The Associativity of Merging Three Ordered Lists: A Proof in ACL2s

Andrew Briasco-Stewart
Northeastern University
Boston, MA
briasco-stewart.a@northeastern.com

Christopher Swagler
Northeastern University
Boston, MA
swagler.c@northeastern.edu

Steve Liu
Northeastern University
Boston, MA
liu.steve@northeastern.edu

Introduction

This paper discusses and describes the proof regarding the associativity of merging three ordered lists of `rational` numbers. In other words, the property that concatenating three ordered lists, no matter how they are grouped, produces the same ordered list. We define a list of `rationals` to be sorted or ordered, if it is in ascending numerical order, with the smallest value as the first element. Two lists of `rationals` are equal if every number of one list appears sequentially in the same order as the other list. We begin by first defining `inserted-ordered`, a recursive function that correctly places a given `rational` into a sorted list of `rationals`, and `merge-ordered-insert`, a recursive insertion algorithm that repeatedly calls `insert-ordered` on the first of two ordered lists. We then explain the tedious process of defining our associativity theorem, `merge-ordered-insert-assoc` and assisting ACL2 in proving it by developing the appropriate lemmas. Finally, we show how our proof can be applied to all types of lists and different sorting algorithms and consider the ramifications in related area of research.

All the code included in this report and files that can be run in ACL2s are included in a GitHub repo, the link to which is: https://github.com/abstewart/acl2s-merge-insert-ordered-assoc. Installation instructions for how to run the `official_proof.lisp` file can be found in the `README.md` file.

Background

The three primary functions we define for our proof are `orderedp`, `insert-ordered`, and `merge-ordered-insert`, which are all contained in Appendix A. The basis of our proof relies on a list of `rationals`, so we define it as a data type.

`orderedp` is a predicate function to ensure the ascending order of a list of `rational` numbers. We define the base case such that if the `cdr` of the list is `nil`, the list is indeed ordered. This covers both the case of an empty list and a single element true-list. Provided that the `car` is less than or equal to the `cadr` of the list, and the order upholds for the rest of the list, we can ensure that a list of `rationals` is in ascending order.

`insert-ordered` is a critical function we define for inserting a single `rational` number into a sorted list of `rationals`. For this function, we derive three distinct cases. First, if the ordered list is empty, the `rational` number is returned in a single element true-list. Knowing that the list

is non-empty in the second case, if the `rational` number is less than or equal to the `car` of the list, that `rational` number is `cons`ed onto the front of the list in order to maintain the desired `orderedp` property. If neither of the first two cases are true, the first element in the list is `cons`ed onto the result of a recursive call. This recursive call repeatedly shrinks the list until the appropriate position for the given `rational` number is reached and the element is inserted. By our function definition, we can specify that the output contract of `insert-ordered` will uphold the `orderedp` property.

`merge-ordered-insert` takes in two lists, the second being ordered, and uses `insert-ordered` to repeatedly insert the elements of the first list into the second. For this function, only two specific cases are necessary. If the first list is empty, the second list can simply be returned. Otherwise, the `car` of the first list is inserted via `insert-ordered` into the recursion of `merge-ordered-insert` using the rest of the first list. Based on our implementation, only the second list needs to be ordered and the resulting list from `merge-ordered-insert` will uphold the `orderedp` property.

Walkthrough

When attempting to prove this theorem in ACL2s, we begin with the "professional method" [1, 2], where a proof is sketched out in general, ignoring minor minutia that can slow down the process. Because the professional method is only concerned with the more complicated cases in proving, our high-level proof by induction sketches out the induction case only. We identify (using `skip-proofs`) two first order lemmas that could assist ACL2s in proving our theorem (see Appendix C); the first of which, Lemma 1.1, is proven easily. The other lemma, Lemma 1.2, fails and is where most of the work lies.

This more difficult lemma covers the property of pulling out a call to `insert-ordered` inside a call to `merge-ordered-insert`. We know this is possible since `merge-ordered-insert` will produce an ordered list, which `insert-ordered` will correctly insert an element into. When this lemma is passed to ACL2s, however, it reports failure under a top-level induction.

The error ACL2s gives at this point is not useless though; it does identify three key checkpoints that it is unable to prove, which leads to three second order lemmas to prove. The first two of these lemmas, when written separately and submitted to the REPL, pass with no additional help, something we found surprising as it was not able to prove them in the larger lemma. The third lemma, however, still fails when written separately. See Appendix C for these lemmas.

Upon closer inspection into this third lemma, the property that ACL2s tries to prove is the ability to re-order calls to `insert-ordered` upon a call to `merge-ordered-insert`. Since we know `merge-ordered-insert` produces an ordered list, we abstract this property to create a more general lemma regarding the commutativity of calls of `insert-ordered` on an ordered list. See Appendix C for this lemma.

