Using ACL2 to Prove the Correctness of Lowering Operators when $J = l + s$

Chen, Xiaohui
Department of Computer Science
The University of Texas at Austin

Abstract—Lowering Operators can be used to calcualte the Clebsch-Gordan coefficients in quantum coupled states. In this paper I am going to use ACL2 to verify that when the lowering operators operates on some initial quantum states, the coefficients match the Clebsch-Gordan coefficients and satisfy the probability rules. In addition, possible future research is proposed.

Index Terms—Quantum coupling, Clebsch-Gordan coefficient, angular momentum, ACL2, recursion and induction

1 Introduction

Clebsch-Gordan coefficients are mathematical symbol used to integrate products of three spherical harmonics [1]. Sphereical harmonics are some functions defined on the surface of a sphere. In quantum phycis, they are used to represent the boundaries of the particles. A spherical harmonic function is defined as $Y_l^{m_l}(\theta, \phi)$ where l and m_l are quantum numbers (details described later). Clebsch-Gordan coefficients represent the probability of the particles in those coupled states. According to fundamental rules of probability, the sum of all probabilities of the coupled states in a certain joint state equals to 1. In this paper, I am using ACL2 to verfy such property is still valid when lowering operators are operated on certain quantum states.

Normally, for two-particle systems, we can easily fiure out the total angular momenta $\vec{J}$ and the z-component of $\vec{J}$, $\vec{m_J}$. However, we could not know the angular momentum of each particle $\vec{l}$ and $\vec{s}$. Clebsch-Gordan coefficients can give the probabilties of the particles in each possible set of angular momenta. Therefore Clebsch-Gordan coefficients are very important in determining the angular momenta of two particle systems.

The handproofs of the correctness of calculating Clebsch-Gordan coefficients using lowering operators is lengthy and tedious. Therefore, if ACL2 theorem prover can prove the correctness, it can save a large amount of work. In addition, ACL2 theorem prover can verify the correctness of the handproofs.

First of all, this paper provides the background knowledge. It introduces the concept of quantum coupling, Clebsch Gordan coefficient, and lowering operators.

Then this paper will introduce ACL2, which is a programming language capable of modeling computer systems as well as proving properties of those systems. The quantum-state model described in this paper is implemented in ACL2 and the properties and main theorems are proved in ACL2.

After introducing my model, this paper will discuss how j-states, coupled states and quantuam states are implemented. Also, properties of those models will be discussed and procedures in proving those properties will be introduced.

After discussing my models, this paper will discuss how lowering operators operate on initial states and verify if the output satisfies Clebsch-Gordan coefficients and probability rules.

Finally, possible future research areas and possible improvements are discussed in this

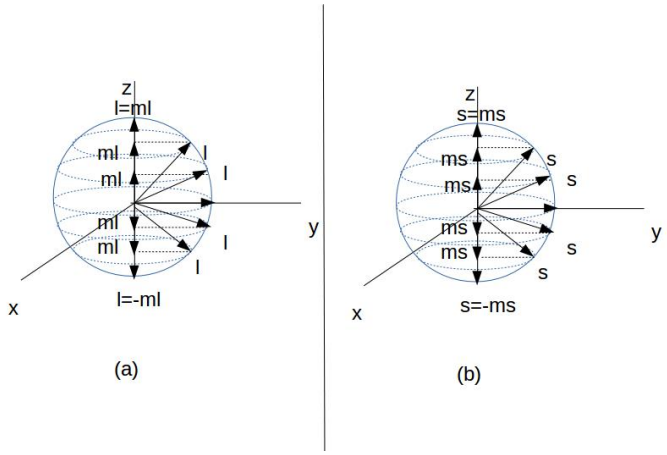

Fig. 1. The graphical representation of l, m_l, s, and m_s. The vectors which has tips on the circle are $\vec{l}$ and $\vec{s}$ recpectively. The vectors on z-axis are $\vec{m}_l$ and $\vec{m}_s$ respectively. Special cases are $|\vec{l}| = |\vec{m}_l|$ and $|\vec{s}| = |\vec{m}_s|$.

paper.

2 Background Knowledge

Some background knowledge is necessary in order to fully understand this paper.

2.1 Angular Momemtum

In this paper, we have two angular momenta. The first angular momentum is denoted as $\vec{l}$ and the second momemtum is denoted as $\vec{s}$. Their projections onto the z-axis are denoted as $\vec{m}_l$ and $\vec{m}_s$ respectively.

Therefore, l, m_l, s, and m_s are quantum numbers. Here l and s represent the two angular momenta, while m_l and m_s represent the z-compoment of l and s. In quantum physics, those quantum numbers satisfy the following conditions:

1) $l > 0$ and $s > 0$
2) l is either an integer or a half-integer (with denominator 2). Same for s
3) $m_l = -l, -l+1, \ldots, l$, $m_s = -s, -s+1, \ldots, s$
4) If l is an integer, then m_l is an integer. Otherwise m_l is a half-integer. Same for s and m_s

The graphical representation of $\vec{l}$, $\vec{s}$, $\vec{m}_l$ and $\vec{m}_s$ is shown in Figure 1.

