Towards Automating Permutation Proofs in Rocq: A Reflexive Approach with Iterative Deepening Search

Nadeem Abdul Hamid
Berry College, Mount Berry, GA, USA

Abstract

The concept of permutations is fundamental in computer science, and is useful for specifying and reasoning about a variety of data structures and algorithms. This paper presents the implementation of a fully automated tactic for proving complex permutation goals within the Rocq Prover (formerly, Coq proof assistant). Our approach leverages proof by reflection and an iterative deepening search procedure to establish permutation relations on arbitrary lists composed of concatenation operations. We detail the construction of mapping/substitution environments, a unification algorithm, and metaprogramming tactics to automate the proof process. The potential impact of the tactic for goals involving permutations is demonstrated by significant reduction in proof script length for an existing non-trivial development.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Automated reasoning

Keywords and phrases permutations, reflection, tactics, Rocq, Coq

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.39

Category Short Paper

Supplementary Material
Software (Source Code): `https://github.com/nadeemabdulhamid/permsolver`
archived at `swh:1:dir:4d13d14ab92f66bd27713d743a7937403b837014`

Acknowledgements The author thanks students Matthew Bowker and Bernny Velasquez for participating in preliminary efforts and discussions on this development.

1 Introduction

The concept of permutations of a list is intuitive, yet readily formalized, and is useful for specifying and reasoning about the properties and behavior of a variety of data structures and algorithms. The Rocq (formerly Coq) proof assistant Standard Library provides an inductive definition of permutations as compositions of adjacent transpositions of elements of a list.[1] The library additionally provides quite a large set of derived properties for reasoning about permutations. However, proving non-trivial permutation relationships between lists, especially in the context of transitivity and premises involving permutations of sublists, quickly turns into a tedious exercise of finding the right swapping order and transitive substitutions to solve the goal.

In this paper, we present the implementation of a fully automated tactic for proving complex goals involving permutations of lists (of arbitrary element type) described as the concatenation of their constituent portions, where some or all of those portions may be universally quantified variables (as opposed to concrete values of the element type). Figure 1 provides an example of a goal state that can be solved by the tactic.

[1] `https://rocq-prover.org/doc/V8.20.0/stdlib/Coq.Sorting.Permutation.html`

```
A : Type
h : A
a, b, a', t, a1, a2 : list A
...
H1 : Permutation (a ++ b) (h :: t)          ++ is list concatentation
H2 : Permutation a (h :: a')                :: is 'cons'
H3 : Permutation (a' ++ b) t
H4 : Permutation (a1 ++ a2) a'
...
----------------------------------------
Permutation ((a1 ++ b) ++ h :: a2) (a ++ b)
```

Figure 1 Goal state solved automatically by `perm_solver`.

Our approach uses proof by reflection (see Section 2) and is implemented entirely within the Rocq system itself. We first generate an environment mapping natural number labels to distinct pieces of all lists formed by concatenation operations. For every list in the proof context, a tree of natural numbers is constructed, such that substituting for the numbers based on the mapping environment results in (i.e. *reflects* into) the original list. Then, we implement a unification procedure on the sets of labels collected from the leaves, which takes into account substitutions representing permutation facts that occur as hypotheses in the context. The algorithm performs an iterative deepening search process to explore possible substitutions in order to unify the values between the two label sets in question. Finally, we prove that if the unification procedure succeeds, it means that the original two lists (corresponding to the trees of numbers, under the mapping environment) are permutations according to the standard library formulation.

In what follows, we review a few examples of permutation-based reasoning in existing Rocq developments (including the particular context that motivated the tactic described in this paper) and discuss related and prior work that inspired our tactic. We then present details of the implementation and conclude with a discussion of future directions.

2 Background and Related Work

Sorting is perhaps the most obvious algorithmic process for which the property of the output "having the same elements as" the input is crucial. Indeed, the initial textbook on the Rocq system [2] presents the specification of a sorting program as a motivating example. Rather than using the current standard library definition of `Permutation`, it introduces a relation on lists that is based on counting the number of times any element appears in both lists. As noted there, this definition is convenient for reasoning about properties, but actually "does not provide a way to determine whether two [concrete] lists are permutations of each other."

