Braids in Lean

Hannah Fechtner

December 2024

project repository : github.com/hannahfechtner/braids_better

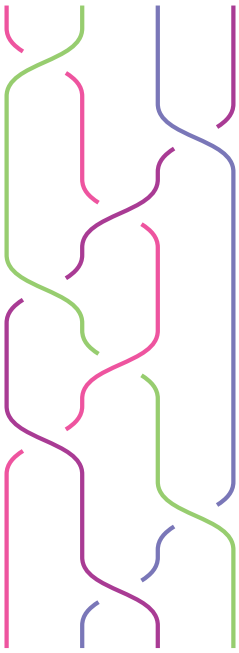
Contents

1 Introduction

1.1 Mathematical Braids

1.1.1 Three Representations

Let's begin with an intuitive overview of the idea of a mathematical braid. We consider strings in three-dimensional space, affixed at top and bottom to the "floor" and "ceiling." These strings can be twisted together or crossed, but must always flow downwards. No doubling-back is allowed. We imagine the strings to be made of infinitely thin rubber — stretchable and shrinkable. They can bend and move, so long as they never double back or intersect one another. This is the sort of image to keep in mind:



Mathematically, there are two ways to define a braid : topologically or algebraically. My Lean formalization uses the algebraic definition. I will present a sketch of the topological definition below, which is useful to get a sense of the mathematical object described, and to follow the historical discussion in Chapter 4. This project follows the definitions and results from Dehornoy in his textbook "The Calculus of Braids" [13].

Topologically, we first define a "static" geometric braid on n strings. A geometric braid is a set of n continuous curves Z_k, $1 \leq k \leq n$, in $\mathbb{R}^3$ (called "braid strands") where all Z_k start along one line and end along another — without loss of generality, let us say Z_k starts at $(k, 1, 0)$ and ends at $(j, 0, 0)$ for some $1 \leq j \leq n$. Braid strands may not intersect each other; they also may not "double back" : for any $y^* \in [0, 1]$, Z_k intersects the plane $y = y^*$ exactly once (and thus at least once, so braid strands have no splits or gaps). The set of geometric braids on n strings is called GB_n.

Now, two geometric braids are considered equivalent if one may be continuously deformed into the other, where the object remains a geometric braid at all points in the deformation. This deformation may be applied either directly to the braid strands themselves (in which case it is a homotopy), or to $\mathbb{R}^3$, the space in which the geometric braid lives[1] (an ambient isotopy). Since the geometric braid

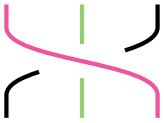
[1] Often, one bounds the braid within a cube in $\mathbb{R}^3$, and needs only to apply the ambient isotopy to said cube.

lives inside $\mathbb{R}^3$, it will be deformed along with the entire space. Dehornoy proves these two notions are equivalent, and so we may simply use $\approx_n$ to denote the deformation relation on braids with n strands. In practice, one defines a homotopy from one braid to another, and thus concludes the two braids isotopic.

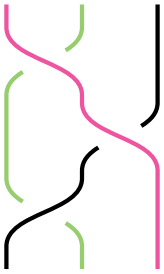
Thus, we define the set of braids on n strands, B_n, to be $GB_n/\approx_n$, the set of equivalence classes of GB_n under $\approx_n$. This is the first representation of a braid.

For any braid, we may consider its projection into 2-d space, with over- and under-crossings labelled in some manner. Two projections of braids are equivalent if and only if the three-dimensional geometric braids they each represent are equivalent.

At the moment, our projection allows for braids with multiple strands crossing at the same height, or even running one behind the other for an extended length :



Luckily, by pulling the green strand to the left, this is equivalent under $\approx_3$ to



This is encouraging. In fact, we can define a set of "nice" geometric braids ${GB_n}^{nice}$ where all of the crossings are separated out (no two occurring at the same height), the strand crossings are at exactly one point (no strands running behind another), and no pathological oscillation occurs (every geometric braid is equivalent to one whose strands are built out of finitely many straight-line segments — this comes directly from the definition of GB_n).

We may then give an equivalence relation that only allows deformation through other "nice" geometric braids, $\approx_n^{nice}$. And quite nicely, Dehornoy proves $B_n = GB_n/\approx_n$ is isomorphic to ${GB_n}^{nice}/\approx_n^{nice}$. Thus, every projection is equivalent to some "nice" projection, where nice-ness of a projection is defined almost exactly as it is for geometric braids. This is the second representation of braids.

We are able to pass from 3-d geometric braids to "nice" projections of those braids, losing no information about the equivalence classes. Thanks to the features of "nice" braids, we may now move to a simpler description of braids (our third representation). This is done by assigning a code to every braid projection. Beginning at the top, each crossing is noted : if a strand in position i $(1 \leq i < n)$ crosses over a strand in position $i+1$, this is called σ_i.

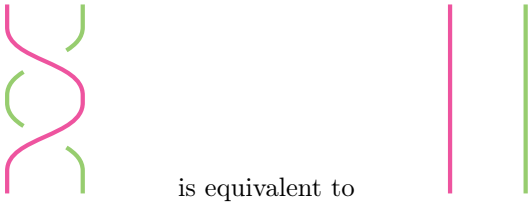
If it crosses under, we code this as $\overline{\sigma_i}$.

The three-strand braid at the center of the previous page is coded $\sigma_1\sigma_2\overline{\sigma_1}$. The braid at the very opening of this chapter is coded $\overline{\sigma_1}\sigma_3\overline{\sigma_2}\sigma_1\overline{\sigma_2}\sigma_1\sigma_3\sigma_2$.

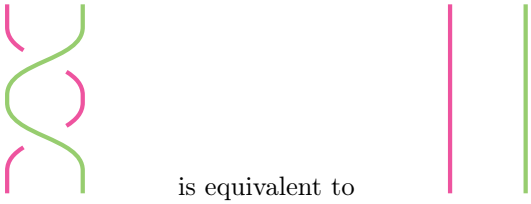
Formally, we define an alphabet $S_n = \{\sigma_i \,|\, 1 \leq i < n\} \cup \{\overline{\sigma_i} \,|\, 1 \leq i < n\}$, and then the set of S_n-strings is called S_n^*. (Here, an S-string is an ordered list of symbols from S_n). S_n^* also contains the empty string ε, which has length zero. ε codes the empty braid, which has no crossings. The empty braid on 4 strings looks like so:

Since there is always a nice projection of any braid, there is a code word for any braid. When are the braids described by two braid codes equivalent? We could translate back to the projection, and then to the geometric braid. That is unwieldy to work with. The idea here will be, as we have discretized the description of the braid, so too shall we discretize the relations that hold between braid codes representing braids equivalent under $\approx_n$.

Clearly, the following two relations hold:

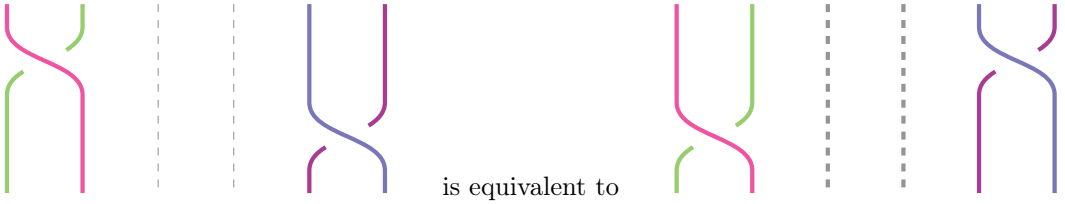


That is, for any i, $\sigma_i\overline{\sigma_i}$ is equivalent to ε.



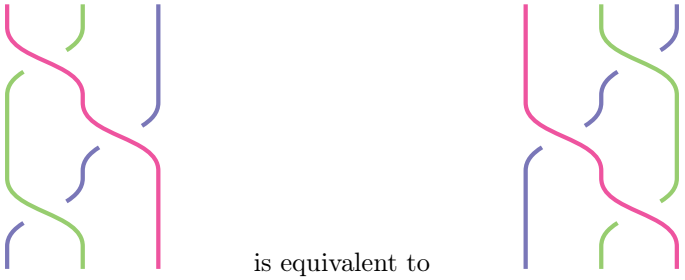
Similarly, for any i, $\overline{\sigma_i}\sigma_i$ is equivalent to ε.

Crossings far apart can be slid up and down :



If $|i-j| \geq 2$, $\sigma_i \sigma_j$ is equivalent to $\sigma_j \sigma_i$.

And the following relation holds when the crossings interfere with one another:



is equivalent to

So, $\sigma_i \sigma_{i+1} \sigma_i$ is equivalent to $\sigma_{i+1} \sigma_i \sigma_{i+1}$

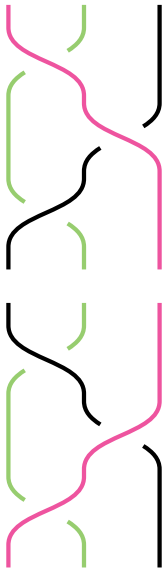
The fact that these hold is almost immediate. The trickier thing to show is that these suffice to encapsulate all possible isotopy/homotopy deformations. A full proof of this fact is beyond the scope of this overview. It is done by meticulous casework on possible moves on a braid projection, where the strands in the projection are made of finitely many straight-line segments.[2]

We thus have a set of braids on n strands for any natural number n, which can be described in three manners (as equivalence classes of either three-dimensional diagrams, two-dimensional projections, or strings of symbols). We next look to show an algebraic structure on this set. Two braids on the same number of strands may be concatenated by appending the bottom ends of one to the analogous upper ends of the next one. This operation is well-defined on projection diagrams and also on codes for braids (here, in the usual string concatenation sense). Notably, concatenating the empty braid either above or below any braid (geometric, projection, or code) leaves the braid unchanged. Further, concatenation of braids is associative. Thus, we see the beginnings of an algebraic structure on the set of braids — we have a binary operation, an identity element, and the associativity property for our operation. Now, let's look for inverses.

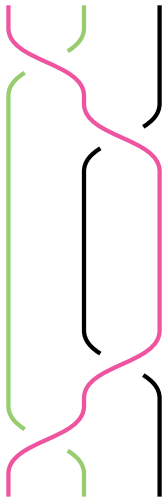
We can "untangle" any braid by appending another given braid below it. Geometrically, we may do this by appending a vertical mirror-image of the braid below it.

[2]Dehornoy gives another theorem stating that every braid is equivalent to one whose projection consists of a finite number of straight-line segments

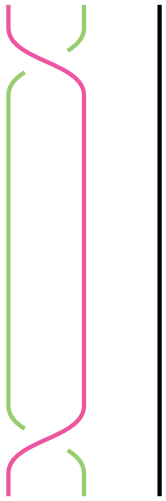
As an illustration, consider the braid $\sigma_1\sigma_2\overline{\sigma_1}$ and its mirror image $\sigma_1\overline{\sigma_2\sigma_1}$:



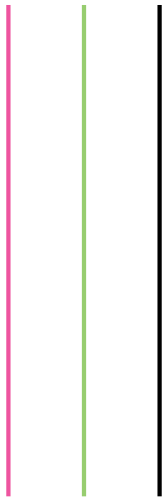
We may then untangle the braid, working from the middle out. First, untangling the green and black:



Next, the pink and black:



And then that very last step of pulling the pink off of the green give us the identity braid:

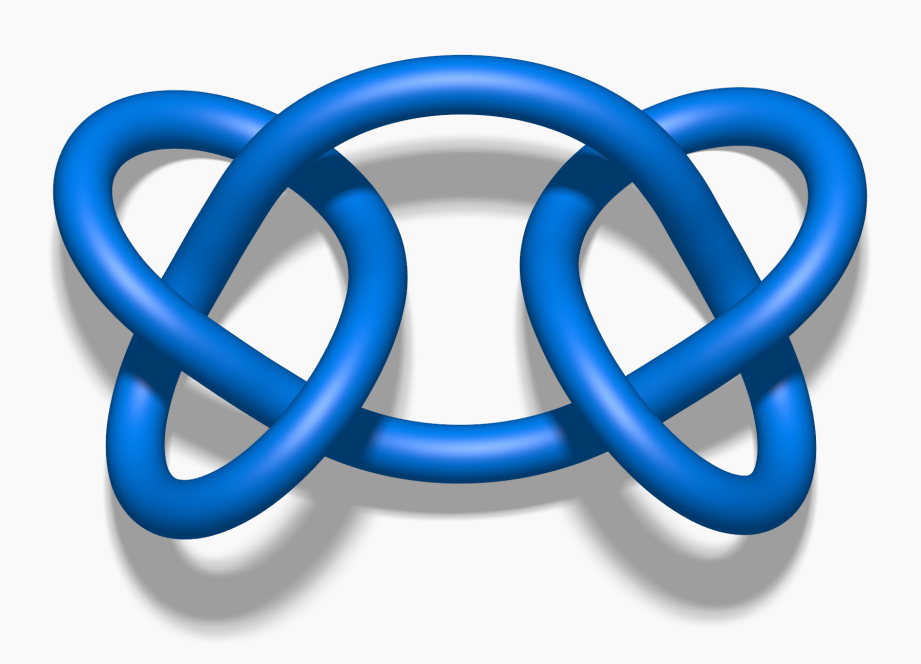


Voila! So, we have a geometric procedure for "undoing" any braid. In terms of braid codes, this corresponds to reversing the order of the symbols in the code, and for each, switching out its overline : every σ_i becomes $\overline{\sigma_i}$ and vice-versa.

We now have a binary operation (concatenation), an identity element (the empty braid), and inverses (mirror image). Thus, braids form a group! The elements are equivalence classes of a given geometric braid, projection, or code.

1.1.2 Not a Knot!

Let us note here briefly the mathematical definition of a knot,[3] and the relationship knots have with braids. Intuitively, one can consider a rubber string, glued together at the ends, which can be stretched, shrunk, twisted, and moved about so long as it never self-intersects.

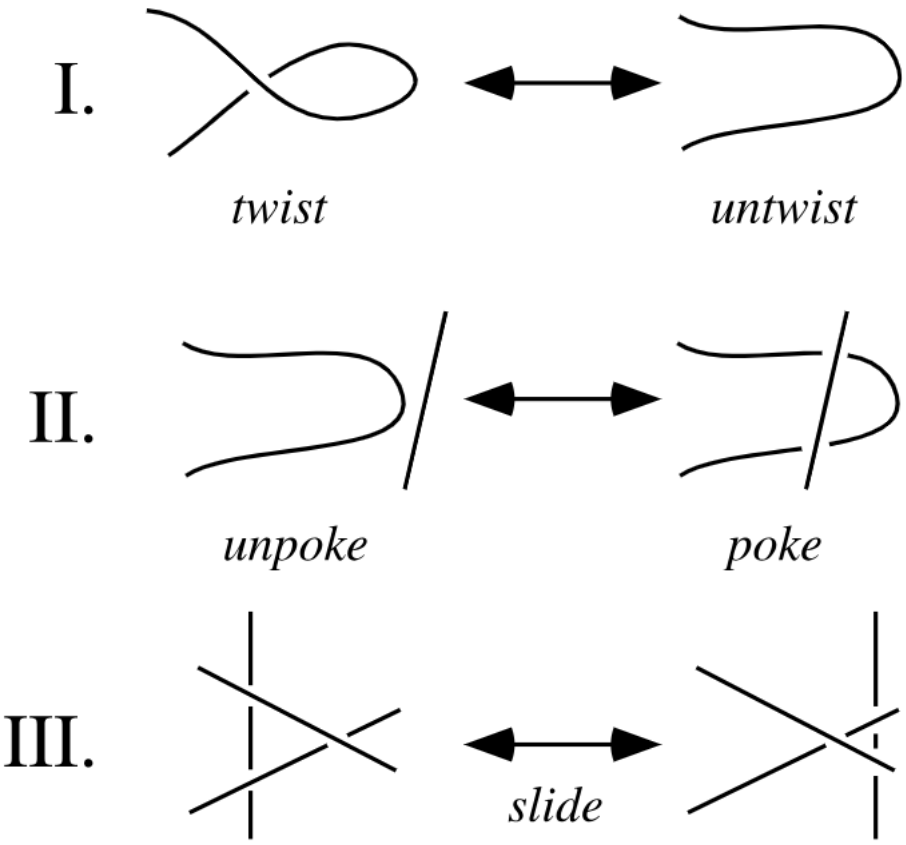


That is, a knot is a closed path in 3-dimensional space which does not self-intersect; namely, an embedding of the circle (S^1) into $\mathbb{R}^3$. Much like braid equivalence, knot equivalence is defined by ambient isotopy. The question to determine if two knots are equivalent is thus : can I twist $\mathbb{R}^3$ – the ambient space – so that my knot lands up on top of the goal knot?

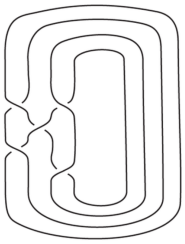
Formally, a knot A (meaning a function $A : S_1 \to \mathbb{R}^3$) is equivalent to a knot B iff there exists some continuous $F : [0, 1] \times \mathbb{R}^3 \to \mathbb{R}^3$ such $F_0 \circ A = A$ and $F_1 \circ A = B$. So $F \circ A$ is a continuous sequence of knots.

We may also consider the projection of a knot. Here again, there are a few basic moves which preserve knot equivalence, called Reidemeister moves.[4]

[3] image from Wikipedia : Square Knot
[4] image from Wolfram : Reidemeister Moves



In the late 1920s, J. W. Alexander showed that every knot can be represented as the closure of a braid [2]. This simply means glueing the top end of a strand to the bottom end of the strand which lands below it. (Image from Cheng and Lin, [9])



Knots do not have a concatenation operator and thus lack the group structure found on braids. The mathematics used to study knots has become distinct from the algebraic approach used for braids. While it is possible to solve the knot isotopy problem (are two given knots equivalent) in the general case, current known approaches to even the unknot problem (is a given knot equivalent to the unknot, which is just a plain circle) do not run in polynomial time [19]. So, knots are often compared using knot invariants — properties that remain the same for all equivalent knots. These invariants are designed to be computationally efficient, and run in polynomial time. Many are not complete — that is, non-equivalent knots may share the same knot-invariant value. But if two braids have different invariant values, one can be certain the two knots are not equivalent.

Braids provide one such invariant, and a formalized theory of braids thus does help make progress on knots. Ultimately, despite both being mathematical models of strings in space under (perhaps restricted) ambient isotopy, the type of mathematics used to study each is largely distinct. (We shall see in the historical section that it was not always so!)

1.1.3 Applications

For a brief expedition into the wonderful world of braids, let us look at a few examples of their use. Certainly, we make braids with ribbons and dough. Chemists have discovered naturally-occurring braided structures in molecule chains, and have used this insight to attach lab-synthesized molecule chains to one another via braiding, not atomic bonds [21]. They use the mathematical language of braids to communicate these techniques.

Robots with power cords can get tangled. One can model this situation with braids, and efficiently outlaw paths which lead to messy entanglements [7]. Configuration spaces, associated with robot motions and factory-floor movement, can be studied via their associated braid group [18]. One considers their two-dimensional motion across a floor, and make the third dimension time. Strings in a braid cannot intersect, and neither can the robots or factory workers.

If we stay in the realm of strings, braids are being used to formalize machine knitting by researchers out of CMU and UW. One can consider a braid with extra generators (these generators are small knot-like objects embedded into the braid). This is called a "fused braid." The normal braid relations are preserved, and a few more are added. A solution to the original braid problem can provide insights into an eventual algorithm for the fused-braid model. Algorithms are being developed which function well empirically, so there is a market for proofs!

Braid groups also appear devoid of any connection to twisted strings. Their properties of non-commutativity along with a solvable word problem make them a tempting candidate on which to base post-quantum security protocols. An initial, straightforward approach used the conjugacy problem (given braids A and B, does there exist a braid G such that $A = GBG^{-1}$?) for encryption [22]. Alas, this fell victim to heuristic, probabilistic attacks. Nevertheless, new systems based on different braid problems are being theorized [8].

1.2 The Lean Theorem Prover

Lean is a functional programming language and interactive theorem prover based on dependent type theory. Leonardo de Moura launched Lean in 2013; the language is now on version 4. Mathematicians worldwide use Lean to formally verify proofs across mathematics, from undergraduate basics to the cutting edge of research. A large collaborative project to formalize as much of mathematics as possible in one central repository, named Mathlib, is ongoing, and currently counts 367 contributors. [5] From algebraic geometry to topology to analysis, Mathlib is ever-growing into all corners of mathematics.

As far as I can tell, there has been no formalization of the braid group in Lean, nor in any other theorem prover. Coxeter groups, another type of presented group which is similar in structure to the braid groups, are the object of an ongoing formalization project. [6] Knots and a few invariants thereof were formalized in Isabelle/HOL by Prathamesh in 2015 [28]. Alas, the mathematics used to study braids (presented groups, localizations, orderings) is of an entirely different sort than that used for knots (polynomial invariants). While it is not formal mathematics, it is worth noting that there are a number of computational packages for working with braids, for example one in MATLAB [31].

[5] Mathlib statistics

[6] https://www.majiajun.org/formalizing-coxeter-group-hecke-algebra-and-kazhdan-lusztig-theory-in-lean/

This mathematics behind braids connects deeply with fundamental concepts in abstract algebra. The project of defining braids led to the construction of presented monoids and rewrite systems in Lean. These arise from the study of computability, and hopefully will find much re-use in future projects : confluence, Post-Markov theorem, Knuth-Bendix completion, etc.

In terms of braids themselves, much current research is focused on developing faster algorithms to resolve the braid word problem. Having a formal foundation for braid definitions and basic properties should aid in verifying the correctness and termination of these algorithms.

One can in fact implement these algorithms in Lean and execute them using the `#eval` function. But Lean is not limited to computable functions or constructive proofs. One can choose to use Lean with constructive or classical logic. When proving correctness of an algorithm, for example, the algorithm must be computable but the proofs of correctness and termination need not be.

1.3 Project Overview

This thesis is a report on the progress made to date of a larger project to implement a verified algorithm for solving the braid word problem[7] in Lean. I am working on Dehornoy's subword-reversing algorithm, and throughout have followed his textbook [13]. I have broken the project into four stages:

1. Defining braid groups
2. Defining a braid monoid and connecting it to a braid group
3. Defining a grid structure and connecting it to re-writing
4. Implementing the algorithm

Items 1 and 2 have been formalized; they are discussed in chapters 3 and 4, respectively. Item 3 requires a novel proof in three stages; in chapter 6 I give a proof sketch of the first stage, rigorous pen-and-paper proof of the second stage, and formalized proof of the third stage. I have not yet begun to formalize item 4; I have given a sketch of the procedure in chapter 5, along with the formalization of a number of requisite mathematical structures.

Beyond this, chapter 2 opens with a historical overview of how braids became the algebraic object they are today.

[7] Details will follow; essentially, given two braid codes, do they represent braids equivalent under ambient isotopy?

2 The History of Mathematical Braids

We give here a sketch of the history of mathematical ideas of braids. Knot theory will be discussed to give context to the development of braids, and when it is intrinsically linked (i.e. Alexander's theorem that every link is a closure of some braid). The primary focus, however, will be on braids. We aim to trace the emergence of an algebraic way of thinking about these inherently physical, topological objects.

2.1 Early History

As far as I have seen, the first mathematician to mention braids is Vandermonde (1735-1796), a French violoinist-turned-mathematician [3]. His approach to the problem is a precursor to modern topology. Topology had not yet been named as such – at the time, it was called "le problème de situation" or "analyse situ." Thus, Vandermonde's 1771 paper was entitled *Remarques sur les problèmes de situation* [**vandermonde_mathematicien_remarques_1771**]. It discussed two related problems : describing intertwined strings in 3-dimensional space, and the motion of a knight on a chessboard (compare his notion of braids as the paths of knights on a chessboard to today's research on braids as the paths of robots in factories!).

Vandermonde approaches the problem from a practical standpoint, considering how an artisan might understand and communicate his process:

> Quelles que soient les circonvolutions d'un ou de plusieurs fils dans l'espace, on peut toujours en avoir une expression par le calcul des grandeurs; mais cette expression ne seroit d'aucun usage dans les Arts. L'ouvrier qui fait une tresse, un réseau, des nœuds, ne les conçoit pas par les rapports de grandeur, mais par ceux de situation ce qu'il y voit, c'est l'ordre dans lequel sont entrelacés les fils. (p. 566)
>
> [Whatever the twistings and turnings of one or more strings in space, one can always describe them by magnitudes; but this expression would be of no use in the Arts. The worker who makes a braid, a fabric, a knot, conceives them not by relations of length, but by those of relative position, the order in which the the threads are interlaced.][8]

We see an abstraction from properties of the physical strands like length and exact coordinate position in space. Rather, the importance is on their relations and intertwinings with one another. Vandermonde claims an artisan understands his work by the steps of twisting and weaving the strands, and thus attempts to give a mathematical notation for such a procedure:

> Il seroit donc utile d'avoir un système de calcul plus conforme à la marche de l'esprit de l'ouvrier, une notation qui ne représentât que l'idée qu'il se forme de son ouvrage, & qui pût suffire pour en refaire un semblable dans tous les temps. Mon objet ici n'est que de faire entrevoir la possibilité d'une pareille notation, & son usage dans les questions sur les tissus de fils. (p. 566)
>
> [It would thus be useful to have a mathematical system more closely mirroring the thought process of the artisan, a notation which represents his own conception of his work, and could serve as instruction to recreate the object years later. My goal here is to show the possibility of such notation and its application to textile geometry.]

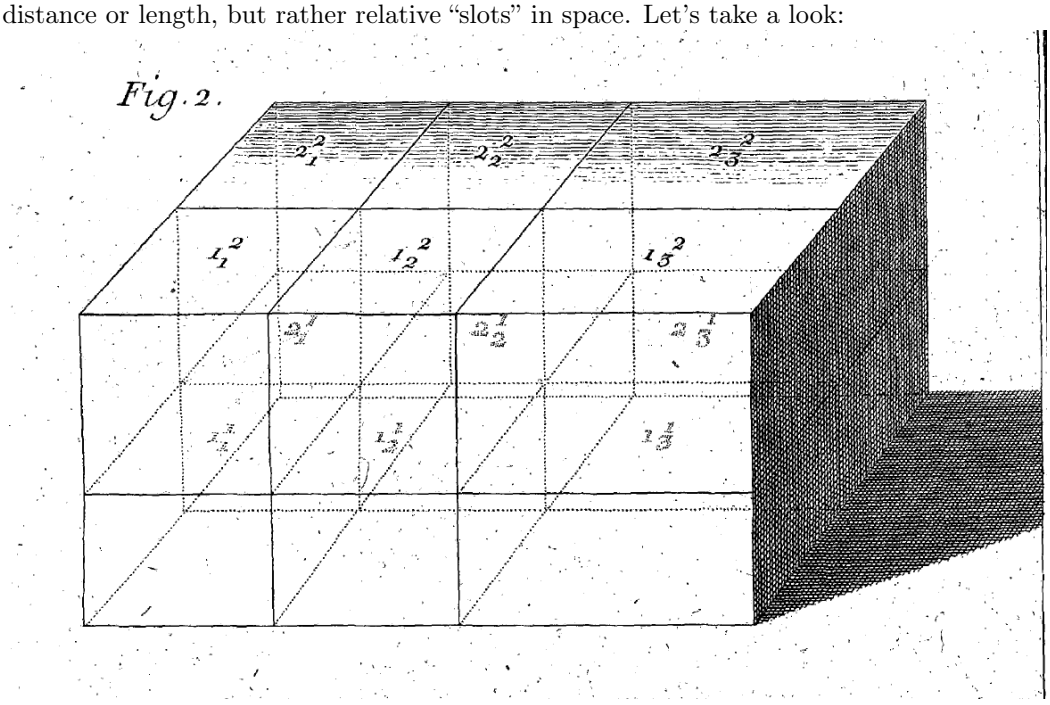
[8] this and other French translations are my own; the translations are loose and not intended for precision

He desires a notation for strings in space which takes into account topological properties moreso than geometric ones of distance and angles. And future generations of mathematicians will heed his call!

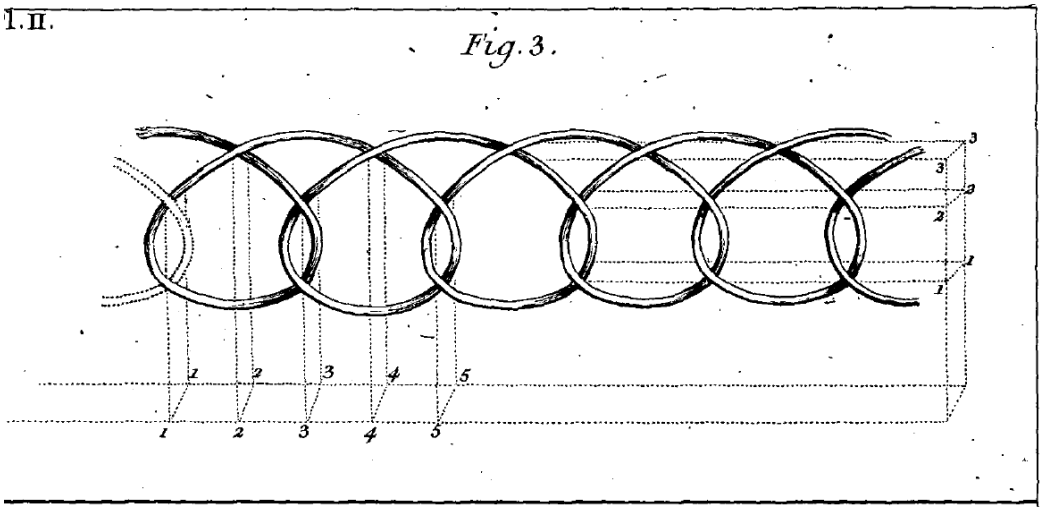
Vandermonde considers this process as a reduction, a way of abstracting a messy problem into a simpler, more manageable form which preserves just the relevant information:

> je le réduis à une simple opération d'Arithmétique, faite sur des nombres qui ne représentent point des quantités, mais des rangs dans l'espace. (p. 566)
>
> [I reduce this to a simple arithmetic operation, based on numbers which represent not magnitude, but denote a chunk of three-dimensional space.]

