COMPUTING THE GENERATING FUNCTION OF A COINVARIANTS MAP

BY

JESSE FROHLICH

A thesis submitted in conformity with
the requirements for the degree of

Doctor of Philosophy

Graduate Department of Mathematics
University of Toronto

ABSTRACT

Computing the generating function of a coinvariants map

Jesse Frohlich

Doctor of Philosophy

Graduate Department of Mathematics

University of Toronto

2023

A well-known source of strong link invariants comes from quantum groups. Typically, one uses a representation of a quantum group to build a computable invariant, though these computations require exponential time in the number of crossings. Recent work has allowed for direct and efficient computations within the quantum groups themselves, through the use of perturbed Gaußian differential operators. This thesis introduces and explores a partial expansion of the tangle invariant Z introduced by Bar-Natan and van der Veen [BNvdV]. We expand the use of the quantum group $\mathfrak{U}(\mathfrak{sl}_{2+}^0)$ to include its space of coinvariants, providing an extension Z^{tr} of Z from open tangles to links.

We compute a basis for the space of coinvariants, then compute a closed-form expression for the corresponding trace map in the form of an exponential generating function. The resulting function is not a compatible perturbed Gaußian with respect to the previous research. To respond to this limitation, we find a method of computing the link invariant for a subclass of links and write a program to compute the invariant Z^{tr} on two-component links. Contrary to expectations, we find that Z^{tr} is neither stronger nor weaker than the Multivariable Alexander polynomial. This unexplained behaviour

warrants further study into the invariant Z^{tr} and its relationship with other invariants.

To someone,

who did something nice.

ACKNOWLEDGEMENTS

Thank you to my advisor, Dror Bar-Natan, whose consistent meetings, guidance, and encouragement brought me to this point.

Thank you to my supervisory committee, Joel Kamnitzer and Eckhard Meinrenken, who helped connect me to more of the department and its research.

Thank you to to Jason Siefken and the other teaching mentors in the math department. You have made me a better educator and communicator.

Thank you to the math department, especially Jemima Merisca and Sonja Injac, whose friendly support through the manifold administrative details was invaluable.

Thank you to the people I met in the department, especially those who opened my thesis to confirm they were included here. Thank you specifically.

Thank you to Assaf Bar-Natan, Vincent Girard, Caleb Jonker, and Adriano Pacifico, whose friendship, bike trips, and cooking sessions provided much joy.

Thank you to the care-taking staff at the St. George campus, especially Maria and Maria, whose diligent work and kind words have made this campus beautiful.

For the Graduate Christian Fellowship community at the U of T: thank you for being a refreshing breadth of perspectives for a graduate student.

Thank you to Rosemary and Alan Johnstone, whose hospitality made me feel immediately welcome in a new city.

My family, who raised me and encouraged me in my academic pursuits from the earliest moments.

Thank you to my mom for embracing my atypical learning styles, for instilling my love of reading, and for teaching my how to think deeply.

Thank you to my dad for teaching me to ride a bike, cook, file taxes, and notice the invisible among us.

Thank you to my brother for all the laughter you bring to my life.

Thank you to the Perrins. You are my second family, and your support and love for me is precious.

Thank you to my fiancée, Emily Langridge, who helped me with the motivation to finish. I look forward to the next chapters of my life with you.

ACRONYMS

RVT Rotational Virtual Tangle

RVK Rotational Virtual Knot

RVL Rotational Virtual Link

MVA Multivariable Alexander polynomial

CONTENTS

1 EXECUTIVE SUMMARY

1.1 CONTEXT

Understanding Knotted Objects

In the field of knot theory, distinguishing between two knots or links has proven to be a difficult task. It is a popular endeavour to describe and compute strong invariants of knotted objects.

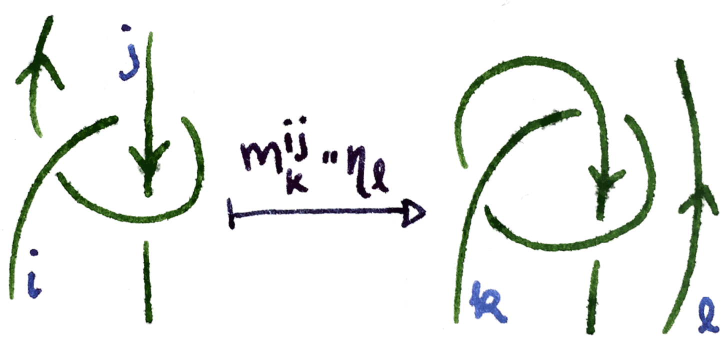

Figure 1.1: Two equivalent knots. Can you see how they are the same?

Merely being able to distinguish between two knotted objects does not always provide us with enough information about these topological structures. For instance, one may ask if a particular link is a satellite of another (roughly: where one knot is embedded into a link by following one of its components), whether a knot is slice (i.e. it is the boundary of a disk in $\mathbb{R}^4$), or whether it is ribbon (i.e. the boundary of a disk in $\mathbb{R}^3$ with restricted types of singularities). Many interesting properties of knots can be phrased in terms of certain topological properties, such as strand doubling (taking a strand and replacing it with two copies of itself, as in figure 1.2) or strand stitching (joining two open components together to form one longer one).

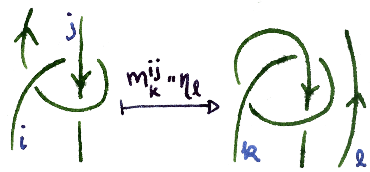

Figure 1.2: An example of strand-doubling.

Open problems such as the Ribbon Slice Conjecture (asking whether there exists a slice knot which is not ribbon) may be advanced by the development of "topologically aware" invariants—those which preserve topological data in a retrievable way.

Quantum Invariants

One such class of topological invariants is derived from quantum groups, which are algebraic structures whose relations mimic those of knotted objects. With this approach, one takes a knotted object and decomposes it into a series of topological operations (such as stitching strands or doubling strands), then maps each of these operations to a corresponding algebraic operation. The composition of these algebraic operations is the value of the invariant.

More specifically, a quantum group (also called a Hopf algebra) is a vector space A together with several maps between various tensor powers (for instance, a multiplication map $m\colon A \otimes A \to A$). The value of the invariant is a vector in some tensor power of A, with one factor assigned to each strand of the knotted object. It is computed by associating to each concatenation of strands with a multiplication of algebra elements. While this formulation is elegant, it has a notable drawback: computing the value of an invariant with many components requires manipulating large tensor powers of A. One remedy is to instead perform the computation in a representation of A, say a small vector space V, though the issue of exponential growth in complexity remains.

Images of the invariant

To avoid the issue of exponential computational complexity, one can instead investigate the nature of the set of all values of the invariant as a subset of the algebra and its tensor powers. For a particular choice of algebra (namely $\hat{\mathfrak{U}}(\mathfrak{sl}^0_{2+})$, as investigated by Dror Bar-Natan and Roland van der Veen in **??**) the space of values the corresponding invariant Z can take is significantly smaller than the whole space; the dimension of the space of values grows only quadratically with the number of crossings in the knotted object. In particular, by looking at the generating functions of the associated maps, instead of a generic power series, the value of Z on tangles always take the form of a (perturbed) Gaußian. Computationally, this means one needs only to keep track of the quadratic form and the perturbation. The invariant Z—which dominates the $\mathfrak{sl}_2$-coloured Jones polynomial—provides an efficient method of computing the Alexander polynomial.

1.2 EXTENDING THE INVARIANT TO MORE KNOTTED OBJECTS

The research program outlined by Bar-Natan and van der Veen computes Z only for (pure) tangles–that is, collections of open strands whose endpoints are fixed to a boundary circle. (Note that this includes long knots, which are exactly the one-component tangles.) This thesis is focused on extending Z and its computations to tangles with closed components.

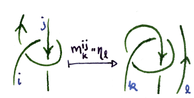

Figure 1.3: A pure tangle.

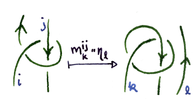

Figure 1.4: A tangle with a closed component.

Computing the extended map

The first task is to determine the space in which the extended invariant, which we will call Z^{tr}, lives. One may observe that in a matrix algebra, one is able to contract two matrices together via matrix multiplication. When one wishes to contract a matrix along itself, one uses the trace map. Analogously, since two strands in a tangle corresponds to multiplication, closing a strand into a loop should correspond algebraically to a trace map.

In a generic algebra A, the trace map is defined as the projection onto the set of coinvariants: $\operatorname{tr}\colon A \to A_A = A/[A, A]$. In order to extend the invariant in this framework, we must first compute the space of coinvariants for $\widehat{\mathfrak{U}}(\mathfrak{sl}_{2+}^0)$, then compute the coinvariants map, and write it as a generating function. (This is accomplished in section 4.2.)

Performing computations

Unfortunately, the resulting trace map does not take the form of a perturbed Gaußian in a way that readily connects to the existing framework. In order to determine whether further study in this direction is merited, we must find an alternative computation method to get a preliminary sense of the strength of Z^{tr}.

For a subclass of links (which includes all two-component links), we compute an explicit closed form for the trace map, then implement a computer program to compute the value of Z^{tr} on all two-component links with up to 11 crossings. When applied to knots, Z computes the Alexander polynomial. When applied to two-component tangles, one may expect that Z^{tr} would produce the natural generalization to multiple components: the Multivariable Alexander polynomial (MVA). Surprisingly, the MVA and Z^{tr} are incomparable, with each being able to distinguish pairs of links the other cannot. (See section 5.1 for more information.)

Further study

As Z^{tr} does not generalize Z in the manner expected, several interesting avenues of further research are opened. Firstly, determining what the relationship between Z^{tr} and the MVA is remains open. Second is the challenge of finding an efficient method for computing Z^{tr} on links with more than two components, which currently is mired in complications with the presence of non-elementary functions where quadratic forms normally appear. Third is the question of the existence of other viable trace candidates. In particular, it may be worth exploring whether a universal trace with respect to the perturbed Gaußian framework defines a sufficiently useful invariant.

2

TENSOR PRODUCTS AND META-OBJECTS

2.1 TENSOR PRODUCT NOTATION

In what follows, we will extensively use tensor products, tensor powers, and generalizations thereof. We begin by introducing the notation that will be used first for traditional tensor products, then for their generalizations.

Let V be a $\mathbb{k}$-vector space (for the moment assumed to be finite dimensional). When working with a large tensor power $V^{\otimes n}$ of V, it will often be more convenient to label tensor factors with elements of a finite set S (with $|S| = n$) rather than by their position in a linear order.