This third order lemma passes in ACL2s as a separate lemma and once introduced, ACL2s cascades back up the various proving levels, first with Lemma 2.3, then with Lemma 1.2, and finally with the property that we originally set out to prove. The file `official_proof.lisp` contains the finished proof in a format that ACL2s will accept and run to prove the property. The

file `walkthrough_proof.lisp` won't run in ACL2s, but it does show the order in which the various lemmas were composed from a logical standpoint with the main goal at the top, then the first order lemmas, then the second order ones, and finally the third order lemma. Our professional method sketch is also included in this file.

Results

The final results of our proof were satisfactory for our standards mainly for two reasons. First, ACL2s accepted the theorem and trivially proved it, and second, for the elegance of the lemma that finally made ACL2s accept the proof.

Leading ACL2s to accept our proof was a long, arduous process and required a great deal of trial and error. The proof was a constant and often demoralizing struggle of finding the right lemmas to prove and ensuring that those lemmas themselves were accepted by ACL2s through various other sub-lemmas. This is illustrated in the subsection "Work with Professor" in `project_work.lisp` where we found ourselves stuck in an increasingly complex maze of sub-lemmas that were seemingly becoming more trivial but were still unprovable by ACL2s. It was difficult to avoid scenarios like this and it proved to be costly every time we did not. Luckily, ACL2s helps us by providing tools to mostly evade these situations.

One such tool that helped to ensure that we avoided "jumping into the rabbit hole" is the ability to `skip-proofs` a theorem. Utilizing this tool, we predicted if a certain lemma would be useful by making ACL2s accept the lemma regardless of whether or not it could be proven. This was immensely useful as it led to the final Lemma 3.1 that connected everything together, `insert-ordered-assoc`. Finding this lemma is evidence that we were successful because of how simple and elegant it is. It contradicts all of our past failed attempts because it did not require us to jump through a rabbit hole of irrelevant sub-lemmas to reach it (it required going three layers deep). It is such a clean and uncomplicated lemma that it made us doubt whether our main theorem was worth proving in the first place.

When we first proved the associativity of merging three ordered lists, our first reaction was surprise followed by doubt. Our Lemma 3.1 was so simple that we questioned whether it was something noteworthy to write a paper about. However, the best solution to a problem is often the one that is the most clean, terse, and efficient. The elegance of the final lemma encapsulates this concept and provides evidence that our initial doubt of the proof only qualifies our achievement and success.

Finally, the results of our proof can be broadly generalized to other lists and sorting algorithms as well. We used an insertion sorting type algorithm on lists of `rationals` in our proof and compared them based on numeric value, but these can be easily replaced with any other data type, comparator function, or sorting method.

Personal Progress

Despite the seemingly simple and straightforward walkthrough of our proof, we encountered countless difficulties before arriving at the solution. In fact, we previously used an

entirely different function. This function, `merge-ordered`, which can be found under "Initial Work" in `project_work.lisp`, was based on taking two ordered lists of `rationals` as input and performing the merging within that function itself instead of using a helper such as `insert-ordered`. The function compared the `cars` of both lists (if they were non-empty) and `cons`ed the smaller of the two onto the recursive call. In the empty case for either of the lists, the other was returned. At this point in time, our proof still had the objective of demonstrating the associativity of this merging function for three ordered lists.

The main issue that we encountered with this version of the merge function was that instead of the ACL2s output terminating to some type of error, the proof never stopped spinning, even after letting it run for nearly an hour. This made it much more difficult to find a starting point since we did not have a way of pinpointing what ACL2s was stuck proving. In these early stages, we also tried rewriting the function to have an accumulator, found in `proposal.lisp`, but this version of the function ended up being more complicated.

With `merge-ordered` infinitely spinning in ACL2s, we attempted to tackle the problem by providing smaller lemmas. The main issue here was that the function had four distinct cases, two of which were non-trivial. We did not even know what types of lemmas to provide since there was no specific output that could point us in the right direction. We examined the proof tree view and found the induction case that ACL2s was stuck on, but it still did not provide a clear view of what needed to be simplified. The resources regarding ACL2s as a theorem prover suggested that we should find where ACL2s was stuck, determine a lemma to simplify that step, then repeat until the proof succeeded [1, 2]. However, no lemmas were apparent given the output. Additionally, it was difficult to come up with any rewrite rules since `merge-ordered` was rather complex and did not simplify easily. We met with our lead lab teaching assistant, Andrew Walter, and he suggested the technique of writing out the proof by hand to see if any lemmas would become more obvious. Upon attempting this we quickly realized the complexity of the induction scheme necessary to move forward, so we abandoned `merge-ordered` for a simpler version that had the same functionality overall.