In this paper, $\vec{J}$ is defined as $\vec{J} = \vec{l} + \vec{s}$. The projection of $\vec{J}$ onto z-axis, $\vec{m}_J$, is therefore

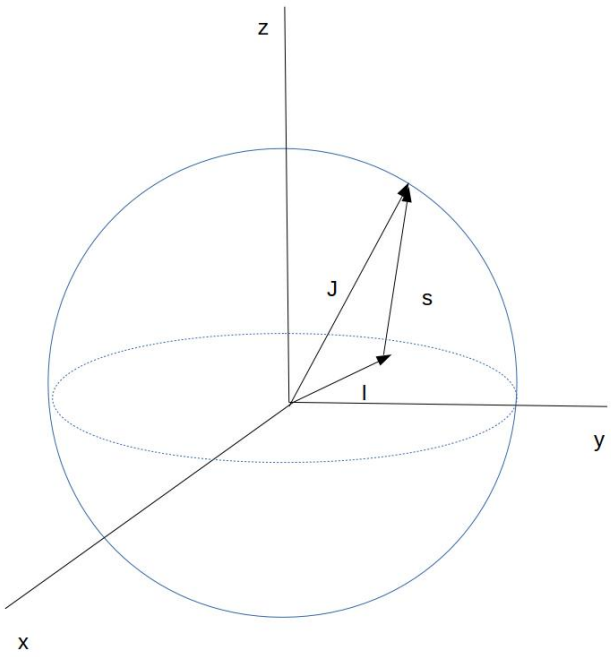

Fig. 2. The graphical representation of $\vec{J}$. Here $\vec{J}$ is the total angular momentum.

$\vec{m}_J = \vec{m}_l + \vec{m}_s$. The graphical representation of $\vec{J}$ is shown in Figure 2.

The quantum numbers J and m_J satisfy the following conditions:

1) $J \geq 0$
2) $J = |l-s|, |l-s|+1, \ldots, l+s$. However, in this project, we only consider $J = l+s$ for simplisity. In the future, we can expand the conditions and let J equals to other values
3) J is either an integer or a half-integer (with denominator 2)
4) $m_J = -J, -J+1, \ldots, J$
5) If J is an integer, then m_J is an integer. Otherwise m_J is a half-integer

Similar to Figure 1, $\vec{J}$ and $\vec{m}_J$ are represented in Figure 3.

2.2 Quantum States

In quantum physics, quantum states represent the states of quantum system. In this paper, the quantum states are represented in ket-notations. For example $|l, m_l\rangle$ means a particle is in a state such that the particle has quantum numbers l and m_l.

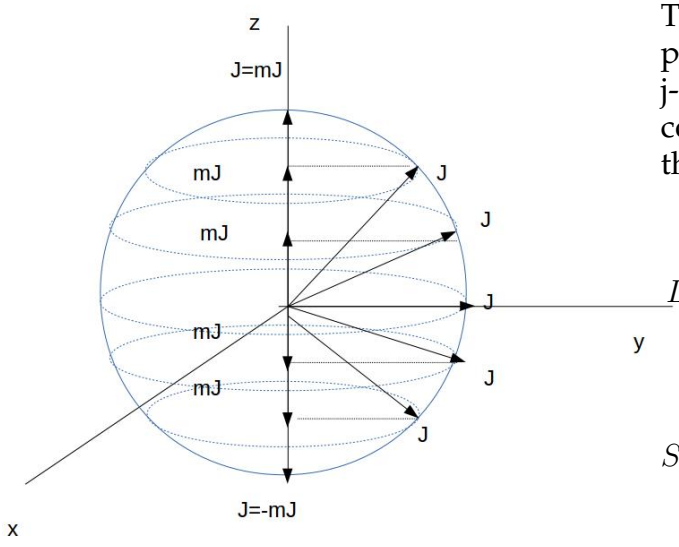

Fig. 3. The graphical representation of $\vec{J}$ and $\vec{m_J}$. $\vec{J}$ and $\vec{m_J}$ are very similar than those vectors represented in Figure 1.

In this paper, $|l, m_l\rangle|s, m_s\rangle$ is a coupled-state, while $|J, m_J\rangle$ is defined as J-state. A quantum state can be represented in the following equation:

$$|J, m_J\rangle = \sum_{m_J=m_l+m_s} c_n|l, m_l\rangle|s, m_s\rangle \quad (1)$$

In equation 1, $|J, m_J\rangle$ is the J-state, which is a state with the quantum numbers of total angular momentum only. In addition, we have $|l, m_l\rangle|s, m_s\rangle$, which is the coupled state with two sets of angular momenta $\vec{l}$ and $\vec{s}$ as quantum numbers. Here, c_n is called the Clebsch-Gordan coefficient. Here $|c_n|^2$ means given the J-state $|J, m_J\rangle$ of a particle, we can know that the particle is in coupled state $|l, m_l\rangle|s, m_s\rangle$ with probability $|c_n|^2$. Therefore, we ca know:

$$\sum_{m_j=m_l+m_s} |c_n|^2 = 1 \quad (2)$$

My models of quantum states are based on equation 1. The main theorem to be proved in ACL2 is in fact equation 2. More detail will be discussed in next section.

2.3 Lowering Operators

Three lowering operators are used in this project: J_-, L_- and S_-. J_- only operates on the j-state while L_- and S_- only operate on the coupled-state. The lowering operators satisfy the following equations:

$$L_-|l, m_l\rangle = \hbar\sqrt{l(l+1) - m_l(m_l-1)}|l, (m_l-1)\rangle \quad (3)$$

$$S_-|s, m_s\rangle = \hbar\sqrt{s(s+1) - m_s(m_s-1)}|s, (m_s-1)\rangle \quad (4)$$

Since $\vec{J} = \vec{l} + \vec{s}$, J_- obeys the following properties:

$$J_- = L_- + S_- \quad (5)$$

$$J_-|J, m_J\rangle = \hbar\sqrt{J(J+1) - m_J(m_J-1)}|J, (m_j-1)\rangle \quad (6)$$

The above equations indicate that the lowering operators do not affect J, l and s. However, after each operators operating on the quantum states, m_J, m_l and m_l are lowered by 1.

In equation 1, J_- operates on left-hand-side while L_- and S_- operate on the right-hand-side. A specific example is given below.

