12/5/20 [redacted]

Proving a Reversed List is a Permutation of the Original in ACL2

In our proof, our goal was to show that a list and the reverse of that list are permutations of each other, meaning that they contain the same elements but may or may not have the same order. To prove this, we began by establishing that permutation is reflexive, symmetric, and transitive; along the way, we proved some helper theorems that were necessary in the proofs of these properties. Next, we showed that the first element of a list is always found in its reverse. We then proved that deleting the first element of a list from the reverse of that list produces a permutation of the rest of the reversed list. Finally, we were able to prove our original goal theorem.

Our proof utilized four main functions, namely in2, del, rev, and permp. The function in2 returns a boolean that indicates whether its first argument is present in its second argument if the second argument is a list. Del deletes the first occurrence of its first argument from its second argument if its second argument is a list. The function rev is used as defined in ACL2 and generates the reverse of a given list. The function of interest in our primary goal, permp, returns a boolean to state whether or not the two given arguments are permutations of each other, if both are lists, or whether they are both atomic data otherwise.

For our proof to succeed, we needed a few additional pieces of ACL2 functionality beyond defthm and implications. We used the :hints feature to guide the theorem prover more specifically for several of our theorems. The types of hints we provided included :induct hints, which instructed the theorem prover to use a certain induction scheme, and :use hints, which encouraged the theorem prover to apply an existing theorem in its proof. In one theorem, perm-cdr-del-car-expanded, we provided a more specific :use hint with :instance, which told the

theorem prover exactly what substitution to make when using the existing theorem. We also used the in-theory feature early in our proof to enable subsequent theorems to use the induction structure of permp.

In proving that permp is an equivalence relation, we consulted two similar proofs that we found helpful. One proof helped us modify our function definitions for in2, del, and permp so that permp would work for inputs of any type (1). This is an important prerequisite for an equivalence relation. The same proof also provided guidance for our proofs of the reflexivity and symmetry of permp (1). We consulted a second proof for assistance in proving the transitivity of permp (2). With the help of these two proofs, we were able to show that permp is an equivalence relation, which was essential for our later helper theorems and our goal.

We dedicated the first section of our proof to proving that permp is an equivalence relation. First, we proved the lemma reflexive-perm, which shows that permp is reflexive; this lemma was accepted without any sub-lemmata. Next, we proved the helper lemma perm-cons, which establishes for lists x and y that if some element e is in x, the following two claims are equal: that x is a permutation of the list formed by adding e to the front of y, and that deleting the first occurrence of e from x results in a permutation of y. Once we had established perm-cons, we were able to prove the lemma symmetric-perm, which shows that permp is symmetric.

In order to prove that permp is transitive, we needed several sub-lemmata. The first helper lemma, in2-del, declares that for some list z and two unequal items e and d, e is present in a list that is formed by deleting the first occurrence of d from z if and only if e is present in z. Our second sub-lemma, perm-in2, shows that if x is a permutation of z and e is present in x, then e is also present in z. Next, we proved the sub-lemma comm-del, which states that the order in which elements are deleted from a list does not matter. Lastly, we created the helper lemma

perm-del-del, which states that if two lists are permutations of each other, removing an element that is present in one list from both lists results in two lists that are still permutations of each other. Once these four helper lemmas had been approved, we were able to prove the lemma transitive-perm, which shows that permp is transitive. Having proved that permp is reflexive, symmetric, and transitive, we were then able to establish that permp is an equivalence relation.

With this subgoal completed, we began working towards our original goal theorem. Our first helper lemma for this goal was car-in-rev, which proves that the first element of a list is present in the reverse of that list. The next sub-lemma we needed was del-in-app, which shows that for a list x and an element e, the list formed by adding e to the end of x and then removing the first occurrence of e from x is a permutation of x itself. We then proved the helper lemma perm-cdr-del-car to show that removing the first element of a list from the reverse of that list produces a permutation of the reverse of the rest of the list. In order for this lemma to be accepted, we had to first declare the lemma perm-cdr-del-car-expanded. This lemma expands the reverse of the list into the reverse of the rest of the list appended to the first item of the list. Then, we could simply apply del-in-app to prove the expanded lemma and apply the expanded lemma to prove the original lemma.

Finally, we created some sub-lemmata for our goal theorem itself. The theorem perm-rev-base establishes that our goal theorem is correct in the base case, which occurs when we have an empty list. We then created the theorem perm-rev-ind and its helper theorem perm-rev-ind-expanded, which demonstrate that our goal theorem holds in the inductive case, with a non-empty list and inductive hypothesis; these two theorems relied on the helper lemmas car-in-rev and perm-cdr-del-car. Once we had these sub-lemmata and our previous helper theorems in place, we were able to prove perm-rev, which was our goal theorem.

When solving this problem, we began by proving the helper lemmas specific to perm-rev, such as car-in-rev and perm-cdr-del-car, since these were the only lemmas required for a pen-and-paper proof of the same theorem. Next, we divided our goal into the sub-lemmata perm-rev-base and perm-rev-ind; the base case was accepted trivially, but the inductive case wouldn't go through with our existing helper lemmas. We decided to prove that permp was transitive to help ACL2 use our lemmas, and after doing some research, we were able to prove the theorem perm-transitive and its helpers. Our inductive case still didn't work, so we decided to also prove that permp was symmetric. Even after this, our inductive case still couldn't be proved.

At this point, we did some reading on how ACL2 applies rewrite rules and discovered that we needed to establish that permp is an equivalence relation in order for ACL2 to make the correct substitutions. In order to do so, we determined that we needed to generalize our four main functions from working with lists of rationals to working with inputs of any type. Once we had done this, we were able to define permp to be an equivalence relation. Then, we reorganized our lemmas so that the helper lemmas for perm-rev came after the equivalence relation declaration. Finally, our proof was accepted by ACL2, and we were even able to generalize perm-rev and its sub-lemmata to work for any true list, not just a list of rationals.

Due to the straightforward nature of our goal theorem, our proof is not strictly necessary as an explanation for why the theorem holds. You could review the helper theorems for perm-rev to improve your understanding of the problem if you were not sure why the property held, but the proof might not be sufficient explanation on its own if you were initially unsure about the theorem.

Our work on this goal opens up some additional proof possibilities involving the permp function. Using our definition of permp as an equivalence relation, we can more easily create

helper lemmas that could be used to prove any number of goals. One possibility would be to create a function that scrambles a list in a different manner than rev and then to prove that the scrambled list is a permutation of the original; we could take this even further by proving that a scrambled list is a permutation of the reverse of the original, or that two lists that are permutations of each other are still permutations of each other after both lists have been scrambled. Overall, there are many interesting options for further proofs based on our work.

References:

1. https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/ACL2____EQUIVALENCE
2. http://gauss.ececs.uc.edu/Courses/c626/exams/prep_final.html