Demazure operators and Lean

Óscar Álvarez Sánchez

Born 25th October 2000 in Boadilla del Monte, Spain

25th October 2024

Master's Thesis Mathematics

Advisor: Prof. Dr. Catharina Stroppel

Second Advisor: Dr. Johannes Flake
Mathematisches Institut

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

1 Introduction

Lean is a computer formalization software developed by Leonardo de Moura in 2013 [Lea]. It enables the formalization of mathematical proofs into machine-verifiable code, ensuring that every assumption is explicitly stated and eliminating the possibility of unnoticed errors. Lean stands out from other proof assistants by incorporating modern features, which make it more accessible and easier to use.

A key component of Lean's ecosystem is *mathlib* [Mata], its standard mathematical library. *Mathlib* is an open-source project that houses a wide variety of results from different areas of mathematics, and its collaborative nature allows contributions from anyone in the community. Some of the work presented in this thesis on Coxeter groups is intended to be included in future versions of *mathlib*.

In this thesis, we leverage Lean to formalize results concerning Demazure operators. Introduced by Demazure in 1974[Dem74], these operators have become essential tools in representation theory, particularly in the study of highest weight modules and Schubert calculus. They offer a combinatorial perspective for understanding the structures of these mathematical objects.

The primary objective of this work is to provide a rigorous, machine-verifiable formalization of Demazure operators and their key properties within a generalized setting. This formalization is designed to serve as a foundation that future work can build upon.

We begin in Section 2 by introducing the necessary mathematical concepts and the specific Lean functions that will be utilized throughout the thesis.

In Section 3, we present an initial definition of Demazure operators, which, while functional, proves cumbersome for certain combinatorial arguments. We then introduce an alternative, more practical definition and demonstrate its equivalence to the initial one. This leads naturally to a generalization of the definition. To support this work, we develop the theory of Coxeter groups in Section 4, formalizing results not only essential for our purposes but also applicable to various other domains in mathematics. The formalization of Matsumoto's theorem in a specific case, using relatively elementary methods, is the core result of this thesis.

In Section 5, we apply our Coxeter group theory to the symmetric group, S_n, by constructing a sketch of the proof that it satisfies the conditions of a Coxeter group. We then use this to define Demazure operators on S_n and prove their well-definedness.

Throughout this document, Lean code will be highlighted with a gray background to distinguish it from the surrounding mathematical text. We will explicitly clarify how the Lean code corresponds to the mathematical concepts being formalized. While we provide an overview of the libraries used and discuss some advanced features of Lean, readers are encouraged to familiarize themselves with the basics of Lean beforehand. We recommend the introductory text *Mathematics in Lean*[Mil] as a starting point.

The full code can be accessed via the attached CD or the GitHub repository [Ála]. It includes documentation for all the results discussed here, along with additional formalizations not covered in this thesis. An online version of the documentation is available at [Álb], or it can be built from the provided source code. Instructions on how to use the Lean code

and generate the documentation can be found in the repository's *README.md*.

Any results already included in *mathlib* will be clearly identified with a corresponding header in the Lean code, like this:

mathlib

```
theorem one_add_one_eq_two {α : Type u_1} [AddMonoidWithOne α] :
    1 + 1 = 2
```

They can be checked in the online mathlib documentation[Matb]. All other results presented here are original contributions, either derived independently or following cited references. The only non-formalized assumption we make is that S_n is a Coxeter group, which is fully explained in Section 5.

2 Preliminaries

We will introduce mathematical structures as needed throughout, but it's helpful to cover some fundamental concepts up front.

2.1 Groups

A **monoid** is a set M equipped with a binary operation

$$\cdot : M \times M \to M$$

that satisfies the following properties:

- **Associativity**: For all $a, b, c \in M$, we have
$$(a \cdot b) \cdot c = a \cdot (b \cdot c).$$
- **Identity element**: There exists an element $e \in M$ such that for all $a \in M$,
$$e \cdot a = a \cdot e = a.$$

We also consider for every $n \in \mathbb{N}$ the exponentiation as

$$m^n = \underbrace{m \cdot m \cdots m}_{\text{n times}}$$

This can be defined inductively by making $m^0 = e$ and $m^{n+1} = m \cdot m^n$.

mathlib

```
class Monoid(M : Type u) extends Semigroup , One : Type u
    mul : M → M → M
    mul_assoc : ∀ (a b c : M), a * b * c = a * (b * c)

    one : M
```

```
    one_mul : ∀ (a : M), 1 * a = a
    mul_one : ∀ (a : M), a * 1 = a

    npow : ℕ → M → M
    npow_zero : ∀ (x : M), Monoid.npow 0 x = 1
    npow_succ : ∀ (n : ℕ) (x : M), Monoid.npow (n + 1) x = Monoid.npow n x * x
```

Definition 2.1. A **group** is a monoid G in which every element has an inverse. Specifically, for each $a \in G$, there exists an element $a^{-1} \in G$ such that

$$a \cdot a^{-1} = a^{-1} \cdot a = e,$$

where e is the identity element.

This enables us to define division as $g/h = g \cdot h^{-1}$ for every $g, h \in G$, and extend the exponentiation to the integers with $g^{-n} = (g^{-1})^n = (g^n)^{-1}$

mathlib

```
class Group(G : Type u) extends DivInvMonoid :
Type u
    ... (Properties of a Monoid)

    inv : G → G
    div : G → G → G
    div_eq_mul_inv : ∀ (a b : G), a / b = a * b⁻¹
    zpow : ℤ → G → G
    zpow_zero' : ∀ (a : G), DivInvMonoid.zpow 0 a = 1
    zpow_succ' : ∀ (n : ℕ) (a : G), DivInvMonoid.zpow (↑n.succ) a =
    DivInvMonoid.zpow (↑n) a * a
    zpow_neg' : ∀ (n : ℕ) (a : G), DivInvMonoid.zpow (Int.negSucc n) a =
    (DivInvMonoid.zpow (↑n.succ) a)⁻¹
    inv_mul_cancel : ∀ (a : G), a⁻¹ * a = 1
```

Notice that we define these mathematical objects with the `class` keyword. Let's show how this mechanism works with an example.

Definition 2.2. A **permutation** of a set X is a bijection from X to X. The set of all permutations of X forms a group under function composition.

In mathlib, permutations of a type α are defined as:

mathlib

```
abbrev Equiv.Perm (α : Sort*) := Equiv α α
```

It's easy to prove that permutations form a group under composition. To formalise this, we show that `Equiv.Perm` is an instance of `Group`, by proving that it satisfies all of its properties. In our case this is already done in mathlib.

mathlib

```
instance Equiv.Perm.permGroup{α : Type u} :
    Group (Equiv.Perm α)
```

This has several benefits over alternative ways of formalizing that this object is a group. If we declared `Group` as a structure, every group would have to be constructed as such, and we could only use its properties and that of sub-structures like monoids.

On the other hand, a single object can be an instance of multiple classes that have nothing to do with each other. For example, we state that permutations can be seen as embeddings in a different instance statement.

mathlib

```
instance Equiv.Perm.coeEmbedding{α : Sort u} :
    Coe (Equiv.Perm α) (α  α)
```

The reason we use instances instead of usual definitions is that it makes the process of finding an object's properties automatic.

For example, let's say that we stated that permutations are a group with a definition.

```
def permGroup{α : Type u} :
    Group (Equiv.Perm α)
```

And we want to build on top of the group structure of permutations. Let's say that we define the operation of squaring an element, which can be defined in general for groups.

```
def square {α : Type} (isGroup : Group α) (s : α) : α :=
  isGroup.mul s s
```

Then to use it in our example, we need to pass the fact that `Equiv.Perm` is a group to the `square` definition explicitly.

```
def p : Equiv.Perm α := ...
def p_squared : Equiv.Perm α := square permGroup p
```

This gets quickly cumbersome, especially when there is a lot of structure, say that we need to use that `Equiv.Perm` is a group but also that it's non-empty, can be seen as a collection of embeddings and has a default element (the identity).

But when declaring it as an instance, we can use it directly on theorems that require that class instance with square brackets. With our previous example:

```
def square {α : Type} [Group α] (s : α) : α :=
  s * s

def p_squared := square p
```

As you can see, we don't have to explicitly indicate that `Equiv.Perm` is a group to use `square`. Lean automatically searches for instances of `Group` and detects that `Equiv.Perm` is one of them. This process is called **type class inference** and it's one of the most useful and interesting capabilities of Lean.

The built-in classes also have the benefit of providing custom notation. For example, after declaring the `Group` instance for `Equiv.Perm`, we can multiply two elements `p` and `q` of this structure with `p * q`, as you saw in the definition of `square`.

2.2 Symmetric group

The specific set X is not important, since any other set in bijection with it leads to the same group. Therefore, when X is finite we only care about its cardinality. This motivates the following definitions:

Definition 2.3. Let $n \in \mathbb{N}$. We define the finite set of n elements as

$$[n] := [n]$$

mathlib

```
structure Fin (n : Nat) where
  /-- Creates a `Fin n` from `i : Nat` and a proof that `i < n`. -/
  mk ::
  /-- If `i : Fin n`, then `i.val : ℕ` is the described number. It can also be
  written as `i.1` or just `i` when the target type is known. -/
  val  : Nat
  /-- If `i : Fin n`, then `i.2` is a proof that `i.1 < n`. -/
  isLt : LT.lt val n
```

So can create elements of $[n]$ with an element $i \in \mathbb{N}$ and a proof `h` that $i < n$ as $\langle$ `i, h` $\rangle$.

A lot of times, we will want to go from $[n]$ to $[n+1]$. Two really useful shorthands are `i.castSucc`, which sends $i \in [n]$ to $i \in [n+1]$ and `i.succ`, which sends $i \in [n]$ to $i+1 \in [n+1]$.

Definition 2.4. The **symmetric group** S_n is the group of all permutations of the finite set $[n]$, i.e.,

$$S_n := \{f : [n] \to [n] \mid f \text{ is bijective}\}.$$

```
abbrev S (n : ℕ) := Equiv.Perm (Fin n)
```

Definition 2.5. Let $n \in \mathbb{N}$. We say that $\sigma := (i_0, i_1, \ldots, i_{p-1}) \in S_n$ for some $p < n$ and $i_k \in [n] \quad \forall 0 < k < p$ is a cycle of length k if it sends

$$(i \in [n]) \mapsto \begin{cases} i_{j+1} & \text{if } i = i_j \text{ with } j < p-1 \\ i_0 & \text{if } i = i_{p-1} \\ i & \text{otherwise} \end{cases}$$

We are specially interested in cycles of length two:

Definition 2.6. Let $n \in \mathbb{N}$. Then **transpositions** or **swaps** are the cycles of order/length two, that is, the elements

$$s_{i,j} := (i, j) \quad \forall\ 0 \leq i, j < n$$

mathlib

```
def Equiv.swap [DecidableEq α] (a : α) (b : α) :
    Equiv.Perm α
```

Remark 2.1. *For simplicity, we sometimes refer to the transposition s_i for $0 \leq i < n-1$ as just s_i.*

Also, it's a well known fact that this family of transpositions $\{s_i \ : \ 0 \leq i < n-1\}$ generates S_n. To formalise what this means, let's explain what subgroups are.

Definition 2.7. A **subgroup** of a group G is a subset $H \subseteq G$ that is closed under the group operation and contains the inverse of each of its elements and the identity.

mathlib

```
structure Subgroup(G : Type u_5) [Group G] extends Submonoid :
Type u_5
carrier : Set G
mul_mem' : ∀ {a b : G}, a ∈ self.carrier → b ∈ self.carrier → a * b ∈
    self.carrier
one_mem' : 1 ∈ self.carrier
inv_mem' : ∀ {x : G}, x ∈ self.carrier → x⁻¹ ∈ self.carrier
```

But to create a subgroup from scratch we have to provide the underlying set (`carrier`). To talk about the subgroup **generated** by a set, we proceed as follows:

Definition 2.8. Let G be a group and $k \subseteq G$ a subset. The subgroup generated by k, denoted by $\langle k \rangle$, is defined as the smallest subgroup of G containing k, that is,

$$\langle k \rangle = \bigcap \{K \subseteq G \mid K \text{ is a subgroup of } G \text{ and } k \subseteq K\}.$$

mathlib

```
def Subgroup.closure (k : Set G) : Subgroup G :=
  sInf { K | k ⊆ K }
```

The fact that s_i generate S_n is formalized in mathlib as:

$$\langle \{s_i \ : \ 0 \leq i < n\} \rangle = S_{n+1}$$

mathlib

```
theorem mclosure_swap_castSucc_succ (n : ℕ) :
    Submonoid.closure (Set.range fun i : Fin n ↦ swap i.castSucc i.succ) =
    Equiv.Perm (Fin n)
```

2.3 Polynomials

An initial distinction to be made is between univariate polynomials and multivariate polynomials, which are found in the `Polynomial` package and `MvPolynomial` package, correspondingly.

Let's start with `Polynomial` since it's way simpler. We define a uni-variate polynomial (semi-)ring as `Polynomial R`, where `R` is a (semi-)ring. Semirings have the same properties as rings, except they are not guaranteed to have an inverse w.r.t. addition.

mathlib

```
structure Polynomial(R : Type u_1) [Semiring R] : Type u_1
```

We will just consider the ring case from now on. This corresponds to the usual polynomial ring $R[x]$. But as we said before, we can consider polynomials as instances of multiple classes. This corresponds with the fact that we can see polynomials as groups, rings, modules, and when R is a field, R-vector spaces and $R-algebra$, among others. We will mostly consider it as a $R-$module but keep in mind the rest of possible structures.

mathlib

```
instance Polynomial.module{R : Type u} [Semiring R]
{S : Type u_1} [Semiring S] [Module S R] :
    Module S (Polynomial R)
```

The simplest polynomials will be the constant ones, that is, elements of R. Lean is strongly typed so we can't just consider an element of R as a polynomial. We have to explicitly convert it to one with the function `Polynomial.C`. It is a ring homomorphism from R to $R[x]$. For example, the polynomial $3 \in \mathbb{Z}[x]$ could be constructed in Lean as `Polynomial.C 3`. You may have noticed that we haven't specified that the coordinate ring is $\mathbb{Z}$ to Lean. That's because Lean can figure it by itself given that $3 \in \mathbb{Z}$. If we look at the definition:

mathlib

```
    def Polynomial.C{R : Type u} [Semiring R] :
        R →+* Polynomial R
```

The argument R is inside curly brackets. That means it's an implicit argument. Since this function's domain is `R`, we can figure out what ring is R from the type of the argument.

You can see another example of type class inference with the argument `[Semiring R]`. It means that Lean must know from some previous theorem that the type `R` has all the properties of a semi-ring (similar to a ring but lacks an additive inverse). In our example, we use the ring $\mathbb{Z}$ which is constructed in mathlib, so we don't even have to prove it beforehand.

We can access the polynomial variable using the definition `Polynomial.X`. The type isn't explicitly mentioned here, but Lean can deduce it—for instance, when we multiply it by a constant in `R`, the expected type can guide the inference. For the sake of simplicity, we'll omit some implicit arguments moving forward.

To save time, we can use the following construction:

mathlib

```
    def Polynomial.monomial (n : ℕ) :
        R →[R] Polynomial R
```

This enables one to get the monomial ax^n as `Polynomial.monomial n a`.

In uni-variate polynomials, evaluation at an element of the ring is simple enough. We have the following definition:

mathlib

```
    def Polynomial.eval :
        R → Polynomial R → R
```

In Lean the functions are grouped from the right, so `R → Polynomial R → R` means `R → (Polynomial R → R)`. That is, `Polynomial.eval` takes $a \in R$ and returns the evaluation function $ev_a : R[x] \to R$.

So for example, if we have the polynomial $p = 3x^2 - x + 1 \in \mathbb{Z}[x]$ and we want to get $p(2)$, we do it with `Polynomial.eval 2 p`, but we can also prove facts about the evaluation function `Polynomial.eval 2` as a map.

This is enough for a lot of proofs but what if we want to evaluate a polynomial at an expression? For example, if we want to compute $p(x^2)$. Then this definition is not enough, and we have to use $\texttt{eval}_2$

mathlib

```
def Polynomial.eval₂ (f : R →+* S) (a : S) (p : Polynomial R) : S
```

As before, $p \in R[x]$ is the polynomial we want to evaluate at some expression. But now instead of having some $a \in R$ we have a parameter a in some other ring S. This lets us evaluate a polynomial outside of the coefficient ring. But we have to define an embedding of R into S to be able to perform this operation, and this is exactly what `f` is. For example, to compute $p(x^2)$ we would have to do:

mathlib

```
\lstinline{Polynomial.eval₂ (Polynomial.C) (Polynomial.monomial 2 1) p}
```

The function `f` is in this case `Polynomial.C`, which embeds the coefficient ring R in $R[x]$. That way, we can supply the element `Polynomial.monomial 2 1`, which corresponds to x^2, and evaluate p at it.

The previous configuration corresponds to the function

$$ev_a : R[x] \to R[x], p \mapsto p(a), \text{where } a \in R[x]$$

In the case where `S` (here denoted `A`) is an $R-$algebra, there is a simpler definition that does the same, namely:

mathlib

```
def Polynomial.aeval [Algebra R A] (x : A) :
    Polynomial R →ₐ[R] A
```

In this case the function `f` is left implicit since it's just the embedding $R \hookrightarrow A$.

Things get more complicated for multivariate polynomials.

mathlib

```
def MvPolynomial(σ : Type u_1) (R : Type u_2) [CommSemiring R] :
    Type (max u_1 u_2)
```

The way to create the ring is similar, just including the amount of variables n as a parameter, with `MvPolynomial (Fin n) R`.We put `Fin n` instead of `n` because we can in fact work with any index set, not just natural indices, but in this case it would correspond to

$P_n := R[x_0, x_1, \ldots, x_{n-1}]$. To evaluate multi-variate polynomials, we don't do so inductively. Instead, we have:

mathlib

```
def MvPolynomial.eval(f : σ → R) :
    MvPolynomial σ R →+* R
```

Basically, instead of passing an element of the coefficient ring like in the uni-variate case, we pass a function that contains the value at every index. As before, it returns a function from the polynomial ring to the coefficient ring.

For example, to get the expression $p(-1, 5, 3)$ for some $p \in \mathbb{Z}[x_0, x_1, x_2]$ we would use

mathlib

```
MvPolynomial.eval f p
```

Where `f` : $\mathrm{Fin}\, 3 \to \mathbb{Z}$ and $f(0) = -1,\ f(1) = 5,\ f(2) = 3$

The evaluation outside of the coefficient ring is also similar to the univariate case, with:

mathlib

```
def MvPolynomial.eval₂ (f : R →+* S₁) (g : σ → S₁) (p : MvPolynomial σ R) :
S₁
```

Again, we have a function `f` that "augments" the coefficients, and now the valuation `g` can take values in the bigger ring. In mathematical language, and with $f = R \hookrightarrow S$, where we assume f to be the natural embedding, this is the function

$$\widetilde{ev}_g^S : P_n \to P_n, \quad p \mapsto p(g(0), g(1), \ldots, g(n-1))$$

$$\text{for some } g : \{0, 1, \ldots, n-1\} \to P_n$$

And as before, with this use case it suffices with the simplified version

mathlib

```
def MvPolynomial.aeval [Algebra R S₁] (f : σ → S₁) :
    MvPolynomial σ R →ₐ[R] S₁
```

Where we don't have to supply `g`. However, for most of this thesis, the theorems that we use are dependant on `eval₂` (since it's the most general), so we will focus the most on it.

3 Demazure Operators

3.1 Definition

We can consider the action of S_n on the ring of polynomials by swapping variables. For simplicity sake, we are going to assume we are working over the complex numbers, that is,

$R = \mathbb{C}$.

Definition 3.1. Let $\sigma \in S_n$ be a permutation and $p \in P_n$ be a (multivariate) polynomial. Then, σ induces a ring isomorphism (over R)

$$\overline{\sigma} : P_n \longrightarrow P_n \text{ sending } p(x_0, x_1, \dots x_{n-1}) \mapsto p(x_{\sigma(0)}, x_{\sigma(1)}, \dots x_{\sigma(n-1)})$$

For example, if we take $\sigma = (0, 2, 3) \in S_4$, then

$$\overline{\sigma}(x_0^2 - 2x_1x_2 + x_0^3x_3^2x_1) = x_2^2 - 2x_1x_3 + x_2^3x_0^2x_1$$

Proving that it's a ring isomorphism is not difficult. It's already implemented in mathlib as:

mathlib

```
def MvPolynomial.renameEquiv (R : Type u_4) [CommSemiring R] (f : σ ≃ τ) :
    MvPolynomial σ R ≃ₐ[R] MvPolynomial τ R
```

Again, we are specially interested in the transposition case, so we define swaps for that specific case to simplify proofs in the future.

Remark 3.1 (Variable swaps)**.** *Given some indices $0 \leq i, j < n$, then the induced $\mathbb{C}-$algebra isomorphism $\overline{s_{i,j}} : P_n \to P_n$ sends:*

$$x_k \mapsto x_{s_{i,j}(k)} = \begin{cases} x_j & \text{if } k = i \\ x_i & \text{if } k = j \\ x_k & \text{otherwise} \end{cases}$$

That is, it swaps the variables x_i and x_j of a polynomial.

```
def SwapVariablesFun (i j : Fin n) (p : MvPolynomial (Fin n) ℂ) :(MvPolynomial
    (Fin n) ℂ) := (renameEquiv ℂ (Transposition i j)) p
```

In Lean, one way to construct certain objects (like ring isomorphisms in this case) is to first give the underlying function, then prove the properties of this type of object, and finally put everything together into the final definition.

First, we need to find its inverse. We do this in Lean by supplying the function that is going to be the inverse (itself, since we are working with a transposition) and two proofs, one for it being the right inverse and another for being the left inverse.

Then, for the homomorphism part, we prove that it respects multiplication and addition, and that it is commutative, all of which are quite trivial proofs that can be found in the code repository.

```
def SwapVariables (i : Fin n) (j : Fin n) :
    AlgEquiv ℂ (MvPolynomial (Fin n) ℂ) (MvPolynomial (Fin n) ℂ) where
  toFun := SwapVariablesFun i j
  invFun := SwapVariablesFun i j
  left_inv := by
    simp[Function.LeftInverse]
```

```
  right_inv := by
    simp[Function.RightInverse]
    intro p
    exact swap_variables_order_two

  map_mul' := swap_variables_mul
  map_add' := swap_variables_add
  commutes' := swap_variables_commutes
```

To give an example of one of them, here we show that applying $\overline{s_{i,j}}$ twice doesn't change a polynomial. We apply the internal lemma `Equiv.eq_symm_comp` and rewrite every layer of abstraction.

```
@[simp]
lemma swap_variables_order_two {i j : Fin n} {p : MvPolynomial (Fin n) ℂ} :
  SwapVariablesFun i j (SwapVariablesFun i j p) = p := by
  simp[SwapVariablesFun, Equiv.swap_mul_self]
  have : (Equiv.swap i j) ∘ (Equiv.swap i j) = Equiv.refl _ := by
    exact (Equiv.eq_symm_comp (Equiv.swap i j) (Equiv.swap i j) (Equiv.refl (Fin
    n))).mp rfl
  rw[this]
  apply MvPolynomial.rename_id
```

Remark 3.2. *`SwapVariables i j` boils down to the expression*

```
MvPolynomial.eval2 (MvPolynomial.C) (Equiv.swap i j) p
```

This is the term that will appear in proofs if we simplify every intermediate step. However, we will try to use the lemmas we have proven about `SwapVariables` to make proofs shorter.

Finally, we are ready to give the initial definition of Demazure operators.

Definition 3.2 (Demazure operators)**.** Given some index $i \in [n]$, the corresponding Demazure operator $\partial_i : P_{n+1} \to P_{n+1}$ maps

$$p \mapsto \frac{p - s_i(p)}{x_i - x_{i+1}}$$

The result stays in the polynomial ring because $x_i - x_{i+1}$ divides $p - s_i(p)$ as we will prove shortly.

Or in Lean,

```
def DemazureNumerator (i : Fin n) (p : MvPolynomial (Fin (n + 1)) ℂ) : Polynomial
    (MvPolynomial (Fin n) ℂ)  :=
  let i' : Fin (n + 1) := Fin.castSucc i
  let i'_plus_1 : Fin (n + 1) := Fin.succ i

  let numerator := p - SwapVariables i' i'_plus_1 p
  let numerator_X_i_at_start : MvPolynomial (Fin (n + 1)) ℂ := SwapVariables i' 0
    numerator
  (finSuccEquiv ℂ n) numerator_X_i_at_start

def DemazureDenominator (i : Fin n) : Polynomial (MvPolynomial (Fin n) ℂ)  :=
```

```
  let X_i : MvPolynomial (Fin n) ℂ := MvPolynomial.X i
  let denominator_X : Polynomial (MvPolynomial (Fin n) ℂ) := (Polynomial.X -
    Polynomial.C X_i)

  denominator_X

def DemazureFun (i : Fin n) (p : MvPolynomial (Fin (n + 1)) ℂ) : MvPolynomial
    (Fin (n + 1)) ℂ  :=
  let numerator := DemazureNumerator i p
  let denominator := DemazureDenominator i

  let division := numerator.divByMonic denominator
  let division_mv : MvPolynomial (Fin (n + 1)) ℂ := (AlgEquiv.symm (finSuccEquiv ℂ
     n)) division

  let i' : Fin (n + 1) := Fin.castSucc i

  SwapVariables i' 0 division_mv
```

As you can see this is a quite complicated definition, specially compared to the mathematical one. The main problem with this definition is the division. Division of multivariate polynomials is not implemented in the current version of mathlib. However, notice that we are dividing by a very simple polynomial. The trick to make this work is considering the polynomials of $n+1$ variables as uni-variate polynomials with coefficients in the ring of polynomials in n variables. In Lean, this is done by "picking" x_0 as the new x (the uni-variate variable). Basically, using

$$\Phi : \mathbb{C}[x_0, x_1, \ldots, x_n] \cong \mathbb{C}[y_0, \ldots, y_{n-1}][y]$$

$$x_i \mapsto \begin{cases} y & \text{if } i = 0 \\ y_{i-1} & \text{if } i > 0 \end{cases}$$

The definition is split between the numerator and the denominator, with the final definition combining them. The only parameter is i, which indicates the variables we swap. In the Lean code, we work over polynomials in $n+1$ variables. This makes it easier to "extract" one of them like we said before.

For the numerator, we first compute $p - s_i(p)$ and store it in `numerator`. Then, we swap the variables x_i and x_0 to prepare the polynomial for extracting x_i as x. And finally, we do so with the function `finSuccEquiv`.

This function is just a wrapper around the following expression:

```
MvPolynomial.eval₂
    (Polynomial.C.comp MvPolynomial.C)
    (fun i ↦ Fin.cases
        Polynomial.X
        (fun k ↦ Polynomial.C (MvPolynomial.X k))
        i
    )
    p
```

Which maps `p` to a polynomial in $\mathbb{C}[y_0, \ldots, y_{n-1}][y]$ via the function `Fin.cases Polynomial.X (fun k ↦ Polynomial.C (MvPolynomial.X k)) i) p`. It sends 0 to the first item (`Polynomial.X`) and then proceeds by induction, so it sends `k + 1` to `Polynomial.C (MvPolynomial.X k)`. This is the function that will actually appear in the proofs.