We have an electron with angular momentum $l = 1$ and spin $s = \frac{1}{2}$

Therefore $m_l = -1, 0, -1$ and $m_s = \pm\frac{1}{2}$

This means $J = \frac{3}{2}$ and $m_J = -\frac{3}{2}, -\frac{1}{2}, \frac{1}{2}, \frac{3}{2}$

We know that the only possibility of forming $|J, m_J\rangle = |\frac{3}{2}, \frac{3}{2}\rangle$ is $|\frac{3}{2}\ \frac{3}{2}\rangle = |1, 1\rangle|\frac{1}{2}, \frac{1}{2}\rangle$.

Applying the lowering operator on the above equation

$$J_-|\frac{3}{2}, \frac{3}{2}\rangle = \hbar\sqrt{\frac{3}{2}\left(\frac{3}{2}+1\right) - \frac{3}{2}\left(\frac{3}{2}-1\right)} \quad (7)$$
$$|\frac{3}{2}, \frac{1}{2}\rangle = \sqrt{3}\hbar|\frac{3}{2}, \frac{1}{2}\rangle$$

$$\begin{aligned} J_-|1,1\rangle|\frac{1}{2},\frac{1}{2}\rangle &= (L_- + S_-)|1,1\rangle|\frac{1}{2},\frac{1}{2}\rangle \\ &= L_-|1,1\rangle|\frac{1}{2},\frac{1}{2}\rangle + |1,1\rangle S_-|\frac{1}{2},\frac{1}{2}\rangle \\ &= \hbar\sqrt{1(1+1)-1(1-1)}|1,0\rangle|\frac{1}{2},\frac{1}{2}\rangle \\ &+\hbar\sqrt{\frac{1}{2}\left(\frac{1}{2}+1\right)-\frac{1}{2}\left(\frac{1}{2}-1\right)}|1,1\rangle|\frac{1}{2},-\frac{1}{2}\rangle \\ &= \hbar\sqrt{2}|1,0\rangle|\frac{1}{2},\frac{1}{2}\rangle + \hbar|1,1\rangle|\frac{1}{2},-\frac{1}{2}\rangle \end{aligned} \tag{8}$$

$$\sqrt{3}\hbar|\frac{3}{2},\frac{1}{2}\rangle = \hbar\sqrt{2}|1,0\rangle|\frac{1}{2},\frac{1}{2}\rangle + \hbar|1,1\rangle|\frac{1}{2},-\frac{1}{2}\rangle \tag{9}$$

Therefore, we have:

$$|\frac{3}{2},\frac{1}{2}\rangle = \sqrt{\frac{2}{3}}|1,0\rangle|\frac{1}{2},\frac{1}{2}\rangle + \frac{1}{\sqrt{3}}|1,1\rangle|\frac{1}{2},-\frac{1}{2}\rangle \tag{10}$$

Equation 10 satisfies eqaution 2. Therefore the lowering operators given in this expample gives valid Clebsch-Gordan Coefficients.

A general form of equation 9 can be expressed as follows, with $\hbar$ eliminated on both sides of the equation:

$$\begin{aligned} \sqrt{A}|J,m_J\rangle &= \sqrt{B}|l_1,m_{l1}\rangle|s_1,m_{s1}\rangle \\ &+\sqrt{C}|l_2,m_{l2}\rangle|s_2,m_{s2}\rangle + \dots \end{aligned} \tag{11}$$

There is one special case when $J = -m_J$, $l = -m_l$ and $s = -m_s$. According to equations 6,3 and 4, we can know that both sides of equation 11 are 0.

2.4 ACL2

ACL2, which stands for "A Computational Logic for Applicative Common Lisp", is a subset of Lisp. It is both a functional programming language based on Common Lisp and a first-order mathematical theory with induction [2].

In addition, ACL2 has a theorem prover, which can be used to prove the properties of the self-defined models. This project utilizes the power to ACL2 to prove that the lowering operators operating on some spefific quantum states gives the correct Clebsch-Gordan coefficients.

Similar to Common Lisp, a function is of the following form:

```
(defun function-name (x)
...)
```

In order to certify that the recrusive functions would terminate at some points, the function definitions should all satisfy the conditions mentioned in [2]. What valid function definitions mean is not the scope of this paper.

The theorems in ACL2 are of this form:

```
(defthm theorem-name
...)
```

When a theorem is complied, ACL2 theorem prover will try to prove the correctness of such theorem using induction. The lemmas and main theorem are proved in this way.

3 My Model

My model in this project is of this form:

```
1  ((A . (j . m_j)) . ;j-state
2  (B . ((l . m_l) . (s . m_s)))
3  (C . ((l . m_l) . (s . m_s)))
4  ...) ;coupled-state
```

As discussed in section 2.3, when the lowering operators operate on the lowest state. The model becomes:

```
1  (0 . 0)
2  ;This happens when the lowering operators
3  ;are operated on the lowest states
```

The model is indeed a pair. The car of the pair is the J-state and the cdr of the pair is a list of coupled-states. Other than the conditions described in previous sections, the model satisfies the following conditions:

1) $A = B + C + \dots$
2) The coefficients should be non-negative
3) In my model, J-state and coupled-states are either both 0 or a pair for a J-state and a list for coupled-states

3.1 J-State

3.1.1 Implementation

As discussed in the previous section, a J-state should be of the form:

```
(A . (j . m_j))
```

Before the actual implementation of J-state, we have several helper functions:

```
(defun j-coefficient (x)
 (car x))

(defun quantum-j (x)
 (car (cdr x)))

(defun quantum-mj (x)
 (cdr (cdr x)))

(defun lower-or-lowest-jstate (x)
  (or (>= (- (quantum-mj x)) (quantum-j x))
     (equal (j-coefficient x) 0)))

(defun half-or-full-integer (x)
  (xor (equal (denominator x) 1)
       (equal (denominator x) 2)))

(defun half-full-match (x y)
  (and (iff (equal (denominator x) 1)
            (equal (denominator y) 1))
       (iff (equal (denominator x) 2)
            (equal (denominator y) 2))))

(defun rational-pair (x)
  (if (atom x)
      nil
      (and (rationalp (car x))
           (< 0 (car x))
           (rationalp (cdr x))
           (natp (- (car x) (abs (cdr x))))
           (half-or-full-integer (car x))
           (half-or-full-integer (cdr x))
           (half-full-match (car x) (cdr x))
           (<= (abs (cdr x)) (car x)))))
```

Functions j-coefficient, quantum-j and quantum-mj returns the specific values of the coefficient or quantum number. The lower-or-lowest-jstate determines whether the j-state is in the lowest state or invalid state ($J < -m_J$). Function half-or-full-integer checks whether the J-state satisfies condition 3 for the J-state (at the end of section 2.1) while half-full-match checks whether condition 5 is satisfied. Finally, rational-pair determines whether condition 4 is satisfied.

