DRAFT

Proving Iterative Palindromization

A paper and project by Jack Cardin, Jalen Gonel, and Eric Yip
For CS2800

Table of Contents

1. **Abstract**
2. **Introduction**
3. **Approach**
4. **Progress made**
5. **Concerns and issues to date**
6. **Next Steps**

1. Abstract

In this paper, we set out to prove a property about iterative palindromization. Specifically, we are setting out to show that any list that is palindromized (turned into a palindrome), will always be a palindrome regardless of the number of times it has been palindromized. We will first elaborate in greater detail as to what the process of *palindromization* and *palindromizing* refers to, as well as what the scope of our project is. Then, we will discuss the approach we took to prove that this theory is true. After discussing our approach we will detail how much progress has been made. For more context as to the concerns and issues faced by the team throughout the project we then describe our original project idea and why we found its completion was deemed infeasible. Finally, we conclude by discussing what our next steps are from this point in the project to complete our proof and present it in full.

2. Introduction

The property of palindromes that we are going to prove is that any list that is palindromized any number of times will always be a palindrome. We will use the term ***palindromization*** and ***palindromize*** to refer to the process of transforming a given list into a palindrome by creating a new list where the first element of the given list is appended to the front and the back of the palindromization of the rest of the list. As an example:

```
( D O G )       ----- palindromized ----> ( D O G G O D )
( J O H N )     ----- palindromized ----> ( J O H N N H O J )
( A )           ----- palindromized ----> ( A A )
( 1 2 3 )       ----- palindromized ----> ( 1 2 3 3 2 1 )
(  )            ----- palindromized ----> (  )
```

An iterative palindromization would be any number of palindromizes repeatedly applied to a given list. For example, the iterative palindromization of DOG would be as follows:

```
( D O G )                                 ----- palindromized ---->
( D O G G O D )                           ----- palindromized ---->
( D O G G O D D O G G O D )               ----- palindromized ---->
( D O G G O D D O G G O D D O G G O D D O G G O D D O G G O D )
```

And this process could be repeated indefinitely in order to create longer palindromes. To prove the aforementioned iterative property of palindromes, we must prove that, for any list of elements and any number of iterations, the iterative palindromization of a list will always be a palindrome.

3. Approach

We approached this proof by first formulating a theorem for ACL2s to solve. Assuming the function necessary have already been implemented, we formatted the proof as to the implication:

```
(implies (and (tlp x) (posp n))
  (palindromep (iteratively-palindromize x n)))
```

In this statement, the term ”`palindromep`” represents a function that takes in a true list and will return a boolean value determining if the given list is a palindrome or not. The term “`iteratively-palindromize`” represents a function that takes in a true list and a positive integer, and palindromizes the given list n times. In english, the implication states “Given a true list x and a positive integer n, is x a palindrome no matter how many times it is palindromized?”

We will now declare the function ”`palindromep`”, which should determine if a list is a palindrome or not. The property that defines all palindrome lists is that the list has the same number of elements in the same order both forwards and backwards. Another way to express this property would be to state that a palindromic list is equal to the reverse of that same list. We can translate this property into ACL2s to get the function definition for ”`palindromep`”:

```
(definec palindromep (l :tl) :bool
  (equal l (rev2 l)))
```

This function will return true if the given list satisfies the condition of being equal to the reversed list, and returns false if the list does not satisfy this condition. The function `rev2` performs the same action as the built-in function `rev`, which reverses the given list.

The other function we must define is “`iteratively-palindromize`”, which, given a list and a positive integer n, should palindromize the list n times. To implement this function, we will use recursion with n as the accumulator. Every function call, we will palindromize the recursive call of this function, with the same list and an n decreased by 1. When n equals 1, we will return the palindromization of the list. Assuming that we have written a function that will palindromize a list a single time, translating this process into ACL2 will give us the definition for “`iteratively-palindromize`”:

```
(definec iteratively-palindromize (x :tl n :pos) :tl
  (if (equal n 1)
```

```
      (palindromize x)
    (palindromize (iteratively-palindromize x (- n 1)))))
```

Finally we must define the function "palindromize". To palindromize a list in ACL2, we will use a recursive scheme where we append the first element of the list to the front and back of the recursive call, which palindromizes the rest of the list. This can be written in code as:

```
(definec palindromize (ls :tl) :tl
  (if (endp ls)
      '()
    (cons (car ls) (app2 (palindromize (cdr ls)) (list (car ls))))))
```

The function app2 works the same as the built-in ACL2 function app, which appends two lists together. With all of these functions defined, we have successfully translated our properties of palindromes to ACL2s and are ready to begin our mechanized proof.