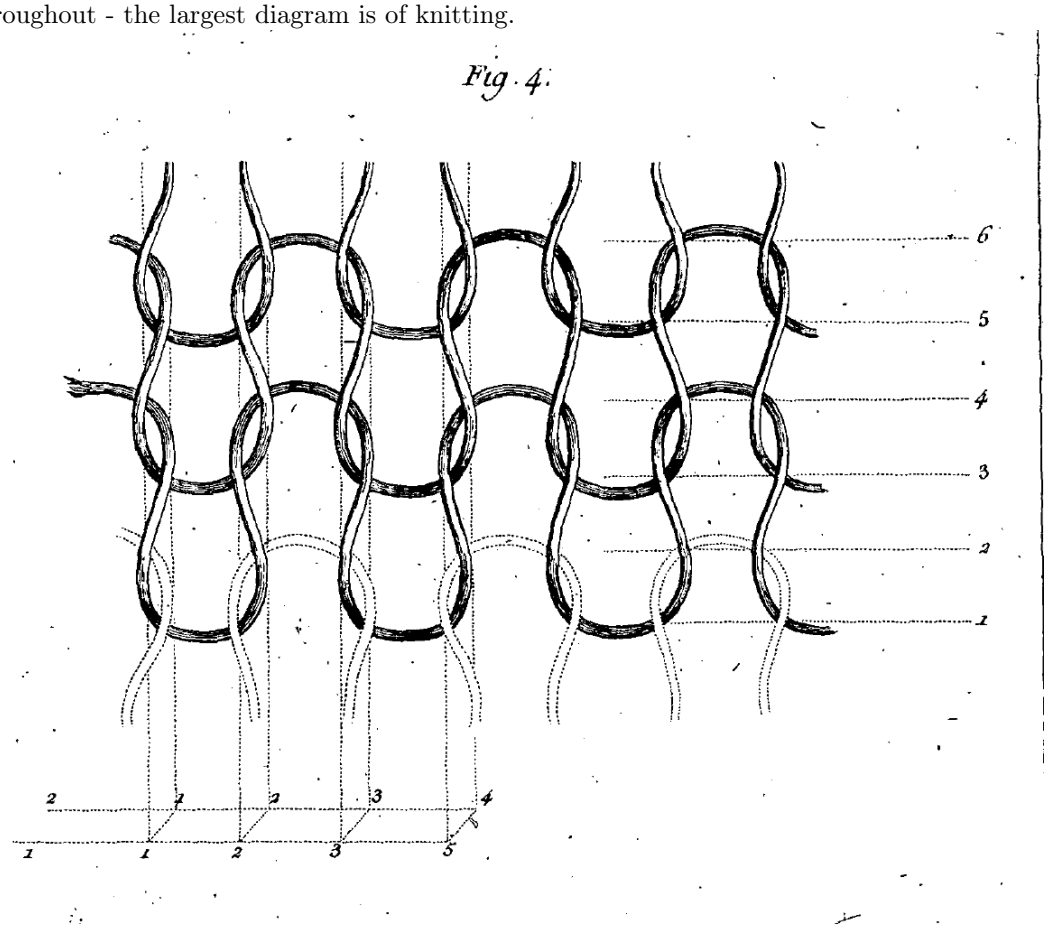
Although he will give numbers to represent positions, he is clear that his numbers do not relate to distance or length, but rather relative "slots" in space. Let's take a look:



Space is divided into rectangular prisms. It is not the relative size of the prisms that matters, but their relative position to one another. With that setup, he can consider a twisted string moving through space :



We note that it is not the crossings that are labelled — rather, at a crossing, the position of each involved string is noted. This method is used to keep track of over- versus under-crossings, which are drawn distinctly on the diagram. Vandermonde's emphasis on the practical fiber arts occurs throughout - the largest diagram is of knitting.

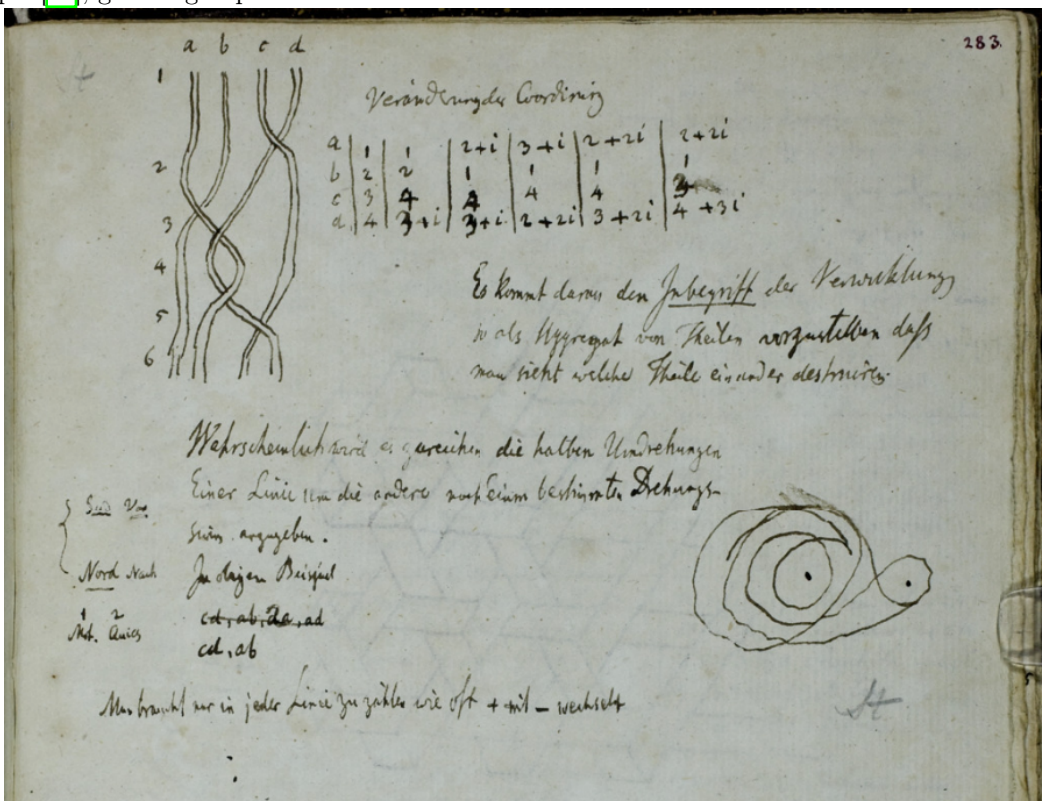


Next up on the historical run-down is Gauss (1777-1855). He has an interest in topology ; he

bemoans that

> Of the geometria situs, which Leibniz foresaw and into which only a pair of geometers (Euler and Vandermonde[9]) were granted the privilege of taking a faint glance, we know and have, after a century and a half, little more than nothing. (Werke, Vol. V, p. 605. Cited and translated in Epple [15])

And so, he takes up the task! A single page of his notebook [17], beautifully analyzed by Moritz Epple [15], gives a glimpse into Gauss's view of braids :



No date is given, but based on the surrounding pages, Epple estimates this was written between 1815-1830. On the top-left there is a sketch of the braid, whose strands are labelled a, b, c, d. Five crossings occur in the braid; the regions between them are labelled 1 through 6. This is done to keep track of the string positions before and after each crossing.

To the right, Gauss made a table showing the change in position of the strands. He also denoted the twists using imaginary numbers. This seems to have been a work-in-progress; no clear rule for which strand in a twist is marked with an i seems to hold. In the first crossing, where strand 3 crosses above 2, we obtain 1 2 4 (3+i). But then the next, when 1 crosses over 2, is (2+i) 1 4 (3+i), not 2 (1 + i) 4 (3+i). In any event, Gauss was the first to tabulate the crossings, and study braids in that way.

Below the table, he writes

> What matters is to represent the whole [Inbegriff] of the entanglement [Verwicklung] in such as way as the aggregate of its parts that one sees which parts destroy one another. (translation from Epple)

[9] Based on Gauss's letters to Olbers in 1802, we know he had read *Remarques sur les problèmes de situation*

These "parts" may be seen as a precursor to the idea of generators : an elementary part here is a single crossing. By representing the whole as an "aggregate of its parts" we see the idea of concatenation. The idea of one part "destroying" another is an early idea of braid inverses : which braids or sub-parts of a braid can be untangled into the empty braid? This brief note shows Gauss had started to think of braids in a sort of algebraic manner, by decomposing and combining discrete parts.

The next remark, in the middle of the page, reads

> Probably it will suffice to list the half twists one of line around the other according to a certain sense of rotation. (translation from Epple)

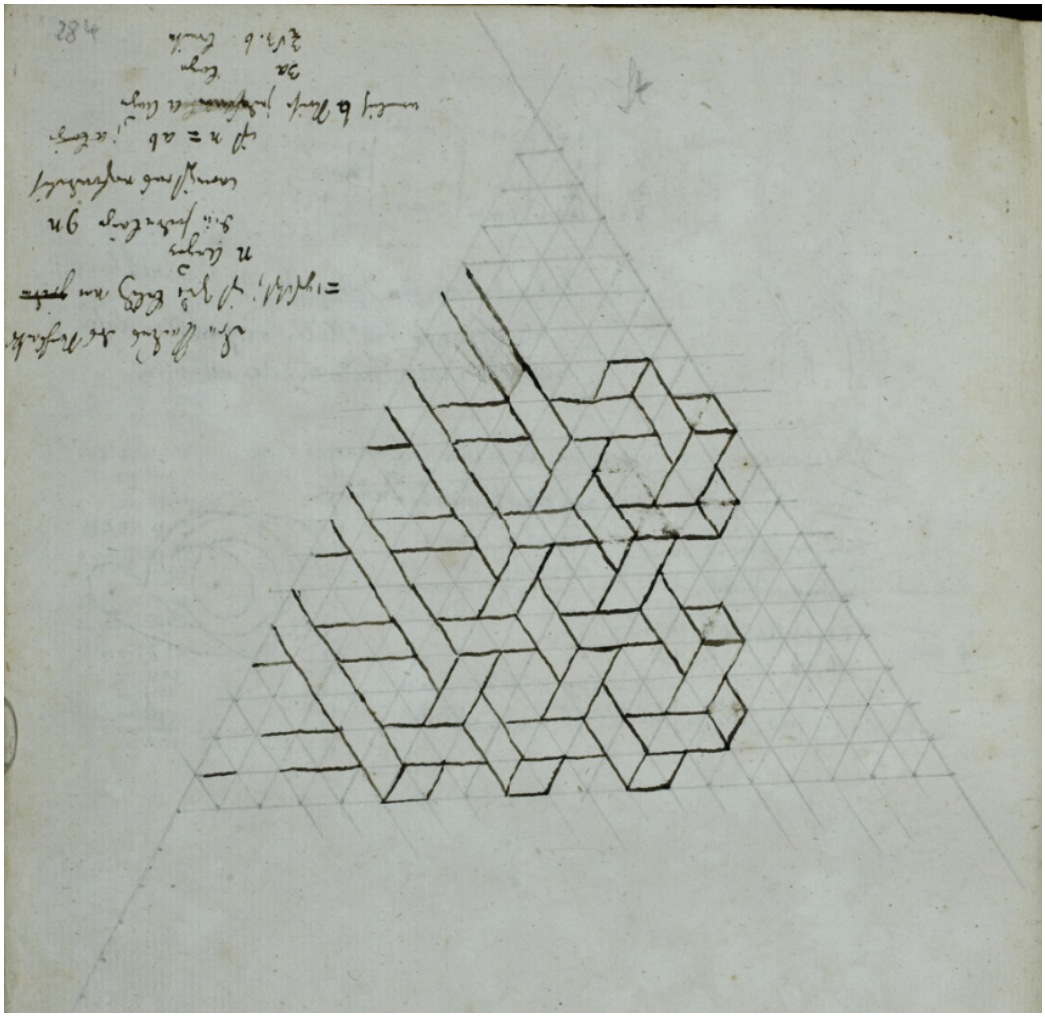
And so Gauss does consider orientation as an important part of structure. To the side, he starts to play with some orientations - "Sud", "Nord", "Vor" and "Nach" ("South", "North", "Before", "After").

He continues on with a new method for denoting "Im obigen Beispiel" [the above example]. Gauss tries a new method of denoting the strand crossings, based on the names of the strands (not Artin's method based on the current position of the strand) : cd, ab, da, ad. This he scratches out, and begins to start again. At the bottom, he conjectures:

> Man braucht nur in jeder Linie zu zählen wie oft + mit - wechselt [You just need to count in each line how often + alternates with -] (my translation)

This, paired with the drawing on the lower-right (showing the linking number of two strands - the number of times one wraps around the other) suggests he was thinking to count the negative/positive twists a strand experiences.

One wonders in what context Gauss began to think of braids. Although he did doodle and study knots, this came later in his life, well after the braid drawing was sketched. We see on the next page a drawing of hexagonal weaving; this may suggest he was indeed thinking of Vandermonde's work on weaving patterns.



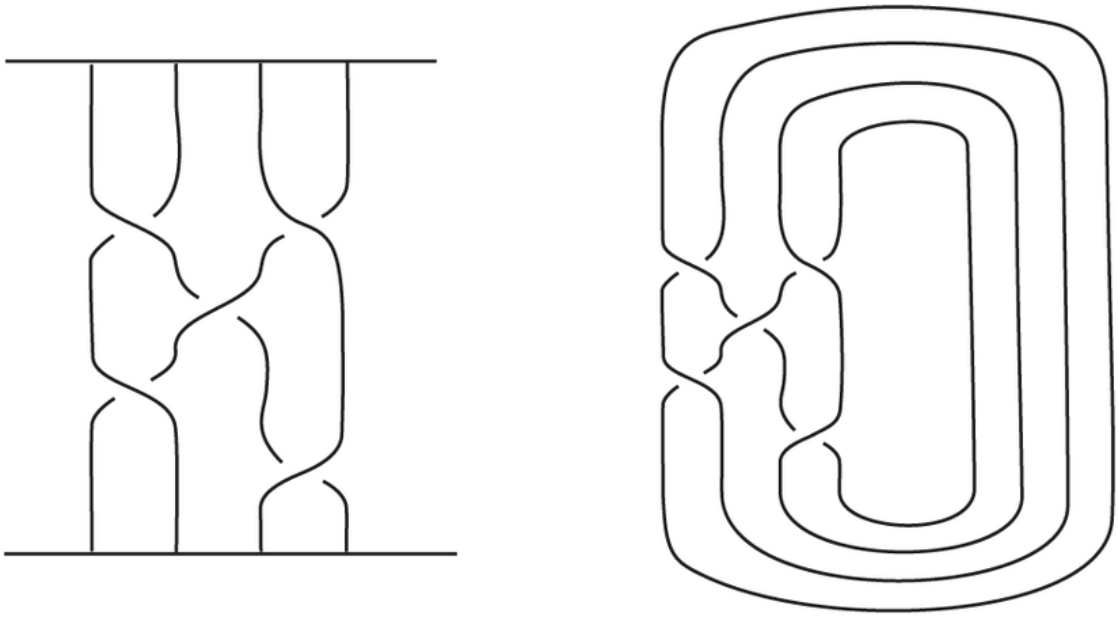
Moritz Epple argues that his interest in braids came not from their connection with knots, but rather "from the exact sciences of of astronomy, geodesy, and electromagnetism" (p. 52). At the time, Gauss has been trying to predict asteroid orbits, as well as some working out calculations in electromagnetism which depended on wires coiling around one another. Again, the practical applications led to an attempt to define, describe, calculate, or otherwise work mathematically with braids.

Gauss's student Johann Listing (1808-1882) picked up on Gauss's interest in knots, publishing a book "Vorstudien zur Topologie" [23]. Notably, he is the first to use the word topology; unsurprisingly, the mathematical focus is on topological, not algebraic, ideas about knots. Twenty years later, physicists once again took an interest in knots, due to a theory that atoms had a knotted structure proposed by Lord Kelvin [32]. His friends James Clerk Maxwell and Peter Guthrie Tait (1831-1901) spent considerable effort looking into knots; Maxwelll calculating linking integrals and Tait tabulating knots [29]. This tabulation work occupied a great deal of time and effort amongst a number of mathematicians — while Tait is often given all the credit, his contemporaries Thomas Kirkman and Charles Little also made great progress, both independently and late in collaboration with Tait [29]. While explicit study of braids was not apparent during this period, the knot theorists kept the field alive until the early 1900s, when a new wave of knot theorists picked up braids once

again.

2.2 Journey to Algebra

As we move into the twentieth century, braids have still not appeared by name in any writing. In 1923, the American James Waddell Alexander (1888-1971) from Princeton gives a description of what we would now call as "closed braid". The idea is to take a braid, and glue together the top ends to the bottom ones, forming either a knot or a link (a knot with multiple components). Here is a planar representation of such a closed braid, (taken from paper on braid representations of DNA by Chen and Lin) [9] :



Alexander relates this to a link in the following manner in his paper "A Lemma on Systems of Knotted Curves" [1] :

> Consider a system S made up of a finite number of simple noninteresting closed curves located in real euclidean 3 space. The curves S may be arbitrarily kntted and linking, but we shall assume, in order to simplify matters as much as possible, that each is composed of a finite number of straight pieces. The problem will be to prove that the system S is always topologically equivalent (in the sense of isotopic) to a simpler system S', where S' is so related to some fixed axis in space that as a point P describes a a curve of S' in a given direction the plane through the axis and the point P never ceases to rotate in the same direction about the axis. (pg. 93)

S' is thus a closed braid, if not described in such words. We note the discrete nature of this definition and its proof : the division of the curve into finitely many straight-line segments. Then operations merging or splitting these toothpick-like segments are used to prove equivalence.

Epple writes that in the 1920s and 1920s, work in knot theory was presented as "built around diagram combinatorics rather than manifold topology (p. 151)" [14]. It was a popular topic — in fact, a bit of an arms race. Alexander and Reidemeister, a German mathematician from the same time period, developed similar knot invariants. Both were presented in a modernist, formal algebraic manner. Was it true that this new combinatorial, diagrammatic methodology led to new insights? Epple argues it was not. He describes in detail how each invariant was derived from geometric notions. Yet these geometric notions were not presented as primary in either paper. Alexander wanted to professionalize mathematics, and so he wrote in this algebraic manner to fit in with the culture at Princeton.

Reidemeister was mentored by great German minds like Dedekind (who he fondly referred to as an uncle[10]). Through Dedekind, one can trace Reidemeister's academic lineage back to Gauss and his abstract approach to mathematics. Dedekind develops in his paper "Was sind und was sollen die Zahlen" [11] an abstract method "dessen Absicht auch von Gauß gebilligt wurde [whose intention/design was also approved by Gauss]", as he proudly points out. Dedekind's structuralist program grew from this methodology, influencing Hilbert and ushering in his axiomatic method, as seen in his "Grundlagen der Geometrie." Reidemeister, having had a close personal connection with Dedekind at the start of his career, also follows in this abstract tradition.

As part of a Vienna school of philosophy, [11] Reidemeister wrote a philosophical article entitled "Exact Thinking" [30] which emphasized the importance of studying the "combinatorics of sign systems" (Epple, 156). Hence, Reidemeister's philosophical circle shaped his presentation of the material. Moreover, Reidemeister had a deep interest in knots (he did fundamental work; the "Reidemeister moves" for showing knot equivalence are named after him). Surely he would have been aware of the work of Max Dehn and Paul Heegaard, which, according to the historian of knots Peter van der Greind [33], "showed the knot problem could be formulated entirely in terms of arithmetic, i.e. combinatorics." Whether or not their original work was discovered via geometrical reasoning (another future avenue to explore), their presentation in terms of combinatorics certainly would have encouraged Reidemeister to follow suit - if not in discovery, at least in the write-up!

Now we have made it to the late 1920s, in which appears the father of the modern theory of mathematical braids : Emil Artin. In his first paper *Theorie der Zöpfe* (1926), he gives a geometric description of braids, an algebraic presentation, as well as a proof that the two coincide. Notably, Artin also had a connection through Vienna; he spent time there and was close with Otto Schreier, another mathematician working on braids. His methodology was thus in many ways similar to that

[10] thank you Wilfried for the information!

[11] He was the organizer of and moderator at the 1930 Königsberg meeting with Carnap, von Neumann, Brouwer, and Gödel)

of Reidemeister. Artin's overarching goal is to "arithmetize" the notion of braids (p. 50). Twenty years later, having moved from Germany to Princeton, he returns to the subject, unhappy with the proofs in his first paper. He opens his 1947 *Theory of Braids* with the following :

> A theory of braids leading to a classification was given in my paper "Theorie der Zopfe" ... Most of the proofs are entirely intuitive. That of the main theorem in [Section] 7 is not even convincing. It is possible to correct the proofs. (p. 101)

Michael Friedman has made a careful comparative study of the two papers. He highlights, first, that

> in order to facilitate an understanding of what the composition of braids looks like in the braid group, Artin immediately introduces a restriction ... —one that is completely visual. As he notes, a braid should not turn in reverse: 'In Figure 1 [see below] a weaving [Geflecht] is drawn as an example, one that we do not consider a braid [Zopf]' (Artin 1926, 48). Such a restriction is given no further algebraic interpretation.

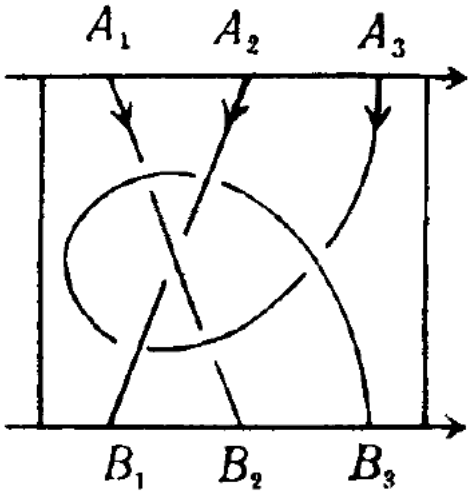


Fig. 1.

While there is a way to describe this in terms of intersections with every z-plane, Artin was working with a system (projection diagrams) which could not easily represent this notion. The 1947 paper recognized this, and made the switch to discussing braid coordinates.

The second use of geometric intuition occurs when Artin defines the braid relations in the braid group. Friedman notes,

> The proof of these relations is entirely visual. Thus, for example, Artin explicates that one can 'extract from the [below] figures' the relation $\sigma_{i+1}^{\pm 1}\sigma_i = \sigma_i^{\mp 1}\sigma_{i+1}\sigma_i^{\pm 1}\sigma_{i+1}^{\pm 1}$ from which he induces the relation $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$. The proof relies on the diagrams depicting what happens when one 'shifts' a crossing σ_i from one side of the crossing σ_{i+1} to the other.

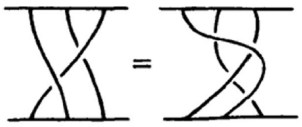


Fig. 5.

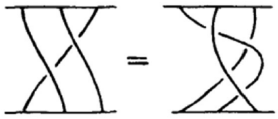


Fig. 6.

This was completely re-worked in the 1947 paper, but it hadn't lay dormant in those two decades! Between the two papers of Artin's, a small cottage industry of mathematicians worked to make the proofs of the first more rigorous, and to also provide alternate descriptions of a braid more suitable for algebraic proofs. Burau discussed matrix representations [6]; Bohnenblust worked out algebraic versions of a number of Artin's proofs [5] (and in fact his article helped supplement Artin's 1947 work). Bohnenblust, having worked closely with Artin, gives this introduction to his (Bohnenblust's) work on braids:

> These results were fully recognized in the paper of Artin mentioned above [1926] and led in particular to a classification of braids. The proofs, however, are partially intuitive. In a recent paper Artin returned to this question making use of a more direct approach which permits an elegant and completely rigorous treatment of the problem. ... By its nature the problem is geometrical and throughout the major part of his paper Artin makes extensive use of geometrical considerations. In the present paper a purely group-theoretical problem is considered.

So, although Artin had obtained rigorous proofs in his 1947 paper, Bohnenblust still strove to make those proofs *algebraic* (as well as providing some purely-algebraic proofs to supplement gaps in Artin's work, which Artin had intentionally skipped, knowing Bohnenblust was also publishing).

2.3 The Leningrad Research Group

So far, we have only discussed European and American mathematicians. What was going on behind the Iron Curtain during all this excitement over braids?

Let's set the stage : Leningrad, 1930s. Markov (Jr.), Ivanovsky, Weinberg. These three mathematicians had some contact with the western world; they had read Artin's work. However, communications were limited under Stalin. Western scientists struggled from an inability to read Russian, in which Stalin insisted scientific works be published; Soviet scientists suffered from government restrictions on communication [20]. However, through the perseverance of Pavel Aleksandrov, an international topological conference was organized in Moscow in 1935 [4]. A pamphlet found nestled in Kolmogorov's papers (pages 590-593) [36] listed the scheduled speakers, including James Alexander, Kurt Reidemeister, Egbert van Kampen (whose diagrams we will see appear in braid theory!), A. A. Markov (Jr.), and A. Ivanovsky.

It appears that Ivanovsky did not end up giving a talk [4]. Nevertheless, this conference put Markov and his collaborators in contact with Western mathematicians. Markov's talk was titled "On the Free Equivalence of Closed Braids" so certainly some talk of braids occurred!

Let's now look a bit more specifically at what work the Leningrad group did. Andrey Andreyevich Markov (1903-1979), son of the famous Andrey Andreyevich Markov, led a small group of researchers

in braid theory in Leningrad in the 1930s. The papers produced by this group discuss algebraic braids with no mention of geometry whatsoever. Nevertheless, they were viewed as belonging to topology. In a review of the past 30 years of Russian mathematics from 1949 (covering 1917-1947) [35], we read

> Некоторые из работ советских математиков о группах, заданных определяющими соотношениями, относятся к топологии, т. е. лежат вне пределов настоящего обзора; таковы, например, работы А. А. Маркова о группам кос.
>
> "Some of the works by Soviet mathematicians relating to finitely presented groups pertain to topology; they lie outside the scope of this review. Such are, for example, the works of A. A. Markov on braid groups" (my translation)

Markov wrote two papers on braids : The first was "Über die freie Äquivalenz der geschlossenen Zöpfe" [26] in 1936, a write-up of his conference talk. Here, he picked up a thread from Alexander's work showing every link can be written as a closed braid. Notably, the closure of the conjugation of any braid (given a braid word w, conjugation by another word a makes the word awa^{-1}) gives the same link as the closure of the braid itself. Intuitively, a and a^{-1} can be slid "around" to meet one another in the back and cancel out. Markov gives some more moves that do not change a closed braid, proceeding in a purely algebraic manner. At the end, he conjectures that the moves he has described are those necessary and sufficient to transform a closed braid to any other equivalent closed braid.

This is picked up and proved by his collaborator N. M. Weinberg in a brief report "Sur l'équivalence libre des tresses fermées" from 1939 [34]. We know very little about Weinberg; he had also published on topology in 1941 in his paper "On the regular closure of topological spaces" [35].

Markov's second paper appeared in 1945, entitled "Foundations of the Algebraic Theory of Tresses" [25], and beats out Artin and Bohnenblust's 1947 update to the 1926 paper by about two years. Markov describes the situation as such :

> Появление настоящей статьи вызвано следующими обстоятельствами. Рассуждения Artin'a основаны на геометрической интуиции и не без упречны в отношении строгости. Желательно было иметь более строгое доказательство основных результатов Artin'a. Может быть, можно было бы достигнуть полной строгости, дополнив как-либо рассуждения Artin'a. Автор, однако, предпочел иной путь. Именно, в этой статье геометрия полностью исключена, и теория кос строится с самого начала на чисто алгебраическом основании.
>
> The appearance of this article is due to the following circumstances. Artin's reasoning is based on geometric intuition and is not impeccable in terms of rigor. It would have been desirable to have a more rigorous proof of Artin's main results. Perhaps it would have been possible to achieve complete rigor by somehow supplementing Artin's reasoning. The author, however, preferred a different path. Namely, in this article geometry is completely excluded, and braid theory is built from the very beginning on a purely algebraic foundation. (my translation, with assistance from A. Malyutin via e-mail correspondence)

Markov worked along with Ivanovsky to give a new solution to the braid word problem. He describes,

> Кроме того, данное Artin'oM решение проблемы тождества пред ставляется недостаточно удобным для дальнейших исследований в этой области, особенно при изучении

проблемы сопряженности. Решение это не проливает света на структуру группы кос. Поэтому в настоящей статье мы даем другое решение проблемы тождества, которое, как нам кажется, свободно от этих недостатков. Это решение, найденное А . Ива новским, основано на установлении некоторой нормальной формы кос. Единственность этой нормальной формы и изоморфия АгНп'ова пред ставления группы кос доказываются ниже одновременно. Нормальная форма дает важные сведения о структуре группы кос, так как она позволяет построить нормальный ряд этой группы с известными факторами.

In addition, the solution of the identity problem given by Artin does not seem to be convenient enough for further research in this area, especially when studying the conjugacy problem.[12] This solution does not shed light on the structure of the braid group. Therefore, in this article we give another solution of the identity problem, which, as it seems to us, is free from these shortcomings. This solution, found by A. Ivanovsky, is based on the establishment of a certain normal form of braids. The uniqueness of this normal form and the isomorphism of the Artin representation of the braid group are proved below simultaneously. The normal form gives important information on the structure of the braid group, since it allows one to construct a normal series of this group with known factors. (my translation)

As we can see, this is done using purely group-theoretic terminology and methods. A mysterious "Ivanovsky" is mentioned here. Records are scant; Ivanovsky was a post-graduate student of Markov's who was killed in the Second World War (page 8 of [24] and personal correspondence with the author). As we noted above, he was scheduled to give a talk at the First International Conference on Topology in Moscow in 1935 entitled "Considerations algèbriques sur le problème d'identité dans le groupe de tresses". Unfortunately, I have been unable to find any writing about the content of the planned talk. From the title, we may surmise that Ivanovsky was ready in 1935 to share this new solution to the braid word problem. In that case, Markov and Ivanovsky beat Artin to the punch by over a decade. Moreover, in the same 1945 paper, Markov notes that Weinberg had an algebraic method for determining the center of the braid group :

В конце статьи мы рассматриваем систему производящих элементов группы кос, полезную для других исследований. В частности, с помощью -этой системы Н. Вейнбергу удалось найти центр группы кос.

