CS2800 Project 2 Report

Proving equality between accumulator and non-accumulator implementations of the nth triangular and factorial number

[redacted]

[redacted]

Due Date: 12/14/2020

Introduction

The goal of this project is to operate an automated theorem prover to demonstrate the equality between an accumulator and non-accumulator recursive function. As of the submission of this report, the ACL2s software system has been guided to prove equality between that the accumulator and non-accumulator functional representations for both the triangular numbers and factorials (Figures 1, 2 and 15).

$$\sum_{i=1}^{n} i$$

Figure 1: Triangular Numbers in Series Notation

$$\prod_{i=1}^{n} i$$

Figure 2: Factorial Numbers Formula

By guiding ACL2s through the unraveling of the induction scheme, both proofs were completed. Though we had completed our primary goals, we also attempted to encompass other pairs of accumulator and non-accumulator recursive functions. By broadening the proof scheme that was used, one could potentially make analogous proofs for more generalized functions. By taking this approach, a combination of these first two function pairs can be used to write a function with an inductive step of both addition and multiplication (Figure 3).

$$F(n) = n \cdot (\beta F(n-1) + \alpha)$$

Figure 3: Generalized Non-Accumulator Inductive Step With Constant Natural Coefficients α and β

Unfortunately, we were not able to complete the generalized combination proof past pen-and-paper, due to the fact that we did not have time to determine how to steer ACL2s to the correct conclusion.

Walkthrough

We defined the nth triangular number (Figure 1) with three equivalent functions (Figure 2). Though each one counts its input *x* and then recurs on *x* - 1, the way they achieve this functionality differs. The simplest recursive implementation, ‘`non-acc-fn-sum`,’ simply adds and recurs. Meanwhile the accumulator implementation, ‘`acc-fn-sum`’ will record the sum as an argument to the function, returning it when *x* is 0. Furthermore, we defined an interface, ‘acc-fn-sum-interface,’ that starts the accumulator implementation with a starting accumulator value of 0. This interface is convenient, as we reference both implementations with identical signatures.

```
(definec acc-fn-sum (x :nat acc :nat) :nat
  (cond ((zp  x) acc)
        (t       (acc-fn-sum (1- x) (+ x acc)))))

(definec non-acc-fn-sum (x :nat) :nat
  (cond ((zp  x) 0)
        (t       (+ x (non-acc-fn-sum  (1- x))))))

(definec acc-fn-sum-interface (x :nat) :nat
  (acc-fn-sum x 0))
```

Figure 4: Triangular Number Function Definitions in Lisp

We claim that (`non-acc-fn-sum n`) and (`acc-fn-sum-interface n`) both represent the nth triangular number. As a result, the output of both functions should be equivalent for all values of 'n,' where 'n' is some natural number (Figure 3).

```
(defthm rec=acc-sum
   (implies (natp x)
            (= (non-acc-fn-sum x)
               (acc-fn-sum-interface x))))
```

Figure 5: Theorem Stating Equality Between Accumulator and Non-Accumulator Sum Implementations

However, ACL2s cannot prove this theorem directly. Although it is able to identify the correct induction scheme, it is unable to unravel the accumulator function one layer down and employ the inductive step (Figure 4), which we would do in a written proof.

```
(acc-fn (1- x) c)
==
(+ c (acc-fn (1- x) 0))
==
(+ c (non-acc-fn (1- x)))
```

Figure 6: Example of Logical Steps that ACL2s Will Not Make Without Assistance

We must assist ACL2s by proving lemmas to perform the steps shown above. The `sum-associativity-zero` theorem is required for ACL2s to rewrite some subgoals in the proof, while `sum-associativity-general` is the equivalent theorem in the general case for acc (Figure 5). The general theorem is required to prove the specific zero case that we are interested in.

```
(defthm sum-associativity-general
  (implies (and (natp a) (natp b) (natp c))
           (equal (acc-fn-sum a (+ c b))
                  (+ b (acc-fn-sum a c)))))


(defthm sum-associativity-zero
  (implies (and (natp a) (natp b) (= acc 0))
           (equal (acc-fn-sum a b)
                  (+ b (acc-fn-sum a acc)))))
```

Figure 7: Theorems Stating Additive Associativity Through Accumulator Function

With both supporting lemmas admitted by ACL2s, we may finally admit `rec=acc-sum` and prove our claim that the accumulator and non-accumulator functions are equivalent.