Therefore, the function which checks whether the input is a real J-state is given below:

```
(defun true-jstate (a)
  (if (atom a)
      (equal a 0)
      (and (rationalp (car a))
           (<= 0 (car a))  ;Condition 2
           (rational-pair (cdr a)))))
```

The J-lowering operator is implemented as follows. It is implemented according to equation 6.

```
(defun j-lowering-operator (x)
  (if (atom x)
      0
    (if (lower-or-lowest-jstate x)
        0
       (cons (* (j-coefficient x)
                (+ (expt (quantum-j x) 2)
                  (quantum-j x)
                  (- (expt (quantum-mj x) 2))
                  (quantum-mj x)))
          (cons (quantum-j x)
             (- (quantum-mj x) 1))))))
```

3.1.2 Main Theorem of J-state

We have to verify that if the input of j-lowering-operator is a J-state, then the output should be a J-state. The theorem is shown below:

```
(defthm j-lowering-valid
  (implies (true-jstate x)
           (true-jstate
            (j-lowering-operator x))))
```

The complete implementation to successfully prove the above theroem is in jstate.lisp.

3.2 Coupled State

3.2.1 Implementation

Some helper functions are defined to ace the proof of theorems.

```
(defun first-coupled-state (x)
 (car x))

(defun first-coupled-state-pair (x)
 (cdr (car x)))

(defun first-coupled-coefficient (x)
 (car (first-coupled-state x)))

(defun first-coupled-l-state (x)
 (car (cdr
       (first-coupled-state x))))

(defun first-coupled-s-state (x)
 (cdr (cdr
       (first-coupled-state x))))

```

```
(defun first-coupled-l (x)
 (car (first-coupled-l-state x)))

(defun first-coupled-ml (x)
 (cdr (first-coupled-l-state x)))

(defun first-coupled-s (x)
 (car (first-coupled-s-state x)))

(defun first-coupled-ms (x)
 (cdr (first-coupled-s-state x)))
```

The above functions gives the specific states, coefficients or quantum numbers.

The functions below determine whether the input a real coupled state.

```
1  (defun true-coupled-list (x)
2   (if (atom x)
3    t
4    (and (rationalp (first-coupled-coefficient x
         ))
5     (<= 0 (first-coupled-coefficient x))
6     (rational-pair (first-coupled-l-state x))
7     (rational-pair (first-coupled-s-state x))
8     (true-coupled-list (cdr x)))))
9
10 (defun true-coupled-state (x)
11  (if (atom x)
12   (equal x 0)
13   (true-coupled-list x)))
```

If the input is an atom, then true-coupled-state verifies whether the input is 0. Otherwise true-coupled-list is called.

Function true-coupled-list has to verify whether the input satisfies all the conditions described in the middle of section 2.1 as well as condition 2 and condition 3 in section 3. Note that the function recursively checks all the elements in the list of states.

All the conditions in the middle of section 2.1 are checked in line 6 and 7 (two rational-pair functions). Condition 2 and condition 3 in section 3 are checked in lines 4 and 5.

3.2.2 Clean up the Zeros in the Output List

After each lowering operations, some or all of the elements might be 0 in the output coupled list. Therefore the following functions clean up the zeros in the list.

```
1  (defun all-zeros (x)
2   (if (atom x)
3    t
4    (and (equal (car x) 0)
5     (all-zeros (cdr x)))))
6
7  (defun clean-up-zero-list (x)
8   (if (atom x)
9    x
10   (if (equal (car x) 0)
11    (clean-up-zero-list (cdr x))
12    (cons (car x)
13     (clean-up-zero-list (cdr x))))))
14
15 (defun clean-up-zero (x)
16  (if (atom x)
17   0
18   (if (all-zeros x)
19    0
20    (clean-up-zero-list x))))
```

Initially, clean-up-zero is called. If the input is an atom, then it simply returns 0. If all the elements in the list are 0, then it also returns 0. Otherwise function clean-up-zero-list is called.

Function clean-up-zero-list recursively removes all the zeros in te list. Since function clean-up-zero is called first, there is at least one non-zero coupled element in the list.

Undoubtedly, the following theorem should be true:

```
1  (defthm clean-up-zero-valid
2   (implies (true-coupled-state x)
3    (true-coupled-state (clean-up-zero x))))
```

3.2.3 L Lowering Operator

The implementation of L_-, which is based on equation 3, is shown below:

```
1  (defun l-lowering-operator-helper (x)
2   (if (atom x)
3    nil
4    (cons (l-lowering-to-state x)
5     (l-lowering-operator-helper (cdr x)))))
6
7  (defun l-lowering-operator (x)
8   (if (atom x)
9    0
10   (clean-up-zero (l-lowering-operator-helper x
       ))))
```

Initially, l-lowering-operator is called. If the input is an atom, then 0 is returned. Otherwise l-lowering-operator-helper is called.