At the end of the article we consider a set of generators for the braid group, useful for other research. In particular, with the help of this system N. Weinberg managed to find the center of the braid group.

He thus seems to have beat Artin by at least five years.[13]

2.4 The Modern Algorithmic Approach

Once braids had appeared on a firm algebraic footing, worked turned to making algorithms to solve braid problems efficiently.

Notably, as far back as Artin's 1926 paper algorithms appeared: Artin describes a combing algorithm which separates out one strand at a time. We will not go into details, but the finished product looks something like this:

[12] Given two braids a and b, does there exist a braid c such that $a \equiv cbc^{-1}$?

[13] Very little is known of Weinberg, but it appears he died in the Siege of Leningrad, placing this discovery before circa 1942

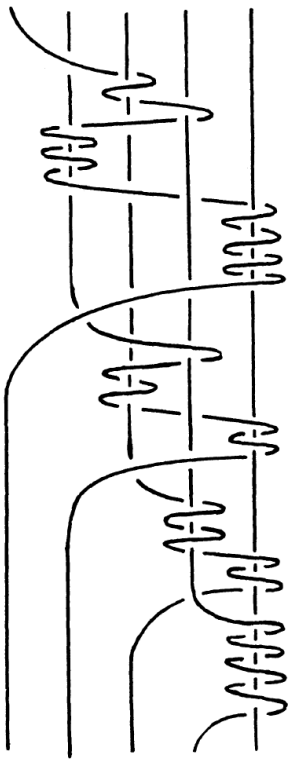


FIGURE 4

The algorithm is by no means efficient: Artin writes

> Although it has been proved that every braid can be deformed into a similar normal form the writer is convinced that any attempt to carry this out on a living person would only lead to violent protests and discrimination against mathematics. He would therefore discourage such an experiment (p. 126)

Thankfully, better solutions have appeared! In his 1965 doctoral thesis from Oxford,[14] Frank Garside gave a new solution to the braid word problem (and the conjugacy problem). This he published as "The Braid Group and Other Groups" in 1969 [16]. He considered a braid *monoid* and its embedding into a braid group. The mathematics for this, Ore localization, was first published by Oystein Ore in 1931 [27]. His article, "Linear Equations in Non-Commutative Fields" addressed the problem of finding a determinant in a matrix whose entries were elements of a non-commutative field. In order to solve this, he gave conditions that a non-commutative ring must satisfy in order to be localized - common multiples and commutativity. The question remains: how did Garside come across this result, and why did he think to apply it to the case of braids?

Firstly, although Ore's result could quickly be shown to hold for monoids (monoids are a simpler structure than rings; rings add a second binary operator and additional axioms to the monoid structure), this was not published explicitly until 1961 [20], in the classic textbook "The Algebraic Theory of Semigroups" by Clifford and Preston [10]. Garside cites this textbook, so it would appear he did see the monoid version of Ore's theorem. Until I get access to more of Garside's writings, I

[14]Unfortunately, the British Library was hit by a cyberattack in October 2023, and has yet to restore access to their microform copies of doctoral theses. It appears it may take a few months before I get access to this document

can make no claim about how he came upon this result. I find it quite plausible that some geometric intuition was used - especially for the proofs of common multiples.

As to the algorithms, Garside's solution to the word problem uses universal denominators. Simply put, he finds a "fraction" form for every braid group element, but the denominator (and hence the numerator) may be far more complicated than needed. Thus, when Dehornoy considers the problem, he is able to come up with a faster algorithm for the word problem by his subword-reversing method. Dehornoy's method runs in polynomial time on the length of the input braid word l. He does not bother to give a strict bound, but notes that in the worst case, it is in $O(l^2)$.

Now, research on braid groups has a bent towards faster solutions to the word problem (and the conjugacy problem, amongst others). We stop our historical survey with Dehornoy, for it is his work that was formalized, but see the last section for future ideas!

3 Defining Braids

We open with a brief description of the main mathematical structures at play. Presented groups have already been implemented in Lean, so we merely summarize the mathematics. A reader comfortable with the basics of presented groups is encouraged to skip to section 3.2. I implemented presented monoids, so a more detailed treatment is given, along with an overview of the Lean implementation.

3.1 Presented Groups

We have seen that braids form a group, so it is no surprise that we begin here.

Definition 3.1 (Group)**.** A group is a set with an associative binary operation, an identity element, and an inverse for every element.

We can be more specific here, in light of our notion of braid codes. These are strings of symbols (and their formal inverses)

Definition 3.2 (S-strings)**.** Given a set S, we define S^* to be the set of all $S-strings$ — that is, ordered lists of elements from S. Note that S^* contains the empty string, ε. An S-string s is a factor of an S-string t if s appears within t; that is, if $t = xsy$ for some (potentially empty) S-strings x and y

Now, for any element $s \in S$, we will denote its formal inverse as $\overline{s}$. The set containing the formal inverses of every element of S will be denoted $\overline{S}$. Strings over S and $\overline{S}$ form a group, called the *free group on* S.[15]

Definition 3.3 (Free Group)**.** The free group over a set S, denoted F_S, is the set $(S \cup \overline{S})^*$ with the binary concatenation operator, and the empty string as the identity.

Thus, we now have groups whose elements are strings. There is more to the braid group, however : it is often possible to code isotopically equivalent braids by different codes. For example, $\sigma_1\sigma_2\sigma_1$ is equivalent to $\sigma_2\sigma_1\sigma_1$. We want to make a structure which groups together codes representing the same braid. This is called a "quotient group" or a "presented group". We begin by defining the idea of "grouping together."

Definition 3.4 (Equivalence Relation)**.** A binary relation R on a set A is an equivalence relation if it is reflexive, symmetric, and transitive.

Definition 3.5 (Equivalence Class)**.** The equivalence class of an element $a \in A$ under an equivalence relation R is the set of all $b \in A$ such that $a\,R\,b$. We denote this $[\![a]\!]_R$. When it is clear from context, we may omit the subscript and write $[\![a]\!]$.

We aim to define a new group G/R whose elements are equivalence classes under R. What should the operation be? It would be nice to define $[\![a]\!]_R \cdot_{Q(G,R)} [\![b]\!]_R := [\![a \cdot_G b]\!]_R$. We must make sure this is well-defined: if $[\![a]\!]_R = [\![a']\!]_R$ and $[\![b]\!]_R = [\![b']\!]_R$, then somehow we must ensure $[\![a \cdot_G b]\!]_R = [\![a' \cdot_G b']\!]_R$

Not every equivalence relation will suffice to make G/R a group! We must give a stronger relation:

Definition 3.6 (Congruence)**.** A congruence is an equivalence relation on a group compatible with the group's operation $(\,\cdot\,)$: that is, if C is a congruence and $x\,C\,y$ and $z\,C\,w$, then $x \cdot z\;C\;y \cdot w$

Now, we have the idea of a quotient group.

[15] We call this group "free" because we are free to define a homomorphism h from the free group to any other group G, without restriction. We will see shortly that this is not the case for all groups!

Definition 3.7. Given a group G and a congruence C on G, the quotient group G/C is the set $\{[\![g]\!]_C \mid g \in G\}$, equipped with the binary operation $\cdot$, defined as $[\![a]\!]_C \cdot [\![b]\!]_C := [\![a \cdot_G b]\!]_C$ with $[\![1_G]\!]_C$ as the identity element, and $([\![a]\!]_C)^{-1} := [\![a^{-1}]\!]_C$.

We now move on to giving the shortest possible description of a quotient group. We specify the underlying group G more economically by writing it using generators :

Definition 3.8 (Generators)**.** We say a group G is generated by a set $S \subseteq G$ if every element of G may be written as product of elements of S and their inverses. We then say S is the set of generators of G.

Remark. A free group over a set S is generated by S

We may also simplify how we define the congruence C. Given a binary relation R on $(S \cup \overline{S})^*$, we may take the reflexive, symmetric, transitive, and multiplicative closure of R to form a congruence C_R :

- reflexivity : For all $a \in (S \cup \overline{S})^*$, $a \, C_R \, a$
- closure : If $a \, R \, b$ then $a \, C_R \, b$
- symmetry : If $a \, C_R \, b$ then $b \, C_R \, a$
- transitivity : If $a \, C_R \, b$ and $a \, C_R \, c$ then $a \, C_R \, c$
- multiplicativity : If $a \, C_R \, b$ and $c \, C_R \, d$ then $ac \, C_R \, bd$

Thus, we may define a presented group by giving the set of generators S and a relation R.

Definition 3.9 (Presented Group)**.** We define the presented group on a set S and relation R as $\langle S, R \rangle = F_S / C_R$.

With all this in place, we are ready to give Artin's definition of the braid group!

3.2 The Braid Group

For simplicity, we begin with the braid group on infinitely many strands, so we need not keep track of the specific number of strands. We will return at the end of the section to discuss the case of finitely many strands. Note : after defending this thesis, but before submitting to the library database, I re-implemented braid groups as a special case of Artin-Tits groups. This section is thus not current; I refer the interested reader to a talk I recently gave.

We begin with the set of generators $S = \{\sigma_i \mid i \in \mathbb{N}\}$. We define the index of σ_i to be i. The set of braid group words is $BW := (S \cup \overline{S})^*$.
The relation on braid groups with infinitely many strands, R_b, is

For all $i, j \in \mathbb{N}$, if $|i - j| = 1$, then $R_b(\sigma_i\sigma_j\sigma_i, \sigma_j\sigma_i\sigma_j)$ (the braid relation)
For all $i, j \in \mathbb{N}$, if $|i - j| \geq 2$, then $R_b(\sigma_i\sigma_j, \sigma_j\sigma_i)$ (the commutative relation)

The `PresentedGroup` structure is already in Mathlib, so the Lean definition of the braid group is almost entirely straightforward.[16] We use the natural numbers[17] to encode the generators, since all the σ_i are indexed by a distinct natural number.

[16] We give the relation in a slightly different format : if we have $xC_{R_b}y$, that means $[\![x]\!]_{C_{R_b}} = [\![y]\!]_{C_{R_b}}$. Since we are in a group, $[\![x]\!]_{C_{R_b}} \cdot ([\![y]\!]_{C_{R_b}})^{-1} = 1$. Thus, $[\![xy^{-1}]\!]_{C_{R_b}} = 1$. So, we can just keep track of the free group element xy^{-1}. We can thus give the relation C_{R_b} as a set of free group elements

[17] Following the Lean definition, here the natural numbers are understood to begin from zero. This leads to an off-by-one situation in terms of generator numbering as compared to Artin and Dehornoy

There is a small subtlety here: `PresentedGroup` generates a congruence on R_b, and then quotients by said congruence. Recall this is done by taking the reflexive, symmetric, transitive, and multiplicative closure of the original relations.

The braid relations are already symmetric, for they are defined on the absolute distance of the index of the relevant generators.[18]

So, for the formalization we simplify the relations to R_s, defined inductively as follows:

For all $i \in \mathbb{N}$, $R_s(\sigma_i\sigma_{i+1}\sigma_i,\ \sigma_{i+1}\sigma_i\sigma_{i+1})$
For all $i, j \in \mathbb{N}$, if $i + 2 \leq j$, $R_s(\sigma_i\sigma_j,\ \sigma_j\sigma_i)$

This restricted definition of the relations will reduce the numbers of cases needed in proofs about braid groups. So, we define our relations `braid_rels_inf` as R_s. In the rest of this write-up, we will write $a \equiv b$ to mean $a\, C_{R_S}\, b$.

Thus, the Lean definition looks like so:

```
def braid_rels_inf : Set (FreeGroup ℕ) :=
  { r | ∃ i : ℕ , r = .of i * (.of (i + 1)) * .of i * (.of (i + 1))⁻¹ * (.of i)⁻¹
    * (.of (i + 1))⁻¹} ∪
    { r | ∃ i j : ℕ, i + 2 ≤ j ∧ r = .of i * .of j * (.of i)⁻¹ * (.of j)⁻¹}

def braid_group_inf := PresentedGroup braid_rels_inf
```

With the addition of a small API,[19] the braid group is thus defined in Lean!

3.3 Presented Monoids

Given that the braid relations themselves contain no inverses, a natural question arises : can we consider a structure with the same generators but NOT their inverses, under the braid relations? The answer is yes.

Definition 3.10 (Monoid)**.** A monoid is a structure with an associative binary relation and an identity element.

Similar to how we defined free groups and presented groups, we may also define free monoids and presented monoids. The task is easier : we need not worry about inverses.

Definition 3.11 (Free Monoid)**.** The free monoid over a set S is the set S^* equipped with the binary concatenation operator and the empty string as the identity.

Congruences hold just the same for monoids as they do for groups; [20] one merely replaces the word "group" with "monoid" in the definition. Thus, we may define quotient monoids.

[18] For example, if we know for some i and j that $R_b(\sigma_i\sigma_j\sigma_i,\ \sigma_j\sigma_i\sigma_j)$, then we must have $R_b(\sigma_j\sigma_i\sigma_j,\ \sigma_i\sigma_j\sigma_i)$. Why? Well, since $R_b(\sigma_i\sigma_j\sigma_i,\ \sigma_j\sigma_i\sigma_j)$, $|i - j| = 1$ (the braid relation is the only one which holds on elements of length 3). Thus, $|j - i| = 1$ as well, so $R_b(\sigma_j\sigma_i\sigma_j,\ \sigma_i\sigma_j\sigma_i)$ as desired. This holds in a similar manner for the commutative part of the relation.

[19] Note that the congruence generated on R_s is on free group elements. We give basic facts about equality within the braid group. This includes facts like $[\![\sigma_{i+1}\sigma_i\sigma_{i+1}]\!]_{C_{R_S}} = [\![\sigma_i\sigma_{i+1}\sigma_i]\!]_{C_{R_S}}$.

We also define the universal property for braid groups (merely a specialization of that for presented groups).

[20] The form given in the previous section is not exactly the form needed for future proofs, so we prove an equivalent definition:

- For all a, $a\, C\, a$ (reflexivity)
- If $a\, C\, b$ and $b\, C\, c$ then $a\, C\, c$ (transitivity)
- If $a\, C\, b$ then for all c and d, $cad\, C\, cbd$ (one-step reduction)
- If $a\, C\, b$ then for all c and d, $cbd\, C\, cad$ (symmetric one-step reduction)

There are many equivalent formulations; we will eventually give about four. All will be needed for various induction proofs.

Definition 3.12. Given a monoid M and a congruence C on M, the quotient monoid $M/^+C$ is the set $\{[\![g]\!]_C \mid g \in M\}$, equipped with the binary operation $\cdot$, defined as $[\![a]\!]_C \cdot [\![b]\!]_C := [\![a \cdot_M b]\!]_C$ and $[\![1_M]\!]_C$ as the identity element.

Again, we may consider a monoid generated by a set S :

Definition 3.13 (Generators)**.** We say a monoid M is generated by a set $S \subseteq M$ if every element of M may be written as product of elements of S. We then say S is the set of generators of M.

Remark. A free monoid over a set S is generated by S

And so all the ingredients are here : we may define a presented monoid.

Definition 3.14 (Presented Monoid)**.** We let $\langle S, R\rangle^+$ denote the presented monoid on a set S and relation R - it is the quotient monoid of the monoid generated by S under C_R

With the mathematical theory described, on we go to the Lean formalization!

Presented monoids were not in Mathlib; currently, about half of my work has made it in. Luckily, there is already a definition of congruences in Lean, so we may build upon that structure.

First, we must give a formal definition of S-strings. It could be a list, a free monoid, a string, etc. However, `Congruence` must be applied to a type endowed with a `Mul` structure. Although a multiplication operation could be easily defined for lists and strings (appending for lists; concatenation for strings), `FreeMonoid` is the only type in Mathlib that has a `Mul` structure. So, we represent the α-strings by `FreeMonoid` instead of `List`.

With that decision made, we must simply quotient by said `Congruence` relation, and our presented monoid will be defined!

Let us briefly note that presented monoids are also often called abstract re-writing systems, string re-writing systems, or Thue systems. This is because they model the idea of re-writing parts of the string according to given rules.

Definition 3.15 (R-derivation)**.** Given a set S and a relation R on S^*, a R-derivation is a finite list of strings $R_0, R_1, ..., R_n$ such that for every k, $0 \leq k < n$, $R_k = axb$, $R_{k+1} = ayb$, and $x\,R\,y$.

R-derivations are simply another way to view a presented monoid.

Theorem 3.1. There exists an R-derivation linking R_0 to R_n if and only if they are equivalent in the corresponding presented monoid : $[\![R_0]\!]_{C_R} = [\![R_n]\!]_{C_R}$

So, a presented monoid is a re-writing system just under another name. We thus give a robust API for a user looking to prove two words are equivalent in a presented monoid. For example, we prove the following amongst many others:

- If $a\,C_R\,b$ then for all c, $ca\,C_R\,cb$
- If $a\,R\,b$ and $c\,C_R\,d$ then $ac\,C_R\,bd$
- If $a\,C_R\,b$ and $c\,R\,d$ then $ac\,C_R\,bd$
- If $a\,R\,b$ then for all c, $ac\,C_R\,bc$

The benefit here is that we keep the definition simple, so that proofs about presented monoids have minimal cases. But the user also has access to a veritable library of lemmas to speed up proofs about the words themselves.

3.4 The "Braid" Monoid

Let us consider a monoid in which the braid relations hold. Note that at this point, we have alluded to a proof that the braid *group* aligns with the topological definition of braids. Although it is tempting to think of a monoid in which the braid relations hold as a "braid monoid" (perhaps braids with only overcrossings), until we can connect such a monoid to the braid group mathematically (perhaps via a map), we have no geometric interpretation of the monoid. It is, for the moment, just an arbitrarily defined mathematical structure with no connection to the braid group.

We begin with the set of generators $S = \{\sigma_i \mid i \in \mathbb{N}\}$. We define the set of braid monoid words to be $BW := S^*$.With the `PresentedMonoid` structure set up, we may apply it to our relations. Again, R_s is defined inductively as follows:

For all $i \in \mathbb{N}$, $R_s(\sigma_i\sigma_{i+1}\sigma_i,\ \sigma_{i+1}\sigma_i\sigma_{i+1})$
For all $i, j \in \mathbb{N}$, if $i + 2 \leq j$, $R_s(\sigma_i\sigma_j,\ \sigma_j\sigma_i)$

This procedure is straightforward, and similar to how we used the `PresentedGroup` structure.

We are able to define a number of functions on the braid monoid. Since the relations leave the length unchanged, length is well-defined on the braid monoid. Note that this is not true in general for any presented monoid : we could very well have a relation that says, say, $aa\,R\,a$ for some a. It is not true in the braid group: $[\![\sigma_i\overline{\sigma_i}]\!] = [\![\varepsilon]\!]$.

We may also define the set of generators in a braid monoid element — again, the braid relations leave this unchanged. In fact we may even reverse a braid monoid element. In order to calculate the length, the set of generators, or the reversed braid element, we perform the relevant operation on a representative of the equivalence class — a free monoid element. So, the length of $[\![a]\!]$ is defined as the length of a.

We will thus need to somehow define length, set of symbols in, and reversing for free monoid elements. Let us return for a moment to the choice made earlier about the type for S-strings. We were forced to select free monoids over lists. This is a bit of a hassle now : `List` is much richer than `FreeMonoid`. `List` already has functions `length`, `toFinset`, `reverse`, etc.

Although it is simple to implement these for `FreeMonoid`, we must tread carefully. While free monoids are in fact defined as lists in Lean, they are meant to perform different roles. Free monoids generally bend more to mathematical theorems in abstract algebra, whereas lists are more used for computation. Even when the two share similarly-minded functionality, it is from a different viewpoint and with different implementation details. For example, if the idea is to apply a function to every element of our string and then combine them, we have `FreeMonoid.lift` but `List.foldr`. It is a balancing act to keep the two structures distinct, but also usable. Each has lemmas the other lacks.

We have decided to add definitions and lemmas to `FreeMonoid` if they will be able to be defined on the eventual *presented* monoid, the braid monoid. Such definitions seem reasonably "algebraic" since they can be defined for even more complex algebraic structures. Thus, we add in

`length` : The length of a FreeMonoid element
`reverse` : reverses a FreeMonoid element - abcd becomes dcba
`symbols` : The set of symbols appearing in a FreeMonoid element

because these play nicely with the braid monoid structure.

We take a different approach for `List` definitions that do not remain invariant in the braid structure, such as `getLast` (this returns the last element in a list). Equivalent braids need not have the same last generator - $\sigma_1\sigma_3$ and $\sigma_3\sigma_1$ certainly do not! In this case, we have decided to use `FreeMonoid.toList` when needed.

This is a bit cumbersome, but thankfully quite rare for this project.

3.5 Finite Braid Groups and Finite Braid Monoids

Above, we have discussed only braid groups and braid monoids on infinitely many strands and the equivalence on them, $\equiv$. We may also consider braids on n strands. A braid on n strands has $n-1$ generators, but that makes no sense for $n=0$. Instead we phrase it as: a braid on $n+1$ strands has n generators. Those generators are numbered $\sigma_0, \sigma_1, ...\sigma_{n-1}$ (this is a difference from the overview in the Introduction, but is necessary because Lean starts natural numbers at 0). Note that if $n=1$, then there are no generators.

We let the set of generators for a finite braid group or monoid on $n+1$ strands to be $S_n = \{\sigma_i | i \in [n]\}$. (Here, $[n] = \{m \in \mathbb{N}, m < n\}$. Then the set of finite-braid group words is $BW_n := (S \cup \overline{S})^*$ and the set of finite-braid monoid words is $BW_n^+ := S^*$.
The relation on braid groups with $n+1$ strands, R_{b_n+1}, is defined inductively as

For all $i, j \in [n]$, if $|i-j| = 1$, then $R_b(\sigma_i\sigma_j\sigma_i,\ \sigma_j\sigma_i\sigma_j)$ (the braid relation)
For all $i, j \in [n]$, if $|i-j| \geq 2$, then $R_b(\sigma_i\sigma_j,\ \sigma_j\sigma_i)$ (the commutative relation)

Once again, we will simplify the relations to avoid doubling-up on symmetry. The equivalence on braids with $n+1$ strands will be given by the congruence on the simplified $R_{b_{n+1}}$, and will be denoted $\equiv_n$. Note that $\equiv_n$ is the restriction of $\equiv$, simply tossing out relations involving generators with index n or higher. We denote the braid group on $n+1$ strings as B_{n+1}; the monoid as B_{n+1}^+.

Infinite braid groups/monoids are simple in Lean - the generators are kept track of via their index, of type $\mathbb{N}$. For finite groups, the induces of the generators are a finite set of consecutive natural numbers. We thus use the `Fin` type. It is a bundled type: an object of type `Fin n` contains a natural number k and a proof that $k < n$. Notably, `5 : Fin 7` and `5 : Fin 8` are different elements. Addition on `Fin` is delicate: we have the potential for overflow errors. There are two options: keep track of the hypotheses, so that when we add, say, `3 : Fin 10` and `5 : Fin 10`, we must provide a proof that $3+5 < 10$ (addition is here on 3 and 5 as natural numbers). Alternatively, there is a successor function which maps `a : Fin n` to `(a+1) : Fin (n+1)`. Here, we needn't juggle hypotheses, but this comes at a cost of changing the upper bound. We can also change the upper bound without changing the number itself : this is `Fin.castSucc`. So, `Fin.castSucc (5 : Fin 7) = (5 : Fin 8)`.

We use the latter approach, with `Fin.succ` and `Fin.castSucc`. Although a bit finicky to set up, we needn't bother with hypotheses floating around. Moreover, we will be able to define our relations with just two applications of either `Fin.succ` or `Fin.castSucc`. The reader may wish to skip the following few paragraphs detailing the implementation of the braid relations for the finite case. It gives a sense of the precision required when defining objects in Lean, and is a fun puzzle, but is not necessary to follow the rest of the discussion.

3.5.1 Implementation

We will describe in detail the implementation[21] of the finite braid groups. The case for the monoid is analogous. To make the definition cleaner, we first break it into smaller pieces. We define the braid relation for inputs i and j as

```
def braid_rel {S : Type _} (i j : S) : FreeGroup S :=
  i * j * i * (↑j)⁻¹ * (↑i)⁻¹ * (↑j)⁻¹
```

- of course making sure we only input adjacent i and j later on!

Similarly, for the commutative relation is

```
def comm_rel {S : Type _} (i j : S) : FreeGroup S :=
  i * j * (↑i)⁻¹ * (↑j)⁻¹
```

[21] Again, I have re-implemented this in a more elegant fashion; the definition given here is no longer current

To recap: we want the generators of the braid group on $n+1$ strands to have type `Fin n`. We define the relations as such:

```
def braid_rels : (n : ℕ) → Set (FreeGroup (Fin n))
  | 0     => ∅
  | 1     => ∅
  | n + 2 =>
    { r | ∃ i : Fin (n + 1), r = braid_rel (i.castSucc) (i.succ) } ∪
    { r | ∃ i j : Fin n, i ≤ j ∧ r = comm_rel (i.castSucc.castSucc)
    (j.succ.succ) }
```

What's going on with this $n+2$ case? First off, any natural number not equal to 0 or 1 is of the form $n+2$ for some natural number n. Let's take the braid relation case: we want the braid relation to hold for any i and $i+1$. Well, we need both i and $i+1$ to have type `Fin (n+2)`. So we need `i.succ` to have type `Fin(n+2)`, meaning `i` must have type `Fin(n+1)` to begin with. In the end, we need both `i` and `i.succ` to have type `Fin(n+2)`, so we will cast `i` to `Fin(n+2)` using `Fin.castSucc`.

The commutative case adds a twist. Let's go through the reasoning step-by-step. We need an end result where both i and j are of type `Fin(n+2)`, and i is at least 2 less than j (meaning, if we start with $i \leq j$, we need to bump up j twice to guarantee the needed distance). So, we make a hypothesis that $i \leq j$, and use `Fin.succ` twice on `j`. So `j` should start off as `Fin n`. For future use, it is simpler to keep `i` and `j` as the same initial type, so we let `i` also have type `Fin n` and use `Fin.castSucc` twice. To be clear, we do not only consider pairs i, j that are exactly two apart. We begin with some i and j such that $i \leq j$, and then widen the existing gap by 2.

Whew!

3.5.2 Finite Relation as a Restriction

Now, there are clear similarities between the finite and infinite braid groups (and respective monoids). When we prove properties about one, it would be nice to have some kind of "converter" to port them to the other group/monoid.

It seems just from the definitions that the infinite case will be easier to do the bulk of the work in. But will that hold when proving properties *about* the braid groups/monoids?

Let's consider another fact about `Fin n` : subtraction is truncated, so 5-7=0. One must be precise: $a - b = c - b$ only implies $a = c$ when $b \leq a$ (or equivalently, $b \leq c$). This makes many arithmetic proofs long. In the case of the braid relations, where one often must prove something of the form $i + 2 \leq j$, this complicates things enormously. Thus, we work within the finite braid monoid as minimally as we can get away with.

So, how to port over the results? Copy-pasting is long, and we would still need to check all the hypotheses needed for `Fin n`.

Luckily, the relation on finite braids is a restriction on that for infinite braids : if $a \equiv_n b$, then $a \equiv_k b$ for all $k > n$, and also $a \equiv b$. The is a small hiccup here : when we say $a \equiv_n b$ this means $a, b \in B_n$, but when we say $a \equiv_k b$ this means $a, b \in B_k$. Luckily, it is quick to see that we have $B_0 \subset B_1 \subset B_2 ... \subset B_\infty$, and so this poses little trouble to the paper-and-pencil mathematician. In Lean, we must typecast from `Fin n` to `Fin k` or $\mathbb{N}$, but this is a quick task. In the other direction, if we have a theorem for infinite braid monoids, we must cast the infinite braid monoid element "down" to a finite braid monoid element. Thus can be done so long as the infinite braid monoid element is appropriately bounded. Thus, we have

```
def make_fin (n : ℕ) (a : FreeMonoid ℕ) (bound : ∀ x ∈ a, x<n) : FreeMonoid (Fin
    n) :=
  (FreeMonoid.pmap (λ i => Fin.mk i ) a) bound
```

The use of pmap instead of map protects us from dependent type theory issues. And when two bounded infinite braid monoids are equivalent under $\equiv$, they are also equivalent under the finite braid relation for any appropriate number of strands.

```
theorem braid_rel_inf_to_fin (n: ℕ) (a b: FreeMonoid ℕ) (holds_in_inf :
braid_rels_m_inf a b)
(bounded_a: ∀ (x : ℕ), x ∈ a → x < n.pred) (bounded_b: ∀ (x : ℕ), x ∈ b →
x < n.pred) :
braid_rels_m n.pred (make_fin n a bounded_a) (make_fin n b bounded_b)
```

We will thus prove all main results for the infinite case, and then port it down to the finite case. This procedure will be discussed along with each result.

4 Localizing Braids

At last it is time to drop the scare quotes around "braid" monoids! We will elucidate the connection between them and braid groups, and show an embedding (an injective homomorphism from the braid monoid into the braid group).

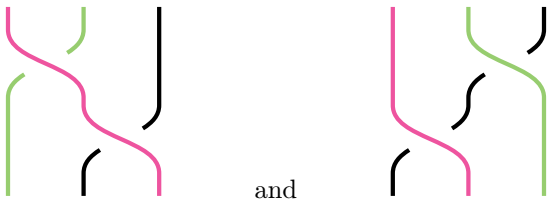
4.1 Ore Localization

We must somehow connect the braid monoid to the braid group. The intuitive idea is to somehow add in inverses to the braid monoid and obtain a structure isomorphic to the braid group.