Our approach to the proof for factorials is similar. To begin, we define the nth factorial number with three different equivalent functions (Figure 8) . A non-accumulator function '`non-acc-fun-mult`' is the most basic function of the three. There is also an accumulator implementation called '`acc-fun-mult`,' and finally an interface function which simply calls '`acc-fun-mult`,' initially setting the accumulator value equal to 1.

```
(definec acc-fn-mult (x :nat acc :nat) :nat
  (cond ((zp  x) acc)
        (t       (acc-fn-sum (1- x) (* x acc)))))

(definec non-acc-fn-mult (x :nat) :nat
  (cond ((zp  x) 1)
        (t       (* x (non-acc-fn-sum  (1- x))))))

(definec acc-fn-mult-interface (x :nat) :nat
  (acc-fn-sum x 1))
```

Figure 8: Factorial Number Function Definitions in Lisp

In order to prove that the accumulator and non-accumulator versions are equal, we must once again guide ACL2s to unravel our induction scheme (Figure 9). In order to do so, we need to prove associativity for the general case, and specifically for the base case, when the accumulator is equal to 1 (Figure 10).

```
(acc-fn (1- x) c)
==
(* c (acc-fn (1- x) 1))
==
(* c (non-acc-fn (1- x)))
```

Figure 9: Example of Logical Steps that ACL2s Will Not Make Without Assistance for Factorials

```
(defthm mult-associativity-general
  (implies (and (natp a) (natp b) (natp c))
           (equal (acc-fn-mult a (* c b))
                  (* b (acc-fn-mult a c)))))


(defthm mult-associativity-one
  (implies (and (natp a) (natp b) (= acc 1))
           (equal (acc-fn-mult a b)
                  (* b (acc-fn-mult a acc)))))
```

Figure 10: Theorems Stating Multiplicative Associativity Through Accumulator Function

Finally, we have established the lemmas needed to allow ACL2s to prove that the non-accumulator and accumulator functions are equal (Figure 11).

```
(defthm rec=acc-mult
   (implies (natp x)
             (= (non-acc-fn-mult x)
                (acc-fn-mult-interface x))))
```

Figure 11: Equality Between Accumulator and Non-Accumulator Multiplication Implementations

Personal Progress

It took us many tries to write theorems that were admissible in ACL2s, and we frequently needed to step back and attack the proof from another angle. Throughout many of our first attempts at our triangular numbers proof (Figure 4), ACL2s was “rewriting subgoals equivalent to a top-level goal.” There were a few approaches to solving this problem.

At the time, we were able to discern the issue might have been that ACL2s required a hint on how to induct on the input. On the other hand, it may have also been an issue of reducing/rewriting the subgoal into a different form. We first attempted to spell out the induction scheme and use `:hints`, which failed.

From scanning the proof logs, we were able to find out where the proof was getting stuck. We simplified the theorem by including an assumption that would allow the theorem to be admissible.

```
(defthm rec=acc-sum-with-assumption
   (implies (and (natp x) (natp acc) (zp acc)
                 ;acl2 only works if we spell it out
                (implies (and (natp a) (natp b))
                          (equal (acc-fn-sum a b)
                                 (+ b (acc-fn-sum a 0)))))
              (equal (non-acc-fn-sum x) (acc-fn-sum x acc))))
```

Figure 12: `rec-acc-sum` with Critical Assumption Required for Admissibility

From this implication we were able to set a clear goal for ourselves. All we needed to do is prove this implication as a theorem. However, this turned out to be very difficult due to rewrite rules. This assumption in Figure 12 was developed into a theorem and proven with the assistance of a more general theorem (Figure 7).

Confusingly, despite the fact that we were able to prove the assumption in Figure 12 as a theorem, `rec=acc-sum` was still not admissible. We encountered a double-rewrite loop error that caused infinite recursion. We were able to discover and debug this error with:

```
(set-gag-mode nil)
:brr t
(cw-gstack :frames 30)
```

