Generating Case Analysis Principles for Inductive Types using MetaCoq

Marcel Ullrich — s8maullr@stud.uni-saarland.de

November 1, 2019

Given an inductive type, Coq can generate and prove case analysis and induction principles using the Scheme command implemented in OCaml. The principles and proofs follow the structure of the inductive type. Our goal is to replace the OCaml code with MetaCoq programs. MetaCoq is a meta-programming framework that allows writing Coq plugins in Gallina instead of OCaml. We want to write a program transforming inductive types into principles and the corresponding proofs. Because the programs are written in Gallina, we can verify the correctness of the principle generation plugin. In this setting, correctness just means that the plugin produces a well-typed term for every inductive type given as input. Therefore the plugin can never fail to generate a principle.

1 Case analysis principles

Every inductive type has a case analysis principle that scrutinizes an instance of the inductive type and allows proving statements for each way to construct the instance instead of a general unknown instance.

A case analysis principle is used to perform case analysis on elements of inductive types. This corresponds to the destruct tactic in a proof. If we want to prove a statement over a natural number n we can perform case analysis and prove the

statement for 0 and for $S\ m$ with $m : \mathbb{N}$. The same strategy applies for inductively defined proposition like $\wedge$, $\vee$ and $\leq$.

An example is the case analysis principle for natural numbers (example 1.1.1):

$$E_{\mathbb{N}} : \ \forall (p : \mathbb{N} \to \mathbb{T}), p\ O \to (\forall m, p(S\ m)) \to \forall x, p\ x$$

The proof of a case analysis principle or induction principle is called the eliminator. The eliminator for induction principle is recursive.

Every inductive datatype has a case analysis principle similar to the induction lemma without inductive hypotheses and therefore without recursion in the eliminator.

A way to generate the case analysis principle in Coq is the `Scheme` command which generates a proof for the case analysis principle of an inductive type `T`.

```
Scheme T_case := Elimination for T Sort U.
```

We want to generate proof terms for the case analysis principles given a representation of an inductive type. For the generation, we use the MetaCoq framework. MetaCoq is a meta-programming framework which allows us to write plugins directly in Gallina. Therefore we can prove correctness properties of the generated principles. In this setting, correctness means that the plugin produces well-typed terms for the inductive type given as input. Our main goal is to replicate the case analysis scheme command:

```
MetaCoq Run Scheme T_case := Elimination for T Sort U.
```

The case analysis principle has to handle the replacement and instantiation of indices, quantification of variables like the m above and additional hypothesis as would be the case for $\leq$ (example 1.1.3).

1.1 Application and examples

Below are some examples to show the constructed lemmas and some features in use.

1.1.1 Natural numbers

$$\mathbb{N} : \mathbb{T}$$

$$n : \mathbb{N} ::= O \mid S\ n$$

$$E_{\mathbb{N}} : \ \forall (p : \mathbb{N} \to \mathbb{T}), p\ O \to (\forall m, p(S\ m)) \to \forall x, p\ x$$

For natural numbers our case analysis principle, also called non recursive eliminator, is quite simple: We have one constructor without arguments and a second one with a natural number as argument. We don't have any parameters or indices.

1.1.2 Disjunction

$$\begin{aligned}
&\vee : \mathbb{P} \to \mathbb{P} \to \mathbb{P} \\
&L : \forall (A : \mathbb{P})(B : \mathbb{P}), A \to A \vee B \\
&R : \forall (A : \mathbb{P})(B : \mathbb{P}), B \to A \vee B
\end{aligned}$$

Disjunction is an inductive type with two parameters A and B defining a proposition.

$$\frac{A}{A \vee B}\,\mathrm{L} \qquad\qquad \frac{B}{A \vee B}\,\mathrm{R}$$

$$\begin{aligned}
E_{\vee} :\ & \forall A\ B, \forall (p : A \vee B \to \mathbb{P}), \\
& (\forall (a : A),\ p(L\ A\ B\ a)) \to \\
& (\forall (b : B),\ p(R\ A\ B\ b)) \to \\
& \forall (x : A \vee B),\ p\ x
\end{aligned}$$

The two parameters A and B are quantified at the beginning and are applied to each constructor. In both cases the proof of either sides is quantified and provided when using the lemma.