The *Verified Functional Algorithms* volume [1] of the *Software Foundations* series presents additional algorithms and data structures for which proving correctness depends on reasoning about whether two collections have the same contents. In this development, the standard library's inductive formalization of `Permutation` is adopted and used not only for sorting algorithms, but also for verifying properties of abstraction relations and representation invariants on data structures such as priority queues. As the formalization of these classical data structures is fairly straightforward, it is only mildly tedious to work with the existing standard library lemmas to manually guide proofs involving permutation relationships.

In a more substantial development, [10] developed a formalization of k-d trees in Rocq, utilizing the standard library definition of `Permutation` for specifying correctness properties of procedures that involve sorting and partitioning input and intermediate data points. That work noted that lack of automation for reasoning about `Permutation`s resulted in significantly complex and tedious proof effort. The goal in Figure 1, for example, is adapted from one of the proof states in that development. Proving this goal using only the standard library facilities requires a long set of lemma invocations to reorder list components, interspersed with appeals to transitivity properties on sub-portions of the lists.

Reasoning about permutations also appears to be an important part of formalized Rocq libraries related to rewriting theory, termination, and program transformation [6, 3, 7]. Contejean [6] notes that most of the formal proofs "actually concern the permutations of the lists..." and "...the proofs are quite long since there are many subcases."

As far as existing work towards automating reasoning about permutations, there are only a few perfunctory developments that we are aware of. Braibant and Pous [5] describe a general-purpose Rocq plugin for rewriting modulo associativity and commutativity with some limited application to lists and permutations. The Rocq Reference Manual provides an example of writing a tactic to prove that a list is a permutation of a second list, based on an alternate inductive definition of the concept, but it only works on lists with concrete individual elements.

An online GitHub repository [8] provides a tactic that transforms `Permutation` goals into solving multiplicity calculation (an alternate, deprecated definition of the notion of permutations based on multisets in the Rocq standard library). In its current form, this tactic is limited to reasoning about lists of natural numbers, although in theory it could be generalized to any type with decidable equality and would appear to be able to solve goals with the same level of complexity as the tactic described in this paper. The requirement of decidable equality for the domain of the list elements, however, introduces the need for "a more complex apparatus," which is unnecessary for our tactic. [8] relies on the `omega` (now `lia`) tactic, a decision procedure for linear integer arithmetic, to trivially solve equations over the group of natural numbers with addition once the permutation statements have all been transformed into multiplicity equations. Preliminary comparisons on goals involving lists of `nat` indicate that our tactic produces much smaller proof terms, reflected in the size of compiled script files. It is unclear how the time to check the larger proof terms compares to the computation time for our reflection-based tactic. Further analysis and comparison of space and time efficiency of these two approaches remains as an item for future work.

Our tactic utilizes the technique of proof by reflection [4, 2], in which explicit reasoning steps (i.e. on a permutation goal) are transformed into some implicit computation that is carried out by the proof assistant. A theorem is then separately established that shows that the result on the computational values is reflected as a property on the original terms (i.e. that a pair of lists satisfies the `Permutation` predicate). We also adapt some inspirations from prior approaches [9, 2] such as building binary trees of `nat` values, flattening, and using the Ltac metalanguage to program a reification tactic, all described in the next section.

3 Implementation

In Section 3.1, we step through the overall approach of our tactic, `perm_solver`, on the goal state of Figure 1, to give a general idea of how it operates. Section 3.2 describes the computational "unification" algorithm, followed by an overview in Section 3.3 of the metaprogramming tactics implemented to reify list objects and apply the reflective technique.

```
Inductive nattree := lf : nat -> nattree | br : nattree -> nattree -> nattree.
Fixpoint nattree_to_list {A} (nt:nattree) (menv:NatMap.t (list A)) : list A := ...

Theorem check_unify_permutation :
  forall A (tenv: list (nattree * nattree)) (nt1 nt2: nattree) menv,
  (forall t1 t2, List.In (t1, t2) tenv
            -> Permutation (nattree_to_list t1 menv) (nattree_to_list t2 menv)) ->
  check_unify (flatten_env tenv) (flatten nt1) (flatten nt2) = true ->
  Permutation (nattree_to_list nt1 menv) (nattree_to_list nt2 menv).
```

Figure 2 Salient definitions for reflecting trees of labels into lists.