Figure 13: Commands to Debug and Discover ACL2s Double-Rewrite Error

This was the most perplexing part of the process, and it required an incriciate knowledge of how rewrite rules in ACL2s operate. To demonstrate, Figure 14 contains two versions of `sum-associativity`, where the former leads to an infinite rewrite loop and the latter does not. Differences between the functions are highlighted for ease of reading.

```
(defthm sum-associativity-zero-bad
  (implies (and (natp a) (natp b))
           (equal (acc-fn-sum a b)
                  (+ b (acc-fn-sum a 0)))))

(defthm sum-associativity-zero
  (implies (and (natp a) (natp b) (= acc 0))
           (equal (acc-fn-sum a b)
                  (+ b (acc-fn-sum a acc)))))
```

Figure 14: Two Logically Equivalent Theorems Contrasted by Admissibility through ACL2s

Due to how rewrite rules work in ACL2s, the literal value 0 will be eagerly rewritten, causing an infinite loop between application of the rewrite rule and simplification of the goal. Using `:do-not '(simplify)` or disabling the theorem does not alleviate the issue. The application of `double-rewrite` was attempted but was also unsuccessful. Our current solution was developed through careful research on rewrite rules[1] and experimentation. We suspect that this stems from an issue discussed briefly in Section 6.9,[2] which suggests that constants ought to be replaced by variables.

This issue was again encountered when developing a proof for the factorial numbers, but as we had already solved this issue once for the triangular numbers, it was relatively easy to adapt a solution for our new proof.

Conclusions and Next Steps

This project's scope can be expanded to any specific associative function, which can then be approached in a similar manner to that of the completed proofs pertaining to multiplication and addition. As an example, one could prove this for the exponentiation of natural numbers. Furthermore, after our attempt at a generalized function was not fully successful, multiplication with a generalized constant was also attempted (Figure 16 of Appendix) by including the natural coefficient as a parameter of each function, while following the steering for the factorial proof. Though its general theorem for associativity passed, it failed in the base case of when the accumulator was equal to 1. Nevertheless, due to the fact that this was not one of our previously determined goals, we did not feel that it would be appropriate to thoroughly discuss this in a report that was already relatively lengthy.

By steering ACL2s to complete our proof goals, we were able to achieve a stronger understanding of what it entails to use a theorem prover. We learned to distinguish between selecting the correct induction scheme and actually being able to unravel it, which a computer has significantly more difficulty doing conceptually. Furthermore, we have been able to better understand the importance of the associative properties of a recursive step's computation when forming similar proofs; this is a theme that could be expanded past the scope of the natural numbers and to other data types such as lists.

[1] www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____REWRITE.

[2] https://via.hypothes.is/https://www.khoury.northeastern.edu/~pete/courses/Logic-and-Computation

References

"ACL2s Documentation." *XDOC (Rewrite)*, www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/?topic=ACL2____REWRITE.

"Reasoning About Programs." *Panagiotis Manolios,* https://via.hypothes.is/https://www.khoury.northeastern.edu/~pete/courses/Logic-and-Computation/2020-Spring/rapind.pdf.

