A formal proof of the Littlewood-Richardson rule

Florent Hivert

LRI / Université Paris Sud 11 / CNRS

September 2018

Outline

1. Motivation : certified proof in combinatorics
2. A short introduction to formal proof in Coq/Mathcomp
3. The Little-Richardson rule
4. Some hard points of the formal proof
5. Should you try ?

Sommaire

Why formalize things on computers

Writing correct programs is hard:

- The human mind is focused on the big picture;
- Hard to take track of all the trivial / particular cases.

Some excerpts of my contribution to `Sagemath`:

- determinant / rank / invertibility of 0×0 and 1×1 matrices
- empty set and its permutation
- empty partition / composition / parking function / tableau ...
- the 0 and 1 species
- ...

What about proofs ?

Why formalize things on computers

Writing correct programs is hard:

- The human mind is focused on the big picture;
- Hard to take track of all the trivial / particular cases.

Some excerpts of my contribution to `Sagemath`:

- determinant / rank / invertibility of 0×0 and 1×1 matrices
- empty set and its permutation
- empty partition / composition / parking function / tableau . . .
- the 0 and 1 species
- . . .

What about proofs ?

Why formalize things on computers

Writing correct programs is hard:

- The human mind is focused on the big picture;
- Hard to take track of all the trivial / particular cases.

Some excerpts of my contribution to `Sagemath`:

- determinant / rank / invertibility of 0×0 and 1×1 matrices
- empty set and its permutation
- empty partition / composition / parking function / tableau ...
- the 0 and 1 species
- ...

What about proofs ?

Are our proofs always correct ?

Donald Knuth:

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*

Often in combinatorics, and particularly in **bijective** combinatorics, proofs are algorithms, together with justifications that they meet their specifications...

Are our proofs always correct ?

Donald Knuth:

> *Beware of bugs in the above code; I have only proved it correct, not tried it.*

Often in combinatorics, and particularly in **bijective** combinatorics, proofs are algorithms, together with justifications that they meet their specifications...

Are our proofs always correct ?

The Littlewood-Richardson rule:

- stated (1934) by D. E. Littlewood and A. R. Richardson, wrong proof, wrong example.
- Robinson (1938), wrong completed proof.
- more wrong published proofs...
- first correct proof: Schützenberger (1977).
- nowadays: dozens of different proofs. . .

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.*

The case of the Littlewood-Richardson rule ?

A footnote in Macdonald's book:

> *Gordon James reports that he was once told that:*
> *"The Littlewood-Richardson rule helped to get men on the moon, but it was not proved until after they had got there. The first part of this story might be an exaggeration."*

This sentence appears in James, G. D. (1987) *The representation theory of the symmetric groups*.

The case of the Littlewood-Richardson rule ?

More quotation of James:

> *It seems that for a long time* ***the entire body of experts in the field was convinced*** *by these proofs; at any rate it was not until 1976 that McConnell pointed out* ***a subtle ambiguity*** *in part of the construction underlying the argument.*
>
> *[...]*
>
> *How was it possible for an incorrect proof of such a central result in the theory of S_n to have been accepted for close to forty years? The level of rigor customary among mathematicians when a combinatorial argument is required, is (probably quite rightly) of the nonpedantic hand-waving kind; perhaps one lesson to be drawn is that a* ***higher degree of care*** *will be needed in dealing with such combinatorial complexities as occur in the present level of development of Young's approach.*

Problem

*Suppose that, back in 1977, they had had our current proof assistant technology. Would it have been **feasible** to check Schützenberger proof ? If so, **how long** would it have taken ?*

Theorem (Constructive answer !)

Yes ! Less than 5 month and two weeks !

```
commit f990146b8c6e062fe025740a08f888deb9481c2d
Date:   Thu Jul 24 17:46:58 2014 +0200
    Schensted's algorithm.

commit 2418282695455261e5459b33d3e8f979d57c3bdb
Date:   Sun Jan 4 15:31:16 2015 +0100
    DONE the proof of the Littlewood_Richardson rule !!!!
```

Problem

Suppose that, back in 1977, they had had our current proof assistant technology. Would it have been ***feasible*** *to check Schützenberger proof ? If so,* ***how long*** *would it have taken ?*

Theorem (Constructive answer !)

