First-Order Reification with MetaCoq

Johannes Hostert

December 12, 2020

Contents

1 Scope and overview

The Programming Systems group is working on first-order logic in Coq. They define the syntax of first-order logic in a reusable way that allows one to quickly define new instances (i.e. first-order peano arithmetics or ZF set theory). From the syntax they can then also automatically derive a semantics by assuming a general model, which has functions being the in-model representations of the syntactic functions defined as part of the theory. Proofs about the theory can then be done by proving things in the model.

For this, it is often necessary to show that certain statements or terms about objects in the model are "representable", meaning that they also are statements in the original theory of which we have a model. We present here a MetaCoq libary for automatically finding the syntactic representations of (semantic) statements about the model, along with proofs that they are indeed correct. The library operates on the syntactic representation of the semantic statements of the model and can be extended to reflect/reify other terms not covered by the standard syntactic constructions.

We call a statement representable if there is an environment and a syntactic first-order form so that evaluating the syntactic form in the environment yields something equivalent to the original term. A similar definition is used for terms. When constructing the syntactic form, we will call it reification.

2 Applicable models of a first-order theory

The system used by the PS group defines evaluation function for the syntax of a first-order theory. These evaluation functions operate on a generic type which has an `interp`, which provides an interpretation for specific arithmetical and logical connectives. For basic logical connectives (and, or, forall-quantification) which are independent of the specific first-order theory, the standart Coq logical connectives are used.

Theories are defined by giving a type of "function symbols", from which terms can be built (for PA, these are O, S, $+$ and $\cdot$), as well as "relation symbols", which create atomic properties (e.g. $\in$ for ZF, or $=$ for PA). Both have an associated arity and induce functions `Vector.t <arity> term → form/term` dependingly.

When assuming a model, one assumes a type `D`, an `interp D`, which provides a semantic function for each of the form/term symbols, by giving a function `Vector.t <arity> D → D/Prop`, and a proof that this model fulfills all the axioms of one's theory. These functions also define operations on the points of `D`. Now, it is significantly simpler to prove statements about the points of `D`, partially constructed using these operations derived from the interpretation of syntactic first-order symbols.

However, since we are in first-order arithmetic, the collection of axioms is often infinite, because some axioms are axiom schemas (e.g. the induction axiom of PA), into which an arbitrary statement of the theory must be substituted first. Therefore, when doing a proof in our model which requires use of such an axiom schema, we must first show that we can actually represent the statement (about points of `D`) which we want to prove using the first-order syntax. For large statements, proving this is significant work, even though it is actually repetitive. This library automates that task.

3 Usage

The libary defines two type classes, for which instances must/may be given:

```
  Class tarski_reflector := {
        fs : funcs_signature;
        ps : preds_signature; (* ps and fs give the syntax *)
        D  : Type;
        I  : @interp fs ps D; (* D and I give our model *)
        emptyEnv : nat → D;  (* We need an "empty" env that returns some default
        ↳ value for our types *)
        isD : Ast.term → bool;  (* Returns true iff the given term references D used
        ↳  above *)
}.
(* Extension point to extend the libary to more syntactic constructs *)
Class tarski_reflector_extensions (t:tarski_reflector) := {
        baseLogicConnHelper : option (string → baseConnectiveReifier);
        baseLogicVarHelper : option (string → baseConnectiveVars);
        ↳ (* for reifying instances of inductive types Coq.Init.Logic.* applied to
        ↳ arguments. Mostly used for reifying eq *)
        termReifierVarHelper : option termFinderVars;
        termReifierReifyHelper : option termFinderReifier;
        ↳ (* for reifying all terms that we fail to reify *)
        formReifierVarHelper : option propFinderVars;
        formReifierReifyHelper : option propFinderReifier
        ↳ (* for reifying all forms that we fail to reify *)
}.
```

The `tarski_reflector` class is crucial and contains a complete description of the theory's syntax and the corresponding model. It can be constructed using the `buildDefaultTarski` helper, which can derive the first four members, and constructs a default env from any given point of `D`. The `isD` must be manually provided. It is notably not simply an Ast.term, but a function, because it can be possible that there are multiple formulations of the same type. For example, Coq will typically infer type `@D <instance of tarski_reflector >`. Since this is common, the framework can already expect this, but to keep things flexible the user is able to make the framework aware of different formulations. Here is a typical declaration:

```
Variable D' : Type.
Context {I : interp D'}.
Instance PA_reflector : tarski_reflector := buildDefaultTarski (i_f (f:=Zero) (Vector
↳ .nil D')) (fun k ⇒ match k with (tVar "D'") ⇒ true | _ ⇒ false end).
```

This example is taken from a Peano arithmetic development - the point therefore is the semantic representation of zero. Other developments might use the representation of the empty set or just assume that the model is nonempty.