Appendix

Figure 15: Proof of Equality Between Accumulator and Non-Accumulator Representations of both Triangular and Factorial Numbers in ACL2s

```
(definec acc-fn-sum (x :nat acc :nat) :nat
  (cond ((zp  x)  acc)
      (t          (acc-fn-sum (1- x) (+ x acc)))))

(definec non-acc-fn-sum (x :nat) :nat
  (cond ((zp  x) 0)
      (t    (+ x (non-acc-fn-sum  (1- x))))))

(definec acc-fn-sum-interface (x :nat) :nat
  (acc-fn-sum x 0))

; Sanity check, does test? find any counterexamples? No. :D
(test? (implies (and (natp x) (natp acc) (zp acc))
            (equal (non-acc-fn-sum x) (acc-fn-sum x acc))))

#|
(set-gag-mode nil)
:brr t
(cw-gstack :frames 30)
|#


#|
Description:
In order to prove sum-associativity-zero, we must prove associativity in
the
general case for our accumulator function.

That is, adding to the accumulator is the same as adding to the result of
the function.

|#
(defthm sum-associativity-general
  (implies (and (natp a) (natp b) (natp c))
            (equal (acc-fn-sum a (+ c b))
                  (+ b (acc-fn-sum a c)))))
```

```
#|
Description:
sum-associativity-zero is required to prove a specific case in rec=acc-sum.
In paticular this handles the case in which c+b = 0, or when acc = 0: the
base case.
This unwinding of the recursion is required to be explicitly handled as a
theorem
for ACL2s to prove rec=acc-sum.

|#
(defthm sum-associativity-zero
  (implies (and (natp a) (natp b) (= acc 0))
           (equal (acc-fn-sum a b)
                 (+ b (acc-fn-sum a acc)))))

#|
Description:
rec=acc-sum claims that the non-accumulator version of the nth triangular
number
equals the accumulator version describing the same function for all natural
numbers x.

|#
(defthm rec=acc-sum
   (implies (and (natp x))

           (= (non-acc-fn-sum x)
                 (acc-fn-sum-interface x))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;

(definec acc-fn-mult (x :nat acc :nat) :nat
  (cond ((zp  x)  acc)
      (t          (acc-fn-mult (1- x) (* x acc)))))

(definec non-acc-fn-mult (x :nat) :nat
  (cond ((zp  x) 1)
      (t    (* x (non-acc-fn-mult  (1- x))))))

(definec acc-fn-mult-interface (x :nat) :nat
  (acc-fn-mult x 1))

; Sanity check, does test? find any counterexamples? No. :D
```

```
(test? (implies (and (natp x) (natp acc) (= 1 acc))
            (equal (non-acc-fn-mult x) (acc-fn-mult x acc))))


#|
Description:
In order to prove mult-associativity-one, we must prove associativity in
the
general case for our accumulator function.

That is, adding to the accumulator is the same as adding to the result of
the function.
|#


(defthm mult-associativity-general
  (implies (and (natp a) (natp b) (natp c))
            (equal (acc-fn-mult a (* c b))
                  (* b (acc-fn-mult a c)))))

#|
Description:
mult-associativity-one is required to prove a specific case in rec=acc-sum.
In paticular, this handles the case in which c * b = 1, or when acc = 1,
which is the base case.
This unwinding of the recursion is required to be explicitly handled as a
theorem
for ACL2s to prove rec=acc-sum.
|#

(defthm mult-associativity-one
  (implies (and (natp a) (natp b) (= acc 1))
            (equal (acc-fn-mult a b)
                  (* b (acc-fn-mult a acc)))))

#|
Description:
rec=acc-mult claims that the non-accumulator version of the nth factorial
equals the accumulator version describing the same function for all natural
numbers x.

|#
(defthm rec=acc-mult
  (implies (and (natp x))

           (= (non-acc-fn-mult x)
                (acc-fn-mult-interface x))))
```

Figure 16: Attempt at Generalized Multiplicative Proof
(This failed at the 'gen-mult-associativity-one' theorem.)

```
(definec acc-fn-gen (x :nat acc :nat b :nat) :nat
  (cond ((zp  x)    acc)
        (t        (acc-fn-gen (1- x) (* x (* b acc)) b))))

(definec non-acc-fn-gen (x :nat b :nat) :nat
  (cond ((zp  x) 1)
        (t       (* x (* b (non-acc-fn-gen (1- x) b) )))))

(definec acc-fn-gen-interface (x :nat b :nat) :nat
  (acc-fn-gen x 1 b))

; Sanity check, does test? find any counterexamples? No. :D
(test? (implies (and (natp x) (natp acc) (= acc 1) (natp b))
              (equal (non-acc-fn-gen x b) (acc-fn-gen x acc b))))


#|
Description:
In order to prove mult-associativity-one, we must prove associativity in
the
general case for our accumulator function.

That is, adding to the accumulator is the same as adding to the result of
the function.
|#


(defthm gen-mult-associativity-general
  (implies (and (natp a) (natp b) (natp c))
           (equal (acc-fn-gen a (* c b) b)
                  (* b (acc-fn-gen a c b)))))
```

```
#|
Description:
mult-associativity-one is required to prove a specific case in rec=acc-sum.
In paticular, this handles the case in which c * b = 1, or when acc = 1,
which is the base case.
This unwinding of the recursion is required to be explicitly handled as a
theorem
for ACL2s to prove rec=acc-sum.
|#

(defthm gen-mult-associativity-one
  (implies (and (natp a) (natp b) (= acc 1))
           (equal (acc-fn-gen a b b)
                  (* b (acc-fn-gen a acc b)))))

#|
Description:
rec=acc-mult claims that the non-accumulator version of the nth factorial
equals the accumulator version describing the same function for all natural
numbers x.

|#
(defthm rec=acc-gen
   (implies (and (natp x) (natp b))

              (= (non-acc-fn-gen x b)
                 (acc-fn-gen-interface x b))))
```