Function l-lowering-operator-helper recursively applies l lowering onto each coupled element. Note that function l-lowering-to-state is indeed an implementation of equation 3. The

implementation of equation 3 is in coupled-state.lisp. The output of l-lowering-operator-helper is the input of clean-up-zero. Therefore we can make sure that if the output is a list, there should be no zeros in it.

We have to prove that if the input is a list of real coupled states or 0 depending on whether the states are valid, then the output of l-lowering-operator should also be a list of real coupled states or 0.

```
(defthm l-lowering-valid
 (implies (true-coupled-state x)
  (true-coupled-state (l-lowering-operator x))
      ))
```

3.2.4 S Lowering Operator

The implementation of S_- is very similar to that of L_-. The major difference is that the implementation of S_- is baed on equation 4.

```
(defun s-lowering-operator-helper (x)
 (if (atom x)
  nil
  (cons (s-lowering-to-state x)
   (s-lowering-operator-helper (cdr x)))))

(defun s-lowering-operator (x)
 (if (atom x)
  0
  (clean-up-zero (s-lowering-operator-helper x
      ))))
```

Initially, s-lowering-operator is called. If the input is an atom, then 0 is returned. Otherwise s-lowering-operator-helper is called.

Function s-lowering-operator-helper recursively applies s lowering onto each coupled element. Note that function s-lowering-to-state is indeed an implementation of equation 4. The implementation of equation 4 is in coupled-state.lisp. The output of s-lowering-operator-helper is the input of clean-up-zero. Therefore we can make sure that if the output is a list, there should be no zeros in it.

We have to prove that if the input is a list of real coupled states or 0 depending on whether the states are valid, then the output of s-lowering-operator should also be a list of real coupled states or 0.

```
(defthm s-lowering-valid
 (implies (true-coupled-state x)
  (true-coupled-state (s-lowering-operator x))
      ))
```

3.2.5 Merge and Append Coupled State Elements

There are possibilities that the outputs of l-lowering-operator and s-lowering-operator contain elements with the same quantum numbers l, m_l, s and m_s. Here is a specific example.

After l-lowering, we have:

```
((A . ((l . m_l) . (s . m_s)))
(B . ((l' . m_l') . (s' . m_s'))))
```

After s-lowering, we have:

```
((C . ((l' . m_l') . (s' . m_s')))
(D . ((l'' . m_l'') . (s'' . m_s''))))
```

The elements with coefficients B and C have the same set of quantum numbers. Therefore we need a function whose output will be of this form in this example:

```
((A . ((l . m_l) . (s . m_s)))
(E . ((l' . m_l') . (s' . m_s')))
(D . ((l'' . m_l'') . (s'' . m_s''))))
```

Here E is totally a new coefficient. In general, it can be calculated using the following equation [3]:

$$E = [(l+s)^2 - (m_l+m_s)^2 + l + s - m_l - m_s] * \frac{(2l)!(2s)!(l+s+m_l+m_s)!(l+s-m_l-m_s)!}{(2*(l+s))!(l+m_l)!(l-m_l)!(s+m_s)!(s-m_s)!} \quad (12)$$

First, we have to implement functions which takes a coupled element and a coupled list as inputs. In the list, the functions should delete all the elements which have the same quantum numbers as the coupled element.

```
(defun delete-same-helper (b y)
 (if (atom y)
  nil
  (if (equal b (first-coupled-state-pair y))
   (delete-same-helper b (cdr y))
   (cons (car y) (delete-same-helper b (cdr y)
       )))))

(defun delete-same (b y)
```

```
 9   (if (atom y)
10    0
11    (if (all-same b y)
12     0
13     (delete-same-helper b y))))
```

Initially, delete-same is called. If the input list is an atom or all of its elements have the same quantum numbers as the input coupled element, then it simply returns 0. Otherwise the function calls delete-same, which recursively delete the elements in the list which has the same quantum numbers as the input coupled element.

Then function merge-same is implemented. It merges the elments with the same quantum numbers to a single coupled element according to equation 12.

```
 1  (defun merge-same (a y)
 2   (if (atom y)
 3    a
 4    (if (equal (cdr a) (first-coupled-state-pair
          y))
 5     (cons (calculate-merge-coefficient
 6            (l-number a)
 7            (ml-number a)
 8            (s-number a)
 9            (ms-number a))
10      (cdr a))
11     (merge-same a (cdr y)))))
```

Finally, function append-and-merge-states is implemented to merge and append two coupled lists:

```
 1  (defun append-and-merge-states-helper (x y)
 2   (if (atom x)
 3    y
 4    (if (atom y)
 5     nil
 6     (cons (merge-same (first-coupled-state x) y
           )
 7      (append-and-merge-states-helper (cdr x)
 8       (delete-same
 9          (first-coupled-state-pair x) y))))))
10
11  (defun append-and-merge-states (x y)
12   (if (atom x)
13    (if (atom y)
14     0
15     y)
16    (if (atom y)
17     x
18     (append-and-merge-states-helper x y))))
```

As usual, we should verify that if the two inputs of function append-and-merge-states are both coupled lists, the output of this function should also be a coupled list.

```
1  (defthm append-valid
2   (implies (and (true-coupled-state x)
3                 (true-coupled-state y))
4    (true-coupled-state
5          (append-and-merge-states x y))))
```

Initially, the proof of this theorem took around 15 minutes and more than 80 million steps. The proof was undoubted unefficient and not optimal. The main reason is due to a large number of unecessary splits. Therefore, a coupled of lemmas are implemented. In addition, some uncessary functions and theorems are disabled. As a result, the running time is reduced to around 4 seconds and the number of steps needed is approximately 70 thousand. This is indeed a significant improvement. Detailed implementations coupled be found in coupled-state.lisp.

Throughout this project, a large numbers of lemmas are proved in order to reduce the case splits and prevent the throrem prover failing. Those lemmas will not be discussed in detail in this paper. However, all the implementations could be found in the code itself.