1.1.3 Less or equal

$$le : \forall (n : \mathbb{N}), \mathbb{N} \to \mathbb{P}$$

Less or equal has one parameter, the first argument, and one index, the second argument.

$$\frac{}{le\ n\ n}\,le\_n \qquad\qquad \frac{le\ n\ m}{le\ n\ (Sm)}\,le\_S$$

$$\begin{aligned}
E_{\leq} :\ & \forall (n : \mathbb{N}), \forall (p : \forall m,\ n \leq m \to \mathbb{P}), \\
& p\ n\ (le_n\ n) \to \\
& (\forall m\ (h : le\ n\ m),\ p\ (S\ m)\ (le_S\ n\ m\ h)) \to \\
& \forall m\ (x : le\ n\ m),\ p\ m\ x
\end{aligned}$$

In the first case the index is instantiated with the parameter n and the constructor has no additional arguments. In the second case the constructor takes a natural number m and the statement $n \leq m$ as arguments. The index is instantiated with $S\ m$ and the all arguments are provided to the constructor. The indices are taken directly from the constructor.

1.2 Structure

Let T be an inductive Type. In general the case analysis principle of T applies to a statement $p\ x$ where x is an instance of T. By applying the case analysis principle x gets instantiated with a concrete application of the constructors. This generates one case for each constructor.

T can have arguments like disjunction (example 1.1.2) or less or equal (example 1.1.3). An argument is called parameter if the instantiation is the same across all calls to T in the constructors and index if the instantiation varies.

Parameters are quantified in front of the principle (example 1.2.2). In order to deal with the indices p does not only take the instance but also the indices (example 1.2.4) as arguments.

1.2.1 Parameter-free types

$$\begin{aligned} &T : \mathbb{T} \\ &E_T : \forall (p : T \to \mathbb{P}), p\ c_0 \to \ldots \to (\forall a_0 \ldots a_n, p\ (c_m\ a_0 \ldots a_n)) \to \forall (x : T), p\ x \end{aligned}$$

An example for a parameter-free type are natural numbers (example 1.1.1).

If the inductive type T does not have any parameters and indices then the case analysis principle E_T has one case for each constructor with quantification over all arguments of the constructor. In this example the constructor c_0 takes no argument, like O for natural numbers, and c_m takes n arguments called a_0 to a_n.

The result of E_T is the statement that p holds for every instance x of T if it holds in each possible way to construct x. The result type of p depends on T and can be $\mathbb{T}$ if T allows large elimination.

An inductive instance of a proposition can be eliminated over $\mathbb{T}$ (perform large elimination) if the inductive type for the proposition or predicate has at most one proof constructor and every non-parametric argument of the constructor is a proof itself.

1.2.2 Index-free types

$$\begin{aligned} T :\ & T_{P_0} \to \ldots \to T_{P_k} \to \mathbb{T} \\ E_T :\ & \forall P_0 \ldots P_k, \forall (p : T\ P_0 \ldots P_k \to \mathbb{P}), \\ & p\ (c_0\ P_0 \ldots P_k) \to \ldots \to (\forall a_0 \ldots a_n, p\ (c_m\ P_0 \ldots P_k\ a_0 \ldots a_n)) \to \\ & \forall (x : T\ P_0 \ldots P_k), p\ x \end{aligned}$$

T_{P_i} is the type of the ith parameter.

An example for an index-free type is the disjunction (example 1.1.2).

If the inductive type T has parameters $P_0, \ldots P_k$ but no indices some quantification has to be added. As the parameters are the same across the whole lemma, they are quantified first. For typing the parameters need to be applied to the constructors.

1.2.3 Non-uniform parameter types

Non-uniform parameters are parameters which can have different instantiations in recursive occurrences of T. They are handled like normal parameters.

1.2.4 Indexed types

$$
\begin{aligned}
T: \;& T_{P_0} \to \ldots \to T_{P_k} \to T_{I_0} \to \ldots \to T_{I_k} \to \mathbb{T} \\
E_T: \;& \forall P_0 \ldots P_k, \forall (p : \forall I_0 \ldots I_l, T\ P_0 \ldots P_k\ I_0 \ldots I_l \to \mathbb{P}), \\
& p\ i_0 \ldots i_l\ (c_0\ P_0 \ldots P_k) \to \ldots \to \\
& (\forall a_0 \ldots a_n, p\ i_0 \ldots i_l\ (c_m\ P_0 \ldots P_k\ a_0 \ldots a_n)) \to \\
& \forall I_0 \ldots I_l, \forall (x : T\ P_0 \ldots P_k\ I_0 \ldots I_l), p\ I_0 \ldots I_l\ x
\end{aligned}
$$

T_{I_i} is the type of the ith index.

An example for an indexed type is less or equal (example 1.1.3).

In the most general case T can have parameters $P_0, \ldots P_k$ and indices $I_0, \ldots I_l$ as well. As the indices can change in each case they need to be provided to p and therefore p quantifies over indices and the instance of T.