Yes ! Less than 5 month and two weeks !

```
commit f990146b8c6e062fe025740a08f888deb9481c2d
Date:   Thu Jul 24 17:46:58 2014 +0200
    Schensted's algorithm.

commit 2418282695455261e5459b33d3e8f979d57c3bdb
Date:   Sun Jan 4 15:31:16 2015 +0100
    DONE the proof of the Littlewood_Richardson rule !!!!
```

Sommaire

History of Coq and Mathcomp

- 1985 – T. Coquand : *Calculus of constructions*
- 1989 – T. Coquand, G. Huet: creation of `Coq`
- 2004 – G. Gonthier, B. Werner : *4 color theorem* in Coq
  Along their way `Ssreflect` "small scale reflection".
- 2006 – 2018 Mathematical component: a library of formalized mathematics.
  - basic data structures, algebra, group an representation theory;
  - the infrastructure for the machine checked proofs of:
- 2012 – Coq checked proof of Feit-Thomson's theorem:
  Every finite group of odd order is solvable.

Formal (mechanized) proofs

Aim

Write a proof that is checked by computer all the way down to the logical foundation.

Proof assistant / interactive theorem prover :
A kind of Integrated Development Environment (IDE) which helps writing such proofs by constantly checking the coherence and keeping track of missing parts.

Note: Proof assistant $\neq$ automated theorem prover

Formal (mechanized) proofs

Aim

Write a proof that is checked by computer all the way down to the logical foundation.

Proof assistant / interactive theorem prover :
A kind of Integrated Development Environment (IDE) which helps writing such proofs by constantly checking the coherence and keeping track of missing parts.

Note: Proof assistant $\neq$ automated theorem prover

What is needed to build a proof assistant ?

Three ingredients:

1. A way to **store algorithms** that allows for **manipulating** them **and reasoning** about them;
2. A way to **store proofs** that allows for **manipulating** them **and reasoning** about them;
3. A way to **mechanically check** everything.

What is needed to build a proof assistant ?

Three ingredients:

1. A way to **store algorithms** that allows for **manipulating** them **and reasoning** about them;
2. A way to **store proofs** that allows for **manipulating** them **and reasoning** about them;
3. A way to **mechanically check** everything.

What is needed to build a proof assistant ?

Three ingredients:

1. A way to **store algorithms** that allows for **manipulating** them **and reasoning** about them;
2. A way to **store proofs** that allows for **manipulating** them **and reasoning** about them;
3. A way to **mechanically check** everything.

Proofs as programs (Curry-Howard)

Suppose that

- we have **data** encoding a proof a and two statements A and B
- the system is able to make so-called **judgments**:
  to verify that a is a correct proof of A (written as $a : A$)

Then, the statement $A \to B$ means that each time we have a proof of A, we can construct a proof of B.

Curry-Howard correspondence in a nutshell

The idea is "simply" to **encode a proof of $A \to B$ by a function** (= a program) which takes a proof of A and returns a proof of B.

Similarly, a proof of $\forall x, P(x)$, is encoded as a function which takes x and returns a proof of $P(x)$.

Proofs as programs (Curry-Howard)

Suppose that

- we have **data** encoding a proof a and two statements A and B
- the system is able to make so-called **judgments**:
  to verify that a is a correct proof of A (written as $a : A$)

Then, the statement $A \to B$ means that each time we have a proof of A, we can construct a proof of B.

Curry-Howard correspondence in a nutshell

The idea is “simply” to **encode a proof of $A \to B$ by a function** (= a program) which takes a proof of A and returns a proof of B.

Similarly, a proof of $\forall x, P(x)$, is encoded as a function which takes x and returns a proof of $P(x)$.

Proofs as programs (Curry-Howard)

Suppose that

- we have **data** encoding a proof a and two statements A and B
- the system is able to make so-called **judgments**:
  to verify that a is a correct proof of A (written as $a : A$)

Then, the statement $A \to B$ means that each time we have a proof of A, we can construct a proof of B.

Curry-Howard correspondence in a nutshell

The idea is "simply" to **encode a proof of $A \to B$ by a function** (= a program) which takes a proof of A and returns a proof of B.

Similarly, a proof of $\forall x, P(x)$, is encoded as a function which takes x and returns a proof of $P(x)$.

Proofs as programs (Curry-Howard)

Suppose that

- we have **data** encoding a proof a and two statements A and B
- the system is able to make so-called **judgments**:
  to verify that a is a correct proof of A (written as $a : A$)

Then, the statement $A \to B$ means that each time we have a proof of A, we can construct a proof of B.