The idea of adding in inverses, or "denominators", is common in mathematics. We obtain the rational numbers by adding in non-zero natural-number "denominators" to the integers. This process is called localization. The common procedure of localizing the integers by the non-zero natural numbers creates a group $(\mathbb{N}\backslash\{0\})^{-1}\mathbb{Z})$ [22] whose elements are fractions of the form $\frac{p}{q}$ where $p \in \mathbb{Z}$ and $q \in \mathbb{Z}\{0\}$. The group operation is fraction multiplication,

$$\frac{a}{b} * \frac{c}{d} = \frac{a \cdot_{\mathbb{Z}} c}{b \cdot_{\mathbb{Z}} d}$$

But note that ordinary fraction multiplication uses commutativity. Alas, concatenation of braids is not commutative in the monoid (nor the group): One may see that $\sigma_1\sigma_2$ does not equal $\sigma_2\sigma_1$ through properties of braid relations (no relations match $\sigma_1\sigma_2$). Or visually, these two braids are distinct:



and

So, we use a clever trick! First let's define two algebraic properties: common multiples and cancellativity. Together, they are called the "Ore Conditions."

Definition 4.1 (Common Multiples)**.** A monoid M with right common multiples satisfies the property that for all a and b in M, there exist c and d in M such that $ac = bd$. Left common multiples are defined similarly.

Definition 4.2 (Cancellativity)**.** A cancellative monoid has the property that for all monoid elements a, b, c, if $a \cdot c = b \cdot c$ then $a = b$, and if $c \cdot a = c \cdot b$, then $a = b$.

The big idea is that we may localize a non-commutative monoid if it has those two properties — right common multiples and cancellativity. In fact it doesn't matter if the common multiples are left or right; the braid monoid happens to have both. Without loss of generality, we will below use right common multiples. Off we go!

We begin with a monoid M and a cancellative submonoid with right common multiples $S \subseteq M$. We consider all pairs of the form (m, s) with $m \in M$ and $s \in S$. We give the following equivalence relation on pairs:

Definition 4.3 ($\sim$)**.** We say $(m_1, s_1) \sim (m_2, s_2)$ iff $m_1 a = m_2 b$, where a and b are such that $s_1 a = s_2 b$.

[22] Usually, we give this construction a ring structure, since we also have an addition operation on $\mathbb{Z}$. Since the braid group only has one operation, we only discuss multiplication here

Theorem 4.1. $\sim$ is well-defined; that is, if we have $a, a', b, b' \in S$ such that $s_1 a = s_2 b$ and $s_1 a' = s_2 b'$, $m_1 a = m_2 b$ if an only if $m_1 a' = m_2 b'$. This means that the definition of $\sim$ is independent of the choice of common multiples.

Theorem 4.2. $\sim$ is an equivalence relations on pairs (m, s).

We omit the proofs of the above two theorems; they follow from the definition of $\sim$.

Now, we consider equivalence classes of pairs under $\sim$. These equivalence classes will be the elements of the group we aim to construct.

Definition 4.4 ($S^{-1}R$)**.** Given a monoid R and a submonoid of R satisfying the Ore conditions S, we define the monoid $S^{-1}R$ as the set $\{[\![(r, s)]\!]_\sim \mid r \in R, s \in S\}$ with $\frac{1_S}{1_S}$ as the identity and a binary operation defined as follows: $\frac{a}{b} \cdot \frac{c}{d} = \frac{ae}{df}$ where e and f are any elements of S such that $be = cf$.[23]

Once again, we must verify that this definition is independent of the choice of common multiples e and f, and once again, we omit the proof here. Similarly, we must prove associativity.

Henceforth, we will write $\frac{r}{s}$ for $[\![(r, s)]\!]_\sim$. We may obtain an injective homomorphism h from R into $S^{-1}R$ by sending $r \in R$ to $\frac{r}{1}$.

- Injectivity : Assume $h(r_1) = h(r_2)$; that is,$\frac{r_1}{1} = \frac{r_2}{1}$. Then $(r_1, 1) \sim (r_2, 1)$. Since $1 \cdot 1 = 1 \cdot 1$, we must have $r_1 \cdot 1 = r_2 \cdot 1$ (from the definition of $\sim$). Hence $r_1 = r_2$ as desired.
- Homomorphism : $h(r_1 \cdot r_2) = \frac{r_1 r_2}{1}$. We must now calculate $h(r_1)h(r_2) = \frac{r_1}{1} \cdot \frac{r_2}{1}$. In this case, $a = r_1, b = 1, c = r_2, d = 1$. Then since $1 \cdot r_2 = r_2 \cdot 1$, we may let $e = r_2$ and $f = 1$. Hence $\frac{r_1}{1} \cdot \frac{r_2}{1} = \frac{r_1 \cdot r_2}{1 \cdot 1}$. Thus, $h(r_1 r_2) = h(r_1)h(r_2)$.

Thus, we are able to obtain an embedding of R intro $S^{-1}R$.

We are lucky that the Ore Localization was already formalized in Lean by Jakob von Raumer for rings; the vast majority of the preliminaries can be used for monoids. A little work remains to specialize it to our particular use case of presented monoids.

4.2 Ore Localization of a Presented Monoid

In our case, we are localizing not by just any submonoid, but in fact by the entire monoid.

Theorem 4.3. For any monoid M satisfying the Ore conditions, $M^{-1}M$ has a group structure.

Proof. We already have shown that $M^{-1}M$ has a monoid structure. Thus, all that remains is to define inverses. We will say that for any $\frac{a}{b} \in M^{-1}M$, $(\frac{a}{b})^{-1} = \frac{b}{a}$. This is possible to do since both a and b must be in M, so a is allowed to serve as a denominator. We shall omit the proof that $\frac{b}{a}$ satisfies the properties of an inverse as this is straightforward. □

Now, let us consider an even more specific case: when the monoid M is a presented monoid. Let $M = [\![S, R]\!]^+$. I claim that $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$ is isomorphic to the presented group on those same generators and relations, $\langle S, R\rangle$. We show this by a careful inspection of the universal properties of $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$ and $\langle S, R\rangle$.

[23] to motivate this definition, one might consider the following chain of reasoning :
$\frac{a}{b} \cdot \frac{c}{d}$ will map to $a \cdot b^{-1} \cdot c \cdot d^{-1}$ in the eventual group we construct. Then we let e and f be such that $be = cf$. Thus, $e = b^{-1}cf$, so $ef^{-1} = b^{-1}c$ Then

$$ab^{-1}cd^{-1} = aef^{-1}d^{-1} \quad (1)$$
$$= (ae)(df)^{-1} \quad (2)$$

which is an element of the fraction group!

Definition 4.5 (Lift)**.** Given a map f from a set S to a monoid M, we may lift f to a function from the free group on S to M - this we will denote f_L.

Definition 4.6 (Universal Property of a presented group)**.** Given a presented group $\langle S, R\rangle$, for any other group G, any map f from S to G, if for all $r \in R$, $f_L(r) = 1_G$, then there exists a unique homomorphism f^* from $\langle S, R\rangle$ to G.

Remark. The identity map on $\langle S, R\rangle$ is a group homomorphism, so there is only one homomorphism from $\langle S, R\rangle$ to $\langle S, R\rangle$.

Definition 4.7 (Universal Property of the Ore Localization of a Presented Monoid)**.** Given a localization of a presented monoid group $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$, for any other group G, any map f from S to G, if for all $(r_1, r_2) \in R$, $f_L(r_1) = f_L(r_2)$, then there exists a unique homomorphism f^* from $\langle S, R\rangle$ to G.

Remark. The identity map on $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$ is a group homomorphism, so there is only one homomorphism from $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$ to $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$.

We use the simplified notation $PML(S, R)$ for $(\langle S, R\rangle^+)^{-1}\langle S, R\rangle^+$ — PML being "presented monoid localization". Our first task will be to define two homomorphisms:

h_1 : homomorphism from $\langle S, R\rangle$ to $PML(S, R)$. Here, we use the universal property for presented groups to lift the function $s \mapsto \frac{s}{1}$.

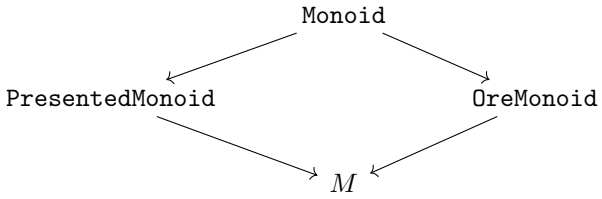
h_2 : homomorphism from $PML(S, R)$ to $\langle S, R\rangle$. Here we use the universal property for Ore localizations of presented monoids to lift the function $s \mapsto [\![s]\!]_{C_R}$.

Note that if we compose h_1 with h_2, we get a homomorphism from $\langle S, R\rangle$ to $\langle S, R\rangle$ – but that must be the identity! Similarly, composing h_2 with h_1 gives the identity. Thus, h_1 is a bijection (and h_2 is as well). Thus, we have an isomorphism between $PML(S, R)$ and $\langle S, R\rangle$. Hence, the localization of a presented monoid is isomorphic to the presented group on those sam generators/relations.

Thus, every element of the braid group can be written in the form $\iota(a)(\iota(b))^{-1}$ for a, b in the braid monoid.

And, since we can embed the braid monoid (no scare quotes now!) into the braid group, we know that means any braid is equivalent to one in which all the overcrossings appear first, followed by the undercrossings.

The Lean implementation gets off to a rocky start : we want to plug in a presented monoid satisfying the Ore conditions to the existing Ore localization API. However, both the `PresentedMonoid` and `OreMonoid` instances carry with them a `.toMonoid` function. If we try to give Lean a type `M` with instances `[PresentedMonoid M] [OreMonoid M]`, Lean can infer the `Monoid` instance on `M` in two different ways. This is referred to as a "bad diamond".



To avoid this, we define mixins : `Prop`-valued typeclasses (`Prop`-valued means the typeclass carries no data). Lean already has a mixin `IsCancelMul`, which carries the properties of cancellation : $a * b = a * c \implies b = c$ and $b * a = c * a \implies b = c$. So, we need only define a mixin for common multiples, and then bind it together with `IsCancelMul` to get an Ore mixin.

We thus define `isCommonMul` on a structure M with multiplication carrying the property that for all $a, b \in M$, there exist $c, d \in M$ such that $ac = bd$.

Unfortunately, this definition is non-constructive. The Ore localization in Mathlib is constructive, as is our upcoming proof that the braid monoid has common multiples. In the future, I hope to update the code so that it can be constructive through-and-through. This is not needed for the proof of correctness and termination of our braid word problem algorithm. However, it would be nice to visualize the magic of the isomorphism between the braid group and the localization of the braid monoid. It's pretty cool that any braid is equivalent to one in this special overcrossings-before-undercrossings form. I would like Lean to be able to output that form for any braid (more to come on this in the "Future Work" section at the end of this document).

Nevertheless, the definition works for the current project. Moving onwards, we are lucky that the universal property for the presented group is already formalized. The universal property of the Ore localization of a presented monoid by itself is merely a specialization of that for general Ore localizations.

Now, we have an injective homomorphism from the localization of a presented monoid into a presented group (over the same relations, converted appropriately). We shall show below that the braid monoid satisfies the conditions for Ore localization, and thus we have an injective homomorphism from said localization into the braid group.

4.3 Common Multiples

We now prove the existence of right common multiples — that is, for any braid monoid elements a and b, we show that there exist braid monoid elements c and d such that $a \cdot c \equiv^{+} b \cdot d$. Currently, this is not implemented constructively, because it feeds into the `hasCommonMul` mixin defined above, which is not constructive.

We could equally well give a function which returns the common multiples of any two braid monoid elements. We'd then prove that this function's outputs work as expected. This may be done in the future — it would be nice to display an image of the common multiples in the infoview of Lean for the user.

Once again, we note that since we are still working on an embedding of B_n^+ into B_n, any mention of strands or imagery is entirely for the reader's intuition. It is not allowed to play any role in the proof.

4.3.1 Infinitely Many Strands

We begin by defining more braid structures.

The first is the $\underline{\sigma}$ braid, representing one strand moving across a number of other consecutive strands. We define[24]

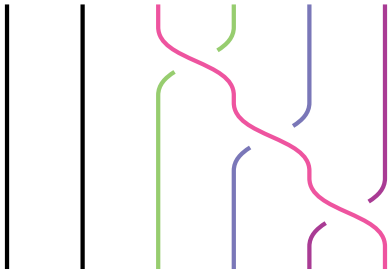
$\underline{\sigma}_{i,i}$ to be the empty word

If $i < j$ then $\underline{\sigma}_{i,j} = \sigma_i \cdot \underline{\sigma}_{i+1,j}$

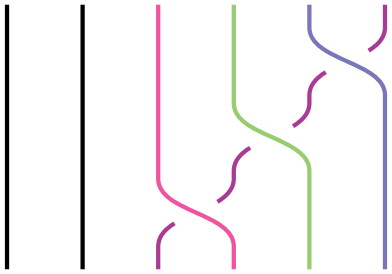
Else, $j < i$ and $\underline{\sigma}_{i,j} = \sigma_{i-1} \cdot \underline{\sigma}_{(i-1),j}$

[24] The Lean definition of this and the following braid structures can be found in the appendix.

Thus, we have $\underline{\sigma}_{3,6} = \sigma_3 \cdot \sigma_4 \cdot \sigma_5$, which we will later see looks like the third string moving over into the place of the sixth:

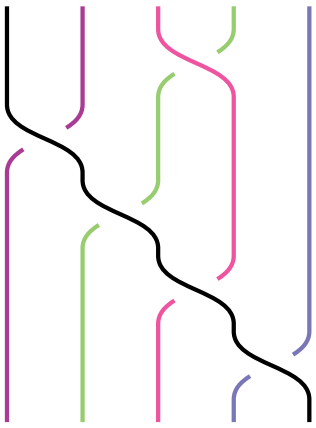


Further, we have $\underline{\sigma}_{6,3} = \sigma_5 \cdot \sigma_4 \cdot \sigma_3$ which looks like the sixth string moving over into the place of the third (note that since we are in the monoid, it must pass under the other strings, as only positive generators are allowed):

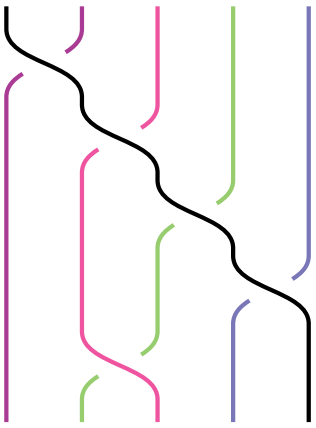


Now we begin to accumulate results about how to move a single generator across a $\underline{\sigma}$ section: For $j \geq i+2$ and $i < k < j$, we have $\sigma_k \underline{\sigma}_{i,j} \equiv^+ \underline{\sigma}_{i,j} \sigma_{k+1}$.

Let's look at an example: $\sigma_3 \underline{\sigma}_{1,5} \equiv^+ \underline{\sigma}_{1,5} \sigma_4$.



should be equivalent to



This seems eminently reasonable! The original proof was by induction on the distance between i and j - but we note that since $i < j$ here, we may instead induct on $j - i$. We write a custom induction principle [25] for inducting on the difference of two numbers, knowing that they are at least 2 apart.

[25] This work has prompted the creation of a number of additional induction principles in Lean. For example, we have added a principle which inducts on the absolute difference between two numbers. We have given another which performs regular (weak) mathematical induction on a variable which we happen to know is bound between two others. These provide a clean API and reduce the hypothesis-wrangling left to the user.

That is, if we have a property p which takes in 2 integers, and we want to show that p holds of some integers i, j where $j \geq i + 2$, our induction will look like

Base Case : For any two integers i', j', if $j' - i' = 2$, then p holds of i' and j'.

Inductive Step : We consider natural numbers j' and i' such that $j' - i' \geq 3$.

Inductive hypothesis : We assume that for all i'' and j'' with $2 \leq j'' - i'' < j' - i'$, p holds of i'' and j''. Wait aim to show that p holds of i' and j'.

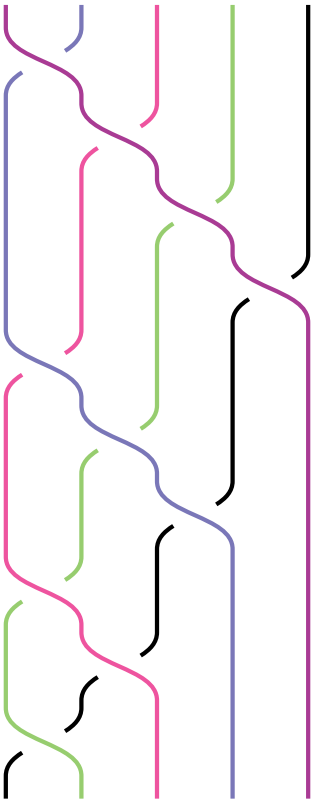
We note another completely analogous result, proved in almost the same manner, that for $j \geq i+2$ and $i < k < j$, we have $\sigma_{k-1}\underline{\sigma}_{j,i} \equiv^{+} \underline{\sigma}_{j,i}\sigma_k$.

Now we move on to even more complicated braids - a half twist! A half twist on n strands is denoted $\underline{\Delta}_n$, and is built up of sigma braids. It is defined inductively as so,

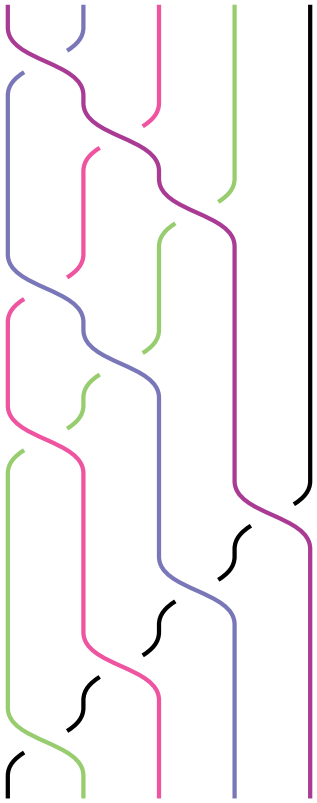
$\underline{\Delta}_0$ is the empty word

If $n \geq 1$ then $\underline{\Delta}_n = \underline{\sigma}_{0,n}\underline{\Delta}_{n-1}$

Thus, $\underline{\Delta}_n = \sigma_1\sigma_2\sigma_3\sigma_4\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1$



Geometrically, we could also perform a half-twist on the first $n-1$ strands, and then pull the last, nth, string all the way back. It might look something like this:



We hence surmise that

Theorem 4.4. For every $n > 0$, $\underline{\Delta}_n \equiv^+ \underline{\Delta}_{n-1}\underline{\sigma}_{n,0}$.

Proof. We proceed by induction.

Base Case : $n = 1$. Then

$$\begin{aligned}\underline{\Delta}_1 &= \sigma_0 \\ &= \underline{\sigma}_{1,0} \\ &= \varepsilon \cdot \underline{\sigma}_{1,0} \\ &= \underline{\Delta}_0\underline{\sigma}_{1,0}\end{aligned}$$

Inductive step: We consider the $n+1$ case

$$\begin{aligned}
\underline{\Delta}_{n+1} &= \underline{\sigma}_{0,n+1}\underline{\Delta}_n && \text{(by def of } \underline{\Delta}) \\
&= \underline{\sigma}_{0,n}\sigma_n\underline{\Delta}_n \\
&\equiv^+ \underline{\sigma}_{0,n}\sigma_n\underline{\Delta}_{n-1}\underline{\sigma}_{n,1} && \text{(by inductive hypothesis - now every symbol occurring in } \underline{\Delta}_{n-1} \\
&&& \text{is far enough away from } \sigma_n \text{ that commutativity applies!)} \\
&\equiv^+ \underline{\sigma}_{0,n}\underline{\Delta}_{n-1}\sigma_n\underline{\sigma}_{n,1} && \text{(by repeated applications of the commutativity relation)} \\
&\equiv^+ \underline{\Delta}_n\sigma_n\underline{\sigma}_{n,1} && \text{(by def of } \underline{\Delta}) \\
&\equiv^+ \underline{\Delta}_n\underline{\sigma}_{n+1,1} && \text{(by def of } \underline{\sigma})
\end{aligned}$$

as desired

□

We have shown that we may replace $\underline{\Delta}_n$ by an expression involving $\underline{\Delta}_{n-1}$; this will prove quite handy for proofs by induction.

Remember that we were able to move a single generator "through" a $\underline{\sigma}$ braid. We next wonder what happens when we try to pull a single crossing through a delta braid. Can we move the crossing here below the twist? In fact, we can!

Theorem 4.5. For any natural numbers n and i with $i < n$, $\sigma_i\underline{\Delta}_n \equiv^+ \underline{\Delta}_n\sigma_{n-i}$

Proof. We would proceed by induction on n, and then casework on i, and use Theorem 3.4 as proved above. □

With the $\underline{\Delta}$ structure now defined, and a few lemmas proved about it, we are ready to attack the common-multiple problem. Remember, we will be looking for c and d such that $ac \equiv^+ bd$. Instead of focusing on c and d themselves, we will instead focus on the entirety of ac (or equivalently, bd). The claim is that for any a, we can find an appropriate c such that ac will be equivalent to some $\underline{\Delta}$ braid. We will begin by working with single generators, before moving to entire words.

Theorem 4.6. For any natural numbers n and i such that $i < n$, there exists $\partial_n(i) \in B^+$ such that $\sigma_i\partial_n(i) \equiv^+ \underline{\Delta}_n$

Proof. We proceed by induction on n. If $n = 0$, the result holds vacuously.

Base Case : $n = 1$. Then since $i < n$, $i = 0$.

$$\begin{aligned}
\underline{\Delta}_n &= \underline{\sigma}_{0,1} \\
&= \sigma_0 \\
&= \sigma_0 \cdot \varepsilon
\end{aligned}$$

Hence $\partial_n(i) = \varepsilon$.

Induction Step : We consider the case for $n+1$. We know that $i < n+1$. We split into cases based on the value of i in relation to n.

Case 1: $i < n$.

$$\begin{aligned}
\underline{\Delta}_{n+1} &\equiv^+ \underline{\Delta}_n\underline{\sigma}_{n,1} \\
&\equiv^+ \sigma_i w(n-1,i)\underline{\sigma}_{n,1} && \text{(by inductive hypothesis for } n \text{ and } i)
\end{aligned}$$

Hence $\partial_{n+1}(i) = \partial_{n-1}(i)\underline{\sigma}_{n,1}$ when $i < n$.

Case 2: $i = n$ To build the word, we will need the shift function $s : s(\sigma_i) = \sigma_{i+1}$. We may apply s to a word by applying it to each of the word's letters. For example, for $0 \leq i \leq n-1$, $\sigma_i \underline{\sigma}_{n,0} \equiv^+ \underline{\sigma}_{n,0} s(\sigma_i)$ as proved above. Since $\underline{\Delta}_n$ contains only the letters σ_i for $0 \leq i \leq n-1$, by repeated applications of 4.5, $\underline{\Delta}_n \underline{\sigma}_{n,1} \equiv^+ \underline{\sigma}_{n,1} s(\underline{\Delta}_n)$

$$\begin{aligned}
\underline{\Delta}_{n+1} &\equiv^+ \underline{\Delta}_n \underline{\sigma}_{n+1,1} \\
&\equiv^+ \underline{\sigma}_{n+1,1} s(\underline{\Delta}_{n+1}) && \text{(by above statement)} \\
&= \sigma_n \underline{\sigma}_{n,1} s(\underline{\Delta}_{n+1}) && \text{(by definition of } \underline{\sigma}\text{)}
\end{aligned}$$

Hence $\partial_{n+1}(n) = \underline{\sigma}_{n,1} s(\underline{\Delta}_{n+1})$ □

By the above theorem, any two generators have a a delta braid as a common multiple. Let's now go for entire words!

Theorem 4.7. For a word a of length at most l, in which the index of the largest generator is i there exists a word $p_l(a)$ such that $a\partial_l(a) \equiv^+ (\underline{\Delta}_i)^l$

Proof. The full proof is only sketched here; it is done by induction on l. We note that it is not sufficient to merely concatenate the $\partial_n(i)$ for each generator, because of the lack of commutativity. If we look closely,

$$\sigma_{a_1}\sigma_{a_2}...\sigma_{a_{l-1}}\sigma_{a_l}\partial_i(a_l)\partial_i(a_{l-1})...\partial_i(a_1)$$

will not work out. We may replace $\sigma_{a_l}\partial_i(a_l)$ by $\underline{\Delta}_i$ in the middle, obtaining

$$\sigma_{a_1}\sigma_{a_2}...\sigma_{a_{l-1}}\underline{\Delta}_i\partial_i(a_{l-1})...\partial_i(a_1)$$

But then $\sigma_{a_{l-1}}$ is not directly next to $\partial_i(a_{l-1})$.

To fix this, we will need the flip function $f_n : f_n(\sigma_i) = \sigma_{n-i}$ (where $i \leq n$). We may apply f_n to a word by applying it to each of the word's letters, so long as the word is bounded by n. Thus, we know that $\underline{\Delta}_i f_i(\partial_i(a_{l-1})) \equiv^+ \partial_i(a_{l-1})\underline{\Delta}_i$.

So, we will set

$$p_l(a) = p_l(\sigma_{a_1}\sigma_{a_2}...\sigma_{a_{l-1}}\sigma_{a_l}) = \partial_i(a_l)\ f_i(\partial_i(a_{l-1}))\ \partial_i(a_{l-2})\ f_i(\partial_i(a_{l-3}))...$$

Above, the ... does not continue infinitely, but whether or not we apply f_i to a_1 depends on the parity of l, so it is easier to let the reader infer the pattern than to write out casework. □

Then, if we have two words a, b and we let l be the maximum of the respective lengths of a and b, the above theorem will give us the common multiples.

4.3.2 Finitely Many Strands

Let us briefly recall that for any two words $a, b \in B_n^+$ (the braid monoid on n strands), saying $a \equiv_n^+ b$ means that a and b are equivalent under the n-strand braid monoid relations. Equivalently, we know there exists a B_n^+-derivation from a to b. Let the derivation be listed as : $R_0, R_1, ..., R_k$.

Definition 4.8. A word w in either B^+ or B_m^+ is bounded by $n \in \mathbb{N}$ if the index of every generator occurring in w is strictly less than n.

Remark. For any word $w \in B_n^+$, a is bounded by n

Theorem 4.8. Let a be bounded by n. If $a \equiv^+ b$, then every word $R_0, R_1, ..., R_k$ in the derivation from a to b is bounded by n

Proof. Formally, use induction on k. We will omit the details; intuitively, this holds because braid monoid relations do not change the set of symbols in a word. □

Thus, if a is bounded above by n, and for some b we have $a \equiv^+ b$, then we also have $a \equiv_n^+ b$. By this reasoning, for any $a \in B_n^+$ where the length of a is l, since $a\partial_n(a) \equiv^+ (\underline{\Delta}_n)^l$ and $\underline{\Delta}_n$ is bounded by n, we know that $\partial_n(i)$ is bounded by n. Therefore, the common multiples of $a, b \in B_n^+$ must both be bounded by n. Hence, since $ac \equiv^+ bd$ and everything is bounded, $ac \equiv_n^+ bd$

In Lean, this requires a bit of delicacy with typecasting and juggling hypotheses. Note that a and b are originally of type `Fin n`, are cast to $\mathbb{N}$, and then we obtain c and d also in $\mathbb{N}$. Finally, we are able to cast ac and bd back down to `Fin n` because of the boundedness hypotheses — but it is far preferable to copy-pasting a thousand lines of code!

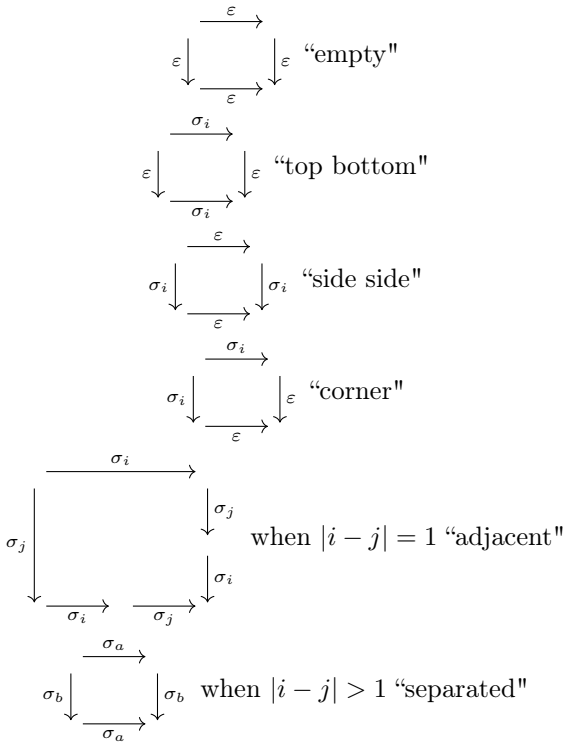
4.4 Cancellativity

Dehornoy pioneeered the use of rectangular grid representations to model re-write sequences [12]. This structure is used to prove cancellativity in the braid monoid. Grids can be generalized to represent presented monoids on any set of relations; here, we only consider the specific case of grids on the braid relations.

4.4.1 Grid Definition

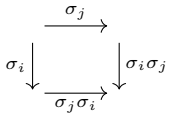
A grid is defined inductively as either a cell, or two grids abutted horizontally or vertically so that the relevant edges match.

Here are the cells (let i and j represent natural numbers):

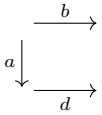
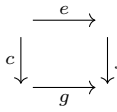
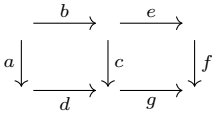


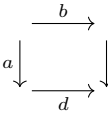
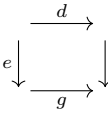
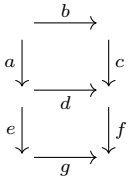
Note that in each cell, the two paths from top-left to bottom-right are equivalent under the braid relations.

