0
5: forall B (l : list B) (x : B), length B x :: l = S (length B l)
```

```
forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]
```

Example

Action of `scope` on a goal

1. inductive datatypes
2. definitions
3. expansion
4. fixpoints
5. elimination of pattern matching
6. applied polymorphic hypotheses

```
A : Type
1: forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'
1: forall B (x : B) (l : list B), [ ] ≠ x :: l
1: forall (n n': nat), S n = S n' → n = n'
1: forall (n : nat), 0 ≠ S n
2: length = (fun B ⇒ fix length l := match l with ... end)
3: forall B l, length B l = (fun B ⇒ fix length l := match l with ... end) B l
4: forall B l, length B l = match l with ... end
5: forall B, length B [ ] = 0
5: forall B (l : list B) (x : B), length B x :: l = S (length B l)
6: length A []= 0
6: forall (l : list A) (x : A), length x :: l = S (length A l)
------------------------------------------------------------------
  forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]
```

Example

Action of scope on a goal

1. inductive datatypes
2. definitions
3. expansion
4. fixpoints
5. elimination of pattern matching
6. applied polymorphic hypotheses

```
A : Type
1: forall B (x y : B) (l l' : list B), x :: l = y :: l' → x = y ∧ l = l'
1: forall B (x : B) (l : list B), [ ] ≠ x :: l
1: forall (n n': nat), S n = S n' → n = n'
1: forall (n : nat), 0 ≠ S n
2: length = (fun B ⇒ fix length l := match l with ... end)
3: forall B l, length B l = (fun B ⇒ fix length l := match l with ... end) B l
4: forall B l, length B l = match l with ... end
5: forall B, length B [ ] = 0
5: forall B (l : list B) (x : B), length B x :: l = S (length B l)
6: length A []= 0
6: forall (l : list A) (x : A), length x :: l = S (length A l)
----------------------------------------------------------------
  forall (l : list A) (n : nat), length A l = S n → l ≠ [ ]
```

Conclusion and Future Work

- General methodology: small transformations from a subset of `Coq` logic to FOL
- Proof of concept: six transformations combined in a tactic (`snipe` = `scope` + `verit`) which calls an external SMT solver.

These transformations are independent from `SMTCoq`!

- General methodology: small transformations from a subset of `Coq` logic to FOL
- Proof of concept: six transformations combined in a tactic (**`snipe`** = **`scope`** + **`verit`**) which calls an external SMT solver.

These transformations are independent from `SMTCoq`!

In the Future.

- More complex transformations: (simple) **dependent types**, dependent pattern matching...
- Add user-defined tactics
- **Benchmarks**

+ improving the performance of our tactic

Try Sniper!

https://github.com/smtcoq/sniper

- General methodology: small transformations from a subset of **Coq** logic to FOL
- Proof of concept: six transformations combined in a tactic (**snipe** = **scope** + **verit**) which calls an external SMT solver.

These transformations are independent from SMTCoq!

In the Future.

- More complex transformations: (simple) **dependent types**, dependent pattern matching...
- Add user-defined tactics
- **Benchmarks**

  + improving the performance of our tactic

Try Sniper!

https://github.com/smtcoq/sniper