Curry-Howard correspondence in a nutshell

The idea is "simply" to **encode a proof of $A \to B$ by a function** (= a program) which takes a proof of A and returns a proof of B.

Similarly, a proof of $\forall x, P(x)$, is encoded as a function which takes x and returns a proof of $P(x)$.

Type theory based proof assistants

Proof assistant = a system that:

- **manipulates** (stores, executes, ...) functions (Λ-calculus)
- **checks judgments** such as $a : A$ (typed Λ-calculus)

To make it more usable, we need also

- **building blocks** for custom data structures: **records, unions** (Calculus of Inductive Construction $\approx$ Galina)
- **helpers** for writing proof/programs incrementally (tactic language).

Type theory based proof assistants

Proof assistant = a system that:

- **manipulates** (stores, executes, ...) functions (Λ-calculus)
- **checks judgments** such as $a : A$ (typed Λ-calculus)

To make it more usable, we need also

- **building blocks** for custom data structures: **records, unions** (Calculus of Inductive Construction $\approx$ Galina)
- **helpers** for writing proof/programs incrementally (tactic language).

Type theory based proof assistants

Proof assistant = a system that:

- **manipulates** (stores, executes, ...) functions (Λ-calculus)
- **checks judgments** such as $a : A$ (typed Λ-calculus)

To make it more usable, we need also

- **building blocks** for custom data structures: **records, unions** (Calculus of Inductive Construction ≈ Galina)
- **helpers** for writing proof/programs incrementally (tactic language).

You only need to remember:

Summary

- proof, statement, data, programs, etc are all the same first class manipulated objects called **terms**
- some terms are allowed (from the logic or by their definition) to appear on the right of the judgment symbol ":". They are called **types**. They encode usual **data types** as well as **statement**
- every term has a type

Enough for the theory. . .

Time for a demo. . .

You only need to remember:

Summary

- proof, statement, data, programs, etc are all the same first class manipulated objects called **terms**
- some terms are allowed (from the logic or by their definition) to appear on the right of the judgment symbol ":". They are called **types**. They encode usual **data types** as well as **statement**
- every term has a type

Enough for the theory...

Time for a demo...

Boolean reflection

Two ways to deal with statements:

- **inductive formulas** (*i.e.* data structure storing a proof): `and, or, exist...`:
  $\Rightarrow$ good for reasoning, deducing, implication chaining...
- **decision procedure** (*i.e.* function returning a boolean):
  $\Rightarrow$ good for combinatorial analysis, automatically taking care of trivial cases...

Boolean reflection

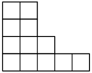
Going back and forth between the two ways:

```
reflect (maxn m n = m) (m >= n).
reflect (exists2 x : T, x \in s & a x) (has a s)
reflect (filter s = s) (all s)
reflect (forall x, x \in s -> a x) (all a s).
reflect (exists2 i, i < size s & nth x0 s i = x) (x \in s).
```

Sommaire

Integer Partitions

Partition: $\lambda := (\lambda_0 \geq \lambda_1 \geq \cdots \geq \lambda_l > 0)$.
$|\lambda| := \lambda_0 + \lambda_1 + \cdots + \lambda_l$ et $\ell(\lambda) := l$.

Ferrer's diagram of a partitions : $(5, 3, 2, 2) \leftrightarrow$ 

is_part

```
Fixpoint is_part sh := (* Predicate *)
  if sh is sh0 :: sh'
  then (sh0 >= head 1 sh') && (is_part sh')
  else true.

Lemma is_partP sh : reflect (* Boolean reflection lemma *)
  (last 1 sh != 0 /\ forall i, (nth 0 sh i) >= (nth 0 sh i.+1))
  (is_part sh).

Lemma is_part_ijP sh : reflect (* Boolean reflection lemma *)
  (last 1 sh != 0 /\ forall i j, i <= j -> (nth 0 sh i) >= nth 0 sh j)
  (is_part sh).

Lemma is_part_sortedE sh : (is_part sh) = (sorted geq sh) && (0 \notin sh).
```

Symmetric Polynomials

n-variables : $\mathbb{X}_n := \{x_0, x_1, \dots x_{n-1}\}$.
Polynomials in $\mathbb{X}$: $\mathbb{C}[\mathbb{X}] = \mathbb{C}[x_0, x_1, \dots, x_{n-1}]$; ex: $3x_0^3x_2 + 5x_1x_2^4$.

