A Haskell implementation of the Young Diagram based algorithm for calculating multiplet structures and particle multiplicities when combining SU(n) multiplets

Michael Dressel

Abstract

This article presents a Haskell implementation of the algorithm described by C. G. Wohl, 2021, in [1] section 48 "SU(n) Multiplets and Young Diagrams". Similar to the $\otimes$ notation, in the implementation described here, the operators $><$ and $>><$ are introduced to produce the resulting multiplet structure when combining 2 SU(n) multiplets and combining a list of multiplets with another multiplet, respectively. E.g. $[1,0] >< [1,0] = [[2,0],[0,1]]$ and $[1,0] >< [1,0] >>< [1,0] = [[3,0],[1,1],[1,1],[0,0]]$. The functions multi and multis are defined to determine the number of particles (multiplicity) in a multiplet and the multiplicities corresponding to a list of multiplets, respectively. E.g. multi $[1,0] = 3$ and multis \$ $[1,0] >< [1,0] >>< [1,0] = [10,8,8,1]$.

Keywords: Young Diagram, SU(n), multiplets, Haskell, functional programming, particle physics

1. Introduction

Functional programming is found not only in dedicated languages like Haskell [2] or Erlang [3] but also in popular environments like Scala [4] using the Java Virtual Machine (JVM) or lambda expressions directly within Java [5].

As a demonstration of how functional programming can be useful within applications of group theory e.g. in the field of particle physics, this article provides an implementation of the algorithm described in [1] section 48 using the functional programming language Haskell.

The source code is available at [6].

The algorithm from [1] uses the method of Young Diagrams to obtain the multiplet structure created by combining two multiplets of group SU(n) and provides formulas to calculate the number of particles or multiplicity in SU(n) multiplets.

While in [1] the multiplet labeling, combination and structure decomposition is denoted by $(\alpha, \beta, ...), \otimes, \oplus$, respectively, it seamed natural to use Haskell's list notation for the labeling of multiplets and to define the operators $><$ and $>><$ to perform the combination of multiplets. The resulting structural decomposition is given as a Haskell list.

Email address: michael.dressel@kloenplatz.de (Michael Dressel)

In the implementation # symbols are used instead of boxes usualy used to draw the Young Diagrams. E.g. the Young Diagrams labeled $[1,0]$, $[0,1]$, $[1,1]$, $[3,0]$ are displayed in Listing 1:

Listing 1: Example Young Diagrams

```
# #    # #    # # #    # # # #
#      # #    # #      #
#      #      #        #
```

The use of the operators >< and >>< are best shown by some examples:

Combining two spin-$\frac{1}{2}$ particles (i.e. the irreducible representations) have the structure:

$$[1] >< [1] = [[2], [0]]$$

and the multiplicities using
multi [1] = 2, multis [[2],[0]] = [3,1] being equivalent to:

$$2 \otimes 2 = 3 \oplus 1$$

Combining three SU(3) particles (i.e. irreducible representations) have the structure

$$[1,0] >< [1,0] >>< [1,0] = [[3,0], [1,1], [1,1], [0,0]]$$

with multiplicities using
multi [1,0] = 3, multis [[3,0],[1,1],[1,1],[0,0]] = [10,8,8,1] being equivalent to:

$$3 \otimes 3 \otimes 3 = 10 \oplus 8 \oplus 8 \oplus 1$$

2. Some details about the implementation

The rules given in [1] for determining admissible sequences of letters are interpreted here such that sequences are considered as being composed of strictly alphabetically ordered chains interlaced with one another. Accordingly a function unchain is defined to extract the longest remaining ordered chain from the sequence. This function in turn is used within the recursive function admis to extract all ordered chains starting with the letter 'a'. The sequence is considered admissible if the whole sequence can be extracted without rest, leaving an empty list.

In order to find all combinations of tableaux when combining one tableau with r rows with one line of length n of another tableau, the r^n positions are determined. From the list of positions a list of new tableaux are produced. The function tabs1 makes use of these procedures to produce a list of tableaux given one tableau and one line of another tableau.

Following [1] lettered diagrams are used for the combination of two tableaux. The conversion to the lettered diagram is done using the function sym2letter.

The function allTsFromSyms determines all possible tableaux initially given two tableaux in symbol and lettered format using tabs1. The function internally uses as arguments a list of tableaux and a single tableau to be combined with the list to produce a list of new tableaux. The internal function is double recursive in order to follow both, the combination of each tableau of the list with each line of the lettered tableau.

Several possible tableaux are rejected in case they do not fulfill the criteria for allowed rows and columns. From the letters within the tableaux the sequence of letters is created using the function readTab.

The function allTs produces a list of tableaux from combining two tableaux identified by their labeling.

The operator >< uses the functions allTs, admis and removes duplicate tableaux to produce the resulting list of tableau labels.

For convenience the function >>< is defined to allow combing a list of tableaux with another tableau using Haskell's list comprehension and concatenation.

The algorithm for calculating a tableau's multiplicity and multiplicities given a list of tableaux is implemented in the functions multi and multis respectively.

3. Conclusion

Functional programming, in particular Haskell, allowed a rather straightforward way to implement the algorithm outlined in [1] section 48.

References

[1] P. Zyla, et al., Review of Particle Physics, PTEP 2020 (8) (2020) 083C01, and 2021 update, article direct link: `https://pdg.lbl.gov/2022/reviews/rpp2022-rev-young-diagrams.pdf`. `doi:10.1093/ptep/ptaa104`.

[2] haskell.org, Haskell language, `https://www.haskell.org`, accessed 2022-05-30 (2022).

[3] erlang.org, Erlang, `https://www.erlang.org`, accessed 2022-05-30 (2022).

[4] Ecole Polytechnique F?d?rale Lausanne (EPFL) Lausanne, Switzerland, The scala programming language, `https://www.scala-lang.org`, accessed 2022-05-30 (2022).

[5] Oracle, Java, `https://www.java.com`, accessed 2022-05-30 (2022).

[6] M. Dressel, Multiplet Combining, `https://github.com/mdrslmr/MultipletCombiner.git`, accessed 2023-05-15 (2022).

[7] The University of Glasgow, The glasgow haskell compiler, `https://www.haskell.org/ghc`, accessed 2022-05-30 (2022).

Appendix A. Usage example

The file MultipletCombiner.hs may be directly loaded into a ghci [7] session. Listing 2 shows an example session.

Listing 2: An example ghci session

```
ghci> :l MultipletCombiner.hs
ghci> [1] >< [1]
[[2],[0]]
ghci> multi [1]
2
ghci> multis [[2],[0]]
[3,1]
ghci> [1,0] >< [1,0] >>< [1,0]
[[3,0],[1,1],[1,1],[0,0]]
ghci> multi [1,0]
3
ghci> multis $ [1,0] >< [1,0] >>< [1,0]
[10,8,8,1]
```