In each case the indices are instantiated with $i_0 \ldots i_l$ as they were in the constructor case. Here $i_0 \ldots i_l$ can be some constants, depend on the parameters or on the arguments $a_0 \ldots a_n$ of the case.

The conclusion is that p holds for every instance x of T with the indices I_0 to I_l if it holds for every way to construct instances of T.

```
E≤ := fun (n : ℕ)
(p : ∀ m : ℕ, n ≤ m → Prop)
(H_le_n : p n (le_n n))
(H_le_S : ∀ (m : ℕ) (H : n ≤ m), p (S m) (le_S n m H)) ⇒
fix f (m : ℕ) (x : n ≤ m) {struct x} : p m x :=
match x as y in (_ ≤ m) return (p m y) with
| @le_n _ ⇒ H_le_n
| @le_S _ m x ⇒ H_le_S m x
end
```

Listing 1: fully annotated case analysis principle proof term for ≤

1.3 Proof

$$
\begin{aligned}
E_T := \; & \lambda P_0 \ldots P_k. && (1)\\
& \lambda (p : \forall I_0 \ldots I_l, T\ P_0 \ldots P_k\ I_0 \ldots I_l \to \mathbb{P}). && (2)\\
& \lambda H_0 \ldots H_m. && (3)\\
& \mathit{fix}\ f\ I_0 \ldots I_l\ (x : T\ P_0 \ldots P_k\ I_0 \ldots I_l). && (4)\\
& \text{match } x \text{ return } p\ I_0 \ldots I_l\ x \text{ with} && (5)\\
& \quad c_i\ a_0 \ldots a_n \Rightarrow H_i\ a_0 \ldots a_n && (6)\\
& \quad \ldots && (7)
\end{aligned}
$$

The proof of the case analysis principles is automated using a general scheme which strictly follows the type of the case analysis principle (compare Listing 1). To construct the case analysis principle, it is enough to construct the proof term and infer the type.

Parameters, p and the cases for the constructors are dealt with λ abstractions (1-3), an intro call in proof scripts.

The quantification over the indices and the instance is handled by a fixpoint declaration (4) as this allows us to use recursion later on. Currently, this is not necessary because case analysis does not use inductive hypotheses.

The main proof is done using a match on the instance x (5) with application of the corresponding case in each constructor case (6). The arguments are forwarded from the constructor to the case.

Using type inference it suffices to provide the type of p, x and the return type of the match.

```
fold (fun t param ⇒ lambda param.name param. type t) params
```

Listing 2: Pseudocode of parameter quantification

1.4 Implementation

The implementation follows the strategy outlined in Section 1.3 except the type inference of arguments which can't be used.

First the `mutual inductive` body and the one `inductive` body of the specific type T are extracted. The `mutual inductive` body contains the parameters and the inductive types, which can be more than one in case of a mutual inductive definition. The one `inductive` body contains the name, type, elimination possibilities and constructors of T. Each constructor is represented by the number of arguments, the name and the type.

Then some information for later implementation is gathered like the types of the indices, the number of constructors and indices and some preparation on the type is done.

Step (1): The parameters are taken using `fold` over the parameter list. For each parameter a λ abstraction is nested around the proof term starting with the remaining proof (see Listing 2). The parameter list needs to be reversed as it is stored in reverse order in the inductive body.

Step (2): The next step is to take p. We can directly adopt the index quantification inside the type of the predicate from the inductive type without parameter quantification.

Step (3): Next the cases for the constructors are taken. The main problem in this step is to construct the type according to the constructor type, remove parameter and construct the corresponding call to p with the constructor at the end. The instantiation of the indices is acquired from the original term by replacing T with p, removing the parameters in the application and appending a call of the constructor with all arguments to construct an element of T for p.

Step (4): The main step happens in the fixpoint declaration. For the type of the fixpoint all indices and the instance are quantified as they are in the type of p. The quantification is changed to λ abstractions for the body of the fixpoint to take each argument. Here we need to lift each parameter access additionally by one in comparison to the type as we have a de bruijn index for f in front.

Step (5): The same λ abstractions are needed inside the return type of the match for the indices and match instance. Our result is again the application to p as it was in the result of our fixpoint.

Step (6): Finally, the match cases are generated where each constructor c_i calls the corresponding case hypothesis H_i and provide the arguments a from the constructor. This is done with a mapping on the constructor types and is nearly the same as in the cases above with different lifting for the parameters and application of the case instead of p.

The program to generate a case analysis principle for type T can be called using the command `MetaCoq Run Scheme T_case := Case for T Sort U`.