For example, consider the vector $u \otimes v \otimes w \in V^{\otimes 3}$. Let us choose an index set $S = \{1, 2, 3\}$. We then may equivalently write this vector by labelling each tensor factor with one of the elements of S, say $u_1 v_2 w_3$. Since the labels serve to distinguish the separate factors, this vector may equivalently be written as $u_1 v_2 w_3 = v_2 u_1 w_3 = w_3 u_2 u_1 \in V^{\otimes S}$. We will write the set $V^{\otimes S}$ with a subscript: V_S. We formalize the idea below:

Definition 2.1 (indexed tensor powers)**.** Let V be a vector space and $S = \{s_1, \dots, s_n\}$ be a finite set. We define the indexed tensor power of V to be the collection of formal linear combinations of functions from S to V

$$V_S \coloneqq \mathrm{span}\{f\colon S \to V\}/\sim \tag{2.1}$$

subject to the standard multilinear relations, namely multi-additivity and the factoring of scalars.

By multi-additivity, we mean that for each $i \in S$ and $f, g \in V_S$ satisfying $f(i) = g(i) = v$, we have:

$$f + g = \left(s \mapsto \begin{cases} v & \text{if } s = i \\ f(i) + g(i) & \text{otherwise} \end{cases} \right) \tag{2.2}$$

In practice, we will write such functions $f\colon S \to V$ with $f(s_i) = v_i$ in the following way:

$$(v_1)_{s_1}(v_2)_{s_2} \cdots (v_n)_{s_n} \coloneqq f \tag{2.3}$$

With this notation, we may easily express the factoring of scalars as:

$$(v_1)_{s_1}(v_2)_{s_2} \cdots (\lambda v_i)_{s_i} \cdots (v_n)_{s_n} = \lambda \cdot (v_1)_{s_1}(v_2)_{s_2} \cdots (v_n)_{s_n} \tag{2.4}$$

Next, we introduce notation for maps between tensor powers so that we may unambiguously refer to appropriate tensor factors while defining morphisms. Let D and C be finite sets, and $T\colon V_D \to V_C$. We will denote T by T_C^D. It is important to note that when T is not symmetric in its arguments, the order of the indices in this notation matters.

Example 2.2. Let $V = \mathbb{R}^2$, and $T_c^{a,b}$ defined by

$$\begin{aligned} T_c^{a,b}(\vec{v}_a(\vec{e}_1)_b) &= \vec{0}_c \\ T_c^{a,b}(\vec{v}_a(\vec{e}_2)_b) &= \vec{v}_c \end{aligned} \tag{2.5}$$

This function zeros out vectors whose b-component is $\vec{e}_1$. If we wish to define an analogous function for the a-component, we may simply reverse the order of the superscript: $T_c^{b,a}$, which sends $\vec{v}_b(\vec{e}_1)_a = (\vec{e}_1)_a\vec{v}_b$ to $\vec{0}_c$ and $\vec{v}_b(\vec{e}_2)_a$ to $\vec{v}_c$.

Finally, we point out that any morphism T_C^D may be extended to one with larger domain and codomain. We introduce the notation $T_C^D[S] \coloneqq T_C^D \otimes \mathrm{id}_S^S$ for this concept, though will also overload the notation T_C^D for the same map: for any $v_D \in V_D$ and $w_S \in V_S$, we may also write $(T_C^D)(v_D \otimes w_S) \coloneqq (T_C^D v_D) \otimes w_S$.

Remark 2.3. There are three special cases with this notation:

- Given a (multi)linear functional $\phi\colon V_S \to \Bbbk \cong V_\emptyset$, we will write ϕ^S instead of $\phi_\emptyset^S$. The linear order on S remains in this notation.

- Elements $v \in V_S$ will be interpreted as a map $v\colon \Bbbk = V_\emptyset \to V_S$ written v_S instead of $v_S^\emptyset$.
- When only one index is present in a subscript or superscript, and its omission does not introduce an ambiguity in an expression, then it may be omitted to improve readability. For instance, a map $\phi\colon V_{\{1,2\}} \to V_{\{3\}}$ may be written as $\phi^{1,2}$ instead of $\phi_3^{1,2}$, with the canonical isomorphism $V \cong V_{\{3\}}$ being suppressed.

When taking the tensor product of two such tensor powers, we follow [BNS] and use the notation "$\sqcup$" instead of "$\otimes$":

$$V_X \sqcup V_Y := V_{X \sqcup Y} \tag{2.6}$$

Additionally, given $\phi_{C_1}^{D_1}$ and $\psi_{C_2}^{D_2}$ such that $D_1 \cap D_2 = \emptyset = C_1 \cap C_2$, we have a product morphism $\phi_{C_1}^{D_1}\psi_{C_2}^{D_2} := \phi \otimes \psi\colon V_{D_1 \sqcup D_2} \to V_{C_1 \sqcup C_2}$, which we also write with concatenation.

2.2 META-OBJECTS

Notation extension beyond vector spaces

While the above notation is helpful when working with vector spaces, we are interested in also using the same notation to describe a tangle. Our formulation of tangles (introduced in section 2.4) is neither a tensor product nor a monoidal category, though it shares many similarities with both concepts. In particular, the domains and codomains of the maps we have discussed so far have only depended on the index set. With this observation, we replace the notation of tensor powers with that of a so-called meta-object:

Definition 2.4 (Meta-object)**.** Let $\mathcal{C}$ be a category. A meta-object in $\mathcal{C}$ is subcategory with objects indexed by the functor

$$\begin{aligned} A\colon \mathbf{FinSet} &\to \mathrm{Ob}(\mathcal{C}) \\ S &\mapsto A_S \end{aligned} \tag{2.7}$$

The homsets of this subcategory are indexed by pairs of finite sets D, C. Morphisms in these homsets will be denoted by $\phi_C^D\colon A_D \to A_C$. For each such morphism ϕ_C^D, there is a finite-set-indexed morphism $\phi_C^D[\cdot]\colon$ **FinSet** $\to$ $\mathrm{Hom}(\mathcal{C})$ such that

1. $\phi[S]\colon A_{C\sqcup S} \to A_{D\sqcup S}$
2. $\phi[\emptyset] = \phi$
3. $(\phi[S])[T] = \phi[S \sqcup T]$

Note that the functor generalizes the map $S \mapsto V^{\otimes S}$, while the morphisms generalize the extension-by-identity $\phi \otimes \mathrm{id}_S^S$.

Composition of morphisms $\phi_{C_1}^{D_1}$ and $\psi_{C_2}^{D_2}$ is defined when $C_1 = D_2$, and is written with the following concatenation operator: †1

$$\phi_{C_1}^{D_1} \mathbin{/\!\!/} \psi_{C_2}^{D_2} := \psi_{C_2}^{D_2} \circ \phi_{C_1}^{D_1} : \mathcal{C}_{D_1} \to \mathcal{C}_{C_2} \tag{2.8}$$

In the general case, we still have the map $\sqcup$ defined by $A_S \sqcup A_T = A_{S\sqcup T}$. Given morphisms $\phi_{C_1}^{D_1}$ and $\psi_{C_2}^{D_2}$ such that $D_1 \cap D_2 = \emptyset = C_1 \cap C_2$, we have a product morphism $\phi_{C_1}^{D_1}\psi_{C_2}^{D_2} := \phi \otimes \psi\colon \mathcal{C}_{D_1\sqcup D_2} \to \mathcal{C}_{C_1\sqcup C_2}$, which we write with concatenation.

Remark 2.5. To make expressions easier to read, in this paper we will introduce the domain extension implicitly in the following context: given morphisms $\phi_{C_1}^{D_1}$ and $\psi_{C_2}^{D_2}$ such that $D_2 \subseteq C_1$ and $C_2 \cap (C_1 \setminus D_2) = \emptyset = D_1 \cap (D_2 \setminus C_1)$, we define:

$$\phi_{C_1}^{D_1} \mathbin{/\!\!/} \psi_{C_2}^{D_2} := \phi_{C_1}^{D_1}[D_2 \setminus C_1] \mathbin{/\!\!/} \psi_{C_2}^{D_2}[C_1 \setminus D_2] \tag{2.9}$$

The two extreme cases of this definition are:

- When $C_1 \cap D_2 = \emptyset$, equation (2.9) becomes $\phi_{C_1}^{D_1}\psi_{C_2}^{D_2}$.
- When $C_1 = D_2$, equation (2.9) becomes the composition $\phi_{C_1}^{D_1} \mathbin{/\!\!/} \psi_{C_2}^{D_2}$ exactly.

Remark 2.6. While the $/\!\!/$ operator is associative, care must be taken that the compositions are well-defined in the presence of duplicated indices. While it is sufficient for all the finite sets in a composition to be pairwise disjoint, this condition will prove too restrictive for clear communication of formulae.

†1 We denote left-to-right composition with the "$/\!\!/$" symbol: $f \mathbin{/\!\!/} g := g \circ f$. Writing function composition in this order assists with readability when there are many functions to apply.

Defining a meta-group

To make the above definition more concrete, we will go through the process of defining a meta-group, which is a generalization of a group object. Traditionally, the data of a group object are the following:

- An object G in a category $\mathcal{C}$.
- A morphism $m\colon G \times G \to G$ called "multiplication".
- A "unit" morphism $\eta\colon \{1\} \to G$.[†2]
- An "inversion" morphism $S\colon G \to G$.
- A collection of relations between the morphisms, written as equalities of morphisms between Cartesian powers of G. For example, associativity may be written:

$$\begin{tikzcd} G \times G \times G \arrow[r, "m\times \mathrm{id}"] \arrow[d, "\mathrm{id}\times m"'] & G \times G \arrow[d, "m"] \\ G \times G \arrow[r, "m"'] & G \end{tikzcd} \tag{2.10}$$

Further, the data of these relations is extended to higher powers of G by acting on other components by the identity:

$$\begin{tikzcd} G^{n+3} \arrow[r, "m\times \mathrm{id}^{n+1}"] \arrow[d, "\mathrm{id}\times m\times \mathrm{id}^n"'] & G^{n+2} \arrow[d, "m\times \mathrm{id}^n"] \\ G^{n+2} \arrow[r, "m\times \mathrm{id}^n"'] & G^{n+1} \end{tikzcd} \tag{2.11}$$

Let us alter how we package these data so as to maximize the clarity of the meta-group structure:

1. Instead of linear orders of factors $G \times \cdots \times G$, we will index factors by a finite set X, writing it $G_X := \{f\colon X \to G\}$ in the style of equation (2.1).
2. The indexed factors will determine how the group operations act. For instance, multiplication of factor i and j together, with the result labelled in factor k is to be written $m^{ij}_k\colon G_{\{i,j\}} \to G_{\{k\}}$.

†2 When $\mathcal{C} = \mathbf{Set}$, we usually write the unit as an element $1 = \eta(1) \in G$

3. Instead of implicitly including extensions of morphisms to higher powers by the identity, we will parametrize the extension by finite sets by $\phi_C^D[X] := \phi_C^D \times \mathrm{id}_X^X$. For example, multiplication $m_k^{ij}\colon G_{\{i,j\}} \to G_{\{k\}}$ generates a family of maps $m_k^{ij}[X]\colon G_{\{i,j\}\sqcup X} \to G_{\{k\}\sqcup X}$, each of which must satisfy the relations of the group object such as equation (2.11).