In mathematical terms, let $Q_n := \mathbb{C}[y_0, \ldots, y_{n-1}]$, and

$$g : \mathrm{Fin}(n+1) \to Q_n[y]$$
$$j \mapsto \begin{cases} y & \text{if } j = 0 \\ y_{j-1} & \text{if } j > 0 \end{cases}$$

Then, `finSuccEquiv` corresponds to $\widetilde{ev}_g^{Q_n[y]}$

To replace x_i with y in the definition of the Demazure operator, first we swap the variables x_i and x_0 and then we apply the aforementioned function;

$$\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))$$

You may have noticed that with a similar function we could skip the swap between x_i and x_0 and send directly x_i to y. However, it's difficult to define functions by cases that are not 0 and the rest (we don't have a lot of lemmas like with `Fin.cases`), so we ended up choosing this approach because it translated into simpler proofs.

We apply the same transformation to the denominator, which becomes

$$\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(x_i - x_{i+1}) = \widetilde{ev}_g^{Q_n[y]}(x_0 - x_{i+1}) = y - y_i$$

And therefore we can define it directly like that.

Then to define the Demazure operator (`DemazureFun` in Lean), we take the numerator and divide it by the (monic) numerator using `divByMonic`. Here it's important to know that `divByMonic` just returns the quotient of the division, regardless of whether it's exact (with zero remainder) or not. So to prove this definition coincides with the mathematical one, we will show this division is in fact exact afterwards.

After computing the quotient of these uni-variate polynomials, we end up with another polynomial in $Q_n[y]$. But of course, we have defined the Demazure operators as functions of P_{n+1}, so now we undo the process of distinguishing x_i into the uni-variate polynomial variable y. First we undo the effect of turning x_0 to x by applying the inverse of `finSuccEquiv` and finally swap x_0 and x_i to get them to their original positions.

So finally, the Demazure operator becomes:

$$s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right)$$

Which can be shown to be equivalent to the original definition:

$$
\begin{aligned}
&s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right) = \\
&= (\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(x_i - x_{i+1})} \right) \\
&= (\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i})^{-1} \circ (\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}) \left(\frac{p - s_i(p)}{x_i - x_{i+1}} \right) \\
&= \frac{p - s_i(p)}{x_i - x_{i+1}} \\
&= \partial_i(p)
\end{aligned}
$$

3.2 Basic results

Proposition 3.3. *For any polynomial $p \in P_{n+1}$, $1 \leq i \leq n$ the polynomial $x_i - x_{i+1}$ divides $p - s_i(p)$*

Proof. We use the alternative definition we just constructed,

$$
\partial_i(p) = s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right)
$$

Since the variable swaps and evaluation are isomorphism, it suffices to prove that the division $\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i}$ is exact, that is, $y - y_i$ divides $\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))$.

Notice that the denominator is a monic polynomial in y, so by the remainder theorem, the result is equivalent to proving that:

$$
ev_{y_i} \circ \widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p)) = 0
$$

But here we can use the following theorem in Lean. Its statement is an equality, which lets us replace one side with the other in any Lean expression (commonly the target of our proof, but also our hypotheses or other propositions).

mathlib

```
theorem MvPolynomial.polynomial_eval_eval₂ (f : R →+* Polynomial S)
    (g : σ → Polynomial S) (p : MvPolynomial σ R)
:
Polynomial.eval x (MvPolynomial.eval₂ f g p) =
MvPolynomial.eval₂ ((Polynomial.evalRingHom x).comp f) (fun (s : σ) =>
    Polynomial.eval x (g s)) p
```

Or in mathematical language,

Theorem 3.4. *Let $p \in P_{n+1}$, $a \in Q_n$ and $g : Fin_n \to Q_n$. Then,*

$$
ev_a \circ \widetilde{ev}_g^{Q_n[y]}(p) = \widetilde{ev}_{ev_a \circ g}^{Q_n}(p) \in Q_n
$$

Let $g' := ev_{y_i} \circ g : Fin(n+1) \to \mathbb{C}[y_0, \ldots, y_{n-1}]$, which corresponds to:

$$j \mapsto \begin{cases} y_i & \text{if } j = 0 \\ y_{j-1} & \text{if } j > 0 \end{cases}$$

We then need to prove:

$$\widetilde{ev}_{g'}^{Q_n} \circ s_{0,i}(p - s_i(p)) = 0$$

Or equivalently;

$$\widetilde{ev}_{g'}^{Q_n} \circ s_{0,i}(p) = \widetilde{ev}_{g'}^{Q_n} \circ s_{0,i} \circ s_i(p) \tag{1}$$

First of all, we combine all the compositions inside the function of `eval₂` with the following theorem:

mathlib

```
theorem MvPolynomial.eval₂_rename (f : R →+* S) (k : σ → τ)
    (g : τ → S) (p : MvPolynomial σ R)
:
MvPolynomial.eval₂ f g ((MvPolynomial.rename k) p) =
    MvPolynomial.eval₂ f (g ∘ k) p
```

When we take `g = g'` and `k = Transposition i j`, then:

```
    (MvPolynomial.rename k) = SwapVariables i j
```

So the theorem translates to:

Theorem 3.5. *For any* $0 \leq i, j \leq n$,

$$\widetilde{ev}_{g'}^{Q_n} \circ s_{i,j}(p) = \widetilde{ev}_{g' \circ \tau_{i,j}}^{Q_n}(p)$$

Therefore, (1) is equivalent to:

$$\widetilde{ev}_{g' \circ \tau_{0,i}}^{Q_n}(p) = \widetilde{ev}_{g' \circ \tau_{0,i} \circ \tau_{i,i+1}}^{Q_n}(p)$$

Let $g_1 = g' \circ \tau_{0,i}$ and $g_2 = g' \circ \tau_{0,i} \circ \tau_{i,i+1}$. The proof now just boils down to the evaluation of a polynomial being equal with these two functions. Lean provides a useful theorem to prove this fact:

mathlib

```
theorem MvPolynomial.eval₂_congr {p : MvPolynomial σ R} (f : R →+* S₁) (g₁ : σ →
    S₁) (g₂ : σ → S₁)
    (h : ∀ {j : σ} {c : σ →₀ ℕ},
        j ∈ c.support → MvPolynomial.coeff c p ≠ 0 → g₁ j = g₂ j
    )
:
MvPolynomial.eval₂ f g₁ p = MvPolynomial.eval₂ f g₂ p
```

So we just need to prove

```
h : ∀ {j : σ} {c : σ →₀ ℕ},
    j ∈ c.support → MvPolynomial.coeff c p ≠ 0 → g₁ j = g₂ j
```

That is, that for every index j and every monomial with exponents given by the function c, if j is one of the non-null exponents and the monomial has a non-zero coefficient in p, then the evaluation of j at g_1 and g_2 must coincide.

For our case, we don't actually use all these hypotheses since it's easy to see that $g_1 = g_2$ as it is. In the following tables we compute the result of applying $g_1 = g' \circ \tau_{0,1}$ and $g_2 = g' \circ \tau_{0,1} \circ \tau_{i,i+1}$ step by step, at every element $0 \leq j \leq n$, separated by cases. Notice that the final result (the last column) is equal in both tables, completing the proof.

j	$\tau_{0,1}(j)$	$g' \circ \tau_{0,1}(j) = g_1(j)$
0	i	y_{i-1}
i	0	y_i
$i+1$	$i+1$	y_i
$j \neq 0, i, i+1$	j	y_{j-1}

j	$\tau_{i,i+1}(j)$	$\tau_{0,1} \circ \tau_{i,i+1}(j)$	$g' \circ \tau_{0,1} \circ \tau_{i,i+1}(j) = g_2(j)$
0	0	i	y_{i-1}
i	$i+1$	$i+1$	y_i
$i+1$	i	0	y_i
$j \neq 0, i, i+1$	j	j	y_{j-1}

□

This proves that the Demazure operators are well defined as functions from P_{n+1} to itself. However, we also required them to be $\mathbb{C}-$linear. As shown before, this is done in Lean by creating a new object `Demazure` and supplying all the properties of a linear operator with the `where` clause.

```
def Demazure (i : Fin n) : LinearMap (RingHom.id ℂ)
(MvPolynomial (Fin (n + 1)) ℂ) (MvPolynomial (Fin (n + 1)) ℂ) where
  toFun := DemazureFun i
  map_add' := demazure_map_add i
  map_smul' := demazure_map_smul i
```

The proofs that Demazure operators are additive and respect scalar multiplication, while trivial mathematically, require some work in Lean due to the complexity of the definition. For example, the proof of `map_smul'` (scalar multiplication) is the following:

```
lemma demazure_map_smul (i : Fin n) :
∀ (r : ℂ) (p : MvPolynomial (Fin (n + 1)) ℂ),
DemazureFun i (r · p) = r · DemazureFun i p := by
  intro r p
  simp[DemazureFun, SwapVariables, MvPolynomial.smul_eq_C_mul]
  nth_rewrite 2 [← swap_variables_commutes]
  rw[← swap_variables_mul]
  apply congr_arg
  nth_rewrite 2 [← MvPolynomial.finSuccEquiv_comp_C_eq_C]
  simp[RingHom.comp]
  rw[← AlgEquiv.map_mul]
  apply congr_arg
```

```
  apply (poly_mul_cancel (demazure_denominator_ne_zero i)).mpr
  rw[← mul_assoc]
  rw [mul_comm (DemazureDenominator i) (Polynomial.C (C r))]
  simp[demazure_division_exact']
  rw[mul_assoc]
  rw[demazure_division_exact' i p]
  exact demazure_numerator_C_mul i p r
```

We start introducing the arbitrary scalar `r` and the polynomial `p` with the `intro` clause. Then, we simplify the definition of the Demazure operator and the variable swapping homomorphism. Then we need to convert the scalar multiplication to C-mul. They are two different ways of conceptualizing multiplication of a scalar with a polynomial.

Scalar multiplication is more general, basically it's defined for every vector space and corresponds with the mathematical definition. Polynomial rings are vector spaces so they come equipped with this operation. On the other hand, C-mul is exclusive to polynomials, and uses the (Mv)Polynomial.C function we talked about before. It converts the scalar to a polynomial and then uses polynomial multiplication.

Of course this two operations are equivalent, as this theorem states:

mathlib

```
theorem MvPolynomial.smul_eq_C_mul (p : MvPolynomial σ R) (a : R) :
    a · p = MvPolynomial.C a * p
```

So after simplification, we have to prove that:

$$r \cdot \partial_i(p) = r \cdot s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right) =$$
$$s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(r \cdot p - s_i(r \cdot p))}{y - y_i} \right) = \partial_i(r \cdot p)$$

Afterwards, with the lines `nth_rewrite 2 [← swap_variables_commutes]` and `rw[← swap_variables_mul]` we use that scalar multiplication commutes with the variable swap homomorphism to **rewrite** the goal (what we want to prove) with this property. Now our goal becomes:

$$s_{0,i} \left(r \cdot (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right) \right) =$$
$$s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(r \cdot p - s_i(r \cdot p))}{y - y_i} \right)$$

In both sides there's an external $s_{0,i}$. If we prove that the inside part is equal, then the goal is implied since we are just applying a function. So it suffices to prove

$$r \cdot (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right) =$$
$$(\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(r \cdot p - s_i(r \cdot p))}{y - y_i} \right)$$

This is exactly what the tactic `apply congr_arg` does. We proceed iteratively in this way, slowly moving towards the inside of the expression until the proof is complete. Most proofs work like this, we prove theorems by "layers", from the outside to the inside. This usually works all right since most of the building blocks are isomorphisms and as such have the properties we require, but it involves a lot of work, even for simple proofs like this one.

Remark 3.6. *The Demazure operators are not multiplicative. For example,*

$$\partial_i(x_i \cdot 1) = \frac{x_i - x_{i+1}}{x_i - x_{i+1}} = 1 \neq 0 = x_{i+1} \cdot 0 = \partial_i(x_i) \cdot \partial_i(1)$$

In Lean,

```
lemma demazure_not_multiplicative :
  ∀ (i : Fin n), ∃(p q : MvPolynomial (Fin (n+1)) ℂ),
  Demazure i (p * q) ≠ Demazure i p * Demazure i q := by
      intro i
      use (X i)
      use C 1
      simp[Demazure, DemazureFun, DemazureNumerator, DemazureDenominator,
       SwapVariables, SwapVariablesFun, Transposition, TranspositionFun,
       fin_succ_ne_fin_castSucc, Fin.succ_ne_zero]

      rw[one_of_div_by_monic_self]
      simp[AlgHom.map_one]
      exact Polynomial.monic_X_sub_C (X i)
```

We want to prove that there exist a pair of polynomials such that the Demazure operators don't respect their product. The way to solve it is by selecting the polynomials (x_i and 1) with the `use` keyboard and proving that specific case.

As before, there's a lot of boilerplate. Now consider the following proposition, outlining some properties of the composition of Demazure operators:

Definition 3.3. Let $p \in P_n$ be a polynomial. We say that p is a **symmetric** polynomial if for every $\sigma \in S_n$,

$$\overline{\sigma}(p) = p$$

Since these induction functions are an action of S_n, we can consider the symmetric polynomials as those invariant by this action. Therefore we denote them by $P_n^{S_n}$.

mathlib

```
def IsSymmetric [CommSemiring R] (φ : MvPolynomial σ R) : Prop :=
  ∀ e : Perm σ, rename e φ = φ
```

Remark 3.7. *In particular, for $i, j \in [n]$, swapping the variables of a symmetric polynomial $p \in P_n$ doesn't modify it, since*

$$\overline{s_{i,j}}(p) = p$$

```
lemma symm_invariant_swap_variables {i j : Fin n}
{g : MvPolynomial (Fin n) ℂ} (h : MvPolynomial.IsSymmetric g) :
    SwapVariablesFun i j g = g := by
  simp[SwapVariablesFun]
  exact h (Equiv.swap i j)
```

Remark 3.8. *The symmetric polynomials form a sub $\mathbb{C}-$algebra of the polynomial ring.*

mathlib

```
def MvPolynomial.symmetricSubalgebra(σ : Type u_5) (R : Type u_6)
[CommSemiring R] :
    Subalgebra R (MvPolynomial σ R)
```

Proposition 3.9. *Let $1 \leq i, j < n$. The following relations hold:*

1. $\partial_i \partial_j = \partial_j \partial_i$ *if* $|i - j| > 1$
2. $\partial_i \partial_{i+1} \partial_i = \partial_{i+1} \partial_i \partial_{i+1}$
3. *Let g be a symmetric polynomial. Then,* $\partial_i(gf) = g\partial_i(f)$

In particular, this shows that the Demazure operators are not only $\mathbb{C}-$linear, but $P_n^{S_n}-$linear as well.

We have already had problems with the complexity of a single Demazure operator, and when we combine them as in this proposition, it quickly gets out of hand, and combinatorial proofs like this would require hundreds of lines. That's why we will introduce an auxiliary definition of Demazure operators that is more suitable for this kind of proof. We aim to improve the following points:

1. Decrease the number of layers in the definition, to make proofs shorter.
2. Handle both input and output in terms of polynomial fractions, extending beyond mere polynomials. This approach enables us to formally chain multiple Demazure operators while preserving the initial polynomial, unlike when using `divByMonic`, which obscures access to the internal polynomial.

3.3 Alternative definition

For this, we create a new structure to represent fractions of polynomials of degree $n \in \mathbb{N}$:

```
structure PolyFraction' (n : ℕ) where
  numerator : MvPolynomial (Fin (n + 1)) ℂ
  denominator : MvPolynomial (Fin (n + 1)) ℂ
  denominator_ne_zero : denominator ≠ 0
```

It features a polynomial for the numerator, another one for the denominator and a proposition stating that the latter is non-zero. To create an instance of a polynomial fraction, we have to specify the three fields:

```
example : PolyFraction' 2 := ⟨X 0 + X 1, 1, one_ne_zero⟩
```

In this case, the fraction is $\frac{x_0+x_1}{1}$. Note that we supply the proposition `one_ne_zero`, which states that $1 \neq 0$. We introduce a helper definition for the previous situation, where we consider a polynomial as a fraction (with denominator 1)

```
def to_frac (p : MvPolynomial (Fin (n + 1)) ℂ) : PolyFraction' n :=
    ⟨p, 1, one_ne_zero⟩
```

The next step is to define the operations between these fractions and the most notable elements:

```
def add' {n : ℕ} : PolyFraction' n → PolyFraction' n → PolyFraction' n :=
  fun p q => ⟨p.numerator * q.denominator + q.numerator * p.denominator,
    p.denominator * q.denominator, mul_ne_zero p.denominator_ne_zero
    q.denominator_ne_zero⟩

def sub' {n : ℕ} : PolyFraction' n → PolyFraction' n → PolyFraction n :=
  fun p q ↦ mk ⟨p.numerator * q.denominator - q.numerator * p.denominator,
    p.denominator * q.denominator, mul_ne_zero p.denominator_ne_zero
    q.denominator_ne_zero⟩

def mul'{n : ℕ} : PolyFraction' n → PolyFraction' n → PolyFraction' n :=
  fun p q => ⟨p.numerator * q.numerator, p.denominator * q.denominator,
    mul_ne_zero p.denominator_ne_zero q.denominator_ne_zero⟩

@[simp]
def one' : PolyFraction' n where
  numerator := 1
  denominator := 1
  denominator_ne_zero := one_ne_zero

@[simp]
def zero' : PolyFraction' n where
  numerator := 0
  denominator := 1
  denominator_ne_zero := one_ne_zero

def neg' (p : PolyFraction' n) : PolyFraction' n :=
  ⟨-p.numerator, p.denominator, p.denominator_ne_zero⟩
```

Now we can perform these operations. For example, if we have to polynomial fractions `x` and `y` we can add them with `add' x y`.

Going forward, we will need to use the fact that this type admits addition in a lot of theorems. There is a class in Lean stating exactly this:

mathlib

```
class Add (α : Type u) where
  /-- `a + b` computes the sum of `a` and `b`. See `HAdd`. -/
  add : α → α → α
```

We define `PolyFraction' n` as an instance of this class to be able to use the notation `p + q` going forward.

```
instance addition' : Add (PolyFraction' n) := ⟨add'⟩
```

Of course, we define the instances for the rest of operations as well:

```
instance : Mul (PolyFraction' n) := ⟨mul'⟩
instance : Sub (PolyFraction' n) := ⟨sub'⟩
```

Now, if we kept the structure like this, in order for two `PolyFraction'` to be equal, all their fields should be equal. But naturally, we want proportional fractions to be equal. To fix this, we introduce an equivalence relation:

```
def r (n : ℕ) : PolyFraction' n → PolyFraction' n → Prop :=
  fun p q => p.numerator * q.denominator = q.numerator * p.denominator
```

(two fractions are equivalent under `r` if both diagonal products are equal) Next, we prove that this is an equivalence relation, which we will need to do the partitioning.

```
lemma r_equiv : Equivalence (r n) := by
    ...
```

In the body of the lemma we have to prove the three properties of an equivalence relation; reflexivity, symmetry and transitivity:

```
refl : ∀ (x : PolyFraction' n), r n x x
symm : ∀ {x y : PolyFraction' n}, r n x y → r n y x
trans : ∀ {x y z : PolyFraction' n}, r n x y → r n y z → r n x z
```

All of them are quite straightforward to prove.

Next, we let Lean know that the type `PolyFraction'` admits an equivalence relation, again using type-class inference:

```
instance s (n : ℕ) : Setoid (PolyFraction' n) where
  r := r n
  iseqv := r_equiv

instance has_equiv : HasEquiv (PolyFraction' n) := instHasEquivOfSetoid
```

Now we can use the notation `a ≈ b` to state that two polynomial fractions `a` and `b` are equivalent under `r`, but most importantly, we can create a new type as the quotient by this equivalence relation:

```
def PolyFraction (n : ℕ) := (Quotient (s n))
```

And definitions to make it easier to create elements of this quotient from the base polynomial fractions or from a polynomial.

```
def mk (p : PolyFraction' n) : PolyFraction n := Quotient.mk (s n) p

def mk' (p : MvPolynomial (Fin (n + 1)) ℂ) : PolyFraction n :=
    mk ⟨p, 1, one_ne_zero⟩
```

From now on, when we will write polynomial fractions to refer to elements in `PolyFraction n` and polynomial fraction representatives for elements in `PolyFraction' n`.

To define the arithmetic operations in the quotient ring, we use the following tool:

mathlib

```
abbrev Quotient.lift₂
    (f : α → β → φ)
    (
        c : ∀ (a₁ : α) (b₁ : β) (a₂ : α) (b₂ : β),
```

```
        a₁ ≈ a₂ → b₁ ≈ b₂ → f a₁ b₁ = f a₂ b₂
    )
    (q₁ : Quotient s₁)
    (q₂ : Quotient s₂) :
φ
```

This enables us to lift functions with two arguments to the quotient. `f` is the function we want to lift, in our case `mul'`. Note that the signature of this function determines α, β, φ. We want the result of the lift to be in `PolyFraction n`, so we need to take `mul'` and project it to the quotient (keep in mind that what we lift is the domain, not the image). So we introduce this function as the candidate for `f`:

```
def mul_mk {n : ℕ} : PolyFraction' n → PolyFraction' n → PolyFraction n :=
  fun p q => mk (mul' p q)
```

This way, we have α = β = `PolyFraction' n` and φ = `PolyFraction n`. Therefore, the two arguments `q₁, q₂` are in the quotient of `PolyFraction' n`, that is, `PolyFraction n`.

Then, the main part is the argument `c`, a proof that the function `f` is well defined w.r.t this relation.

```
lemma mul'_s {n : ℕ} : ∀ a₁ b₁ a₂ b₂ : PolyFraction' n, a₁ ≈ a₂ → b₁ ≈ b₂ →
    (mul_mk a₁ b₁) = (mul_mk a₂ b₂) := by
  intro a1 b1 a2 b2
  intro h1 h2
  simp[mul_mk, mul']
  ring
  rw[← equiv_r] at h1
  rw[← equiv_r] at h2
  simp[r] at h1
  simp[r] at h2

  rw[mul_comm a1.numerator]
  rw[mul_assoc b1.numerator]
  rw[h1]

  rw[mul_comm b1.numerator]
  rw[mul_assoc (a2.numerator * a1.denominator)]
  rw[h2]
  ring
```

We start by introducing the polynomial fraction representatives and their equivalence relations. We unfold the addition in both sides and then also the definition of being equivalent with `rw[← equiv_r] at h1` and `rw[← equiv_r] at h2`. The last two paragraphs are manipulating the goal until the left hand side of the (unfolded) identities `h1` and `h2` appear, and then replacing them with their right hand sides. Finally, the tactic `ring` takes care of showing the equality of both sides of the goal.

Now we can define multiplication in the quotient ring:

```
def mul : PolyFraction n → PolyFraction n → PolyFraction n :=
  fun p q ↦ Quotient.lift₂ (mul_mk) (mul'_s) p q
```

Again, we let Lean know that `PolyFraction n` admits multiplication by creating an instance of `Mul`.

```
instance : Mul (PolyFraction n) := ⟨mul⟩
```

And we proceed in the same way for addition and subtraction.

For the following proofs we will need some basic arithmetic properties. For example:

```
@[simp]
lemma add_comm (p q : PolyFraction n) : add p q = add q p := by
  rcases get_polyfraction_rep p with ⟨p', hp⟩
  rcases get_polyfraction_rep q with ⟨q', hq⟩
  simp[add]
  rw[← hp]
  rw[← hq]
  simp[lift2_r]
  simp[add_mk, add']
  ring
```

The most noteworthy part of the proof is the use of `get_polyfraction_rep`. It's defined this way:

```
lemma get_polyfraction_rep (p : PolyFraction n) : ∃p' : PolyFraction' n, mk p' =
    p := by
    simp[mk]
    apply Quotient.exists_rep p
```

Basically it tells us that every quotient class has a representative. In the proof of `add_comm`, we extract the representative and the proposition stating that its projection is in that class with the tactic `rcases`.

After unpacking the representatives and simplifying the addition, we end up with the goal

```
⊢ Quotient.lift₂ add_mk (mk p') (mk q') =
Quotient.lift₂ add_mk (mk q') (mk p')
```

To prove that two lifts applied to projections are equal, we use this helper lemma:

```
@[simp]
lemma lift2_r {a b : PolyFraction' n}
{f : PolyFraction' n → PolyFraction' n → PolyFraction n}
{
    c :  ∀ (a₁ b₁ a₂ b₂ : PolyFraction' n),
    a₁ ≈ a₂ → b₁ ≈ b₂ → f a₁ b₁ = f a₂ b₂
} :
Quotient.lift₂ f c (mk a) (mk b) = f a b := by rfl
```

It just says that under the conditions needed to lift a function `f`, the image of the lift of `f` at $[a]_\sim, [b]_\sim$ is just the image of `f` at `a, b`. Once we are at `PolyFraction' n`, it suffices to unfold the definition of addition and use the `ring` tactic to prove commutativity.

With this construction we are ready to give our alternate definition of Demazure operators. As we mentioned before, we can easily extend the definition of these operators to polynomial fractions to make composition easier. Let $p, q \in P_{n+1},\ 0 \leq i < n$. Then,

$$\partial_i\left(\frac{p}{q}\right) = \frac{\frac{p}{q} - s_i(\frac{p}{q})}{x_i - x_{i+1}} = \frac{\frac{p\cdot s_i(q) - s_i(p)\cdot q}{q\cdot s_i(q)}}{x_i - x_{i+1}} = \frac{p\cdot s_i(q) - s_i(p)\cdot q}{q\cdot s_i(q)\cdot(x_i - x_{i+1})}$$

Or in Lean,

```
def DemAux' (i : Fin n) : PolyFraction' n →  PolyFraction' n := fun p =>
⟨
  p.numerator * (SwapVariables (Fin.castSucc i) (Fin.succ i) p.denominator) -
    (SwapVariables (Fin.castSucc i) (Fin.succ i) p.numerator) * p.denominator,

  p.denominator * (SwapVariables (Fin.castSucc i) (Fin.succ i) p.denominator) *
    (X (Fin.castSucc i) - X (Fin.succ i)),

  mul_ne_zero (mul_ne_zero p.denominator_ne_zero (swap_variables_ne_zero
    (Fin.castSucc i) (Fin.succ i) p.denominator p.denominator_ne_zero))
    (demazure_denominator_not_null i)
⟩
```

`mul_ne_zero` states that the multiplication of two polynomials (or any instance of Mul) is non-zero if both operands are non-zero. We use it to prove that the denominator is not null, since none of its factors is.

Also notice that, as with the original definition, $i \in \operatorname{Fin} n$, meaning that $0 \leq i < n$. But the polynomial fractions are defined in $n+1$ variables, so the indices have to be converted to elements of $\operatorname{Fin}(n+1)$. To make the conversion we use `Fin.castSucc` to cast i into `Fin (n+1)` while retaining its value, and `Fin.succ` to get $i+1$ as an element of `Fin (n+1)`.