Now, one typically defines notations that ease applying the semantic and syntactic connectives, like this (again taken from a PA development. `Succ` and `Zero` are constructos of the `funcs_signature`, `Eq` is from the `preds_signature` instance.):

```
(* Syntactic connectives *)
Notation zero := (func Zero (Vector.nil term)).
Notation "succ␣x" := (func Succ (Vector.cons x (Vector.nil term))) (at level 37).
Notation "x␣'=='␣y" := (atom Eq (Vector.cons term x 1 (Vector.cons term y 0 (Vector.
↳ nil term))) ) (at level 40).
(* Semantic connectives *)
Notation iZero := (@i_f _ _ D' I Zero (Vector.nil D')).
Notation "'iSucc␣d" := (@i_f _ _ D' I Succ (Vector.cons d (Vector.nil D'))) (at level
↳  37).
Notation "x␣'i='␣y" := (@i_P _ _ D' I Eq (Vector.cons x (Vector.cons y (Vector.nil D
↳ '')))) (at level 40).
```

For complicated reasons, it is crucial to use exactly the type of the model, and not `D`. This is because the notations should already be reduced and not change when one uses e.g. `cbn.`

One can now use the tactic `represent` and its siblings on goals like `representableP 0 (forall d ↳ , d i= i0)`. The `0` means that the second argument is a function with 0 arguments (i.e. just `Prop`). Alternatively, one could have `representableP 2 (fun a b ⇒ a i= b)`, which means that the binary equality predicate is representable.

3.1 Tactics

The basic tactic one wants to use is `represent`. It automatically constructs a suitable instance of an environment, and a syntactic form of the theory, and then proves that this is equivalent to the given statement. There also is `representNP`, which elides the proof and hopes that the given first-order reification, when instantiated into `sat`, is computationally equal to the given statement - trying to close the goal with easy. If it fails to do so, the user must close the proof theirselves. Then there are `constructEnv`, which constructs the environment as well as the "MetaCoq helper" that tells the reification utils what term is bound where in the environment. The most basic (and most powerful) tactic is `representEnvP` (as well as its non-proof sibling `representEnvPNP`), which needs to be given an environment and the "MetaCoq helper" (which maps Coq terms to their position in the environment), and can then derive a term as well as a proof. This can be used if one wants to use a specific environment.

Note that these tactics only work on goals of the form `representableP <number> <term to reify ↳ >`, because an implicit argument is used to infer the instance of type class `tarski_reflector`.

3.2 Non-proof mode

Constructing the proof of correctness along with the syntactic reification leads to a significant increase in run-time. We could not figure out why this happens since it is a syntactic operation similar to the derivation of the reification, but since the derived proof is often trivial, we included no-proof mode so that the proof can be elided and be given by something like `easy`. Users are encouraged to first try closing a goal with non-proof mode, and if that fails, use the more powerful tactic. These modes currently exist as two seperate functions which perform mostly similar things - trying to merge them again incurred the same slowdown.

4 Inner workings

When looking at the code, you will often find seemingly-identical versions of two functions/definitions, where one is suffixed with `NP`. These are indeed similar, the one tagged with `NP` just elides the proof, as mentioned.

The basic idea is to pattern-match on various syntactic construct in the MetaCoq-quoted term. The two main functions that do this are `findTermRepresentation` and `findPropRepresentation`. The former matches on terms (terms are things that have type `D` (semantically) or `term` (syntactically)), the latter on forms (of type `Prop`/`form`). These call various helper methods which construct a quoted reification and a quoted proof term. While the process is written to be structurally recursive, it is extensible and thus many functions are instead written structurally recursive on some fuel.