This way of packaging the data leads us to the following generalization:

Definition 2.7. A meta-group in $\mathcal{C}$ is the following data:

- A family of objects $G_X \in \mathcal{C}$, indexed over finite sets X.
- A family of morphisms $m_k^{ij}[X]\colon G_{\{i,j\}\sqcup X} \to G_{\{k\}\sqcup X}$ called "multiplication".
- A family of "unit" morphisms $\eta_i[X]\colon G_X \to G_{\{i\}\sqcup X}$.
- An family of "inversion" morphisms $S_j^i[X]\colon G_{\{i\}\sqcup X} \to G_{\{j\}\sqcup X}$.
- A collection of relations between the morphisms, written as equalities of morphisms between the G_X's. For example, associativity may be written:

$$\begin{tikzcd} G_{\{1,2,3\}\sqcup X} \arrow[r, "m_1^{1,2}[X\sqcup\{3\}]"] \arrow[d, "m_2^{2,3}[X\sqcup\{1\}]"'] & G_{\{1,3\}\sqcup X} \arrow[d, "m_1^{1,3}[X]"] \\ G_{\{1,2\}\sqcup X} \arrow[r, "m_1^{1,2}[X]"'] & G_{\{1\}\sqcup X} \end{tikzcd} \tag{2.12}$$

2.3 ALGEBRAIC DEFINITIONS

We now introduce the algebraic structures which will be used to define the tangle invariant. These definitions follow those given by Majid in [Maj], although the ones presented below are given in a way that their corresponding meta-structure is readily visible.

Definition 2.8 (meta-algebra)**.** A meta-algebra (or meta-monoid) is a collection of objects $\{A_X\}_X$ in $\mathcal{C}$ together with an associative multiplication $m_k^{i,j}\colon A_{\{i,j\}} \to A_{\{k\}}$ (satisfying equation (2.13)), and a unit $\eta_i\colon A_\emptyset \to A_{\{i\}}$ satisfying equation (2.14).

Remark 2.9. When $\mathcal{C} = \mathbf{Vect}$ and $A_X = V^{\otimes X}$ for some vector space V, definition 2.8 becomes the more familiar definition of an algebra. When $A_\emptyset$

is a field, it is more common think of the unit as an element $\mathbf{1} \in V$. The unit map is then defined by linearly extending the assignment $\eta_i(1) = \mathbf{1}_i$.

$$\begin{tikzcd} A_{\{1,2,3\}} \arrow[r, "m_1^{1,2}"] \arrow[d, "m_2^{2,3}"'] & A_{\{1,3\}} \arrow[d, "m_1^{1,3}"] \\ A_{\{1,2\}} \arrow[r, "m_1^{1,2}"'] & A_{\{1\}} \end{tikzcd} \tag{2.13}$$

$$\begin{tikzcd} A_{\{1\}} \arrow[r, "\eta_2"] \arrow[dr, "\mathrm{id}"'] & A_{\{1,2\}} \arrow[d, shift right, "m_1^{2,1}"'] \arrow[d, shift left, "m_1^{1,2}"] \\ & A_{\{1\}} \end{tikzcd} \tag{2.14}$$

Remark 2.10. From now on, we will denote repeated multiplication as in equation (2.13) by using extra indices. For instance: $m_\ell^{i,j,k} := m_r^{i,j} /\!\!/ m_\ell^{r,k} = m_s^{j,k} /\!\!/ m_\ell^{i,s}$.

There is also the dual notion of a <u>coalgebra</u>, which arises by reversing the arrows in equations (2.13) and (2.14):

Definition 2.11 (meta-coalgebra)**.** A <u>meta-colagebra</u> (or <u>meta-comonoid</u>) is a collection $\{C_X\}_X$ together with a <u>comultiplication</u> $\Delta^i_{jk} \colon C_{\{i\}} \to C_{\{j,k\}}$ which is <u>coassociative</u> (equation (2.15)) and a <u>counit</u>, which is a map $\epsilon^i \colon A_i \to A_\emptyset$ satisfying equation (2.16).

$$\begin{tikzcd} C_{\{1,2,3\}} & C_{\{2,3\}} \arrow[l, "\Delta^1_{1,2}"'] \\ C_{\{1,2\}} \arrow[u, "\Delta^2_{2,3}"] & C_{\{1\}} \arrow[l, "\Delta^1_{1,2}"] \arrow[u, "\Delta^1_{1,3}"'] \end{tikzcd} \tag{2.15}$$

$$\begin{tikzcd} C_{\{1\}} & C_{\{1,2\}} \arrow[l, "\epsilon^2"'] \\ & C_{\{1\}} \arrow[ul, "\mathrm{id}"] \arrow[u, shift left, "\Delta^1_{1,2}"] \arrow[u, shift right, "\Delta^1_{2,1}"'] \end{tikzcd} \tag{2.16}$$

Remark 2.12. From now on, we will denote repeated comultiplication as in equation (2.15) by using extra indices. For instance: $\Delta^i_{j,k,\ell} := \Delta^i_{j,r} /\!\!/ \Delta^r_{k,\ell} = \Delta^i_{s,\ell} /\!\!/ \Delta^s_{j,j}$.

If a meta-object $\{B_X\}_x$ satisfies both definitions of an algebra and a coalgebra, we introduce a definition for when the structures are compatible with each other in the following way:

Definition 2.13 (meta-bialgebra)**.** A <u>meta-bialgebra</u> (or <u>meta-bimonoid</u>) is a meta-algebra (B, m, η) and a meta-coalgebra (B, Δ, ϵ), such that Δ and ϵ are meta-algebra morphisms. †3

†3 B_X inherits a (co)algebra structure from B, given by $(B_X)_Y := B_{X^Y}$ and component-wise operations. The bialgebra structure on $B_\emptyset$ is given by $m = \eta = \Delta = \epsilon = \mathrm{id}$.

$$\begin{tikzcd} B_{\{1,2\}} \arrow[r, "m_1^{1,2}"] \arrow[d, "\Delta^1_{1_1,1_2} /\!\!/ \Delta^2_{2_1,2_2}"] & B_{\{1\}} \arrow[d, "\Delta^1_{1,2}"] \\ B_{\{1_1,1_2,2_1,2_2\}} \arrow[r, "m_1^{1_1,1_2} /\!\!/ m_2^{2_1,2_2}"'] & B_{\{1,2\}} \end{tikzcd} \tag{2.17}$$

$$\begin{tikzcd} & B_{\{1\}} \arrow[d, "\Delta^1_{1,2}"] \\ B_\emptyset \arrow[ur, "\eta_1"] \arrow[r, "\eta_1 /\!\!/ \eta_2"'] & B_{\{1,2\}} \end{tikzcd} \tag{2.18}$$

$$\begin{tikzcd} B_{\{1,2\}} \arrow[r, "m_1^{1,2}"] \arrow[dr, "\epsilon^1 /\!\!/ \epsilon^2"'] & B_{\{1\}} \arrow[d, "\epsilon^1"] \\ & B_\emptyset \end{tikzcd} \tag{2.19}$$

$$\begin{tikzcd} B_\emptyset \arrow[r, "\eta_1"] \arrow[dr, "\mathrm{id}"'] & B_{\{1\}} \arrow[d, "\epsilon^1"] \\ & B_\emptyset \end{tikzcd} \tag{2.20}$$

Remark 2.14. The conditions for Δ being an algebra morphism are presented in equations (2.17) and (2.18), while those for ϵ are in equations (2.19) and (2.20).[†4] Observing invariance under arrow reversal, it may not come as a surprise that equations (2.17) and (2.19) also are the conditions for m being a coalgebra morphism, and equations (2.18) and (2.20) tell us that η is as well.

Next, we introduce a notion of invertibility which extends a bialgebra to a Hopf algebra.

Definition 2.15 (meta-Hopf algebra)**.** A meta-Hopf algebra (or meta-Hopf monoid)is a bialgebra H together with a map $S\colon H \to H$ called the antipode, which satisfies $\Delta^1_{1,2} /\!\!/ S^1_1 /\!\!/ m_1^{1,2} = \epsilon^1 /\!\!/ \eta_1 = \Delta^1_{1,2} /\!\!/ S^2_2 /\!\!/ m_1^{1,2}$. As a commutative diagram, this looks like equation (2.21)

$$\begin{tikzcd} H_{\{1\}} \arrow[r, "\epsilon^1"] \arrow[d, "\Delta^1_{1,2}"'] & H_\emptyset \arrow[r, "\eta_1"] & H_{\{1\}} \\ H_{\{1,2\}} \arrow[r, shift left, "S^2_2"] \arrow[r, shift right, "S^1_1"'] & H_{\{1,2\}} \arrow[ur, "m_1^{1,2}"'] & \end{tikzcd} \tag{2.21}$$

In order to do knot theory, we need an algebraic way to represent a crossing of two strands. This is accomplished by the so-called $\mathcal{R}$-matrix:

Definition 2.16 (quasitriangular meta-Hopf algebra)**.** A quasitriangular meta-Hopf algebra (or quasitriangular meta-Hopf monoid) is a Hopf algebra

†4 While notation explicitly naming each tensor factor appears cumbersome in these diagrams, it will prove invaluable later when used on tangle diagrams, so we leave it as is for the sake of consistency.

H, together with an invertible element $\mathcal{R}_{i,j} \in H_{i,j}$, called the $\underline{\mathcal{R}\text{-matrix}}$, which satisfies the following properties: (we will denote the inverse by $\overline{\mathcal{R}}$)

$$\mathcal{R}_{12} \,/\!\!/\, \Delta^2_{23} = \mathcal{R}_{a2}\mathcal{R}_{b3} \,/\!\!/\, m^{ab}_1 \tag{2.22}$$

$$\mathcal{R}_{13} \,/\!\!/\, \Delta^1_{12} = \mathcal{R}_{1b}\mathcal{R}_{2a} \,/\!\!/\, m^{ab}_3 \tag{2.23}$$

$$\Delta^1_{21} = \Delta^1_{12}\mathcal{R}_{1_i,2_i}\overline{\mathcal{R}}_{1_f,2_f} \,/\!\!/\, m_1^{1_i,1,1_f} \,/\!\!/\, m_2^{2_i,2,2_f} \tag{2.24}$$

Definition 2.17 (Drinfeld element)**.** In a quasitriangular meta-Hopf algebra H, the <u>Drinfeld element</u>, $\mathfrak{u} \in H$ is given by:

$$\mathfrak{u} := \mathcal{R}_{21} \,/\!\!/\, S^2_2 \,/\!\!/\, m^{12} \tag{2.25}$$

Definition 2.18 (monodromy)**.** Each quasitriangular meta-Hopf algebra has a <u>monodromy</u> $Q_{12} := \mathcal{R}_{12}\mathcal{R}_{34} \,/\!\!/\, m^{14}_1 \,/\!\!/\, m^{23}_2$. Its inverse will be denoted $\overline{Q}_{12} = \overline{\mathcal{R}}_{12}\overline{\mathcal{R}}_{34} \,/\!\!/\, m^{14}_1 \,/\!\!/\, m^{23}_2$.

