A Toolbox for Mechanised First-Order Logic

Johannes Hostert, Mark Koch, and Dominik Kirst

Saarland University, Saarland Informatics Campus, Germany

We report on three tools developed for the evolving mechanisation of first-order logic [1, 2, 5, 4] as part of the Coq Library of Undecidability Proofs [3]. The employed deep embedding of first-order logic is based on a de Bruijn encoding of the syntax, an embedding into Coq's logic to define Tarski semantics, and inductive predicates representing deduction systems. While the de Bruijn approach is well-suited for the mechanisation of meta-theoretic concepts (parallel substitution, weakening, etc.), it is rather unhandy for working with concrete formulas, models, or derivations (which is why compromises have been proposed [7]).

Consider the commutativity of addition in Peano arithmetic (PA) as an example. Expressed in Coq's logic for the type $\mathbb{N}$, it states that $n + m = m + n$ for all $n, m : \mathbb{N}$ and is proven by an easy induction and a modest amount of rewriting. In first-order logic, however, some unwanted overhead is introduced:

1. Defining the first-order formula $\varphi := \forall\forall\, x_1 + x_0 = x_0 + x_1$ is bearable but as soon as formulas feature more complex quantifier nesting, chasing de Bruin indices will become unfeasible and prone to error. Already formulating the induction scheme $\lambda\psi.\, \psi[0] \to (\forall\, \psi \to \psi[Sx_0; \uparrow]) \to \forall\, \psi$ of PA is not trivial.
2. If we want to verify φ semantically, i.e. show $\mathsf{PA} \vDash \varphi$, we assume a model $\mathcal{M}$ of PA and show $\mathcal{M} \vDash \varphi$. Now at least the recursive definition of $\mathcal{M} \vDash \varphi$ evaluates to a statement in Coq's logic, namely that $x +^{\mathcal{M}} y = y +^{\mathcal{M}} x$ for all $x, y : \mathcal{M}$. But when applying the induction scheme to proceed with the proof, the user is still required to supply a concrete induction instance ψ by hand.
3. Even more cumbersome, when verifying φ deductively, i.e. showing $\mathsf{PA} \vdash \varphi$, no support from Coq's logic can be used and the proof rules defining $\vdash$ need to be applied one by one. In particular the rules for quantifiers and rewriting are tedious to use by hand since they introduce substitutions.

Our tool support for these three problems is addressed in the following three sections, respectively.

Syntax Interface: HOAS Input Language In order to avoid the intricacies with de Bruijn indices when defining concrete formulas, we employ a special input language using higher order abstract syntax. The idea is to define quantified formulas through functions (for example $\lambda xy : \mathbb{N}.\, x + y = y + x$), which allows us to rely on the existing binder mechanism of Coq. Together with notations, this enables us to define formulas in a more natural way, namely by `<< ∀' x y, x + y == y + x`. The notation `<<` translates the HOAS formula into the usual de Bruijn format, such that we end up with exactly the same φ as in (1) above.

Semantic Proofs: Reification Tactic Our reification tactic works by recursively inspecting the Coq term we want to reify, while matching known constructs to their reified representations. Along with this, we generate a proof that the reification is correct. Our tactic is extensible, so that users can add support for their own custom syntax by registering a type class, which is later invoked by the reification framework. MetaCoq [8] is used to deeply inspect Gallina's internal AST, which is necessary to properly reify bound variables, while not carrying the overhead involved with a custom Coq plugin written in OCaml.

In the example below, we employ a semantic reformulation of PA induction for a given model $\mathcal{M}$

$$\forall P : \mathcal{M} \to \mathsf{Prop}.\, \mathsf{reifiable}\, P \to P\, 0 \to (\forall n : \mathcal{M}.\, P\, n \to P\, (S\, n)) \to \forall n : \mathcal{M}.\, P\, m$$