As with the basic operations, we define the auxiliary Demazure operator in the quotient by proving that it has the same image at elements of the same class.

```
lemma DemAux_well_defined (i : Fin n) :
    ∀ (p q : PolyFraction' n) (h : p ≈ q),
    ((mk ∘ DemAux' i) p) = ((mk ∘ DemAux' i) q) := by
  intro p q h
  simp[DemAux']
  rw[← equiv_r] at h
  simp[r] at h
  ring
  rw[mul_comm p.numerator]
  rw[mul_assoc (SwapVariablesFun (Fin.castSucc i) (Fin.succ i) p.denominator)]
  rw[h]

  rw[mul_comm (SwapVariablesFun (Fin.castSucc i) (Fin.succ i) p.numerator)]
  rw[mul_assoc p.denominator]
  rw[mul_comm (SwapVariablesFun (Fin.castSucc i) (Fin.succ i) p.numerator)]
  rw[mul_assoc p.denominator]
  rw[mul_assoc q.denominator]
  rw[← swap_variables_mul]
  rw[h]

  simp[swap_variables_mul]
  ring
```

And finally we can write the definition in the quotient using `Quotient.lift`, in a similar way of how we did it with multiplication, just a bit simpler since DemAux only takes one polynomial fraction.

```
def DemAux (i : Fin n) (p : PolyFraction n) : PolyFraction n :=
  Quotient.lift (mk ∘ (DemAux' i)) (DemAux_well_defined i) p
```

The primary purpose of introducing this alternative definition is to simplify proofs, making it crucial to demonstrate that this new definition is indeed equivalent to the original one. That is, we want to prove that our first Demazure operator definition

$$\partial_i(p) = s_{0,i} \circ (\widetilde{ev}_g^{Q_n[y]})^{-1} \left(\frac{\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p))}{y - y_i} \right)$$

Is equivalent to the new definition when applied to a fraction with denominator 1 (using `to_frac` in Lean)

$$\partial_i \left(\frac{p}{1} \right) = \frac{p \cdot s_i(1) - s_i(p) \cdot 1}{1 \cdot s_i(1) \cdot (x_i - x_{i+1})}$$

```
lemma demazure_definitions_equivalent' :
∀ i : Fin n, ∀ p : MvPolynomial (Fin (n + 1)) ℂ,
 mk (DemAux' i (to_frac p)) = mk' (DemazureFun i p)
```

After simplifying the expression for $\partial_i \left(\frac{p}{1} \right)$, we turn the goal into showing that the cross product is equal, that is,

$$p - s_i(p) = \partial_i(p) \cdot (x_i - x_{i+1})$$

```
intro i p
simp[mk']
simp[DemAux', to_frac]
```

Now, we apply $\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}$ at both sides to turn the goal into

$$\widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(p - s_i(p)) = \widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(\partial_i(p) \cdot (x_i - x_{i+1})) \tag{2}$$

```
apply (SwapVariables (Fin.castSucc i) (0 : Fin (n + 1))).injective
apply (MvPolynomial.finSuccEquiv ℂ n).injective
```

Then, we prove two equalities; First, if $\partial_i(p) = q/r$ for some polynomials $q, r \in P_{n+1}$ and r monic (where q/r is the **quotient** of the polynomial division) we know that, since the division is exact,

$$r \cdot \partial_i(p) = q$$

```
have h :
    DemazureDenominator i * ((DemazureNumerator i p) /ₘ (DemazureDenominator i)) =
    DemazureNumerator i p
:= demazure_division_exact' i p
```

On the other hand, by definition of `DemazureNumerator`,

$$q = \widetilde{ev}_g^{Q_n[y]} \circ s_{i,0}(p - s_i(p))$$

```
have h2 : DemazureNumerator i p =
    (MvPolynomial.finSuccEquiv ℂ n)
        ((SwapVariables (Fin.castSucc i) 0)
            (p - SwapVariablesFun (Fin.castSucc i) (Fin.succ i) p
        )
:= by
    simp [DemazureNumerator]
```

Rewriting both of these equalities at Eq. (2) yields:

$$r \cdot \partial_i(p) = \widetilde{ev}_g^{Q_n[y]} \circ s_{0,i}(\partial_i(p) \cdot (x_i - x_{i+1}))$$

We will continue to peel back layers gradually until reaching an identity, at which point the proof will be finalized. Further details about this process are available in the code.

3.4 Combinatorial results

We now possess all the necessary tools to demonstrate combinatorial properties of Demazure operators. To begin, let's consider a straightforward example.

Proposition 3.10. *Let* $0 \leq i < n$, $p \in P_{n+1}$. *Then,* $\partial_i \circ \partial_i(p) = 0$

```
lemma demazure_order_two : ∀ (i : Fin n)
    (p : MvPolynomial (Fin (n + 1)) ℂ),
  Demazure i (Demazure i p) = 0
```

Proof. Using the alternate definition in Lean,

```
lemma demaux_order_two : ∀ (i : Fin n) (p : PolyFraction n),
  (DemAux i  ∘ DemAux i) p = zero
```

We start getting representatives in Lean to go down to `PolyFraction' n` and unfolding the definitions.

```
intro i p
rcases get_polyfraction_rep p with ⟨p', rfl⟩
simp[DemAux]
rw[lift_r]
rw[lift_r]
rw[zero]
apply Quotient.sound
rw[← equiv_r]
simp[r, DemAux']
```

Then it's just a computation.

$$\partial_i \circ \partial_i(p) = \partial_i\left(\frac{p - s_i(p)}{x_i - x_{i+1}}\right) = \frac{\frac{p - s_i(p)}{x_i - x_{i+1}} - s_i(\frac{p - s_i(p)}{x_i - x_{i+1}})}{x_i - x_{i+1}} =$$

$$\frac{\frac{p - s_i(p)}{x_i - x_{i+1}} + \frac{s_i(p) - s_i \circ s_i(p)}{x_i - x_{i+1}}}{x_i - x_{i+1}} = \frac{\frac{p - s_i(p) + s_i(p) - p}{x_i - x_{i+1}}}{x_i - x_{i+1}} = 0$$

In Lean, once we are at the `PolyFraction' n` level, without polynomial division, quotients or any other construct, these computational proofs are as easy as using the built-in ring arithmetic tactic.

```
ring
```

Now to get the result for the original definition, we will always proceed the same way; first descend to the quotient

```
intro i p
apply eq_zero_of_mk'_zero.mp
simp[Demazure]
```

Now the goal is

```
⊢ mk' (DemazureFun i (DemazureFun i p)) = zero
```

So we just interchange the definitions and apply the lemma of the auxiliary definition, completing the proof:

```
rw[← demazure_definitions_equivalent]
rw[← demazure_definitions_equivalent]

exact demaux_order_two i (mk' p)
```

□

All the proofs of combinatorial results follow the same recipe, first prove them with the auxiliary definition and then use it to get the equivalent result for the original definition with the previous procedure. Therefore, from now we will just focus on the proofs for the auxiliary definition.

We are ready to tackle more advanced results.

Proof of Proposition 3.9.

(1) $\partial_i \partial_j = \partial_j \partial_i$ if $|i - j| > 1$

We first need to define what it means for two indices to be non-adjacent in Lean:

```
def NonAdjacent (i j : Fin n) : Prop :=
  (Fin.castSucc i : Fin (n + 1)) ≠ (Fin.castSucc j : Fin (n + 1)) ∧
  (Fin.castSucc i : Fin (n + 1)) ≠ (Fin.succ j : Fin (n + 1)) ∧
  (Fin.succ i : Fin (n + 1)) ≠ (Fin.castSucc j : Fin (n + 1)) ∧
  (Fin.succ i : Fin (n + 1)) ≠ (Fin.succ j : Fin (n + 1))
```

This definition ensures that neither the original indices nor their successors overlap, which is crucial for the commutativity of the corresponding transpositions. In mathematical notation, this means:

$$i \neq j,\ i \neq j+1,\ i+1 \neq j,\ i+1 \neq j+1$$

We then prove that transpositions with non-adjacent indices commute:

```
lemma transposition_commutes_non_adjacent (i j : Fin n)
    (h : NonAdjacent i j) :
  Equiv.swap (Fin.castSucc i) (Fin.succ i) *
    Equiv.swap (Fin.castSucc j) (Fin.succ j) =
  Equiv.swap (Fin.castSucc j) (Fin.succ j) *
    Equiv.swap (Fin.castSucc i) (Fin.succ i)
```

The key insight is that non-adjacent transpositions act on disjoint sets of elements, which implies they commute. In Lean, we prove this using the following steps:

First, we destructure the `NonAdjacent` hypothesis:

```
rcases h with ⟨h1, h2, h3, h4⟩
```

We want to prove that the transpositions are disjoint, which is captured in the built-in proposition `Equiv.Perm.Disjoint`

```
have h_disjoint : Equiv.Perm.Disjoint
  (Equiv.swap i.castSucc i.succ)
  (Equiv.swap j.castSucc j.succ)
```

Two transpositions are disjoint if for any element $k \in [n]$, if it's moved by one transposition, it cannot be moved by the other. We introduce this k and assume that $s_i(k) \neq k$.

```
intro k
apply or_iff_not_imp_left.mpr
intro h
```

As a consequence of the `NonAdjacent` inequalities, we can easily prove that $i \neq j$

```
have heq : i ≠ j := by
    intro h'
    apply h1
    simp[h']
```

Now, the goal becomes to prove that $s_j(k) = k$.

```
⊢ (Equiv.swap j.castSucc j.succ) k = k
```

First, let's understand what `Equiv.eq_or_eq_of_swap_apply_ne_self` does: Given a transposition (a, b) and an element x where $(a, b)(x) \neq x$, it states that either $x = a$ or $x = b$. This is clear by the definition of transpositions.

So we consider both cases, k must be either i or $i + 1$.

```
rcases Equiv.eq_or_eq_of_swap_apply_ne_self h with h | h
```

Next, `Equiv.swap_apply_of_ne_of_ne` states: If $x \neq a$ and $x \neq b$, then $(a, b)(x) = x$ So it suffices to prove that $x \neq j, j + 1$ when $x = i$ or $x = i + 1$. These proofs by cases are a great match for the `repeat` clause, that, as the name suggests, applies a tactic repeatedly until it can no longer advance.

```
repeat
    simp[h, heq, h1, h2, h3, h4, h1.symm, h2.symm, h3.symm, h4.symm]
```

We tell it to simplify the goal with all our hypotheses:

- `h` tells us which of the two cases from step 2 we're in
- `heq` states that $i \neq j$
- `h1,h2,h3,h4` are our non-adjacency conditions

The `.symm` property flips both sides of an equality, so if `h : a = b`, then `h.symm : b = a`. This finished the proof that s_i and s_j are disjoint.

Finally, we apply the fundamental theorem that disjoint permutations commute:

```
rw[Equiv.Perm.Disjoint.commute h_disjoint]
```

This commutativity at the level of permutations trivially implies the commutativity of their induced isomorphisms, so we omit the proof here.

```
lemma swap_variables_commutes_non_adjacent
    (i j : Fin n) (h : NonAdjacent i j)
    {p : MvPolynomial (Fin (n + 1)) ℂ} :
  SwapVariablesFun (Fin.castSucc i) (Fin.succ i)
    (SwapVariablesFun (Fin.castSucc j) (Fin.succ j) p) =
  SwapVariablesFun (Fin.castSucc j) (Fin.succ j)
    (SwapVariablesFun (Fin.castSucc i) (Fin.succ i) p)
```

Finally, we can prove that the Demazure operators of non-adjacent indices commute.

```
lemma demaux_commutes_non_adjacent
(i j : Fin n)  (h : NonAdjacent i j) :
    ∀ p : MvPolynomial (Fin (n + 1)) ℂ,
        (DemAux i ∘ DemAux j) (mk' p) = (DemAux j ∘ DemAux i) (mk' p) := by
```

We start by picking representatives of the quotient, and turning the proof into showing that the cross product of the fractions at both sides are equal.

```
intro p
simp[DemAux, mk']
repeat rw[lift_r]
apply mk_eq.mpr
simp[DemAux']
```

The resulting goal is quite long, since we are composing two Demazure operators.

$$\begin{aligned}
&((p - s_j(p)) \cdot (s_i(x_j) - s_i(x_{j+1})) - (s_i(p) - s_i(s_j(p))) \cdot (x_j - x_{j+1})) \\
&\cdot ((x_i - x_{i+1}) \cdot (s_j(x_i) - s_j(x_{i+1})) \cdot (x_j - x_{j+1})) \\
&= (x_j - x_{j+1}) \cdot (s_i(x_j) - s_i(x_{j+1})) \cdot (x_i - x_{i+1}) \\
&\cdot ((p - s_i(p)) \cdot (s_j(x_i) - s_j(x_{i+1})) - (s_j(p) - s_j(s_i(p))) \cdot (x_i - x_{i+1}))
\end{aligned}$$

Thankfully, since this is an expression directly in the polynomial ring (thanks to the auxiliary definition!), we can let Lean solve it with its built-in arithmetic engine. We just need to supply the fact that variable swaps commute for this indices and all the non-adjacent inequalities.

```
simp[swap_variables_commutes_non_adjacent i j h]

rcases h with ⟨h1, h2, h3, h4⟩
simp[h1, h2, h3, h4, h1.symm, h2.symm, h3.symm, h4.symm]
ring
```

(2) $\partial_i \partial_{i+1} \partial_i = \partial_{i+1} \partial_i \partial_{i+1}$

This part is really similar to the previous one, so we won't get into much detail. We prove that transpositions commute in a similar way.

$$s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1}$$

```
lemma transposition_commutes_adjacent {i : Fin n} {j : Fin (n + 1)}
(h0 : i < n + 1) (h1 : i + 1 < n + 1) (h2 : i + 2 < n + 1) :
  Equiv.swap ⟨i, h0⟩ ⟨i + 1, h1⟩
    (Equiv.swap ⟨i + 1, h1⟩ ⟨i + 2, h2⟩ (Equiv.swap ⟨i, h0⟩ ⟨i + 1, h1⟩ j)) =
  Equiv.swap ⟨i + 1, h1⟩ ⟨i + 2, h2⟩
    (Equiv.swap ⟨i, h0⟩ ⟨i + 1, h1⟩ (Equiv.swap ⟨i + 1, h1⟩ ⟨i + 2, h2⟩ j))
```

In this case we don't have that many cases, we just prove that the result of applying the functions at both sides is equal when $i = i, i+1, i+2$ or something else.

```
simp[Equiv.swap_apply_def]

by_cases c0 : j = ⟨i, h0⟩
simp[c0]
by_cases c1 : j = ⟨i + 1, h1⟩
simp[c1]
by_cases c2 : j = ⟨i + 2, h2⟩
simp[c2]

simp[c0,c1,c2]
```

And use this property to prove the equivalent for Demazure operators:

```
lemma demaux_commutes_adjacent (i : Fin n) (h : i + 1 < n) :
    ∀ p : MvPolynomial (Fin (n + 1)) ℂ,
  (DemAux i ∘ DemAux ⟨i+1, h⟩ ∘ DemAux i) (mk' p) =
  (DemAux ⟨i+1, h⟩ ∘ DemAux i ∘ DemAux ⟨i+1, h⟩) (mk' p)
```

In this scenario, both sides involve two compositions, leading to an increased number of calculations. Nonetheless, Lean is capable of managing these computations provided we slightly extend the maximum time allowed for each simplification step:

```
set_option maxHeartbeats 10000000
```

Using this setup, it requires approximately 30 seconds to conduct a verification on a standard laptop, which is a reasonably short time. (3) Let g be a symmetric polynomial. Then, $\partial_i(gf) = g\partial_i(f)$

First, we extend the definition of symmetric polynomials to polynomial fractions, where we say that $\frac{p}{q}$ for some $p, q \in P_n$ is symmetric if both p and q are.

mathlib

```
def IsSymmetric (p : PolyFraction n) : Prop := ∃p' : PolyFraction' n,
    mk p' = p ∧
    MvPolynomial.IsSymmetric p'.numerator ∧
    MvPolynomial.IsSymmetric p'.denominator
```

Then, we can state our goal as

```
lemma demaux_mul_symm (i : Fin n) (g f : PolyFraction n) (h : IsSymmetric g) :
  DemAux i (g*f) = g*(DemAux i f)
```

As usual, we take representatives of the polynomial fractions $f = \frac{f_1}{f_2}$ and $g = \frac{g_1}{g_2}$. Note that we use `h` to get the representative for g, so we know directly that its numerator and denominator are symmetric polynomials.

```
rcases h with ⟨g', ⟨rfl, g_num_symm, g_denom_symm⟩⟩
rcases get_polyfraction_rep f with ⟨f', rfl⟩
```

Then, we apply the usual lemmas to transform the problem into just working with representatives, where we have to prove

$$\partial_i \left(\frac{g_1 \cdot f_1}{g_2 \cdot f_2} \right) = g \cdot \partial_i(f)$$

```
rw[mk_mul]
simp[DemAux]
repeat rw[lift_r]
```

Remember that this equation is between polynomial fractions, so we can apply the demazure operator definitions and show that the cross product of the resulting expressions is equal

$$\begin{aligned}(g_1 f_1 \cdot (s_i(g_2) \cdot s_i(f_2)) - s_i(g_1) \cdot s_i(f_1) \cdot (g_2 \cdot f_2)) \\ \cdot (g_2 \cdot (f_2 \cdot s_i(f_2) \cdot (x_i - x_{i+1}))) \\ = \\ g_2 \cdot f_2 \cdot (s_i(g_2) \cdot s_i(f_2)) \cdot (x_i - x_{i+1}) \\ \cdot (g_1 \cdot (f_1 \cdot s_i(f_2) - s_i(f_1) \cdot f_2))\end{aligned}$$

```
rw[← simp_mul']
rw[← simp_mul]
rw[mk_mul]
rw[mk_eq]
simp[DemAux']
```

Once again, thankfully we can use the ring arithmetic tactic after canceling $\overline{s_{i,j}}(p) = p$ wherever p is a symmetric polynomial to obtain the result.

```
simp[symm_invariant_swap_variables g_num_symm,
    symm_invariant_swap_variables g_denom_symm]
ring
```

In particular, this implies the result for the case $f_2 = g_2 = 1$, which is what we are interested in. We could have done the proof directly for that case, but thanks to the `simp` and `ring` tactics, there is no extra cost associated to proving this generalized version. □

As we saw in this proof, the combinatorial properties of Demazure operators closely resemble those of the transpositions s_i. In fact, we proved most of them based on the equivalent for transpositions.

And we know that the transpositions s_i generate S_{n+1}. This means that any $w \in S_{n+1}$ can be expressed as

$$w = s_{i_0} s_{i_1} \cdots s_{i_{p-1}}$$

For some $i_k \in [n]$ for every $k \in [p]$ Therefore, one is tempted to generalize the definition of Demazure operators to S_{n+1}, where we take $\partial_{s_i} := \partial_i$ and expand with the group operation, so with $w \in S_{n+1}$ as before,

$$\partial_w = \partial_{i_0} \circ \partial_{i_1} \cdots \partial_{i_{p-1}}$$

The idea is good, but this naive approach has multiple problems:

1. If w has two of the same transposition in a row, there is a fundamental difference in how it behaves at the transposition and Demazure operator levels. Of course, $s_i^2 = e$, so we should have

   $$\partial_{s_i \cdot s_i} = \partial_e = \mathrm{id}$$

   But using the previous definition and Proposition 3.10, we get that

   $$\partial_{s_i \cdot s_i} = \partial_i \circ \partial_i = 0$$

   A clear contradiction.

2. There are multiple ways to get w with a product of transpositions, so how can we guarantee that for two different expressions $w = s_{i_0} s_{i_1} \cdots s_{i_{p-1}} = s'_{j_0} s'_{j_1} \cdots s'_{j_{r-1}}$, the Demazure operators will agree?

   $$\partial_{s_{i_0} s_{i_1} \cdots s_{i_{p-1}} = s'_{j_0} s'_{j_1}} = \partial_{i_0} \circ \partial_{i_1} \cdots \partial_{i_{p-1}} \stackrel{?}{=} \partial_{j_0} \circ \partial_{j_1} \cdots \partial_{j_{r-1}} = \partial_{j_0} \circ \partial_{j_1} \cdots \partial_{j_{r-1}}$$

In order to address these issues and extend the concept of Demazure operators, a thorough examination of the combinatorial characteristics of the transpositions s_i in S_{n+1} is required. Interestingly, their behaviour is not limited to the symmetric group and can be analysed within a broader class of groups.

4 Coxeter groups

We will give a tour of the already formalised landscape of Coxeter groups (named after H. S. M. Coxeter, who introduced them in [Cox34]) and then prove some major results that weren't formalised before.

We mainly follow [AB05], since it's the book that has been taken a reference for the majority of results already in mathlib. However, our proofs will sometimes differ, specially in the section about Matsumoto's theorem.

4.1 Definition

Let B be any set. We will call the elements of B "indices".

Definition 4.1. A matrix $M : B \times B \to \mathbb{N} \cup \{0\}$ is a Coxeter matrix if it satisfies the following properties for all $i, j \in B$:

1. $M(i,j) = M(j,i)$ (M is symmetric)
2. $M(i,i) = 1$
3. $M(i,i') \neq 1$ if $i \neq j$

Coxeter matrices are part of Lean's default mathematics library as:

mathlib

```
structure CoxeterMatrix(B : Type u_1) : Type u_1

isSymm : self.M.IsSymm
diagonal : ∀ (i : B), self.M i i = 1
off_diagonal : ∀ (i i' : B), i ≠ i' → self.M i i' ≠ 1
```

A Coxeter matrix M determines a Coxeter group (accessible in lean through `M.group`)

$$W = \langle \{s_i\}_{i \in B} \mid (s_i s_j)^{M(i,j)} \ \forall i, j \in B \rangle \tag{3}$$

So, $S = \{s_i : i \in B\}$ are the generators of the Coxeter group (think of the transpositions in the S_n case). Although we did not explicitly request them to be distinct, this can be demonstrated based on the definition of a Coxeter group. (Proposition 1.1.1 of [AB05])

In particular, given that $M(i,i) = 1$, it follows that $s_i^2 = e$. These are known as **nil** relations. For indices i and j where $i \neq j$, the expression $(s_i s_j)^{M(i,j)}$ is equivalent to

$$\underbrace{s_i s_j s_i s_j \dots}_{m(s_i,s_j)} = \underbrace{s_j s_i s_j s_i \dots}_{m(s_i,s_j)}.$$

These are the **braid** relations.

If a group W has a presentation like the one in 3, then the pair (W, M) is called a **Coxeter system**.

Example 4.1. As we anticipated, the symmetric group S_{n+1} along with the transpositions $S = \{s_0, \dots, s_{n-1}\}$ form a Coxeter system indexed by $[n]$.

The Coxeter matrix associated to S_{n+1} (commonly referred as A_n) is the following, for every $i, j \in [n]$

1. $M(i,i) = 1$
2. $M(i,j) = 2$ if $|i - j| > 1$
3. $M(i, i+1) = 3$

This comes from the relations of the symmetric group that we proved in Proposition 3.9

1. $s_i^2 = e$
2. $s_i s_j = s_j s_i$ if $|i - j| > 1$

3. $s_i s_{i+1} s_i = s_{i+1} s_i s_{i+1}$

Keep in mind that to prove that S_{n+1} corresponds to this Coxeter group, we have to show that it is completely determined by these relations, which is not trivial. We will go over this in more detail in Section 5.

Coxeter systems are formalized in Lean by:

mathlib

```
structure CoxeterSystem{B : Type u_1} (M : CoxeterMatrix B) (W : Type u_2) [Group
    W] :
Type (max u_1 u_2)

mulEquiv : W ≃* M.Group
```

In `mulEquiv` we supply an isomorphism of W with the presentation of the form 3. This way, we can recognize the Coxeter group structure of a previously defined group.

4.2 Basic properties and facts

Let us fix a set of indices B, a Coxeter matrix $M : B \times B \to \mathbb{N} \cup \{0\}$ and a Coxeter system (W, M). We also do this in Lean through the use of variables. At the start of the files in which we work with Coxeter groups, the first lines are as follows:

```
variable {B : Type}
variable {W : Type} [Group W] [DecidableEq W]
variable {M : CoxeterMatrix B} (cs : CoxeterSystem M W)
```

In Lean, the variable command allows us to introduce variables that we can reuse across multiple definitions, eliminating the need to repeatedly mention the same types or assumptions. For example, instead of passing common parameters like a type, group structure, or matrix into every definition or lemma, we can declare them once with a variable as we did above.

Then, take this example of working with Coxeter groups:

```
lemma braid_relation (i j : B) : (s i * s j) ^ M i j = 1 := by
  simp [CoxeterMatrix]
```

Here, `B`, `M`, and other structures like the group `W` and Coxeter system `cs` are already introduced as variables earlier in the file. The variable command at the beginning ensures that the types and assumptions (such as `W` being a group and `M` a Coxeter matrix) are automatically considered within the scope of every lemma or definition that follows. This means we don't have to specify them again and again in each lemma.

If we had not used the variable command, we would have to declare all these types and structures explicitly every time, leading to much more verbose code. For example, without variable, the `braid_relation` lemma would look something like this:

```
lemma braid_relation (B : Type) (W : Type) [Group W] [DecidableEq W]
  (M : CoxeterMatrix B) (cs : CoxeterSystem M W) (i j : B) :
```

```
  (s i * s j) ^ M i j = 1 := by
  simp [CoxeterMatrix]
```

It is worth emphasizing the different ways to refer to an element in a Coxeter group. Sometimes we are interested in the actual representation of this element in terms of the generators. We call these expressions **words** and use only the indices, so we consider them as elements of B^*, the free group generated by B.

For example $l = i_1 \cdots i_k \in B^*$ for some $k \in \mathbb{N}$. Then the **product** of a word is defined as follows:

Definition 4.2. The product of a word $i_1 i_2 \cdots i_k$ for some $k \in \mathbb{N}$ is:

$$\pi(i_1 i_2 \cdots i_k) := s_{i_1} s_{i_2} \cdots s_{i_k}$$

Notice that different words can equal the same element in W using the aforementioned relations. For example, $iij \neq j$ but $s_i s_i s_j = s_j$, using the nil relation.

With this notation, we emphasize the difference between both objects and make the mathematical notation more closely mimic the Lean syntax, where every object has a specific type.

To understand how words are handled in Lean, we first need to explore Lists. Lists are an example of an **inductive type** — a type defined by specifying various constructors (or ways to build elements of that type). Inductive types are particularly powerful because their constructors can reference the very type being defined, allowing for recursive definitions. This characteristic gives inductive types their name and their flexibility.

For example, the natural numbers are defined this way:

mathlib

```
inductive Nat where
| nil : Nat
| succ : Nat → Nat
```

This corresponds to the fact that a natural number can either be zero, constructed by `Nat.nil` or the successor of another natural number, with `Nat.succ n`, where `n : Nat`. Whenever we have a natural number `n : Nat`, we can separate both cases with the `match` keyboard.

```
match (n : Nat) with
| nil => ... (replaces n with 0)
| succ i => ... (replaces n with i + 1)
```

This structure lends itself well to induction proofs, as we will soon see.

