A first-order logic proof mode

Mark Koch

December 12, 2020

Contents

1 Introduction

The goal of this project is to develop a proof mode for first-order logic in Coq, inspired by the Iris proof mode. This prototype allows the user to prove statements in a first-order deduction system using many Coq-like tactics while also aiding with context management. It allows significantly shorter proofs (usually 2-3 times) that are also more readable.

My main contributions are contained in the Coq files `ProofMode.v` and `Theories.v`. Example use cases can be found in the demo files `DemoPA.v` for Peano arithmetic and `DemoZF.v` for Zermelo–Fraenkel set theory.

The first part of this document contains a brief users guide describing the setup and the available tactics. The second part is a more in depth description of how the proof mode works internally.

2 User guide

2.1 Setup

To use the proof mode, you need to perform some setup steps beforehand. We follow the example for Peano arithmetic in `DemoPA.v` which can easily be adapted to other systems.

First, you need to prove that your signatures have equality deciders. This should be trivial:

```
Instance eqdec_funcs : EqDec PA_funcs_signature.
Proof. intros x y; decide equality. Qed.

Instance eqdec_preds : EqDec PA_preds_signature.
Proof. intros x y. destruct x, y. now left. Qed.
```

If you use custom definitions (not notations!) inside your formulas (for example `zero` in the PA demo or `subst` in the ZF Demo), you need to register them for the proof mode. Simply override the tactics `custom_fold` and `custom_unfold` with the corresponding `fold` and `unfold` calls like this:

```
Ltac custom_fold ::= fold zero in *; ...
Ltac custom_unfold ::= unfold zero in *; ...
```

Also if you have custom simplification lemmas, for example substitution invariance for specific terms (like `numeral_subst_invariance` in the PA

demo), or any other simplifications that might be helpful in your domain, you can register them by overriding the tactic `custom_simpl`:

```
Ltac custom_simpl ::= try rewrite !numeral_subst_invariance; ...
```

If you want to use rewriting with equalities, you need to show that your signatures satisfy the `Leibniz` type class:

```
Instance PA_Leibniz : Leibniz PA_funcs_signature PA_preds_signature.
```

This requires providing the equality predicate and the minimal set of axioms that is needed for proving the following Leibniz rule:

```
leibniz A phi t t' : Axioms <<= A -> A ⊢ eq t t' -> A ⊢ phi[t..] -> A ⊢ phi[t'..]
leibniz_T T phi t t' : Axioms_T ⊑ T -> T ⊩ eq t t' -> T ⊩ phi[t..] -> T ⊩ phi[t'..]
```

Note that you can already use the other proof mode tactics to prove this.

TODO
This could possibly be automated. Then the user would only need to prove the auto-generated axioms.

2.2 Overview

The proof mode is invoked with the tactic `fstart`. This will change the way your goal looks like:

- The different hypotheses in the context will be displayed one underneath the other just like the Coq context. Also each hypothesis gets a name that can later be used to refer to it.
- The de Bruijn indices get replaced with names to make goals and hypotheses more readable.

If you want to leave the proof mode again, simply call the `fstop` tactic. But note that your custom hypothesis names will get lost and cannot be recovered again.

Remark
It is not necessary to start the proof mode if you only want to use the custom tactics described later. These work out of the box on any goal, regardless if the proof mode is started or not. For simple examples this may already be enough, but for more complex proofs, starting the proof mode is recommended.

2.3 Input syntax

The proof mode supports an easier way of writing down formulas that avoids the use of de Bruijn indices. Instead you can simply used names as binders like you are used to from Coq. For a demo of this, look at the last section of the PA demo file.

Before using the input syntax, you need define notations for your functions and predicates as `bFunc` and `bAtom`:

```
Require Import Hoas.

Notation "x '==' y" := (bAtom Eq (Vector.cons bterm x 1
                                  (Vector.cons bterm y 0
                                  (Vector.nil bterm))))
                       (at level 40) : hoas_scope.

Notation "'σ' x" := (bFunc Succ (Vector.cons bterm x 0
                                 (Vector.nil bterm)))
                    (at level 37) : hoas_scope.
...
```

Important
Note that you are required to put the type `bform` in the `Vector.cons` and `Vector.nil` call for the coercions to work. Also the notations must be put into the scope `hoas_scope`.