Definition (Symmetric polynomial)

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

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

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

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

Schur symmetric polynomials (Jacobi)

Definition (Schur symmetric polynomial)

Partition $\lambda := (\lambda_0 \geq \lambda_1 \geq \cdots \geq \lambda_{l-1})$ *with* $l \leq n$*; set* $\lambda_i := 0$ *for* $i \geq l$.

$$s_\lambda = \frac{\sum_{\sigma\in\mathfrak{S}_n} \operatorname{sign}(\sigma)\mathbb{X}_n^{\sigma(\lambda+\rho)}}{\prod_{0\leq i<j<n}(x_j - x_i)} = \frac{\begin{vmatrix} x_1^{\lambda_{n-1}+0} & x_2^{\lambda_{n-1}+0} & \dots & x_n^{\lambda_{n-1}+0} \\ x_1^{\lambda_{n-2}+1} & x_2^{\lambda_{n-2}+1} & \dots & x_n^{\lambda_{n-2}+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{\lambda_1+n-2} & x_2^{\lambda_1+n-2} & \dots & x_n^{\lambda_1+n-2} \\ x_1^{\lambda_0+n-1} & x_2^{\lambda_0+n-1} & \dots & x_n^{\lambda_0+n-1} \end{vmatrix}}{\begin{vmatrix} 1 & 1 & \dots & 1 \\ x_1 & x_2 & \dots & x_n \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{n-1} & x_2^{n-1} & \dots & x_n^{n-1} \end{vmatrix}}$$

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

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 $c^\nu_{\lambda,\mu}$ of the expansion of the product:

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

Fact: $s_\lambda(\mathbb{X}_{n-1}, x_n := 0) = s_\lambda(\mathbb{X}_{n-1})$ if $\ell(\lambda) < n$ else 0.
Consequence: $c^\nu_{\lambda,\mu}$ are independant of the number of variables.

Young Tableau

Definition

- *Filling of a partition shape;*
- *non decreasing along the rows;*
- *strictly increasing along the columns.*

- *row reading*

$$\begin{array}{|c|c|c|c|c|c|c|}\hline d & d & e \\ \cline{1-3} b & c & c & c & d \\ \cline{1-5} a & a & a & b & b & d & e \\ \hline \end{array} = ddebcccdaaabbde$$

$$\begin{array}{|c|c|c|c|c|}\hline 5 \\ \cline{1-1} 2 & 6 & 9 \\ \cline{1-3} 1 & 3 & 4 & 7 & 8 \\ \hline \end{array} = 526913478$$

Young Tableau: formal definition

dominate

```
Variable (T : ordType) (Z : T). (* Ordered type Order <A *)

Definition dominate (u v : seq T) :=
  (size u <= size v &&
   (all (fun i => nth Z u i >A nth Z v i) (iota 0 (size u)))).

Lemma dominateP u v :
  reflect (size u <= size v /\
             forall i, i < size u -> nth Z u i >A nth Z v i)
          (dominate u v).
```

is_tableau

```
Fixpoint is_tableau (t : seq (seq T)) :=
  if t is t0 :: t' then
    [&& (t0 != [::]), sorted t0,
      dominate (head [::] t') t0 & is_tableau t']
  else true.

Definition to_word t := flatten (rev t).
```

Combinatorial definition of Schur functions

Definition

$$s_\lambda(\mathbb{X}) = \sum_{T \text{ tableaux of shape } \lambda} \mathbb{X}^T$$

where $\mathbb{X}^T$ *is the product of the elements of* T*.*

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

$$s_{(2,1)}(a,b,c) = \begin{array}{|c|c}\hline b & \\ \hline a & \multicolumn{1}{c|}{a} \\ \hline\end{array} + \begin{array}{|c|c}\hline b & \\ \hline a & \multicolumn{1}{c|}{b} \\ \hline\end{array} + \begin{array}{|c|c}\hline c & \\ \hline a & \multicolumn{1}{c|}{a} \\ \hline\end{array} + \begin{array}{|c|c}\hline b & \\ \hline a & \multicolumn{1}{c|}{c} \\ \hline\end{array} + \begin{array}{|c|c}\hline c & \\ \hline a & \multicolumn{1}{c|}{b} \\ \hline\end{array} + \begin{array}{|c|c}\hline c & \\ \hline b & \multicolumn{1}{c|}{b} \\ \hline\end{array} + \begin{array}{|c|c}\hline c & \\ \hline a & \multicolumn{1}{c|}{c} \\ \hline\end{array} + \begin{array}{|c|c}\hline c & \\ \hline b & \multicolumn{1}{c|}{c} \\ \hline\end{array}$$

Note: I'll prove the equivalence of the two definitions as a *consequence of a particular case of the LR-rule (Pieri rule)* by relating it with recursively unfolding determinants.

