[redacted]
CS2800

Pascal's Triangle in ACL2s

Introduction

In this project, we are working with theorems and proofs surrounding Pascal's triangle in ACL2s. Pascal's triangle consists of rows of numbers, where the first row has a length of one, and each subsequent row in the triangle has one more number than the previous row, thus forming a triangular shape, as shown in figure 1 below. The values in each row begin and end with 1 Each nth value of a row is the sum of the nth and (n 1)th value of the previous row

We have two functions that generate Pascal's triangle with *n* rows: one that uses an accumulator and one that is purely recursive. Using ACL2s, we prove two properties of Pascal's triangle and the relationship between the two functions Firstly, we show that the second diagonal of Pascal's triangle represents the counting number sequence. Secondly, we verify that the result of the function is always in a triangle shape -- each row one longer than the previous starting from 1. The last theorem proves that both the purely recursive and accumulator versions of the function are equivalent

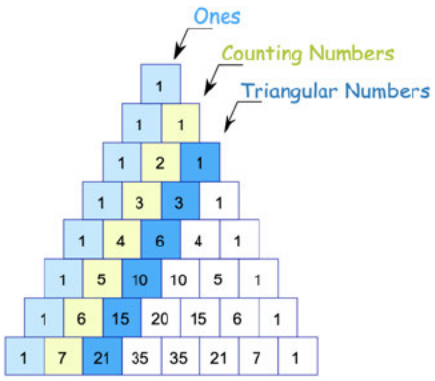

Figure 1 - Example of Pascal's triangle with 8 rows

Overall Structure

We do not use any special data structures for these proofs — Pascal's triangle is represented as a *llon* where each row is its own *lon* (list of nats). In our representation of Pascal's triangle, the list is ordered such that the bottom-most, or newest or longest, rows are first in the list. The only other special ACL2s features we use are in the second theorem, where we use hints to force ACL2s to use a certain lemma to prove a subgoal in another lemma. There are two main functions that create Pascal's triangle with n rows with the signature *nat* → *llon* One is purely recursive, called [redacted] and the other, [redacted] has the same signature but

wraps an accumulator helper Both these functions also utilize the functions [redacted] with signature *llon* → *lon* which gets the next row of a given triangle, and [redacted] with signature *lon* → *lon* that creates the next row of a given triangle given the previous row. The structure of the purely recursive and accumulator functions are similar to that of the fibonacci functions from Homework 10.

Walkthrough 1: Second Diagonal

The second diagonal of a Pascal's triangle, or by our definition, the second value of every row, is the counting sequence. The second value of the nth row in the triangle is equivalent to n-1 for all rows except the first, as shown by the first theorem

```
(defthm second-diagonal
    (implies (and (natp n) (   n 0))
        (equal (nth 1 (car (pascal triangle (1+ n)))) n)))          (Thm 1)
```

Since each value in Pascal's triangle is a sum of two values from the previous row, we first proved a [redacted] that the second value in a row is specifically the sum of the first and second values from the row below. Since the first value in each row is always 1 by definition of the *new-row* function, ACL2s can prove this lemma by induction.

Walkthrough 2: Triangle Shape

Pascal's triangles are always in a format such that the last row in the list is length 1, and each successive row is one element longer, which is the property proven in our second theorem. We wrote a function that checks for the triangular shape of a given *llon*, [redacted] and use this function in the second theorem

```
(defthm triangle is triangle
    (implies (natp n)  (is-pascal-shape (pascal-triangle n))))      (Thm. 2)
```

For ACL2s to accept this theorem, we first prove two lemmas: one to show that the longest row in a Pascal's triangle is equal to the number of rows in the triangle as a whole ([redacted]) and another regarding the length of the *row-helper* output which was needed for a subgoal of that lemma ([redacted]). This theorem holds true for a Pascal's triangle of any size, including if input *n* is zero

Walkthrough 3: equal-pascal with Accumulator

For this theorem, we prove that *pascal-triangle* and *pascal-triangle-acc* are equivalent:

```
(defthm equal-pascal
    (implies (natp n)
        (equal (pascal triangle n)  (pascal triangle acc n))))      (Thm 3)
```

The definition of *pascal triangle acc*, accumulator wrapper function, is simply a call to the actual function whose signature includes the accumulator, which stores the triangle created so far: [redacted]. Thus, in order to prove this theorem, we must first prove that *pascal triangle* is also equivalent to *pascal triangle acc h*

```
(equal (pascal triangle n)  (pascal triangle acc h n '()))          (Lemma 3 1')
```

In *pascal-triangle-acc-h*, the accumulator stores the rows of the triangle that have been built already, while the parameter *n* serves as a counter for how many more rows to add to the accumulator Therefore, to build a triangle with *n* rows, if there are *c* rows left to add, then the accumulator is a pascal triangle of n - c rows. A generalized lemma of Lemma 3.1 is needed to relate the accumulator to *pascal-triangle*:

```
(defthm equal-pascal-acc
        (implies (and (natp n) (natp c) (>= n c))
                 (equal (pascal-triangle n)
                        (pascal-triangle-acc-h c (pascal-triangle (- n c))))))          (Lemma 3.1)
```

After accepting Lemma 3.1, Thm 3 is also successfully proved by ACL2s.