3.1 General Approach to Automating Permutation Proofs

To begin with, we build a *mapping environment* that assigns arbitrary labels to distinct subterms of all concatenated list terms in the proof context. For Figure 1, this results in:

```
[6 |-> a2, 5 |-> a1, 4 |-> a', 3 |-> t, 2 |-> [h], 1 |-> b, 0 |-> a]
```

We use a finite map data structure from the Rocq standard library, with keys ranging over the data type of unary natural numbers, `nat`. Occurrences of `cons` (`::`) are rewritten into concatenations (`++`) of a singleton list in this process (i.e. `h::t` becomes `[h]++t`).

With the mapping environment prepared, we now reify (to a first approximation) every `Permutation` term into a pair of lists (treated as multisets) of numbers. Pairs corresponding to the `Permutation` hypotheses are collected in a *substitution environment*, such as:

```
[ ([0; 1], [2; 3]), ([0], [2; 4]), ([4; 1], [3]), ([5; 6], [4]) ]
```

and the goal of Figure 1 would correspond to `([5; 1; 2; 6], [0; 1])`. We now apply the unification algorithm (Section 3.2) in order to equate the pair of sets in the goal, by searching for a sequence of appropriate subset substitutions based on the substitution environment. For example, replacing the `0` on the right with `[2; 4]`, and then the `4` with `[5; 6]` results in the set `[2; 5; 6; 1]` on the right, which contains the same elements as `[5; 1; 2; 6]`.

In reality, we do not reify terms forming list concatenations directly into flat multisets of `nat`s, because that loses information about the grouping order of the operations in the original. Instead, we define `nattree` (Figure 2), a data type of binary trees with `nat` values at the leaves. The `nattree_to_list` function of Figure 2 *reflects* a `nattree` back into a list of elements, under the mapping environment, `menv`. A term like `Permutation ((a1 ++ b) ++ h :: a2) (a ++ b)` can now be rewritten into:

```
Permutation
    (nattree_to_list (br (br (lf 6) (lf 2))  (br (lf 1) (lf 5))) M)
    (nattree_to_list (br (lf 1) (lf 0)) M))
```

which is equivalent under Rocq's conversion rules, for a properly constructed environment `M`. Note that `nattree_to_list` swaps the order of the branches as the tree is flattened, for convenience in the proof development.

With this reflection set up, and the `check_unify` function (see next section), we establish the `check_unify_permutation` theorem in Figure 2. The `tenv` list is the substitution environment introduced earlier, but maintained as a list of `nattree` pairs, rather than pairs of lists. The first premise of the theorem expresses an obligation that for every pair of `nattree`s in the substitution environment, there is a proof that the reflected lists are permutations

```
Fixpoint CUD (d:nat) (env:list (list nat * list nat)) (lft rgt:list nat) (rpt:bool)
  : bool := match d with | 0 => false     (* hit depth limit *)
  | S d' => match (remove_common lft rgt) with
       | (nil, nil) => true               (* lft and rgt contain same elements *)
       | _ => fold_left                   (* recurse through applicable_subs *)
            (fun result i => result ||
                  let env' := if rpt then env else (drop_nth i env) in
                  let (e1, e2) := nth i env (nil, nil) in
                      (is_sublist e1 lft && CUD d' env' (subst e1 e2 lft) rgt rpt)
                   || (is_sublist e2 lft && CUD d' env' (subst e2 e1 lft) rgt rpt))
            (applicable_subs env lft)     (* list of indices *)
            false                         (* default result *)              end end.
```

Figure 3 Implementation of the unification algorithm.

of each other. The second premise expresses that running the unification algorithm on the lists of `nat`s resulting from flattening all trees (in the goal and the substitution environment) succeeds with a result of `true`. The theorem is proven through lemmas showing that each manipulation of the sets of numbers carried out by the unification algorithm (under the appropriately constructed environments) is sound with respect to the permutation relation.

