# Strengthening the *inversion* Tactic in Coq

Anne Mulhern

May 10, 2010

Coq's *inversion* tactic fails when it is required to invert an hypothesis in *Prop* to prove a goal that is not in *Prop*.

There are some exceptions to this rule.

- If the type of the hypothesis has zero constructors the *inversion* tactic succeeds.
- If the type of the hypothesis has just one constructor, and deconstructing the term yields no further information about the term than is already given by the type, the *inversion* tactic succeeds.

Terms with types that fit these conditions are always invertible. The *Derive Inversion* command behaves analogously to the *inversion* tactic.

The type, `False`, is a canonical example of a type that has only invertible inhabitants, since it has zero constructors, hence zero inhabitants. The type, `and`, is another canonical example of a type that has only invertible inhabitants. The type, `odd`, is a non-example. Like `and`, it has just one constructor. However, the arguments to its unique constructor, `odd_S` are more informative than its type.

We argue that the condition

*whenever the inductive definition has just zero or one constructors and the arguments to its unique constructor, if any, are no more informative than its type*

is too strong. The weaker condition

*whenever the type of the term to be inverted fully determines its constructor (with some technical restrictions, which we are only able to mention briefly)*

which is implied by the stronger condition is more appropriate. We have extended Coq's *inversion* tactic within the *Ltac* language so that it succeeds when this condition holds.

The *inversion* tactic derives for each constructor in the inverted term all the necessary conditions that must hold for the term to be proved by that constructor. Some of these conditions may lead to an easily proved contradiction, in which case the *inversion* tactic (or the *Derive Inversion* command) automatically derives the contradiction and proves the subgoal for that constructor. The statement "the type of the term fully determines its constructor" is thus equivalent to the statement "the *inversion* tactic will automatically solve all goals *or* it will automatically solve all but one goal (if the goal is in *Prop*)".

We treat the "zero constructor" case first, as it is by far the simplest. The definition of `odd` has just one constructor, `odd_S`. However, the hypothesis `odd O` leads immediately to a contradiction for that constructor, since one of the necessary conditions for `odd O` to hold is that `O = S n`. Consequently, it is possible to prove the sublemma, `odd O -> False`. Regardless of the sort of the goal, when there is an hypothesis `odd O`, inversion is possible by

1. automatically constructing a proof of `odd O -> False`
2. applying this proof to the hypothesis `odd O` to derive a contradiction
3. making use of the contradiction to prove the goal

This result generalizes to any situation where, for every constructor, the necessary conditions that must hold for the term to be proved by that constructor yield a contradiction.

For the "unique constructor" case, it is possible to discover the necessary conditions for that constructor to hold. The definition of `even` has two constructors, `even_O` and `even_S`. If the hypothesis is `even (S n)` for some `n` then a necessary condition for the `even_O` constructor, `O = S n`, yields a contradiction. Consequently, it is possible to prove the sublemma `forall n, even (S n) -> odd n`, since `odd n` is the necessary condition for the constructor `even_S`. Regardless of the sort of the goal, when there is an hypothesis `even (S n)` for some `n`, inversion is possible by

1. automatically constructing a proof of `forall n, even (S n) -> odd n`
2. applying this proof to the hypotheses `n` and `even (S n)` to prove `odd n`
3. inserting the hypothesis `odd n` into the context of the current goal

This result generalizes to many situations where, for all but one constructor, the necessary conditions that must hold for the term to be proved by that constructor yield a contradiction.

The `even_S` constructor has just one necessary condition, `odd n`. In general, however, a constructor will have several necessary conditions. In that case, the return type of the automatically constructed sublemma must be the conjunction of these necessary conditions. Since terms in `and` are always invertible, regardless of the sort of the goal, the result of applying the automatically generated sublemma to the hypothesis will yield a conjunction of terms that can alway be decomposed into separate terms.

This approach works well for the sort of inductive definitions that appear to arise naturally in mechanizing metatheory [1]. For example, the definition for local closure has exactly as many constructors as there are constructors for the definition of terms in the lambda-calculus. If the type of any proof of local closure is fully precise then the type fully determines the constructor. Moreover, each constructor has in general, a fairly simple type, e.g, an application is locally closed if both its subterms are locally closed. The return type of the inversion sublemma is the conjunction of the statements of local closure for each subterm of the application.

However, it is not possible to invert every term where the type uniquely defines the constructor. For example, suppose that the hypothesis is `ex (fun x => even x)`. `ex` has just one constructor, `ex_intro`, with type `forall x, P x -> ex P`. However, the necessary sublemma must introduce a new variable, the witness, as well as a proof that `P` holds for the witness. Clearly, this sublemma must be impossible, otherwise one could extract the witness and make use of it to build a goal in any sort.

We have implemented our stronger *inversion* tactic entirely within the *Ltac* tactic language. We have made use of a sandboxing technique, i.e., we assert our intention to prove `True` by means of the hypothesis to be inverted. In the subgoal thus constructed it is possible to make use of Coq's standard *inversion* tactic to discover what the type of our automatically generated subgoal should be, to prove it, and to return the proof and the type to the original goal. If the return type of the sublemma is `False`, then it can be used to solve the original goal. If the return type of the sublemma is a conjunction of propositions then this conjunction can be decomposed, and the parts inserted into the proof context. If the *inversion* tactic leaves more than one subgoal in the sandbox, or the remaining subgoal is not tractable, the stronger *inversion* tactic fails. Our stronger tactic defaults to the standard *inversion* tactic wherever that will succeed.

We argue that it would be desirable to strengthen the *inversion* tactic within the Coq system, in the way that we have described and to strengthen the *Derive Inversion* command in the identical manner.

Our strengthened tactic optionally memoizes the sublemmas that it automatically generates. We would be interested in a new tactic that facilitates caching and lookup of sublemmas generated during the execution of *Ltac* tactic scripts.

## References

[1] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15, New York, NY, USA, 2008. ACM.