Now, we may simplify our diagrams by artificially drawing two arrows as one, and concatenating their labels. This does not change the grid structure; it is merely notation. Thus, for the "separated" case, we could draw:

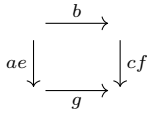


Now, we compose the cells to form larger grids.

Horizontal abutting: If  and  are grids, then so too is 

Vertical abutting: If  and  are grids, then so too is 

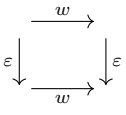
Once again, we use simplified notation - a, b, c, d, e, f may be braid monoid words of any length. We may also write a grid omitting its inner arrows, simply showing the four sides and the words labelling them. The last-drawn grid would thus be

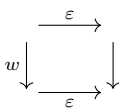


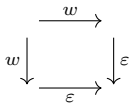
As a non-diagrammatic representation, we may write that we have a grid from (ae, b) to (cf, g). In Lean, we give a predicate on four inputs, representing the left, top, right, and bottom sides. Thus, we define

```
inductive grid : FreeMonoid ℕ → FreeMonoid ℕ → FreeMonoid ℕ → FreeMonoid ℕ →
    Prop
  | empty : grid 1 1 1 1
  | top_bottom (i : ℕ) : grid 1 (of i) 1 (.of i)
  | sides (i : ℕ) : grid (of i) 1 (of i) 1
  | top_left (i : ℕ) : grid (of i) (of i) 1 1
  | adjacent (i k : ℕ) (h : i.dist k = 1) : grid (of i) (of k) (of i * of k) (of
    k * of i)
  | separated (i j : ℕ) (h : i.dist j > 1) : grid (of i) (of j) (of i) (of j)
  | vertical (h1: grid u v u' v') (h2 : grid a v' c d) : grid (u * a) v (u' * c) d
  | horizontal (h1: grid u v u' v') (h2 : grid u' b c d) : grid u (v * b) c (v' * d)
```

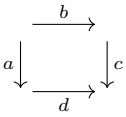
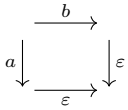
By induction, we see that for any word w, we have the following grids:

 "top bottom word"

 "side side word"

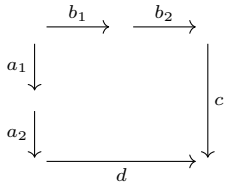


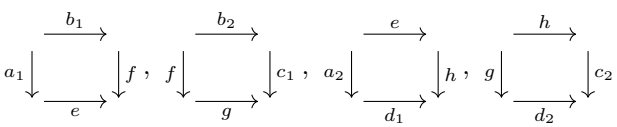
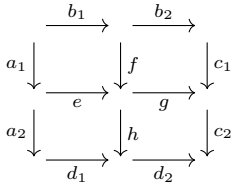
"corner word"

Now it is time to begin to relate these grids to braid monoid equality. Another quick proof by induction shows that if we have a grid  then $ad \equiv^+ bc$. More specifically, if we have  then $a \equiv^+ b$. Thus, if we have a grid, we can deduce equality under the braid monoid relations. What about the other direction — if we know two words a, b are equivalent under the braid monoid relations, is there a grid witnessing their equality? How about a grid who bottom and right sides are both ε? (or in our shorter notation, is there a grid from (a, b) to $(\varepsilon, \varepsilon)$?) In the following sections, we set up the machinery to answer this in the affirmative.

4.4.2 Splitting Grids

Given a grid like so, with words of length > 1 on the top and right sides, one might well guess that it was made out of four[26] smaller grids:



And that would be correct.[27] By induction on the structure of the above grid, we can prove that there exist four grids  , making 

Unfortunately, this theorem does not tell us anything further about the values of e, f, g, h, and only that $c_1 \cdot c_2 = c$ and $d_1 \cdot d_2 = d$.

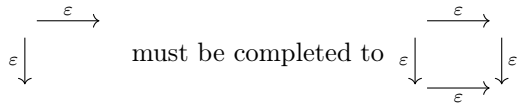
4.4.3 Determinative Spines

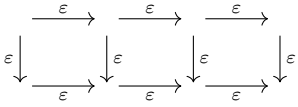
For some grids, however, knowing their left and top sides (the *spine* of a grid) is sufficient to know their bottom and right sides. These proofs are not given explicitly in the reference text, so I will list out the results here.[28] We see that

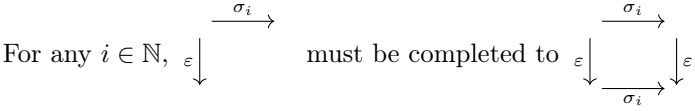
[26] One need not split into four – a grid could be split horizontally intro two pieces or vertically into two pieces. Repeated applications could split a grid all the way down to its individual cells

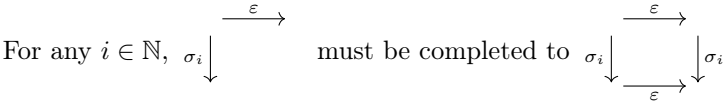
[27] Note that this property only holds for splitting the top arrow or the left-hand arrow. We have grids with word length > 1 on the bottom and/or right that are not made of smaller grids – see the "adjacent" cell

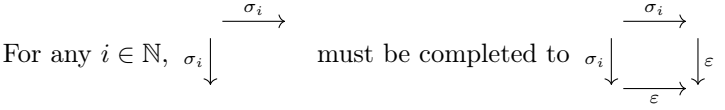
[28] The proofs are routine, and I will spare the reader: each is approx 100 lines, as we must consider each of the 7 cases for a grid.



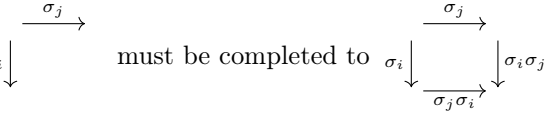
Here, we take advantage of the notation shortcuts from the previous section. We draw only the edges of a grid, leaving the inside blank - it is entirely possible for the above right grid to look like so when fully drawn :  . Continuing on, we have

For any $i \in \mathbb{N}$, 

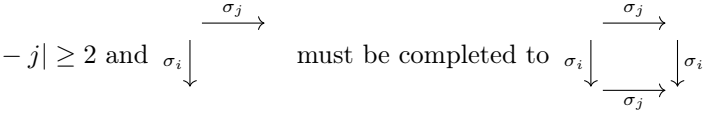
For any $i \in \mathbb{N}$, 

For any $i \in \mathbb{N}$, 

Now we discuss the case where both legs of the spine are non-empty, but distinct.

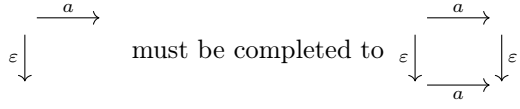
If $|i-j| = 1$, then 

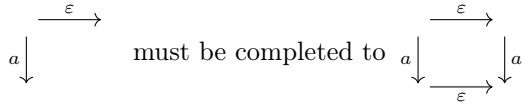
(again we note that these are the labels on the entire bottom/right edges - no separate arrows are shown)

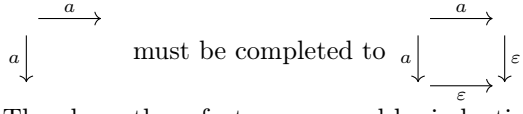
Otherwise, we have $|i-j| \geq 2$ and 

The above properties are proved by a quick induction on the structure of a grid having the given spine. Overall, these are straightforward inductions. We prove them in the order shown above, so that the more complicated ones may use the simpler ones as lemmas or base cases.

No longer restricting ourselves to a single generator σ_i on a spine leg, but rather allowing an entire word a :







The above three facts are proved by induction on the length of a.

At this point, it is tempting to try to see that a grid with *any* spine is fully determined. In fact this is the case! We will show that given any spine, there exists a grid with that spine, and the labels on the other two edges are uniquely determined.

Let's being with uniqueness.

Theorem 4.9. Assume we have two grids, $\begin{array}{ccc} \cdot & \xrightarrow{b} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle c} \\ \cdot & \xrightarrow[d]{} & \cdot \end{array}$ and $\begin{array}{ccc} \cdot & \xrightarrow{b} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle c'} \\ \cdot & \xrightarrow[d']{} & \cdot \end{array}$

Then the words c and c' must be equal, and the same holds for d and d'.

Proof. We proceed by structural induction on $\begin{array}{ccc} \cdot & \xrightarrow{b} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle c} \\ \cdot & \xrightarrow[d]{} & \cdot \end{array}$. In the case where this grid is a cell, the above properties of determinative spines will finish the proof. All that remains is the horizontal and vertical cases. We give a proof for the horizontal case; the vertical case is analogous.

The grids must look like so, $\begin{array}{ccccc} \cdot & \xrightarrow{b_1} & \cdot & \xrightarrow{b_2} & \cdot \\ {\scriptstyle a}\downarrow & & & & \downarrow{\scriptstyle c} \\ \cdot & \xrightarrow[d_1]{} & \cdot & \xrightarrow[d_2]{} & \cdot \end{array}$ and $\begin{array}{ccccc} \cdot & \xrightarrow{b_1} & \cdot & \xrightarrow{b_2} & \cdot \\ {\scriptstyle a}\downarrow & & & & \downarrow{\scriptstyle c'} \\ \cdot & \xrightarrow[d'_1]{} & \cdot & \xrightarrow[d'_2]{} & \cdot \end{array}$ where $d = d_1 d_2$, $d' = d'_1 d'_2$, and and $b = b_1 b_2$. By splittability, we know that for some e and e', the grids have the following structure:

$$\begin{array}{ccccc} \cdot & \xrightarrow{b_1} & \cdot & \xrightarrow{b_2} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle e} & & \downarrow{\scriptstyle c} \\ \cdot & \xrightarrow[d_1]{} & \cdot & \xrightarrow[d_2]{} & \cdot \end{array} \quad \begin{array}{ccccc} \cdot & \xrightarrow{b_1} & \cdot & \xrightarrow{b_2} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle e'} & & \downarrow{\scriptstyle c'} \\ \cdot & \xrightarrow[d'_1]{} & \cdot & \xrightarrow[d'_2]{} & \cdot \end{array}$$

By our induction hypothesis applied to the left-hand subgrid, we know that $d_1 = d'_1$ and $e = e'$. Thus, the second grid becomes: $\begin{array}{ccccc} \cdot & \xrightarrow{b_1} & \cdot & \xrightarrow{b_2} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle e} & & \downarrow{\scriptstyle c'} \\ \cdot & \xrightarrow[d_1]{} & \cdot & \xrightarrow[d'_2]{} & \cdot \end{array}$ By another application of the inductive hypothesis now to the right-hand side of the second grid, we see that $c = c'$ and $d_2 = d'_2$. Hence, $d = d'$, as desired. □

Now, assuming a grid exists with a given spine, its right-hand and bottom edges are uniquely defined. But does every spine have an associated grid? We will see that the answer is yes, but first we develop a helpful property of grids, called *stability*.

4.4.4 Stability of Grids

Definition 4.9 (Stability)**.** A grid $\begin{array}{ccc} \cdot & \xrightarrow{b} & \cdot \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle c} \\ \cdot & \xrightarrow[d]{} & \cdot \end{array}$ is called stable if for $a', b' \in B^+$ such that $a \equiv^+ a'$ and $b \equiv^+ b'$, there exist $c', d' \in B^+$ such that we have a grid $\begin{array}{ccc} \cdot & \xrightarrow{b'} & \cdot \\ {\scriptstyle a'}\downarrow & & \downarrow{\scriptstyle c'} \\ \cdot & \xrightarrow[d']{} & \cdot \end{array}$ with $c \equiv^+ c'$ and $d \equiv^+ d'$

Theorem 4.10. All grids are stable.

The proof itself is a bit of a bear : it requires a specific inductive form of re-writing. We proceed by triple induction : first on the length of ac, then structurally on the derivation $a \equiv^+ a'$, and then structurally on the derivation $b \equiv^+ b'$.

Here are a few lemmas to begin with :

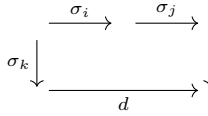
Lemma 4.11. Grids are symmetric : If we have a grid from (a, b) to (c, d), then there is a grid from (b, a) to (d, c)

Lemma 4.12. Stability is symmetric : If we the grid from (a, b) to (c, d) is stable, then there is a (unique!) grid from (b, a) to (d, c), and it is stable

Proceeding on with the proof of Theorem 4.10, we first consider a few specific cases, and break them out into lemmas.

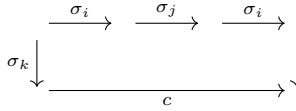
We begin with the case where the left-hand side of the grid has length 1 — that is, if it is σ_k, and the top of the grid is one part of a braid relation — either $\sigma_i\sigma_j$ with $|i - j| > 1$ or $\sigma_i\sigma_j\sigma_i$ with $|i - j| = 1$. We show the results for any value of k, which requires casing within the proof based on the relative value of k as compared to i and j.

Lemma 4.13. For any $k \in \mathbb{N}$, $i, j \in \mathbb{N}$ with $|i - j| > 1$, and braid monoid words c and d, the grid



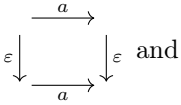
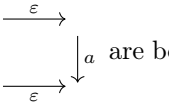
is stable

Lemma 4.14. For any $k \in \mathbb{N}$, $i, j \in \mathbb{N}$ with $|i - j| = 1$, and braid monoid words c and d, the grid



is stable

Next, we consider the case where one part of the spine is the empty string.

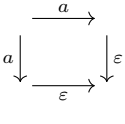
Lemma 4.15. For any braid monoid word a, the grids  and  are both stable.

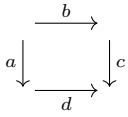
Note that once we have proven one of the two to be stable, Lemma 4.12 lets us conclude the other is stable.

With these lemmas complete, we embark on the main proof. Details will be omitted here, for the three nested inductions lead to many, many cases. Essentially, we split the grid when possible and apply the induction hypothesis or previous lemmas.

Stability does give us a result we looked for early in this chapter:

Theorem 4.16. If $a \equiv^+ b$, there there is a grid from (a, b) to $(\varepsilon, \varepsilon)$.

Proof. From the section on determinative spines, we know that there is a grid  . Since

this grid is stable, and $a \equiv^+ a$ and $a \equiv^+ b$ (by hypothesis), we know there is a grid 

where $c \equiv^+ \varepsilon$ and $d \equiv^+ \varepsilon$. Since length is preserved under braid monoid equivalence, we know the length of both c and d must equal zero — hence $c = \varepsilon$ and $d = \varepsilon$. Thus we have obtained the desired grid. □

Next up, armed with the powerful theorem of stability, we are able to prove existence of a grid from any given spine. Intuitively, when building a grid, there is only one choice of cell to add in at any step. Never do we reach an impasse, nor a multiplicity of choices.

Theorem 4.17. For any $a, b \in B^+$, there is a grid with spine $\begin{smallmatrix} & \xrightarrow{b} & \\ a\downarrow & & \\ & & \end{smallmatrix}$

Proof. Let $c, d \in B^+$ be such that $ac \equiv^+ bd$. Then we know there is a grid $\begin{smallmatrix} & \xrightarrow{bd} & \\ ac\downarrow & & \downarrow\varepsilon \\ & \xrightarrow{\varepsilon} & \end{smallmatrix}$. We may split this grid to obtain

$$\begin{array}{ccccc} & \xrightarrow{b} & & \xrightarrow{d} & \\ a\downarrow & & f\downarrow & & \varepsilon\downarrow \\ & \xrightarrow{e} & & \xrightarrow{g} & \\ c\downarrow & & h\downarrow & & \varepsilon\downarrow \\ & \xrightarrow{\varepsilon} & & \xrightarrow{\varepsilon} & \end{array}$$

And thus we have a grid $\begin{smallmatrix} & \xrightarrow{b} & \\ a\downarrow & & f\downarrow \\ & \xrightarrow{e} & \end{smallmatrix}$

□

4.4.5 Final Result

We prove left-cancellativity first. That is, we aim to prove that if $ca \equiv^+ cb$, then $a \equiv^+ b$. We proceed by induction on the length of c. We show here only the case where the length of c is one; when the length of c is zero the result is immediate, and other cases follow from the inductive hypothesis.

Since the length of c is one, c must equal σ_i for some i. Thus, $\sigma_i a \equiv^+ \sigma_i b$. By theorem **??**, we know we have a grid

$$\begin{array}{ccc} & \xrightarrow{\sigma_i b} & \\ \sigma_i a\downarrow & & \downarrow\varepsilon \\ & \xrightarrow{\varepsilon} & \end{array}$$

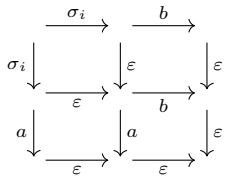
Then for some $c, d, e, f, g, h, i, j, k$ we may split this grid as so:

$$\begin{array}{ccccc} & \xrightarrow{\sigma_i} & & \xrightarrow{b} & \\ \sigma_i\downarrow & & c\downarrow & & h\downarrow \\ & \xrightarrow{d} & & \xrightarrow{e} & \\ a\downarrow & & f\downarrow & & i\downarrow \\ & \xrightarrow{j} & & \xrightarrow{k} & \end{array}$$

Since $hi \equiv^+ \varepsilon$ and $jk \equiv^+ \epsilon$, we have $h = i = j = k = \varepsilon$. By determinative spines on the top-left subgrid, we know $c = \varepsilon$ and $d = \varepsilon$. Hence, we have

$$\begin{array}{ccccc} & \xrightarrow{\sigma_i} & & \xrightarrow{b} & \\ \sigma_i\downarrow & & \varepsilon\downarrow & & \varepsilon\downarrow \\ & \xrightarrow{\varepsilon} & & \xrightarrow{e} & \\ a\downarrow & & f\downarrow & & \varepsilon\downarrow \\ & \xrightarrow{\varepsilon} & & \xrightarrow{\varepsilon} & \end{array}$$

Again by determinative spines, we know $f = a$ and $e = b$.



Hence the bottom-right grid witnesses $a \equiv^+ b$ as desired! Left cancellativity is thus shown.

For right-cancellativity, we will make good use of the reversing function rev, which reverses the letters in a representative of the equivalence class. Reversing is well-defined on braid monoid elements, as we saw previously.

Lemma 4.18. For all braid monoid words a and b, $a \equiv^+ b$ if and only if $rev(a) \equiv^+ rev(b)$. Moreover, $rev(ab) = rev(b)rev(a)$.

Now, let us assume $ac \equiv^+ bc$. Thus, we know $rev(ca) \equiv^+ rev(cb)$. Thus, $rev(a)rev(c) \equiv^+ rev(b)rev(c)$ Then we may apply left-cancellativity to conclude that $rev(a) \equiv^+ rev(b)$, so $a \equiv^+ b$ as desired! This procedure of turning a left property into a right property (or vice verse) can be used for common multiples as well.

We have now shown common multiples and cancellativity, so we can perform the Ore localization on the braid monoid and obtain an embedding into the braid group - finally, we can officially drop the quotes around "braid" monoids!

5 Solving the Word Problem

The word problem in a presented group is to determine, given any two words a, b in the presented group, if $a \equiv b$.

There are a number of methods to solve the word problem for braids - there are normal forms, combing algorithms, handle reduction algorithms.

We have chosen Dehornoy's reversing algorithm [13] for the Lean formalization — it runs in polynomial time and is sufficiently simple for a first attempt. Dehornoy goes on later in his book to describe improvements and refinements. This could be done in later work, especially since the machinery so far developed (localization) would be re-used. NOTE — sections 5.3-5.5 have not been completely formalized in Lean. They are included as background information to understand the novel pen-and-paper proof presented in chapter 6.

As a brief sketch, we will be able to convert every braid word to one in a special form, and then compare the special forms of the two braids.

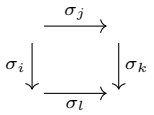
5.1 Reading Grids Backwards

At the moment, we have only discussed reading words off grids following the direction of the arrows. We can expand our understanding of a grid structure by defining a procedure for reading *against* the direction of the arrows.

Instead of the labels on arrows representing elements in BW^+ (the set of braid monoid words), we now allow them to represent elements of BW (the set of braid group words). Inverses are obtained by reading against the direction of the arrow.

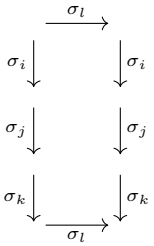
We note that we are discussing words, not elements of the braid group. To make this clear, following Dehornoy, we write $\overline{\sigma_i}$ in place of σ_i^{-1}. Notably, $\overline{\sigma_i}\sigma_i \neq \varepsilon$, for they are clearly different strings. However, $\overline{\sigma_i}\sigma_i \equiv \varepsilon$.

Thus, we can read two paths from the bottom-left to the top-right in the following grid:



the upper is denoted $\overline{\sigma_i}\,\sigma_j$ and the lower is $\sigma_l\overline{\sigma_k}$.

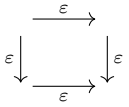
For a side labelled with a word of length > 1, we read the letters of the word in opposite order, and add an overbar to each letter. Consider the following grid:



The upper bottom-left to top-right path would be read as $\overline{\sigma_k}\,\overline{\sigma_j}\,\overline{\sigma_i}\sigma_l$, and the lower $\sigma_l\overline{\sigma_k}\,\overline{\sigma_j}\,\overline{\sigma_i}$. Given a word $w = \sigma_{a_0}\sigma_{a_1}...\sigma_{a_n}$, let $\overline{w}$ denote $\overline{\sigma_{a_n}}\,...\overline{\sigma_{a_1}}\,\overline{\sigma_{a_0}}$

What about arrows labelled ε? Since ε represents the empty string, it makes no contribution to the word read off the grid. Even when read in the opposite direction, it has the same (lack of) effect. For now, we will use the unsatisfactory answer of simply writing ε when faced with an arrow labelled ε, no matter the direction of the arrow.

Why unsatisfactory? In this grid:



both paths from top-left to bottom right, as well as both paths from bottom-left to top-right, would all be labelled ϵ. For the moment, this will pose no problem to our algorithm : it terminates immediately when given an empty word, so we do not need to model the situation with a grid. We will return to this issue in chapter 6, and give a more detailed treatment there.

Notably, we have only given examples of reading a downwards-pointing arrow upwards. We could read left-to-right arrows backwards as well, using the same procedure. However, we will not find a use for this in the following algorithm.

5.2 Semi-Thue Systems

In the section on presented monoids, we discussed a rewriting system which is reflexive, transitive, and symmetric. This is also called a Thue system. Often, however, one's system is not symmetric — this occurs especially when trying to reduce a string to some simpler or canonical form. In that case, we define a *semi*-Thue system. Due to the lack of symmetry, the re-writing relation here is not an equivalence relation, let alone a congruence. We thus cannot follow the approach used for presented monoids, which used quotients.

Instead, we simply semi-Thue systems inductively. Given an alphabet A and a binary relation R on A-strings, we use the $\to_R$ operator to represent the one-step closure : $\forall\, a\, b\, c\, d \in A^*$, $R(a, b) \implies cad \to_R cbd$

Then $\overset{*}{\to}_R$ represents the reflexive, transitive closure of $\to_R$. Inductively, it is defined as :

- reflexivity : For all $a \in A^*$, $a \overset{*}{\to}_R a$
- closure : For all $a, b \in A^*$, $a \to_R b$ implies $a \overset{*}{\to}_R b$
- transitivity : For all $a, b, c \in A^*$, if $a \overset{*}{\to}_R b$ and $b \overset{*}{\to}_R c$, then $a \overset{*}{\to}_R c$

We now define S_R, an alternate construction for $\overset{*}{\to}_R$, which will be useful for certain induction proofs, defined inductively as :

- reflexivity : For all $a \in A^*$, $S_R(a, a)$
- one-step transitive closure : if $S_R(a, b)$ and $b \to_R c$, then $S_R(a, c)$

It is easy to prove by induction that for all $a, b \in A^*$, we have $S_R(a, b)$ if and only if $a \overset{*}{\to}_R b$.

As for the Lean implementation for Semi-Thue systems, we once again have multiple options for the type of A^* : `FreeMonoid A`, `List A`, or `String A`. Because we are not using `Congruence`, we do not need a `Mul` structure on the type we use. There is thus no need for `FreeMonoid`. The `List` API is much stronger than that of `FreeMonoid`. In this case, we will be using our Semi-Thue system for actual calculation, so features like `List.getLast` (this returns the last element of the list) will be quite useful. Moreover, since `FreeMonoid` is defined as `List`, a semi-Thue system based on `List` will still work nicely with the `FreeMonoid`-based `PresentedMonoid` definition.

Although `String` seems at first glance like exactly what we'd want, it is defined as a list of `Char`. For the coming applications, we will need a more general alphabet, one which allows for more complex expressions (e.g. involving option types and ordered pairs) to act as a single element of our alphabet.

5.3 Reversing a Word

(This section is has not been formalized in Lean.) Dehornoy's algorithm is based on a reversing procedure, which we will call `reverse_braid`. When given an initial input word w of the form $\overline{u}v$ (where u and v contain no inverses), `reverse_braid` returns a word w' of the form $u'\overline{v'}$ (where u' and v' contain no inverses), such that $w \equiv w'$.

In fact `reverse_braid` can be applied multiple times to transform any word w into one of the form $u'\overline{v'}$ (where u' and v' contain no inverses), such that $w \equiv w'$. We call this `reverse_any_braid`. It is defined as such : Any word can be written as $\overline{u_1}v_1\overline{u_2}v_2...\overline{u_n}v_n$ (where any u_k or v_k may equal the empty string). One recursively calls `reverse_any_braid` to reverse $\overline{u_2}v_2...\overline{u_n}v_n$ to $u'\overline{v'}$, where $\overline{u_2}v_2...\overline{u_n}v_n \equiv u'\overline{v'}$. Then, we have

$$\overline{u_1}v_1\overline{u_2}v_2...\overline{u_n}v_n \equiv \overline{u_1}v_1u'\overline{v'}$$
$$w \equiv \overline{u_1}v_1u'\overline{v'}$$

Then using `reverse_braid`, $\overline{u_1}v_1$ reverses to some $u''\overline{v''}$, such that $\overline{u_1}v_1 \equiv u''\overline{v''}$. Thus we obtain

$$w \equiv u''\overline{v''}u'\overline{v'}$$

With one last application of `reverse_braid`, we reverse $\overline{v''}u'$ to $v'''\overline{u'''}$, obtaining

$$w \equiv u''v'''\overline{u'''}\ \overline{v'}$$
$$w \equiv u''v'''\overline{v'u'''}$$

And so w is reversed!

So far, we have treated `reverse_braid` as a black box. Now let's look at exactly how this magic is performed.

We begin with a reversing relation R_{rev} generated inductively by:

$R_{rev}(\overline{\sigma_i}\sigma_i, \varepsilon)$ for all i
$R_{rev}(\overline{\sigma_i}\sigma_j, \sigma_j\sigma_i\overline{\sigma_j}\ \overline{\sigma_i})$ when $|i-j| = 1$
$R_{rev}(\overline{\sigma_i}\sigma_j, \sigma_j\overline{\sigma_i})$ when $|i-j| \geq 2$

Note that if we have $R_{rev}(a, b)$, $a \equiv b$ under the braid relations; this extends to the Semi-Thue system generated by R_{rev} as well.

We define the algorithm `reverse_braid`[29] which, given a braid group word, iteratively replaces the first occurrence of a negative-positive pair by an equivalent positive-negative word (following R_{rev}) until no negative-positive pairs remain.[30] Thus, the input, final output, and every intermediate step are all equivalent under $S_{R_{rev}}$.

Remark. If a braid group word w reverses (under `reverse_braid` or `reverse_any_braid`) to $a\bar{b}$ then $w \equiv a\bar{b}$

Now, we must verify termination. Notably, once we show termination for `reverse_braid`, termination for `reverse_any_braid` follows immediately. So, we will below simply show termination for `reverse_braid`.

[29] we will use the terminology `reverse_braid` for the case when the initial input is in negative-positive form, and `reverse_any_braid` otherwise. Note the recursive definition of S_R, which we will use to model the reversing, always preserves the first element of the initial input pair in any other appeal to S_R

[30] The replacement may introduce new negative-positive pairs, so termination is not straightforward

5.4 Reversing Grids

(this section hasn't been fully formalized in Lean). We show termination by connecting `reverse_braid` to grids (which have a finite number of cells; hence we will see there are only a finite number of steps in `reverse_braid`).

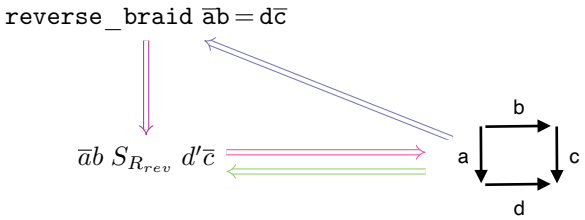
Lemma 5.1. If there is a grid $\begin{array}{ccc} & \xrightarrow{b} & \\ a\downarrow & & \downarrow c \\ & \xrightarrow{d} & \end{array}$ then `reverse_braid` $\overline{\texttt{a}}\texttt{b}$ terminates and returns $\texttt{d}\overline{\texttt{c}}$.

This lemma is not found in Dehornoy;[31] I aim to give a rigorous proof[32] as future work.

We now show that `reverse_braid` terminates. Consider some input $\overline{a}b$. We know by theorem 4.17 that there is a grid from (a, b) to some c, d. Thus, we may apply our lemma 5.1 to see that `reverse_braid` $\overline{\texttt{a}}\texttt{b}$ terminates and returns $\texttt{d}\overline{\texttt{c}}$.

Next, in order to show correctness, we connect `reverse_braid` to grids via an intermediate structure, $S_{R_{rev}}$. This is done with four subproofs, each corresponding to an implication arrow in the diagram below. We show the purple and green[33] arrows; the blue arrow is lemma 5.1; the pink requires a novel proof and will be discussed in chapter 6.