4.1 Finding term representations

The type of `findTermRepresentation` is `Ast.term →nat →Ast.term →Ast.term →(Ast ↳ .term → FailureMonad nat) → FailureMonad (pair Ast.term Ast.term)`. There are two additional arguments, namely a `tarski_reflector` and a corresponding `tarski_reflector_extensions ↳` . These are context arguments in the source code. Here is a description of the other arguments:

1. The previous `tarski_reflector` , quoted.
2. The fuel. If it is 0, we stop.
3. The actual term to reify.
4. The env term, quoted This is the environment which will be used in the representability proof (the first thing constructed by the `createEnv` tactic). The default argument is the "empty" environment which returns an arbitrary point for all inputs.
5. A function that, for a given part of a term, yields the index that term has in the env term. The default argument fails for all inputs. This function is called when we reach a term that we cannot represent otherwise, like `tRel n`. We then look where `tRel n` is in the environment (let's say at position `k`) and then `var k` is the representative, and `eq_refl` the proof.

We then return a pair consisting of the quoted reification and a quoted proof of its correctness.

The function only does three things:

1. Check if the term is a semantic connective (e.g. `S`, `O`, + in PA). In this case we recursively find a representation of the subterms and connect them using the syntactic reifier.
2. Call the extension point.
3. Look up the environment.

If neither of these yield a reification, we fail.

4.2 Finding form representations

As before, we discuss the arguments. Note that the same 2 context-arguments we had before are also present.

1. As before, `tarski_reflector` , quoted.
2. The fuel. If it is 0, we stop.
3. The actual term to reify.
4. The "frees". When trying to prove something like `representableP frees P`, the `frees` used there. We use it to first "introduce" the `n` free variables required. Afterwards, it is mostly 0, except when reifying the existential quantifier, where it is 1.

5. The env term, quoted, as above.
6. The env lookup term, as above.

We again return a pair consisting of reification and proof.

Here, we match a few more constructs

1. Base logical connectives. A base logical connective is an inductive type from `Coq.Init.Logic`. In this case we call a base logical connective handler which does further delegation/recursion/extension point calling.
2. Semantic connectives (atoms, e.g. $=$ or $\in$ in PA/ZF). We then recursively reify the subterms and build a proof.
3. As a special case, we handle the `iff` ($<\!\!\rightarrow$), which is defined in `Coq.Init.Logic` but not an inductive type.
4. Handling $\rightarrow$. Since $\rightarrow$ is just notation for a product type, we must check whether a product type quantifies over `D`. If that is the case, we are reifying a forall quantifier, otherwise we are reifying an implication. In the latter case we must adjust indices of the implied statement since it is inside a product term, but should not be.
5. If the `frees` is > 0, we introduce a variable.
6. We call the extension point.

If none of these cases matches, we fail.

4.2.1 Base logical connectives

We merge the handling of different "base logical connectives", by which we mean inductive types defined in `Coq.Init.Logic`. For each of these we handle, we define a function that yields the reification as well as one that yields a proof that the reification is correct. These are then instantiated with the correct sub-reifications (and their proofs). Finally, we have a function that calls the correct case based on the connective name, (e.g. given `"and"`, we use `reifyAnd`).

The arguments are as follows:

1. As before, `tarski_reflector`, quoted.
2. The list of all the arguments the original inductive type was specialized with
3. fuel
4. env term
5. env helper
6. A reduced-argument recursive-call-helper variant of `findPropRepresentation`, where the first and second arguments are chosen appropriately.
7. A reduced-argument recursive-call-helper variant of `findTermRepresentation`, where the first and third argument are chosen appropriately.

Again, the return value is a pair of a reified term and a correctness proof.

4.3 Building correctness proofs and reifiers

The individual reification helpers mainly call the recursive function and then plug the resulting partial terms/proofs into a "merge helper" which then builds a next larger proof. These merge helpers are written to accept the right amount of arguments in a specific order. Building the actual MetaCoq term is thus easy since we simply use `tApp`. The actual proof can be done in the merge helpers in the usual way. The merge helpers for base logical connectives should be straightforward. For the

connectives that are part of the syntax of the specific logic, an interesting solution was found that, given the respective connective specifier (i.e. a member of `funcs_signature` or `preds_signature`), then accepts a varying number of arguments so that we do not need to build a vector in MetaCoq terms. An extended version of this also builds the correctness proofs.

5 Building the environment

It is often necessary to build a specific environment in order to show representability. For example, consider the statement `forall d:D, representableP 0 (iZero i= d)`. This can be represented by the syntactic form $zero == \$0$ in the environment

$$env(x) = \begin{cases} 0 & x = 0 \\ \text{undefined} & otw. \end{cases}$$

However, we need to build the environment to contain `d` and know at which index it is placed.