A glimmer of hope was in sight when we switched from `merge-ordered` to the current version of `merge-ordered-insert` that used `insert-ordered`. Rather than infinitely spinning when trying to directly prove our associativity theorem, it instead terminated with an error. This was promising, since we could use it as a starting point to assist ACL2s. We decided to start fresh and approach the problem with Walter's technique of hand sketching the proof along with the professional method, which can be found under "New Functions" in `project_work.lisp` [1, 2]. As per the professional method, we began with the inductive case and spent some time designing a way to complete the proof. With the inductive hypothesis and our progress so far, a necessary lemma became obvious (see Lemma 1.2). Again, using the professional method, we assumed that this lemma could be proved, and used it to carry on with the rest of the proof. Further into the proof, another lemma seemed apparent to use (see Lemma 1.1) which eventually led us to a completed proof. Having used this technique, we derived two lemmas that if proved, would lead ACL2s to accept our proof.

One of the two lemmas to prove was Lemma 1.1, `car-cdr-insert-ordered`. Essentially, we needed a way to demonstrate that if we know a list of `rationals` is ordered, inserting the `car` of that list into the `cdr` of the same list would be equivalent to the original list. This theorem sounded trivial and in fact was, having passed instantaneously in ACL2s.

Lemma 1.2, `insert-with-merging`, instead took most of our time to prove. We needed a way to pull the `insert-ordered` out from `(merge-ordered-insert (insert-ordered a b) c)` to get `(insert-ordered a (merge-ordered-insert b c))`. Thankfully, the ACL2s output terminated and did not spin infinitely, but also did not pass trivially. For the next step, we tried to change the induction scheme using hints, since ACL2s defaulted to `(insert-ordered a b)`. We attempted several induction schemes: `lorp`, `tlp`, `orderedp`, `lorp` of `insert-ordered`, etc., none of which got us anywhere.

Frustrated with how close we seemingly were to a solution, we decided to backtrack and try to prove other lemmas that could potentially help. The objective was to simplify the problem as much as we could, then try increasing the complexity of it to see where exactly it failed. We first tried proving our associativity theorem with all single element `rational` lists. This case passed trivially, but we began running into issues when we upped the complexity to having only one of the three lists be a single element. When the first of the three lists was a single element, the theorem would hard fail, whereas when the second or third were single element lists, it would spin forever.

At this point, we felt that we had plateaued in terms of our progress, so we met with our professor, Dr. Jason Hemann, to ask for advice for how to proceed. When we showed him our progress thus far, he suggested using `skip-proofs` on `insert-with-merging`. This would allow ACL2s to accept the theorem without proving it and use it as necessary. Just as we had suspected, when we did this, our actual theorem was able to pass. The light at the end of the tunnel was there, we just needed to get this theorem to pass and then we would have our associativity theorem proven.

Despite our hope, we ended up spiraling into a deep rabbit hole of issues for getting the theorem to pass, all of which is contained in the "Work with Professor" section in `project_work.lisp`. During this session, we tried to simplify lemmas regarding `merge-ordered-insert` down to the `cons` level, with what seemed like the simplest cases. We expanded instances of `cond`s to nested if statements, we tried proving those conditions individually, and we even tried simplifying different properties of `merge-ordered-insert` at the most basic level. Ultimately, our objective was to rewrite `insert-ordered` as a `cons` of an element with `merge-ordered-insert`. At the end of the session, the progress we had made seemed minimal and we were even questioning if our base assumptions were correct.

Once again, we backtracked to see exactly where `insert-with-merging` failed. Since the ACL2s output terminated, we were able to see the cases where it went wrong. Three distinct cases emerged in the output, so we analyzed each of them in "Final Testing" from `project_work.lisp`. When trying to run the first two cases as their own theorems, they surprisingly passed with no additional help. The third case failed, so we took a closer look at what ACL2s was attempting to

prove, and we found that it was trying to prove a property that seemed trivial. Since ACL2s was unable to prove it, we decided to write a separate theorem, `insert-ordered-assoc`, that exhibited that property. On its own, `insert-ordered-assoc` passed instantaneously. We then tried resubmitting the third case, which also passed. The long-awaited day had arrived, and our original proof finally submitted with these theorems provided.

Summary

Our proof further enhances ACL2s' ability to reason about merging ordered lists which is evidenced by its competency to prove properties that it was previously stuck on. For example, ACL2s struggled with proving the commutativity of `merge-ordered-insert`—the property that the order in which two sorted lists are merged does not matter. However, after proving our associativity theorem, ACL2s trivially proves the commutativity theorem, indicating that ACL2s is correctly able to use our lemmas to expand its ability to reason. Furthermore, future researchers are now able to use our proof for other issues related to lists and sorting.