In the below diagram, we use `reverse_braid` $\texttt{a}\overline{\texttt{b}}$ `=` $\overline{\texttt{d}}\texttt{c}$ to mean "`reverse_braid ab` terminates and returns $\overline{\texttt{d}}\texttt{c}$ for some braid monoid words `d, c`"



Lemma 5.2. For $a, b, c, d \in BW^+$, if `reverse_braid` $\overline{\texttt{a}}\texttt{b}$ terminates and returns $\texttt{d}\overline{\texttt{c}}$ then $\overline{a}b \; S_{R_{rev}} \; d\overline{c}$

Proof. Immediate from the definition of `reverse_braid`. □

Theorem 5.3. If there is a grid $\begin{array}{ccc} & \xrightarrow{b} & \\ a\downarrow & & \downarrow c \\ & \xrightarrow{d} & \end{array}$ then $\overline{a}b \; S_{R_{rev}} \; d\overline{c}$

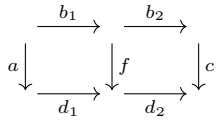
Proof. Assume we have a grid $\begin{array}{ccc} & \xrightarrow{b} & \\ a\downarrow & & \downarrow c \\ & \xrightarrow{d} & \end{array}$. We proceed by induction on the grid. If it is a cell, either the relation $S_{R_{rev}}$ holds by reflexivity or by a single application of R_{rev}. The induction

[31] He implicitly uses the claim that `reverse_braid` $\overline{\texttt{a}}\texttt{b}$ terminates and returns $\texttt{d}\overline{\texttt{c}}$ if and only if $\overline{a}b \; S_{R_{rev}} \; d\overline{c}$. The forward direction is immediate; the reverse is tricky. I am thus taking a different path to show termination.

[32] Vague proof sketch to convince the reader this is true; a rigorous proof will be future work : We will consider a list of paths through the grid from the bottom-left to the top-right, arranged in a particular order (beginning from the left side - top side path, we will "pop in" the first up-over corner (aka the first positive-negative string of length 2) in the path, continuing until we reach the bottom side - right side path). Each will correspond to a step in `reverse_braid` (due to empty arrows, multiple paths consecutive in the list may correspond to the same step in `reverse_braid`).

[33] In the end, the green arrow will not be necessary. However, I have formalized it in Lean, and I suspect it may be helpful for creating a proof for the blue arrow (lemma 5.1), so I have included the details here

steps proceed straightforwardly thanks to properties of semi-Thue systems — here is a sketch of the horizontal appending case. The grid must take the form



where $b = b_1 b_2$ and $d = d_1 d_2$. By our inductive hypothesis on the left-hand subgrid, we know

$$\overline{a} b_1 \; S_{R_{rev}} \; d_1 \overline{f}$$

By properties of semi-Thue systems, this means

$$\overline{a} b_1 b_2 \; S_{R_{rev}} \; d_1 \overline{f} b_2$$

By our inductive hypothesis on the right-hand grid,

$$\overline{f} b_2 \; S_{R_{rev}} \; d_2 \overline{c}$$

And again by properties of semi-Thue systems,

$$d_1 \overline{f} b_2 \; S_{R_{rev}} \; d_1 d_2 \overline{c}$$

By transitivity, we thus achieve our goal, showing

$$\overline{a} b_1 b_2 \; S_{R_{rev}} \; d_1 d_2 \overline{c}$$

Or more succinctly,

$$\overline{a} b \; S_{R_{rev}} \; d \overline{c}$$

□

5.5 Correctness

(this section has not been formalized in Lean) We begin with the procedure for showing equality in the braid monoid. Given two braid monoid words a, b, $a \equiv^+ b$ exactly when $\overline{a} b S_{R_{rev}} \varepsilon$.

Theorem 5.4. *$a \equiv^+ b$ if and only if $\overline{a} b S_{R_{rev}} \varepsilon$*

Proof. By theorem 5.3, we have $a \equiv^+ b$ if any only if we have a grid from (a, b) to $(\varepsilon, \varepsilon)$, which was shown in theorem 4.16. □

How about solving the problem in the group? If we want to know if $w_1 \equiv w_2$, we can show that $w_1 (w_2)^{-1} \equiv \varepsilon$. Here the the procedure : given words $w1, w_2$, reverse $w_1 (w_2)^{-1}$ to a word of the form $u\overline{v}$. Next, reverse $\overline{v}u$ (note the change in order!) to something of the form $w\overline{z}$. I claim $w_1 \equiv w_2$ exactly when $w\overline{z} = \varepsilon$. As this procedure consists of two reversings, and we have shown that reversing terminates (for the original reversing procedure, which extends to the general reversing procedure for any word in BW), this procedure terminates.

We simply must show correctness. Since $w_1 (w_2)^{-1} \equiv u\overline{v}$ (because reversed words are equivalent under the braid group relations as previously shown),

$$
\begin{aligned}
w_1(w_2)^{-1} \equiv 1 &\iff u\overline{v} \equiv 1 \\
&\iff u \equiv v \\
&\iff u \equiv^{+} v && \text{(by embedding)} \\
&\iff \text{there is a grid from} (u, v) \text{ to } (\varepsilon, \varepsilon) && \text{(by theorem 4.16)} \\
&\iff \overline{u}v \; S_{R_{rev}} \; \varepsilon && \text{(by theorem 5.3 and the pink arrow)}
\end{aligned}
$$

Et voila! Thus, we have a terminating and correct algorithm to solve the braid word problem! Except ... we skipped proving the pink arrow, didn't we?

6 Fortifying Dehornoy's Approach

6.1 Dehornoy's Work

Let's review the bit to be proved:

Lemma 6.1. If $\overline{a}b\ S_{R_{rev}}\ a'\overline{b'}$, then there is a grid $\begin{array}{ccc} & \xrightarrow{\ b\ } & \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b'} \\ & \xrightarrow[\ a'\ ]{} & \end{array}$

Patrick Dehornoy wrestled with this notion. In the textbook, he gives the following proof sketch:

> Inversely, when $\overline{u}v\ S_{R_{rev}}\ u_o\overline{v_0}$, consider a finite sequence of reversing steps leading from $\overline{u}v$ to $u_o\overline{v_0}$. Let Γ_0 be the diagram obtained by writing vertically and from top to bottom the letters of u and horizontally from left to right those of v starting from a common summit. Then following the sequence of elementary reversings starting with$\overline{u}v$ corresponds to constructing step by step from Γ_0 a grid Γ with source (u, v) and target (u_0, v_0).

It's so tempting, one can practically see the grid being built step-by-step in one's mind. Alas, this is not a proof, and other papers of Dehornoy's which discuss reversing grids also do not provide proof [12]. Naive attempts based on keeping track of the cells filled in unfortunately fail to capture the complexity of the construction required.

Below, we give a pen-and-paper proof of this claim. The proof defines two intermediate structures: grid-style re-writing and partial grids. We give a proof sketch showing we can convert the normal re-writing to grid-style rewriting. Then we show grid-style rewriting corresponds to a partial grid. Finally, we connect the idea of partial grids to regular grids.

6.2 Grid-Style Re-Writing

Recall that Dehornoy's "reversing" relation R_{rev} is defined inductively as so:

$R_{rev}(\overline{\sigma}_i\sigma_i, \varepsilon)$ for all i
$R_{rev}(\overline{\sigma_i}\sigma_j, \sigma_j\sigma_i\overline{\sigma_j}\ \overline{\sigma_i})$ when $|i-j| = 1$
$R_{rev}(\overline{\sigma_i}\sigma_j, \sigma_j\overline{\sigma_i})$ when $|i-j| \geq 2$

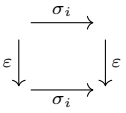
And from this we generate the Semi-Thue system $S_{R_{rev}}$ Because the pair $\overline{\sigma}_i\sigma_i$ re-writes under $S_{R_{rev}}$ to the empty string, no trace of its former existence would be preserved in the re-write sequence. Yet such an occurrence is carefully noted in a grid; it would look like $\begin{array}{ccc} & \xrightarrow{\ \sigma_i\ } & \\ {\scriptstyle \sigma_i}\downarrow & & \downarrow{\scriptstyle \varepsilon} \\ & \xrightarrow[\ \varepsilon\ ]{} & \end{array}$. The arrows labelled ε remain present in the grid, even though they each represent an empty string. While we will ignore them when reading off a path through the grid, they are essential for the construction of a grid — these rectangular grids may not have any gaps, and cells with ε-labelled arrows fill in any potential gaps.

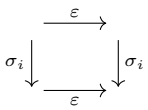
So, we need to build a system which will preserve these empty strings and the direction in which their arrow points in the grid. We add two new symbols to our alphabet : ι and $\overline{\iota}$. ι represents a right-pointing ε-arrow, and $\overline{\iota}$ represents a down-pointing ε-arrow. Now, we give a new relation R_{grid} which tracks these ι's and $\overline{\iota}$'s. R_{grid} is defined inductively by

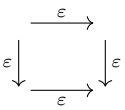
$R_{grid}(\overline{\sigma}_i\sigma_i, \iota\overline{\iota})$ for all $i \in \mathbb{N}$
$R_{grid}(\overline{\iota}\sigma_i, \sigma_i\overline{\iota})$ for all $i \in \mathbb{N}$
$R_{grid}(\overline{\sigma}_i\ \iota, \iota\ \overline{\sigma}_i)$ for all $i \in \mathbb{N}$

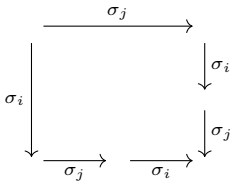
$R_{grid}(\overline{\iota}\,\iota,\ \iota\,\overline{\iota})$
$R_{grid}(\overline{\sigma_i}\sigma_j, \sigma_j\sigma_i\overline{\sigma_j}\,\overline{\sigma_i})$ when $|i-j|=1$
$R_{grid}(\overline{\sigma_i}\sigma_j, \sigma_j\overline{\sigma_i})$ when $|i-j|\geq 2$

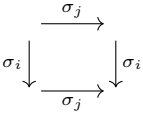
These are based on grid cells; if we have $R_{grid}(a,b)$, we know there is a cell with a as the left-top border and b as the the bottom-right border (reading, as always in the section, north-east, and using an overbar to denote travelling opposite an arrow's direction).

$R_{grid}(\overline{\iota}\sigma_i, \sigma_i\overline{\iota})$ corresponds to the cell 

$R_{grid}(\overline{\sigma}_i\ \iota, \iota\,\overline{\sigma}_i)$ corresponds to the cell 

$R_{grid}(\overline{\iota}\,\iota,\ \iota\,\overline{\iota})$ corresponds to the cell 

$R_{grid}(\overline{\sigma_i}\sigma_j, \sigma_j\sigma_i\overline{\sigma_j}\,\overline{\sigma_i})$ when $|i-j|=1$ corresponds to the cell 

$R_{grid}(\overline{\sigma_i}\sigma_j, \sigma_j\overline{\sigma_i})$ when $|i-j|\geq 2$ corresponds to the cell 

Next, we make explicit the procedure to convert from reversing re-writes to grid-style rewrites!

6.3 Building Grid-Style Rewriting

Let $S = \{\sigma_i | i \in \mathbb{N}\}$

Definition 6.1. A positive word is an element of the set S^*; a negative word is an element of the set $(\overline{S})^*$

Definition 6.2. A positive grid word is an element of the set $(S \cup \{\iota\})^*$; a negative grid word is an element of the set $(\overline{S \cup \{\iota\}})^*$

Note that every negative word is a negative grid word, and similarly for positive words. Conversely, we may convert grid words back to "regular" words by deleting all instances of ι or $\overline{\iota}$ from them.

Theorem 6.2. Let a, b, c, d be positive words. If $\overline{a}b$ rewrites to $c\overline{d}$ under R_{rev}, then there exist positive grid words e, f such that $\overline{a}b$ rewrites to $e\overline{f}$ under R_{grid}. Moreover, when we convert e to a regular word, it will be a positive word equal to c. When we convert f to a word, it will be a positive word equal to d.

Proof. (Sketch)

We consider the relations of R_{grid} which do not correspond to those of R_{rev}, and call them R_{one}. That is, R_{one} is defined inductively by

$R_{one}(\overline{\iota}\,\iota,\ \iota\,\overline{\iota})$

$R_{one}(\overline{\sigma}_i\ \iota, \iota\ \overline{\sigma}_i)$
$R_{one}(\bar{\iota}\sigma_i, \sigma_i\bar{\iota})$

Given any string in our alphabet (all the generators, their overbar-versions, and ι and $\bar{\iota}$), we can define the algorithm `move_ones` which iteratively re-writes the first occurrence of $\bar{\iota}\ \iota$, $\overline{\sigma}_i\ \iota$, or $\bar{\iota}\sigma_i$ according to R_{one}.

I claim that this `move_ones` algorithm terminates. First, let us give an order $<_A$ on our alphabet. $<_A$ is the smallest transitive relation satisfying:

- $\sigma_i <_A \sigma_j$ iff $i < j$
- $\overline{\sigma_i} <_A \overline{\sigma_j}$ iff $i < j$
- $\sigma_i <_A \overline{\sigma_j}$ for all natural numbers i, j
- $\iota <_A \bar{\iota}$
- $\sigma_i <_A \iota$ for all natural numbers i
- $\bar{\iota} <_A \overline{\sigma_i}$ for all natural numbers i

$<_A$ is well-founded; the proof is by tedious casework, so I shall omit it here. (It is formalized in Lean and available in the project repository).

Definition 6.3. Given an alphabet S' and a relation $R_{S'}$ on S', the shortlex relation over $R_{S'}$ is defined on S'-strings by first comparing string length, and in the case of equal length, comparing lexicographically over $R_{S'}$.

Theorem 6.3. If $R_{S'}$ is well-founded, then the shortlex order over $R_{S'}$ is also well-founded. Again, the proof is tedious, so omitted here. But the reader can rest assured it is formalized!

Thus, the shortlex order on $<_A$ is well-founded. Note that every call to `move_ones` is strictly decreasing under the shortlex order on $<_A$. Why? The length of the string never changes when we apply `move_ones`. Thus, we move to the lexicographic ordering, and careful inspection of the re-writes shows that the lexicographic order induced by $<_A$ decreases at each re-write. Thus, thanks to the well-foundedness of the shortlex order on $<_A$, `move_ones` is a terminating procedure.

Now, with the machinery set up, we can give a constructive proof. Here is the algorithm: Begin with the string $\overline{a}b$. Now, repetitively follow this procedure: Apply the next re-write from the R_{rev}-derivation, then apply `move_ones`. Repeat for each step in the R_{rev}-derivation.

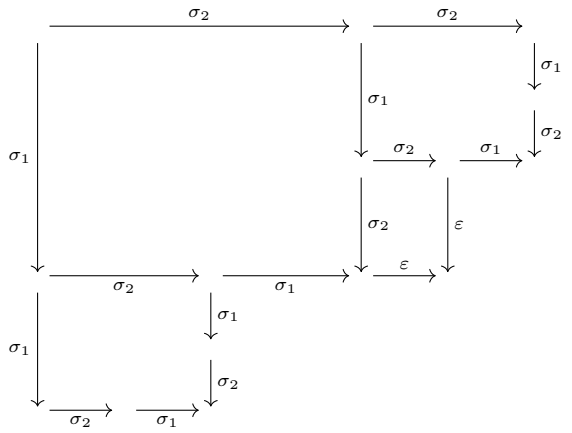
Termination is clear, for derivations are finite and we have shown `move_ones` terminates.

However, at each next step, when we go to perform the re-write from R_{rev}, how do we know that we are set up and in position for the that re-write? Well, note that in `move_ones` you never change the relative order of the elements from $S \cup \overline{S}$. The only possible issue could be if there were an ι or a $\bar{\iota}$ between them. I claim this is impossible: since any rewrite pair is of the form $\overline{\sigma_i}\sigma_j$, any ι or $\bar{\iota}$ between them would have been moved away by the `move_ones` procedure.

So, the algorithm is well-defined and terminating. Is it correct? Indeed it is! At the last step, the word is the same as the last step in R_{rev}, just with some ιs and $\bar{\iota}s$ floating around that will be deleted when we go back to regular words, since they represent the empty string. So the theorem is proved. □

6.4 Partial Grids

A partial grid is an inductively defined structure which generalizes the notion of a grid to include "unfinished" grids, which require more cells to be added in the bottom/right area. Let's use the following as a motivating example:



We keep track of the following data from a partial grid : the spine (made of the left side and top side), and the frontier, divided into three parts: the bottom frontier (which is the section that runs across), the middle frontier (the zigzag-like portion in the middle), and the right frontier (the last part of the frontier which points upwards). Let's see the definition! We define partial grids inductively as

a cell $\begin{array}{ccc} & \xrightarrow{b} & \\ a\downarrow & & \downarrow c \\ & \xrightarrow[d]{} & \end{array}$ (where $\begin{array}{ccc} & \xrightarrow{b} & \\ a\downarrow & & \downarrow c \\ & \xrightarrow[d]{} & \end{array}$ is a grid cell)

with total frontier: $\begin{array}{cc} & \downarrow c \\ \xrightarrow[d]{} & \end{array}$

bottom frontier : $\xrightarrow[d]{}$

middle frontier : empty

right frontier : $\downarrow c$

a skeleton (of words a and b, each of length at least 1) $\begin{array}{cc} & \xrightarrow{b} \\ a\downarrow & \end{array}$

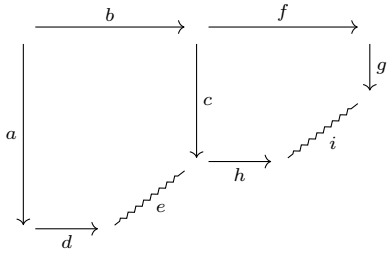
with total frontier: $\begin{array}{cc} & \xrightarrow{b} \\ a\downarrow & \end{array}$

bottom frontier : empty

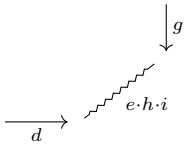
middle frontier : $\begin{array}{cc} & \xrightarrow{b} \\ a\downarrow & \end{array}$

right frontier : empty

horizontal concatenation to a partial grid with non-empty middle frontier (e is non-empty):



with total frontier :

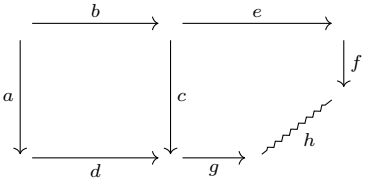


bottom frontier :

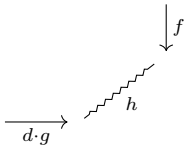
middle frontier :

right frontier :

horizontal concatenation to a partial grid with empty middle frontier :



Which has total frontier :



bottom frontier :

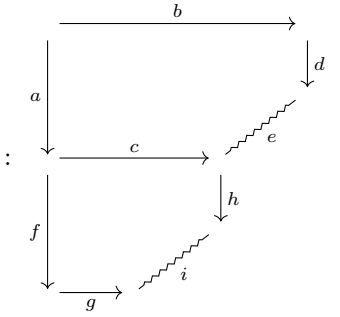
middle frontier :

right frontier :

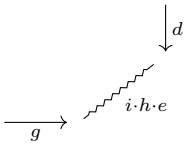
Note that while the left-hand sub-partial-grid looks suspiciously like a grid, we have yet to prove this.

vertical concatenation to a partial grid with non-empty middle frontier (e is non-empty)

:



with has total frontier :

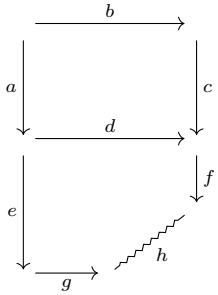


bottom frontier :

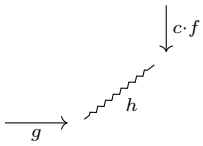
middle frontier :

right frontier :

vertical concatenation to a partial grid with empty middle frontier :



with total frontier :



bottom frontier :

middle frontier :

right frontier :

We end with two short lemmas about partial grids, both shown by structural induction:

Lemma 6.4. The middle frontier of any partial grid is either empty, or is of the form $\overline{\sigma_i} m' \sigma_j$ for some $i, j \in \mathbb{N}$ and some grid word m.

Lemma 6.5. If the frontier of a partial grid is of the form $a\overline{b}$ with a and b both positive grid words, the grid has empty middle frontier.

6.5 Building the Partial Grid

Next, let's define an `add_cell` function on partial grids. When given a partial grid Γ with a subset of its frontier corresponding to the first element of a relation in R_{rev}, so of the form exf, where there exists some y such that $x\ R_{rev}\ y$, `add_cell` outputs a partial grid Γ' with frontier eyf. We define `add_cell` function recursively.

If Γ is a cell, we have a contradiction : its frontier is a positive word followed by a negative word, so no element of its frontier can match the first element of an R_{grid} re-write.

If Γ is a skeleton (a, b), then let $a = \sigma_i a'$ and $b = \sigma_j b'$ (note that the labels on a skeleton must each have length one for the definition of a partial grid). Then its frontier is $\overline{a'}\overline{\sigma_i}\sigma_j b'$. There is exactly one positive-negative pair right at the middle, $\overline{\sigma_i}\sigma_j$. Thus, we look at R_{rev} to determine the g and h such that $R_{grid}(\overline{\sigma_i}\sigma_j, g\overline{h})$. This determines the cell $\begin{array}{ccc} & \xrightarrow{\sigma_j} & \\ {\scriptstyle\sigma_i}\downarrow & & \downarrow{\scriptstyle h} \\ & \xrightarrow[g]{} & \end{array}$.

Notably, we must have $e = a', f = b', x = \overline{\sigma_i}\sigma_j, y = g\overline{h}$. So we are looking for a partial grid with spine (a, b) and frontier $\overline{a'}g\overline{h}b'$ Now, if both a' and b' are ϵ, we are done. If not, there are three cases to consider. We spell out the longest of the three, when neither a' nor b' is ε.

From here, we could do first vertical appending to obtain the partial grid

$$\begin{array}{ccc} & \xrightarrow{\sigma_j} & \\ {\scriptstyle\sigma_i}\downarrow & & \downarrow{\scriptstyle h} \\ & \xrightarrow[g]{} & \\ {\scriptstyle a'}\downarrow & & \end{array}$$

Note that the middle frontier is non-empty - it is $a'g\overline{h}$. Thus, we may do horizontal appending with another skeleton to obtain

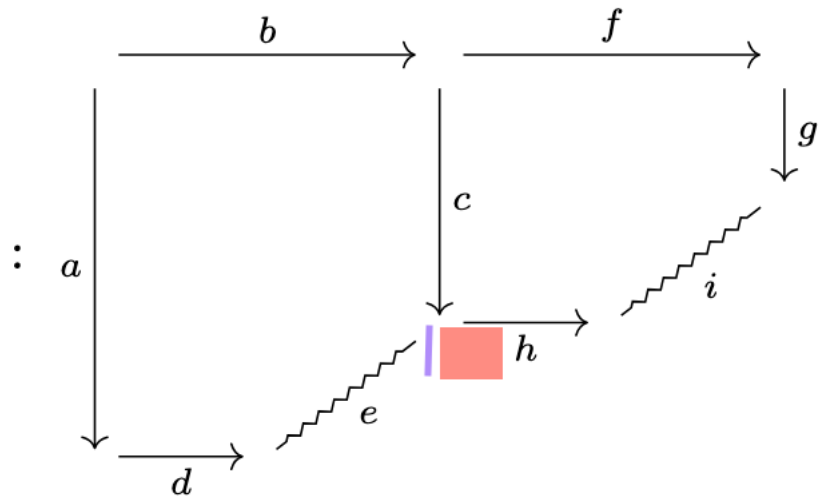
$$\begin{array}{ccccc} & \xrightarrow{\sigma_j} & & \xrightarrow{b'} & \\ {\scriptstyle\sigma_i}\downarrow & & \downarrow{\scriptstyle h} & & \\ & \xrightarrow[g]{} & & & \\ {\scriptstyle a'}\downarrow & & & & \end{array}$$

Thus, we have a partial grid with spine a, b and frontier $\overline{a'}g\overline{h}b'$ - notably, the frontier is what the old frontier re-wrote to!

If Γ was formed by appending a partial grid Γ_0 to a grid, recursively add the new cell into the Γ_0 (we omit details of showing the frontier is correct, as this is straightforward but lengthy).

The remaining two cases are trickier. Luckily, they are analogous, so we need only consider the case of horizontally appending two partial grids Γ_1 and Γ_2, the first with non-empty middle frontier. If the re-write fully occurs in the frontier of either Γ_1 or Γ_2, recursively add it in there, and update the frontiers accordingly. Again, details are omitted.

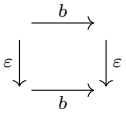
What happens if the re-write spans the space between the two? It might look something like this, where the end of the middle frontier e is the purple arrow, and the cell to be added is the coral rectangle:



The idea is that the re-write would correspond to a cell in the pink space, whose tiny spine would span both frontiers. This would require that the last letter in e be $\overline{\sigma_i}$ for some i. However, by our lemma 6.4, the last letter in any non-empty middle frontier is of the form σ_i. Thus, this case is not possible.

Theorem 6.6. If $\overline{a}b \; S_{R_{grid}} c$, then there is a partial grid with spine (a, b) and frontier c.

Proof. We proceed by induction on the derivation of c from $\overline{a}b$.

If the step was reflexivity, we are quickly done. Since $\overline{a}b$ rewrites to $\overline{a}b$ thus $\overline{a}b$ must also be in positive-negative form. Hence either $a = \varepsilon$ or $b = \varepsilon$. In the case where $a = \varepsilon$, we have proved that there is a grid 

Since any grid is a partial grid with empty frontier, we are done. A similar argument works for $b = \varepsilon$.

For the other case, the transitive one-step closure, we know $\overline{a}b$ rewrites to to exf, $x \; R_{grid} \; y$, and hence $\overline{a}b$ rewrites to to eyf.

Inductively, we know there is a partial grid Γ_1 with spine (a, b) and total frontier exf. We aim to add to this partial grid, obtaining one with spine (a, b) and total frontier eyf. Aha! We may simply call our `add_cell` function, which performs exactly as required. □

6.6 Empty Middle Frontiers

This theorem has been formalized in Lean.

Theorem 6.7. Given a partial grid Γ with spine (a, b), bottom frontier d, middle frontier ε, and right frontier c, I claim that there is a grid from (a, b) to (c, d).

Proof. We proceed by induction on the structure of a partial grid Γ with empty middle frontier.

- If Γ is a cell, we are done as all cells are grids.
- If Γ is a skeleton, its middle frontier is non-empty.
- If Γ is the horizontal concatenation of a partial grid Γ_0 with empty middle frontier and a partial grid Γ_1, since the middle frontier of Γ is defined as the middle frontier of Γ_1, Γ_1 has empty middle frontier. By the induction hypothesis, both Γ_1 and Γ_0 are grids. Therefore, by appending the two grids Γ_0 and Γ_1 we see that Γ is itself a grid.
- If Γ is the horizontal concatenation of two partial grids Γ_1 and Γ_2, where the middle frontier of Γ_1 is non-empty, Γ will have a non-empty middle frontier, so we are done.
- The vertical cases are analogous.

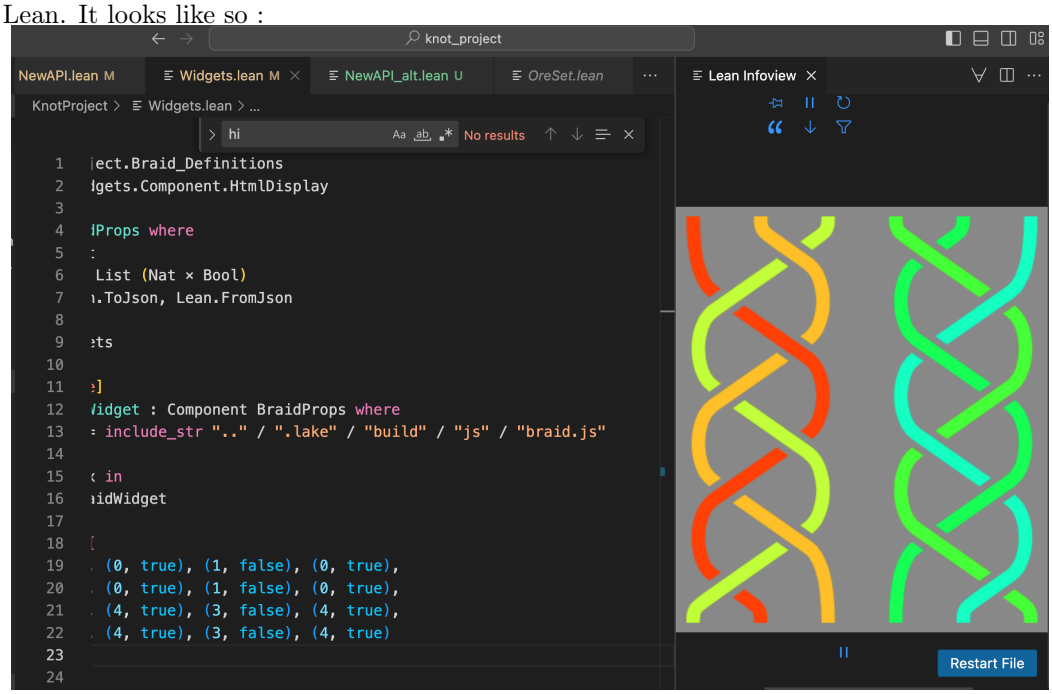
Time to put everything together! Assume $\overline{a}b\ S_{R_{rev}}\ a'\overline{b'}$. Then we know $\overline{a}b\ S_{R_{grid}}\ a''\overline{b''}$ where deleting all occurrence of ι and $\overline{\iota}$ leaves a', and similarly for b'' and b'. Thus, by theorem 6.5, we know there is a there is a partial grid Γ_0 with spine (a, b) and frontier $a'\overline{b'}$. Thus, the middle frontier of Γ_0 is empty by lemma 6.5, so there is a grid $\begin{array}{ccc} & \xrightarrow{b} & \\ a\downarrow & & \downarrow b' \\ & \xrightarrow[a']{} & \end{array}$ as desired. Voilà! □

7 Future Work

I certainly aim to complete the Lean proof of the equivalence between grids and reversing re-writes. That will lead naturally into implementing a verified Lean algorithm to solve the braid isotopy problem. Other close-in-reach formalization goals include defining the center of the braid group, and implementing a (slow!) solution to the braid conjugacy problem. Further-down-the-line goals would be to implement a faster algorithm, Dehornoy's handle-reduction algorithm. This requires giving an order on braids.

