Northeastern University

CS2800 Project

Joshua Moffat, Ali Rana, Liam McEniry, Kevin Tang

April 2021

1 Introduction

In this paper, we prove a property loosely related to the shuffling trick known as the "Gilbreath Shuffle". While shuffling is inherently hard to reason about in ACL2 due to the randomness, the function `interleave` can simulates a "perfect" riffle shuffle where the two halves of the deck are evenly dispersed in an `ABABABAB` pattern. The Gilbreath Shuffle involves splitting a deck, reversing one half and then riffle shuffling the halves together. We used the Gilbreath shuffle as inspiration to form the following conjecture.

`Conjecture:`

```
When a deck of cards is split into halves A and B, one of A or B is
reversed, and the halves are interleaved back together; the resulting
deck will have an alternating ABABABAB pattern
```

In this proof we begin by imagining the initial "deck" as a list of symbols and naturals. We chose these data types as they are atomic and they don't overlap. Initially, a list in this form of length n is arranged in the pattern

$$(A_1, A_2, ..., A_{\frac{n}{2}}, B_1, B_2, ...B_{\frac{n}{2}})$$

with A and B being the two datatypes. To represent these lists we created the datatypes `lon` and `los` representing list of natural and list of symbol respectively. We then created a predicate, `everyotherp`, to represent the goal of the proof. `everyotherp` holds for true lists that alternate between symbols and naturals.

We could then construct a mechanized version of English language conjecture as follows in Listing 1.

Listing 1: Conjecture 1

```
(defthm conjecture-1
  (implies (and
            (lonp x)
            (losp y)
            (equal (len2 x) (len2 y)))
           (everyotherp (interleave x (rev2 y)) nil)))
```

We achieved this proof by pursuing the simplification of `conjecture-1` to a version without `(rev2 y)`, which the ACL2 theorem prover could prove on it's own. This hinged on the property

$$\texttt{(losp y)} \rightarrow \texttt{(losp (rev2 y))}.$$

2 Background

Originally determined by mathematician Norman Gilbreath [1], the Gilbreath Shuffle is one of several well-known "false shuffles," as it allows a seemingly random shuffle to create a deck that retains many of the same properties as the original deck. For example, if the original deck alternates between black and red cards, the deck produced by applying the Gilbreath Shuffle to it will have each pair of cards consist of a red and black card. Alternatively, if the original deck consists of four-card groups consisting of one card of each suit, applying the Gilbreath Shuffle will result in a deck that maintains the same pattern - drawing four cards from the deck will yield one card of each suit. This property ultimately makes the Gilbreath Shuffle a key component of several card tricks, as it provides the illusion of randomness while still allowing the shuffler to maintain the deck's properties. This maintenance of the deck's properties hinges on way the deck is halved, one half is reversed, and the two halves are interleaved.

In order to properly understand the Gilbreath Shuffle, one needs to look closer to this process, which led us to prove a simpler subset of the problem - that taking two halves of a deck, reversing one, and interleaving the two will always yield a deck that alternates between elements of each deck.

For the sake of clarity, we decided to use `lon`, a list of natural numbers to represent elements of one half of the deck, and `los`, a list of symbols to represent elements of the second half. Additionally, we created a predicate function, `everyotherp`, to check if a list alternates between natural numbers and symbols. Finally, we achieved the actual interleaving process by adapting an interleave function from CS2500 to use the proper syntax for ACL2.

Through the coding process, there were a few unexpected occurrences. For example, deciding the input contract for `everyotherp` proved troublesome. Even though it was taking in a list of numbers and symbols, we found that without `:all` as the input contract, ACL2 would create, and attempt to prove, sub goals concerning input contracts. Additionally, `everyotherp` needed to be naturally recursive instead of using an accumulator, otherwise it would infer unsuccessful induction schemes. For these reasons, there were multiple iterations of `everyotherp` that led to its final product.

We also used hints and rewrites to build rules from formulas we made, while increasing ACL2's readability of our code and simplifying its operations.

3 Walkthrough

In this proof we represent the two lists being interleaved as lists of symbols and rationals for ease of distinction. We created data definitions for these to guide the theorem prover.

Listing 2: Definitions

```
(defdata lon (listof nat))
(defdata los (listof symbol))
```

The functions `interleave` and `rev2` were also defined

Listing 3: interleave

```
(definec interleave (x :tl y :tl) :tl
  (cond ((endp x) y)
        ((endp y) nil)
        (t (cons (car x) (cons (car y) (interleave (cdr x) (cdr y)))))))
```

Listing 4: rev2

```
(definec rev2 (x :tl) :tl
  (if (endp x)
      nil
    (app2 (rev2 (rest x)) (list (first x)))))
```

In order for us to prove the conjecture, we required a predicate that only holds when the goal has been reached, we called this predicate `everyotherp`. This predicate had several requirements in order to be useful to us. First, as a predicate, it had to take the signature `:all` → `:bool`. Second, to be easily interpreted by the ACL2 theorem prover, it must be naturally recursive. Although not an issue in the proof, the predicate had a limitation in that we must also pass it a boolean which acts as a switch to indicate which of symbol or rational should be first in the list.