To use the input syntax, prefix the formula with `<<` and add the `hoas` scope delimiter:

```
Lemma division_theorem :
   FAI ⊢ <<(∀' x y, ∃' q r, (x == r ⊕ q ⊗ σ y) ∧ r ≤ y)%hoas.
Proof.
```

Note that the quantifiers must have a prime behind them.

2.4 Working with the context

There are many different ways you can refer to a hypothesis in the context:

- If you started the proof mode, introduced hypotheses have names in the context. You can simply give the according string to a tactic to refer to one of these hypotheses. This is the recommended way to work with the context.

  Example: `fapply "H3". frewrite "H".`

- Alternatively you can directly give a formula that is in the context. This is useful when working with named axioms.

  Example: `fapply ax_sym. frewrite (x == y).`

- You can also refer to a hypothesis by its index in the context. This is useful when the proof mode is not active.

  Example: `fapply 3.`

  Important
  If parts of your context are folded behind an identifier (like `FA ⊢ ...` in the Peano example), this will not work. The only way is to unfold or give the term directly.

- The Coq context itself is also accessible. If you have a hypothesis `H : C ⊢ phi` you can pass `H` as an argument.

2.5 Tactics

2.5.1 `fintro`

The `fintro` tactic works analogously to the Coq `intro` tactic. There is also the `fintros`-variant for introducing multiple things at once. There are a few things to note:

- If you introduce a forall-quantifier, you can give an identifier name as a string. If you do not give an argument or put the string `"?"`, a new name is automatically generated.

  Example: `fintro "x". fintro. fintro "?".`

- When introducing implications you can also give a name to the hypothesis as a string. Note that this has only an effect if the proof mode is active.

- `fintros` can take multiple strings. If you do not give any arguments, everything will get introduced.

  Example: `fintros "x" "H". fintros.`

- The tactics also support intro patterns similar to Coq to recursively destruct conjunctions, disjunctions and existentials.

  Example: `fintros "[H1 ?]" "[H|[x H2]]" "[|]" "[ ]".`

Important
Intro pattern parsing is very limited at the moment. You are not allowed to put any extra spaces inside the pattern or it will not be recognized. For example `"[H1|H2]"` is valid, while `"[H1 | H2]"` is not. If you leave out names, the only valid forms are `"[H ]"`, `"[ H]"`, `"[ ]"`, `"[H|]"`, `"[|H]"`, `"[|]"`. This restriction also applies to every other tactic that uses intro patterns.

2.5.2 `fapply`

The `fapply` and `feapply` tactic works just like in Coq. You can specialize but the tactic is also able to find the correct instantiations of quantifiers automatically. Additional premises may be added as goals for the user to prove.

Example: `fapply ax_sym. feapply ("H" x z). feapply ("H1" "H2").`

You can also apply in hypotheses with intro pattern support. Application of equivalences is also supported.

Example: `feapply ax_pair in "H1" as "[H1|H1]".`

Important
The `fapply` tactic may not work, if the goal already contains evars. In that case you should use `feapply` instead.

2.5.3 `frewrite`

For rewriting to work, make sure that you followed the corresponding setup steps. You need an assumption of the form `C ⊢ x == y` and can rewrite in the goal. Quantified assumptions are also supported, but you need to give the arguments explicitly.

Example: `frewrite "H". frewrite <- (ax_sym zero x).`

Remark
Rewriting under quantifiers is supported but you should be aware of the following fact: If you know that `x == y` you actually cannot rewrite in something like ∀ `x == z`. Because of the quantifier you need shifted occurence of `x` like ∀ `x`[↑] `== z`.

2.5.4 `fdestruct`

Destructs conjunctions, disjunctions and existentials with intro pattern support. If no pattern is given, the hypothesis is maximally destructed by auto generated names.

Example: `fdestruct "H". fdestruct "H" as "[x H]".`

2.5.5 `fassert`

Behaves like the Coq tactic `assert` and supports intro patterns.

