

Computing the Littlewood-Richardson coefficients

Jean-Christophe Filliâtre and Florent Hivert

LRI / Université Paris Sud / CNRS

IFIP WG 1.9/2.15, July 2015

- ① what are Littlewood-Richardson coefficients
- ② how do we compute them: the Littlewood-Richardson rule
- ③ a Coq proof of the rule (previous work by Florent Hivert)
- ④ an efficient program and its proof

Definition (Symmetric polynomial)

A polynomial is *symmetric* if it is invariant under any permutation of the variables: for all $\sigma \in \mathfrak{S}_n$,

$$P(x_0, x_1, \dots, x_{n-1}) = P(x_{\sigma(0)}, x_{\sigma(1)}, \dots, x_{\sigma(n-1)})$$

$$P(a, b, c) = a^2b + a^2c + b^2c + ab^2 + ac^2 + bc^2$$

$$Q(a, b, c) = 5abc + 3a^2bc + 3ab^2c + 3abc^2$$

integer partitions

different ways of decomposing an integer $n \in \mathbb{N}$ as a sum:

$$5 = 5 = 4+1 = 3+2 = 3+1+1 = 2+2+1 = 2+1+1+1 = 1+1+1+1+1$$

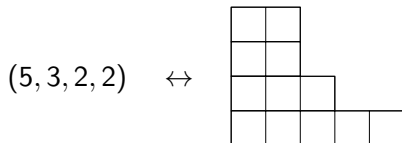
Definition

A partition of n is a non-increasing sequence

$\lambda := (\lambda_0 \geq \lambda_1 \geq \dots \geq \lambda_{l-1} > 0)$ such that

$n = \lambda_0 + \lambda_1 + \dots + \lambda_{l-1}$. We pose $\ell(\lambda) := l$.

Young diagram of a partition:



Definition (semistandard Young tableau of shape λ)

A Young diagram of shape λ filled with integers

- non decreasing along the rows (left to right)
- strictly increasing along the columns (bottom up)

example:

3				
2	3			
1	2	2		
0	0	1	1	2

Schur symmetric polynomials

Definition (Schur symmetric polynomial)

Given a partition $\lambda := (\lambda_0 \geq \lambda_1 \geq \dots \geq \lambda_{l-1})$ with $l \leq n$
(set $\lambda_i := 0$ for $i \geq l$)

$$s_\lambda(x_0, \dots, x_{n-1}) = \sum_T x_0^{t_0} \cdots x_{n-1}^{t_{n-1}}$$

where the sum is over all semistandard Young tableaux of shape λ
and t_i is the number of occurrences of i in T .

example

$$n = 3, \lambda = (2, 1)$$

1	
0	0

a^2b

2	
0	0

a^2c

1	
0	1

ab^2

2	
0	1

$2abc$

1	
0	2

2	
1	1

b^2c

2	
0	2

ac^2

2	
1	2

bc^2

$$s_{(2,1)}(a, b, c) = a^2b + ab^2 + a^2c + 2abc + b^2c + ac^2 + bc^2$$

Littlewood-Richardson coefficients

Proposition

The family $(s_\lambda(\mathbb{X}_n))_{\ell(\lambda) \leq n}$ is a (linear) basis of the ring of symmetric polynomials on \mathbb{X}_n .

Definition (Littlewood-Richardson coefficients)

Coefficients $c_{\lambda,\mu}^\nu$ of the expansion of the product:

$$s_\lambda s_\mu = \sum_{\nu} c_{\lambda,\mu}^\nu s_\nu .$$

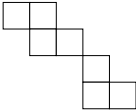
Fact: $c_{\lambda,\mu}^\nu$ are independent of the number of variables.

how to compute the Littlewood-Richardson coefficients

Definition (Skew shape)

A skew shape ν/λ is a pair of partitions such that the Young diagram of ν contains the Young diagram of λ .

example:

$$(5, 4, 3, 2)/(3, 3, 1) =$$


Definition (Skew semistandard Young tableau)

A skew shape ν/λ filled with integers

- non decreasing along the rows (left to right)
- strictly increasing along the columns (bottom up)

example:

0	2			
	0	1		
			1	
			0	0

Definition (row reading of a tableau)

the word obtained by concatenating the rows of a skew tableau, from top to bottom

example:

$$\begin{array}{|c|c|} \hline 0 & 2 \\ \hline \end{array} \begin{array}{|c|c|} \hline 0 & 1 \\ \hline \end{array} \begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|c|} \hline 0 & 0 \\ \hline \end{array} = 0201100$$

Yamanouchi words

notation: $|w|_x$ = number of occurrence of x in w .