Combinatorial definition of Schur functions

Definition

$$s_\lambda(\mathbb{X}) = \sum_{T \text{ tableaux of shape } \lambda} \mathbb{X}^T$$

where $\mathbb{X}^T$ is the product of the elements of T.

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

$$s_{(2,1)}(a,b,c) = \begin{array}{|c|c|}\cline{1-1} b & \multicolumn{1}{c}{} \\ \hline a & a \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} b & \multicolumn{1}{c}{} \\ \hline a & b \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} c & \multicolumn{1}{c}{} \\ \hline a & a \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} b & \multicolumn{1}{c}{} \\ \hline a & c \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} c & \multicolumn{1}{c}{} \\ \hline a & b \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} c & \multicolumn{1}{c}{} \\ \hline b & b \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} c & \multicolumn{1}{c}{} \\ \hline a & c \\ \hline\end{array} + \begin{array}{|c|c|}\cline{1-1} c & \multicolumn{1}{c}{} \\ \hline b & c \\ \hline\end{array}$$

Note: I'll prove the equivalence of the two definitions as a *consequence of a particular case of the LR-rule (Pieri rule)* by relating it with recursively unfolding determinants.

tabsh

```
Variable n : nat.
Variable R : comRingType.  (* Commutative ring *)

(* 'I_n          : integer in 0,1,...,n-1                              *)
(* 'P_d          : partition of the integer d                          *)
(* {mpoly R[n]} : the ring of polynomial over the commutative ring R *)
(*                 in n variables (P.-Y. Strub)                        *)

Definition is_tab_of_shape sh :=
  [ pred t :> seq (seq 'I_n.+1) | (is_tableau t) && (shape t == sh) ].

Structure tabsh sh := TabSh {tabshval; _ : is_tab_of_shape sh tabshval}.
[...]
Canonical tabsh_finType sh := [...] (* Finite type *)
```

Schur

```
Definition Schur d (sh : 'P_d) : {mpoly R[n]} :=
  \sum_(t : tabsh n sh) \prod_(i <- to_word t) 'X_i.
```

Yamanouchi Words

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

Definition

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

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

Equivalently $(|w|_i)_{i \leq \max(w)}$ is a partition and $w_1, \dots, w_{l-1}$ is also Yamanouchi.

$$(), 0, 00, 10, 000, 100, 010, 210,$$

$$0000, 1010, 1100, 0010, 0100, 1000, 0210, 2010, 2100, 3210$$

The LR Rule at last !

Theorem (Littlewood-Richardson rule)

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

$$C^{5432}_{331,421} = 3 \qquad \begin{array}{ccccc} 1&2&&& \\ &0&0&& \\ &&&1& \\ &&&0&0 \end{array} \qquad \begin{array}{ccccc} 0&2&&& \\ &0&1&& \\ &&&1& \\ &&&0&0 \end{array} \qquad \begin{array}{ccccc} 0&1&&& \\ &0&2&& \\ &&&1& \\ &&&0&0 \end{array}$$

$$C^{7542}_{431,4321} = 4 \qquad \begin{array}{ccccccc} 2&3&&&&& \\ &1&1&2&&& \\ &&&0&1&& \\ &&&&0&0&0 \end{array} \qquad \begin{array}{ccccccc} 2&3&&&&& \\ &0&1&2&&& \\ &&&1&1&& \\ &&&&0&0&0 \end{array} \qquad \begin{array}{ccccccc} 1&3&&&&& \\ &0&2&2&&& \\ &&&1&1&& \\ &&&&0&0&0 \end{array} \qquad \begin{array}{ccccccc} 0&3&&&&& \\ &1&2&2&&& \\ &&&1&1&& \\ &&&&0&0&0 \end{array}$$

$$C^{7542}_{4321,431} = 4 \qquad \begin{array}{cccccc} 2&&&&& \\ &1&1&&& \\ &&0&1&& \\ &&&0&0&0 \end{array} \qquad \begin{array}{cccccc} 1&&&&& \\ &1&2&&& \\ &&0&1&& \\ &&&0&0&0 \end{array} \qquad \begin{array}{cccccc} 1&&&&& \\ &0&2&&& \\ &&1&1&& \\ &&&0&0&0 \end{array} \qquad \begin{array}{cccccc} 0&&&&& \\ &1&2&&& \\ &&1&1&& \\ &&&0&0&0 \end{array}$$