3.3 Quantum State

If quantum states is a pair whose car is a J-state and whose cdr is a list of coupled-state. If a quantum state is invalid, the quantum state is $(0\ .\ 0)$. In fact, besides the conditions listed in previous sections, a quantum state satisfies the following conditions:

1) The sum of coefficients of the coupled states equals to the coefficient of the J-state
2) For all coupled states, $J = l + s$
3) For all coupled states, $m_J = m_l + m_s$
4) If a quantum state is a valid state, it is a pair whose car is a J-state and whose cdr is a list of coupled-state
5) If a quantum state is not a state with valid quantum numbers, then it has the form $(0\ .\ 0)$

Before the actual implementation of quantum state, there are several helper functions:

```
1  (defun get-jstate (x)
2   (car x))
```

```
3
4  (defun get-coupled-state (x)
5   (cdr x))
6
7  (defun sum-of-coupled-coefficient (a)
8   (if (atom a)
9    0
10   (+ (first-coupled-coefficient a)
11    (sum-of-coupled-coefficient (cdr a)))))
```

Function get-jstate returns the J-state and get-coupled-state returns the list of coupled states. Function sum-of-coupled-coefficient takes a list of coupled states as input and return the sum of coefficients of the coupled states.

In addition, we have the following helper functions:

```
1  (defun equal-j (x y)
2   (if (atom y)
3    t
4    (and (equal (quantum-j x)
5          (+ (first-coupled-l y)
6           (first-coupled-s y)))
7     (equal-j x (cdr y)))))
8
9  (defun equal-mj (x y)
10  (if (atom y)
11   t
12   (and (equal (quantum-mj x)
13         (+ (first-coupled-ml y)
14          (first-coupled-ms y)))
15    (equal-mj x (cdr y)))))
```

Function equal-j takes a J-state and a list of coupled states as inputs and resursively verify if all the coupled states satisfy condition (2) above. Function equal-mj takes a J-state and a list of coupled states as inputs and recursively verify if all the coupled states satisfy condition (3) above.

Finally, function true-quantum-state verifies whether the input is a real quantum state.

```
1  (defun true-quantum-state (x)
2   (xor (and (equal (get-jstate x) 0)
3         (equal (get-coupled-state x) 0))
4    (and (true-jstate (get-jstate x))
5     (true-coupled-state (get-coupled-state x))
6     (equal (sum-of-coupled-coefficient
7             (get-coupled-state x))
8      (caar x))
9     (equal-j (get-jstate x)
10     (get-coupled-state x))
11    (equal-mj (get-jstate x)
12     (get-coupled-state x)))))
```

Before applying the lowering operators, the quantum states have to be normalized.

3.3.1 Normalize

The normalization devide each coefficient in J-state and in the coupled states by the coefficient of J-state.

Before the normalization, the quantum states has the following forms:

```
1  ((A . (j . m_j)) . ;j-state
2  (B . ((l . m_l) . (s . m_s))) ;coupled-state
3  (C . ((l . m_l) . (s . m_s)))
4  ...)
```

or

```
1  (0 . 0)
2  ;This happens when the lowering operators
3  ;are operated on the lowest states
```

After the normalization, the quantum states have the following forms:

```
1  ((1 . (j . m_j)) . ;j-state
2  (B/A . ((l . m_l) . (s . m_s))) ;coupled-state
3  (C/A . ((l . m_l) . (s . m_s)))
4  ...)
```

or

```
1  (0 . 0) ;When the input is (0 . 0)
```

The normalization is implemented by function normalize-state.

```
1  (defun normalize-helper (a y)
2   (if (atom y)
3    y
4    (cons (cons (/ (first-coupled-coefficient y)
          a)
5           (first-coupled-state-pair y))
6     (normalize-helper a (cdr y)))))
7
8  (defun normalize-state (x)
9   (if (and (equal (get-jstate x) 0)
10        (equal (get-coupled-state x) 0))
11   x
12   (if (< 0 (car (get-jstate x)))
13    (cons (cons '1 (cdr (get-jstate x)))
14     (normalize-helper (car (get-jstate x))
15      (get-coupled-state x)))
16    (cons '0 '0))))
```

The funtion normalize-state checks whether the input is $(0\ .\ 0)$. If so, it simply returns $(0\ .\ 0)$. Otherwise, if the coefficient of J-state is 0, then the function also returns $(0\ .\ 0)$.

Otherwise, normalize-helper is called and the actual normalization is operated recursively for each coupled elements.

We can easily verify manually that if the input of normalize-state is a quantum state, the output of this function should also be a quantum state. Hence, we also have to verify this theorem in ACL2 theorem prover:

```
(defthm normalize-valid
 (implies (true-quantum-state x)
  (true-quantum-state (normalize-state x))))
```

The verification of this theorem requires approximately 30 lemmas. This paper will not discuss them in details. However, those lemmas could be found in quantum-state.lisp.

3.3.2 Lowering Operators

Function quantum-operator is the function which applies the J-lowering operator on the J-state and L-lowering and S-Lowering operators on the coupled states. The implementation is shown as follows:

```
(defun quantum-operator-helper (x)
 (cons (j-lowering-operator (get-jstate x))
  (append-and-merge-states
   (l-lowering-operator (get-coupled-state x))
   (s-lowering-operator (get-coupled-state x))
       )))

(defun quantum-operator (x)
 (if (and (equal (get-jstate x) 0)
          (equal (get-coupled-state x) 0))
  x
  (quantum-operator-helper (normalize-state x)
      )))
```

Initially, function quantum-operator is called. If the input is (0 . 0), then the function simply returns (0 . 0). Otherwise, quantum-operator-helper is called.

Function quantum-operator-helper takes a quantum state with valid quantum numers as the input. It applies the J-lowering operator on the J-state and L-lowering and S-lowering operators on the coupled states. The two outputs of L-lowering operator and S-lowering operator are the inputs of append-and-merge-states, which merge and append the two lists of coupled states.