Lemma 2.19. *The Drinfeld element* $\mathfrak{u}$ *satisfies for all* $h \in H$*:*

$$\mathfrak{u}_1 h_2 \mathfrak{u}_3 \,/\!\!/\, m^{1,2,3} = h \,/\!\!/\, S \,/\!\!/\, S \tag{2.26}$$

$$\mathfrak{u} \,/\!\!/\, \Delta_{12} = \mathfrak{u}_1\mathfrak{u}_2\overline{Q}_{34} \,/\!\!/\, m^{13}_1 \,/\!\!/\, m^{24}_2 \tag{2.27}$$

Proof. See Majid's work in [Maj] or Etingof and Schiffmann in [ES] for more details on this standard result. Note that the proof does not rely on the additive structure of the Hopf algebra, which allows us to extend this result to the realm of meta-Hopf algebras. □

Definition 2.20 (ribbon meta-Hopf algebra)**.** A quasitriangular meta-Hopf algebra H is called <u>ribbon</u> if it has an element $\nu \in Z(H)$ such that:

$$\nu_1\nu_2 \,/\!\!/\, m^{12} = \mathfrak{u}_1\mathfrak{u}_2 \,/\!\!/\, S^2_2 \,/\!\!/\, m^{12} \tag{2.28}$$

$$\nu_1 \,/\!\!/\, \Delta^1_{12} = \nu_1\nu_2 \,/\!\!/\, \overline{Q}_{34} \,/\!\!/\, m^{13}_1 \,/\!\!/\, m^{24}_2 \tag{2.29}$$

$$\nu \,/\!\!/\, S = \nu \tag{2.30}$$

$$\nu \,/\!\!/\, \epsilon = \eta \,/\!\!/\, \epsilon = 1 \tag{2.31}$$

Definition 2.21 (distinguished grouplike element (spinner))**.** A distinguished grouplike element (or spinner) in a quasitriangular meta-Hopf algebra H is an invertible element $C \in H$ (with inverse $\overline{C}$) such that for all $x \in H$:

$$C_1 \nu_2 C_3 \mathbin{/\!/} S_2^2 \mathbin{/\!/} m^{123} = \nu \tag{2.32}$$

$$C_1 \mathbin{/\!/} \Delta^1_{12} = C_1 C_2 \tag{2.33}$$

$$C \mathbin{/\!/} S = \overline{C} \tag{2.34}$$

$$C_1 x_2 \overline{C}_3 \mathbin{/\!/} m^{1,2,3} = x \mathbin{/\!/} S \mathbin{/\!/} S \tag{2.35}$$

$$C \mathbin{/\!/} \epsilon = \eta \mathbin{/\!/} \epsilon = 1 \tag{2.36}$$

Lemma 2.22 (spinners and ribbon Hopf algebras)**.** *If a Hopf algebra has either a ribbon element ν or a spinner C, then it must have the other as well, given by the formula: $C_1 \nu_2 \mathbin{/\!/} m^{12} = \mathfrak{u}$.*

2.4 THE META-ALGEBRA OF TANGLE DIAGRAMS

The particular structures introduced were chosen for their ability to represent the topological properties of knotted objects. We will now introduce the notion of a tangle and demonstrate its meta-algebraic structure.

Upright tangles

For our purposes, a tangle will be visualised as follows: take a stiff circular metal frame and attach a collection of strings to the wire, ensuring that the strings always remain inside the circle, and that each string is tied to the metal frame in two unique locations (that is, no two strings share an endpoint).

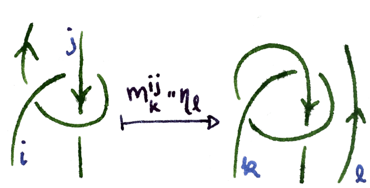

Figure 2.1: Example of a tangle

Definition 2.23 (pure tangle)**.** A pure tangle is an embedding of line segments (called components or strands) into the thickened unit disk $D \times$

$[-1, 1]$ (or a disjoint union of such disks) such that the endpoints of the line segments are fixed along $\partial D \times \{0\}$. Two pure tangles are considered equivalent if there exists an isotopy of the embedding which fixes the endpoints of the strands. The term "pure" refers to the absence of closed loops.

Definition 2.24 (framed tangle)**.** The object which is more natural for us to deal with is the framed tangle, which is a pure tangle together with a choice of section of the normal bundle for each component. One may equivalently think of replacing the strings in a pure tangle with thin ribbons. Unless otherwise mentioned, it will be assumed that all tangles are framed.

In order to best capture the combinatorial properties of a tangle, observe that a generic projection of a tangle to its central core $D \times \{0\}$ will result in the strands forming a graph, with each crossing of two strands in the tangle producing a vertex in the graph. By assigning to each vertex the sign of the corresponding crossing (either "positive" or "negative"), we end up with a combinatorial object which is equivalent to the original concept of a tangle.

Definition 2.25 (pure tangle diagram)**.** A pure tangle diagram is a projection of a tangle onto its central core such that all the line segments are immersions which intersect both the boundary disk and the other strands transversally, together with an assignment of a sign to each strand intersection. Small open neighbourhoods of these intersections are called crossings, while the complement of the crossings is a collection of embedded line segments called arcs.

Two pure tangle diagrams are considered equivalent if they differ by a finite sequence of Reidemeister moves, as outlined in **??**–2.4

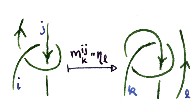

Figure 2.2: Reidemeister move $R1'$

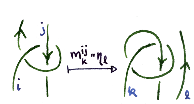

Figure 2.3: Reidemeister move $R2$

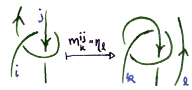

Figure 2.4: Reidemeister move $R3$

The rotation numbers of arcs will play a role in this thesis, so we will capture these data in the following way (as described in **??**):

Definition 2.26 (upright pure tangle diagram)**.** We will put a further requirement on our tangles: each arc must begin and end pointing upwards, and each crossing must involve only upward-pointing curves. This way, each arc will have a well-defined integer rotation number. Two tangles are considered equivalent if they agree under the "rotational Reidemeister moves".

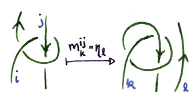

Figure 2.5: Rotational Reidemeister move $R1'_{\mathrm{rot}}$

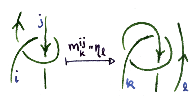

Figure 2.6: The two rotational Reidemeister moves $R2_{\mathrm{rot}}$

Fortunately, ambient isotopy allows us to rotate any classical tangle into an upward-pointing form. Additionally, there is only one way to do this. We reproduce the proof of this fact by Bar-Natan and van der Veen in [BNvdV] below:

Lemma 2.27 (tangles inject into upright tangles)**.** *To each pure tangle diagram D there exists an upright pure tangle diagram D' obtained from D by a planar isotopy. Further, if D'' is another such upright pure tangle diagram obtained from D, then D' and D'' differ by a finite sequence of rotational Reidemeister moves and a change of rotation number at the endpoints.*

Proof. Each arc and crossing in the diagram D may be rotated so that its endpoints are pointing upwards, giving rise to a diagram D'. Two (nonupright) tangle diagrams are equivalent when they differ by a finite

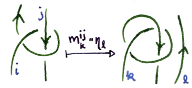

Figure 2.7: Rotational Reidemeister move $R3_{\text{rot}}$

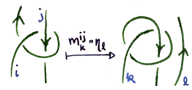

Figure 2.8: Whirling move Wh

sequence of Reidemeister moves. Each of these Reidemeister may also be rotated to an equivalence of upright tangles, each of which is given as a rotational Reidemeister move figures 2.5 to 2.7. The last possibility is the rotation of an entire crossing, which is covered by figure 2.8. □

The meta-algebra structure of upright tangle diagrams

Let us now formally connect tangle diagrams with meta-algebras.

Theorem 2.28 (tangles form a ribbon meta-Hopf algebra)**.** *Define $\mathcal{T}_X$ to be the set of upright tangles with $|X|$ strands, each labelled by an element $i \in X$. Then the collection $\{\mathcal{T}_X\}_X$ forms a quasitriangular ribbon meta-Hopf algebra with the following operations:*

- *multiplication $m^{ij}_k[X]$ takes a tangle with strands $X \sqcup \{i, j\}$ and glues the end of strand i to strand j, labelling the resulting strand k.*
- *the unit $\eta_i[X]$ takes a tangle diagram with strands X and introduces a new strand i which does not touch any of the other strands.*
- *the comultiplication $\Delta^i_{jk}[X]$ takes a tangle with strands $X \sqcup \{i\}$ and doubles strand i, separating the two strands along the framing of strand i, calling the right strand j and the left one k.*[†5]
- *the counit $\epsilon^i[X]$ takes a tangle with strands indexed by $X \sqcup \{i\}$ and returns the tangle with strand labelled by i deleted.*

†5 While this convention appears unfortunate, we follow the notation laid out in [BNvdV] so that the antipode and spinner have a more memorable representation, namely looking like the letters they are represented by (see **??** for more details).

- *The antipode $S^i_j[X]$ takes a tangle with strands labelled by $X \sqcup \{i\}$ and reverses the direction of strand i, then adds a counter-clockwise cap to the new beginning, and a clockwise cup to the end. This new strand is called j. When applied to a single vertical strand, the resulting tangle looks like the letter "S".*

- *the $\mathcal{R}$-matrix $\mathcal{R}_{ij}$ is given by the two-strand tangle with a single positive crossing of strand i over strand j. The inverse $\mathcal{R}$-matrix $\overline{\mathcal{R}}_{ij}$ is the two-strand tangle with a* <u>*negative*</u> *crossing of strand i over strand j.*

- *The spinner $C_i[X]$ takes a tangle in $\mathcal{T}_X$ and adds a new strand with rotation number 1 which has no interactions with any other strands. This new strand looks like the letter "C".*

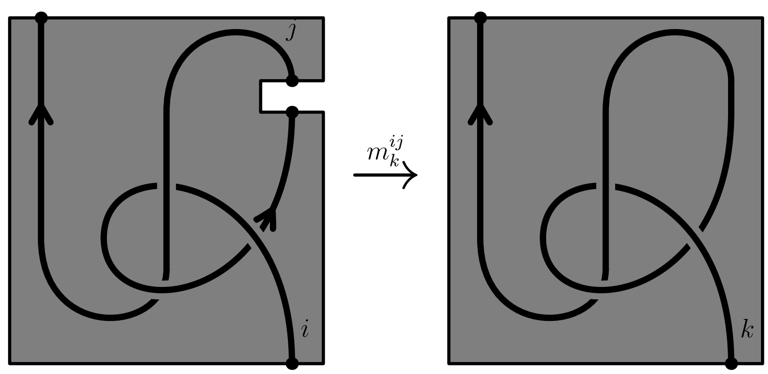

Figure 2.9: Multiplication m^{ij}_k stitches two strands in a tangle together.