The formal statement

definition of LR-yam tableaux

```
(** yameval P = type of Yamanouchi word of evaluation P *)
Lemma is_skew_reshape_tableauP (w : seq nat) :
  size w = sumn (P / P1) ->
  reflect
    (exists tab, [/\ is_skew_tableau P1 tab,
                     shape tab = P / P1 & to_word tab = w])
    (is_skew_reshape_tableau P P1 w).

Definition LRyam_set :=
  [set y : yameval P2 | is_skew_reshape_tableau P P1 y].
Definition LRyam_coeff := #|LRyam_set|.
```

Then

the Littlewood-Richardson rule

```
Theorem LRyam_coeffP :
  Schur P1 * Schur P2 =
  \sum_(P : 'P_(d1 + d2) | included P1 P) Schur P *+ LRyam_coeff P.
```

Sommaire

Getting definition right

There is not a single “good” definition:

- Lots of different equivalent ways. (eg: partitions, tableaux)
- Even more difficult for algorithms (standardization, shuffle):

  Constraints:
  - **Pure functional** programming:
    no variable, no mutable data structure
  - All function must be **total** (e.g. `nth` but `option`)
  - Only trivially **terminating programs** are allowed:
    Only recursive call on subterms are allowed.

Getting definition right

There is not a single “good” definition:

- Lots of different equivalent ways. (eg: partitions, tableaux)
- Even more difficult for algorithms (standardization, shuffle):

  Constraints:
  - **Pure functional** programming:
    no variable, no mutable data structure
  - All function must be **total** (e.g. `nth` but `option`)
  - Only trivially **terminating programs** are allowed:
    Only recursive call on subterms are allowed.

Choices in constructive mathematics

Sometime you have a choice to make where **any choice will do**:

Example: contructing a **conjugating permutation** between two permutations with same cycle type: Conjugacy classes of S_n

```
(* (s ^ c)%g == s conjugated by c *)
Theorem conj_permP s t :
  reflect (exists c, t = (s ^ c)%g) (cycle_type s == cycle_type t).
```

- currently, you have to write a **precise program** to make the choice and prove that it works
- the proof is usually easy because any choice will do
- but writing the program making the choice can be harder than expected.

Choices in constructive mathematics

Sometime you have a choice to make where **any choice will do**:

Example: contructing a **conjugating permutation** between two permutations with same cycle type: Conjugacy classes of S_n

```
(* (s ^ c)%g == s conjugated by c *)
Theorem conj_permP s t :
  reflect (exists c, t = (s ^ c)%g) (cycle_type s == cycle_type t).
```

- currently, you have to write a **precise program** to make the choice and prove that it works
- the proof is usually easy because any choice will do
- but writing the program making the choice can be harder than expected.

Equality / set theory

Coq is based on CIC $\neq$ set theory.

- Constructive logic (not that much a problem in combinatorics)
  Excluded middle `P \/ ~ P` is not provable (but can be added as an axiom).

- SSReflect deals smoothly with objects with decidable equality

  This forbids generating series !
  but see C. Cohen, B. Djalal *Formalization of a Newton Series Representation of Polynomials*

- The equality in type theory is "stronger" that in set theory
  No proof of functional extensionality:

```
(forall x, f x = f y) -> x = y
```

Greene Theorem

Disjoint support increasing subsequences:

$$ababcabbadbab$$

$RS(w)$: Robinson-Schensted tableau of w:

Theorem

For any word w, and integer k

- *The sum of the length of the k-first row of $RS(w)$ is the maximum sum of the length of k disjoint support increasing subsequences of w;*

Greene Theorem

Disjoint support increasing subsequences:

$$ababcabbadbab$$

$RS(w)$: Robinson-Schensted tableau of w:

Theorem

For any word w, and integer k

- *The sum of the length of the k-first row of $RS(w)$ is the maximum sum of the length of k disjoint support increasing subsequences of w;*

Equality on dependent type nightmare

Subsequences of a word *w* encoded by subsets of the indices of the letter of *w*: `{set 'I_(size w)}`. But, when

- `x := u ++ [:: a; b] ++ v`
- `y := u ++ [:: b; a] ++ v`

`x` and `y` are two different words !
`{set 'I_(size x)}` and `{set 'I_(size y)}`: different types

Equality on dependent type

On can only write `u = v` if `u` and `v` are of the **same type**.