But the most important inductive type for us is `List` α, where α is any type.

Definition 4.3. A list l of elements of any type α is one of the following:

1. An empty list

2. The concatenation of its **head** a (the first element) and its **tail** t (the rest of the list, a smaller list).

In Lean,

mathlib

```
inductive List (α : Type u) where
  | nil : List α
  | cons (head : α) (tail : List α) : List α
```

Often we represent the case `List.nil` with `[]` and the case `List.cons a t` with `a :: t`. Of course, Lean provides a lot of additional ways of constructing a list.

1. Explicitly: For example, we can create a list of natural numbers directly as `[1,2,3,5,2]`
2. Concat: If we want to add an element `i` to a preexisting list `l` at the end instead of at the beginning, we can use

   ```
   l.concat i
   ```

3. Append: If we want to join two lists `l` and `l'`, we can do so with the operator `++`:

   ```
   l ++ l'
   ```

In the upcoming sections, we will employ this list structure to conduct induction proofs directly on lists, a method not commonly employed in conventional mathematics. Nevertheless, this approach can significantly simplify some proofs in Lean, contrasting with the traditional method of performing induction based on list length.

But first, let's explore the basic definitions and lemmas of products of words included in mathlib that we will often use.

To get the generators from any index, we use the following definition:

mathlib

```
def CoxeterSystem.simple (cs : CoxeterSystem M W) (i : B) : W
```

So, to get the generator s_i for $i \in B$ we use the term `cs.simple i`. We have to explicitly specify the Coxeter system because two different Coxeter systems could share the same index set.

In fact, to make the code less verbose and more reminiscent of the mathematical notation, we introduce the following abbreviation, where `cs` is a `CoxeterSystem`:

```
local prefix:100 "s" => cs.simple
```

So now we use `s i` to get the generator s_i.

To get the product of a whole word, we use

mathlib

```
def wordProd (w : List B) : W := prod (map cs.simple w)
```

The `map` operator applies a function (`cs.simple` in our case) to every element of a list, and the `prod` operator returns the product of all its elements.

As before, we introduce an abbreviation to mimic the mathematical notation:

```
local prefix:100 "π" => cs.wordProd
```

So the product of a word `l : List B` can be obtained with the expression π `l`.

For empty lists, the result is clearly the neutral element of W.

mathlib

```
theorem CoxeterSystem.wordProd_nil (cs : CoxeterSystem M W) :
    cs.wordProd [] = 1
```

Often we will require to get the product of a list to which we appended a new head. As expected, this product will be the genererator indexed at the head of the list times the product of the tail.

mathlib

```
theorem CoxeterSystem.wordProd_cons
(cs : CoxeterSystem M W) (i : B) (ω : List B) :
cs.wordProd (i :: ω) = cs.simple i * cs.wordProd ω
```

A similar theorem explains what happens to the product of a word when we append an element at the end (we just multiply the generator of the new element to the right instead of the left)

mathlib

```
theorem CoxeterSystem.wordProd_concat
(cs : CoxeterSystem M W) (i : B) (ω : List B) :
    cs.wordProd (ω.concat i) = cs.wordProd ω * cs.simple i
```

Lemma 4.1. *π is multiplicative, that is, $\forall l, l' \in B^*$,*

$$\pi(l \cdot l') = \pi(l) \cdot \pi(l')$$

Remember than multiplication of words is defined with `List.append` in Lean.

mathlib

```
theorem CoxeterSystem.wordProd_append
(cs : CoxeterSystem M W) (ω : List B) (ω' : List B) :
    cs.wordProd (ω ++ ω') = cs.wordProd ω * cs.wordProd ω'
```

A really important fact is that if (W, S) is a Coxeter system, any of its elements is the product of (at least) one word.

mathlib

```
theorem CoxeterSystem.wordProd_surjective (cs : CoxeterSystem M W) :
    Function.Surjective cs.wordProd
```

Definition 4.4. The reverse/inverse of a word $l = i_0 i_1 \cdots i_{p-1} \in B^*$ is

$$l^{-1} := i_{p-1} i_{p-2} \cdots i_1 i_0$$

In lean, we use `l.reverse`.

Lemma 4.2. *For any word $l \in B^*$,*

$$\pi(l^{-1}) = \pi(l)^{-1}$$

mathlib

```
theorem CoxeterSystem.wordProd_reverse
(cs : CoxeterSystem M W) (ω : List B) :
    cs.wordProd ω.reverse = (cs.wordProd ω)⁻¹
```

4.3 Alternating words

Definition 4.5. An **alternating word** from i to j of length m is the word ending in j that alternates between the two elements m times. That is:

$$a_p(i,j) = \underbrace{\cdots ijij}_{m}$$

In Lean, it is defined inductively:

mathlib

```
def alternatingWord (i i' : B) (m : ℕ) : List B :=
  match m with
  | 0    => []
  | m+1  => (alternatingWord i' i m).concat i'
```

Recall that the `concat` function adds an element to the end of the list. That is why in the `m+1` case, i and i' are flipped, since when we add an element at the end, we change the last element.

We know from the definition how to construct a larger alternating word by concatenating it with the corresponding index:

$$a_{m+1}(i,j) = a_m(j,i) \cdot j$$

mathlib

```
theorem CoxeterSystem.alternatingWord_succ (i : B) (i' : B) (m : ℕ) :
    CoxeterSystem.alternatingWord i i' (m + 1) = (CoxeterSystem.alternatingWord
    i' i m).concat i'
```

But sometimes we may want to construct it by appending an element to the beginning of the alternating word. In that case, we have to keep in mind that the first element of an alternating word depends of the parity of its length.

Theorem 4.3. *Let $i, j \in B$ and $m \in \mathbb{N}$. Then,*

$$a_{m+1}(i,j) = \begin{cases} j \cdot a_m(i,j) & \text{if } m \text{ is even} \\ i \cdot a_m(i,j) & \text{if } m \text{ is odd} \end{cases}$$

mathlib

```
theorem CoxeterSystem.alternatingWord_succ' (i : B) (i' : B) (m : ℕ) :
    CoxeterSystem.alternatingWord i i' (m + 1) =
    (if Even m then i' else i) :: CoxeterSystem.alternatingWord i i' m
```

When we take the product of an alternating word, we obtain a power of $s_i s_j$ with an additional factor if the length is odd.

Theorem 4.4. *Let $i, j \in B$ and $m \in \mathbb{N}$. Then,*

$$\pi(a_m(i,j)) = \begin{cases} (s_i s_j)^{\lfloor m/2 \rfloor} & \text{if } m \text{ is even} \\ s_j (s_i s_j)^{\lfloor m/2 \rfloor} & \text{if } m \text{ is odd} \end{cases}$$

mathlib

```
theorem CoxeterSystem.prod_alternatingWord_eq_mul_pow
(cs : CoxeterSystem M W)(i : B) (i' : B) (m : ℕ) :
    cs.wordProd (CoxeterSystem.alternatingWord i i' m) =
    (if Even m then 1 else cs.simple i') *
                                    (cs.simple i * cs.simple i') ^ (m / 2)
```

The explicit description of each element of an alternating word isn't available in mathlib, so we proved it ourselves. It also exemplifies how proofs by induction work in the context of Coxeter words.

Theorem 4.5. *Let $a_m(i,j) = a_0 a_1 \cdots a_{p-1}$. Then, for every $0 \leq k < p$,*

$$a_k = \begin{cases} i & \text{if } p+k \text{ is even} \\ j & \text{if } p+k \text{ is odd} \end{cases}$$

```
lemma getElem_alternatingWord
(i j : B) (p : ℕ) (k : Fin ((alternatingWord i j p).length)) :
  (alternatingWord i j p)[k] = (if Even (p + k) then i else j)
```

Proof. We prove it by induction on p.

```
induction p with
  | zero => ...
  | succ n h => ...
```

The case where $p = 0$ is trivial, since the alternating word is empty, so there is nothing to prove. We divide k into `k`, the natural number, and `hk` the proof that it is less than `(alternatingWord i j 0).length`. Then we unfold the length of the alternating word at `hk` to end up with $k < 0$, a contradiction.

```
rcases k with ⟨k, hk⟩
simp[alternatingWord] at hk
```

When $p > 0$, we substitute it with $n + 1$ for $n := p - 1$ and we get the induction hypothesis `h`. Now we want to separate the head of the alternatingWord to apply the induction hypothesis, but if we were to do it directly, like

```
rw [alternatingWord_succ' i j n]
```

Lean throws the error

```
tactic 'rewrite' failed, motive is not type correct
```

This happens when Lean loses track of some implicit fact about a dependent argument. In our case, the goal is

```
⊢ (alternatingWord i j (n + 1))[k] = if Even (n + 1 + ↑k) then i else j
```

So it contains explicitly the assumption that $k <$ length$(a_{n+1}(i, j))$. However, when we apply the previous lemma, the goal turns into:

```
((if Even n then j else i) :: alternatingWord i j n)[k] =
if Even (n + 1 + k) then i else j
```

So Lean tries to find a proof that k is less than the length of the list in the left hand side but doesn't find it. The workaround is to extract k as a general parameter with

```
revert k
```

Now the goal becomes

```
⊢ ∀ (k : Fin (alternatingWord i j (n + 1)).length),
(alternatingWord i j (n + 1))[k] = if Even (n + 1 + k) then i else j
```

So when we use the rewrite tactic, we also change the definition of k accordingly, and it matches the goal. Afterwards, we introduce again k with

```
rintro ⟨k, hk⟩
```

(in this case with the natural value and the inequality $k <$ ... separate variables)

Our goal then becomes:

```
((if Even n then j else i) :: alternatingWord i j n)[⟨k, hk⟩] =
if Even (n + 1 + ⟨k, hk⟩) then i else j
```

Then, we proceed by induction on k.

In the case $k = 0$, the element we get is the head, so the goal becomes

```
(if Even n then j else i) = if Even (n + 1) then i else j
```

The solution then becomes a simple proof by cases

```
by_cases h2 : Even n
· have : ¬ Even (n + 1) := by
  simp
  exact Even.add_one h2
  simp [h2, this]
· have : Even (n + 1) := by
  simp at h2
  exact Odd.add_one h2
  simp [h2, this]
```

If $k > 0$, we substitute it with $k+1$ as usual. Then we prove that `k < (alternatingWord i j n).length` and use it to get rid of the head, with the lemma

mathlib

```
theorem List.getElem_cons_succ
(a : α) (as : List α) (i : Nat) (h : i + 1 < (a :: as).length) :
    (a :: as)[i + 1] = as[i]
```

The goal is now

```
(alternatingWord i j n)[k] = if Even (n + 1 + (k + 1)) then i else j
```

Which looks a lot like our induction hypothesis. We rewrite the left hand side with it and simplyfy it

```
rw[h ⟨k, this⟩]
simp
ring
```

To obtain

```
⊢ (if Even (n + k) then i else j) = if Even (2 + n + k) then i else j
```

Again, the result comes from a simple proof by cases

```
have (m : ℕ) : Even (2 + m) ↔ Even m := by
  have aux : m ≤ 2 + m := by linarith
  apply (Nat.even_sub aux).mp
  simp
by_cases h_even : Even (n + k)
· simp [if_pos h_even]
  rw[← this (n+k)] at h_even
  rw[← Nat.add_assoc 2 n k] at h_even
  simp [if_pos h_even]
· simp [if_neg h_even]
  rw[← this (n+k)] at h_even
  rw[← Nat.add_assoc 2 n k] at h_even
  simp [if_neg h_even]
```

□

The most important alternating words are the following:

Definition 4.6. An alternating word from i to j of length $M(i,j)$ is called a **braid word**, and denoted as:

$$b(i,j) := a_{M(i,j)}(i,j)$$

mathlib

```
abbrev braidWord (M : CoxeterMatrix B) (i i' : B) : List B := alternatingWord i
    i' (M i i')
```

Of course, the name comes from the fact that braid words are related by the braid relations:

$$\underbrace{\ldots s_i s_j s_i s_j}_{m(s_i,s_j)} = \underbrace{\ldots s_j s_i s_j s_i}_{m(s_i,s_j)} \iff \pi(b(i,j)) = \pi(b(j,i))$$

This is captured in the following lemma:

Lemma 4.6. *Let $i, j \in B$. Then,*

$$\pi(b(i,j)) = \pi(b(j,i))$$

mathlib

```
theorem CoxeterSystem.wordProd_braidWord_eq {M : CoxeterMatrix B}
(cs : CoxeterSystem M W) (i : B) (i' : B) :
    cs.wordProd (CoxeterSystem.braidWord M i i') =
    cs.wordProd (CoxeterSystem.braidWord M i' i)
```

4.4 Length of words

When talking about words, we define their length simply as the length of the list/free product.

Definition 4.7. Let $i_0 i_1 \cdots i_{p-1} \in B^*$ for some $p \geq 1$. Then,

$$\text{len}(i_0 i_1 \cdots i_{p-1}) := p$$

For products, the definition is more complicated because two words of different lengths can equal the same product.

Definition 4.8. The length of $w \in W$ is the minimum length of a word whose product is w.

$$\widetilde{\text{len}}(w) = \min_{\substack{l \in B^* \\ \pi(l) = w}} \text{len}(l)$$

To define it in Lean, mathlib makes use of `wordProd_surjective` in this (more explicit) alternate form:

mathlib

```
private theorem exists_word_with_prod (w : W) :
    ∃ n ω, ω.length = n ∧ π ω = w := by
        rcases cs.wordProd_surjective w with ⟨ω, rfl⟩
        use ω.length, ω
```

We use the `rcases` tactic to find a preimage of an element in the codomain of a surjective function.

Then, the length of a product is defined as:

mathlib

```
noncomputable def length (w : W) : ℕ := Nat.find (cs.exists_word_with_prod w)
```

`Nat.find` returns the minimum natural number that satisfies the exist clause in the proposition `cs.exists_word_with_prod w`. Note that is set as `noncomputable` because we don't give a procedure to find this element. Therefore this is not a constructive definition.

Again, we shorten the definition to `len`.

```
local prefix:100 "len" => cs.length
```

Remark 4.7. *Note that if $l \in B^*$, we can have the situation*

$$\text{len}(l) \neq \widetilde{\text{len}}(\pi(l))$$

For example, if $l = i_3 i_1 i_1 i_2$, we have that $\text{len}(l) = 4$, but

$$\pi(l) = s_{i_3} s_{i_1} s_{i_1} s_{i_2} = s_{i_3} s_{i_2} = \pi(i_3 i_2) \implies \widetilde{\text{len}}(\pi(l)) \leq 2$$

We are specially interested in words where the previous condition does hold.

Definition 4.9. We say that a word $l \in B^*$ is **reduced** if its length is that of its product (the minimum of all equivalent words). That is,

$$\text{len}(l) = \widetilde{\text{len}}(\pi(l))$$

mathlib

```
def CoxeterSystem.IsReduced (cs : CoxeterSystem M W) (ω : List B) : Prop :=
    (cs.length (cs.wordProd ω) = ω.length)
```

Lemma 4.8. *For every product $w \in W$, there exists a reduced word $l \in B^*$ such that $\pi(l) = w$.*

mathlib

```
theorem CoxeterSystem.exists_reduced_word' (cs : CoxeterSystem M W) (w : W) :
    ∃ (ω : List B), cs.IsReduced ω ∧ w = cs.wordProd ω
```

We prove the following lemmas, which are straight-forward from the definition:

Lemma 4.9. *Let $l, l' \in B^*$ such that $\text{len}(l) = \text{len}(l')$ and $\pi(l) = \pi(l')$. Then, if l is reduced, so is l'.*

```
theorem isReduced_of_eq_length (l l' : List B)
(h_len : l.length = l'.length)  (h_eq : π l = π l') (hr : cs.IsReduced l) :
    cs.IsReduced l'
```

Proof.

$$\text{len}(l') = \text{len}(l) = \widetilde{\text{len}}(\pi(l)) = \widetilde{\text{len}}(\pi(l'))$$

```
simp[IsReduced]
simp[IsReduced] at hr

calc
    len π l' = len π l := by rw[h_eq]
    _ = l.length := by rw[hr]
    _ = l'.length := by rw[h_len]
```

□

Lemma 4.10. *Let $l, l' \in B^*$ such that they are both reduced and $\pi(l) = \pi(l')$. Then,*

$$\text{len}(l) = \text{len}(l')$$

```
theorem eq_length_of_isReduced (l l' : List B)
(h_eq : π l = π l') (hr : cs.IsReduced l) (hr' : cs.IsReduced l') :
    l.length = l'.length
```

Proof.

$$\text{len}(l) = \widetilde{\text{len}}(\pi(l)) = \widetilde{\text{len}}(\pi(l')) = \text{len}(l')$$

```
rw[IsReduced] at hr
rw[IsReduced] at hr'

calc l.length = len π l := by rw[hr]
    _ = len π l' := by rw[h_eq]
    _ = l'.length := by rw[hr']
```

□

4.5 Reflections and inversions

We aim to understand the operation of products within Coxeter groups. As noted, the generators are crucial. However, many of their characteristics can be extended to a larger category.

Definition 4.10. The set of **reflections** of a Coxeter system (W, S) is

$$T := \{wsw^{-1} \ : \ s \in S, \ w \in W\}$$

To formalize this we first define a proposition stating that $t \in W$ is a reflection.

mathlib

```
def CoxeterSystem.IsReflection (cs : CoxeterSystem M W) (t : W) : Prop :=
    ∃ w i, t = w * s i * w⁻¹
```

Then, as we do mathematically, the set of reflections is defined as those elements of W that satisfy this property.

```
def T : Type := {t : W // IsReflection cs t}
```

It is clear from the definition that $S \subseteq T$ and that $t^2 = 1$ for every $t \in T$.

mathlib

```
theorem CoxeterSystem.isReflection_simple(cs : CoxeterSystem M W) (i : B) :
    cs.IsReflection (cs.simple i)
```

mathlib

```
theorem CoxeterSystem.IsReflection.pow_two
{cs : CoxeterSystem M W} {t : W} (ht : cs.IsReflection t) :
    t ^ 2 = 1
```

Also, the conjugate of a reflection is another reflection.

mathlib

```
theorem CoxeterSystem.IsReflection.conj
{cs : CoxeterSystem M W} {t : W} (ht : cs.IsReflection t) (w : W) :
    cs.IsReflection (w * t * w⁻¹)
```

We introduce an auxiliary definition for conjugation for reflections, to automatically prove that the result is also a reflection.

```
def conj (t : cs.T) (w : W) : cs.T :=
  ⟨w * t.1 * w⁻¹, IsReflection.conj t.2 w⟩
```

Definition 4.11. Let $w \in W$. A left (resp. right) **inversion** $t \in T_L(w)$ (resp. $T_R(w)$) of w is a reflection that decreases the length of w when multiplied on the left (resp. right).

We will focus on the left case, but most properties are analogous for the right inversions. In equation form, $t \in T_L(w)$ if:

$$\widetilde{\text{len}}(t \cdot w) < \widetilde{\text{len}}(w)$$

mathlib

```
def IsLeftInversion (w t : W) : Prop :=
    cs.IsReflection t ∧ len (t * w) < len w
```

A really important class of inversions is:

Definition 4.12. The left inversion sequence of a word $l := s_{i_0} s_{i_1} \cdots s_{i_{p-1}}$ for some $p \in \mathbb{N}$ is the sequence $\hat{T}(l) := (t_k)_{0 \le k < p}$ where:

$$t_k := s_{i_0} s_{i_1} \cdots s_{i_k} \cdots s_{i_1} s_{i_0} \quad \forall 0 \le k < p$$

In mathlib, it is defined inductively, using $\hat{T}(e) = e$ and

$$\hat{T}(i \cdot l) = (t'_k)_{0 \le k < p+1}$$

Where $t'_0 = s_i$ and $t'_k = s_i t_{k-1} s_i$ if $1 \le k < p+1$.

mathlib

```
def leftInvSeq (ω : List B) : List W :=
  match ω with
  | []          => []
  | i :: ω      => s i :: List.map (MulAut.conj (s i)) (leftInvSeq ω)
```

Then our original definition is recovered in

mathlib

```
theorem CoxeterSystem.getD_leftInvSeq
(cs : CoxeterSystem M W) (ω : List B) (j : ℕ) :
    (cs.leftInvSeq ω).getD j 1 =  cs.wordProd (List.take j ω) *
    (Option.map cs.simple (ω.get? j)).getD 1 *
    (cs.wordProd (List.take j ω))⁻¹
```

The expression `(...).getD j d` returns the element at position `j` if it is within bounds and `d` otherwise. `get?` acts similarly, but returns a special value `Option.none` in the second case. This value can be propagated with `Option.map` (it acts as the normal operator `map` otherwise) and finally set to the default value with the last `getD`.

This approach has the benefit of not having to pass around the proofs that the indices are within bounds, but it makes the statements more convoluted and inductive proofs harder (we have to consider the out-of-bounds cases too). Therefore, we have opted to use the normal `get` operator while supplying proofs that the index is within bounds. For example, we re-state the previous lemma in this way.

```
theorem get_leftInvSeq (w : List B) (j : Fin w.length) :
  (cs.leftInvSeq w).get ⟨j, by simp⟩ =
  cs.wordProd (List.take j w) * s (w.get ⟨j, by simp⟩) * (cs.wordProd (List.take j
    w))⁻¹ := by

  have h : j < (cs.leftInvSeq w).length := by simp

  rw [← List.getD_eq_get ((cs.leftInvSeq w)) 1 h]
  rw [getD_leftInvSeq]
  simp
```

With this formulation, j is guaranteed to be less than the length of w. Note that sometimes the proofs have to be adapted to the specific list. For example, to use `(cs.leftInvSeq w).get` we need to prove that `j < (cs.leftInvSeq w).length`, but in most cases this proof is automatic with `⟨j, by simp⟩`

The left inverse sequence of alternating words is especially important.

Theorem 4.11. *Let $i, j \in B$ and $p, k \in \mathbb{N}$ with $k < 2p$. Let $(t_k)_{1 \le k \le 2p} := \hat{T}(a_{2p}(i, j))$. Then,*

$$t_k = \pi(a_{2k+1}(j, i))$$

```
theorem alternatingWord_of_get_leftInvSeq_alternatingWord
(i j : B) (p : ℕ) (k : ℕ) (h : k < 2 * p) :
  (leftInvSeq cs (alternatingWord i j (2 * p))).get ⟨k, by simp; linarith ⟩ =
  π alternatingWord j i (2 * k + 1)
```

The proof is really manual, proving an induction relation by using the the explicit descriptions of alternating words and left inversion sequences.

Lemma 4.12. *Let $i, j \in B$ and $p, k \in \mathbb{N}$ with $k < 2p$. Let $(t_k)_{1 \leq k \leq 2p} := \hat{T}(a_{2p}(i, j))$. Then, the left inverse sequence repeats in the following way:*

$$t_{M(i,j)+k} = t_k$$

```
lemma leftInvSeq_repeats : ∀ (k : ℕ) (h : k < M i j),
    (cs.leftInvSeq (alternatingWord i j (2 * M i j))).get
        ⟨M i j + k, (by simp[h]; linarith)⟩
    =
    (cs.leftInvSeq (alternatingWord i j (2 * M i j))).get
        ⟨k, (by simp[h]; linarith)⟩
```

The proof consists on using the explicit definition (formalised in Theorem 4.11) and the previous lemma to show the result by direct means. Both of these proofs can be found in the code repository.