If we can verify that the output of function quantum-operator is a quantum state given that the input is a quantum state, then this project is done. However, determining the input is not that simple. Details are discussed in the next section.

4 Initial State

The Clebsch-Gordan coefficient table is shown in Figure 4. According to the table, the coefficients are determined for every set of quantum numbers. The only quantum states whose coefficients can be determined easily are the ones with $J = m_J$, $l = m_l$ and $s = m_s$. Such quantum states are called initial states.

Besides the conditions of quantum states, initial states also satisfy the following conditions:

1) The initial states have this form: ((1 . (j . m_j)) . ((1 . ((l . m_l) . (s . m_s)))))
2) $m_l = l$, $m_s = s$, $m_j = j$
3) There is only one coupled element in the coupled list

Function initial-quantum-state verifies whether the input satisfies the conditions above. It is implemented as follows:

```
(defun initial-quantum-state (x)
 (and (equal (j-coefficient (get-jstate x)) 1)
  (equal (first-coupled-coefficient
          (get-coupled-state x))
   1)
  (equal (len (get-coupled-state x)) 1)
  (equal (quantum-j (get-jstate x))
   (quantum-mj (get-jstate x)))
  (equal (first-coupled-l (get-coupled-state x
      ))
   (first-coupled-ml (get-coupled-state x)))
  (equal (first-coupled-s (get-coupled-state x
      ))
   (first-coupled-ms (get-coupled-state x)))
  (true-quantum-state x)))
```

Serveral properties of initial states are proved in ACL2 theorem prover.

4.1 An Initial State is a Quantum State

The implementation of this theorem is shown as follows:

```
(defthm initial-quantum-state-valid
 (implies (initial-quantum-state x)
  (true-quantum-state x)))
```

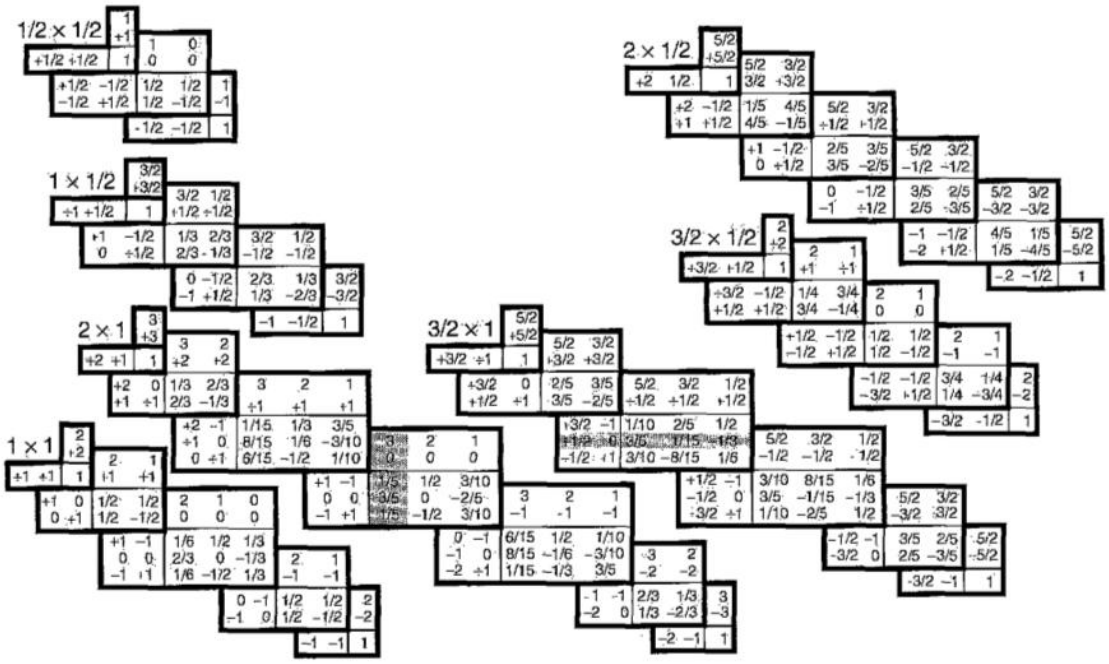

Fig. 4. The Clebsch-Gordan coefficient table. [4] (A squareroot sign is understood for every entry (see equation 11); The minus sign, if present, goes outside the radical.) In this paper, we only condider the case $J = l + s$. In this case all the coefficients are with positive sign.

4.2 Normalize Initial States

Since the coefficient of J-state and the coefficient of the only coupled states are 1, the normalization of initial state is indeed the initial state itself. The implementation of the theorem is shown as follows:

```
(defthm initial-state-lemma-5
 (implies (initial-quantum-state x)
  (equal (normalize-helper 1
                           (get-coupled-state x))
   (get-coupled-state x))))
```

4.3 Cleaning up Zeros in Initial States

Since an initial state does not have elements equal to 0, function clean-up-zero does not change the output of the lowering operators applying on the initial states. The theorems which describe this property is implemented as follows:

```
(defthm initial-state-lemma-6
 (implies (and (none-zero x)
               (consp x)
               (true-coupled-list x))
  (equal (clean-up-zero x) x)))

(defthm initial-state-lemma-10
 (implies (initial-quantum-state x)
  (none-zero
   (l-lowering-operator-helper (cdr x)))))

(defthm initial-state-lemma-15
 (implies (initial-quantum-state x)
  (none-zero
   (s-lowering-operator-helper (cdr x)))))
```