Cast between dependent type nightmare

Here is a solution:

- Prove that `Hcast : size x = size y`
- Then `ordcast Hcast : 'I_(size w) -> 'I_(size y)`
- Then define:

cast_set

```
(*  f @: S == image of S by the function f *)
Definition cast_set n m (H : n = m) : {set 'I_n} -> {set 'I_m} :=
  [fun s : {set 'I_n} => (cast_ord H) @: s].
```

swap_set

```
Definition swap (i : 'I_(size x)) : 'I_(size x) :=
  if i == pos0 then pos1 else if i == pos1 then pos0 else i.
Definition swap_setX :=
  [fun S : {set 'I_(size x)} => swap @: S : {set 'I_(size x)}].
Definition swap_set : {set 'I_(size x)} -> {set 'I_(size y)} :=
  (fun S : {set 'I_(size x)} =>
       [set cast_ord Hcast x | x in S]) \o swap_setX.
```

Sommaire

Should you try (or is this a big waste of time) ? My two cents

- I'm pretty convinced (I'm not the only one: Voevodsky, Hales) that in the future (how far ?), formal math (not Coq/CIC) will becomes very important (as is computation today).
- However, currently, experts are **not satisfied with the foundation** (equality. . .).
- This was **much easier** that I first expected !
- Boolean reflection : very good job **dealing with trivial cases**

Should you try (or is this a big waste of time) ? My two cents

- I'm pretty convinced (I'm not the only one: Voevodsky, Hales) that in the future (how far ?), formal math (not Coq/CIC) will becomes very important (as is computation today).
- However, currently, experts are **not satisfied with the foundation** (equality. . .).
- This was **much easier** that I first expected !
- Boolean reflection : very good job **dealing with trivial cases**

Should you try (or is this a big waste of time) ? My two cents

- I'm pretty convinced (I'm not the only one: Voevodsky, Hales) that in the future (how far ?), formal math (not Coq/CIC) will becomes very important (as is computation today).
- However, currently, experts are **not satisfied with the foundation** (equality. . .).
- This was **much easier** that I first expected !
- Boolean reflection : very good job **dealing with trivial cases**

Should you try (or is this a big waste of time) ? My two cents

- I'm pretty convinced (I'm not the only one: Voevodsky, Hales) that in the future (how far ?), formal math (not Coq/CIC) will becomes very important (as is computation today).
- However, currently, experts are **not satisfied with the foundation** (equality. . .).
- This was **much easier** that I first expected !
- Boolean reflection : very good job **dealing with trivial cases**

Should you try (or is this a big waste of time) ? My two cents

- Lots of time spent on **reusable basic stuff** (tableau / partition / rewriting systems / symmetric fncts)...
- Some other results:
  - The hook-length formulas: 3 weeks (joint work w. C. Paulin)
  - Cycle decomposition: 2 months (T. Benjamin, undergrad)
  - Basic theory of symmetric functions: 3 months
  - Character theory of the symmetric groups: 1 month
- This is transforming math into a video-game

Fun if you like it, but very addictive !

Should you try (or is this a big waste of time) ? My two cents

- Lots of time spent on **reusable basic stuff** (tableau / partition / rewriting systems / symmetric fncts)...
- Some other results:
  - The hook-length formulas: 3 weeks (joint work w. C. Paulin)
  - Cycle decomposition: 2 months (T. Benjamin, undergrad)
  - Basic theory of symmetric functions: 3 months
  - Character theory of the symmetric groups: 1 month
- This is transforming math into a video-game

Fun if you like it, but very addictive !

Want to have a closer look ?

The code:

- `https://github.com/hivert/Coq-Combi`

The documentation:

- `http://hivert.github.io/Coq-Combi`