Definition 4.13. Let l be a word and t be a reflection. Then, $n(l, t)$ is the amount of times that t appears in $\hat{T}(l)$.

```
def nReflectionOccurrences (l : List B) (t : cs.T) : ℕ :=
  (cs.leftInvSeq l).count t.1
```

A lot of times we are only interested in the parity of this number, which motivates the following definition:

Definition 4.14. Let l be a word and t a reflection. Then,

$$\eta(l, t) = n(l, t) \pmod 2 \in \mathbb{Z}/2\mathbb{Z}$$

```
def parityReflectionOccurrences (w : List B) (t : cs.T) : ZMod 2 :=
  (nReflectionOccurrences cs w t : ZMod 2)
```

As a special case, if $l = i$, then the only element in $\hat{T}(l)$ is s_i, so we have

$$\eta(i, t) = \begin{cases} 1 & \text{if } t = s_i \\ 0 & \text{if } t \neq s_i \end{cases}$$

We define this case separately in Lean, and we will later link it to the original definition.

```
def eta (i : B) (t : cs.T) : ZMod 2 :=
  if (s i = t.1) then 1 else 0
```

Lemma 4.13. *For every $t \in T$ and $i \in B$,*

$$\eta(i, t) = \eta(i, s_i t s_i^{-1})$$

```
lemma eta_simpleConj_eq_eta (i : B) (t : cs.T) :
    eta cs i t = eta cs i (cs.conj t (s i))
```

Proof. First, we unfold the definition of `eta` and `t`.

```
simp [eta]
rcases t with ⟨t, ht⟩
```

Then the goal becomes

```
(if cs.simple i = ⟨t, ht⟩ then 1 else 0) =
if cs.simple i = (cs.conj ⟨t, ht⟩ (cs.simple i)) then 1 else 0
```

We proceed by proving that $s_i = t \iff s_i * t = 1$. From left to right, we apply a simple substitution plus the fact that $t^2 = 1$ (`IsReflection.mul_self ht`). From right to left, we multiply by t on both sides and apply $t^2 = 1$ again.

```
have : s i = t ↔ s i * t = 1 := by
    constructor
    · intro h'
      rw [h']
      exact IsReflection.mul_self ht
    · intro h'
      apply (mul_left_inj t).mp
      simp [IsReflection.mul_self ht]
      exact h'
```

Then, we split the cases $s_i = t$ and $s_i \neq t$, and just supplying the `simp` tactic with the previous lemmas is enough for it to figure out the result, that if $s_i = t$ then $s_i t s_i^{-1} = s_i^{-1} = s_i$ by the previous equivalence and $s_i^2 = 1$.

```
by_cases h : s i = t
  · simp [this, h, conj]
  · simp [this, h, if_neg, conj]
```

□

Lemma 4.14. *Let $t \in T$ and $i, j \in B$. Then,*

$$\eta(a_{2M(i,j)}(i,j), t) = 0$$

```
lemma parityReflectionOccurrences_braidWord (t : cs.T) :
  parityReflectionOccurrences cs (alternatingWord i j (2 * M i j)) t = 0
```

Proof. By definition of η as the projection of n to $\mathbb{Z}/2\mathbb{Z}$, it suffices to prove that the number of occurrences $n(a_{2M(i,j)}(i,j), t)$ is even.

```
suffices Even (nReflectionOccurrences cs (alternatingWord i j (2 * M i j)) t)
    from by
    simp[this, parityReflectionOccurrences]
    apply ZMod.eq_zero_iff_even.mpr this
```

We show this by pairing up the elements in $\hat{T}(a_{2M(i,j)}(i,j))$, according to the relations in Lemma 4.12. This is done in another technical result `nReflectionOccurrences_even_braidWord`

□

Definition 4.15. For every $i \in B$, the permutation map $P_i : T \times \mathbb{Z}/2\mathbb{Z} \to T \times \mathbb{Z}/2\mathbb{Z}$ sends

$$(t, z) \mapsto P_i(t, z) = (s_i t s_i, z + \eta(s, t))$$

```
def permutationMap (i : B) : cs.T × ZMod 2 → cs.T × ZMod 2 :=
  fun (t , z) => (cs.conj t (s i), z + eta cs i t)
```

Theorem 4.15. *For every $i \in B$, $P_i^2 = \mathrm{id}$*

```
theorem permutationMap_orderTwo (i : B) :
    permutationMap cs i ∘ permutationMap cs i = id
```

Proof. To demonstrate the equality of two functions in Lean, we employ the `funext` tactic, which reformulates the objective to show that the functions yield the same result for each input. In our case,

```
funext ⟨t, z⟩
simp [permutationMap]
```

turns the goal into

```
⊢ cs.conj (cs.conj t (cs.simple i)) (cs.simple i) = t ∧
z + cs.eta i t + cs.eta i (cs.conj t (cs.simple i)) = z
```

That is, we want to prove that $s_i(s_i t s_i^{-1})s_i^{-1} = t$ and $z + \eta(i,t) + \eta(i, s_i t s_i^{-1}) = z$. We split both goals with

```
constructor
  · (...)
  · (...)
```

For the first half, we simply use $s_i^2 = 1$ and the associativity of multiplication (plus unfolding the definition of `conj`).

```
simp[conj, mul_assoc]
```

For the second half, we use Lemma 4.13 to turn the goal into $z + 2 \cdot \eta(i,t) = z$ and prove it by showing $2 = 0$ in $\mathbb{Z}/2\mathbb{Z}$ with the `rfl` tactic after some manipulations.

```
rw [← eta_simpleConj_eq_eta cs i t]
ring_nf
simp
right
rfl
```

□

4.6 Coxeter lifts

Our objective is to prove the following lifting theorem:

Theorem 4.16. *The map $(i \in B) \mapsto P_i$ extends to an homomorphism $(w \in W) \mapsto P_w$ such that if $w = s_{i_1} s_{i_2} \cdots s_{i_p}$, then*

$$P_w = P_{i_1} \circ P_{i_2} \circ \cdots \circ P_{i_p}$$

mathlib provides a really useful theorem for constructing lifts like this:

Theorem 4.17. *Let G be any monoid and $f : B \to G$ be a map that satisfies*

$$(f(i) \cdot f(j))^{M(i,j)} = 1 \tag{4}$$

Then there exists a homomorphism $\tilde{f} : W \to G$ extending f, that is,

$$\tilde{f}(s_{i_1} s_{i_2} \cdots s_{i_p}) = f(i_1) \cdot f(i_2) \cdots f(i_p) \tag{5}$$

The property Eq. (4) is formalised as

mathlib

```
def _root_.CoxeterMatrix.IsLiftable
{G : Type*} [Monoid G] (M : CoxeterMatrix B) (f : B → G) : Prop :=
∀ i i', (f i * f i') ^ M i i' = 1
```

Then, for any f that satisfies `IsLiftable`, we define its lift with

mathlib

```
def CoxeterSystem.lift (cs : CoxeterSystem M W) {G : Type u_5} [Monoid G] :
    { f : B → G // M.IsLiftable f } ≃ (W →* G)
```

The lift property Eq. (5) comes from the fact that the lift is a monoid homomorphism plus the following theorem

mathlib

```
theorem CoxeterSystem.lift_apply_simple (cs : CoxeterSystem M W)
{G : Type u_5} [Monoid G] {f : B → G} (hf : M.IsLiftable f) (i : B) :
    (cs.lift ⟨f, hf⟩) (cs.simple i) = f i
```

In our case, the monoid G will be the functions $T \times \mathbb{Z}/2\mathbb{Z} \to T \times \mathbb{Z}/2\mathbb{Z}$. For this, we define the multiplication as the composition and the element one as the identity function, and we prove the following facts for all $f, g, h \in T \times \mathbb{Z}/2\mathbb{Z} \to T \times \mathbb{Z}/2\mathbb{Z}$:

1. $1 \cdot f = f$, that is, $\mathrm{id} \circ f = f$
2. $f \cdot 1 = f$, that is, $f \circ \mathrm{id} = f$
3. $f \cdot (g \cdot h) = (f \cdot g) \cdot h$, that is, $f \circ (g \circ h) = (f \circ g) \circ h$

All of these properties are immediate from the properties of function composition.

```
instance instMul : Mul (cs.T × ZMod 2 → cs.T × ZMod 2) where
  mul := fun f g => f ∘ g

lemma mulDef (f g : cs.T × ZMod 2 → cs.T × ZMod 2) : f * g = f ∘ g := rfl

instance : Monoid (cs.T × ZMod 2 → cs.T × ZMod 2) where
  one := id
  mul := (instMul cs).mul
  one_mul := by
    intro f
    funext x
    suffices (id ∘ f) x = f x from by
      rw[← this]
      rfl
    simp
  mul_one := by
    intro f
    funext x
    suffices (f ∘ id) x = f x from by
      rw[← this]
      rfl
    simp
  mul_assoc := by
    intro f g h
    funext x
    repeat rw[mulDef]
    rfl
```

Then, we define inductively the permutation map of a word.

Definition 4.16. Let $l = i_0 i_1 \cdots i_{p-1} \in B^*$ for some $p \in \mathbb{N}$ be a word. Then,

$$P_l := P_{i_0} \cdot P_{i_1} \cdots P_{i_{p-1}}$$

```
def permutationMap_ofList (l : List B) : cs.T × ZMod 2 → cs.T × ZMod 2 :=
  match l with
  | [] => id
  | a :: t => permutationMap cs a * permutationMap_ofList t
```

We can give an explicit desciption of this function.

Theorem 4.18. *Let $l = i_0 i_1 \cdots i_{p-1} \in B^*$ for some $p \in \mathbb{N}$ be a word. Then for every $t \in T$, $z \in \mathbb{Z}/2\mathbb{Z}$,*

$$P_l(t,z) = (\pi(l) \cdot t \cdot \pi(l)^{-1}, z + \eta(l^{-1}, t)) = (s_{i_0} \cdots s_{i_{p-1}} t s_{i_{p-1}} \cdots s_{i_0}, z + \eta(i_{p-1} \cdots i_0, t))$$

```
lemma permutationMap_ofList_mk (l : List B) (t : cs.T) (z : ZMod 2) :
  (permutationMap_ofList cs l ⟨t,z⟩) = ⟨cs.conj t (π l),
   z + parityReflectionOccurrences cs l.reverse t⟩
```

Proof. We will prove the theorem in parts by first focusing on the first coordinate. When we have an element of a Cartesian product $x \in A \times B$, like $P_l(t,z) \in T \times \mathbb{Z}/2\mathbb{Z}$, we can access the first coordinate with `x.1` and similarly for the second. We will denote this by x_1 and x_2 in mathematical language.

```
lemma permutationMap_ofList_mk_1 (l : List B) :
  (permutationMap_ofList cs l ⟨t,z⟩).1 = cs.conj t (π l)
```

After unfolding the definition of `conj`, we proceed by induction on l.

If $l = e$, then both P_l and conjugation by $\pi(l) = e$ act as the identity, so it is enough to unfold all the definitions.

```
simp[permutationMap_ofList, permutationMap, nReflectionOccurrences]
```

Otherwise, we substitute l with $a :: l$. We can remove the head as

$$(P_{a::l}(t,z))_1 = ((P_a \cdot P_l)(t,z))_1 = (P_a(P_l(t,z)))_1$$

```
calc
(permutationMap_ofList cs (a :: l) (t, z)).1 =
((permutationMap cs a * permutationMap_ofList cs l) (t, z)).1 := by
    simp[permutationMap_ofList]
_ = (permutationMap cs a (permutationMap_ofList cs l (t, z))).1 := by
    rfl
```

Then, simplify with the induction hypothesis ($(P_l(t,z))_1 = \pi(l) \cdot t \cdot \pi(l)^{-1}$). Lean is able to figure out that if $P_l = (P_l^1, P_l^2)$, then $(P_a(P_l(t,z)))_1 = (P_a^1(P_l(t,z))_1)$ (since $(P_a(t,z))_1$ only depends on t) to get $P_{a::l}(t,z) = P_a(\pi(l) \cdot t \cdot \pi(l)^{-1})$

```
simp[permutationMap, conj, h]
```

So we have to show that $s_a \pi(l) t \pi(l)^{-1} s_a = \pi(a :: l) t \pi(a :: l)^{-1}$

```
⊢ ⟨cs.simple a * (cs.wordProd l * ↑t * (cs.wordProd l)⁻¹) * cs.simple a, ...⟩ =
  ⟨cs.wordProd (a :: l) * ↑t * (cs.wordProd (a :: l))⁻¹, ...⟩
```

This is solved with one of the properties of π and associativity

```
simp[cs.wordProd_cons, mul_assoc]
```

□

On to the second part, we want to show that $(P_l(t,z))_2 = \eta(l^{-1}, t)$

```
lemma permutationMap_ofList_mk_2 (l : List B) :
 (permutationMap_ofList cs l ⟨t,z⟩).2 = z + parityReflectionOccurrences cs
    l.reverse t
```

Again we proceed by induction on l, with the base case being trivial. Otherwise we substitute l by $a :: l$, and then substitute

$$(P_{a::l}(t,z))_2 = (P_a(P_l(t,z)))_2 = (P_a(P_l^1(t,z), z + \eta(l^{-1}, t)))_2$$

So after applying P_a, the goal becomes

$$z + \eta(l^{-1}, t) + \eta(i, P_l^1(t,z)) = z + \eta(l^{-1} \cdot i, t) \tag{6}$$

```
rw[permutationMap_ofList, mulDef]
simp[permutationMap, h]

⊢ z + cs.parityReflectionOccurrences l.reverse t +
    cs.eta i (cs.permutationMap_ofList l (t, z)).1 =
  z + cs.parityReflectionOccurrences (l.reverse ++ [i]) t
```

Now we convert the append expression `l.reverse ++ [i]` into concat and apply the lemma `leftInvSeq_concat` which tells us that

$$\hat{T}(l^{-1} \cdot i) = \hat{T}(l^{-1}) + +[\pi(l)^{-1} \cdot s_i \cdot \pi(l)]$$

Here, $++$ means concatenation of lists/sequences, like in Lean.

```
rw[← List.concat_eq_append]
rw[leftInvSeq_concat]
```

But then, the amount of times that t appears in $\hat{T}(l \cdot i)$ is

$$\eta(l^{-1} \cdot i, t) = \eta(l^{-1}, t) + \begin{cases} 1 & \text{if } t = \pi(l)^{-1} \cdot s_i \cdot \pi(l) \\ 0 & \text{otherwise} \end{cases}$$

Since $\eta(l^{-1}, t)$ is the number of times it appears in $\hat{T}(l^{-1})$.

```
simp [List.count_singleton]
```

Therefore, it suffices to show that

$$\eta(i, P_l^1(t, z)) = \begin{cases} 1 & \text{if } t = \pi(l)^{-1} \cdot s_i \cdot \pi(l) \\ 0 & \text{otherwise} \end{cases}$$

since substituting this in Eq. (6) plus the previous observation yields the result. We formalise this in Lean with the following tactic. We write `suffices h from (...)` and prove our original goal in the expression body of `from`, using the hyphothesis `h` (accessible as `this` from there). Subsequently, `h` becomes the new goal. In our case,

```
suffices cs.eta i (permutationMap_ofList cs l (t, z)).1 = if (cs.wordProd l)⁻¹ *
    cs.simple i * cs.wordProd l = t.1 then 1 else 0 from by
      rw[this]
      simp[add_assoc]
```

To prove this we use the first part of the theorem and the definition of η to transform the goal into

$$\begin{cases} 1 & \text{if } s_i = \pi(l) \cdot t \cdot \pi(l)^{-1} \\ 0 & \text{otherwise} \end{cases} = \begin{cases} 1 & \text{if } t = \pi(l)^{-1} \cdot s_i \cdot \pi(l) \\ 0 & \text{otherwise} \end{cases}$$

```
simp[eta, permutationMap_ofList_mk_1, conj]

⊢ (if cs.simple i = cs.wordProd l * ↑t * (cs.wordProd l)⁻¹ then 1 else 0) =
  if (cs.wordProd l)⁻¹ * cs.simple i * cs.wordProd l = ↑t then 1 else 0
```

The two conditions are clearly equivalent by simple multiplication of $\pi(l)$ and its inverse. In Lean, we finish the proof by cases.

```
by_cases h' : (cs.wordProd l)⁻¹ * cs.simple i * cs.wordProd l = t.1
· simp[h']
  rw[← h']
  simp[mul_assoc]
· simp[h']
  by_contra h''
  rw[h''] at h'
  simp[mul_assoc] at h'
```

Now we are ready to prove the lifting condition of the permutation map.

Theorem 4.19. *The map $i \mapsto P_i$ satisfies the lifting condition of Theorem 4.17. That is,*

$$(P_i \cdot P_j)^{M(i,j)} = 1$$

```
theorem permutationMap_isLiftable : M.IsLiftable (cs.permutationMap)
```

Proof. We start by unfolding the definition of the liftable property by introducing the arbitrary $i, j \in B$.

```
intro i j
```

First, we want to prove that, for any $p \in \mathbb{N}$,

$$(P_i \cdot P_j)^p = P_{a_{2p}(i,j)}$$

We do so by induction on p.

```
have h (p : ℕ) : (cs.permutationMap i * cs.permutationMap j) ^ p =
    permutationMap_ofList cs (alternatingWord i j (2 * p)) := by
    induction p with
    | zero =>
      simp[permutationMap_ofList, permutationMap, permutationMap_orderTwo]
      rfl
    | succ p h => (...)
```

As we can see above, the initial case is trivial once the definitions unfold, since both $(P_i \cdot P_j)^0 = 1$ by the definition of the product and $P_{a_0(i,j)} = P_e = 1$ by definition of the permutation map.

When $p > 0$, we substitute it with $p + 1$. Then, we substitute $(P_i \cdot P_j)^{p+1} = (P_i \cdot P_j) \cdot (P_i \cdot P_j)^p$ and apply the induction hypothesis to transform the goal into

$$(P_i \cdot P_j) \cdot P_{a_{2p}(i,j)} = P_{a_{2p+2}(i,j)}$$

We prove this by using that $P_{a_{2p+2}(i,j)} = P_{i \cdot j \cdot a_{2p}(i,j)} = P_i \cdot P_j \cdot P_{a_{2p}(i,j)}$ In Lean this works by applying the inductive definition of alternating words, with the fact that $2p+2 = 2p+1+1$, and taking care of the parity of the head.

```
have : 2 * (p + 1) = 2 * p + 1 + 1 := by ring
rw[this]
rw[alternatingWord_succ']
rw [if_neg (Nat.not_even_bit1 p)]
rw[permutationMap_ofList]
```

```
rw[alternatingWord_succ']
rw [if_pos (even_two_mul p)]
rw[permutationMap_ofList]
simp[mul_assoc]
```

We apply this auxiliary result with $p = M(i,j)$ and introduce an arbitrary $t \in T$ and $z \in \mathbb{Z}/2\mathbb{Z}$ to transform the goal into

$$P_{a_{2M(i,j)}(i,j)}(t,z) = (t,z)$$

```
rw[h (M i j)]
funext ⟨t, z⟩
```

Now apply Theorem 4.18 to get the goal

$$(\pi(a_{2M(i,j)}(i,j)) \cdot t \cdot \pi(a_{2M(i,j)}(i,j))^{-1} = t) \wedge (\eta(a_{2M(i,j)}(i,j)^{-1}, t) = 0)$$

```
simp[permutationMap_ofList_mk, conj]

⊢ ⟨cs.wordProd (alternatingWord i j (2 * M.M i j)) * ↑t * (cs.wordProd
    (alternatingWord i j (2 * M.M i j)))⁻¹, ...⟩ = t ∧
  cs.parityReflectionOccurrences (alternatingWord i j (2 * M.M i j)).reverse t = 0
```

For the left equality, recall that $\pi(a_{2M(i,j)}) = (s_i \cdot s_j)^{M(i,j)} = 1$, which yields the result.

```
simp[cs.prod_alternatingWord_eq_mul_pow]
```

For the right equality, we use the properties of alternating words and Coxeter matrices to get that $a_{2M(i,j)}(i,j)^{-1} = a_{2M(i,j)}(j,i) = a_{2M(j,i)}(j,i)$. Finally, apply Lemma 4.14 to get the result.

```
rw[alternatingWord_reverse]
rw[M.symmetric]
exact parityReflectionOccurrences_braidWord cs t
```

□

With this, we can finally define the permutation map for any $w \in W$.

Proof of Theorem 4.16. As we just proved in Theorem 4.19, the permutation map over a word $l \in B^*$ satisfies the Coxeter relations, so it can be defined for an element $w \in W$ using any expression as representative. In Lean, we define it with

```
def permutationMap_lift : W →* cs.T × ZMod 2 → cs.T × ZMod 2 :=
  cs.lift ⟨cs.permutationMap, permutationMap_isLiftable cs⟩
```

And to prove that it agrees with the definition with a word ($P_{\pi(l)} = P_l$) as in Eq. (5), we do so by induction on l. We extract the head of the word (list) on both sides using the definition of `permutationMap_ofList` and that the lift of P is a homomorphism. Then, the induction hypothesis plus the lemma `CoxeterSystem.lift_apply_simple` complete the proof.

```
theorem permutationMap_lift_mk_ofList (l : List B) (t : cs.T) (z : ZMod 2) :
  permutationMap_lift cs (cs.wordProd l) ⟨t,z⟩ = permutationMap_ofList cs l ⟨t,z⟩
    := by
  induction l with
  | nil =>
    simp[permutationMap_lift, cs.wordProd_nil, permutationMap_ofList]
    rfl
  | cons i l h =>
    rw[cs.wordProd_cons]
    rw[permutationMap_ofList]
    simp[mulDef]
    rw[← h]
    simp[permutationMap_lift]
```

□

We now get this result for free:

Theorem 4.20. *1. η can be defined in $W \times T$ by lifting the original definition.*

2. Let $l, l' \in B^*$ and $t \in T$ with $\pi(l) = \pi(l')$. Then,

$$\eta(l,t) = \eta(l',t)$$

Proof. For the first part, let $w \in W$ and $t \in T$ We lift η by taking the second coordinate of P_w.

```
def parityReflectionOccurrences_lift (w : W) (t : cs.T) : ZMod 2 :=
  (permutationMap_lift cs w⁻¹ ⟨t,0⟩).2
```

If $w = \pi(l)$, $\eta(w,t) = \eta(l,t)$ because $P_{w^{-1}}(t,0) = P_{l^{-1}}(t,0)$ by Theorem 4.16, which means that their second coordinates agree. The second coordinate of $P_{w^{-1}}(t,0)$ is $\eta(w,t)$ by definition and the second coordinate of $P_{l^{-1}}(t,0)$ is $\eta(l,t)$ by Theorem 4.18.

```
theorem parityReflectionOccurrences_lift_mk (l : List B) (t : cs.T) :
  parityReflectionOccurrences_lift cs (cs.wordProd l) t =
    parityReflectionOccurrences cs l t := by
  rw[parityReflectionOccurrences_lift]
  rw[← wordProd_reverse]
  rw[permutationMap_lift_mk_ofList cs l.reverse t 0]
  rw[permutationMap_ofList_mk cs l.reverse t 0]
  simp
```

The second part of the theorem is a direct consequence of the first;

$$\eta(l,t) = \eta(\pi(l),t) = \eta(\pi(l'),t) = \eta(l',t)$$

```
theorem parityReflectionOccurrences_ext (l l' : List B) (t : cs.T) (h : π l = π
    l') :
  parityReflectionOccurrences cs l t = parityReflectionOccurrences cs l' t := by

  calc
    parityReflectionOccurrences cs l t = parityReflectionOccurrences_lift cs
    (cs.wordProd l) t := by rw[parityReflectionOccurrences_lift_mk]
```

```
    _ = parityReflectionOccurrences_lift cs (cs.wordProd l') t := by rw[h]
    _ = parityReflectionOccurrences cs l' t := by
    rw[parityReflectionOccurrences_lift_mk]
```

□

Therefore, the lifted version of Theorem 4.18 is:

Theorem 4.21. *Let $w = s_{i_1}s_{i_2}\cdots s_{i_p} \in W$, $t \in T$ and $z \in \mathbb{Z}/2\mathbb{Z}$. Then,*

$$P_w(t,z) = (w \cdot t \cdot w^{-1}, z + \eta(w^{-1}, t))$$

```
theorem permutationMap_lift_mk (w : W) (t : cs.T) (z : ZMod 2) :
  permutationMap_lift cs w ⟨t,z⟩ = ⟨⟨w * t.1 * w⁻¹, IsReflection.conj t.2 w⟩ , z +
    parityReflectionOccurrences_lift cs w⁻¹ t⟩ := by
  obtain ⟨l, _, rfl⟩ := cs.exists_reduced_word' w
  apply Prod.ext
  · simp[permutationMap_lift_mk_ofList, permutationMap_ofList_mk, conj]
  · simp[parityReflectionOccurrences_lift]
    rw[permutationMap_lift_mk_ofList cs l t 0]
    rw[permutationMap_lift_mk_ofList cs l t z]
    simp[permutationMap_ofList_mk]
```

A very important property is how the permutation map behaves with regards to reflections.

Theorem 4.22. *Let $t \in T$ and $z \in \mathbb{Z}/2\mathbb{Z}$. Then,*

$$P_t(t,z) = (t, z+1)$$

```
lemma permutationMap_lift_of_reflection (t : cs.T) : ∀ (z : ZMod 2),
  permutationMap_lift cs t.1 (t, z) = ⟨t, z + 1⟩
```

Proof. Let $t = w \cdot s_p \cdot w'$ for $w \in W$ and $p = \widetilde{\text{len}}(w)$. We can use Lemma 4.8 to get a reduced expression l of w, such that

$$w = s_0 s_1 \cdots s_{p-1}$$

```
rcases t with ⟨t, t_refl⟩
rcases t_refl with ⟨w, p, rfl⟩
obtain ⟨l, _, rfl⟩ := cs.wordProd_surjective w
```

Then, $t = s_0 s_1 \cdots s_{p-1} \cdots s_1 s_0$ and we proceed by induction on l. The case where $l = e$ is clear. Otherwise,

$$\begin{aligned}
P_{s_0\cdots s_{p-1}\cdots s_0}(s_0\cdots s_{p-1}\cdots s_0, z) = & \\
&= P_{s_0}P_{s_1\cdots s_{p-1}\cdots s_1}(s_1\cdots s_{p-1}\cdots s_1, z + \eta(s_0; s_0\cdots s_{p-1}\cdots s_0)) \\
&= P_{s_0}(s_0\cdots s_{p-1}\cdots s_0, z + 1 + \eta(s_0; s_1\cdots s_{p-1}\cdots s_1)) \\
&= (s_0\cdots s_{p-1}\cdots s_0, z + 1 + 2\cdot\eta(s_0; s_1\cdots s_{p-1}\cdots s_1)) \\
&= (t, z+1)
\end{aligned}$$

The formalization is straightforward; we follow the procedures used in previous proofs with the explicit description of the permutation map. For additional information, refer to the Lean code. [Ála] □

4.7 The Strong Exchange Theorem

Now we are ready to state and prove one key theorem of Coxeter groups.

Definition 4.17. Let $k, p \in \mathbb{N}$ such that $k < p$ We define the erase at k function as

$$e_k : \{l \in B^* : \text{len}(l) > 0\} \longrightarrow B^*, \qquad i_0 \cdots i_{k-1} i_k i_{k+1} \cdots i_{p-1} \mapsto i_0 \cdots i_{k-1} i_{k+1} \cdots i_{p-1}$$

A lot of times we will use the more expressive notation $i_0 i_1 \cdots \hat{i_k} \cdots i_{p-1}$. As expected, we give an alternative inductive definition in Lean.

$$e_k(l) = \begin{cases} e & \text{if } l = e \\ l' & \text{if } l = i_0 \cdot l' \text{ and } k = 0 \text{ for some } i_0 \in B, l' \in B^* \\ i_0 \cdot e_{k'}(l') & \text{if } l = i_0 \cdot l' \text{ and } k = k' + 1 \text{ for some } i_0 \in B, l' \in B^* \end{cases}$$

mathlib

```
def eraseIdx : List α → Nat → List α
  | [],    _   => []
  | _::as, 0   => as
  | a::as, n+1 => a :: eraseIdx as n
```

Theorem 4.23 (Strong Exchange Theorem)**.** *Let $l = i_0 i_1 \cdots i_{p-1} \in B^*$ for some $p \in \mathbb{N}$ and $t \in T_L(\pi(l))$. Then, there exists some $k < p$ such that:*

$$t \cdot \pi(i_0 i_1 \cdots i_k \cdots i_{p-1}) = \pi(e_k(l)) = \pi(i_0 i_1 \cdots \hat{i_k} \cdots i_{p-1})$$

In Lean, we accomplish the deletion with the function `eraseIdx` as such:

```
theorem strongExchangeProperty (l : List B) (t : cs.T)
(h' : cs.IsLeftInversion (cs.wordProd l) t.1) :
  ∃ (k : Fin l.length), t.1 * π l = π (l.eraseIdx k)
```

First, we prove it for a specific case.