Personal Progress

After proving the third theorem for equivalency between the recursive and accumulator functions, we initially attempted to prove the property of the third diagonal in Pascal's triangle as a triangular sequence. This ultimately was quite difficult to do (the remnants of these attempts are in the commit history), but in the process, we were able to prove the properties shown by the first and second theorems instead Throughout the process of trying to prove a theorem, we tried to come up with different lemmas based on the output of a failed proof attempt by ACL2s. These outputs were often very verbose, and since our functions for creating Pascal's triangle required many helper functions, and in cases of checking some nth value in the triangle, some of the lemmas we tried were complex, with many *car* and *cdr* calls. Sometimes, we misinterpreted these complex statements and spent time trying to prove something that was actually untrue before realizing the error.

In other cases, we would encounter roadblocks in proving theorems due to the structure of the functions we had written. For instance, our first iteration of the *is-pascal-shape* function called a helper function with a counter argument that made it difficult for ACL2s to prove Thm 2 We had to take a step back from trying different lemmas and rewrite the function instead as the solution to ACL2s accepting the theorem.

Conclusion and Next Steps

This project topic came from a discussion following Homework 10 with Fibonacci numbers We wanted to work with an interesting mathematical concept and prove properties

related to that concept, settling on Pascal's triangle The theorems we proved certify different properties of the triangular shape and values of Pascal's triangle and compare the equivalency of accumulator and non accumulator versions of the same function.

Our proofs for the property relating to the second diagonal of Pascal's triangle and the associated relation about the values in a row as a sum of other values could serve as the basis for a future theorem relating the third diagonal to a triangular sequence in ACL2s. The third diagonal of Pascal's triangle comes partially from summing values in the second diagonal, so having a theorem for the values of the second diagonal would be useful in proving the third

