

Towards Automating Permutation Proofs in Rocq: A Reflexive Approach with Iterative Deepening Search (Short Paper)

Nadeem Abdul Hamid



ITP 2025, Reykjavic, Iceland

Reasoning about permutations

```

Hr6 :
  size datapt (sum_dist query)
  (knn sum_dist K k rgt (snd (bb_split bb ax (nth ax pt 0))) query
   (knn sum_dist K k lft (fst (bb_split bb ax (nth ax pt 0))) query
    (insert_bounded K datapt (sum_dist query) pt pq))) < K -> loRgt = []
Hr5 : all_in_leb (sum_dist query) pqlstRgt loRgt
H : Permutation result pqlstRgt
HresiSm_Lg : all_in_leb (sum_dist query) resiSm resiLg
eSmSm, eSmLg, lstSmSm, lstSmLg : list (list nat)
HresiStuff :
  Permutation (eSmSm ++ eSmLg) eSm /\
  Permutation (lstSmSm ++ lstSmLg) lstSm /\
  Permutation (eSmSm ++ lstSmSm) resiSm /\
  Permutation (eSmLg ++ lstSmLg) resiLg /\
  all_in_leb (sum_dist query) eSmSm eSmLg /\ all_in_leb (sum_dist query) lstSmSm lstSmLg
HlresSm_Lg : all_in_leb (sum_dist query) lresSm lresLg

```

Inductive Permutation : list A -> list A -> **Prop** :=

```

| perm_nil: Permutation [] []
| perm_skip x l l' : Permutation l l' -> Permutation (x::l) (x::l')
| perm_swap x y l : Permutation (y::x::l) (x::y::l)
| perm_trans l l' l'' : Permutation l l' -> Permutation l' l'' -> Permutation l l''.

```

```

eSmSmSm, eSmSmLg, lstSmSmSm, lstSmSmLg : list (list nat)
HresiStuff2 :
  Permutation (eSmSmSm ++ eSmSmLg) eSmSm /\
  Permutation (lstSmSmSm ++ lstSmSmLg) lstSmSm /\
  Permutation (eSmSmSm ++ lstSmSmSm) resiSmSm /\
  Permutation (eSmSmLg ++ lstSmSmLg) resiSmLg /\
  all_in_leb (sum_dist query) eSmSmSm eSmSmLg /\
  all_in_leb (sum_dist query) lstSmSmSm lstSmSmLg
H12 : Permutation (resiSmSm ++ lfttreeSmSm) lresSm
H13 : Permutation (eSmSmSm ++ lstSmSmSm) resiSmSm

```

(1/1)

Permutation (eSmSmSm ++ lstSmSmSm ++ lfttreeSmSm) lresSm

Prior Work

- Rewriting theory, termination, and program transformation [6, 3, 7]
- Tactics for reasoning modulo AC (associativity and commutativity) [5]
 - Some limited attention to lists and permutations
- Tactic to transform Permutation goals into solving multiplicity calculations (alt. defn. of permutations)
 - GitHub repo <https://github.com/foreverbell/permutation-solver>
 - Could be generalized to any type with decidable equality
 - Reduces to solving equations with `lia`

General Approach

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

1. Build a **mapping environment** (*nat* to atomic list terms)
2. **Reify** permutation terms into pairs of binary trees with *nat* leaves
3. Collect tree pairs from hypotheses into a **substitution environment**
4. Run a “**unification**” algorithm on the substitution environment and goal trees.
5. Apply the **reflection** theorem

1. Build a mapping environment

```

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

```

```

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

```

- Collect arguments of all Permutation propositions
- Match terms around ++ and generate a map and reverse map (Ltac pseudo-list of (*atom* , *n*) pairs -- used to see if an atom is already mapped)

2. Reify lists into binary trees

Permutation ((a1 ++ b) ++ h :: a2) (a ++ b)

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



Permutation

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

3. Build a substitution environment

A : Type

h : A

a, b, a', t, a1, a2 : list A

...

H1 : Permutation (a ++ b) (h :: t)

H2 : Permutation a (h :: a')

H3 : Permutation (a' ++ b) t

H4 : Permutation (a1 ++ a2) a'

...

Permutation ((a1 ++ b) ++ h :: a2) (a ++ b)

([0,1], [2,3]),

([0], [2,4]),

([4,1], [3]),

([5, 6], [4])

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

4. Unification

<pre> A : Type h : A a, b, a', t, a1, a2 : list A ... H1 : Permutation (a ++ b) (h :: t) H2 : Permutation a (h :: a') H3 : Permutation (a' ++ b) t H4 : Permutation (a1 ++ a2) a' ... </pre> <hr/> <pre> Permutation ((a1 ++ b) ++ h :: a2) (a ++ b) </pre>	<pre> ([0,1], [2,3]), ([0], [2,4]), ([4,1], [3]), ([5, 6], [4]) </pre>	<pre> [6 -> a2 5 -> a1 4 -> a' 3 -> t 2 -> [h] 1 -> b 0 -> a] </pre>
---	--	---

[5,1,2,6]

[0,1]

- Substitute associated sets from the substitution environment in the left goal until it matches the right.
- Implemented as a **Fixpoint** that computes to **bool**.