Lemma 4.24. *Let $l = i_0 i_1 \cdots i_{p-1} \in B^*$ for some $p \in \mathbb{N}$ and $t \in \hat{T}(l)$. Then, there exists some $0 \leq k < p$ such that:*

$$t \cdot \pi(i_0 i_1 \cdots i_k \cdots i_{p-1}) = \pi(i_0 i_1 \cdots \hat{i_k} \cdots i_{p-1})$$

```
lemma eraseIdx_of_mul_leftInvSeq (l : List B) (t : cs.T)
(h : t.1 ∈ cs.leftInvSeq l) :
  ∃ (k : Fin l.length), t.1 * π l = π (l.eraseIdx k)
```

Proof. First, we get that there is some $0 \leq k < p$ such that $t = t_k$, where $\hat{T}(l) = (t_k)_k$.

```
have : ∃ (k : Fin (cs.leftInvSeq l).length), (cs.leftInvSeq l).get k = t.1 :=
      List.get_of_mem h
rcases this with ⟨k, hk⟩
```

Then, we state that this k is the one appearing on the lemma statement. We need to do some manipulations to the restrictions on k (basically proving that `(cs.leftInvSeq l).length = l.length`).

```
use ⟨k, by rw[← length_leftInvSeq cs l] ; exact k.2⟩
```

Then we substitute t by the specific t_k and apply mathlib's lemma

mathlib

```
theorem getD_leftInvSeq_mul_wordProd (ω : List B) (j : ℕ) :
    ((lis ω).getD j 1) * π ω = π (ω.eraseIdx j)
```

Afterwards, a simple convertion from `getD` to `get` yields the result.

```
simp
rw[← List.get?_eq_getElem?]
rw [List.get?_eq_get k.2]
simp
```

□

Lemma 4.25. *Let $l = i_0 i_1 \cdots i_{p-1} \in B^*$ for some $p \in \mathbb{N}$ and $t \in \hat{T}(l)$. Then, there exists some $0 \leq k < p$ such that:*

$$t \cdot \pi(i_0 i_1 \cdots i_k \cdots i_{p-1}) = \pi(i_0 i_1 \cdots \hat{i_k} \cdots i_{p-1})$$

```
lemma eraseIdx_of_mul_leftInvSeq (l : List B) (t : cs.T)
(h : t.1 ∈ cs.leftInvSeq l) :
  ∃ (k : Fin l.length), t.1 * π l = π (l.eraseIdx k)
```

Proof. We begin by finding k such that $t = t_k$, where t_k is an element of $\hat{T}(l)$. This follows from the fact that $t.1$ belongs to cs.leftInvSeq(l), meaning t corresponds to some index k.

```
have : ∃ (k : Fin (cs.leftInvSeq l).length), (cs.leftInvSeq l).get k = t.1 :=
      List.get_of_mem h
rcases this with ⟨k, hk⟩
```

Next, we identify the index k and verify that the length of cs.leftInvSeq(l) matches the length of l. This allows us to correctly apply the index k in l.

```
use ⟨k, by rw[← length_leftInvSeq cs l] ; exact k.2⟩
```

Subsequently, we substitute t with t_k using the previously obtained k, and apply the lemma `getD_leftInvSeq_mul_wordProd`, which establishes the desired multiplication and erasure.

```
rw[← hk]
rw[← getD_leftInvSeq_mul_wordProd cs l k]
```

Finally, we simplify the expression and apply the conversion from `getD` to `get` using standard list operations.

```
simp
rw[← List.get?_eq_getElem?]
rw [List.get?_eq_get k.2]
simp
```

□

Lemma 4.26. *Let $l \in B^*$ and $t \in T$ such that $\eta(l,t) = 1$. Then, $t \in \hat{T}(l)$.*

```
lemma isInLeftInvSeq_of_parityReflectionOccurrences_eq_one
(l : List B) (t : cs.T) (h : parityReflectionOccurrences cs l t = 1) :
    t.1 ∈ cs.leftInvSeq l
```

Proof. We begin by observing that $\eta(l,t) = 1$ is equivalent to $n(l,t)$ being odd, by definition of η.

```
simp [parityReflectionOccurrences] at h
rw [← @odd_iff_parity_eq_one (nReflectionOccurrences cs l t)] at h
```

Then, since it's an odd natural number it must be at least one. We prove this in an auxilliary lemma in Lean.

```
lemma gt_one_of_odd (n : ℕ) : Odd n → n > 0 := by
  intro h
  simp[Odd] at h
  rcases h with ⟨m, rfl⟩
  suffices m ≥ 0 from by linarith
  exact Nat.zero_le m
```

When simplifying `Odd` we get that $\exists m \in \mathbb{N}$ such that $n = 2m+1$ and the resulting inequality is easily solved by the `linarith` tactic with the fact that $n \geq 0$.

So we apply it, and simplifying the definition of $n(l,t)$ (amount of times that t appears in $\hat{T(l)}$) completes the proof.

```
apply gt_one_of_odd (nReflectionOccurrences cs l t) at h
simp[nReflectionOccurrences] at h
exact h
```

□

There are two slight variations of the last lemma that are worth noting.

Lemma 4.27. *Let $l \in B^*$ and $t \in T$ such that $\eta(l,t) = 1$. Then, t is a left inversion of $\pi(l)$.*

```
lemma isLeftInversion_of_parityReflectionOccurrences_eq_one
(l : List B) (t : cs.T) :
  parityReflectionOccurrences cs l t = 1 →
  cs.IsLeftInversion (cs.wordProd l) t.1
```

Proof. While it seems obvious from the previous result, we have to take care since only elements of the left inversion sequence of a reduced word are guaranteed to be left inversions (otherwise its terms can cancel out). But we can find a reduced representative of $\pi(l)$ and apply Theorem 4.20 to circumvent this.

```
intro h

rcases cs.exists_reduced_word' (π l) with ⟨u, u_reduced, hu⟩
```

```
rw[hu]
apply cs.isLeftInversion_of_mem_leftInvSeq u_reduced

rw[cs.parityReflectionOccurrences_ext l u t hu] at h
exact isInLeftInvSeq_of_parityReflectionOccurrences_eq_one cs u t h
```

□

Similarly, we can prove it for the lifted version of η.

Lemma 4.28. *Let $w \in W$ and $t \in T$ such that $\eta(w,t) = 1$. Then, t is a left inversion of w.*

```
lemma isLeftInversion_of_parityReflectionOccurrences_lift_eq_one
(w : W) (t : cs.T) :
  parityReflectionOccurrences_lift cs w t = 1 → cs.IsLeftInversion w t.1
```

Proof. The proof just consists of applying the previous lemma to a representant of w.

```
intro h
obtain ⟨l, _, rfl⟩ := cs.exists_reduced_word' w
simp[parityReflectionOccurrences_lift_mk] at h
apply isLeftInversion_of_parityReflectionOccurrences_eq_one cs l t h
```

□

The most important and final lemma for the proof is:

Lemma 4.29. *Let $l \in B^*$ and $t \in T$. Then, t is a left inversion of l $(\mathrm{len}(t \cdot l) < \mathrm{len}(l))$ if and only if $\eta(l,t) = 1$*

```
lemma isLeftInversion_iff_parityReflectionOccurrences_eq_one
(l : List B) (t : cs.T) :
  cs.IsLeftInversion (cs.wordProd l) t.1 ↔
  parityReflectionOccurrences cs l t = 1
```

Proof. We prove both directions;

```
constructor
  · (=>)
  · (<=)
```

The right to left case is Lemma 4.27.

```
exact isLeftInversion_of_parityReflectionOccurrences_eq_one cs l t
```

For the left to right case, we proceed by contradiction.

```
intro h
by_contra h'
```

The `by_contra` tactic introduces the negation of the goal as a hypothesis and sets $\vdash$ `False` as the new goal (finding a contradiction).

Therefore, we assume that $\eta(l,t) \neq 1$, that is, $\eta(l,t) = 0$. This equality requires a bit of work with the `ZMod` properties.

```
have h'' : parityReflectionOccurrences cs l t = 0 := by
  simp [parityReflectionOccurrences]
  rw [ZMod.eq_zero_iff_even]
  simp[parityReflectionOccurrences] at h'
  rw[ZMod.eq_one_iff_odd] at h'
  exact Nat.not_odd_iff_even.mp h'
```

Then, it suffices to prove that $t{\cdot}\pi(l) \in T_L(l)$, that is, $t{\cdot}\pi(l) \in T$ and $\widetilde{\text{len}}(t{\cdot}t{\cdot}\pi(l)) < \widetilde{\text{len}}(\pi(l))$.

```
suffices cs.IsLeftInversion (t.1 * π l) t.1 from by
  simp[IsLeftInversion] at this
  rw[← mul_assoc] at this
  rcases this with ⟨_, ht⟩
```

We will focus on the inequality

```
ht : cs.length (↑t * ↑t * cs.wordProd l) < cs.length (↑t * cs.wordProd l)
```

Since $t \in T$, we have $t^2 = 1$, so it's equivalent to $\widetilde{\text{len}}(\pi(l)) < \widetilde{\text{len}}(t \cdot \pi(l))$.

```
rw[IsReflection.mul_self t.2] at ht
simp at ht
```

But also $t \in T_L(l)$, so $\widetilde{\text{len}}(t \cdot \pi(l)) < \widetilde{\text{len}}(\pi(l))$, which is a contradiction.

```
simp[IsLeftInversion] at h
linarith
```

Furthermore, it suffices to prove that

$$P_{(t\cdot\pi(l))^{-1}}(t,0) = (\pi(l)^{-1} \cdot t \cdot \pi(l), 1)$$

Because in that case, matching the second coordinates, we have $\eta(t \cdot \pi(l), t) = 1$, which by Lemma 4.28 means that $t \cdot \pi(l) \in T_L(l)$.

```
suffices permutationMap_lift cs (t.1 * π l)⁻¹ ⟨t, 0⟩ =
    ⟨cs.conj t (π l)⁻¹, 1⟩ from by
  apply isLeftInversion_of_parityReflectionOccurrences_lift_eq_one
    cs (t.1 * π l) t
  rw[permutationMap_lift_mk cs (t.1 * π l)⁻¹ t 0] at this
  simp at this
  simp[this.2]
```

Then, we finish the proof by proving that equality:

$$\begin{aligned}
&P_{(t\cdot\pi(l))^{-1}}(t,0) = \\
&= P_{\pi(l)^{-1}}(P_t(t,0)) \quad \text{(by definition of } P \text{ and } t^{-1} = 1) = \\
&= P_{\pi(l)^{-1}}(t,1) \qquad \text{(by Theorem 4.22)} = \\
&= (\pi(l)^{-1} \cdot t \cdot \pi(l), 1 + \eta(\pi(l),t)) \quad \text{(by applying } P) = \\
&= (\pi(l)^{-1} \cdot t \cdot \pi(l), 1) \quad \text{(by the lift of our absurd assumption)}
\end{aligned} \tag{7}$$

```
calc
  permutationMap_lift cs (t.1 * π l)⁻¹ ⟨t, 0⟩ =
    permutationMap_lift cs (π l)⁻¹ (permutationMap_lift cs t.1 ⟨t, 0⟩) := by
        simp[IsReflection.inv t.2]
        rfl
  _ = permutationMap_lift cs (π l)⁻¹ ⟨t, 1⟩ := by
        rw[permutationMap_lift_of_reflection cs t 0]
        simp[permutationMap_lift_mk]
  _ = ⟨cs.conj t (π l)⁻¹, 1 + parityReflectionOccurrences_lift cs (π l) t⟩ := by
        simp[permutationMap_lift_mk, conj]
  _ = (cs.conj t (cs.wordProd l)⁻¹, 1) := by
        simp
        simp[parityReflectionOccurrences_lift_mk, h'']
```

□

We are finally ready to prove the Strong Exchange Theorem.

Proof of Theorem 4.23. We know that it suffices to prove $t \in \hat{T}(l)$ from Lemma 4.25.

```
suffices t.1 ∈ cs.leftInvSeq l from eraseIdx_of_mul_leftInvSeq cs l t this
```

We replace our hypothesis that t is a left inversion of l with $\eta(l,t) = 1$ using Lemma 4.29 and then use Lemma 4.27 to get that $t \in \hat{T}(l)$, finalizing the proof.

```
rw [isLeftInversion_iff_parityReflectionOccurrences_eq_one cs l t] at h'
exact isInLeftInvSeq_of_parityReflectionOccurrences_eq_one cs l t h'
```

□

4.8 Coxeter moves and Matsumoto's theorem

We have seen a multitude of properties of products in Coxeter groups. However, a fundamental question remains. How do we know if two words correspond to the same product? The definition of Coxeter groups with a group presentation doesn't provide an easy answer, since the space of operations we can perform on words while maintaining their product is infinite.

In this section we will study this process of modifying different expressions of a product with the application of nil and braid moves, culminating in the statement and proof of Matsumoto's theorem.

Definition 4.18.

$$s_{i_0}\cdots s_{i_{p-2}}\overline{s_{i_{p-1}}s_{i_p}}s_{i_{p+1}}\cdots s_{i_{l-1}} \xrightarrow{n[i,p]} \begin{cases} s_{i_0}\cdots s_{i_{p-2}}s_{i_{p+1}}\cdots s_{i_{l-1}} & \text{if } i_{p-1} = i = i_p \\ s_{i_0}\cdots s_{i_{p-2}}s_{i_{p-1}}s_{i_p}s_{i_{p+1}}\cdots s_{i_{l-1}} & \text{otherwise} \end{cases}$$

For example, we can apply a nil move to the word $i_0\overline{i_1 i_1}i_3 i_2 \xrightarrow{n[1,1]} i_0 i_3 i_2$. The two words are obviously not equal, but their products are, because of the Coxeter group relations. $s_{i_0}\overline{s_{i_1}s_{i_1}}s_{i_3}s_{i_2} = s_{i_0}s_{i_3}s_{i_2}$

```
structure NilMove (cs : CoxeterSystem M W) where
  i : B
  p : ℕ
```

And we also have the braid moves.

Definition 4.19.

$$s_{i_0} \cdots s_{i_{p-1}} \cdot b(i,j) \cdots s_{i_{l-1}} \xrightarrow{\beta[i,j,p]} s_{i_0} \cdots s_{i_{p-1}} \cdot b(j,i) \cdots s_{i_{l-1}}$$

$$s_{i_0} \cdots s_{i_{l-1}} \xrightarrow{\beta[i,j,p]} s_{i_0} \cdots s_{i_{l-1}} \quad \text{otherwise}$$

For example, if $M(0,1) = 3$, then

$$i_2\overline{i_0 i_1 i_0} i_3 i_0 \xrightarrow{\beta[0,1,1]} i_2\overline{i_1 i_0 i_1} i_3 i_0 \implies s_{i_2}\overline{s_{i_0}s_{i_1}s_{i_0}}s_{i_3}s_{i_0} = s_{i_2}\overline{s_{i_1}s_{i_0}s_{i_1}}s_{i_3}s_{i_0}$$

```
structure BraidMove (cs : CoxeterSystem M W) where
  i : B
  j : B
  p : ℕ
```

Then we define the notion of a **Coxeter move** as either a nil or a braid move. In lean,

```
inductive CoxeterMove(cs : CoxeterSystem M W) where
| nil : cs.NilMove → cs.CoxeterMove
| braid : cs.BraidMove → cs.CoxeterMove
```

For example, we can say `cs.CoxeterMove.nil nm`, where `nm : cs.NilMove`. And if we have a general coxeter move `cm : cs.CoxeterMove`, we can split between the cases where `cm` is a nil or braid move with the match keyboard:

```
match cm with
| nil nm => ...
| braid bm => ...
```

With this machinery, we are ready to apply these moves to any word. Consider the following function, that takes a nil move and a word and outputs the result of applying the move:

```
def apply_nilMove (nm : cs.NilMove) (l : List B) : List B :=
  match nm with
  | NilMove.mk i p =>
    match p with
    | 0 =>
      if l.take 2 = [i, i] then
        l.drop 2
      else
        l
    | p + 1 =>
      match l with
      | [] => []
      | h::t => h :: apply_nilMove (NilMove.mk i p) t
```

First, we extract the generator index `i` of the move and the position of the move `p` with the instruction `match`. Then, we divide into the cases where `p=0` and where `p` is a successor of another natural number, in which case we substitute `p` with `p+1`.

If `p = 0`, it means that the nil move should be applied at the beginning of the word. Therefore, we check that the first two elements of the word `l` are, in fact, `i`, and in that case we return the list without those first two elements. Otherwise, we return the original word, because the move could not be applied.

If `p` if not zero, we write it as `p + 1` and then act depending on `l`. If `l = []`, we cannot apply the nil move, so we return `[]` (the original list). Otherwise, we consider the list `l` as a first element `h` (the head) and a tail `t`. These two cases make up the entire array of possibilities, since if `l` is not empty it has a first element that can be considered as its head.

In the second case, we take the tail `t`, apply the nil move one position before (at `p` instead of `p - 1`) and return the result with `h` appended as the first element.

The result is that we apply the second case repeatedly until `p=0` at which point we apply the nil move at the beginning of the remaining word and append the first `p` elements back. This has the effect of applying the nil move at position `p`.

Then, to apply a braid move we proceed in a similar way as with nil moves:

```
def apply_braidMove (bm : cs.BraidMove) (l : List B) : List B :=
  match bm with
  | BraidMove.mk i j p =>
    match p with
    | 0 =>
      if l.take (M i j) = braidWord M i j then
        braidWord M j i ++ l.drop (M i j)
      else
        l
    | p + 1 =>
      match l with
      | [] => []
      | h::t => h :: apply_braidMove (BraidMove.mk i j p) t
```

Matsumoto's theorem aims to shed light on the process of converting equivalent words into each other. At first glance, the process of turning one word into an equivalent could involve not only braid and nil moves, but also reverse nil moves (introducing $ii \in B^*$ between two letters of the word). But this theorem assures us that it suffices with the first two, and just braid moves when relating two reduced words.

Theorem 4.30 (Matsumoto [AB05])**.** *Let $w \in W$*

1. *Any expression $s_1 s_2 \cdots s_q$ for w can be transformed into a reduced expression for w by a sequence of nil-moves and braid-moves.*
2. *Every two reduced expressions for w can be connected via a sequence of braid-moves.*

We will prove just the second part, because it is the one we need to show that Demazure operators are well defined.

With our notation, it means that for every two reduced words $l, l' \in B^*$ with $\pi(l) = \pi(l')$, there exists $r \in \mathbb{N}$ and a finite sequence of braid moves $\beta = (\beta_k)_{0 \leq k \leq r-1}$ and a sequence of words $(l_k)_{0 \leq k \leq r}$ such that

$$l = l_0 \xrightarrow{\beta_0} l_1 \xrightarrow{\beta_1} l_2 \xrightarrow{\beta_2} \cdots \xrightarrow{\beta_{r-1}} l_r = l'$$

We abbreviate this by

$$l \xrightarrow{\beta} l'$$

```
def apply_braidMoveSequence (bms : List (cs.BraidMove)) (l : List B) : List B :=
  match bms with
  | [] => l
  | bm :: bms' => cs.apply_braidMove bm (apply_braidMoveSequence bms' l)
```

Also, if we have two braid move sequences $\beta^{(1)} = (\beta^{(1)}_k)_{0 \leq k \leq r-1}$ and $\beta^{(2)} = (\beta^{(2)}_k)_{0 \leq k \leq s-1}$, we define their product as the concatenation of both sequences. That is,

$$\beta^{(1)} \cdot \beta^{(2)} = (\beta_k)_{0 \leq k \leq r+s-1} \text{ where } \beta_k = \begin{cases} \beta^{(1)}_k & \text{if } k < r \\ \beta^{(2)}_{k-r} & \text{if } r \leq k < r+s \end{cases}$$

Lemma 4.31. *Let $l, l', l'' \in B^*$ and β, β' be braid move sequences such that*

$$l \xrightarrow{\beta} l' \xrightarrow{\beta'} l''$$

Then, $l \xrightarrow{\beta \cdot \beta'} l''$

```
theorem concatenate_braidMove_sequences (l l' l'' : List B)
(h : ∃ bms : List (cs.BraidMove), cs.apply_braidMoveSequence bms l = l')
(h' : ∃ bms' : List (cs.BraidMove),
    cs.apply_braidMoveSequence bms' l' = l'') :
  ∃ bms'' : List (cs.BraidMove), cs.apply_braidMoveSequence bms'' l = l''
```

Proof. We use an auxiliary lemma to work with Lean's definition in an easier way. Here, we proceed by induction in the first sequence. If it's empty the result is trivial since the concatenation is just β, and otherwise we apply the first move in both sides and use the induction hypothesis.

```
theorem apply_braidMove_sequence_append (bms bms' : List (cs.BraidMove)) (l :
    List B) :
  cs.apply_braidMoveSequence (bms ++ bms') l = cs.apply_braidMoveSequence bms
    (cs.apply_braidMoveSequence bms' l) := by
  induction bms with
  | nil =>
    simp[apply_braidMoveSequence]
  | cons bm bms ih =>
    simp[apply_braidMoveSequence]
    apply congr_arg
    exact ih
```

Then, the proof just constructs the concatenation of both sequences and uses the previous lemma to prove that it acts as expected.

```
rcases h with ⟨bms, hbms⟩
rcases h' with ⟨bms', hbms'⟩
use bms' ++ bms
simp[apply_braidMove_sequence_append]
simp[hbms', hbms]
```

□

Let's check that braid moves are well defined in the Coxeter group.

Theorem 4.32. *Applying a braid move β to a word $l \in B^*$ does not change its product. That is,*

$$l \xrightarrow{\beta} l' \implies \pi(l) = \pi(l')$$

```
theorem braidMove_wordProd (bm : cs.BraidMove) (l : List B) :
    π (cs.apply_braidMove bm l) = π l
```

Proof. Let $i, j \in B$ and $p \in \mathbb{N}$ such that $\beta = \beta[i, j, p]$. We use the `rcases` tactic to extract the data contained in a structure, in this case the index and position of a braid move.

```
rcases bm with ⟨i, j, p⟩
```

Then, we prove the result by induction on p.

```
match p with
| 0 => (...)
| p + 1 => (...)
```

1. $p = 0$: This is the moment where we apply the braid move. Therefore we unfold the definition with the `simp` tactic and divide the proof into two scenarios: one where the braid move can be applied and one where it cannot. This corresponds to the word starting with the braid word $b(i, j)$.

   ```
   simp[apply_braidMove]
   by_cases h : List.take (M.M i j) l = braidWord M i j
   · (...)
   · (...)
   ```

   That is, $w = b(i, j) \cdot w'$ for some $w' \in W$. Then, we want to prove that $w = b(i, j) \cdot w' = b(j, i) \cdot w'$. First we use the lemma Lemma 4.1 to cancel the multiplication on the right by the tail w' and be left with the goal of proving $b(i, j) = b(j, i)$, which is exactly Lemma 4.6.

   ```
   simp[h]
   have h' : l = l.take (M.M i j) ++ l.drop (M.M i j) := by simp
   nth_rewrite 2 [h']
   repeat rw[wordProd_append]
   rw[h]
   simp[wordProd_braidWord_eq]
   ```

   The case where w doesn't start with $b(i, j)$ is trivial, since the braid move doesn't change w. It suffices to use the `simp` tactic with the statement of this case.

```
simp[h]
```

2. $p > 0$: In this case we substitute `p` with `p + 1` and match `l` with either the empty list (the result is trivial in this case) or a list starting with `h`.

```
  match l with
  | [] => simp[apply_braidMove]
  | h::t => (...)
```

Which along with the definition of `apply_braidMove`, leaves us with the goal

```
simp[apply_braidMove, wordProd_cons]

⊢ cs.wordProd (cs.apply_braidMove i j p t) = cs.wordProd t
```

That is exactly the induction hypothesis. Lean is smart enough to figure this when using the `match` keyword, so we can reference the theorem we are proving to finish the proof by induction.

```
exact braidMove_wordProd (BraidMove.mk i j p) t
```

□

We are ready to formalise the statement of the second part of Matsumoto's theorem, relating to reduced words.

```
theorem matsumoto_reduced [MatsumotoCondition cs] (l l' : List B)
(hr : cs.IsReduced l) (hr' : cs.IsReduced l') (h : π l = π l') :
  ∃ bms : List (cs.BraidMove), cs.apply_braidMoveSequence bms l = l'
```

It tells us that two reduced words with the same product can be related by a sequence of braid moves, just like we wanted.

Notice the requirement that our Coxeter system `cs` is an instance of the class `MatsumotoCondition`. This class holds two technical assumptions:

Lemma 4.33. *Let $i, j \in B$ and $p \in \mathbb{N}$ be such that $i \neq j$ and $0 < p < M(i, j)$. Then,*

$$(s_i s_j)^p \neq e$$

This assures us that the product an alternating word of length less than $2 * M(i, j)$ is not equal to one (the neutral element of W). That is, the order of $s_i s_j$ is exactly $M(i, j)$ and not just a divisor of it, as the relations directly imply.

Lemma 4.34. *For every $i, j \in B$, $M(i, j) \geq 1$.*

This assumption says that M has no zero entries, that is, every two generators are part of a braid relation.

```
class MatsumotoCondition where
  one_le_M : ∀ i j : B, 1 ≤ M i j
  alternatingWords_ne_one : ∀ (i j : B) (_ : i ≠ j)
    (p : ℕ) (_ : 0 < p) (_ : p < M i j), (s i * s j) ^ p ≠ 1
```

While Matsumoto's theorem holds for every Coxeter group, these assumptions serve the following purposes:

1. Lemma 4.33 is true of every Coxeter group(Proposition 1.1.1 of [AB05]). However, the proof involves constructing a linear representation of W, that is, a homomorphism $\Phi : W \longrightarrow \mathrm{GL}(U)$, where $\mathrm{GL}(U)$ denotes the group of invertible linear transformations of some vector space U into itself.

   This proof is complex and relies on certain results that are currently absent from mathlib. Consequently, it falls outside the scope of this thesis. Nonetheless, if it were proven in the future, it would show that every Coxeter group adheres to this aspect of `MatsumotoCondition`.

2. Lemma 4.34 assures us that there is a braid relation between every two generators. This braid relation is a key element of the proof of Matsumoto's theorem that we partially follow from [AB05], where this assumption is not stated nor worked around. This highlights the usefulness of formalizing proofs to ensure that no implicit assumptions are taken.

   We are mostly interested in the case where the Coxeter groups are finite (for the symmetric group) so we can easily prove this assumption, but it's worth noting that the proof needs further work to hold in complete generality.

With these assumptions clarified, we can use them to prove intermediate results needed for the proof of Matsumoto's theorem.