which is analogous to the induction scheme for $\mathbb{N}$, except for the precondition $\mathsf{reifiable}\, P$ expecting a formula ψ reifying P. For concrete P, this condition can be discharged using our `represent` tactic. This way, the user can perform inductive proofs in the first-order model $\mathcal{M}$, just like when working with $\mathbb{N}$ in Coq's logic.

```
Lemma add_comm a b : a +^M b = b +^M a.
Proof.
elim a using PA_induction.
- represent.
  (* Goal here was: *)
  (* reifiable (fun a => a +^M b = b +^M a) *)
- now rewrite add_zero_l, add_zero_r.
- intros a' IH. now rewrite add_succ_l, add_succ_r.
Qed.
```

```
Lemma add_comm a b : a +^M b = b +^M a.
Proof.
elim a using PA_induction.
- exists ($0 + $1 == $1 + $0).
  exists (fun _ => b). (* environment, s.t. 0 ↦ b *)
  intros d. cbn. rewrite D_eq_ext. now split.
- now rewrite add_zero_l, add_zero_r.
- intros a' IH. now rewrite add_succ_l, add_succ_r.
Qed.
```

The examples above are taken from the demo files. We also provide a detailed documentation.

Deductive Proofs: Proof Mode We observed three major obstacles that occur regularly when syntactically verifying first-order formulas in a deduction system:

1. Proving statements at the level of individual deduction rules by hand is very tedious. This gets even worse as soon as rules require substitutions that need to be handled and simplified.
2. While our HOAS input language hides the de Bruijn indices for definitions, during the proof, they are visible and can get confusing, especially for larger formulas.
3. It can get challenging to keep track of all the formulas in the context.

Inspired by the Iris proof mode [6], we developed a similar proof mode for first-order logic to alleviate those issues. By calling the tactic `fstart` a custom goal view is activated, where de Bruijn indices are replaced by named binders and the context is displayed in a Coq like fashion with hypothesis names. Thus, the underlying de Bruijn encoding is completely hidden from the user. We also developed custom tactics like `fintro`, `fapply`, or `frewrite` that behave just like their Coq counterparts, handling all of the substitutions and deduction rules internally. Hypotheses in the context can be specialized, applied to each other and destructed with introduction patterns. Below is a proof of commutativity of addition using the proof mode (on the left) as well as the custom goal view that is visible for the user at point $\star$ (on the right):

```
Lemma add_comm : PA ⊢ << ∀' x y, x + y == y + x.
Proof.
  fstart. fapply ((ax_induction (<< Free x, ∀' y, x+y == y+x))).
  - fintros "x". frewrite (ax_add_zero x).
    frewrite (add_zero_r x). fapply ax_refl.
  - fintros "x" "IH" "y". (* ⋆ *) frewrite (add_succ_r y x).
    frewrite <- ("IH" y). frewrite (ax_add_rec y x).
    fapply ax_refl.
Qed.
```

```
 x, y : term
 ──────────────────────────────
(1/1)
  PA
  "IH" : ∀ x0, x[↑] + x0 == x0 + x[↑]
 ──────────────────────────────
  (σ x + y == y + σ x)
```

We implemented the custom goal view by defining aliases for deduction, list cons and variables with names as extra arguments and printing them using notations. This way, we can control when to show the notations and maintain computational equality with the original goal, such that all deduction rules can still be applied. Due to the modular architecture, the proof mode should also scale to other variants of (first-order) logic. We provide demo files and a manual with all available tactics and further implementation details.

Acknowledgements The concrete approach to implement a separate HOAS input language was suggested by Cyril Cohen to Yannick Forster for a deep embedding of the lambda calculus and then adopted to first-order logic. Although our library on first-order logic does not use Autosubst 2 [9] support anymore, the design in principle still follows earlier versions using this tool.

References

[1] Yannick Forster, Dominik Kirst, and Gert Smolka. On synthetic undecidability in Coq, with an application to the Entscheidungsproblem. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 38–51, 2019.

[2] Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory: Extended version. *Journal of Logic and Computation*, 31(1):112–151, 2021.

[3] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. A Coq library of undecidable problems. In *CoqPL 2020 The Sixth International Workshop on Coq for Programming Languages*, 2020.

[4] Dominik Kirst and Marc Hermes. Synthetic undecidability and incompleteness of first-order axiom systems in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[5] Dominik Kirst and Dominique Larchey-Wendling. Trakhtenbrot's theorem in Coq. In *International Joint Conference on Automated Reasoning*, pages 79–96. Springer, 2020.

[6] Robbert Krebbers, Amin Timany, and Lars Birkedal. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 205–217, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Olivier Laurent. An anti-locally-nameless approach to formalizing quantifiers. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 300–312, 2021.

[8] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 64, 06 2020.

[9] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 166–180, 2019.