Definition (Yamanouchi word)

A word w_0, \dots, w_{l-1} of integers such that for all k, i ,

$$|w_i, \dots, w_{l-1}|_k \geq |w_i, \dots, w_{l-1}|_{k+1}$$

Consequence: the evaluation $(|w|_i)_{i \leq \max(w)}$ is a partition.

$\epsilon, 0, 00, 10, 000, 100, 010, 210,$

$0000, 1010, 1100, 0010, 0100, 1000, 0210, 2010, 2100, 3210, \text{etc.}$

Theorem (Littlewood-Richardson rule)

$c_{\lambda, \mu}^{\nu}$ is the number of skew semistandard tableaux of shape ν/λ , whose row reading is a Yamanouchi word of evaluation μ .

$$C_{331,421}^{5432} = 3$$

$$C_{431,4321}^{7542} = 4$$

$$C_{4321,431}^{7542} = 4$$

for a proof, see Lascoux, Leclerc, and Thibon, *The Plactic monoid*, in M. Lothaire, Algebraic combinatorics on words, CUP.

- stated (1934) by D. E. Littlewood and A. R. Richardson, wrong proof, wrong example
- Robinson (1938), wrong completed proof
- First correct proof: Schützenberger (1977)
- Dozens of theses and papers about this proof (Zelevinsky 1981, Macdonald 1995, Gasharov 1998, Duchamp-H-Thibon 2001, van Leeuwen 2001, Stembridge 2002)

Wikipedia: *The Littlewood–Richardson rule is notorious for the number of errors that appeared prior to its complete, published proof. Several published attempts to prove it are incomplete, and it is particularly difficult to avoid errors when doing hand calculations with it: even the original example in D. E. Littlewood and A. R. Richardson (1934) contains an error.*

- #-P complete (Narayanan, 2005)

- multiplicity of induction or restriction of irreducible representations of the symmetric groups
- multiplicity of the tensor product of the irreducible representations of linear groups
- geometry: number of intersections in a Grassmanian variety, cup product of the cohomology
- Horn problem: eigenvalues of the sum of two Hermitian matrices
- extension of Abelian groups (Hall algebra)
- application in quantum physics (spectrum rays of the Hydrogen atoms)

a Coq proof of the Littlewood–Richardson rule

a Coq proof of the Littlewood–Richardson rule

- author: Florent Hivert
- uses the Ssreflect extension of Coq and the Mathematical Components libraries
- 15,000 lines of script
- available at <https://github.com/hivert/Coq-Combi>

outline of the proof

first introduce Schur functions and coefficients $c_{\lambda, \mu}^{\nu}$

```
Schur P1 * Schur P2 =  
  \sum_P (Schur P) *+ LRtab_coeff P1 P2 P
```

then introduce tableaux, Yamanouchi words, and define

```
Definition LRyam_set P1 P2 P :=  
  [ set y : yameval_finType P2 |  
    is_skew_reshape_tableau P P1 y ].
```

```
Definition LRyam_coeff P1 P2 P :=  
  #|LRyam_set P1 P2 P|.
```

finally, prove

```
Theorem LR_coeff_yamP:  
  forall P1 P2 P, included P1 P →  
  LRtab_coeff P1 P2 P = LRyam_coeff P1 P2 P.
```

an implementation

the Coq proof also includes an **implementation**, i.e. a function that uses backtracking to compute the coefficients

```
LRcoeff: seq nat → seq nat → seq nat → nat
```

it comes with a proof of correctness

Theorem LR_yamtabE:

```
forall P1 P2 P, included P1 P →  
LRyam_coeff P1 P2 P = LRcoeff P1 P2 P.
```

an OCaml program

using Coq extraction mechanism, we get an OCaml code

```
type nat = 0 | S of nat
...
val lRcoeff: nat list → nat list → nat list → nat
```

with glue code to parse the command line, we can play with it

```
./lRcoeff 11 10 9 8 7 6 5 4 3 2 1 - 7 6 5 5 4 3 2 1 -  
          7 6 5 5 4 3 2 1  
268484
```

this program is rather inefficient
(2 orders of magnitude slower than the C library `lrcalc`)