Lemma 4.35. *Lists whose length is $p+1$ for some $p \in \mathbb{N}$ have a first element and can be written as the first element appended at the back of a tail list.*

```
lemma cons_of_length_succ {α : Type} (l : List α) {p : ℕ} (h : l.length = p + 1) :
  ∃ (a : α) (t : List α), l = a :: t ∧ t.length = p
```

Proof. The case where `l` is empty leads to a contradiction so it must be of the form `a::t`, in which case a is the element we were looking for.

```
cases l with
  | nil =>
    simp at h
  | cons a t =>
    simp at h
    use a, t
```

□

Definition 4.20. We say that we **shift** a braid move $\beta[i,j,p]$ when we increase the position at which the move is applied by one. That is, we turn it into $s(\beta[i,j,p]) := \beta[i,j,p+1]$

```
def shift_braidMove (bm : cs.BraidMove) : cs.BraidMove :=
  match bm with
  | BraidMove.mk i j p => BraidMove.mk i j (p + 1)
```

Lemma 4.36. *Let $i_0 \in B$, $l, l' \in B^*$ and $\beta[i,j,p]$ be a braid move such that $l \xrightarrow{\beta[i,j,p]} l'$. Then,*

$$i_0 \cdot l \xrightarrow{s(\beta[i,j,p])} i_0 \cdot l'$$

```
lemma braidMove_cons (bm : cs.BraidMove) (l : List B) (a : B) :
  a :: cs.apply_braidMove bm l = cs.apply_braidMove (cs.shift_braidMove bm) (a ::
    l)
```

Proof. Clear by definition of the position of a braid move and shifting.

```
rcases bm with ⟨i, j, p⟩
simp[shift_braidMove, apply_braidMove]
```

□

This can be extended to a braid move sequence with an usual induction proof:

Lemma 4.37. *Let $i_0 \in B$, $l, l' \in B^*$ and β be a sequence of braid moves such that $l \xrightarrow{\beta} l'$. Then,*

$$i_0 \cdot l \xrightarrow{s(\beta)} i_0 \cdot l'$$

Here $s(\beta) = s((\beta_k)_k) := (s(\beta_k))_k$.

```
lemma braidMoveSequence_cons (bms : List (cs.BraidMove)) (l : List B) (a : B) :
  a :: cs.apply_braidMoveSequence bms l =
  cs.apply_braidMoveSequence (List.map cs.shift_braidMove bms) (a :: l)
```

Proof. As we said, we proceed by induction. If the braid move sequence is empty the proof is trivial. Otherwise, we separate the first braid move to be applied, use the previous lemma and then use the induction hypothesis with the tail of the list.

```
induction bms with
| nil =>
   simp[apply_braidMoveSequence]
| cons bm bms ih =>
  rw[apply_braidMoveSequence]
  rw[cs.braidMove_cons bm]
  simp[apply_braidMoveSequence] at ih
  rw[ih]
  simp[apply_braidMoveSequence_cons]
```

□

Lemma 4.38. *Let $i_0 \in B$ and $l \in B^*$ such that $i_0 \cdot l \in B^*$ is a reduced word. Then l is also a reduced word.*

```
theorem isReduced_cons (a : B) (l : List B) : cs.IsReduced (a :: l) →
    cs.IsReduced l
```

Lemma 4.39. *Let $i_0 \in B$ and $l \in B^*$ such that $i_0 \cdot l \in B^*$ is a reduced word. Then, i_0 is a left descent of $i_0 \cdot l \in B^*$.*

```
lemma leftDescent_of_cons (i : B) (l : List B) (hr : cs.IsReduced (i :: l)) :
    cs.IsLeftDescent (π (i :: l)) i
```

Proof. We unfold the definition of a left descent, which means that $\widetilde{\text{len}}(s_{i_0} \cdot s_{i_0} \cdot l) < \widetilde{\text{len}}(s_{i_0} \cdot l)$

```
apply cs.isLeftDescent_iff.mpr
```

But of course, $s_{i_0} \cdot s_{i_0} \cdot l = l$ and both $i_0 \cdot l$ and l are reduced (the second by Lemma 4.38) so we just need to prove that the length of l is less than the length of $i_0 \cdot l$ (as words), which is trivial.

```
rw[hr]
simp[wordProd_cons]
rw[← IsReduced]
apply cs.isReduced_cons i l hr
```

□

As a consequence,

Lemma 4.40. *Let $i_0 \in B$ and $l \in B^*$ such that $i_0 \cdot l \in B^*$ is a reduced word. Then, s_{i_0} is a left inversion of $i_0 \cdot l \in B^*$.*

```
lemma leftInversion_of_cons
(i : B) (l : List B) (hr : cs.IsReduced (i :: l)) :
    cs.IsLeftInversion (π (i :: l)) (s i) :=

   (cs.isLeftInversion_simple_iff_isLeftDescent (π (i :: l)) i).mpr
    (cs.leftDescent_of_cons i l hr)
```

In our reference [AB05], the proof of Matsumoto uses the fact that a Coxeter group has a lattice structure under some order relation. This approach is really general and can be applied to a lot of other results, but we are mainly interested in the proof of Matsumoto for other purposes. Therefore we proved it directly without developing all this theory.

The main intermediate result that we proved by elementary means is the following:

Lemma 4.41. *Under the assumptions Lemma 4.33 and Lemma 4.34, let $l, l' \in B^*$, $i, j \in B$ such that $i \neq j$ but $\pi(i \cdot l) = \pi(j \cdot l')$ with both $i \cdot l$ and $j \cdot l'$ being reduced.*

Then, there's a word $t \in B^*$ such that:

1. $\pi(i \cdot l) = \pi(b(i,j) \cdot t)$
2. $b(i,j) \cdot t$ *is reduced*

```
lemma prefix_braidWord [MatsumotoCondition cs]
(l l' : List B) (i j : B) (i_ne_j : i ≠ j) (pi_eq : π (i :: l) = π (j :: l'))
(hr : cs.IsReduced (i :: l)) (hr' : cs.IsReduced (j :: l')) :
    ∃ t : List B, π (i :: l) = π (braidWord M i j ++ t) ∧
    cs.IsReduced (braidWord M i j ++ t)
```

To prove this, we will need an important results about alternating words.

Lemma 4.42. *Under the assumptions Lemma 4.33 and Lemma 4.34, let $i, j \in B$, $p \in \mathbb{N}$ with $p < M(i,j)$ and $i \neq j$. Then, for every $k = 0, 1, 2, 3$*

$$\pi(a_{p+1}(i,j)) \neq \pi(e_k(a_p(i,j)))$$

```
theorem alternatingWord_succ_ne_alternatingWord_eraseIdx [MatsumotoCondition cs]
(i j : B) (p : ℕ) (hp : p < M i j) (hij : i ≠ j) :
    ∀ (k : ℕ) (hk : k < p),
    π (alternatingWord i j (p + 1)) ≠ π (alternatingWord i j p).eraseIdx k
```

Proof. First of all, let's consider i and j as general members of B since they will switch places in the proof.

```
revert i j
```

We proceed by induction. The case $p = 0$ is trivial, since the condition $k \in \mathbb{N} \ : \ k < 0$ is empty.

Otherwise, we replace p with $p + 1$ and keep track of the induction hypothesis.

```
succ p ih =>
```

Our goal then becomes $\pi(a_{p+1+1}(i, j)) \neq \pi(e_k(a_{p+1}(i, j)))$ We can use the inductive definition of alternating words to extract their last element, turning the goal into $\pi(a_{p+1}(j, i)\cdot j) \neq \pi(e_k(a_p(j, i) \cdot j))$

```
rw[alternatingWord_succ]
nth_rewrite 2 [alternatingWord_succ]
simp
```

Now we can consider the case where we delete the last element or one in the middle.

1. $k < p$: In this case, the del function commmutes with multiplication on the right, so our goal becomes
$$\pi(a_{p+1}(j, i) \cdot j) \neq \pi(e_k(a_p(j, i)) \cdot j)$$
```
rw[List.eraseIdx_append_of_lt_length h_erase [j]]
```
We will prove it by contradiction. Suppose that the two sides are equal. Then, $\pi(a_{p+1}(j, i) \cdot j) = \pi(a_{p+1}(j, i)) \cdot s_j$ and $\pi(e_k(a_p(j, i)) \cdot s_j$, so we can cancel the s_j's out to get the negative of the induction hypothesis. In lean this requires some work to swap i and j before applying `ih`.
```
have hij' : j ≠ i := by
    intro h
    apply hij
    rw[h]
have h_erase' : k < p := by simp at h_erase; exact h_erase

apply ih j i hp'' hij' k h_erase' h_contra
```

2. $k = p$: Actually, in Lean we will work with the condition $k \geq p$ since it's the direct negation of the previous case and it's actually better suited for the next mathlib lemma we will need:

mathlib

```
lemma List.eraseIdx_append_of_length_le {l : List α} {k : ℕ}
(hk : l.length ≤ k) (l' : List α) :
  (l ++ l').eraseIdx k = l ++ l'.eraseIdx (k - l.length)
```

This lemma applied to this specific case explains that since $k \geq p$,

$$e_k(l \cdot j) = l \cdot e_{k-p}(j) = l \quad (k - p = 0)$$

This turns our goal into

$$\pi(a_{p+1}(j,i) \cdot j) \neq \pi(a_p(j,i))$$

```
rw[List.eraseIdx_append_of_length_le h_erase' [j]]
have h_erase_k : [j].eraseIdx (k - (alternatingWord j i p).length) = [] := by
    apply List.eraseIdx_eq_nil.mpr
    right
    simp
    apply Nat.sub_eq_zero_iff_le.mpr
    linarith

rw[h_erase_k]
simp

⊢ ¬cs.wordProd (alternatingWord j i (p + 1) ++ [j]) =
    cs.wordProd (alternatingWord j i p)
```

We proceed by contradiction, that is, we assume that

$$\pi(a_{p+1}(j,i) \cdot j) = \pi(a_p(j,i)) \tag{8}$$

```
intro h_contra
```

Let's start by simplifying the left hand side:

$$\pi(a_{p+1}(j,i) \cdot j) = \pi(a_{p+2}(i,j))$$

```
have : cs.wordProd (alternatingWord j i (p + 1) ++ [j]) =
cs.wordProd (alternatingWord i j (p + 2)) := by
        simp[alternatingWord_succ]

rw[this] at h_contra
```

Now we apply Theorem 4.4 to get the value of both sides of the equation in terms of s_i and s_j. The result depends on the parity of p, so we consider both cases. If p is even, then $p+2$ is even as well, and

$$\pi(a_p(j,i)) = (s_j s_i)^{\frac{p}{2}} \quad \text{and} \quad \pi(a_{p+2}(i,j)) = (s_i s_j)^{\frac{p+2}{2}} = (s_i s_j)^{\frac{p}{2}+1}$$

```
simp[prod_alternatingWord_eq_mul_pow] at h_contra
by_cases p_even : Even p
·   have p_even' : Even (p + 2) := by
    apply Nat.even_add.mpr
    simp
    exact p_even
    simp[p_even, p_even'] at h_contra
```

We apply this equalities to Eq. (8) and multiply both sides by $((s_j s_i)^{\frac{p}{2}})^{-1}$ to obtain

$$(s_i s_j)^{\frac{p}{2}+1} \cdot ((s_j s_i)^{\frac{p}{2}})^{-1} = e$$

But a simple computation shows that

$$((s_j s_i)^{\frac{p}{2}})^{-1} = ((s_j s_i)^{-1})^{\frac{p}{2}} = (s_i^{-1} s_j^{-1})^{\frac{p}{2}} = (s_i s_j)^{\frac{p}{2}}$$

So we obtain

$$e = (s_i s_j)^{\frac{p}{2}+1} \cdot ((s_j s_i)^{\frac{p}{2}})^{-1} = (s_i s_j)^{\frac{p}{2}+1} \cdot (s_i s_j)^{\frac{p}{2}} = (s_i s_j)^{p+1}$$

```
apply mul_inv_eq_one.mpr at h_contra
rw[← inv_pow (s j * s i) (p/2)] at h_contra
simp at h_contra
rw[← pow_add] at h_contra
have : p / 2 + 1 + p / 2 = p + 1 := by
  have : 2 * (p / 2) = p := Nat.two_mul_div_two_of_even p_even
  ring
  simp[mul_comm, this]
rw[this] at h_contra
```

This clearly contradicts Lemma 4.33, since $0 < p < M(i,j)$.

```
apply MatsumotoCondition.alternatingWords_ne_one i j hij (p + 1)
    zero_lt_p_succ _ h_contra
linarith
```

The case where p is odd is completely analogous.

□

Now we are ready to prove Lemma 4.41, which we will do by slowly building the braid word. For that we introduce the following auxiliary lemma:

Lemma 4.43. *Under Lemma 4.33, let $w \in W$, $l, l \in B^*$ and $i, j \in B$ such that $i \neq j$, and both $\pi(i \cdot l)$ and $\pi(j \cdot l')$ are reduced.*

Then, for every $p \leq M(i,j)$, there is a word $t \in B^*$ such that

1. $\pi(a_p(i,j) \cdot t) = w$
2. $a_p(i,j) \cdot t$ *is reduced*

```
lemma prefix_braidWord_aux [MatsumotoCondition cs]
(w : W) (l l' : List B) (i j : B) (i_ne_j : i ≠ j)
(hil : π (i :: l) = w) (hjl' : π (j :: l') = w)
(hr : cs.IsReduced (i :: l)) (hr' : cs.IsReduced (j :: l')) :
    ∀ (p : ℕ) (h : p ≤ M i j), ∃ t : List B,
        π (alternatingWord i j p ++ t) = w ∧
        cs.IsReduced (alternatingWord i j p ++ t)
```

Proof. We proceed by induction on p. The base case ($p = 0$) is trivial, since $a_0(i,j) = e$, so we can take $t = i \cdot l$.

```
intro _
simp[alternatingWord]
use i :: l
```

Then, we assume the goal for p and try to prove it for $p+1$, under the constraint $p+1 \leq M(i,j)$. This clearly implies that $p \leq M(i,j)$, so we can apply the induction hypothesis to obtain $t \in B^*$ such that $\pi(a_p(i,j) \cdot t) = w$ and $a_p(i,j) \cdot t$ is reduced.

```
intro hp
have hp' : p ≤ M i j := by linarith
have hp'' : p < M i j := by linarith
rcases ih hp' with ⟨t, ht, htr⟩
```

So now we can substitute w with this expression to get the following goal:

$$\exists t' \in B^* \ : \ \pi(a_{p+1}(i,j) \cdot t') = \pi(a_p(i,j) \cdot t) \wedge \ a_{p+1}(i,j) \cdot t' \text{ is reduced}$$

```
rw[← ht]

⊢ ∃ t_1,
  cs.wordProd (alternatingWord i j (p + 1) ++ t_1) =
    cs.wordProd (alternatingWord i j p ++ t)
  ∧
  cs.IsReduced (alternatingWord i j (p + 1) ++ t_1)
```

Now, we extract the first letter of $a_{p+1}(i,j)$ using Theorem 4.3, which again leads to two cases. Let's assume that p is even, since the case where it is odd is really similar. Then, $\pi(a_{p+1}(i,j) \cdot t') = \pi(j \cdot a_p(i,j) \cdot t') = s_j \cdot \pi(a_p(i,j) \cdot t')$. So our goal becomes

$$\exists t' \in B^* \ : \ s_j \cdot \pi(a_p(i,j) \cdot t') = \pi(a_p(i,j) \cdot t) \wedge \ j \cdot a_p(i,j) \cdot t' \text{ is reduced}$$

The first part looks similar to the statement of the Strong Exchange Theorem. We can modify it until this theorem can be applied. For that, let's prove that it suffices to show

$$\exists k \leq \text{len}(t) \ : \ s_j \cdot \pi(a_p(i,j) \cdot t) = \pi(a_p(i,j) \cdot e_k(t))$$

```
suffices ∃ k : Fin t.length,
    s j * cs.wordProd (alternatingWord i j p ++ t) =
    cs.wordProd (alternatingWord i j p ++ (t.eraseIdx k))
from by
```

If we manage to prove this, then we can pick $t' = e_k(t)$ to quickly show that

$$s_j \cdot \pi(a_p(i,j) \cdot e_k(t)) = \pi(a_p(i,j) \cdot t) \tag{9}$$

Which implies the first part of the original goal ($\pi(a_{p+1}(i,j) \cdot t') = \pi(a_p(i,j) \cdot t)$). For the second part, we will need some extra work.

```
rcases this with ⟨k, hk⟩
use (t.eraseIdx k)

have hw :
  cs.simple j * cs.wordProd (alternatingWord i j p ++ t.eraseIdx k) =
  cs.wordProd (alternatingWord i j p ++ t)
:= by
  rw[← hk]
  simp[cs.wordProd_cons]

constructor
· exact hw
· (...)
```

To prove that $j \cdot a_p(i,j) \cdot e_k(t)$ is reduced, we need to show that

$$\widetilde{\text{len}}(\pi(j \cdot a_p(i,j) \cdot e_k(t))) = p + 1 + \text{len}(e_k(t))$$

While the induction hypothesis tells us that $a_p(i,j) \cdot t$ is reduced, that is,

$$\widetilde{\text{len}}(\pi(a_p(i,j) \cdot t)) = p + \text{len}(t)$$

```
simp[IsReduced]
simp[IsReduced] at htr
```

But using Eq. (9), $\pi(j \cdot a_p(i,j) \cdot e_k(t)) = s_j \cdot \pi(a_p(i,j) \cdot e_k(t)) = \pi(a_p(i,j) \cdot t)$, so the goal becomes

$$p + 1 + \text{len}(e_k(t)) = \widetilde{\text{len}}(\pi(j \cdot a_p(i,j) \cdot e_k(t))) = \widetilde{\text{len}}(\pi(a_p(i,j) \cdot t)) = p + \text{len}(t)$$

```
rw[cs.wordProd_cons]
rw[hw]
rw[htr]

⊢ p + t.length = p + (t.eraseIdx ↑k).length + 1
```

Of course, $\text{len}(e_k(t)) = \text{len}(t) - 1$, completing the result. This part is surprisingly long in Lean, since we need to prove that $\text{len}(t) \geq 1$ to make sure that $\text{len}(t) - 1$ is still a natural number in order to use their properties.

```
rw[List.length_eraseIdx k.2]
simp[add_assoc]

have : 1 ≤ t.length := by
    apply Nat.le_of_not_lt
    intro h'
    rw[Nat.lt_one_iff] at h'
    rw[h'] at k
    have wah := k.2
    linarith

rw[Nat.sub_add_cancel this]
```

So we know it suffices to prove

$$\exists k \leq \text{len}(t) \ : \ s_j \cdot \pi(a_p(i,j) \cdot t) = \pi(a_p(i,j) \cdot e_k(t))$$

We begin by proving that $s_j \in T_L(\pi(a_p(i,j) \cdot t))$. From our hypotheses,

$$\pi(a_p(i,j) \cdot t) = w = \pi(j \cdot l')$$

And $s_j \in T_L(\pi(j \cdot l'))$ by Lemma 4.40.

```
have h_left_inversion_j :
    cs.IsLeftInversion (cs.wordProd (alternatingWord i j p ++ t)) (s j)
:= by
    rw[ht, ← hjl']
    apply cs.leftInversion_of_cons j l' hr'
```

Knowing this, we can apply the Strong Exchange Theorem (Theorem 4.23) to get

$$\exists k \leq \text{len}(a_p(i,j) \cdot t) \ : \ s_j \cdot \pi(a_p(i,j) \cdot t) = \pi(e_k(a_p(i,j) \cdot t))$$

```
rcases cs.strongExchangeProperty
  (alternatingWord i j p ++ t) ⟨s j, cs.isReflection_simple j ⟩ h_left_inversion_j
with ⟨k, hk⟩
```

Let's fix this k and proceed by cases, depending on its value:

1. Case $k < p$: We will prove that this case is impossible by finding a contradiction. This is done in Lean by using the tactic

   ```
   exfalso
   ```

   Which ignores the goal and turns it into proving $\vdash$ `False` (finding a contradiction). Since $k < p = \text{len}(a_p(i,j))$, we can apply e_k on the first part of the product $a_p(i,j) \cdot t$ to get

   $$s_j \cdot \pi(a_p(i,j) \cdot t) = \pi(e_k(a_p(i,j)) \cdot t)$$

   ```
   have k_lt_len' : k < (alternatingWord i j p).length := by simp[k_lt_len]
   rw[List.eraseIdx_append_of_lt_length k_lt_len' t] at hk
   ```

   Using that π is a homomorphism, we can extract the tails,

   $$s_j \cdot \pi(a_p(i,j)) \cdot \pi(t) = \pi(e_k(a_p(i,j))) \cdot \pi(t)$$

   ```
   simp[cs.wordProd_append] at hk
   ```

   And cancel them

   $$s_j \cdot \pi(a_p(i,j)) = \pi(e_k(a_p(i,j)))$$

   ```
   rw[← mul_assoc] at hk
   rw[mul_right_cancel_iff] at hk
   ```

   Finally, we incorporate s_j into the alternating word with Theorem 4.3 to get

   $$\pi(a_{p+1}(i,j)) = \pi(e_k(a_p(i,j)))$$

   ```
   rw[← wordProd_cons] at hk
   have : j :: alternatingWord i j p = alternatingWord i j (p + 1) := by
       simp[alternatingWord_succ', p_even]
   rw[this] at hk


   ---------
   hk : cs.wordProd (alternatingWord i j (p + 1)) =
        cs.wordProd ((alternatingWord i j p).eraseIdx ↑k)
   ```

   This is a contradiction by Lemma 4.42, since $p < M(i,j)$.

   ```
   exact cs.alternatingWord_succ_ne_alternatingWord_eraseIdx
       ha i j p hp'' i_ne_j k k_lt_len hk
   ```

2. Case $k \geq p$: In this case we get

$$s_j \cdot \pi(a_p(i,j) \cdot t) = \pi(e_k(a_p(i,j) \cdot t)) = \pi(a_p(i,j) \cdot e_{k-p}(t))$$

```
rw[List.eraseIdx_append_of_length_le] at hk
```

Which we can plug into our goal, turning it into (we state it in terms of k' to avoid confusion with the k we got from the Strong Exchange Theorem):

$$\exists k' \leq \text{len}(t) \ : \ \pi(a_p(i,j) \cdot e_{k-p}(t)) = \pi(a_p(i,j) \cdot e_{k'}(t))$$

So taking $k' = k - p$ completes the proof. We just have to show that $0 \leq k' < \text{len}(t)$, which comes from the fact that $k \geq p$ and $k < \text{len}(a_p(i,j) \cdot t) = p + \text{len}(t)$.

```
have : k - (alternatingWord i j p).length < t.length := by
  have kle := k.2
  simp at kle
  simp
  apply (Nat.sub_lt_iff_lt_add _).mpr kle
  exact k_lt_len

use ⟨k - (alternatingWord i j p).length, this⟩
```

□

Proof of Lemma 4.41. Use the previous lemma with $w = \pi(i \cdot l)$ and $p = M(j,i)$. Of course $M(j,i) \leq M(i,j)$ so we can use it this way. Then, just unfold the definition of a braid word as an alternating word and apply it at both parts of the theorem.

```
have h : M i j ≤ M i j := by linarith
have h' : π (j :: l') = π (i :: l) := Eq.symm pi_eq

rcases cs.prefix_braidWord_aux
    (π (i :: l)) l l' i j i_ne_j rfl h' hr hr' (M i j) h
with ⟨t, ht, htr⟩
use t
rw[braidWord]
constructor
· simp[ht]
· exact htr
```

□

Proof of Theorem 4.30. We prove Matsumoto's theorem by induction on the length of the words. For that, we prove an auxiliary lemma that exposes the length of l and l' as $p \in \mathbb{N}$.

Lemma 4.44. *Under Lemma 4.33 and Lemma 4.34, let $p \in \mathbb{N}$ and $l, l' \in B^*$ such that $\text{len}(l) = p = \text{len}(l')$, l, l' are reduced, and $\pi(l) = \pi(l')$.*

Then, there exists a braid move sequence β such that $l \xrightarrow{\beta} l'$

```
theorem matsumoto_reduced_aux [MatsumotoCondition cs]
(p : ℕ) (l l' : List B) (len_l_eq_p : l.length = p)
(len_l'_eq_p : l'.length = p) (l_reduced : cs.IsReduced l)
(l'_reduced : cs.IsReduced l') (h_eq : π l = π l') :
  ∃ bms : List (cs.BraidMove), cs.apply_braidMoveSequence bms l = l'
```

As we said, we will proceed by induction on p. But first, we generalise l and l', since we will use a different pair inside the proof as part of the induction argument.

When $p = 0$, we have $\text{len}(l) = \text{len}(l') = 0$, which clearly means that $l = e = l'$. This result is formalised in the following lemma:

mathlib

```
theorem length_eq_zero : length l = 0 ↔ l = []
```

so no braid moves are needed to show that they are equal. Therefore, we use an empty list as the sequence of braid moves (which doesn't modify l) and $l = l'$ yields the result.

```
intro l l' hl hl' _ _ _
have h_len : l.length = l'.length := by rw[hl, hl']
simp at h_len
use []
simp[apply_braidMoveSequence]
apply List.length_eq_zero.mp at hl
apply List.length_eq_zero.mp at hl'
rw[hl, hl']
```

Otherwise, we assume the result is true for p and prove it for $p+1$. So $\text{len}(l) = \text{len}(l') = p + 1$, which means that there exists $i, j \in B$ such that $l = i \cdot l_t$ and $l' = j \cdot l'_t$ for some $l_t, l'_t \in B^*$, by applying Lemma 4.35.

```
intro l l' len_l_eq_p len_l'_eq_p l_reduced l'_reduced h_eq
rcases cons_of_length_succ l len_l_eq_p with ⟨i, l_t, rfl, len_l_t_eq_p⟩
rcases cons_of_length_succ l' len_l'_eq_p with ⟨j, l'_t, rfl, len_l'_t_eq_p⟩
```

Now, there are two cases, depending of whether $i = j$ or not.

If the equality holds, we will use the induction hypothesis to find a sequence of braid moves from l_t to l'_t. Then, since the first letter is equal, the shifted sequence will transform l into l'.

But to apply the induction hypothesis, we need to satisfy all its requirements:

- **l_t and l'_t are reduced** They are the result of removing the first letter from l and l' respectively. Hence, Lemma 4.38 yields the result.

  ```
  have htr : cs.IsReduced l_t := cs.isReduced_cons i l_t l_reduced
  have htr' : cs.IsReduced l'_t := cs.isReduced_cons j l'_t l'_reduced
  ```

- $\pi(l_t) = \pi(l'_t)$ This is proven by a simple chain of equivalences;

$$\pi(l) = \pi(l') \implies \pi(i \cdot l_t) = \pi(i \cdot l'_t) \implies s_i\pi(l_t) = s_i\pi(l') \implies \pi(l_t) = \pi(l'_t)$$

```
have h_prod : π l_t = π l'_t := by
    apply @mul_left_cancel _ _ _ (cs.simple i) _ _
    rw[← cs.wordProd_cons i l_t, ← cs.wordProd_cons i l'_t, h_eq]
    rw[← first_letter_eq]
```

- $\text{len}(l_t) = \text{len}(l'_t) = p$ By definition.

So using the previous properties we can apply the induction hypothesis at them to get that there exists a sequence of braid moves $\beta = (\beta_k)_k$ such that $l_t \xrightarrow{\beta} l'_t$

```
have ⟨bms, ih'⟩ := ih l_t l'_t len_l_t_eq_p len_l'_t_eq_p htr htr' h_prod
```

Then, Lemma 4.37 implies that

$$l = i \cdot l_t \xrightarrow{s(\beta)} i \cdot l'_t = l'$$

Making $s(\beta)$ the obvious choice for the braid move sequence.

```
apply (List.cons_inj_right j).mpr at ih'
rw[← ih']
rw[braidMoveSequence_cons]
use (List.map cs.shift_braidMove bms)
```

This case will appear later, so we abstract it as a separate lemma in lean:

```
lemma matsumoto_reduced_inductionStep_of_firstLetterEq
(p : ℕ) (l_t l'_t : List B) (i : B)
(len_l_t_eq_p : l_t.length = p) (len_l'_t_eq_p : l'_t.length = p)
(h_eq : π (i :: l_t) = π (i :: l'_t))
(l_reduced : cs.IsReduced (i :: l_t)) (l'_reduced : cs.IsReduced (i :: l'_t))
(ih: ∀ (l l' : List B),
  l.length = p →
    l'.length = p →
      cs.IsReduced l → cs.IsReduced l' → cs.wordProd l = cs.wordProd l' → ∃
    bms, cs.apply_braidMoveSequence bms l = l'):
∃ bms, cs.apply_braidMoveSequence bms (i :: l_t) = i :: l'_t
```

Now assume that $i \neq j$. From Lemma 4.34 we know that $M(i,j) \geq 1$, so we can pick $m := M(i,j) - 1$ and rewrite it as

$$M(j,i) = M(i,j) = m + 1$$

```
obtain ⟨m, hm⟩ : ∃ m : ℕ ,M i j = m + 1 := by
    use M i j - 1
    simp[MatsumotoCondition.one_le_M cs i j]

have hm' : M j i = m + 1 := by
    simp[M.symmetric]
    exact hm
```

Then we have to consider the case where m is even or odd. As usual, we only write the case where m is even, because the odd case is really similar.