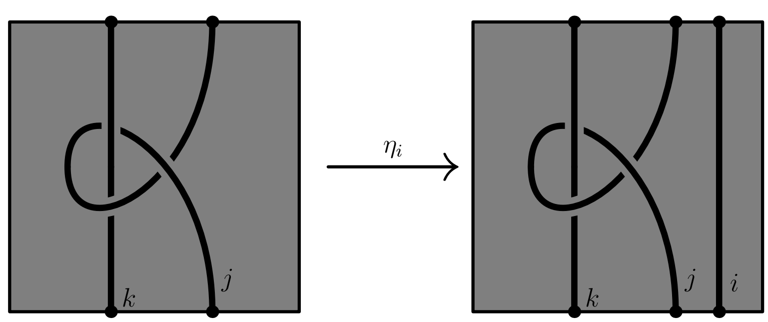

Figure 2.10: The unit η_i introduces a new strand in a tangle.

Proof. Associativity of multiplication (equation (2.13)) follows from the fact that stitching strands together amounts to concatenating the order

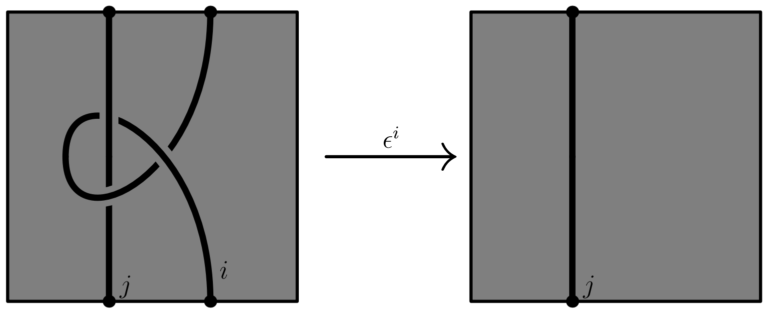

Figure 2.11: The counit ϵ^i deletes a strand in a tangle.

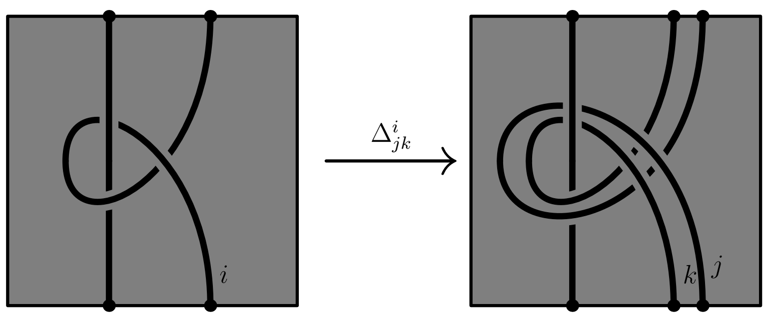

Figure 2.12: The comultiplication Δ^i_{jk} doubles a strand in a tangle along its framing. Notice the right-to-left strand labels.

of the crossings each strand interacts with. Since list concatenation is an associative operation, associativity follows in this case as well.

Adding a non-interacting strand to a diagram, then stitching it to an existing strand (equation (2.14)) does not change any of the combinatorial data in the diagram, and results in identical diagrams.

Establishing coassociativity (equation (2.15)) amount to the same argument that cutting a piece of paper into three strips does not depend on the order of cutting.

The counit identity (equation (2.16)) states deleting a strand is the same operation as first doubling it, then deleting both resulting strands.

The meta-bialgebra axioms we verify next:

Equation (2.17) states that if two strands are stitched together, then the resulting strand is doubled, this could have equivalently been achieved by doubling each of the original strands, then performing a stitching on both resulting pairs of strands.

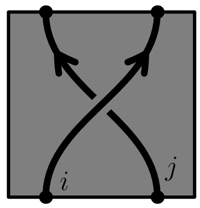

Figure 2.13: The $\mathcal{R}$-matrix $\mathcal{R}_{ij}$ represents a tangles with a single positive crossing.

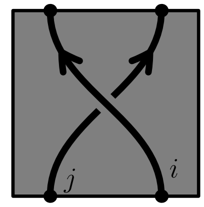

Figure 2.14: The inverse $\mathcal{R}$-matrix $\overline{\mathcal{R}}_{ij}$ represents a tangle with a single negative crossing.

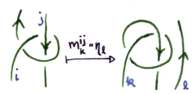

Figure 2.15: The Drinfeld element in the meta-Hopf algebra of tangles.

Equation (2.19) simply states that stitching two strands together, then removing the resulting strand could have equally been achieved by removing both of the original strands without stitching them first.

Equation (2.20) states that introducing a strand, then immediately removing it is the identity operation.

Equation (2.18) says that doubling a newly-introduced (and therefore free of crossings) strand is the same operation as introducing two strands separately. (Recall that in the virtual case, proximity of strands is not accounted for)

Equation (2.21) states that when a strand is doubled, then one of the two strands is reversed, multiplying the two strands together results in a strand which can be rearranged to not interact with any of the other strands. This can be readily seen, as this newly-created strand looks like a snake weaving through the tangle diagram. One can remove the snake by applying a series of Reidemeister 2 moves, resulting in a strand disjoint from the rest of the diagram. This is the same as deleting the original strand, then introducing a new disjoint one.

The quasitriangular axioms are equalities of pairs of three-strand tangles:

- Equations (2.22) and (2.23) tell us that doubling a strand involved in a single crossing can also be built by adjoining two crossings together.

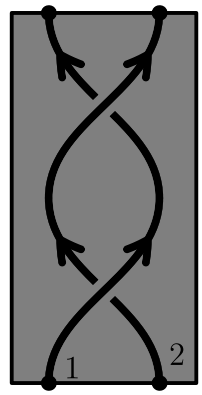

Figure 2.16: The monodromy in the meta-Hopf algebra of tangles.

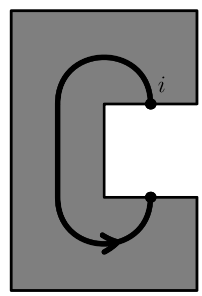

Figure 2.17: The spinner C_i represents a strand with rotation number 1.

- Equation (2.24) tells us that we can swap the order of a doubled strand by adding crossings to either end (reminiscent of a Reidemeister 2 move)

Finally, we observe that the quotient we introduce to tangle diagrams by the Reidemeister moves does not introduce any new relations. Reidemeister 2 follows from the invertibility of the $\mathcal{R}$-matrix. Next, it is readily seen that the quasitriangular relations governing the $\mathcal{R}$-matrix force it to solve the Yang-Baxter equation, which is one equivalent to the Reidemeister 3 in this case.

Using lemma 2.22, it is enough to verify the spinner axioms (equations (2.32) to (2.36)). All these axioms have corresponding pictures one can draw, keeping in mind the orientations in the definitions of the relevant operations. □

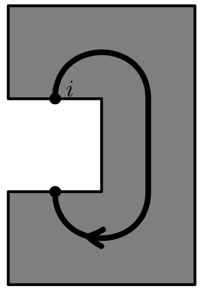

Figure 2.18: The inverse spinner $\overline{C}_i$ represents a strand with rotation number -1.

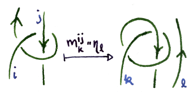

Figure 2.19: A ribbon element in the meta-Hopf algebra of tangles. One can use lemma 2.22 to verify this is compatible with the spinner.

2.5 THE *ybax* META-ALGEBRA

Here we define the ribbon Hopf algebra U.

Define the Lie algebra

$$\mathfrak{g} := \operatorname{span}\left\{ \mathbf{y}, \mathbf{b}, \mathbf{a}, \mathbf{x} \;\middle|\; [\mathbf{a}, \mathbf{x}] = \mathbf{x}, [\mathbf{a}, \mathbf{y}] = -\mathbf{y}, [\mathbf{x}, \mathbf{y}] = \mathbf{b}, [\mathbf{b}, \,\cdot\,] = 0 \right\} \tag{2.37}$$

Then the algebra U is defined to be the universal enveloping algebra $\mathfrak{U}(\mathfrak{g})$. The bialgebra structure of U is: for any $\mathbf{z} \in \{\mathbf{y}, \mathbf{b}, \mathbf{a}, \mathbf{x}\}$, we have

$$\Delta_{i,j}(\mathbf{z}) = \mathbf{z}_i + \mathbf{z}_j \tag{2.38}$$

$$\epsilon(\mathbf{z}) = 0 \tag{2.39}$$

Next, we define the Hopf algebra structure by defining the antipode

$$S(\mathbf{z}) = -\mathbf{z} \tag{2.40}$$

The ribbon structure of U requires we introduce both the $\mathcal{R}$-matrix and the spinner C:

$$\mathcal{R}_{i,j} := \exp\left(\mathbf{a}_j \mathbf{b}_i + \frac{1 - \mathbf{B}_i}{\mathbf{b}_i} \mathbf{y}_i \mathbf{x}_j \right) \tag{2.41}$$

$$C := \sqrt{\mathbf{B}} \tag{2.42}$$

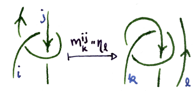

Figure 2.20: Example of a tangle satisfying equation (2.22)

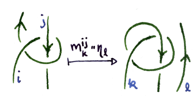

Figure 2.21: Example of a tangle satisfying equation (2.23)

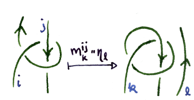

Figure 2.22: Example of a tangle satisfying equation (2.24)

3

PERTURBED GAUSSIANS

We now summarize the work of Bar-Natan and van der Veen in [BNvdV], which develops a universal knot invariant using perturbed Gaußians.

3.1 EXPRESSING MORPHISMS AS GENERATING FUNCTIONS

When defining a morphism-valued tangle invariant, one needs a compact way of encoding the morphism. In [BNvdV] this is achieved through the use of generating functions, whose definition we reproduce below:

For A and B finite sets, consider the set $\hom(\mathbb{Q}[z_A], \mathbb{Q}[z_B])$ of linear maps between multivariate polynomial rings. Such a map is determined by its values on the monomials $z_A^{\mathbf{n}}$ for each multi-index $\mathbf{n} \in \mathbb{N}^A$.