Example: `fassert (x == y). fassert (a ∨ b) as "[A|B]" by tac.`

Important
If you use the `by` syntax with multiple tactics you need to put the whole tactic inside parenthesis.

Example: `...; (fassert t by tac1; tac2; tac3); ...`

2.5.6 `fspecialize`

Can be used to specialize formulas.

Example: `fspecialize (H x y "H3"). fspecialize H with a, b.`

2.5.7 `ctx`

The `ctx` tactic solves goals that are contained in the context.

2.5.8 Classical logic

In a classical system, there is a tactic `fclassical` that performs a case distinction on a formula. You get two cases, one with the formula and one with the negated formula in the context. Also supports intro patterns.

Example: `fclassical phi. fclassical (a ∧ b) as "[A B]" "H".`

The tactic `fcontradict` allows for proof by contradiction. It puts the negated original goal in the context and leaves the user to prove falsity.

Example: `fcontradict phi. fcontradict (a ∧ b) as "H".`

2.5.9 Other tactics

There are some other tactics that behave just like their Coq counterpart:

Example: `fexfalso. fsplit. fleft. fright. fexists x`

3 Behind the scenes

3.1 Proof Mode

My goal for the proof mode was to have the special notation only active inside the Coq goal. Deductions in the Coq context should be displayed in the usual `C ⊢ phi` notation. Therefore simply overriding the `prv` notation to get the horizontal bar is not sufficient. The main trick is to define a function

```
pm C phi := prv C phi
```

that acts like an alias for the `prv` type. By defining the proof mode notation only for `pm`, we can manually control when to use it while preserving computational equivalence. For the same reason there are aliases for `cons` and `nil` on formula lists

```
cnil := @nil form
ccons (s : string) phi C := @cons form phi C
```

They are used to print hypotheses one under another in the context. Note the extra string argument `s` in `ccons`. It is used to give names to the hypotheses. Additionally there is an alias for complete formula lists

```
cblackbox (C : list form) := C
```

that is wrapped around lists that are not syntactically known to be nil or cons. This is only used to indent these lists nicely in the context. For deduction on theories there are also functions `tpm`, `tnil`, `tcons` and `tblackbox` that are defined in the same way.

The tactic `fstart` then simply replaces the `prv` or `tprv` with `pm` or `tpm` and updates the context accordingly so that the notation gets applied. The `fstop` tactic unfolds all of these functions to get back to the original type.

Starting the proof mode also replaces de Bruijn indices with named binders. This is done again with the usual trick of defining aliases

```
named_quant op (x : string) phi := quant op phi
named_var n (x : string) := var n
```

where `x` is the display name of the variable. The tactic `update_binder_names` ensures that all names are given correctly and should be called if the quantifiers in the goal or the context change.

TODO

When creating the context, automatically generated names are used. Especially when leaving and reentering the proof mode it might be nice to allow the user to pick names themselves. Maybe something like `fstart with "Hx" "H" "H'"`?.

More importantly, the variable names are also given automatically and change (!) after an intro. This might be very confusing for the user but remembering the names across tactic calls seems difficult, especially considering the compatibility layer. But I think the current solution is still better than de Bruijn indices.

3.2 Tactic compatibility

As described in the last section there are four different types of goals our tactics should be able to work with: `prv`, `tprv`, `pm` and `tpm`. It would be very inconvenient if all tactics would need to be written in a way that handles all of these different types. Especially maintaining the proof mode aliases would be tedious. Therefore there are some compatibility levels that should make writing tactics more easy.

The most important design decision is, that all tactics available to the user assume that the proof mode is *inactive*. Therefore they only have to work on goals of the form `prv` or `tprv` (notable exceptions are the `fintro` and `fdestruct` tactic, because they need to alter the context). To make tactics compatible with the proof mode, there is the higher order tactic `make_compatible` that takes a tactic and lifts it to work with the proof mode.

If the proof mode is active, `make_compatible` stops it while remembering the context, executes the given tactic and restores the proof mode to the state it has been before. Also tactics lifted by `make_compatible` get an extra argument where the current context is filled in. This way, tactics get

access to the hypothesis names that are only present if the proof mode is active.

Note that `make_compatible` only works, if the given tactic does not alter the context!