Listing 5: Predicate

```
(definec everyotherp (ls :all symb :bool) :bool
  (if (tlp ls)
    (cond
     ((endp ls) t)
     (symb (and (symbolp (car ls)) (everyotherp (cdr ls) nil)))
     ((not symb) (and (rationalp (car ls)) (everyotherp (cdr ls) t))))
    nil))
```

Using the predicate and the data definitions created, we formed a mechanized form of the English language conjecture. This conjecture represented the main goal of the proof.

Listing 6: Conjecture 1

```
(defthm conjecture-1
  (implies (and
            (lonp x)
            (losp y)
            (equal (len2 x) (len2 y)))
           (everyotherp (interleave x (rev2 y)) nil)))
```

There was an extra stipulation that the lengths of the two lists must be the same. this is to prevent cases such as

`(interleave '(AAAAA) '(BB)) → (ABABAAA)` which would cause the proof to fail as it doesn't satisfy `everyotherp`. The first step was to simplify the problem. We did this by eliminating the `(rev2 y)` from the conjecture. To see if this would be worth pursuing, we created the simplified conjecture `conjecture-1-simp`.

Listing 7: Simplified Conjecture

```
(defthm conjecture-1-simp
  (implies (and
             (lonp x)
             (losp y)
             (equal (len2 x) (len2 y)))
            (everyotherp (interleave x y) nil)))
```

`conjecture-1-simp` was accepted by ACL2, therefore we could proceed towards eliminating `(rev2 y)`.

First, since `conjecture-1-simp` passes on any `(losp y)` that is the same length as x, we defined the following property which was easily accepted by ACL2 with no further input.

Listing 8: Lemma 1

```
(defthm lemma1
  (implies (losp ls)
            (losp (rev2 ls))))
```

Then we created a lemma that could serve as a rewrite rule, showing the following equivalence.

```
(equal (everyotherp (interleave x y) nil)
       (everyotherp (interleave x (rev2 y)) nil))
```

We defined this equivalence as `y-rev2-y-equivalence`.

Listing 9: y-rev2-y-equivalence

```
(defthm y-rev2-y-equivalence
  (implies (and (lonp x)
                (losp y)
                (equal (len2 x) (len2 y)))
           (equal (everyotherp (interleave x y) nil)
             (everyotherp (interleave x (rev2 y)) nil))))
```

In order to prove this equivalence we need to convince ACL2 that it can evaluate `(everyotherp (interleave x (rev2 y)) nil)` in the same way that it evaluates `(everyotherp (interleave x y) nil)`. To do this we can use `conjecture-1-simp`.

Listing 10: y-rev2-y-equivalence

```
(defthm y-rev2-y-equivalence
  (implies (and (lonp x)
                (losp y)
                (equal (len2 x) (len2 y)))
           (equal (everyotherp (interleave x y) nil)
             (everyotherp (interleave x (rev2 y)) nil)))
  :hints (("Goal"
           :use (:instance conjecture-1-simp (x x) (y (rev2 y))))))
```

This doesn't go through on it's own, so we must also use `lemma-1` to include `(losp (rev2 y))` in the context, since `(rev2 y)` is still a `los` according to `lemma-1`. This gives the following which evaluates to true.

Listing 11: y-rev2-y-equivalence

```
(defthm y-rev2-y-equivalence
  (implies (and (lonp x)
                (losp y)
                (losp (rev2 y))
                (equal (len2 x) (len2 y)))
           (equal (everyotherp (interleave x y) nil)
                  (everyotherp (interleave x (rev2 y)) nil)))
  :hints (("Goal"
           :use (:instance conjecture-1-simp (x x) (y (rev2 y))))))
```

We can now use `conjecture-1-simp` along with the rewrite rule we just created to revise `conjecture-1` to it's final form.

Listing 12: Conjecture 1

```
(defthm conjecture-1
  (implies (and
            (lonp x)
            (losp y)
            (equal (len2 x) (len2 y)))
           (everyotherp (interleave x (rev2 y)) nil))
  :rule-classes ((:rewrite))
  :hints (("Goal"
           :in-theory (disable conjecture-1-simp)
           :use (:instance conjecture-1-simp (x x) (y y)))))
```

ACL2 is now able to accept our conjecture.

4 Conclusion

In this paper, we proved the property that when two equal sized decks are shuffled together, the resulting deck will have cards from each deck in an alternating pattern, regardless if one of the decks was reversed beforehand. This property was born from our curiosity into the

"Gilbreath Shuffle", where a deck is split, one half reversed and riffle shuffled together. Randomness is hard to prove and reason about, so we used the interleave function from CS2510 to simulate perfect riffle shuffles. We used lists of different data types to represent the two halves of the deck and we used a custom predicate, `everyotherp` to determine if the goal had been reached.

In order to prove our conjecture, we required several lemmas to link our conjecture to a simplified version that the ACL2 theorem prover could prove trivially.

First we created a simplified version of the conjecture, `conjecture-1-simp` which didn't include the reversal of one of the lists. Then we created a rewrite rule that stated the equality:

```
(equal (everyotherp (interleave x y) nil)
       (everyotherp (interleave x (rev2 y)) nil))
```

In order to show this equality, we created `lemma-1` to show that the reverse of a list of symbols is still a list of symbols.

Once we created these stepping stones between our conjecture and the simplified conjecture, ACL2 was able to accept our proof.

5 Appendix

Source Code

References

[1] R Diaconis, P Graham. *The Mathematical Ideas That Animate Great Magic Tricks.* Princeton University Press, 2011.