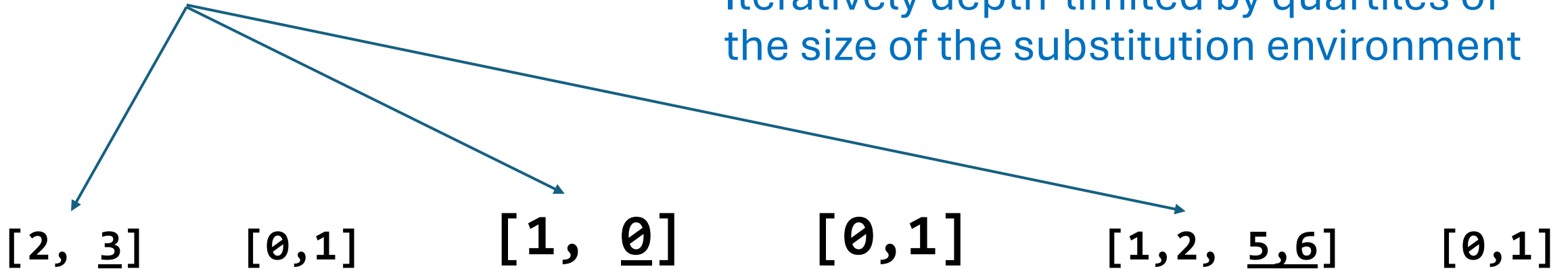
4. Unification - demo

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

$[5,1,2,6] \quad [0,1]$

$[1,2, \underline{4}] \quad [0,1]$

- Determine applicable substitutions on the left
- Recursively try applicable substitutions
- Iteratively depth-limited by quartiles of the size of the substitution environment



5. Apply the reflection theorem

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).
```

- Use one more tactic to clear the obligations relating the substitution environment to Permutation assumptions

Put it all together!

```
Goal forall A (pq:list A) pqsm pqlg D Dsm Dlg R L x y,  
  Permutation pq (x :: pqsm ++ rev pqlg) ->  
  Permutation (rev pqlg) pqlg ->  
  Permutation (Dsm ++ Dlg) D ->  
  Permutation R (pqsm ++ y :: Dsm) ->  
  Permutation L (Dlg ++ pqlg) ->  
  Permutation (R ++ x :: L) (y :: D ++ pq).
```

Proof.

```
  intros; perm_solver.
```

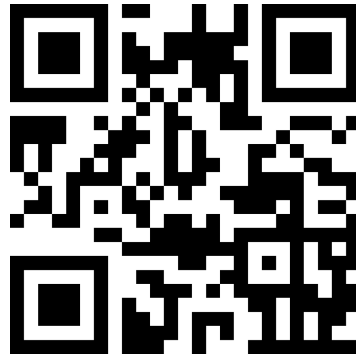
Qed.

Usage and Experience

- ~20% (707/3277 lines) reduction in large proof development involving lots of reasoning about permutations.
- Seems reasonably efficient in practice with IDS
- Obvious optimizations don't have noticeable effect
 - Binary nat representation
 - Sorting lists into canonical form
- Future work:
 - Port to Ltac2

Thank you!

nadeem@acm.org



<https://github.com/nadeemabdulhamid/permsolver>

Towards Automating Permutation Proofs in Rocq:
A Reflexive Approach with Iterative Deepening Search (Short Paper)

Permutations based on multisets

There exists a permutation between two lists iff every element has the same multiplicity in the two lists

```
Inductive multiset : Type := Bag : (A -> nat) -> multiset.
```

```
Definition permutation (l m:list A)  
  := meq (list_contents l) (list_contents m) .
```

```
Definition meq (m1 m2:multiset) :=  
  forall a:A, multiplicity m1 a = multiplicity m2 a.
```

```
Definition multiplicity (m:multiset) (a:A) : nat  
  := let (f) := m in f a.
```

Continuation-Passing Style

```
Ltac build_env_and_go A :=
  normalize_append A;
  match goal with (@Permutation A ?X ?Y) =>
    collect_hyps_perm_terms A constr:((X, (Y, tt)))
    ltac:(fun hyps_perm_terms
      => gen_map_all hyps_perm_terms constr:(0) constr:(empty (list A)) tt
        ltac:(fun ctr env rmap =>
          let name := (fresh "env") in set (name := env);
          rewrite_hyp_perms A rmap name;
          build_tenv constr:((@nil (nattree * nattree)))
          ltac:(fun tenv => let tname := (fresh "tenv")
            in set (tname := tenv);
            apply check_unify_permutation with tname;
            [ apply tenv_perm_forall;
              repeat (apply tp_cons; auto);
              apply tp_nil | reflexivity ])))
    end.
```