Appendix- ACL2s file

```
(set gag mode nil)
:brr t

#|
Pascal's triangle is a symmetric triangle of nats, where each subsequent
row of the triangle has a length of one longer than the row below it  Each row
starts and ends with a 1, and each value in the triangle is the sum of the two
numbers below it, as described here:
https://www.mathsisfun.com/pascals-triangle.html

In our representation, Pascal's triangle is represented as a list of list of
nats,
for instance: (pascal-triangle 4) => ((1 3 3 1) (1 2 1) (1 1) (1))
|#

(defdata lon (listof nat))
(defdata llon (listof lon))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; PASCAL'S TRIANGLE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Helper for new-row. Generates the next row of a pascal's triangle given the
previous row,
;; without the leading 1. Each value is the sum of the the two values "below"
it in the previous row
(definec row-helper (prev-row :lon) :lon
  (cond ((endp prev-row) nil)
        ((endp (cdr prev-row)) '(1))
        (t (cons (+ (car prev row) (cadr prev row))
                 (row-helper (cdr prev-row))))))

;; Creates the next row of pascal's triangle given the triangle so far, where
the row starts
;; and ends with 1s, and each value is the sum of the the two values "below"
it in the previous row
(definec new-row (triangle-list :llon) :lon
  (cons 1 (row-helper (car triangle-list))))

;; Creates a Pascal's triangle with n rows as a list, where each element is a
list
;; representing the numbers in one row of the triangle. The first item is the
;; bottom most (longest) row of the triangle
(definec pascal triangle (n  nat)  llon
  (cond ((zp n) '())
        (t (let ((rest-triangle (pascal-triangle (- n 1))))
                 (cons (new-row rest-triangle) rest-triangle)))))

(check= (row-helper '(1 3 3 1)) '(4 6 4 1))
(check= (row-helper '(1)) '(1))

(check  (new row '((1     1) (1   1) (1 1) (1))) '(1 4 6 4 1))
(check  (new row '((1))) '(1 1))

(check= (pascal-triangle 0) '())
(check= (pascal-triangle 1) '((1)))
```

```
(check  (pascal triangle  ) '((1 1) (1)))
(check= (pascal-triangle 5) '((1 4 6 4 1) (1 3 3 1) (1 2 1) (1 1) (1)))


;;;;;;;;;;;;;;;;;;;;;; PASCAL'S TRIANGLE WITH ACCUMULATOR ;;;;;;;;;;;;;;;;;

(definec pascal-triangle-acc-h (n :nat acc :llon) :llon
  (if (zp n)
    acc
    (pascal triangle acc h (  n 1) (cons (new row acc) acc))))

;; Creates a Pascal's triangle with n rows as a list, where each element is a
list
;; representing the numbers in one row of the triangle  The first item is the
;; bottom most (longest) row of the triangle
;; Is equivalent to pascal-triangle, but uses an accumulator
(definec pascal-triangle-acc (n :nat) :llon
  (pascal-triangle-acc-h n '()))

(check= (pascal-triangle-acc 0) '())
(check= (pascal-triangle-acc 1) '((1)))
(check= (pascal-triangle-acc 2) '((1 1) (1)))
(check  (pascal triangle acc 5) '((1 4 6 4 1) (1     1) (1   1) (1 1) (1)))
(check= (pascal-triangle-acc 6) (pascal-triangle 6))


;;;;;;;;;;;;;;;;;;;;;;;;;; THEOREM: SECOND DIAGONAL ;;;;;;;;;;;;;;;;;;;;;;;;
#|
The second diagonal of pascal's triangle is the counting number sequence.
Specifically in per this data definition, the second element of each row will
decrease by 1
and therefore, the second element of each nth row is the value n 1
Only triangles with at least 2 rows will have a second diagonal.

(defthm second-diagonal
  (implies (and (natp n) (> n 0))
           (equal (nth 1 (car (pascal triangle (1+ n)))) n)))
|#

;; Lemma 1.1: prove that the second element in a row of Pascal's triangle is
(defthm second diagonal sum
  (implies (and (natp n) (> n 0))
           (equal (cadar (pascal-triangle (1+ n)))
                  (+ (cadar (pascal-triangle n))
                     (caar (pascal-triangle n))))))

;; Thm 1: the second value in each row of a pascal's triangle is one less than
;; the row number it is in (there is no second diagonal for row 1)
(defthm second-diagonal
  (implies (and (natp n) (  n 0))
           (equal (nth 1 (car (pascal-triangle (1+ n)))) n)))



;;;;;;;;;;;;;;;;;;;;;; THEOREM  TRIANGLE SHAPE ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Determines whether the given llon is in the shape of a triangle--
```

```
;; each element in the list (each row) has length one longer then the next
element
;; ending at length 0
(definec is-pascal-shape (tri :llon) :bool
  (cond ((endp tri) (equal (llen (car tri)) 0))
        (t (and (equal (llen (car tri)) (llen tri)) (is pascal shape (cdr
tri))))))

(check= (is-pascal-shape (pascal-triangle 0)) t)
(check  (is pascal shape (pascal triangle 4)) t)
(check= (is-pascal-shape '((1 4 1) (1 3 3 1) (1))) nil)

;; Lemma 2.1: the row-helper function returns a row of the same length as the
input
(defthm row helper len
  (implies (lonp l)
           (equal (llen (row-helper l))
                  (llen l))))

;; Lemma 2.2: the length of the longest row in a triangle is the same as the
number of
;; rows in the triangle
(thm
 (implies (natp n)
          (let ((tri (pascal-triangle n)))
            (equal (llen (car tri)) (llen tri))))
 :hints (("Subgoal *1.1/5" :use row-helper-len)))

;; Thm 2: a pascal triangle is always in triangle shape
(defthm triangle-is-triangle
  (implies (natp n)
           (is pascal shape (pascal triangle n))))



;;;;;;;;;;;;;;;;;;;; THEOREM: PASCAL VS ACCUMULATOR ;;;;;;;;;;;;;;;;;;;;;;;;
#|
THEOREM: two functions that generate Pascal's triangle, one purely recursive
and one
using an accumulator, are equivalent.

(defthm equal-pascal
  (implies (natp n)
           (equal (pascal-triangle n) (pascal-triangle-acc n))))
|#

;; Lemma 3.1: Relating the accumulator to the recursive pascal triangle
function
(defthm equal-pascal-acc
  (implies (and (natp n) (natp c) (   n c))
           (equal (pascal-triangle n) (pascal-triangle-acc-h c
(pascal-triangle (- n c))))))

;; Thm 3: equal-pascal as described above
(defthm equal pascal
  (implies (natp n)
           (equal (pascal-triangle n) (pascal-triangle-acc n))))
```