By adding this compatibility layer, tactics now only need to differentiate between deduction on lists or theories. To make this as painless as possible, I defined a type class `DeductionRules` that includes all of the first-order deduction rules and showed that `tprv` also satisfies them (see `Theories.v`). There is also a type class for classical deduction rules and Weakening that is satisfied by `tprv`.

The advantage is, that you can now apply rules like `II`, `AllE`, `Weak` etc. independent of the concrete system you are in and tactics do not need to differentiate these cases that would result in a lot of duplicated code. There is also the additional benefit that this can easily be extended, for example to deduction on a tuple of a list and a theory.

On top of that there are the tactics `get_context` and `get_form` that return the context or the formula of the goal or of a Coq hypothesis. This avoids matching on the goal. For the same reason there are the tactics `assert_comp` and `enough_comp` that can be used in place of the original Coq tactic.

Example: `(assert_comp (phi --> psi) as H by tac)` asserts `H : C ⊢ phi -> psi` or `H : C ⊩ phi -> psi` depending on the current context `C` and the deduction system.

3.3 General tactic design

Most tactics need to interact with a hypothesis in some way or another. As mentioned before, hypotheses can be referenced by a name, a context index, a Coq hypothesis name or by the formula type. Therefore the first step is always to move the referenced formula `phi` into the Coq context by asserting a new hypothesis `H : C ⊢ phi`, where `C` is the current context. Working with a Coq hypothesis has the benefit, that we are isolated from the goal and can apply tactics directly in `H`. After everything is finished, `H` gets cleared again.

Another common pattern is the handling of specialization of hypotheses in tactic arguments. For example you might want to give arguments before applying a function:

```
frewrite (ax_trans x y z)   or   fapply ("H3" a "H")
```

Supporting this notation with an unconstrained number of arguments is quite difficult in general. Tactics with variable numbers of arguments are possible

with continuation passing style, but do not support the notation above. Thus the best solution seems to be defining a tactic notation for each argument length you want to support, effectively limiting the possible maximum number of arguments. But by covering the range anyone could reasonably use in practice, this should be acceptable.

The tactic notation then calls the "real" tactic (`frewrite'`, `fapply'`, etc.) that accepts a list of all the arguments. Each tactic then internally uses the `fspecialize` tactic that performs the necessary specializations and lookups in the context on the Coq hypothesis `H`.

Also noteworthy is the tactic `simpl_subst` that can simplify substitutions in the goal or a Coq hypothesis. This is also used in many places.

3.4 Intros and intro patterns

In contrast to most of the other tactics, the `fintro` tactic needs to be aware if the proof mode is active or not, because it changes the context. Also introduction of variables behaves differently, depending on whether you do deduction on formula lists or theories:

With lists, every context is finite which allows us to use a different notion of forall-introduction. Instead of shifting the entire context to make variable 0 free, we can find a new variable that is not used and avoid the shifting. The lemma used to achieve this gives us a term instead of the actual variable which has the benefit, that the handling of the introduced variables can happen on then Coq level. Also it seems more natural to introduce a term `x`, instead of a variable `$n`.

Avoiding the shift is not possible when using theories as they can contain infinitely many formulas. There we must use the theory variant of the vanilla forall-introduction rule. The variable `$0` is again replaced with a Coq variable `x` using a substitution. I also want to mention, that the `mapT (subst_term ↑)` call cannot be simplified computationally. Instead, the tactic `simpl_context_mapT` is used to evaluate the call in a syntactic way by repeatedly applying a rewrite lemma.

Another critical fact is that because of the purely syntactic matching of Ltac, we cannot look behind defined constants. But if the user has a definition that hides a forall-quantifier (for example the subset predicate in the ZF demo) we would not be able to detect it using matches. That is one of the reasons why the user is required to register those definitions by overriding the `custom_fold` and `custom_unfold` tactic. This way, `fintro` can unfold if it gets stuck and check if there are hidden quantifiers. This trick is also used by some other tactics.