Definition 3.1 (Exponential generating function)**.** The exponential generating function of a map $\Phi\colon \mathbb{Q}[z_A] \to \mathbb{Q}[z_B]$ between polynomial spaces is

$$\mathcal{G}(\Phi) := \sum_{\mathbf{n}\in\mathbb{N}^A} \frac{\Phi(z_A^{\mathbf{n}})}{\mathbf{n}!} \zeta_A^{\mathbf{n}} \in \mathbb{Q}[z_B][\![\zeta_A]\!] \tag{3.1}$$

Remark 3.2. Extending the definition of Φ to $\mathbb{Q}[z_B][\![\zeta_A]\!]$ by the extending scalars to $\mathbb{Q}[\![\zeta_A]\!]$ gives us an equivalent formulation:

$$\mathcal{G}(\Phi) = \Phi\left(\sum_{\mathbf{n}\in\mathbb{N}^A} \frac{(z_A\zeta_A)^{\mathbf{n}}}{\mathbf{n}!}\right) = \Phi\Big(\mathcal{G}(\mathrm{id}_{\mathbb{Q}[z_A]})\Big) \tag{3.2}$$

By the PBW theorem, we know that U is isomorphic as a vector space to the polynomial ring $\mathbb{Q}[y, b, a, x]$ by choosing an ordering of the generators (following [BNvdV], we use $(\mathbf{y}, \mathbf{b}, \mathbf{a}, \mathbf{x})$):

$$\begin{aligned} \mathbb{O}\colon \mathbb{Q}[y, b, a, x] &\xrightarrow{\sim} U \\ y^{n_1} b^{n_2} a^{n_3} x^{n_4} &\mapsto \mathbf{y}^{n_1} \mathbf{b}^{n_2} \mathbf{a}^{n_3} \mathbf{x}^{n_4} \end{aligned} \tag{3.3}$$

Using this vector space isomorphism, [BNvdV] expresses all Hopf algebra operations as perturbed Gaußians. To extend the resulting tangle invariant to one on links, one would need to define a trace operator on U. The first natural place to look is the coinvariants, $U_U = U/[U,U]$. In what follows, we will compute U_U, determine a vector space isomorphism to a suitable polynomial ring, and compute the corresponding generating function of the quotient map $\mathrm{tr}\colon U \to U_U$.

3.2 ROTATIONAL TANGLE INVARIANTS FROM A RIBBON HOPF ALGEBRA

Here we describe the morphism from the category of pure rotational virtual tangles to a ribbon Hopf algebra, as outlined by Bar-Natan and van der Veen in [BNvdV].

We define the morphism of meta-ribbon Hopf algebras in two steps:

1. Given a pure tangle, we write out a sequence of meta-ribbon Hopf algebra operations which produce the tangle. Each operation is then mapped to the corresponding operation on the algebra, with a sequence of operations mapped to composition.

2. Since computing compositions of operations is an essential component to this invariant, we then define an equivalent category which allows for the more efficient computation. This is done by replacing morphisms

To efficiently describe $\Bbbk$-linear maps between tensor powers of the algebra U, we define categories $\mathcal{U}$, $\mathcal{H}$, and $\mathcal{C}$ with objects finite sets and morphisms:

$$\mathrm{Hom}_{\mathcal{U}}(J,K) := \mathrm{Hom}_{\Bbbk}(U^{\otimes J}, U^{\otimes K}) \tag{3.4}$$

$$\mathrm{Hom}_{\mathcal{H}}(J,K) := \mathrm{Hom}_{\Bbbk}(\mathbb{Q}[z_J], \mathbb{Q}[z_K]) \tag{3.5}$$

$$\mathrm{Hom}_{\mathcal{C}}(J,K) := \mathbb{Q}[z_K][\![\zeta_J]\!] \tag{3.6}$$

There exist monoidal isomorphisms between these categories, namely $\mathbb{O}\colon \mathcal{H} \xrightarrow{\sim} \mathcal{U}$ and $\mathcal{G}\colon \mathcal{H} \xrightarrow{\sim} \mathcal{C}$ as introduced in equations (3.1) and (3.3).

We use this formulation because of the existence of a computationally amenable subcategory of $\mathcal{C}$ which contains the image of this invariant.

Formulating composition in other categories

Composing operations in $\mathcal{U}$ or $\mathcal{H}$ is straightforward to define, but lacks a closed form. However, on $\mathcal{C}$, the corresponding definition of composition takes the following form (quoted from [BNvdV, Lemma 3]):

Lemma 3.3 (Composition of generating functions). *Suppose J, K, L are finite sets and $\phi \in \operatorname{Hom}(\mathbb{Q}[z_J], \mathbb{Q}[z_K])$ and $\psi \in \operatorname{Hom}(\mathbb{Q}[z_K], \mathbb{Q}[z_L])$. We have*

$$\mathcal{G}(\phi \mathbin{/\!/} \psi) = \left(\mathcal{G}(\phi)\big|_{z_K \to \partial\zeta_K} \mathcal{G}(\psi) \right)\Bigg|_{\zeta_K = 0} \tag{3.7}$$

Since the above notation will occur several times, we will use the notion of contraction used by Bar-Natan and van der Veen (taken from [BNvdV, Definition 4]):

Definition 3.4 (Contraction). Let $f \in \Bbbk[\![r, s]\!]$ be a powerseries. The contraction of $f = \sum_{k,l} c_{k,l} r^k s^l$ along the pair (r, s) is:

$$\langle f \rangle_{(r,s)} := \sum_k c_{k,k} k! = \sum_{k,l} c_{k,l} \partial_s^k s^l \Bigg|_{s=0} \tag{3.8}$$

Further, this notation is to be extended to allow for multiple consecutive contractions for $f \in \Bbbk[\![r_i, s_i]\!]_{i \leq n}$:

$$\langle f \rangle_{((r_i)_i, (s_i)_i)} := \left\langle \left\langle \langle f \rangle_{(r_1, s_1)} \right\rangle_{(r_2, s_2)} \cdots \right\rangle_{(r_n, s_n)} \tag{3.9}$$

It is important to note that contraction does not always define a convergent expression. We will focus our attention on cases when convergence is well-defined, and especially those where the computation is accessible.

The theorem we will rely heavily on in this thesis is the following, taken from [BNvdV, Theorem 6]:

Theorem 3.5 (Contraction theorem). *For any $n \in \mathbb{N}$, consider the ring $R_n = \mathbb{Q}[r_j, g_j][\![s_j, W_{ij}, f_j \mid 1 \leq i, j \leq n]\!]$. Then*

$$\langle \mathbf{e}^{gs + rf + rWs} \rangle_{r,s} = \det(\tilde{W}) \mathbf{e}^{g\tilde{W}f} \tag{3.10}$$

where $\tilde{W} = (1 - W)^{-1}$.

The main takeaway of this theorem is this: morphisms whose generating functions are Gaußians have a clean formula for composition. Furthermore, this formula is computationally reasonable, growing only polynomially in complexity with n. This is contrasted with the conventional approach of choosing a representation V of U. When one considers morphisms between large tensor powers $V^{\otimes n}$, the computational complexity is exponential in n.

Expressing Hopf algebra operations as perturbed Gaußians

Theorem 3.6 (The meta-Hopf structure of U is Gaußian). *We will now observe that the meta-Hopf algebra operations for U as defined in* **??** *all have the form of a perturbed Gaußian. Namely:*

$$\mathcal{G}\left(m_k^{ij}\right) = \exp\left((\alpha_i+\alpha_j)a_k + (\beta_i+\beta_j+\xi_i\eta_j)b_k + \left(\frac{\xi_i}{\mathcal{A}_j}+\xi_j\right)x_k + \left(\frac{\eta_j}{\mathcal{A}_i}+\eta_i\right)y_k\right) \tag{3.11}$$

$$\mathcal{G}(\eta_i) = 1 \tag{3.12}$$

$$\mathcal{G}\left(\Delta^i_{jk}\right) = \exp\left(\beta_i(b_j+b_k) + \alpha_i(a_j+a_k) + \eta_i(y_j+y_k) + \xi_i(x_j+x_k)\right) \tag{3.13}$$

$$\mathcal{G}(\epsilon^i) = 1 \tag{3.14}$$

$$\mathcal{G}(S^i_i) = \exp(-a_i\alpha_i - b_i\beta_i - \eta_i\mathcal{A}_i y_i - \mathcal{A}_i\xi_i x_i + \eta_i\mathcal{A}_i\xi_i b_i) \tag{3.15}$$

$$\mathcal{G}(\mathcal{R}_{ij}) = \exp\left(a_j b_i + \frac{B_i-1}{-b_i}y_i x_j\right) \tag{3.16}$$

$$\mathcal{G}(C_i) = \sqrt{B_i} \tag{3.17}$$

$$\mathcal{G}(\nu_i) = \sqrt{B_i}\exp\left(a_i b_i + \frac{1-B_i}{b_i}x_i y_i\right) \tag{3.18}$$

Proof. To prove equation (3.11), we first note the Weyl canonical commutation relation:

$$\mathbf{e}^{\xi\mathbf{x}}\mathbf{e}^{\eta\mathbf{y}} = \mathbf{e}^{-\xi\eta\mathbf{b}}\mathbf{e}^{\eta\mathbf{y}}\mathbf{e}^{\xi\mathbf{x}} \tag{3.19}$$

Secondly, using equation (4.4) and $\mathcal{A} := \mathbf{e}^{\alpha}$, we compute

$$\mathbf{e}^{\alpha\mathbf{a}}\mathbf{e}^{\eta\mathbf{y}} = \mathbf{e}^{\alpha\mathbf{a}}\sum_n \frac{(\eta\mathbf{y})^n}{n!} = \sum_n \frac{(\eta\mathbf{y})^n}{n!}\mathbf{e}^{\alpha(\mathbf{a}-n)} = \sum_n \frac{(\eta\mathbf{y})^n}{n!}\mathbf{e}^{\alpha\mathbf{a}}\mathcal{A}^{-n} = \mathbf{e}^{\frac{\eta}{\mathcal{A}}\mathbf{y}}\mathbf{e}^{\alpha\mathbf{a}} \tag{3.20}$$

similarly,

$$\mathbf{e}^{\xi\mathbf{x}}\mathbf{e}^{\alpha\mathbf{a}} = \sum_n \frac{(\xi\mathbf{x})^n}{n!}\mathbf{e}^{\alpha\mathbf{a}} = \sum_n \mathbf{e}^{\alpha(\mathbf{a}-n)}\frac{(\xi\mathbf{x})^n}{n!} = \sum_n \mathbf{e}^{\alpha\mathbf{a}}\frac{(\frac{\xi\mathbf{x}}{\mathcal{A}})^n}{n!} = \mathbf{e}^{\alpha\mathbf{a}}\mathbf{e}^{\frac{\xi}{\mathcal{A}}\mathbf{x}} \tag{3.21}$$