Then, we have l'_t, l_t reduced words and $i, j \in B$ with $i \neq j$ such that $\pi(i \cdot l_t) = \pi(j \cdot l'_t)$, so we can apply Lemma 4.41 to get $b_t \in B^*$ such that $b(j,i) \cdot b_t$ is reduced and

$$\pi(j \cdot l'_t) = \pi(b(j,i) \cdot b_t)$$

```
obtain ⟨b_tail, hb, b_reduced⟩ :=
          cs.prefix_braidWord ha l'_t l_t j i j_ne_i (Eq.symm h_eq) l'_reduced
    l_reduced
```

Of course, that immediately implies that $\pi(i \cdot l) = \pi(b(j,i) \cdot b_t)$ as well.

```
have hb' : cs.wordProd (i :: l_t) = cs.wordProd (braidWord M j i ++ b_tail) := by
  rw[← hb]
  exact h_eq
```

Now, our goal is to find a braid move sequence β such that $l \xrightarrow{\beta} l'$. We will do this in steps, by finding three braid move sequences $\beta^{(1)}, \beta^{(2)}$ and $\beta^{(3)}$ such that

$$l = i \cdot l_t \xrightarrow{\beta^{(1)}} b(j,i) \cdot b_t \xrightarrow{\beta^{(2)}} b(i,j) \cdot b_t \xrightarrow{\beta^{(3)}} j \cdot l'_t$$

We use Lemma 4.31 to split the proof like this. First, we apply it with $i \cdot l_t$ and $b(j,i) \cdot b_t$ to divide the proof in two goals, showing $\exists \beta^{(1)}$ with $i \cdot l_t \xrightarrow{\beta^{(1)}} b(j,i) \cdot b_t$ and $\exists \beta^{(*)}$ with $b(j,i) \cdot b_t \xrightarrow{\beta^{(*)}} j \cdot l'_t$. Afterwards, we will split the second goal further.

```
apply cs.concatenate_braidMove_sequences
    (i :: l_t) (braidWord M j i ++ b_tail) (j :: l'_t)
```

We will solve the first goal by first observing that the words at both sides start with i, since using Theorem 4.3,

$$b(j,i) \cdot b_t = a_{m+1}(j,i) \cdot b_t = i \cdot a_m(j,i) \cdot b_t \tag{10}$$

```
have b_word_cons : (braidWord M j i ++ b_tail) = i :: (alternatingWord j i m ++
    b_tail) := by
  simp[braidWord]
  rw[hm']
  simp[alternatingWord_succ']
  simp[m_even]

rw[b_word_cons]
```

Furthermore, the length of $a_m(j,i) \cdot b_t$ is p. To prove this, it suffices to show that $\text{len}(b(j,i) \cdot b_t) = p + 1$

```
suffices (braidWord M j i ++ b_tail).length = p + 1 from by
...
```

This is because these conditions are clearly equivalent, given that

$$\text{len}(b(j,i) \cdot b_t) = \text{len}(b(j,i)) + \text{len}(b_t) = m + 1 + \text{len}(b_t) = \text{len}(a_m(j,i) \cdot b_t) + 1$$

And the previous fact is easy to see with Lemma 4.10, which knowing that $\pi(b(j,i) \cdot b_t) = \pi(i \cdot l_t)$ and they are both reduced, tells us that their lengths are equal, and $\text{len}(i \cdot l_t) = p+1$ by induction hypothesis.

```
rw[← cs.eq_length_of_isReduced (i :: l_t)
    (braidWord M j i ++ b_tail) hb' l_reduced b_reduced]
exact len_l_eq_p
```

Then, we can re-use the argument we used before when the first letters of both reduced equivalent words were equal to get the result. We just have to rewrite the previous transformation in some hypotheses.

```
rw[b_word_cons] at hb'
rw[b_word_cons] at b_reduced
apply cs.matsumoto_reduced_inductionStep_of_firstLetterEq p l_t (alternatingWord
    j i m ++ b_tail) i len_l_t_eq_p b_len_p hb' l_reduced b_reduced ih
```

Now that the first step is done, we break down further the second goal

$$\exists \beta^{(*)} \text{ with } b(j,i) \cdot b_t \xrightarrow{\beta^{(*)}} j \cdot l'_t$$

Into the following two sub-goals, using Lemma 4.31 as before

$$\exists \beta^{(2)} \text{ with } b(j,i) \cdot b_t \xrightarrow{\beta^{(2)}} b(i,j) \cdot b_t$$

$$\exists \beta^{(3)} \text{ with } b(i,j) \cdot b_t \xrightarrow{\beta^{(3)}} j \cdot l'_t$$

```
apply cs.concatenate_braidMove_sequences
    (braidWord M j i ++ b_tail) (braidWord M i j ++ b_tail) (j :: l'_t)
```

The first goal clearly consists in a single braid move $\beta[j,i,0]$, which turns

$$b(j,i) \cdot b_t \xrightarrow{\beta[j,i,0]} b(i,j) \cdot b_t$$

```
use [BraidMove.mk j i 0]
simp[apply_braidMoveSequence]
simp[apply_braidMove]
```

For the second case, we observe again that the first letter is j in both sides, since $b(i,j) \cdot b_t = j \cdot a_m(i,j) \cdot b_t$:

```
have b_word_cons : (braidWord M i j ++ b_tail) =
    j :: (alternatingWord i j m ++ b_tail) := by
  simp[braidWord]
  rw[hm]
  simp[alternatingWord_succ']
  simp[m_even]
```

And the rest of the prerequisites to apply our previous argument are present (mainly that the length of both sides is p)

```
have b_len_p : (alternatingWord i j m ++ b_tail).length = p
```

So we can finish the proof by using it again:

```
rw[b_word_cons] at hb'
rw[b_word_cons] at b_reduced'

apply cs.matsumoto_reduced_inductionStep_of_firstLetterEq
    p (alternatingWord i j m ++ b_tail) l'_t j b_len_p len_l'_t_eq_p
    (Eq.symm hb') b_reduced' l'_reduced ih
```

Now that we have the proof of this auxiliary lemma, the proof of Theorem 4.30 is just a special case:

```
theorem matsumoto_reduced [...] := by
  apply cs.matsumoto_reduced_aux (l.length) l l' rfl _ hr hr' h
```

Notice that all hypotheses for the auxiliary theorem are directly given from those of the final theorem except for `len_l_eq_p` and `len_l'_eq_p`. For the first, since we supplied `l.length` as `p`, the goal becomes

```
⊢ l.length = l.length
```

Goals like this, where we have to prove `⊢ a = a` are resolved with the reflexivity tactic `rfl`.

And for `len_l'_eq_p`, the goal becomes

```
⊢ l.length = l'.length
```

This proof is a bit more complicated so we leave it for later with the placeholder keyword (`_`). Then, the `calc` tactic helps us prove this through a chain of equalities.

`l'.length` is equal to `len (π l')` (the length of its product) because `l'` it's a reduced word (`hr'`). Then, `len (π l')` is equal to `len (π l)` from the hypothesis `h`, which is in turn equal to `l.length` because `l` is also reduced (`hr`).

```
calc
  l'.length = len (π l') := by
    rw[IsReduced] at hr'
    rw[← hr']
  _ = len (π l) := by rw[h]
  _ = l.length := by
    rw[IsReduced] at hr
    rw[← hr]
```

□

5 Demazure operators over S_{n+1}

5.1 S_{n+1} as a Coxeter group

With Matsumoto's theorem in hand, we can broaden the definition of Demazure operators. Think back to the difficulties encountered during our initial expansion attempt and consider how we can now overcome those issues.

To apply Matsumoto's theorem (and the rest of results of the previous section), we need to show that S_{n+1} is in fact a Coxeter group. Surprisingly, this is not trivial and, in fact, it has not been formalised yet.

We didn't find a simple way to do it, so we didn't think it was worth spending a lot of time into, specially knowing that the mathlib community is working on a generalised way to identify Coxeter groups.

We leave a blueprint to prove this fact from Exercise 1.5 of [AB05], which requires a highly technical but direct induction proof, and it's probably the shortest way to prove this fact with the current tools. Another way is by using the Exchange property characterization of Coxeter groups (Proposition 1.5.4 of [AB05]), which is also not too far off now that the Strong Exchange Theorem is formalised.

Sadly, this means that this becomes the only part of this work that is not formalised and instead taken for granted. However, this is a basic fact in almost every reference about Coxeter groups[Bou08; AB05; Hum90; Dav08] so its veracity is hardly disputed.

As we explained in Example 4.1, let Cox_n be the Coxeter group associated to the matrix A_n, with generators $\{\overline{s}_i\}_{i\in[n]}$

```
variable (n : ℕ)
def M := CoxeterMatrix.Aₙ n
def cs := (M n).toCoxeterSystem
abbrev Cox (n : ℕ) := (M n).Group
```

Then let's consider the symmetric group S_{n+1}, with generators $\{s_i\}_{i\in[n]}$

```
abbrev S (n : ℕ) := Equiv.Perm (Fin n)
instance : Group (S (n + 1)) := Equiv.Perm.permGroup
```

Theorem 5.1. *There is a natural homomorphism of groups $f : \mathrm{Cox}_n \longrightarrow S_{n+1}$ sending*

$$\overline{s}_i \mapsto s_i$$

Proof. As described in Section 4.6, we construct this map by lifting the map $\tilde{f} : [n] \longrightarrow S_{n+1}$ that sends $i \mapsto s_i$.

```
def f_simple : Fin n → S (n + 1) :=
  fun i => Equiv.swap i.castSucc i.succ
```

We show that f is liftable with a similar argument to those used in Section 4.6.

```
theorem f_liftable : (M n).IsLiftable (f_simple n) := by
    ...
```

Then, the function we are looking for is

```
def f := lift (cs n) ⟨ f_simple n, f_liftable n ⟩
```

□

Lemma 5.2. *f is surjective.*

Proof. Being surjective is equivalent to having the range of f be the whole monoid S_{n+1} (the top element of the subgroup order, denoted by `T` in lean)

```
apply MonoidHom.mrange_top_iff_surjective.mp
⊢ MonoidHom.mrange (f n) = T
```

But we also have that the closure of $\{s_0, \ldots, s_{n-1}\}$ is S_{n+1}, since these transpositions generate the symmetric group.

```
have : Submonoid.closure (Set.range fun (i : Fin n) ↦ Equiv.swap i.castSucc
    i.succ) =  := by
    exact Equiv.Perm.mclosure_swap_castSucc_succ n
```

Therefore, it suffices to prove that these generators are in the image of f, which is trivial by its definition.

```
rw[← this]
simp

intro p hp
simp at hp
rcases hp with ⟨ i, rfl ⟩
simp
use (cs n).simple i
simp[f_apply_simple]
```

□

Theorem 5.3 (Proposition 1.5.4 of [AB05])**.** *f is an isomorphism*

Sketch of proof. We construct the isomorphism as that with underlying function f and using the previous lemma to show it's surjective. The proof of injectivity is missing.

```
def f_equiv : (S' n) ≃* S (n + 1) := by
  apply MulEquiv.ofBijective (f n)
  constructor
  · sorry
  · exact f_surjective n
```

□

Now, we can consider S_{n+1} as a Coxeter group with the previous isomorphism.

```
def S_cox : CoxeterSystem (M n) (S (n + 1)) := ⟨ (f_equiv n).symm ⟩
```

Remark 5.4. *As expected, the generator set S is the set of transpositions s_i*

```
theorem S_cox_simple (i : Fin n) :
    (S_cox n).simple i = (Equiv.swap i.castSucc (i.succ)) := by
  rw[← f_equiv_apply_simple]
  rfl
```

Theorem 5.5. *The symmetric group S_{n+1} satisfies the conditions needed to apply Matsumoto's Theorem.*

Proof. 1. Lemma 4.34: $\forall i, j \in [n],\ 1 \leq M(i,j)$

We simply show that the inequality holds for every i and j, regardless of whether they are equal, adjacent or none of them.

```
intro i j
simp[M, CoxeterMatrix.A_n]
by_cases h1 : i = j
repeat
  by_cases h2 : j.val + 1 = i.val ∨ i.val + 1 = j
  repeat simp [h1, h2]
```

2. Lemma 4.33: $\forall i, j \in B$ with $i \neq j$ and $p \in \mathbb{N}$ such that $0 < p < M(i,j)$, $(s_i s_j)^p \neq e$

We proceed by cases. For the case where i and j are adjacent we proved the following technical lemma:

Lemma 5.6. *Let $i, j \in [n]$ such that $i = j + 1$ or $j = i + 1$. Then, $s_i s_j$ is a cycle of length three*

```
lemma cycle_of_adjacent_swap (i j : Fin n) (hij : i ≠ j)
    (h1 : j.succ = i.castSucc ∨ i.succ = j.castSucc) :
  Equiv.Perm.IsThreeCycle
    (Equiv.swap i.castSucc i.succ * Equiv.swap j.castSucc j.succ)
```

As expected, this means that $s_i s_j$ has order three. This is formalised in

mathlib

```
theorem Equiv.Perm.IsThreeCycle.orderOf {g : Equiv.Perm α} (ht :
    g.IsThreeCycle) :
    orderOf g = 3
```

Then, we unfold the definition of order, which tells us two things; That $(s_i s_j)^3 = e$, which we ignore by using the _ placeholder and that $(s_i s_j)^p \neq e$ when $0 < p < 3$, exactly what we are looking for.

```
obtain ⟨ _, ho ⟩ := (orderOf_eq_iff zero_lt_three).mp
    (Equiv.Perm.IsThreeCycle.orderOf this)
simp at ho

apply ho p hp' hp
```

When i and j aren't adjacent, $M(i,j) = 2$ so the only value within bounds is $p = 1$. Therefore, we need to show that $s_i s_j \neq e$ This is achieved by showing that $s_i s_j(j) = s_i(j+1) = j + 1$

```
calc
    (Equiv.swap i.castSucc i.succ * Equiv.swap j.castSucc j.succ) j =
        Equiv.swap i.castSucc i.succ (Equiv.swap j.castSucc j.succ j)
      := by rfl
    _ = Equiv.swap i.castSucc i.succ j.succ
      := by simp[h1]
    _ = j.succ := by
        apply Equiv.swap_apply_of_ne_of_ne
        intro h
        simp at h1
        apply Fin.ext_iff.mp at h
```

```
        simp at h
        simp[h] at h1
        intro h
        apply Fin.succ_inj.mp at h
        apply hij h.symm
```

The only non-trivial part is showing that $s_i(j+1) = j+1$ by proving $j+1 \neq i$ and $j+1 \neq i+1$ in $[n]$, which requires some type conversions.

□

5.2 Extending the definition

Definition 5.1. Let $l = i_0 i_1 \cdots i_{p-1} \in B^*$ be a word ($p \in \mathbb{N}$, $i_k \in B$ $\forall 0 \leq k < p$). The Demazure operator at l is defined as:

$$\partial_l := \partial_{i_0} \circ \partial_{i_1} \circ \cdots \circ \partial_{i_{p-1}}$$

```
def DemazureOfWord (l : List (Fin n)) : LinearMap (RingHom.id ℂ) (MvPolynomial
    (Fin (n + 1)) ℂ) (MvPolynomial (Fin (n + 1)) ℂ) :=
  match l with
  | [] => LinearMap.id
  | i :: l => LinearMap.comp (Demazure i) (DemazureOfWord l)
```

Some properties are immediate from the definition:

Lemma 5.7. *Let $l, l' \in B^*$ be two words. Then,*

$$\partial_{l.l'} = \partial_l \circ \partial_{l'}$$

```
lemma demazureOfWord_append (l l' : List (Fin n)) : DemazureOfWord (l ++ l') =
    LinearMap.comp (DemazureOfWord l) (DemazureOfWord l') := by
  induction l with
  | nil => simp[DemazureOfWord]
  | cons i l ih => simp[DemazureOfWord, ih, LinearMap.comp_assoc]
```

Now we want to lift this definition to S_{n+1}, while avoiding the pitfalls we identified before the Coxeter section. We can't have two equal transpositions next to each other or the resulting operator will be zero. To avoid this, we only consider reduced words for the definition.

Definition 5.2. Let $w \in W$. Then, let $l \in B^*$ be a reduced word representing w, that is, such that $\pi(l) = w$. This word is guaranteed to exist by Lemma 4.8. Then, the Demazure operator at w is defined as:

$$\partial_w := \partial_l$$

To formalise this definition we will use `Classical.choose` to get a specific reduced word from the existence statement in Lemma 4.8. This means that this definition is non-computable.

```
noncomputable def DemazureOfProd (w : S (n + 1)) :
LinearMap (RingHom.id ℂ) (MvPolynomial (Fin (n + 1)) ℂ) (MvPolynomial (Fin (n +
    1)) ℂ) :=
  DemazureOfWord (Classical.choose ((Symm n).exists_reduced_word' w))
```

To establish that this is well defined, we have to prove that the choice of reduced word l is irrelevant. We begin by showing that executing a braid move does not change the Demazure operator for a given word.

Lemma 5.8. *Let $l, l' \in B^*$ related by a braid move $\beta[i,j,p]$, that is,*

$$l \xrightarrow{\beta[i,j,p]} l'$$

Then, $\partial_l = \partial_{l'}$

```
theorem demazure_of_braidMove (l : List (Fin n)) (bm : cs.BraidMove) :
    DemazureOfWord l = DemazureOfWord (cs.apply_braidMove bm l)
```

Proof. We proceed by induction on l, generalizing the braid word. The initial case is trivial, since if $l = e$, there's no word to apply and both Demazure operators are equal to the identity.

```
evert bm
  induction l with
  | nil =>
    rintro ⟨i, j, p⟩
    simp[DemazureOfWord, apply_braidMove]
    rw[apply_braidMove.eq_def]
    simp[braidWord_ne_nil]

    match p with
    | 0 => simp[DemazureOfWord]
    | _ + 1 => simp[DemazureOfWord]

  | cons i' l ih => ...
```

Now, let's assume that the result holds for l and let's prove it for $i' \cdot l$. That is, for any braid move β and $l' \in B^*$ with $l \xrightarrow{\beta} l'$ we have

$$\partial_l = \partial_{l'}$$

Now let $\beta[i,j,p]$ be a braid move with $i, j \in B$ and $p \in \mathbb{N}$. We proceed by cases on p. If it's greater than zero, we write it as $p+1$. Then, applying $\beta[i,j,p+1]$ to $i' \cdot l$ results in $i' \cdot l'$, where $l \xrightarrow{\beta[i,j,p]} l'$ by the definition of braid words. But then,

$$\partial_{i' \cdot l} = \partial_{i'} \circ \partial_l = \partial_{i'} \circ \partial_{l'} = \partial_{i' \cdot l'}$$

We have applied the induction hypothesis in the middle equality.

```
match p with
| p + 1 =>
  simp[DemazureOfWord, apply_braidMove]
  simp[LinearMap.comp, Function.comp]
  apply congr_arg
  rw[ih ⟨i, j, p⟩]
```

If $p = 0$, we apply the braid move right away, so we split by cases depending on whether $i' \cdot l$ starts with $b(i,j)$ or not.

```
simp[apply_braidMove]
by_cases h : List.take (M n i j) (i' :: l) = braidWord (M n) i j
```

Of course if it doesn't the proof is trivial since the braid move doesn't modify l. So we assume that $i' \cdot l = b(i,j) \cdot l_t$. Therefore, we need to show that

$$\partial_{b(i,j)\cdot l_t} = \partial_{b(j,i)\cdot l_t}$$

```
nth_rewrite 1 [← List.take_append_drop (M n i j) (i' :: l)]
simp[h]
rw[demazureOfWord_append]
rw[demazureOfWord_append]
```

So, since $\partial_{b(i,j)\cdot l_t} = \partial_{b(i,j)} \circ \partial_{l_t}$ (and similarly for the right side), it suffices to show

$$\partial_{b(i,j)} = \partial_{b(j,i)}$$

```
suffices DemazureOfWord (braidWord (M n) i j) =
    DemazureOfWord (braidWord (M n) j i) from by
  rw[this]
```

First, we discard the case where $i = j$ since it's trivial, and simplify braid words to alternating words of length $M(i,j)$ (resp. $M(j,i)$)

```
simp[braidWord]
by_cases h_eq : i = j
· simp[h_eq]
```

Now we consider the case where i and j are non adjacent. Then, after unfolding the definition of `NonAdjacent` we apply the inequalities at the Coxeter matrix A_n to get that $M(i,j) = 2$. Therefore, the goal turns into

$$\partial_{a_2(i,j)} = \partial_{a_2(j,i)}$$

```
obtain ⟨_, h2, h3, _⟩ := by exact h_adjacent
simp at h2 h3

have h2' := not_imp_not.mpr Fin.eq_of_val_eq h2
have h3' := not_imp_not.mpr Fin.eq_of_val_eq h3

simp at h2' h3'

simp[M, CoxeterMatrix.A_n, h_eq, j_ne_i, h2', h3', Ne.symm h2', Ne.symm h3']
```

By the definition of alternating words, the previous equation turns into

$$\partial_i \circ \partial_j = \partial_j \circ \partial_i$$

Which is one of the equations we proved in Proposition 3.9.

```
simp[alternatingWord, DemazureOfWord, Demazure, LinearMap.comp, Function.comp]
funext p
apply demazure_commutes_non_adjacent i j h_adjacent p
```

If i and j are adjacent, this means that either $i = j + 1$ or $j = i + 1$ as natural numbers by simply inverting the inequalities of `NonAdjacent`

```
have h_adjacent' : j.val + 1 = i.val ∨ i.val + 1 = j.val := by
    rw[NonAdjacent] at h_adjacent
    simp at h_adjacent
    by_contra h_contra
    simp at h_contra
    rcases h_contra with ⟨h1, h2⟩
    apply h_eq
    apply h_adjacent h_eq
    intro h_fin
    apply h1
    apply Eq.symm
    apply Fin.val_eq_of_eq h_fin

    intro h_fin
    apply h2
    apply Fin.val_eq_of_eq h_fin
```

Then, we use these two cases to get that $M(i, j) = 3$ and therefore turn the goal into

$$\partial_j \circ \partial_i \circ \partial_j = \partial_i \circ \partial_j \circ \partial_i$$

```
simp[M, CoxeterMatrix.Aₙ, j_ne_i, h_eq, h_adjacent', Or.comm.mp h_adjacent']
simp[alternatingWord, DemazureOfWord, Demazure, LinearMap.comp, Function.comp]
```

Let's assume that $i = j + 1$ (the other case is analogous). Observe that $j + 1 = i < 0$, so we can see it as an element of $[n]$. This part is mostly a formality to get the equality $i = j$ in $[n]$ and not only in $\mathbb{N}$.

```
have hj : j.val + 1 < n := by
  rw[h1]
  simp

have h1' : ⟨j.val + 1, hj⟩  = i := by
  apply Fin.ext
  simp[h1]
```

Then we substitute i by $j + 1$ and apply the second part of Proposition 3.9 to finish the proof.

```
rw[← h1']
funext p
exact demazure_commutes_adjacent j hj p
```

□

We can easily extend this result to a sequence of braid moves with an induction argument.

Lemma 5.9. *Let $l, l' \in B^*$ related by a sequence of braid moves β, that is,*

$$l \xrightarrow{\beta} l'$$

Then, $\partial_l = \partial_{l'}$

```
lemma demazure_of_braidMoveSequence
(l : List (Fin n)) (bms : List (Symm n).BraidMove) :
    DemazureOfWord l = DemazureOfWord ((Symm n).apply_braidMoveSequence bms l) :=
    by
  induction bms with
  | nil =>
    simp[apply_braidMoveSequence]
  | cons bm bms ih =>
    rw[apply_braidMoveSequence]
    rw[← demazure_of_braidMove ((Symm n).apply_braidMoveSequence bms l) bm]
    exact ih
```

Finally, we can prove that this definition stays the same when taking an equivalent word, thanks to Matsumoto's Theorem.

Theorem 5.10. *Let $l, l' \in B^*$ such that they are both reduced words and $\pi(l) = \pi(l')$. Then,*

$$\partial_l = \partial_{l'}$$

```
theorem DemazureOfWord_eq_equivalentWord (l l' : List (Fin n))
(h_eq : π l = π l') (hr : cs.IsReduced l) (hr' : cs.IsReduced l') :
  DemazureOfWord l = DemazureOfWord l'
```

Proof. We know that it suffices to prove that there is a sequence of braid moves β such that $l \xrightarrow{\beta} l'$ with Lemma 5.9 implies the result.

```
suffices ∃ (bms : List (Symm n).BraidMove),
  (Symm n).apply_braidMoveSequence bms l = l' from by
    rcases this with ⟨bms, h⟩
    rw[← h]
    exact demazure_of_braidMoveSequence l bms
```

This is precisely what Matsumoto's theorem states, given that l and l' are reduced words with $\pi(l) = \pi(l')$.

```
exact (Symm n).matsumoto_reduced l l' hr hr' h_eq
```

□

References

[AB05] Francesco Brenti (auth.) Anders Bjorner. *Combinatorics of Coxeter Groups*. 1st ed. Graduate Texts in Mathematics №231. Springer, 2005. ISBN: 3540442383; 9783540442387; 9783540275961; 3540275967.

[Bou08] N. Bourbaki. *Lie Groups and Lie Algebras: Chapters 4-6*. Elements de mathematique [series] partes 4-6. Springer Berlin Heidelberg, 2008. ISBN: 9783540691716.

[Cox34] H. S. M. Coxeter. "Discrete Groups Generated by Reflections". *Annals of Mathematics* 35.3 (1934), pp. 588–621. ISSN: 0003486X, 19398980.

[Dav08] Michael W. Davis. *The geometry and topology of Coxeter groups*. English. Vol. 32. Lond. Math. Soc. Monogr. Ser. Princeton, NJ: Princeton University Press, 2008. ISBN: 978-0-691-13138-2.

[Dem74] Michel Demazure. "Désingularisation des variétés de Schubert généralisées". fr. *Annales scientifiques de l'École Normale Supérieure* 4e série, 7.1 (1974), pp. 53–88. DOI: 10.24033/asens.1261.

[Hum90] James E. Humphreys. *Reflection Groups and Coxeter Groups*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1990.

[Lea] *Lean webpage*. URL: https://lean-lang.org/about/.

[Mil] *Mathematics in Lean*. URL: https://leanprover-community.github.io/mathematics_in_lean/.

[Mata] *mathlib*. URL: https://github.com/leanprover-community/mathlib4.

[Matb] *mathlib documentation*. URL: https://leanprover-community.github.io/mathlib4_docs/.

[Ála] Óscar Álvarez. *Formalization of the Demazure operators in lean (code)*. URL: https://github.com/bolito2/DemazureOperatorsLean.

[Álb] Óscar Álvarez. *Formalization of the Demazure operators in lean (documentation)*. URL: https://bolito2.github.io/DemazureOperatorsLean/.