Now, given a goal like Figure 1, the `perm_solver` tactic invokes the tactic metaprograms described in Section 3.3 to construct the mapping environment, substitution environment, and rewrite all `Permutation` terms using `nattree_to_list`. It then applies the `check_unify_permutation` theorem and uses some helper tactics to satisfy the first premise. The second premise is trivially satisfied as long as the carefully constructed arguments to `check_unify` result in the entire term simplifying to `true`.

3.2 Unification Algorithm

Our "unification" algorithm takes two `nat` lists, a *left* and *right*, and attempts to transform the *left* into the *right* through a backtracking process of applying substitutions from the substitution environment. Each potential substitution is represented as a pair of `nat` lists, and can be applied in either direction. The core of our unification algorithm is presented in Figure 3. The recursion is limited to a fixed `d`epth, and the `rpt` argument specifies whether substitution pairs in `env` are retained as they are applied, so they may be applied multiple times if it is `true`. Otherwise, once a substitution is applied, it is removed from `env` in the subsequent recursive search. The `remove_common` function reduces its arguments to a pair of lists where any common elements to both have been deleted from each. The expression `(subst a b lst)` substitutes `a` for `b` in `lst` by appending `b` to the result of removing all elements of `a` from `lst`. The `is_sublist` guard conditions are necessary before the recursive calls to ensure that the substitution is valid in that direction.

As a concrete example, the following would compute to `true`:

```
let env := [([1; 0], [3; 2]); ([0], [4; 2]); ([1; 2], [3; 3])]
 in CUD 5 env [1; 4; 2; 1; 4; 2] [3; 2; 3; 2] true.
```

For this example, `(applicable_subs env [1; 4; 2; 1; 4; 2])` results in the list `[1; 2]`. That is, only the second and third elements of `env` represent applicable substitution pairs (one from right to left, and the other from left to right) for the list `[1; 4; 2; 1; 4; 2]`. In this example, applying `false` to `CUD` instead of `true` would cause the result to compute to `false` overall, because there is no way to transform the elements of the left list into the right without repeating the use of a substitution.

Table 1 Summary of helper tactics used by `perm_solver`.

Sub-tactic	Purpose
`normalize_append`	Normalize all hypotheses (convert `cons` to `++`, etc.)
`collect_hyps_perm_terms`	Collect a list of all "atomic" list terms separated by concatenations (++) in the goal and hypotheses
`gen_map_all`	Generate the *mapping environment*
`rewrite_hyp_perms`	Rewrite every `Permutation` term with convertible `nattree_to_list` expressions, using a `build_nattree` tactic to reify the original lists into `nattree`s
`build_tenv`	Build the *substitution environment*
`apply check_unify_permutation`	Invoke the reflection theorem
`apply tenv_perm_forall`	Solve substitution environment subgoal introduced by `check_unify_permutation` (the implication of Figure 2 involving `List.in`)

The full `check_unify` function that is mentioned in Section 3.1 generates a list of depths that are quartiles of the length of the substitution environment. It invokes `CUD` with those iteratively deepening values and a `rpt` argument of `false`. In the usage we have surveyed so far, it appears that despite a large number of `Permutation` assumptions appearing in the hypotheses of a goal, the number of substitutions that needs to occur is often less, within a quarter of that number at most. Furthermore, other than contrived examples, it does not appear that substitutions ever need to be applied more than once. In cases where these norms do not hold, our library provides a "fixed-depth" variant of `check_unify` where the user specifies an explicit depth limit and the `rpt` argument is supplied a value of `true`.

3.3 Metaprogramming Tactics

Based on the general approach and unification algorithm described in the preceding sections, we now need to automate the construction of mapping and substitution environments, and reify list terms into `nattree` objects. The top-level tactic, `perm_solver`, does this by invoking several sub-tactics, outlined in Table 1. We developed `perm_solver` using Rocq's Ltac language, as opposed to its successor, Ltac2, due to being relatively more familiar with the former and for relative paucity of documentation and examples with commentary for the latter. The major awkwardness in implementing the various tactics in Table 1 is needing to use continuation-passing style wholesale, since the language has no notion of `return` values at the level of tactical metaprogramming. For space reasons, we omit further elaboration and refer the interested reader to the actual implementation in the supplemental material.