For this we have functions similar to the ones already outlined, except that they don't yield reifications, but rather a list of all unbound variables. These can then be stitched together into an environment. The arguments of the unbound variable finding functions are similar to the arguments of the functions they mirror, except that typically the env terms are not there.

6 Extension points

The out-of-the-box reifier can reify the outlined semantic connectives of the theory, as well as forall and existential quantification and the logical and/or/implication/equivalence/truth/falsity constructs and variables. This should be sufficient for smaller terms. However, it requires one to only construct ones terms from the basic syntactic constructs available. This is not sufficient, because one might add more complicated additional terms or forms. (For example, one might consider having a function `repNat : nat → D`, which constructs a term representing a natural number (i.e. by building the von Neumann set in ZF). One can then have another function `synNat : nat → term`, which builds a term for the same number in the syntactic theory. If we work in places where the argument is arbitrary, they cannot be reified.)

To solve this, the user can declare an instance of the `tarski_reflector_extensions` class. This will be used as a fallback if the existing development fails to find a reification. In this, there are two fields for each extension point, which should both be filled. Unused extension points should be declared with `None`. The three extension points are outlined here. One of the two fields is used during the env building stage, the other during the actual reification.

Extension points are given the mostly the same arguments as the functions they are extending – what is missing are the implicit context arguments as well as the fuel. They also get recursion helpers to compute the problem for subparts of the original expression. Recursion helpers are used because otherwise the extension point would need to supply the original function with the extension point class instance, which of course does not already exist (in fact, this would likely lead to a proof of falsity).

6.1 Base logical connective

The main point of this extension point is to reify Coq's inbuilt equality, if one uses a model where this is a suitable representation. Most other inductive types defined here are already covered and not handled by the extension point. The extension point takes as arguments the name of the base logical connective (the last part of `Coq.Init.Logic.xxx`), the list of arguments, the fuel, the env term and env helper (not during env building) the quoted `tarski_reflector` instance, the recursion helper for `findPropRepresentation`, and the one for `findTermRepresentation`

6.2 Term reification

This extension point is called when term reification does not match a semantic connective. The arguments are the quoted instance, the fuel, the term to reify, the env term and env helper (not during env building) and a reflection helper for `findTermRepresentation`

6.3 Form reification

Called when form reification fails. Arguments: quoted instance, fuel, actual term, frees, env term, env helper, reflection helpers for `findTermRepresentation` and `findPropRepresentation`.

When called in no-proof mode, the second element of the tuple returned by the reflection helpers will be the dummy term `noProofDummy`. Whatever the user returns in the second entry of the return from the extension point is discarded. This design allows the user to always build a proof, and as long as the proof is not part of the term itself, things will work.

7 Design choices

This library perfoms reflection based on the syntactic layout of the Coq term. This implies, of course, that it is very easy to break this by simply hiding the term one wants reified behind definitions. Users have to be careful to only define things as notations, in order to not break the syntax matching (or write a custom extension point for their added functions).

We handle the environment in a seperate pass because the initial use case did not expect environments, and because recursively constructing the proof and the environment would lead to several issues.

Since one use case has an ad-hoc definition of equality (by assuming that Coq's inbuilt equality and the equality you get by semanically interpreting the syntactic connective that is part of the theory are equal), we originally set out to reify with a proof in order to correctly handle equality. However, since that turned out to be slow (for unknown reasons), and because a prototype once had worked this way, we added no-proof mode in order to make this more efficient.

8 Further work

One aspect of further work is finding out why building the proof incurs a slowdown. We tried to make building the proof conditional on a flag, but that did only provide minor increases in speed. You can find the result of this here.

Another possibility would be to make the reification engine more powerful by adding support for function reifications. Functions can be modeled as total, functional relations, and then statements of the form $f(x) = y$ in Coq can be reified to something similar to $\exists k : P_f(x, k) \land k = y$, where P_k is a predicate equivalent to $f(x) = k$. This would make it possible to show representability for a significantly larger and more complex set of propositions. However, the reification engine would then need to keep track of all functions f for which a representability proof (establishing the existance of an P_f) has been given, which increases complexity because we need some kind of "state" or database to store these in. These representability proofs could potentially be automatically derived by assembling them from representability schemes for certain computational primitives (like `if` or recursion, which can all be modeled in PA and any model that contains PA, since all computable functions are PA-representabe (see https://en.wikipedia.org/wiki/Kleene's_T_predicate)).