Generating induction principles and subterm relations for inductive types using MetaCoq

Bohdan Liesnikov[1,2], Marcel Ullrich[1], and Yannick Forster[1]

[1] Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
[2] IMPRS-CS, Saarland Informatics Campus, Saarbrücken, Germany

When one defines an inductive type in Coq there are various derivable definitions one frequently uses, like induction principles, subterm relations, equality deciders, countability proofs, alternative representations of constructors, etc. Some of these can be derived automatically by Coq or Coq plugins. For instance, Coq derives induction principles automatically: When one defines a type of balanced trees with leaves labelled by elements of `A` as follows

```
Inductive brtree A : nat → Type :=
| Leaf (a : A) : brtree A 0   | Node (n : nat) (l : list (brtree A n)) : brtree A (S n).
```

Coq automatically derives the principle

```
brtree_rect : ∀ (A : Type) (P : ∀ n, brtree A n → Type),
(∀ a, P 0 (Leaf A a)) → (∀ n l, P (S n) (Node A n l)) → ∀n (t : brtree A n), P n t.
```

As is well-known, the automatic derivation ignores the nested recursive occurrence of `brtree` in the argument `l` of the constructor `Node`.

Furthermore, the Equations [5] plugin for Coq for instance comes with a command

```
Derive Subterm for brtree.
```

which derives the direct subterm relation for `brtree` and tries to prove it well-founded – but again this command ignores the recursive occurrence in `Node`.

Tassi's Elpi plugin [6] already covers induction principles for nested inductive types like `brtree`. It would be interesting to directly extend Coq's induction principle mechanism to cover nested inductives as well. However, writing Coq plugins and extending the Coq source code is at least highly non-trivial. For non-experts, the OCaml code of both Coq and Coq plugins is hard to access and almost impossible to adapt without the help of experts.

The MetaCoq project [3] aims at making the implementation of Coq plugins easier. Instead of writing OCaml code, one implements a syntax transformation over an inductive type of terms as pure Coq function. We present three plugins written in this style and hope that the proposed presentation can make writing MetaCoq plugins more accessible.

Our plugins are available at https://github.com/uds-psl/metacoq-examples-coqws.

Alternative constructors for types When one works with dependent inductive types like `brtree` it might be helpful to define

```
Node_eqs : ∀ n (l : list (brtree A b)) m, m = S n → brtee A m
```

Writing a syntax transformation on MetaCoq's inductive type of Coq terms transforming the type of `Node` to the type of `Node_eqs` is straightforward. In MetaCoq, the type `∀ x : A. B` for instance is represented by the element `tProd (nNamed "x") A B` of the inductive type `term`, and bindings are implemented by `tRel n` using de Bruijn indices. We provide a command

```
MetaCoq Run Derive Generalized Constructor for Node as Node_eqs.
```

applicable to non-mutual inductive types, which derives `Node_eqs` via the following function:

```
Fixpoint abstract_eqns (Σ : global_env_ext) (Γ : context) (ty : term) (n : nat) : term :=
  match ty with
  | tProd na A B ⇒
    tProd na A (abstract_eqns Σ (Γ,, na ↦A) B 0)
  | tApp L A ⇒
    let type_of_x := try_infer Σ Γ (lift (2 * n) 0 A) in (* returns type or tRel 0 *)
    let eqn := mkApps tEq [type_of_x; tRel 0; lift (1 + 2 * n) 0 A] in
    tProd (nNamed "x") type_of_x
    (tImpl eqn (abstract_eqns Σ (Γ,"x" ↦type_of_x,eqn) L (S n)))
  | B ⇒ mkApps B (map (fun m ⇒ tRel (1 + 2 * m)) (seq 0 n))
  end.
```

Induction principles for nested inductive types Similarly, we implement a syntax transformation generating a proof term of an induction principle given a representation of an inductive type. Using ideas from Tassi's Elpi plugin [6] we support non-mutual nested inductive types by re-using the unary parametricity translation implemented in [1] and provide a command

```
MetaCoq Run Scheme Induction for brtree.
```

which derives the strongest possible induction principle:

```
brtree_ind_MC : ∀ p : ∀ (A : Type) (n : nat), brtree A n → Type,
(∀ A a, p A 0 (Leaf A a)) →
(∀ A n l, is_list (brtree A n) (p A n) l → p A (S n) (Node A n l)) →
∀ A n t, p A n t
```

Note that `is_list` is essentially the `List.Forall` relation lifted to `Type`.

Subterm relation Lastly, we implement a syntax transformation which generates the subterm relation for non-mutual inductive types (not covering nested inductives). For instance,

```
MetaCoq Run Derive subterm for list.
```

derives

```
Inductive list_direct_subterm : ∀ A : Type, list A → list A → Prop :=
| cons_subterm0 : ∀ (A : Type) (a : A) (l : list A), list_direct_subterm A l (a :: l).
```

Future work It should not be hard to extend the derivation of subterm relations to also cover nested inductive types. In principle MetaCoq also allows the verification of plugins, for instance by relying on the verified type inference from [4] one could prove that the term generated in the first plugin is well-typed, that the type of the induction principle is well-formed, or that the reflexive transitive closure of the subterm relation is well-founded. We are trying to find the right abstractions to make such proofs feasible. It would also be interesting to implement automatic countability and finiteness proofs in MetaCoq, like [2] does based on type classes.

References

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In *ITP 2018*. Springer.

[2] Arthur Azevedo de Amorim. Deriving instances with dependent types. *CoqPL 2020*.

[3] 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*, February 2020.

[4] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–28, 2019.

[5] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in coq. *PACMPL*, 3(ICFP), 2019.

[6] Enrico Tassi. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *ITP 2019*, LIPIcs, Dagstuhl, Germany, 2019.