4. Progress made

We first tried to run our theorem in ACL2s to see if the proof could be completed trivially. Of course, ACL2s was not able to prove our theorem, and required some lemma in order to complete the proof. To determine the lemmas we would need, we wrote a pen-and-paper proof with the theorem as our conjecture (which can be found in the repository, in the file that contains our mechanized proof). We decided to prove this theorem by induction on the function `(iteratively-palindromize x n)`. Our contract case passed fine, but our base case seemingly required a lemma that showed equivalence between a palindromized list and the reverse of that palindromized list. We formed the lemma palindromize-id to prove this property, which can be written as the implication:

```
(implies (tlp x)
  (equal (palindromize x) (rev2 (palindromize x))))
```

We also had to prove this lemma by induction, this time on the function `(tlp x)`. The base case was simple to complete, but the inductive case proved to be very complex, requiring two other lemmas. The first lemma we needed to use was rev2-app2, which essentially states that reversing the append of two lists is equivalent to appending the reverse of the second list to the reverse of the first list. Translated into ACL2, this can be written:

```
(implies (and (tlp x) (tlp y))
  (equal (rev2 (app2 x y))
         (app2 (rev2 x) (rev2 y))))
```

Lemma rev2-app2 was trivial, and ACL2s was able to prove it without requiring any additional help. We also needed a second lemma that represents the associative property of appending lists to complete the induction on `(tlp x)`. The lemma app2-assoc states that, given

lists x, y, and z, appending x to the append of y and z is equivalent to appending the append of x and y to z. This lemma can be written in ACL2 as:

```
(implies (and (tlp x) (tlp y) (tlp z))
  (equal (app2 x (app2 y z))
         (app2 (app2 x y) z)))
```

This lemma was also trivial, with ACL2s being able to prove it without needing other lemmas. With the lemmas rev2-app2 and app2-assoc, we were able to complete the inductive proof for the lemma palindromize-id. With this new lemma, we were also able to finish the base case for the iterative palindromization conjecture. The induction case for the proof was also able to be completed, since it only required the palindromize-id lemma for help. With all cases of the inductive proof completed, our pen-and-paper proof of the iterative palindromization theorem was finished. We have added the required lemmas to our mechanized proof, which have all passed, excluding the theorem itself.

5. Concerns and Issues to date

To date, our group has faced many issues. Since starting out we have had to pivot away entirely from our original intentions. Originally, our group's plan was to do a QBF encoding for the game **dots and boxes**. For those unfamiliar, dots and boxes is a game in which a player draws a two-dimensional grid of dots on a piece of paper. 2 players take alternating turns drawing vertical or horizontal lines between two adjacent dots. A player scores a point whenever they draw a line such that a 1x1 box is made between 4 dots, going for another turn after every completed box. The game ends when all the dots on the grid have boxes drawn between them and no more lines can be drawn. The player with the most completed boxes at the end is declared winner.

Our original project idea was to produce a QBF encoding that would prove that on a 5x5 dots and boxes board, assuming both players played optimally and player 1 went first, player 1 would always win. We came up with an outline as to how this QBF project would play out.

```
QBF Outline:

There exists M0 (player 1's move)
such that for every M1 (player 2's move)
there exists M2 (player 1's move)
such that for every M3 (player 2's move)
...

there exists M(k-1) (player 1's final move)
such that for every M(k) (player 2's final move)
there exists a board position at step 1
and there exists a board position at step 2
...
and there exists a board position at step k
there exists an illegal move at step 1
and there exists an illegal move at step 2
```

```
...

and there exists an illegal move at step k
there exists a boolean variable z
such that:

An initial board setup
and transition rules for player 1
and transition rules for player 2
and a goal (player 1's score > player 2's score)
OR z is true

AND

z is true if and only if
no illegal moves are made

-------------------------------------------

Goal: (p1 > p2)

Z1 v Z2 v Z3 v Z4...

Z: (p1 > p2)

C1i and C2i and C3i and C4i

Cxy: (a player y (even / odd) gets the point for box x)

M(i-1) and (3 vars and 1 negated var)
AND
M(i) and (all 4 vars)

OR

... (3 more, 4 times total, one for each possible side)

Initial board setup:

All negated variables (no lines drawn, can be changed)

Transition rules:

M(i) = M(i-1) with only one negated variable unnegated

w1(i-1) iff w1(i)
w2(i-1) iff w2(i)
...
wn(i-1) xor wn(i)
```

While this certainly would be an interesting project, we ran into a few problems. One issue is that many QBF solvers only accept QBF input in conjunctive normal form. Considering our implementation required many different clauses within clauses, this would have made it difficult, if not impossible, for these solvers to accept our encoding. Furthermore, we found that it

was seemingly impossible to succinctly express the game's score using boolean expressions. Our encoding would require a clause for every combination of moves that would allow player 1 to gain more points than player 2, most of which would likely have to be manually created. The intended 5x5 grid would result in manual input of 40 factorial clauses. Even when downsizing the scenario to a 3x3 board we still would have to manually input 12 factorial clauses, or almost 500 million total. Even if we were able to automate the process of creating these clauses, inputting such large numbers of clauses into a solver would be problematic in many ways. Outright impossible to complete within a reasonable amount of time and out of scope for this class, we decided to pivot our project to the current palindromization proof.

As far as concerns and issues go relating to our current issue, there aren't many. We have almost entirely mechanized our proof. The pen-and-paper proof that we have written passed in the class proof checker. The only part of the mechanized proof that doesn't pass in ACL2s is the theorem itself, which enters an infinite loop trying to apply one of our lemmas to the proof.

6. Next Steps

While our proof currently passes the class proof checker it does not run in ACL2 due to the aforementioned infinite loop error. After fixing this error, the proof should be completed. The remaining steps after our proof being completed will likely be, in order:

1. Cleaning up code with robust tests and comments
2. Finish describing our findings in our research paper
3. Creating a presentation describing the overall course of this project and findings as a group

References:

Link to repository (contains ACL2 code for mechanized proof and full pen-and-paper proof):
https://github.com/jacrdn/CS2800-GROUP-PROJECT/tree/main