Another problem that occurred when developing the `fintro` tactic was how to turn a variable name, given by the user as a string, into a fresh Coq identifier. Sadly there is no easy way to turn a Gallina string into an Ltac string that can be used to create a fresh name. The first solution I came up with was to split up the introduction of variables and hypotheses into two different tactics so that one can take Coq identifiers as an argument and the other strings. The main problem is that this does not work, if the variable name occurs inside the pattern, as is the case with the destruction of an existential. Also we want an `fintros` tactic that can intro multiple things at once. This would need be implemented using tactic notations that figure out which version of `fintro` to call depending on the argument, leading to an exponential number of tactic notations that would be required.

Later I found a solution by Tej Chajed developed for the Iris proof mode that translates strings into Coq identifiers[1]. This uses Ltac2 and converts between the different string representations, but seems a bit hacky and might break down in future versions of Coq[2].

Right now this solves all of the problems but the other solution is also supported, so you can intro either with strings or identifiers.

The `fintro` tactic is also the main driver of intro patterns. Every other tactic that supports intro patterns internally calls `fintro`. For example if your goal is `C ⊢ psi` and you want to call `fdestruct` on a context formula `"H" : phi`, the formula will be moved to the goal and `fintro` will be called on `C ⊢ phi → psi` using the user given intro pattern.

The implementation of the intro patterns is straightforward. They get parsed into the inductive type `intro_pattern` and then recursively processed. For example if you intro on `phi ∧ psi → G` using the pattern `"[P1 P2]"`, the goal gets turned into `phi → psi → G` and we intro recursively using `"P1"` and `"P2"`.

TODO
The parsing could be improved. At the moment additional white spaces cannot be handled. See also the note in 2.5.1.

3.5 Application

The main driver behind the `fapply` and `feapply` tactic is a tactic called `fapply_without_quant`. It takes a Coq hypothesis `H : C ⊢ p1 -> ...`

[1]The code can be found here `https://gitlab.mpi-sws.org/iris/string-ident`
[2]For more detailed discussion see here `https://github.com/coq/coq/issues/7922`

`-> pn -> g` and solves the goal `C ⊢ g` by applying `H` and leaving the premises `C ⊢ p1`, ..., `C ⊢ pn` for the user to prove.

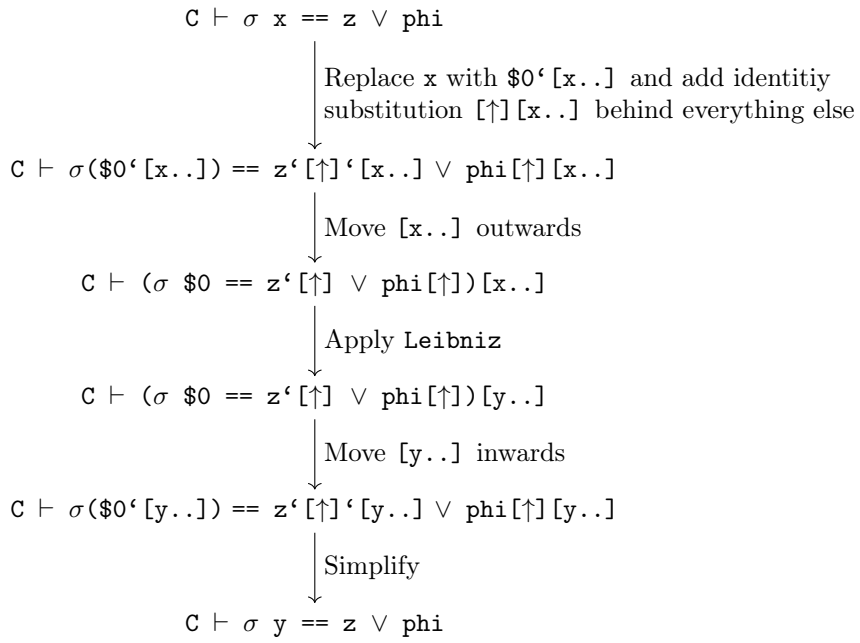
The actual `fapply` and `feapply` tactic simply call this tactic and perform some bookkeeping around it. One noteworthy trick is that quantifiers inside the formula to apply with are instantiated with evars. This way they can automatically be unified and the user does not need to give the arguments explicitly.