Personally, I am intrigued by the project of showing the geometric definition of braids coincides with Artin's algebraic presentation. This would be a larger undertaking! I am also interested in keeping up with the fused braids research (formalizing machine knitting) - perhaps I could be some help in getting a few of the proofs for fused braids.

Stepping a bit outside straight formalization, I also aim to take advantage of Lean's infoview to make a widget which displays braids. So far, I am able to display an image of a braid word typed in Lean. It looks like so :



Thanks to Jim McCann for the visualization code!

The CMU Textiles Lab has also made a visualizer displaying clickable arrows to effect braid moves. I would like to have the user interact with the Lean widget and generate Lean code. For example, a user could be given a braid and prove it equal to the empty braid by physically untangling the strands by clicking and dragging.

I do not expect this functionality to aid professional mathematicians in their research. I see this as a fantastic opportunity for outreach and education. It would be similar to the existing interactive Lean widgets (a rubik's cube, a maze, and a sudoku puzzle.)

I hope to spend more time poring through old Russian records to track down Weinberg, Ivanovsky, and Markov. I plan to beef up the historical section with more context, close readings, and supplementary works by the main authors (i.e. Reidemeister's philosophy). I think it would be nice to write up a history of braids, especially since there is plenty of primary source material that remains unstudied!

References

[1] J. W. Alexander. "A Lemma on Systems of Knotted Curves". In: *Proceedings of the National Academy of Sciences of the United States of America* 9.3 (Mar. 1923), pp. 93–95. ISSN: 0027-8424. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1085274/.

[2] J. W. Alexander. "Topological Invariants of Knots and Links". In: *Transactions of the American Mathematical Society* 30.2 (1928). Publisher: American Mathematical Society, pp. 275–306. ISSN: 0002-9947. DOI: 10.2307/1989123. URL: https://www.jstor.org/stable/1989123.

[3] *Alexandre-Theophile Vandermonde - Biography*. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Vandermonde/.

[4] D. E. Apushkinskaya, A. I. Nazarov, and G. I. Sinkevich. *In Search of Shadows: the First Topological Conference, Moscow 1935*. arXiv:1903.02065. May 2019. URL: http://arxiv.org/abs/1903.02065.

[5] F. Bohnenblust. "The Algebraical Braid Group". In: *The Annals of Mathematics* 48.1 (Jan. 1947), p. 127. ISSN: 0003486X. DOI: 10.2307/1969219. URL: https://www.jstor.org/stable/1969219?origin=crossref.

[6] Werner Burau. "Über Zopfgruppen und gleichsinnig verdrillte Verkettungen". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 11.1 (Dec. 1935), pp. 179–186. ISSN: 1865-8784. DOI: 10.1007/BF02940722. URL: https://doi.org/10.1007/BF02940722.

[7] Muqing Cao et al. *Path Planning for Multiple Tethered Robots Using Topological Braids*. arXiv:2305.00271. June 2023. DOI: 10.48550/arXiv.2305.00271. URL: http://arxiv.org/abs/2305.00271.

[8] Xiaoming Chen et al. *A New Cryptosystem Based on Positive Braids*. arXiv:1910.04346. Oct. 2019. DOI: 10.48550/arXiv.1910.04346. URL: http://arxiv.org/abs/1910.04346.

[9] Xiao-Sheng Cheng and Xian'an Jin. "The Braid Index of Complicated DNA Polyhedral Links". en. In: *PLoS ONE* 7.11 (Nov. 2012). Ed. by Laurent Kreplak, e48968. ISSN: 1932-6203. DOI: 10.1371/journal.pone.0048968. URL: https://dx.plos.org/10.1371/journal.pone.0048968.

[10] A. H. Clifford and G. B. Preston. *Algebraic Theory of Semigroups, Volume 1*. Mathematical Surveys and Monographs v.7. Providence: American Mathematical Society, 2014. ISBN: 978-1-4704-1234-0 978-0-8218-0271-7.

[11] Richard Dedekind. *Was sind und was sollen die Zahlen?* Braunschweig, 1918.

[12] Patrick Dehornoy. *The subword reversing method*. arXiv:0912.4272. Dec. 2009. URL: http://arxiv.org/abs/0912.4272.

[13] Patrick Dehornoy, Danièle Gibbons, and Greg Gibbons. *The Calculus of Braids: an introduction, and beyond*. London Mathematical Society student texts 100. Cambridge ; New York, NY: Cambridge University Press, 2022. ISBN: 978-1-108-84394-2.

[14] Moritz Epple. "Knot Invariants in Vienna and Princeton during the 1920s: Epistemic Configurations of Mathematical Research". In: *Science in Context* 17.1-2 (June 2004), pp. 131–164. ISSN: 0269-8897, 1474-0664. DOI: 10.1017/S0269889704000079. URL: https://www.cambridge.org/core/product/identifier/S0269889704000079/type/journal_article.

[15] Moritz Epple. "Years ago: Orbits of asteroids, a braid, and the first link invariant". In: *The Mathematical Intelligencer* 20.1 (Mar. 1998), pp. 45–52. ISSN: 0343-6993, 1866-7414. DOI: 10.1007/BF03024400. URL: http://link.springer.com/10.1007/BF03024400.

[16] F. A. Garside. "The Braid Group and Other Groups". en. In: *The Quarterly Journal of Mathematics* 20.1 (1969), pp. 235–254. ISSN: 0033-5606, 1464-3847. DOI: 10.1093/qmath/20.1.235. URL: https://academic.oup.com/qjmath/article-lookup/doi/10.1093/qmath/20.1.235.

[17] Carl Friedrich Gauss. *Werke*. Vol. 7. URL: https://gdz.sub.uni-goettingen.de/id/DE-611-HS-3354040?.

[18] Robert Ghrist. "Configuration spaces, braids, and robotics". In: *Braids*. Vol. Volume 19. Lecture Notes Series, Institute for Mathematical Sciences, National University of Singapore Volume 19. World Scientific, Dec. 2009, pp. 263–304. ISBN: 978-981-4291-40-8. DOI: 10.1142/9789814291415_0004. URL: https://www.worldscientific.com/doi/abs/10.1142/9789814291415_0004.

[19] Joel Hass, Jeffrey C. Lagarias, and Nicholas Pippenger. *The Computational Complexity of Knot and Link Problems*. arXiv:math/9807016. July 1998. URL: http://arxiv.org/abs/math/9807016.

[20] Christopher Hollings. *Mathematics across the Iron Curtain: a history of the algebraic theory of semigroups*. History of mathematics volume 41. Providence (R.I.): American mathematical society, 2014. ISBN: 978-1-4704-1493-1.

[21] Christopher D. Jones et al. "Braiding, branching and chiral amplification of nanofibres in supramolecular gels". In: *Nature Chemistry* 11.4 (Apr. 2019). Publisher: Nature Publishing Group, pp. 375–381. ISSN: 1755-4349. DOI: 10.1038/s41557-019-0222-0. URL: https://www.nature.com/articles/s41557-019-0222-0.

[22] Ki Hyoung Ko et al. "New Public-Key Cryptosystem Using Braid Groups". In: *Advances in Cryptology — CRYPTO 2000*. Ed. by Gerhard Goos et al. Vol. 1880. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 166–183. ISBN: 978-3-540-67907-3 978-3-540-44598-2. DOI: 10.1007/3-540-44598-6_10. URL: http://link.springer.com/10.1007/3-540-44598-6_10.

[23] Johann Benedikt Listing. *Vorstudien zur Topologie*. Vandenhoeck und Ruprecht, 1848.

[24] A. V. Malyutin and A. M. Vershik. *Boundary of the braid groups and Markov–Ivanovsky normal form*. arXiv:0707.1109 [math]. Sept. 2008. DOI: 10.48550/arXiv.0707.1109. URL: http://arxiv.org/abs/0707.1109.

[25] A. Markov. "Foundations of the algebraic theory of tresses". In: *Travaux Inst. Math. Stekloff* 16 (1945).

[26] A. Markov. "Über die freie Äquivalenz der geschlossenen Zöpfe". In: *Rec. Math. Moscou* 1 (1936), pp. 73–78.

[27] Oystein Ore. "Linear Equations in Non-Commutative Fields". In: *Annals of Mathematics* 32.3 (July 1931), pp. 463–477. URL: https://www.jstor.org/stable/1968245.

[28] T. V. H. Prathamesh. "Formalising Knot Theory in Isabelle/HOL". In: *Interactive Theorem Proving*. Ed. by Christian Urban and Xingyuan Zhang. Cham: Springer International Publishing, 2015, pp. 438–452. ISBN: 978-3-319-22102-1. DOI: 10.1007/978-3-319-22102-1_29.

[29] Jozef H. Przytycki. *History of Knot Theory*. arXiv:math/0703096. Mar. 2007. DOI: 10.48550/arXiv.math/0703096. URL: http://arxiv.org/abs/math/0703096.

[30] Karl Reidemeister. *Exaktes Denken*. 1928.

[31] Jean-Luc Thiffeault and Marko Budisic. *Braidlab: A Software Package for Braids and Loops*. arXiv:1410.0849 [math]. Nov. 2019. DOI: 10.48550/arXiv.1410.0849. URL: http://arxiv.org/abs/1410.0849.

[32] William Thomson. "On Vortex Atoms". en. In: *Proceedings of the Royal Society of Edinburgh* 6 (1869), pp. 94–105. ISSN: 0370-1646. DOI: 10.1017/S0370164600045430. URL: https://www.cambridge.org/core/product/identifier/S0370164600045430/type/journal_article.

[33] "A History of Topological Knot Theory". In: *History and science of knots*. Ed. by J. C. Turner and Pieter van de Griend. K & E series on knots and everything v. 11. Singapore ; River Edge, N.J: World Scientific, 1996. ISBN: 978-981-02-2469-1.

[34] N Weinberg. "Sur l'équivalence libre des tresses fermées". In: *Comptes Rendus (Doklady) de l'Académie des Sciences de l'URSS* XXIII.3 (1939).

[35] *МАТЕМАТИКА в СССР ЗА ТРИДЦАТЬ ЛЕТ 1917-1947 (Mathematics in the USSR for 30 years 1917-1947)*. ГОСУДАРСТВЕННОЕ ИЗДАТЕЛЬСТВО ТЕХНИКО - ТЕОРЕТИЧЕСКОЙ ЛИТЕРАТУРЫ (State Publishing House of Technical and Thoeretical Literature), 1948.

[36] Осипов, Ю. С., Садовничий, В. А., and Ширяев, Альберт Николаевич, eds. *Kolmogorov and contemporary mathematics: (Moscow, June 16-21, 2003) : in commemoration of the centennial of Andrei Nikolaevich Kolmogorov (25.IV.1903-20.X.1987) : abstracts : international conference*. OCLC: 1053626258. Moscow: Механико-математический факультет МГУ им. М.В. Ломоносова, 2003.

A Code

Below is an outline of the major definitions and theorems developed in this project. Proof of lemmas and most theorems are omitted. The full, up-to-date code repository is available at https://github.com/hannahfechtner/br

A.1 Definition of Artin-Tits Groups

```
variable (M : α → α → ℕ)

def alternate (a b : α) (k : ℕ) :=
  match k with
  | 0 => 1
  | Nat.succ n => FreeGroup.of a * alternate b a n

def artin_tits_rel : α → α → FreeGroup (α) := fun i j => alternate i j (M i j) *
     (alternate j i (M i j))⁻¹

def ArtinTitsGroup := PresentedGroup (artin_tits_rel M)
```

A.2 Definition of Braid Groups

```
def M_braid (i j : ℕ) : ℕ :=
  match j-i with
  | 0 => 0
  | Nat.succ n =>
    match n with
    | 0 => 3
    | Nat.succ _ => 2

def M_braid_fin (n : ℕ) (i j : Fin n) : ℕ := if n = 0 ∨ n = 1 then 1 else
    M_braid i.val j.val

def BraidGroupInf := ArtinTitsGroup M_braid

def BraidGroupFin (n :  ℕ) := ArtinTitsGroup (M_braid_fin n)

def σi (k : ℕ) : BraidGroupInf := PresentedGroup.of k

def σ {n : ℕ} (k : Fin n) : BraidGroupFin (n + 1) := PresentedGroup.of k
```

A.3 Presented Monoids

```
variable {α : Type*}

/-- Given a set of relations, `rels`, over a type `α`, `PresentedMonoid` constructs
    the monoid with generators `x : α` and relations `rels` as a quotient of a
    congruence structure over rels. -/
def PresentedMonoid (rel : FreeMonoid α → FreeMonoid α → Prop) :=
  (conGen rel).Quotient

def PresentedMonoid.rel (rel : FreeMonoid α → FreeMonoid α → Prop) :=
    ConGen.Rel rel

instance {rels : FreeMonoid α → FreeMonoid α → Prop} : Monoid (PresentedMonoid
    rels) := Con.monoid (conGen rels)

/-- The quotient map from the free monoid on `α` to the presented monoid with the
    same generators and the given relations `rels`. -/
def mk (rels : FreeMonoid α → FreeMonoid α → Prop) : FreeMonoid α →*
    PresentedMonoid rels where
  toFun := Quotient.mk (conGen rels).toSetoid
  map_one' := rfl
  map_mul' := fun _ _ => rfl

/-- `of` is the canonical map from `α` to a presented monoid with generators `x : α`
    . The term `x` is mapped to the equivalence class of the image of `x` in `
    FreeMonoid α`. -/
def of (rels : FreeMonoid α → FreeMonoid α → Prop) (x : α) : PresentedMonoid
    rels := mk rels (.of x)

section inductionOn

variable {α₁ α₂ α₃ : Type*} {rels₁ : FreeMonoid α₁ → FreeMonoid α₁ → Prop}
  {rels₂ : FreeMonoid α₂ → FreeMonoid α₂ → Prop} {rels₃ : FreeMonoid α₃ →
    FreeMonoid α₃ → Prop}

local notation "P₁" => PresentedMonoid rels₁
local notation "P₂" => PresentedMonoid rels₂
local notation "P₃" => PresentedMonoid rels₃

@[elab_as_elim, induction_eliminator]
protected theorem inductionOn {δ : P₁ → Prop} (q : P₁) (h : ∀ a, δ (mk rels₁ a)) :
     δ q := Quotient.ind h q

@[elab_as_elim]
protected theorem inductionOn₂ {δ : P₁ → P₂ → Prop} (q₁ : P₁) (q₂ : P₂)
    (h : ∀ a b, δ (mk rels₁ a) (mk rels₂ b)) : δ q₁ q₂ :=
  Quotient.inductionOn₂ q₁ q₂ h

@[elab_as_elim]
protected theorem inductionOn₃ {δ : P₁ → P₂ → P₃ → Prop} (q₁ : P₁)
    (q₂ : P₂) (q₃ : P₃) (h : ∀ a b c, δ (mk rels₁ a) (mk rels₂ b) (mk rels₃ c)) :
    δ q₁ q₂ q₃ := Quotient.inductionOn₃ q₁ q₂ q₃ h

end inductionOn
```

```
section ToMonoid

variable {α M : Type*} [Monoid M] (f : α → M) {rels : FreeMonoid α → FreeMonoid
    α → Prop} (h : ∀ a b : FreeMonoid α, rels a b → FreeMonoid.lift f a =
    FreeMonoid.lift f b)

/-- The extension of a map `f : α → M` that satisfies the given relations to a
    monoid homomorphism from `PresentedMonoid rels → M`. -/
def lift : PresentedMonoid rels →* M := Con.lift _ (FreeMonoid.lift f)
    (Con.conGen_le h)

theorem toMonoid.unique (g : MonoidHom (conGen rels).Quotient M)
    (hg : ∀ a : α, g (of rels a) = f a) : g = lift f h :=
  Con.lift_unique (Con.conGen_le h) g (FreeMonoid.hom_eq fun x ↦ let_fun this :=
    hg x; this)

end ToMonoid
```

API for rewrite system

```
theorem refl : PresentedMonoid.rel rels a a := ConGen.Rel.refl _
theorem reg_rw (c d) (h : rels a b) : PresentedMonoid.rel rels (c * a * d) (c * b *
     d) :=
  ConGen.Rel.mul (ConGen.Rel.mul (ConGen.Rel.refl _) (ConGen.Rel.of _ _ h))
        (ConGen.Rel.refl _)
theorem symm_rw (c d) (h : rels a b) : PresentedMonoid.rel rels (c * b * d) (c *
    a * d) :=
  ConGen.Rel.mul (ConGen.Rel.mul (ConGen.Rel.refl _)
        (ConGen.Rel.symm (ConGen.Rel.of _ _ h))) (ConGen.Rel.refl _)

theorem mul (h1 : PresentedMonoid.rel rels a b) (PresentedMonoid.rel rels c d) :
  PresentedMonoid.rel rels (a * c) (b * d) := ConGen.Rel.mul h1 h2
theorem mul_left (h1 : rels a b) (h2 : rels c d) :
  PresentedMonoid.rel rels (a * c) (b * d) := ConGen.Rel.mul (ConGen.Rel.of _ _
    h1) h2
theorem mul_right (h1 : PresentedMonoid.rel rels a b) (h2 : rels c d) :
  PresentedMonoid.rel rels (a * c) (b * d) := ConGen.Rel.mul h1 (ConGen.Rel.of _
    _ h2)

theorem append_left (h : PresentedMonoid.rel rels c d) :
  PresentedMonoid.rel rels (a * c) (a * d) := ConGen.Rel.mul refl h
theorem append_right (h : PresentedMonoid.rel rels a b) :
  PresentedMonoid.rel rels (a * c) (b * c) := ConGen.Rel.mul h refl

theorem rel_left (h : rels c d) : PresentedMonoid.rel rels (a * c) (a * d) :=
  ConGen.Rel.mul refl (ConGen.Rel.of _ _ h)
theorem rel_right (h : rels a b) : PresentedMonoid.rel rels (a * c) (b * c) :=
  ConGen.Rel.mul (ConGen.Rel.of _ _ h) refl
theorem rel (h : rels a b) : PresentedMonoid.rel rels a b := ConGen.Rel.of _ _ h

theorem symm (h : rels a b) : PresentedMonoid.rel rels b a :=
  ConGen.Rel.symm (ConGen.Rel.of _ _ h)
theorem swap (h : PresentedMonoid.rel rels a b) PresentedMonoid.rel rels b a :=
    ConGen.Rel.symm h
```

Equivalent version

```
inductive rw_system (rels : FreeMonoid α → FreeMonoid α → Prop) : FreeMonoid α
    → FreeMonoid α → Prop
  | refl : rw_system rels a a
  | reg : ∀ c d, rels a b → rw_system rels (c * a * d) (c * b * d)
  | symm : ∀ c d, rels a b → rw_system rels (c * b * d) (c * a * d)
  | trans : rw_system rels a b → rw_system rels b c → rw_system rels a c

private theorem rw_system_equiv_presented_monoid (rels : FreeMonoid α →
    FreeMonoid α → Prop) : rw_system rels a b ↔ rel rels a b

theorem rel_induction_rw {P : FreeMonoid α → FreeMonoid α → Prop} {a b :
    FreeMonoid α}
    (h : rel rels a b)
    (h1 : ∀ (a : FreeMonoid α), P a a)
    (h2 : ∀ a b {c d}, rels a b → P (c * a * d) (c * b * d))
    (h3 : ∀ a b {c d}, rels b a → P (c * a * d) (c * b * d))
    (h4 : ∀ a b c, P a b ∧ P b c → P a c) : P a b
```

Universal Property

```
variable {α M : Type*} [Monoid M] (f : α → M)
variable {rels : FreeMonoid α → FreeMonoid α → Prop}
variable (h : ∀ a b : FreeMonoid α, rels a b →  FreeMonoid.lift f a =
    FreeMonoid.lift f b)

/-- The extension of a map `f : α → M` that satisfies the given relations to a
    monoid homomorphism from `PresentedMonoid rels → M`. -/
def toMonoid : MonoidHom (PresentedMonoid rels) M :=
  Con'.lift _ (FreeMonoid.lift f) (Con.conGen_le h)

theorem toMonoid.unique (g : MonoidHom (conGen rels).Quotient M)
    (hg : ∀ a : α, g (of rels a) = f a) : g = toMonoid f h :=
  Con'.lift_unique (proof_1 f h) g (FreeMonoid.hom_eq fun x ↦ let_fun this := hg
    x; this)
```

Support for presented monoids over isomorphic types

```
variable {β : Type*} (e : α ≃ β) (rels : FreeMonoid α → FreeMonoid α → Prop)

/-- presented monoids over isomorphic types (with the relations converted
    appropriately) are isomorpic -/
noncomputable def equivPresentedMonoid (rel : FreeMonoid β → FreeMonoid β →
    Prop) :
    PresentedMonoid rel ≃* PresentedMonoid (FreeMonoid.comap_rel e rel) :=
  (Con.comapQuotientEquivOfSurj _ _ (FreeMonoid.congr_iso
    e).surjective).symm.trans <|
  Con.congr (Con.comap_conGen_of_Bijective (FreeMonoid.congr_iso e)
    (MulEquiv.bijective _) _ rel)
```

A.4 Braid Monoid

```
inductive braid_rels_multi {n : ℕ} : FreeMonoid (Fin (n + 2)) → FreeMonoid (Fin
    (n + 2)) → Prop
  | adjacent (i : Fin (n + 1)) : braid_rels_multi (of i.castSucc * of i.succ * of
    i.castSucc) (of i.succ * of i.castSucc * of i.succ)
  | separated (i j : Fin n) (h : i ≤ j) : braid_rels_multi (of
    i.castSucc.castSucc * of j.succ.succ) (of j.succ.succ * of i.castSucc.castSucc)

def braid_rels_m : (n : ℕ) → (FreeMonoid (Fin n) → FreeMonoid (Fin n) → Prop)
  | 0     => (λ _ _ => False)
  | 1     => (λ _ _ => False)
  | n + 2 => @braid_rels_multi n

def BraidMonoid (n : ℕ) := PresentedMonoid (braid_rels_m n.pred)

inductive braid_rels_m_inf : FreeMonoid ℕ → FreeMonoid ℕ → Prop
  | adjacent (i : ℕ): braid_rels_m_inf (of i * of (i+1) * of i) (of (i+1) * of i *
     of (i+1))
  | separated (i j : ℕ) (h : i +2 ≤ j) : braid_rels_m_inf (of i * of j) (of j *
    of i)

def BraidMonoidInf := PresentedMonoid braid_rels_m_inf

namespace BraidMonoidInf

def rel := PresentedMonoid.rel braid_rels_m_inf

protected def mk := PresentedMonoid.mk (braid_rels_m_inf)

@[induction_eliminator]
protected theorem inductionOn {δ : BraidMonoidInf → Prop} (q : BraidMonoidInf)
    (h : ∀ a, δ (BraidMonoidInf.mk a)) : δ q :=
  Quotient.ind h q

/--length of an element of the braid monoid -/
def length : BraidMonoidInf → ℕ :=
  PresentedMonoid.lift_of_mul (FreeMonoid.length)
  (fun h1 h2 => by rw [length_mul, length_mul, h1, h2]) (fun _ _ h => by
  induction h with
  | adjacent i => simp only [length_mul, length_of, Nat.reduceAdd]
  | separated i j _ => simp only [length_mul, length_of, Nat.reduceAdd])

/-- the set of generators appearing in a braid word -/
def generators : BraidMonoidInf → Finset ℕ :=
  PresentedMonoid.lift_of_mul (FreeMonoid.symbols)
  (fun ih1 ih2 => by rw [symbols_mul, symbols_mul, ih1, ih2])
  (fun a b h => by induction h with
  | adjacent i =>
    ext x
    simp only [symbols_mul, symbols_of, Finset.union_assoc, Finset.mem_union,
    Finset.mem_singleton]
    tauto
  | separated i j h =>
    simp only [symbols_mul, symbols_of]
    exact Finset.union_comm _ _)
```

```
private theorem reverse_helper (a b : FreeMonoid ℕ) (h : braid_rels_m_inf a b) :
    mk braid_rels_m_inf a.reverse = mk braid_rels_m_inf b.reverse

/-- reverses the braid monoid code -/
def reverse : BraidMonoidInf → BraidMonoidInf :=
  PresentedMonoid.lift_of_mul (fun x => mk braid_rels_m_inf <| FreeMonoid.reverse
    x)
  (fun h1 h2 => by simp [reverse_mul, mul_mk, h1, h2]) reverse_helper

end BraidMonoidInf
```

Injection into the braid group

```
lemma BraidGroupInf.braid (i : ℕ) :
    σi i * σi i.succ * σi i = σi i.succ * σi i * σi i.succ

lemma BraidGroupInf.comm {i j : ℕ} (h : i.dist j > 1) :
    σi i * σi j = σi j * σi i

theorem embed_inf_helper (a b : FreeMonoid ℕ) (h : braid_rels_m_inf a b) :
    (FreeMonoid.lift fun a => σi a) a = (FreeMonoid.lift fun a => σi a) b :=
  braid_rels_m_inf.casesOn h BraidGroupInf.braid (fun _ _ d => BraidGroupInf.comm
    d)

def embed_inf : BraidMonoidInf →* BraidGroupInf :=
  PresentedMonoid.toMonoid (fun a => σi a) embed_inf_helper
```

A.5 Ore Localization

Here we give the constructive version, which may not make it into Mathlib.
We begin with the case of the Ore localization of a monoid by itself

```
class CommonLeftMultipleMonoid (M : Type*) extends Monoid M where
  cl₁ : M → M → M
  cl₂ : M → M → M
  cl_spec : ∀ a b : M, cl₂ a b * a = cl₁ a b * b

class OreMonoid (M : Type*) extends CommonLeftMultipleMonoid M, CancelMonoid M

open OreMonoid
variable {M : Type*} [OreMonoid M]

instance : OreLocalization.OreSet (⊤ : Submonoid M) where
  ore_right_cancel  := by
    intro r1 r2 s eq
    use 1
    simp only [OneMemClass.coe_one, one_mul]
    exact mul_right_cancel eq
  oreNum r s := CommonLeftMultipleMonoid.cl₁ r s
  oreDenom r s := ⟨CommonLeftMultipleMonoid.cl₂ r s, trivial⟩
  ore_eq := fun r s => CommonLeftMultipleMonoid.cl_spec _ _

local notation "OreLocalizationSelf" => @OreLocalization M _ (⊤ : Submonoid M) _
    M _

/-- when localizing by the entire monoid, the result is a group -/
instance : Group (OreLocalizationSelf) where
  inv := OreLocalization.liftExpand (fun a b => b.val / ⟨a, trivial⟩)
    fun a b c d => by
      apply OreLocalization.oreDiv_eq_iff.mpr
      use 1, b
      simp
  mul_left_inv := OreLocalization.ind fun _ _ => OreLocalization.mul_inv _ _

/-- simplified universal property when localizing by the entire monoid -/
def fraction_group_to_group {G₁ : Type} [Group G₁] (f : M →* G₁) :
    OreLocalizationSelf →* G₁ :=
  OreLocalization.universalMulHom f
  ⟨⟨(fun (x : ((⊤ : Submonoid M))) => toUnits (f x.val)),
  by simp only [OneMemClass.coe_one, map_one]⟩, by simp only
  [Submonoid.coe_mul, map_mul, Subtype.forall, implies_true, forall_const]⟩
  (by intro s ; simp)

/-- uniqueness of the simplified universal property when localizing by the entire
    monoid -/
theorem fraction_group_to_group_unique {G₁ : Type} [Group G₁] (f : M →* G₁)
    (φ : OreLocalizationSelf →* G₁)
    (h : ∀ (r : M), (φ ∘ OreLocalization.numeratorHom) r = f r)
    : φ = fraction_group_to_group f :=
  OreLocalization.universalMulHom_unique f _ _ _ h
```

Next we consider the case of a presented monoid being localized

```
variable {α : Type} [Monoid α] {rels : FreeMonoid α → FreeMonoid α → Prop}

def pm_rels_to_pg_rels (rels : FreeMonoid α → FreeMonoid α → Prop) : Set
    (FreeGroup α) :=
  {FreeMonoid.lift (FreeGroup.of) x.1 * (FreeMonoid.lift (FreeGroup.of) x.2)⁻¹ |
  x ∈ setOf (fun (a : FreeMonoid α × FreeMonoid α) => rels a.1 a.2)}

theorem rels_pg_iff_rels_pml {G₁ : Type} [Group G₁]
    {rels : FreeMonoid' α → FreeMonoid' α → Prop}
    (f : α → G₁) :
    (∀ r ∈ (pm_rels_to_pg_rels rels), ((FreeGroup.lift f) r ) = 1) ↔ (∀ r₁ r₂,
    rels r₁ r₂ →
    (FreeMonoid'.lift f r₁ = FreeMonoid'.lift f r₂))
```