Sorting is a massive topic in the world of computer science and researchers are constantly pursuing better and more efficient sorting algorithms. This research includes accurately and effectively reasoning about the correctness of those algorithms. As mentioned previously, one could adapt these results to many other sorting algorithms and other types of sorted lists. Thus, our universally quantified proof is widely applicable and formally advances known properties about lists and sorting.

Acknowledgements

We would like to thank Dr. Jason Hemann for helping us by providing useful resources and assistance during the times we got stuck. We would also like to thank our head teaching assistant, Andrew Walter, for providing us with methods to handle complicated proofs by demonstrating some of the proper ways to steer ACL2s.

References

[1] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. 2000. The Mechanical Theorem Prover. In Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, USA, 119–210.

[2] Panagiotis Manolios. Feb. 26, 2020. Reasoning About Programs. Ch 6, 7. Draft.

Appendix

Appendix A: Function Definitions

Orderedp

```
(definec orderedp (l :lor) :bool
  (cond
   ((endp (cdr l)) t)
   (t (and (<= (car l) (cadr l)) (orderedp (rest l))))))
```

Insert-ordered

```
(definec insert-ordered (elem :rational l :lor) :lor,
  :ic (orderedp l)
  :oc (orderedp (insert-ordered elem l))
  (cond
    ((endp l) (list elem))
    ((<= elem (car l)) (cons elem l))
    (t (cons (car l) (insert-ordered elem (cdr l))))))
```

Merge-ordered-insert

```
(definec merge-ordered-insert (l1 :lor l2 :lor) :lor,
  :ic (orderedp l2)
  :oc (orderedp (merge-ordered-insert l1 l2))
  (if (endp l1)
    l2
    (insert-ordered (car l1) (merge-ordered-insert (cdr l1) l2))))
```

Appendix B: Main Proof

```
(defthm merge-ordered-insert-assoc
  (implies (and (lorp a) (orderedp a) (lorp b) (orderedp b) (lorp c) (orderedp c))
           (equal (merge-ordered-insert (merge-ordered-insert a b) c)
                  (merge-ordered-insert a (merge-ordered-insert b c)))))
```

Appendix C: Proof Lemmas

Lemma 1.1

```
(defthm car-cdr-insert-ordered (implies (and (lorp a) (consp a) (orderedp a))
       (equal (insert-ordered (car a) (cdr a))
      a)))
```

Lemma 1.2

```
(defthm insert-with-merging
  (implies (and (rationalp a) (lorp b) (orderedp b) (lorp c) (orderedp c))
   (equal (merge-ordered-insert (insert-ordered a b) c)
  (insert-ordered a (merge-ordered-insert b c)))))
```

Lemma 2.1

```
(defthm pt2.1
  (IMPLIES (AND (RATIONALP (CAR B))
                (RATIONAL-LISTP C)
                (ORDEREDP C)
                (CONSP B)
                (RATIONALP A)
                (NOT (CDR B))
                (< (CAR B) A))
           (EQUAL (INSERT-ORDERED (CAR B)
                                  (INSERT-ORDERED A C))
                  (INSERT-ORDERED A (INSERT-ORDERED (CAR B) C)))))
```

Lemma 2.2

```
(defthm pt2.2
  (IMPLIES (AND (RATIONALP (CAR B))
                (NOT (CDR B))
                (RATIONAL-LISTP C)
                (ORDEREDP C)
                (CONSP B)
                (RATIONALP A)
                (<= (CAR B) 0)
                (< (CAR B) A))
           (EQUAL (INSERT-ORDERED (CAR B)
                                  (INSERT-ORDERED A C))
                  (INSERT-ORDERED A (INSERT-ORDERED (CAR B) C)))))
```

Lemma 2.3

```
(defthm pt1.3
   (IMPLIES
    (AND (RATIONALP (CAR B))
         (RATIONAL-LISTP (CDR B))
         (RATIONAL-LISTP C)
         (ORDEREDP C)
         (CONSP B)
         (EQUAL (MERGE-ORDERED-INSERT (INSERT-ORDERED A (CDR B))
                                      C)
                (INSERT-ORDERED A (MERGE-ORDERED-INSERT (CDR B) C)))
         (RATIONALP A)
         (ACL2-NUMBERP (CADR B))
         (<= (CAR B) (CADR B))
         (ORDEREDP (CDR B))
         (< (CAR B) A))
    (EQUAL (INSERT-ORDERED (CAR B)
                           (INSERT-ORDERED A (MERGE-ORDERED-INSERT (CDR B) C)))
           (INSERT-ORDERED A
                           (INSERT-ORDERED (CAR B)
                                           (MERGE-ORDERED-INSERT (CDR B) C))))))
```

Lemma 3.1

```
(defthm insert-ordered-assoc
  (implies (and (rationalp a) (rationalp b) (lorp c) (orderedp c))
           (equal (insert-ordered a (insert-ordered b c))
                  (insert-ordered b (insert-ordered a c)))))
```