Using this relation allows us to commute exponentials past each other to bring expressions into $ybax$-order. Below we omit the index k for readability:

$$\begin{aligned}
\mathcal{G}(m^{ij}) &= \left(\mathbb{O}^{-1}\circ m^{ij}\circ\mathbb{O}\right)\left(\mathbf{e}^{\eta_i y_i+\beta_i b_i+\alpha_i a_i+\xi_i x_i}\mathbf{e}^{\eta_j y_j+\beta_j b_j+\alpha_j a_j+\xi_j x_j}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{\eta_i\mathbf{y}}\mathbf{e}^{\beta_i\mathbf{b}}\mathbf{e}^{\alpha_i\mathbf{a}}\mathbf{e}^{\xi_i\mathbf{x}}\mathbf{e}^{\eta_j\mathbf{y}}\mathbf{e}^{\beta_j\mathbf{b}}\mathbf{e}^{\alpha_j\mathbf{a}}\mathbf{e}^{\xi_j\mathbf{x}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{\eta_i\mathbf{y}}\mathbf{e}^{\beta_i\mathbf{b}}\mathbf{e}^{\alpha_i\mathbf{a}}\left(\mathbf{e}^{-\xi_i\eta_j\mathbf{b}}\mathbf{e}^{\eta_j\mathbf{y}}\mathbf{e}^{\xi_i\mathbf{x}}\right)\mathbf{e}^{\beta_j\mathbf{b}}\mathbf{e}^{\alpha_j\mathbf{a}}\mathbf{e}^{\xi_j\mathbf{x}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{(\beta_j+\beta_i-\xi_i\eta_j)\mathbf{b}}\mathbf{e}^{\eta_i\mathbf{y}}\mathbf{e}^{\frac{\eta_j}{\mathcal{A}_i}\mathbf{y}}\mathbf{e}^{\alpha_i\mathbf{a}}\mathbf{e}^{\xi_i\mathbf{x}}\mathbf{e}^{\alpha_j\mathbf{a}}\mathbf{e}^{\xi_j\mathbf{x}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{(\beta_j+\beta_i-\xi_i\eta_j)\mathbf{b}}\mathbf{e}^{(\eta_i+\frac{\eta_j}{\mathcal{A}_i})\mathbf{y}}\mathbf{e}^{\alpha_i\mathbf{a}}\mathbf{e}^{\alpha_j\mathbf{a}}\mathbf{e}^{\frac{\xi_i}{\mathcal{A}_j}\mathbf{x}}\mathbf{e}^{\xi_j\mathbf{x}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{\left(\eta_i+\frac{\eta_j}{\mathcal{A}_i}\right)\mathbf{y}}\mathbf{e}^{(\beta_j+\beta_i-\xi_i\eta_j)\mathbf{b}}\mathbf{e}^{(\alpha_i+\alpha_j)\mathbf{a}}\mathbf{e}^{\left(\frac{\xi_i}{\mathcal{A}_j}+\xi_j\right)\mathbf{x}}\right)\\
&= \mathbf{e}^{\left(\eta_i+\frac{\eta_j}{\mathcal{A}_i}\right)y}\mathbf{e}^{(\beta_j+\beta_i-\xi_i\eta_j)b}\mathbf{e}^{(\alpha_i+\alpha_j)a}\mathbf{e}^{\left(\frac{\xi_i}{\mathcal{A}_j}+\xi_j\right)x}
\end{aligned} \tag{3.22}$$

Since this expression is now written in the **ybax**-order, we conclude that the corresponding generating function is this same expression, but written with commuting variables.

The other computation we must verify is the antipode, which follows similarly:

$$\begin{aligned}
\mathcal{G}(S) &= \left(\mathbb{O}^{-1}\circ S\circ\mathbb{O}\right)\left(\mathbf{e}^{\eta y+\beta b+\alpha a+\xi x}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{-\xi\mathbf{x}}\mathbf{e}^{-\alpha\mathbf{a}}\mathbf{e}^{-\beta\mathbf{b}}\mathbf{e}^{-\eta\mathbf{y}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{-\xi\mathbf{x}}\mathbf{e}^{-\mathcal{A}\eta\mathbf{y}}\mathbf{e}^{-\alpha\mathbf{a}}\mathbf{e}^{-\beta\mathbf{b}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{-\xi\mathcal{A}\eta\mathbf{b}}\mathbf{e}^{-\mathcal{A}\eta\mathbf{y}}\mathbf{e}^{-\xi\mathbf{x}}\mathbf{e}^{-\alpha\mathbf{a}}\mathbf{e}^{-\beta\mathbf{b}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{-\mathcal{A}\eta\mathbf{y}}\mathbf{e}^{-\xi\mathcal{A}\eta\mathbf{b}}\mathbf{e}^{-\beta\mathbf{b}}\mathbf{e}^{-\xi\mathbf{x}}\mathbf{e}^{-\alpha\mathbf{a}}\right)\\
&= \mathbb{O}^{-1}\left(\mathbf{e}^{-\mathcal{A}\eta\mathbf{y}}\mathbf{e}^{-(\xi\mathcal{A}\eta+\beta)\mathbf{b}}\mathbf{e}^{-\alpha\mathbf{a}}\mathbf{e}^{-\xi\mathcal{A}\mathbf{x}}\right)
\end{aligned} \tag{3.23}$$

□

Notational conventions

The generating function of a tangle is not the entirety of this definition, for the additional data is the domain and codomain of the corresponding morphism.

We will thereby write a morphism with domain D, codomain C, and generating function $f(\zeta_D, z_C)$ as $f(\zeta_D, z_C)^D_C$.

$$Z(K_{3,1}) = \left(\frac{1}{B_1^{-1} + 1 + B_1^1}\right)^{\emptyset}_{\{1\}} = \Delta(K_{3,1})^{-1} \tag{3.24}$$

$$Z(K_{11a10}) = \left(\frac{1}{2B_1^{-3} - 11B_1^{-2} + 25B_1^{-1} - 31 + 25B_1 - 11B_1^2 + 2B_1^3}\right)^{\emptyset}_{\{1\}} = \Delta(K_{11a10})^{-1} \tag{3.25}$$

Since each tangle is expressed as an object, the domains in these examples are empty.

4

CONSTRUCTING THE TRACE

4.1 EXTENDING A PURE TANGLE INVARIANT TO LINKS AND GENERAL TANGLES

Thus far, the algebraic setting we have defined allows us to describe invariants of tangles with no closed components. We now extend the notion of a meta-Hopf algebra to include closed components.

Definition 4.1 (traced meta-algebra)**.** A traced meta-algebra is a family of meta-algebras: for each finite set L, we assign one meta-algebra $\{A_{L,S}\}_S$.[†1] The multiplication maps $m_k^{i,j}[L]$ then take the form:

$$m_k^{i,j}[L][S] \colon A_{\{i,j\}\sqcup S,L} \to A_{\{k\}\sqcup S,L} \tag{4.1}$$

for i, j, k disjoint from both S and L.

There is an additional structure, called a trace. The compatibility of the families of meta-algebras is governed this trace in the following way: $\mathrm{tr}^i \colon A_{\{i\}\sqcup S,L} \to A_{S,\{i\}\sqcup L}$ which satisfies the cyclic property:

$$m_k^{i,j} \,/\!\!/\, \mathrm{tr}^k = m_k^{j,i} \,/\!\!/\, \mathrm{tr}^k \tag{4.2}$$

The first example we give is that of impure tangles.

Definition 4.2 (Impure Rotational Virtual Tangles)**.** Let $\mathcal{T}^{\mathrm{rv}}_{L,S}$ be the set of rotational virtual tangles with open strands indexed by S and closed strands indexed by L. The operations $\phi[L][S]$ are defined analogously to the $\phi[S]$ given in theorem 2.28. (Here ϕ varies over m, η, Δ, ϵ, S, $\mathcal{R}$, and C.)

†1 These sets index the "strands" S and the "loops" L.

Lemma 4.3 (tangles as a traced algebra)**.** *The collection of all* $\mathcal{T}^{rv}_{L,S}$ *is a traced ribbon meta-Hopf algebra, with trace map given by closing a strand into a loop.*

Proof. When $L = \emptyset$, the situation is exactly the case of **??**, so $\mathcal{T}^{\mathrm{rv}}_{\emptyset,S} = \mathcal{T}_S$ is a meta-Hopf algebra. Furthermore, since the Reidemeister moves are local operations, the presence of closed components does not affect our ability to verify the identities on the Hopf-algebra operations.

The last point to verify is that closing a strand into a loop is a cyclic operation. Given two strands, we must verify that stitching one end together, then tracing the other yields the same diagram as stitching the other ends together, then taking the trace. However, by definition of trace, these two actions yield identical diagrams, the two strands replaced by the same closed loop. □

Lemma 4.4 (coinvariants as a trace map)**.** *Let A be an algebra, and denote by A_A its set of coinvariants. Then define $A_{S,L} := A^{\otimes S} \otimes A_A^{\otimes L}$. Then A defines a traced meta-algebra with trace map given by $\mathrm{tr}^i_j \colon A_i \to (A_A)_j$.*

Proof. Observe that for any choice of L, extending morphisms by the identity yield an isomorphism of traced meta-Hopf algebras:

$$\begin{aligned} \phi_L \colon \{A^{\otimes S}\}_S &\xrightarrow{\sim} \{A^{\otimes S} \otimes A_A^{\otimes L}\}_S \\ A^{\otimes S} &\mapsto A^{\otimes S} \otimes A_A^{\otimes L} \\ f &\mapsto f \otimes \mathrm{id}_{A_A}^{\otimes L} \end{aligned} \tag{4.3}$$

Next, we wish to show that the two maps with shape $f \colon A^{\otimes\{i,j\}\sqcup S} \otimes A_A^{\otimes L} \to A^{\otimes S} \otimes A_A^{\otimes\{k\}\sqcup L}$ are equivalent. This amounts to showing that, given $u, v \in A$, that $\overline{uv} = \overline{vu} \in A_A$. However, by the construction of the coinvariants, $\overline{uv} - \overline{vu} = \overline{uv - vu} = \overline{0} \in A$, and we are done. □

4.2 THE COINVARIANTS OF U

We start with a result which simplifies working with coinvariants:

Lemma 4.5 (Coinvariant simplification)**.** *Let $\mathfrak{h}$ be a Lie algebra. Then* $\mathfrak{U}(\mathfrak{h})_{\mathfrak{U}(\mathfrak{h})} = \mathfrak{U}(\mathfrak{h})_{\mathfrak{h}}$.