because

- it uses **unary Peano numbers** for
 - partitions
 - tableaux indices
 - tableaux values
 - solution count
- it uses **linear time** `nth/upd` functions on lists everywhere

in particular, the OCaml GC is heavily solicited

an efficient program and its proof

an efficient OCaml implementation using

- arrays instead of lists
- efficient arithmetic instead of type `nat`
 - either arbitrary-precision arithmetic with GMP
 - or even better machine arithmetic

we use Why3 to do this

Why3 in a nutshell

a program verifier featuring

- a **first-order polymorphic logic** with recursive definitions, algebraic data types, inductive predicates
- support for **many theorem provers**, both automated and interactive (Z3, Alt-Ergo, Coq, etc.)
- an **ML-like programming language** with behavioral specifications, a VCgen, and an automatic translation to OCaml

four arrays

outer (ν)	<table border="1"><tr><td>7</td><td>5</td><td>4</td><td>2</td></tr></table>	7	5	4	2		
7	5	4	2				
inner (λ)	<table border="1"><tr><td>4</td><td>3</td><td>1</td><td>0</td></tr></table>	4	3	1	0	(same length as outer)	
4	3	1	0				
eval (μ)	<table border="1"><tr><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr></table>	4	3	2	1	0	(ends with a 0 sentinel)
4	3	2	1	0			
innev	<table border="1"><tr><td>3</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	3	1	0	0	0	
3	1	0	0	0			

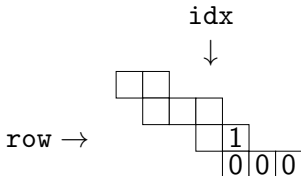
a matrix for the tableau under construction

work

				1		
				0	0	0

backtracking algorithm

```
count(row, idx) =  
  if row = length(outer) then return 1 // found one solution  
  if idx < 0 then return count(row + 1, ...) // move to next row  
  s ← 0  
  for v in the range of possible values for work[row, idx]  
    work[row, idx] ← v  
    innev[v] ← innev[v] + 1  
    s ← s + count(row, idx - 1) // move to next cell  
    innev[v] ← innev[v] - 1  
  return s
```



we prove

- termination
- absence of array access out of bounds
- absence of arithmetic overflow
- correctness: computes the Littlewood–Richardson coefficient

proof: termination

immediate: as we make progress in the tableau, the quantity

$$(\text{length}(\text{outer}) - \text{row}, \text{idx})$$

decreases lexicographically

proof: absence of array access out of bounds

not very difficult, but requires

- the 0 sentinel in `eval`
- some invariants
(e.g. values in `work` are smaller than `length(eval)`)
- some frame properties

proof: absence of arithmetic overflow

- with arbitrary precision arithmetic
 - ⇒ nothing to do
 - ⇒ but will translate to arbitrary precision arithmetic (GMP)
- with 64-bit machine arithmetic
 - array indices and tableaux values indeed fit in 64 bits (both bounded by some array length, itself a 64-bit integer)
 - ⇒ all VCs easily discharged
 - but what about the value returned (the total count)?

Peano arithmetic

to implement the counter, we use a different, abstract type of integers, with limited operations

```
type peano
constant zero: peano
function succ (p:peano) : peano
```

compiled into OCaml's type of 64-bit integers

see: Martin Clochard, Jean-Christophe Filliâtre, Andrei Paskevich.
How to avoid proving the absence of integer overflows. VSTTE
2015

$\nu = 24, 24, 22, 20, 18, 18, 16, 16, 15, 13, 10, 8, 8, 7, 5, 5, 5, 3, 2, 1$
 $\lambda = 20, 18, 18, 18, 18, 16, 15, 13, 13, 11, 10, 8, 7, 6, 5, 5, 3, 2, 2, 0$
 $\mu = 10, 8, 5, 4, 3, 2, 0$

$$c_{\lambda, \mu}^{\nu} = 13,911,775$$

extracted from Coq	extracted from Why3		C library lrcalc
	ZArith	machine arith	
31.6	26.7	1.18	0.35

proof of correctness

```
predicate is_part (a: array int) =  
  ... a is a partition ...
```

