# Length of a Powerlist

[redacted]

December 2020

## 1 Introduction

A powerset $P(S)$ of a set $S$ is the set of all subsets including the empty set and $S$ itself. We adapt powersets to work with lists. We define the *powerlist* of a list $L$ to be a list of all sublists of $L$, where a sublist is a list that only contains elements that appear in the original list, and these elements appear in the same order as they do in the original list. As an example, the powerlist of `(1 2 3) is '(() (1) (2) (3) (1 2) (2 3) (1 3) (1 2 3))`.

A well known property of powersets is that for powerset $P(S)$, $|P(S)| = 2^{|S|}$. In this project, we aim to prove an analogous property of powerlists. Namely, if the length of a list is $n$, then length of that list's powerlist is $2^n$.

## 2 Overall Structure

We started off by contextualizing the proof by defining the relevant functions required to generate a powerlist in ACL2s.

### 2.1 Contextualization of the Proof

We first started off by implementing the functions required to generate a powerlist in ACL2s. Along with the `len2` and `app2` functions we defined previously in class, we created two new functions: `expt2` and `cons-each`.

```
;; raises x to the y
(definec expt2 (x :nat y :nat) :nat
  (if (zp y)
    1
    (* x (expt2 x (1- y)))))

;; Conses x onto each element in l
(definec cons-each (x :all l :tl) :tl
  (if (endp l)
    l
    (cons (cons x (car l)) (cons-each x (cdr l)))))
```

The `expt2` function takes in two natural numbers $x$ and $y$ and returns $x^y$ as a natural number. For example, `(expt2 2 5)` would return 32, or $2^5$. The `cons-each` function takes in any argument $x$ along with a list $l$ and returns a list with $x$ "consed" onto each element of $l$. For example, `(cons-each 'x '((1 2) (2 3))) returns '((x 1 2) (x 2 3))`. Using `cons-each` allowed us to implement the function `powerlist` that takes in a list $l$ and returns a list of all the sublists of $l$.

```
;; Returns a list of all the sublists of l.
;; If you think of l as a set, this function returns the powerset of l.
(definec powerlist (l :tl) :tl
  (if (endp l)
    (list nil)
    (let ((ps (powerlist (cdr l))))
      (app2 ps (cons-each (car l) ps)))))
```

Finally, we contextualized our proof in ACL2s:

```
(thm (implies (tlp l)
              (equal (len2 (powerlist l))
                     (expt2 2 (len2 l)))))
```

We tackled this proof by using induction on $l$.

## 3 Proof

Like any proof by induction, we started off by writing down the rudimentary proof obligations.

1.
   ```
   (implies (not (tlp l))
         (implies (tlp l)
                  (equal (len2 (powerlist l))
                         (expt2 2 (len2 l)))))
   ```

   For proof obligation 1, the contract case, we show that the proof holds when $l$ is not a true-list.

2.
   ```
   (implies (and (tlp l)
                 (endp l))
         (implies (tlp l)
                  (equal (len2 (powerlist l))
                         (expt2 2 (len2 l)))))
   ```

   For proof obligation 2, we show that the proof holds when $l$ is an empty list.

3.
   ```
   (implies (and (tlp l)
                 (not (endp l))
   ```

```
             (implies (tlp (cdr l))
                      (equal (len2 (powerlist (cdr l)))
                             (expt2 2 (len2 (cdr l))))))
            (implies (tlp l)
                     (equal (len2 (powerlist l))
                            (expt2 2 (len2 l)))))
```

Then, for proof obligation 3, we show that the proof holds for a nonempty list $l$ under the assumption that the proof holds for `(cdr l)`. To do so, we require two lemmas. The first lemma aims to establish that if two lists $x, y$ are appended, the length of the resulting list is equal to the sum of the lengths of $x$ and $y$. The second lemma aims to establish the length of the list produced by `cons-each` is the same length as that of the original list.