For readability, I have used `pml` to be the presented monoid localization of the presented monoid with relations `rels` by the entire presented monoid.

```
/-- the universal property for the Ore localization of a presented monoid by
    itself -/
def presented_fraction_group_to_group {G₁ : Type} [Group G₁] (f : α → G₁)
    (universal_h : ∀ r₁ r₂, rels r₁ r₂ → (FreeMonoid'.lift f r₁ = FreeMonoid'.lift
    f r₂)) : pml →* G₁ :=
  @OreLocalization.universalMulHom ((PresentedMonoid rels)) _
    (⊤ : Submonoid (PresentedMonoid rels)) _ G₁ _
  ⟨⟨PresentedMonoid.lift_hom f (lift_eq_lift_of_rel f universal_h), rfl⟩,
  by simp only [map_mul, implies_true]⟩ (map_denom_into_units f universal_h) (fun
    _ => rfl)

theorem presented_fraction_group_to_group_unique {G₁ : Type} [Group G₁] (f : α →
    G₁) (universal_h : ∀ r₁ r₂, rels r₁ r₂ → (FreeMonoid'.lift f r₁ =
    FreeMonoid'.lift f r₂)) (φ : pml →* G₁) : (∀ (r : α), φ
    (@OreLocalization.numeratorHom _ _ _ _
    (PresentedMonoid.of rels r)) = f r) → φ =
    presented_fraction_group_to_group _ _ _ f universal_h

def pml_to_presented_group : pml  →* PresentedGroup (pm_rels_to_pg_rels rels)

def presented_group_to_pml : PresentedGroup (pm_rels_to_pg_rels rels) →* pml

theorem comp_pg_pml_pml_pg_eq_id : MonoidHom.comp presented_group_to_pml
    pml_to_presented_group = ⟨⟨id, rfl⟩, fun _ _ => rfl⟩

theorem comp_pml_pg_pg_pml_eq_id : MonoidHom.comp pml_to_presented_group
    presented_group_to_pml = ⟨⟨id, rfl⟩, fun _ _ => rfl⟩

--will go in the monoid hom file - it is true more generally, but it's convenient
    to have the MulHom form here
theorem comp_eq_of_hom_comp_eq {α β γ : Type*} [Monoid α] [Monoid β] [Monoid γ]
    {ab : MonoidHom α β}
    {bc : MonoidHom β γ} {ac : MonoidHom α γ} (h : MonoidHom.comp bc ab = ac) :
    bc.toFun ∘ ab.toFun = ac.toFun:=
  funext fun x ↦ ((congrArg (fun y ↦ (bc ∘ ab) x = y x) h.symm)).mpr rfl

/-- the localization of a presented monoid is isomorphic to the presented group
    over the same relations-/
```

```
def presentedMonoidLocalizationEquivPresentedGroup : pml ≃* PresentedGroup
    (pm_rels_to_pg_rels rels) :=
  ⟨⟨pml_to_presented_group, presented_group_to_pml,
    Function.leftInverse_iff_comp.mpr <| comp_eq_of_hom_comp_eq
    comp_pg_pml_pml_pg_eq_id, Function.rightInverse_iff_comp.mpr <|
    comp_eq_of_hom_comp_eq comp_pml_pg_pg_pml_eq_id⟩, map_mul
    pml_to_presented_group⟩
```

A.6 Common Multiples

A.6.1 $\overline{\sigma}$ Braids

```
local instance : Coe ℕ (FreeMonoid ℕ) :=
  ⟨of⟩

private def count_up_helper : ℕ → ℕ → ℕ → FreeMonoid ℕ
| 0, _, _ => 1
| n+1, i, j => (of i) * (count_up_helper n (i+1) j)

/-- `count_up i j` returns a FreeMonoid element corresponding to a list of
    consecutive ascending integers beginning at i (inclusive) up to j (exclusive)
    -/
def count_up (i j : ℕ) : FreeMonoid ℕ := count_up_helper (j - i) i j

private def count_down_helper : ℕ → ℕ → ℕ → FreeMonoid ℕ
| 0, _, _ => 1
| n+1, i, j => (count_down_helper n i (j+1)) * (j)

/-- `count_down i j` returns a FreeMonoid element corresponding to a list of
    consecutive descending integers beginning at i-1 down to j (inclusive) -/
def count_down (i j : ℕ) : FreeMonoid ℕ := count_down_helper (i - j) i j

/-- A FreeMonoid element corresponding to a list of consecutive numbers between i
    and i, including the smaller of i and j and excluding the larger of i and j -/
def sigma_bar (i j : ℕ) : FreeMonoid (ℕ) :=
  if i = j then 1 else if i < j then count_up i j else count_down i j

theorem prepend_k (i j k : ℕ) (h1: i + 2 ≤ j) (h2 : i < k ∧ k < j) :
    BraidMonoidInf.mk (of k * (sigma_bar i j)) = BraidMonoidInf.mk ((sigma_bar i
    j) * (of (k-1)))

theorem append_k (i j k : ℕ) (h1: i + 2 ≤ j) (h2 : i<k∧k<j) : BraidMonoidInf.mk
    (of (k-1) * (sigma_bar j i)) = BraidMonoidInf.mk ((sigma_bar j i) * of k)
```

A.6.2 $\underline{\Delta}$ Braids

```
def delta_bar : ℕ → FreeMonoid ℕ
  | 0 => 1
  | n+1 => (sigma_bar 0 (n+1)) * (delta_bar n)

theorem delta_bar_bounded (n : ℕ) : ∀ k ∈ delta_bar n, k < n

theorem factor_delta (n : ℕ) (h : 1 ≤ n) : BraidMonoidInf.mk (delta_bar n) =
    BraidMonoidInf.mk ((delta_bar (n-1))*(sigma_bar n 0))

theorem swap_sigma_delta (n : ℕ)  : ∀ i : ℕ , (i ≤ n-1) →
    BraidMonoidInf.mk (of i * (delta_bar n)) = BraidMonoidInf.mk (delta_bar n *
    of (n-1-i))

theorem swap_word_delta {n : ℕ} {w : FreeMonoid ℕ} (w_bounded : ∀ x, x ∈ w → x <
     n) :
    BraidMonoidInf.mk (w * delta_bar n) = BraidMonoidInf.mk  (delta_bar n *
    FreeMonoid.map (λ i => (n-1)-i) w)
```

```
def boundary (n i: ℕ) : FreeMonoid ℕ :=
  match n with
  | 0 => 1
  | 1 => 1
  | k+2 => if i = n - 1 then sigma_bar (n-1) 0 * FreeMonoid.map (λ i => i+1)
    (delta_bar (n-1))
          else boundary (k+1) i * sigma_bar n 0

theorem boundary_spec (i n : ℕ) (h_n : n > 0) (h : i ≤ n-1) : BraidMonoidInf.mk
    (delta_bar n) = BraidMonoidInf.mk (of i * boundary n i)

theorem boundary_bounded (i n : ℕ) (h_n : 0 < n) (h : i ≤ n-1) : ∀ x ∈ boundary
    n i, x < n

theorem multiple_delta_bar (u : FreeMonoid ℕ) (l n : ℕ) (h : FreeMonoid.length u
    <= l)
  (bounded : ∀ x, x ∈ u → x < n) : ∃ w, BraidMonoidInf.mk (u * w) =
    BraidMonoidInf.mk ((delta_bar n)^l) ∧ (∀ x, x ∈ w → x < n)

theorem common_right_mul_inf (u v : BraidMonoidInf) : ∃ (u' v' : FreeMonoid ℕ ),
    u * BraidMonoidInf.mk v' = v * BraidMonoidInf.mk u' := by
  induction' u with u
  induction' v with v
  rcases (is_bounded u) with ⟨k1, hk1⟩
  rcases (is_bounded v) with ⟨k2, hk2⟩
  have u_under : ∀ x ∈ u, x < Nat.max k1 k2 :=
    fun x h => Nat.lt_of_lt_of_le (hk1 x h) (Nat.le_max_left k1 k2)
  have v_under : ∀ x ∈ v, x < Nat.max k1 k2 :=
    fun x h => Nat.lt_of_lt_of_le (hk2 x h) (Nat.le_max_right k1 k2)
  have u_length := Nat.le_max_left (FreeMonoid.length u) (FreeMonoid.length v)
  have v_length := Nat.le_max_right (FreeMonoid.length u) (FreeMonoid.length v)
  rcases (multiple_delta_bar u (Nat.max (FreeMonoid.length u) (FreeMonoid.length
    v)) (Nat.max k1 k2)
    u_length u_under) with ⟨v', hv', _⟩
  rcases (multiple_delta_bar v (Nat.max (FreeMonoid.length u) (FreeMonoid.length
    v)) (Nat.max k1 k2)
    v_length v_under) with ⟨u', hu', _⟩
  exact Exists.intro u' (Exists.intro v' (hv'.trans hu'.symm))
```

Note that this has been implemented non-constructively; a constructive version is possible and in the works!

A.7 Cancellativity

A.7.1 Grids

```
/-- a grid modelling re-writes, inductively defined as a basic cells, or vertical
    or horizontal closure under abutting entire sides -/
inductive grid : FreeMonoid ℕ → FreeMonoid ℕ → FreeMonoid ℕ → FreeMonoid ℕ →
    Prop
  | empty : grid 1 1 1 1
  | top_bottom (i : ℕ) : grid 1 (of i) 1 (.of i)
  | sides (i : ℕ) : grid (of i) 1 (of i) 1
  | top_left (i : ℕ) : grid (of i) (of i) 1 1
  | adjacent (i k : ℕ) (h : i.dist k = 1) : grid (of i) (of k) (of i * of k) (of
    k * of i)
  | separated (i j : ℕ) (h : i.dist j > 1) : grid (of i) (of j) (of i) (of j)
  | vertical (h1: grid u v u' v') (h2 : grid a v' c d) : grid (u * a) v (u' * c) d
  | horizontal (h1: grid u v u' v') (h2 : grid u' b c d) : grid u (v * b) c (v' * d)

/-- relating grid equivalence to braid equivalence, one way -/
theorem braid_eq_of_grid (h : grid a b c d) :
    BraidMonoidInf.mk (a * d) = BraidMonoidInf.mk (b * c)
```

Splittability

```
def split_vertically (a b c d : FreeMonoid ℕ) := ∀ b₁ b₂, b = b₁ * b₂ →
  ∃ u d₁ d₂, grid a b₁ u d₁ ∧ grid u b₂ c d₂ ∧ d = d₁ * d₂

theorem splittable_vertically_of_grid {a b c d : FreeMonoid ℕ} (h : grid a b c d)
    :
    split_vertically a b c d

def split_horizontally (a b c d : FreeMonoid ℕ) := ∀ a₁ a₂, a = a₁ * a₂ →
  ∃ u c₁ c₂, grid a₁ b c₁ u ∧ grid a₂ u c₂ d ∧ c = c₁ * c₂

theorem splittable_horizontally_of_grid {a b c d : FreeMonoid ℕ} (h : grid a b c
    d) :
    split_horizontally a b c d
```

A.7.2 Determinative Spines

```
theorem determinative_one_one (h : grid 1 1 c d) : c = 1 ∧ d = 1

theorem determinative_left_one {b c d : FreeMonoid ℕ} (h : grid 1 b c d) :
    c = 1 ∧ d = b

theorem determinative_top_one {a c d : FreeMonoid ℕ} (h : grid a 1 c d) :
    c = a ∧ d = 1

theorem determinative_comm_rel {c d : FreeMonoid ℕ} {i j : ℕ} (h1 : i.dist j >
    1) (h : grid (of i) (of j) c d) : c = of i ∧ d = of j

theorem determinative_braid_rel {c d : FreeMonoid ℕ} {i j : ℕ} (h1 : i.dist j =
    1) (h : grid (of i) (of j) c d) : c = of i * of j ∧ d = of j * of i
```

A.7.3 Stability

```
def stable (u v : FreeMonoid ℕ) := ∀ a b, grid u v a b → ∀ u' v',
  BraidMonoidInf.mk u = BraidMonoidInf.mk u' →
  BraidMonoidInf.mk v = BraidMonoidInf.mk v' → ∃ a' b',
    grid u' v' a' b' ∧ BraidMonoidInf.mk a = BraidMonoidInf.mk a' ∧
    BraidMonoidInf.mk b = BraidMonoidInf.mk b'
```

Lemmas to prove stability of all grids

```
theorem stable_far_apart (i j k : ℕ) (h : Nat.dist j k >= 2) :
 stable (FreeMonoid.of i) (FreeMonoid.of j * FreeMonoid.of k)

theorem stable_close (i j k : ℕ) (h : Nat.dist j k = 1) : stable (FreeMonoid.of
    i) (of j * of k * of j)

theorem stable_swap (u v : FreeMonoid ℕ) : stable u v → stable v u

theorem stable_first_one (v : FreeMonoid ℕ) : stable 1 v

theorem stable_second_one (v : FreeMonoid ℕ) : stable v 1
```

Then we have lemmas for the full proof, each of which requires the full inductive hypothesis:

```
theorem reg_helper (ih : ∀ (u v a b : FreeMonoid ℕ), n ≥ u.length + b.length →
    grid u v a b → ∀ (u' v' : FreeMonoid ℕ), BraidMonoidInf.mk u =
    BraidMonoidInf.mk u' → BraidMonoidInf.mk v = BraidMonoidInf.mk v' → ∃ a' b',
    grid u' v' a' b' ∧ BraidMonoidInf.mk a = BraidMonoidInf.mk a' ∧
    BraidMonoidInf.mk b = BraidMonoidInf.mk b') (br : braid_rels_m_inf f g) (gr :
    grid e (i * f * j) c d) (len : n + 1 ≥ e.length + d.length) : ∃ a' b', grid e
    (i * g * j) a' b' ∧ BraidMonoidInf.mk c = BraidMonoidInf.mk a' ∧
    BraidMonoidInf.mk d = BraidMonoidInf.mk b'

theorem symm_helper (ih : ∀ (u v a b : FreeMonoid ℕ), n ≥ u.length + b.length →
    grid u v a b → ∀ (u' v' : FreeMonoid ℕ), BraidMonoidInf.mk u =
    BraidMonoidInf.mk u' → BraidMonoidInf.mk v = BraidMonoidInf.mk v' → ∃ a' b',
    grid u' v' a' b' ∧ BraidMonoidInf.mk a = BraidMonoidInf.mk a' ∧
    BraidMonoidInf.mk b = BraidMonoidInf.mk b') (br : braid_rels_m_inf f g) (gr :
    grid e (i * g * j) c d) (len : n + 1 ≥ e.length + d.length) : ∃ a' b', grid e
    (i * f * j) a' b' ∧ BraidMonoidInf.mk c = BraidMonoidInf.mk a' ∧
    BraidMonoidInf.mk d = BraidMonoidInf.mk b'

-- a grid is stable when only the second element moves
theorem stable_second (ih : ∀ (u v a b : FreeMonoid ℕ), n ≥ u.length + b.length
    → grid u v a b →
    ∀ (u' v' : FreeMonoid ℕ), BraidMonoidInf.mk u = BraidMonoidInf.mk u' →
    BraidMonoidInf.mk v = BraidMonoidInf.mk v' → ∃ a' b', grid u' v' a' b' ∧
    BraidMonoidInf.mk a = BraidMonoidInf.mk a' ∧ BraidMonoidInf.mk b =
    BraidMonoidInf.mk b')
    (b_is : BraidMonoidInf.mk f = BraidMonoidInf.mk i) :
    ∀ (d : FreeMonoid ℕ), n + 1 ≥ a.length + d.length →
    ∀ (c : FreeMonoid ℕ), grid a f c d → ∃ a' b', grid a i a' b' ∧
    BraidMonoidInf.mk c = BraidMonoidInf.mk a' ∧ BraidMonoidInf.mk d =
    BraidMonoidInf.mk b'
```

```
theorem stability (u v : FreeMonoid ℕ) : stable u v := by
  have H1 : ∀ t u v, ∀ a b, t >= u.length + b.length → grid u v a b → ∀ u' v',
      BraidMonoidInf.mk u = BraidMonoidInf.mk u' →
      BraidMonoidInf.mk v = BraidMonoidInf.mk v' → ∃ a' b',
      grid u' v' a' b' ∧ BraidMonoidInf.mk a = BraidMonoidInf.mk a' ∧
      BraidMonoidInf.mk b = BraidMonoidInf.mk b' := by
    intro t
    induction t with
    | zero =>
      intro u _ _ _ length
      have : u.length = 0 := by linarith [length]
      rw [FreeMonoid.eq_one_of_length_eq_zero this]
      exact stable_first_one _ _
    | succ n ih =>
      intro a b c d e f a₁ b₁ a_is b_is
      revert c; revert d; revert b
      apply PresentedMonoid.rel_induction_rw (PresentedMonoid.exact a_is)
      · exact fun _ b b_is => stable_second ih b_is
      · intro g i e f br b b_is d len c gr
        have easy_len : n + 1 ≥ b.length + c.length := by
          rw [← grid_diag_length_eq gr]
          exact len
        rcases reg_helper ih br (grid_swap gr) easy_len with ⟨a1, b1,
            swapped_grid, da, cb⟩
        apply grid_swap at swapped_grid
        have easy_len2 : n + 1 ≥ (e * i * f).length + a1.length := by
          simp only [length_mul] at len
          simp only [length_mul]
          rw [← BraidMonoidInf.length_eq da, ← BraidMonoidInf.length_eq
            (PresentedMonoid.sound (PresentedMonoid.rel_alone br))]
          assumption
        rcases stable_second ih b_is a1 easy_len2 b1 swapped_grid with ⟨a2, b2,
            second_fact⟩
        use a2, b2
        exact ⟨second_fact.1, ⟨cb.trans second_fact.2.1, da.trans second_fact.2.2⟩⟩
      · intro _ _ g i br b b_is d len c gr
        have easy_len : n + 1 ≥ b.length + c.length := by
          rw [← grid_diag_length_eq gr]
          exact len
        rcases symm_helper ih br (grid_swap gr) easy_len with ⟨a1, b1,
            swapped_grid, da, cb⟩
        apply grid_swap at swapped_grid
        rename_i x x2
        have easy_len2 : n + 1 ≥ (g * x2 * i).length + a1.length := by
          simp only [length_mul] at len
          simp only [length_mul]
          rw [← BraidMonoidInf.length_eq da, BraidMonoidInf.length_eq
            (PresentedMonoid.sound (PresentedMonoid.rel_alone br))]
          assumption
        rcases stable_second ih b_is a1 easy_len2 b1 swapped_grid with ⟨a2, b2,
            second_fact⟩
        use a2, b2
        exact ⟨second_fact.1, ⟨cb.trans second_fact.2.1, da.trans second_fact.2.2⟩⟩
      · intro ha1 hb1 hc1 ih b b_is d len c gr
```

```
      rcases ih.1 b b_is d len c gr with ⟨c₁, d₁, first_fact⟩
      have H_len : n + 1 ≥ hb1.length + d₁.length := by
        have Hb : b₁.length = b.length := (congr_arg BraidMonoidInf.length
          b_is).symm
        have Hc : c₁.length = c.length := (congr_arg BraidMonoidInf.length
          first_fact.2.1).symm
        rw [← grid_diag_length_eq (grid_swap first_fact.1), Hb, Hc,
          ← grid_diag_length_eq gr]
        exact len
      rcases ih.2 b₁ rfl d₁ H_len c₁ first_fact.1 with ⟨c₂, d₂, second_fact⟩
      use c₂, d₂
      exact ⟨second_fact.1, ⟨first_fact.2.1.trans second_fact.2.1,
        first_fact.2.2.trans second_fact.2.2⟩⟩
  exact fun c d => H1 (u.length + d.length) u v c d (Nat.le_refl _)
```

A.7.4 Existence and Uniqueness

```
theorem existence : ∀ a b, ∃ c d, grid a b c d := by
  intro a b
  rcases common_mul a b with ⟨c1, d1, h⟩
  have big_grid : grid (a * c1) (b * d1) 1 1 := by
    apply grid_of_eq
    rw [h]
  rcases splittable_horizontally_of_grid big_grid _ _ rfl with ⟨_, c₁, c₂,
    top_grid, _, side_one⟩
  rw [(FreeMonoid.prod_eq_one side_one.symm).1] at top_grid
  rcases splittable_vertically_of_grid top_grid _ _ rfl with ⟨top_vert, m₁, m₂,
    top_left, _, _⟩
  use top_vert, m₁

theorem unicity (h1 : grid a b c d) : ∀ c' d', grid a b c' d' → c' = c ∧ d' = d
```

A.8 Grids and Rewriting

A.8.1 Partial Grids Def

```
inductive cell : List ℕ → List ℕ → List ℕ → List ℕ → Prop
  | empty : (cell [] [] [] [] : Prop)
  | top_bottom (i : ℕ) : cell [] [i] [] [i]
  | sides (i : ℕ) : cell [i] [] [i] []
  | top_left (i : ℕ) : cell [i] [i] [] []
  | adjacent (i k : ℕ) (h : Nat.dist i k = 1) : cell [i] [k] [i, k] [k, i]
  | separated (i j : ℕ) (h : i +2 ≤ j ∨ j+2 <= i) : cell [i] [j] [i] [j]

lemma grid_from_cell (h : cell a b c d) : grid a b c d

def to_up (a : List ℕ) : List (Option ℕ × Bool) :=
  match a with
  | [] => [(none, false)]
  | _ => List.map (fun x => (some x, false)) a.reverse

def to_over (a : List ℕ) : List (Option ℕ × Bool) :=
  match a with
  | [] => [(none, true)]
  | _ => List.map (fun x => (some x, true)) a

/-- A partial grid generalizes the notion of a grid to include "unfinished"
    grids. -/
inductive PartialGrid : List (Option ℕ × Bool) → List (Option ℕ × Bool) →
  List (Option ℕ × Bool) → List (Option ℕ × Bool) → List (Option ℕ × Bool) →
    Prop
  | single_grid (h : cell a b c d): PartialGrid (to_up a) (to_over b) (to_over d)
    [] (to_up c)
  | empty (a b : List (Option ℕ × Bool)) (ha : a.length > 0) (ha1 : is_false a)
      (hb : b.length > 0) (hb : is_true b) : PartialGrid a b [] (a ++ b) []
  | horizontal_append_one {a b bot up b2 bot2 mid2 up2} (g1 : PartialGrid a b bot
    [] up)
      (g2 : PartialGrid up b2 bot2 mid2 up2) : PartialGrid a (b ++ b2) (bot ++
    bot2) mid2 up2
  | horizontal_append {a b bot mid up b2 bot2 mid2 up2 : List (Option ℕ × Bool)}
      (h : mid.length > 0)
      (g1 : PartialGrid a b bot mid up) (g2 : PartialGrid up b2 bot2 mid2 up2) :
      PartialGrid a (b ++ b2) bot (mid ++ bot2 ++ mid2) up2
  | vertical_append_one (g1 : PartialGrid a b bot [] up) (g2 : PartialGrid a1 bot
    bot2 mid2 up2) :
      PartialGrid (a1 ++ a) b bot2 mid2 (up2 ++ up)
  | vertical_append (g1 : PartialGrid a b bot mid up) (g2 : PartialGrid a1 bot
    bot2 mid2 up2) (h : mid.length > 0) :
      PartialGrid (a1 ++ a) b bot2 (mid2 ++ up2 ++ mid) up

/-- this removes the symbols for empty arrows -/
def remover : (a : List (Option ℕ × Bool)) → List ℕ
  | [] => []
  | (some a, _) :: c => a :: remover c
  | (none, _) :: c => remover c

lemma remover_up : remover (to_up a) = a.reverse
```

```
lemma remover_over : remover (to_over a) = a

lemma grid_option_append_horiz (h1 : grid_option a b c d) (h2 : grid_option c e f
    g) : grid_option a (b ++ e) f (d ++ g)

lemma grid_option_append_vert (h1 : grid_option a b c d) (h2 : grid_option e d f
    g) : grid_option (e ++ a) b (f ++ c) g

/-- this converts back from labelled arrows to labelled sides, and asserts that
    we have a grid -/
def grid_option (a b c d : List (Option ℕ × Bool)) : Prop := grid (remover
    a.reverse) (remover b)
  (remover c.reverse) (remover d)

/-- a partial grid with an empty middle frontier is in fact a grid (lemmas
    referenced are available on github) -/
theorem grid_of_PartialGrid (h : PartialGrid a b d [] c) : grid_option a b c d :=
    by
  generalize he : ([] : List (Option ℕ × Bool)) = e at h
  induction h with
  | single_grid h =>
    unfold grid_option
    simp only [remover_up_rev, remover_over]
    exact grid_from_cell h
  | empty a b =>
    exfalso
    apply congr_arg List.length at he
    rename_i ha hb
    simp [ha, hb] at he
    linarith
  | horizontal_append_one _ _ ih1 ih2 =>
    specialize ih1 rfl
    specialize ih2 he
    exact grid_option_append_horiz ih1 ih2
  | horizontal_append _ _ _ g1_ih g2_ih =>
    simp only [List.append_assoc, List.nil_eq_append, List.append_eq_nil] at he
    specialize g1_ih he.1.symm
    specialize g2_ih he.2.2.symm
    have H := grid_option_append_horiz g1_ih g2_ih
    rw [he.2.1, List.append_nil] at H
    exact H
  | vertical_append_one _ _ ih1 ih2 =>
    specialize ih1 rfl
    specialize ih2 he
    exact grid_option_append_vert ih1 ih2
  | vertical_append _ _ _ g1_ih g2_ih =>
    simp only [List.append_assoc, List.nil_eq_append, List.append_eq_nil] at he
    specialize g1_ih he.2.2.symm
    specialize g2_ih he.1.symm
    have H := grid_option_append_vert g1_ih g2_ih
    rw [he.2.1, List.nil_append] at H
    exact H
```

A.8.2 Grid-style reversing

```
inductive grid_style : List (Option ℕ × Bool) → List (Option ℕ × Bool) → Prop
| basic {n : ℕ} : grid_style [(some n, false), (some n, true)] [(none, true),
    (none, false)]
| over {n : ℕ} : grid_style [(n, false), (none, true)] [(none, true), (n, false)]
| up {n : ℕ} : grid_style [(none, false), (some n, true)] [(n, true), (none,
    false)]
| empty : grid_style [(none, false), (none, true)] [(none, true), (none, false)]
| apart {i j : ℕ} (h : Nat.dist i j > 1) : grid_style [(i, false), (j, true)]
    [(j, true), (i, false)]
| close {i j : ℕ} (h : Nat.dist i j = 1) : grid_style [(i, false), (j, true)]
    [(j, true), (i, true), (j, false), (i, true)]
```

A.8.3 SemiThue

```
inductive SemiThue (rels : List α → List α → Prop) : List α → List α → Prop
| refl (a : List α) : SemiThue rels a a
| reduction {a b c d : List α} (h : rels a b) : SemiThue rels (c++a++d) (c++b++d)
| trans (a b c : List α) : SemiThue rels a b → SemiThue rels b c → SemiThue
    rels a c

inductive SemiThue_one_step (rels : List α → List α → Prop) : List α → List α
    → Prop
| refl (a : List α) : SemiThue_one_step rels a a
| one_step {a b c d e : List α} (h1 : SemiThue_one_step rels e (c++a++d)) (h2 :
    rels a b) :
    SemiThue_one_step rels e (c++b++d)

theorem one_step_equiv_reg {a b : List α} : SemiThue rels a b ↔
    SemiThue_one_step rels a b
```

A.8.4 Shortlex

```
def Shortlex {α : Type*} (r : α → α → Prop) : List α → List α → Prop :=
  fun a b => Prod.Lex (fun n1 n2 => n1 < n2) (fun a b => List.Lex r a b)
    (a.length, a) (b.length, b)

theorem acc_empty {α : Type*} (r : α → α → Prop) : Acc (Shortlex r) []

theorem acc_singleton {α : Type*} (r : α → α → Prop) {h : WellFounded r} {i : α
    } : Acc (Shortlex r) [i]

theorem acc_pair {α : Type*} (r : α → α → Prop) {h : WellFounded r} (i j : α) :
     Acc (Shortlex r) [i, j]

theorem lexAccessible' {a : α} (n : ℕ) (aca : Acc r a)
    (acb : (b : List α) → b.length < n → Acc (Shortlex r) b) (b : List α) (hb :
    b.length < n)
    (ih : ∀ l : List α, l.length < (a::b).length → Acc (Shortlex r) l) :
    Acc (Shortlex r) ([a] ++ b)

theorem wf {α : Type*} (r : α → α → Prop) {h : WellFounded r} : WellFounded
    (Shortlex r) := by
  apply WellFounded.intro
```

```
  have H : ∀ n, ∀ (a : List α), a.length = n → Acc (Shortlex r) a := by
    intro n
    induction n using Nat.strongInductionOn
    rename_i n ih
    cases n with
    | zero =>
      intro a len_a
      simp only [List.length_eq_zero] at len_a
      rw [len_a]
      exact acc_empty r
    | succ n =>
      intro a
      cases a with
      | nil =>
        intro len_a
        simp only [List.length_nil, self_eq_add_left, add_eq_zero, one_ne_zero,
            and_false]
          at len_a
      | cons head tail =>
        intro len_a
        simp only [List.length_cons, Nat.succ_eq_add_one, add_left_inj] at len_a
        apply lexAccessible' r (n+1)
        · exact WellFounded.apply h head
        · exact fun b bl => ih b.length bl _ rfl
        · rw [len_a]
          exact lt_add_one n
        · intro l ll
          apply ih l.length
          simp only [List.length_cons, Nat.succ_eq_add_one] at ll
          · rw [← len_a]
            exact ll
          rfl
  exact fun a => H a.length _ rfl
```

Showing well-foundedness of our relation on grid words

```
def lt_a : (Option ℕ × Bool) → Option ℕ × Bool →  Prop
  | (_, true), (_, false) => true
  | (none, true), (some _, true) => true
  | (some i, true), (some j, true) => i < j
  | (some i, false), (some j, false) => i < j
  | (some _, false), (none, false) => true
  | (_, _), (_, _) => false

theorem lt_a_acc_none_true : Acc lt_a (none, true)
theorem lt_a_some_zero_true : Acc lt_a (some 0, true)
theorem lt_a_acc_some_true : Acc lt_a (some val, true)
theorem lt_a_acc_some_false : Acc lt_a (some val, false)
theorem lt_acc_none_false : Acc lt_a (none, false)

theorem lt_a_acc : ∀ (a : Option ℕ × Bool), Acc lt_a a

instance : WellFounded lt_a := WellFounded.intro fun a ↦ lt_acc a
```