Ideally the differentiation between `fapply` and `feapply` should be just like in Coq. But `fapply` also uses the evars trick to handle unification and behaves exactly like `feapply` internally. The current heuristic is that `fapply` fails, if the goal still contains evars after the application. This obviously fails in cases, were the goal already contained evars before the tactic was called.

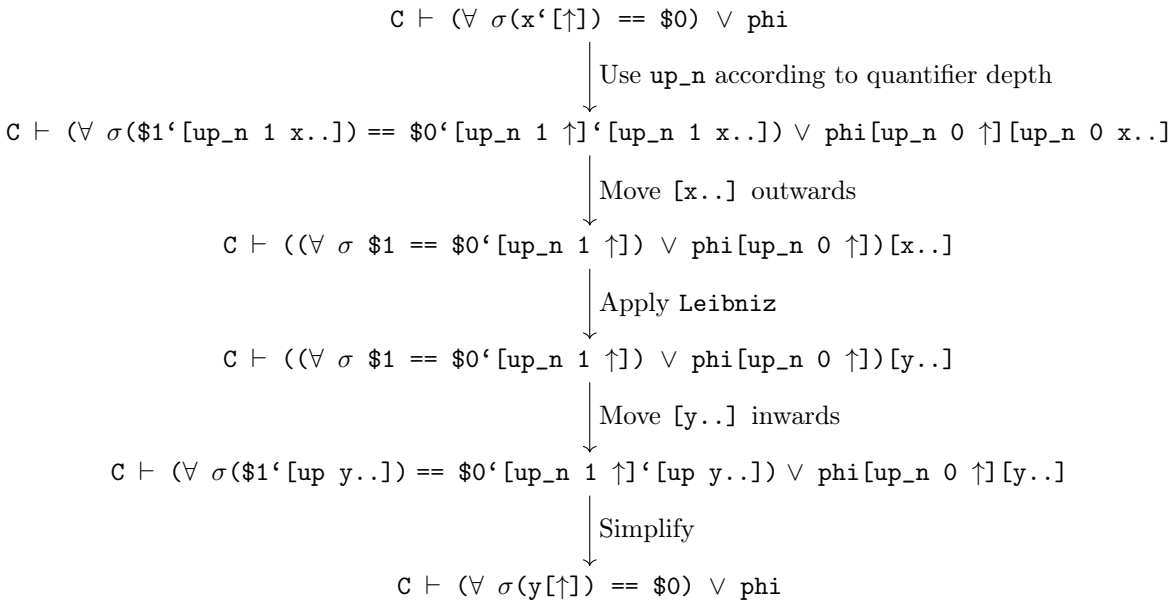
TODO
This should be fixed. One would need to remember the names of the evars that `fapply` created and only check for the existence of those in the goal.

3.6 Rewriting

The `frewrite` tactic is maybe the most complicated one. It supports rewriting under equality with the Leibniz rule and rewriting with logical equivalences (but currently not under quantifiers). The equality predicate and proof of the Leibniz rule are given by the user through the type class `Leibniz`. While the rewriting with equivalences is performed through repeated application of lemmas for each operator, I tried a different syntactic approach for the equality rewriting. The general idea can be illustrated with the following example in PA where `x` should be replaced with `y`:



Sadly this scheme does not work, if the term to rewrite is behind a quantifier, because a substitution `s` gets turned into `up s` inside the scope of a quantifier. To address this, I generalize the idea by using a function `up_n` that gives the iterated application of `up`:



Note that in the case of no quantifiers, we only use `up_n 0 s` which is equal to `s`, so we get essentially the same as in the first example.

TODO
There are some things that could be improved regarding the `frewrite` tactic:

- Support rewriting in hypotheses
- Support equivalence rewriting under quantifiers
- Support `frewrite ... at n`
- If you want to rewrite under quantifiers, the term needs to be already shifted (in the example above it needs to be `x‘[↑]` under the all quantifier). In some cases this is irrelevant because the term is substitution invariant (for example `num` in PA). Provide a way for the user to register such invariance lemmas and apply them accordingly.
- The evar trick from `fapply` does not work here because I use syntactic matching to find occurrences of the rewrite variable. Thus the user always needs to give the instantiation for variables. But improving this might be tricky...