Lemma 10 and lemma 15 verifythat the L-lowering operators and S-lowering operators do not give lists with zero-elements. Then those two lemmas and lemma 6 together proves that operators operating on the initial states will not have 0-elements. As the following theorems shown:

```
(defthm initial-state-lemma-19
 (implies (initial-quantum-state x)
  (equal (clean-up-zero-list
          (l-lowering-operator-helper
           (get-coupled-state x)))
   (l-lowering-operator-helper
    (get-coupled-state x)))))

(defthm initial-state-lemma-23
 (implies (initial-quantum-state x)
  (equal (clean-up-zero-list
          (s-lowering-operator-helper
```

```
13              (get-coupled-state x)))
14     (s-lowering-operator-helper
15      (get-coupled-state x))))
```

4.4 L-lowering Operator and S-lowering Operator on Initial States

This property is very straightforward. If the input is an initial state, then the outputs of l-lowering-operator and s-lowering-operator are real coupled states. The theorems are implemented below.

```
1  (DEFTHM
2   INITIAL-STATE-LEMMA-29
3   (IMPLIES (INITIAL-QUANTUM-STATE X)
4    (TRUE-COUPLED-LIST
5       (L-LOWERING-OPERATOR-HELPER (CDR X)))))
6
7  (DEFTHM
8   INITIAL-STATE-LEMMA-30
9   (IMPLIES (INITIAL-QUANTUM-STATE X)
10   (TRUE-COUPLED-LIST
11      (S-LOWERING-OPERATOR-HELPER (CDR X)))))
```

Note that the two theorems above are proved by the ACL2 proof checker. The advantage of proof in this case is that it can avoid unnecessary case splits and incoreect induction schemes. The detailed implementation is shown in initial-quantum-state.lisp.

5 Main Theorem: Lowering Operators Operating on Initial States

5.1 Proof by Hand

The proof by hand is very straightforward. According to equation 11, we have

$$|J, m_J\rangle = |l, m_l\rangle|s, m_s\rangle \tag{13}$$

Then we apply the J-lowering operator to the left-hand side of equation 13, and L-lowering as well as S-lowering operator on the right-hand side. According to equations 6, 3 and 4, we have

$$\begin{aligned}\sqrt{2J}|J, m_J - 1\rangle = \sqrt{2l}|l, m_l - 1\rangle|s, m_s\rangle \\ +\sqrt{2s}|l, m_l\rangle|s, m_s - 1\rangle\end{aligned} \tag{14}$$

Since $J = l + s$, we can know $2J = 2l + 2s$. The Clebsch-Gordan coefficients calculated using lowering operators in this case is correct.

5.2 Proof by ACL2

The ACL2 theorem is listed below:

```
1  (DEFTHM INITIAL-STATE-LOWERING-VALID
2   (IMPLIES (INITIAL-QUANTUM-STATE X)
3    (TRUE-QUANTUM-STATE
4          (QUANTUM-OPERATOR X))))
```

The implementation of this theorem is in main-theorem-1.lisp. Initially, theorem prover is used to expand and simplify the functions. Then a prove command is used in order to utilize ACL2 power to prove this main theorem. This theorem is successfully proved and hence the application of lowering operators operaing on initial states gives correct Clebsch-Gordan coefficients.

6 Future Research

In fact, given an initial state, any number of times the application of the lowering operators on the initial state until the state with invalid quantum numbers (0 . 0) returns a state with correct Clebsch-Gordan coefficients (equation 2). The idea is shown as follows:

```
1  (skip-proofs
2   (defun all-lowering-valid (x)
3    (if (equal x (cons '0 '0))
4     t
5     (and (true-quantum-state
6           (quantum-operator x))
7      (all-lowering-valid
8       (quantum-operator x)))))
9
10 (defthm all-quantum-lowering-valid
11         (implies (inital-quantum-state x)
12                  (all-lowering-valid x)))
```

Here function all-lowering-valid detemines whether any number of lowering operations on the input until (0 . 0) gives a correct quantum state. Theorem all-quantum-lowering-valid is served as proving the property described above.

However, function all-lowering-valid is not a valid function definition according to [2]. Therefore a valid definition is essential. In addition, proving this theorem will need a large number of lemmas due to the fact that equation 12 is complicated. Those difficulties are need to overcome in the future.

We can also eliminate the restriction that $J = l + s$. Consequently, $J = |l - s|, |l - s| + 1, \ldots, l+s$. There are more cases to consider and the equation of calculating merged coefficient (similar to equation 12) will be signifincanly more complicated. Hence, the genralization of lowering operators applying to all quantum states is the ultimate goal.

7 Conclusion

This paper provides a detailed description on how to use ACL2 theorem prover to prove the correctness of lowering operators operating on the initial state when $J = l + s$. My model of quantum state consists of a J-state and a list of coupled states. The J-lowering operator operates on J-state while L-lowering and S-lowering operators operate on coupled states. The input and output of lowering operators are of the form shown in equation 11.

The main theorem of this project is given an initial state, the first application of lowering operators return a quantum state with correct Clebsch-Gordan coefficients. Proving this theorem manually is not complicated. However, the proof of this theorem is a fundamental step on the future research shown in Section 6.

The ultimate goal for this project is to prove the correctness of lowering operators applying to all quantum states. Due to the fact that quantum states have a large number of cases, the number of case splits will be large. Thus special care is necessary in the future research.

Acknowledgments

The author would like to thank Professor Warren Hunt, Teaching Assistant Nathan Wetzler, Dr. Matthew Kaufmann and Cuong Chau for helping me implementing this project. Their help is very important. I learned significant amount of knowledge from them and I believe such knowledge will be very useful in the future.

References

[1] E. W. Weisstein, "Clebsch-gordan coefficient," 2003.
[2] J. S. Moore, "Recursion and induction," 2008.
[3] M. Tukerman, "The general problem," 2005.
[4] D. Griffiths, *Introduction to Quantum Mechanics*, ser. Pearson international edition. Pearson Prentice Hall, 2005. [Online]. Available: http://books.google.com/books?id=z4fwAAAAMAAJ