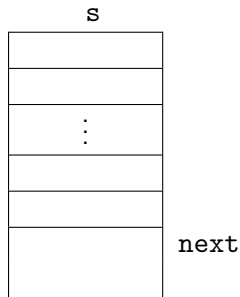
```
predicate valid_input (outer inner: array int) =  
  ... inner and outer are partitions ...  $\wedge$   
  ... inner is included in outer ...
```

```
predicate valid_eval (eval: array int) =  
  ... eval is a partition ...  $\wedge$   
  ... ends with at least a 0 ...
```

ghost code stores solutions within a global (ghost) array s

```
predicate is_solution work =  
  ...
```

```
predicate good_solutions s =  
  ... s is sorted ...  $\wedge$   
  ... s contains only solutions ...  $\wedge$   
  ... s contains all solutions ...
```



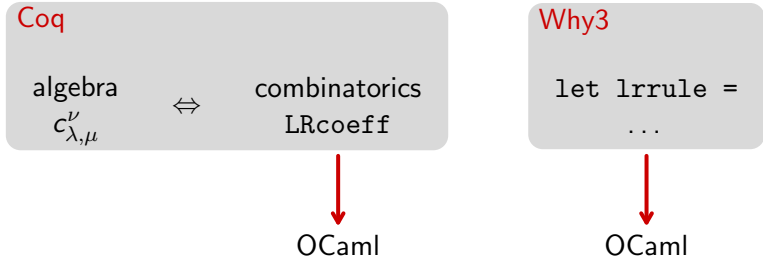
```
let lrrule (outer inner eval: array int) : int =  
  requires { valid_input outer inner }  
  requires { valid_eval eval }  
  requires { sum_array eval =  
            sum_array outer - sum_array inner }  
  requires { s.next = 0 }  
  ensures { result = s.next }  
  ensures { good_solutions outer inner eval s }
```

note: the proof is **not yet completed**

connection with the Coq proof

a legitimate question

are the Why3 and Coq programs computing the same thing?

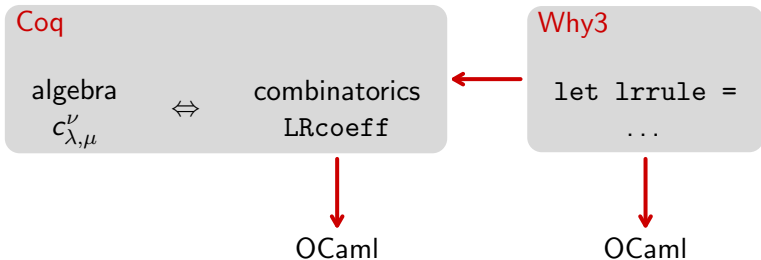


after all, we have

- distinct data structures (nats/lists vs ints/arrays)
- different specifications (partition, tableau, Yamanouchi word)
- different algorithms

the big picture

let us show the equivalence, using Coq



all Why3 definitions are translated into Coq, automatically
(for Why3, Coq is a prover like any other)

proof of equivalence

in Coq, we define two functions `part` and `partw` to convert partitions from one type to the other

```
part: array int → seq nat
```

```
partw: nat → seq nat → array int
```

(the first argument is the length)

proof of equivalence

Coq

from Why3

inner

partw
→

inner

outer

outer

eval

part
←

eval

inputSpec

valid_input

valid_eval

outputSpec

is_solution

LRcoeff

good_solutions

proof of equivalence

Theorem Why3Correct:

```
forall innerw outerw evalw: array int,  
valid_input outerw innerw →  
valid_eval evalw →  
sum_array evalw =  
  sum_array outerw - sum_array innerw →  
forall s, good_solutions innerw outerw evalw s →  
s.next =  
  LRcoeff (part innerw) (part outerw) (part evalw)
```

the other way round

Theorem Why3complete:

```
forall inner outer eval: seq nat,  
inputSpec inner eval outer →  
let l = max (1 + size eval)  
          (max (size inner) (size outer)) in  
forall s,  
good_solutions (partw l inner)  
                (partw l outer) (partw l eval) s →  
s.next = LRcoeff inner outer eval
```

what is the TCB?

- the Coq definitions and statements
- the definition of `partw`

```
Definition partw (l: nat) (p: seq nat) : array int :=  
  mkseq (fun i => Posz (nth 0 p i)) l.
```

in particular, you don't have to read **any** Why3 definition

we have presented

- an efficient program to compute the Littlewood–Richardson coefficients
- its proof of safety, including absence of arithmetic overflows
- its proof of correctness (WIP)
- its equivalence, at the specification level, with another program already proved correct

long-term goal: have such verified software integrated in SageMath (Florent is one of the main developers of SageMath)