## 3.1 Lemma 1: app-list-len

Lemma 1 establishes the relationship between `app2` and `len2`. Namely, if two lists are appended together, the length of the resulting list is equal to the sum of the lengths of the original two lists.

```
(implies (and (tlp x)
              (tlp y))
         (equal (len2 (app2 x y)) (+ (len2 x) (len2 y))))
```

By using induction on $x$, we write down the three proof obligations here:

1. Contract Case:

```
(implies (not (tlp x))
         (implies (and (tlp x)
                       (tlp y))
                  (equal (len2 (app2 x y)) (+ (len2 x) (len2 y)))))
```

2. Base Case:

```
(implies (and (tlp x)
              (endp x))
         (implies (and (tlp x)
                       (tlp y))
                  (equal (len2 (app2 x y)) (+ (len2 x) (len2 y)))))
```

3. Inductive Case:

```
(implies (and (tlp x)
              (not (endp x))
              (implies (and (tlp (rest x))
                            (tlp y))
```

```
                (equal (len2 (app2 (rest x) y)) (+ (len2 (rest x)) (len2 y)))))
            (implies (and (tlp x)
                          (tlp y))
                     (equal (len2 (app2 x y)) (+ (len2 x) (len2 y)))))
```

By satisfying the three proof obligations, we successfully proved that the length of a list of two lists appended together equals the sum of the lengths of the original two lists.

### 3.2 Lemma 2: cons-each-len

We then establish the relationship between `cons-each` and `len2` in Lemma 2 by showing that the length of the list produced by the `cons-each` function is the same length as that of the original list.

```
(implies (and (allp x) (tlp y))
         (equal (len2 y) (len2 (cons-each x y))))
```

By using induction on $y$, we write down the three proof obligations.

1. Contract Case:

```
(implies (not (tlp y))
         (implies (and (allp x) (tlp y))
                  (equal (len2 y) (len2 (cons-each x y)))))
```

2. Base Case:

```
(implies (and (tlp y)
              (endp y))
         (implies (and (allp x) (tlp y))
                  (equal (len2 y) (len2 (cons-each x y)))))
```

3. Inductive Case:

```
(implies (and (tlp y)
              (not (endp y))
              (implies (and (allp x) (tlp (cdr y)))
                       (equal (len2 y) (len2 (cons-each x (cdr y))))))
         (implies (and (allp x) (tlp y))
                  (equal (len2 y) (len2 (cons-each x y)))))
```

By satisfying the three proof obligations, we proved that the length of a list produced by the `cons-each` function is the same length as that of the original list.

## 3.3 Usage of Lemmas

We get stuck at line 270, because we cannot simplify `(len2 (let ((ps (powerlist (cdr l)))) (app2 ps (cons-each (car l) ps)))` further without establishing a relationship between `len2` of a list and `app2` of two lists. Therefore, we need Lemma 1 to progress in our proof.

Then we get stuck at 275, because we haven't established a relationship between `len2` and `cons-each`.

```
(+ (expt2 2 (len2 (cdr l))) (len2 (cons-each (car l) (powerlist (cdr l)))))
```

Lemma 2 advances the proof with the understanding that `cons-each` does not change the length of the list. After replacing

```
    (len2 (cons-each (car l) (powerlist (cdr l))))
```

with

```
    (len2 (powerlist (cdr l))
```

the proof can then continue by applying the inductive hypothesis.

## 3.4 Automation

ACL2s initially failed to prove our theorem. After writing out a complete pen-and-paper proof, we knew exactly what lemmas we needed. We began the process of automating the proof by contextualizing these two lemmas in ACL2s as rewrite rules using `defthm`:

```
(defthm app-list-len
  (implies (and (tlp x)
                (tlp y))
           (equal (len2 (app2 x y)) (+ (len2 x) (len2 y)))))

(defthm cons-each-len
  (implies (and (allp x)
                (tlp y))
           (equal (len2 y) (len2 (cons-each x y)))))
```

ACL2s managed to prove the two lemmas without any problems. Using these rewrite rules, ACL2s successfully accepted our overall theorem.

## 4 Conclusion

We decided to use powerlists as seen in our Fundamentals 1 class to try and automate a non-trivial, established proof in mathematics that requires the usage of lemmas. Powerlists have numerous applications, including describing parallel algorithms such as the Fast Fourier Transform and Batcher sorting scheme. By contextualizing the proof using ACL2s and incorporating relevant lemmas, we successfully automated the proof.

## 5 References

Misra, J. (1994). Powerlist: A structure for parallel recursion. ACM Transactions on Programming Languages and Systems, 16(6), 1737-1767. doi:10.1145/197320.197356