4 Discussion and Conclusion

To date, we have utilized our `perm_solver` tactic to rewrite proofs from the k-d tree verification proofs of [10]. The results appear to be significant, with some 707 lines ($\sim 20\%$) reduced out of 3277 lines of definitions and proof scripts – some of this due to refactoring of the scripts supported by the incorporation of `perm_solver`. It thus seems promising that the tactic would help speed up proof developments that use the `Permutation` definition. In future work, we would like to validate its utility on a broader scale – seeking out existing developments for which it could be adopted and/or promoting its use along with the standard library's notion of `Permutation` where appropriate.

In the short term, we anticipate porting the implementation to Rocq's Ltac2, in line with recommended practice. Along with that, it would be interesting to incorporate support for proving goals related to `Permutation`, such as the `List.In` predicate or length properties.

The unification algorithm as presented is certainly not complete. Since it is depth-limited, it is straightforward to formulate contrived examples where it fails to successfully unify lists when it should (i.e., a situation where the minimum number of substitutions needed exceeds the maximum depth). In practice however, so far, we have found that the iterative deepening approach works fine, even in pathological cases where there are a large number of `Permutation` assumptions in the proof context. We have not profiled the algorithm, but we did investigate the use of Rocq's binary representation of numbers and the incorporation of sorting lists into a canonical form. Since the reified lists of numeric labels are usually short, neither of these optimizations was worth the effort – we found that implementing them did not produce noticeable improvement in speed; and the sorting perhaps even introduced additional overhead. Use of Rocq's `positive` type speeds up comparison of numeric values, but does not affect the recursion depth of any of the key functions which dominate the run time. It would be ideal to discover a more sophisticated, efficient, and/or complete decision procedure for the unification process. At a very minimum, future improvement might focus on a pre-processing step to clear unrelated permutation facts/substitutions from the environment before the main work of the tactic.

References

1. Andrew W. Appel. Verified functional algorithms. In Benjamin C. Pierce, editor, *Software Foundations*, volume 3. Electronic textbook, 2024. Version 1.5.5, `http://softwarefoundations.cis.upenn.edu`.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. `doi:10.1007/978-3-662-07964-5`.
3. Frédéric Blanqui, Solange Coupet-Grimal, William Delobel, Sébastien Hinderer, and Adam Koprowski. CoLoR: a Coq library on rewriting and termination. In *Eighth International Workshop on Termination (WST'06)*, August 2006.
4. Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, LNCS, pages 515–529, Berlin, Heidelberg, 1997. Springer. `doi:10.1007/BFb0014565`.
5. Thomas Braibant and Damien Pous. Tactics for reasoning modulo AC in Coq. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 167–182, Berlin, Heidelberg, 2011. Springer. `doi:10.1007/978-3-642-25379-9_14`.
6. Evelyne Contejean. A certified AC matching algorithm. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications*, pages 70–84. Springer, Berlin, Heidelberg, 2004. `doi:10.1007/978-3-540-25979-4_5`.
7. Evelyne Contejean. Modeling permutations in Coq for Coccinelle. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, pages 259–269. Springer, Berlin, Heidelberg, 2007. `doi:10.1007/978-3-540-73147-4_13`.
8. foreverbell. permutation-solver. Accessed 2025-02-24. URL: `https://github.com/foreverbell/permutation-solver`.
9. Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, LNCS, pages 98–113, Berlin, Heidelberg, 2005. Springer. `doi:10.1007/11541868_7`.
10. Nadeem Abdul Hamid. (Nearest) Neighbors you can rely on: Formally verified k-d tree construction and search in Coq. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024*, SAC '24, pages 1684–1693, New York, NY, USA, 2024. Association for Computing Machinery. `doi:10.1145/3605098.3635960`.