Proof. First, observe that for any u, v, $f \in \mathfrak{U}(\mathfrak{h})$, $\mathrm{ad}_{uv}(f) = \mathrm{ad}_u(vf) + \mathrm{ad}_v(fu)$. Proceeding inductively, for any monomial $\mu \in \mathfrak{U}(\mathfrak{h})$, $\mathrm{ad}_\mu(u)$ is a linear combination of elements of $[\mathfrak{h}, \mathfrak{U}(\mathfrak{h})]$. By linearity of ad, we conclude $[\mathfrak{U}(\mathfrak{h}), \mathfrak{U}(\mathfrak{h})] = [\mathfrak{h}, \mathfrak{U}(\mathfrak{h})]$. □

Theorem 4.6. *The coinvariants of U, U_U, has basis $\{y^n a^k x^n\}_{n,k\geq 0}$.*

Proof. Using lemma 4.5, we need only compute $[\mathfrak{g}, U]$ to determine U_U. Given a polynomial f, we have the following relations in U:

$$f(a)y^r = y^r f(a-r) \qquad x^r f(a) = f(a-r)x^r \tag{4.4}$$

Next we compute the adjoint actions of y, a, and x. (Recall b is central.)

$$\mathrm{ad}_a\, f(x) = xf'(x) \qquad \mathrm{ad}_a\, f(y) = -yf'(y) \tag{4.5}$$

$$\mathrm{ad}_x\, f(y) = bf'(y) \qquad \mathrm{ad}_x\, f(a) = -\nabla[f](a)x \tag{4.6}$$

$$\mathrm{ad}_y\, f(x) = -bf'(x) \qquad \mathrm{ad}_y\, f(a) = y\nabla[f](a) \tag{4.7}$$

(Here ∇ is the backwards finite difference operator $\nabla[f](x) := f(x) - f(x-1)$.) Observe for any n, m, k, and polynomials f and g:

$$\mathrm{ad}_a(y^m g(b,a)x^n) = (n-m)y^m g(b,a)x^n \tag{4.8}$$

$$\mathrm{ad}_x(y^{n+1}b^{m-1}f(a)x^k) = (n+1)y^n b^m f(a)x^k - y^{n+1}b^{m-1}\nabla[f](a)x^{k+1} \tag{4.9}$$

$$\mathrm{ad}_y(y^n b^{m-1}f(a)x^{k+1}) = -(k+1)y^n b^m f(a)x^k + y^{n+1}b^{m-1}\nabla[f](a)x^{k+1} \tag{4.10}$$

By equation (4.8), any monomial whose powers of y and x differ vanish in $U_\mathfrak{g}$. As a consequence, in equations (4.9) and (4.10), the only nontrivial case is when $n = k$, resulting in the same relation. By induction on n, we conclude that:

$$y^n b^m f(a)x^k \sim \delta_{nk}\frac{n!}{(n+m)!}y^{n+m}\nabla^m[f](a)x^{n+m} \tag{4.11}$$

Observing when f is a monomial in equation (4.11), we see $U_\mathfrak{g}$ is spanned by $\{y^n a^k x^n\}_{n,k\geq 0}$. Since all relations are accounted for, setting $m = 0$ demonstrates this set is linearly independent, and we have a basis. □

A generating function for the coinvariants

In order to define a generating function, we need to choose an appropriate basis for the space of coinvariants. We define an isomorphism from the space of coinvariants to a polynomial space, tweaking the earlier-defined basis by scalar multiples. Since it plays the role of the ordering map, we also name it $\mathbb{O}$.

$$\begin{aligned} \mathbb{O} \colon \mathbb{Q}[a,z] &\xrightarrow{\sim} U_U \\ a^n z^k &\mapsto \frac{1}{k!} y^k a^n x^k \\ k! \nabla^m[f](a) z^{k+m} &\mapsfrom y^k b^m f(a) x^k \end{aligned} \tag{4.12}$$

This defines a commutative square upon whose bottom edge $\tau = \mathbb{O} \,/\!\!/\, \mathrm{tr} \,/\!\!/\, \mathbb{O}^{-1}$ we compute the generating function:

$$\begin{tikzcd} U \arrow[r, "\mathrm{tr}"] & U_U \\ \mathbb{Q}[y,b,a,x] \arrow[u, "\mathbb{O}"] \arrow[r, "\tau"] & \mathbb{Q}[a,z] \arrow[u, "\mathbb{O}"] \end{tikzcd} \tag{4.13}$$

We begin with a result on finite differences:

Lemma 4.7 (finite differences of exponentials)**.** *The finite difference operator acts in the following way on exponentials:*

$$\nabla^n[\mathbf{e}^{\alpha a}](a) = (1 - \mathbf{e}^{-\alpha})^n \mathbf{e}^{\alpha a} \tag{4.14}$$

Proof. Using the fact that $\nabla^n[f](x) = \sum_{k=0}^n \binom{n}{k}(-1)^k f(x-k)$, we see that $\nabla^n[\mathbf{e}^{\alpha a}](a) = \sum_{k=0}^n \binom{n}{k}(-1)^k \mathbf{e}^{\alpha a - \alpha k} = (1-\mathbf{e}^{-\alpha})^n \mathbf{e}^{\alpha a}$. □

We now are ready to compute the generating function for the trace:

Theorem 4.8 (Generating function for the trace of U)**.**

$$\mathcal{G}(\mathrm{tr}) = \exp\Big(\alpha a + \big(\eta\xi + \beta(1 - \mathbf{e}^{-\alpha})\big)z\Big) \tag{4.15}$$

Proof. Using lemma 4.7 and the extension of scalars of tr to $\mathbb{Q}[\![\eta,\beta,\alpha,\xi]\!]$, we see

$$\begin{aligned}
&\mathcal{G}(\mathbb{O} \mathbin{/\!/} \operatorname{tr} \mathbin{/\!/} \mathbb{O}^{-1}) = (\mathbf{e}^{\eta y}\mathbf{e}^{\beta b}\mathbf{e}^{\alpha a}\mathbf{e}^{\xi x}) \mathbin{/\!/} \operatorname{tr} \mathbin{/\!/} \mathbb{O}^{-1} \\
&= \mathbb{O}^{-1} \sum_{i,j,k} \operatorname{tr}\left(\frac{(\eta y)^i}{i!}\frac{(\beta b)^j}{j!}\mathbf{e}^{\alpha a}\frac{(\xi x)^k}{k!}\right) \\
&= \sum_{i,j} \frac{\eta^i\beta^j\xi^i}{i!j!}(1-\mathbf{e}^{-\alpha})^j\mathbf{e}^{\alpha a}z^{i+j} = \mathbf{e}^{\alpha a+(\eta\xi+\beta(1-\mathbf{e}^{-\alpha}))z} \qquad \square
\end{aligned} \tag{4.16}$$

Evaluation of the trace on a generic element

Here we will outline a computation involving the trace by using Bar-Natan and van der Veen's Contraction Theorem.

A typical value for a tangle invariant that arises is of the form:

$$P\mathbf{e}^{c+\alpha a_i+\beta b_i+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_iy_i} \tag{4.17}$$

Here, c, α, and β denote constants with respect to the variables y_i, b_i, a_i, and x_i (collectively referred to as "v_i"s), while ξ, η, and λ are potentially b_i-dependent, and P is a (rational) function in (the square root of) B_i.

Theorem 4.9 (The trace of a Gaußian)**.** *With symbols as defined above, let* $f(y_i,b_i,a_i,x_i) = P(B_i)\mathbf{e}^{c+\alpha a_i+\beta b_i+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_iy_i}$. *Then*

$$\left\langle f(y_i,b_i,a_i,x_i)\operatorname{tr}^i\right\rangle_{v_i} = \frac{P(\mathbf{e}^{-\mu})}{1-\lambda(\mu)\bar{z}_i}\mathbf{e}^{c+\alpha\bar{a}_i+\beta\mu+\frac{\eta(\mu)\xi(\mu)\bar{z}_i}{1-\lambda(\mu)\bar{z}_i}} \tag{4.18}$$

where $\mu := (1-\mathbf{e}^{-\alpha})\bar{z}_i$.

Proof. Let us compute the trace of equation (4.17). For clarity, we will put bars over the coinvariants variables a_i and z_i, as they do not play a role in the contraction.

$$
\begin{aligned}
&\langle P(B_i)\mathbf{e}^{c+\alpha a_i+\beta b_i+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_iy_i}\,\mathrm{tr}^i\rangle_{v_i}\\
&= \langle P(B_i)\mathbf{e}^{c+\beta b_i+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_iy_i+\eta_i\xi_i\bar{z}_i+\beta_i(1-\mathbf{e}^{-\alpha_i})\bar{z}_i}\mathbf{e}^{\alpha a_i+\alpha_i\bar{a}_i}\rangle_{v_i}\\
&= \mathbf{e}^{\alpha\bar{a}_i}\langle P(B_i)\mathbf{e}^{c+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_iy_i+\eta_i\xi_i\bar{z}_i}\mathbf{e}^{\beta b_i+\beta_i(1-\mathbf{e}^{-\alpha})\bar{z}_i}\rangle_{b_i,x_i,y_i}
\end{aligned}
$$

In what follows, we let $\mu := (1-\mathbf{e}^{-\alpha})\bar{z}_i$:

$$
\begin{aligned}
&= \mathbf{e}^{c+\alpha\bar{a}_i+\beta\mu}P(\mathbf{e}^{-\mu})\langle \mathbf{e}^{\eta(\mu)y_i}\mathbf{e}^{(\xi(\mu)+\lambda(\mu)y_i)x_i+\xi_i\eta_i\bar{z}_i}\rangle_{x_i,y_i}\\
&= \mathbf{e}^{c+\alpha\bar{a}_i+\beta\mu}P(\mathbf{e}^{-\mu})\langle \mathbf{e}^{\eta(\mu)y_i+\xi(\mu)\bar{z}_i\eta_i+\lambda(\mu)\bar{z}_i\eta_iy_i}\rangle_{y_i}\\
&= \frac{P(\mathbf{e}^{-\mu})}{1-\lambda(\mu)\bar{z}_i}\mathbf{e}^{c+\alpha\bar{a}_i+\beta\mu+\frac{\eta(\mu)\xi(\mu)\bar{z}_i}{1-\lambda(\mu)\bar{z}_i}}
\end{aligned}
\tag{4.19}
$$

□

We point out that the outcome of this computation is not guaranteed to be a Gaußian. This puts a limitation on the applicability of this formula to links with more than two components, explored in chapter 5.

Computational examples

Using the formula given in equation (4.18), let us do some preliminary examples:

$$\mathrm{tr}^i(R_{ij}) = 1 \tag{4.20}$$

$$\mathrm{tr}^j(R_{ij}) = \mathbf{e}^{b_i\bar{a}_j} \tag{4.21}$$

$$
\begin{aligned}
&\mathrm{tr}^2\left(\sqrt{B_2}\mathbf{e}^{-a_2b_1-a_1b_2+\frac{(B_1-1)x_2y_1}{b_1B_1}+\frac{(B_2-1)x_1y_2}{b_2B_2}}\right) =\\
&\mathbf{e}^{\frac{a_1(\bar{z}_2-B_1\bar{z}_2)}{B_1}-b_1\bar{a}_2+\frac{e^{B_1\bar{z}_2}(x_1y_1e^{B_1^{-1}\bar{z}_2}-x_1y_1e^{\bar{z}_2})}{b_1}+\frac{1}{2}B_1^{-1}\bar{z}_2-\frac{\bar{z}_2}{2}}
\end{aligned}
\tag{4.22}
$$

Equations (4.20) and (4.21) are the values one obtains for the two (virtual) one-crossing, two-component link, while equation (4.22) is the value of the invariant on the Hopf link.

When computing this on a link, however, it is important to keep track of which strands are open, and which are closed. We will extend the notation from the previous section to differentiate between open and closed indices. We write a morphism with domain $D = D_\mathrm{o} \sqcup D_\mathrm{c}$, codomain $C = C_\mathrm{o} \sqcup C_\mathrm{c}$ (here D_o denotes domain indices which are open, while D_c those which are closed, with the same convention for C) and generating function $f(\zeta_D, z_C)$ as $f(\zeta_D, z_C)^{(D_\mathrm{o}, D_\mathrm{c})}_{(C_\mathrm{o}, C_\mathrm{c})}$.

5 CONCLUSIONS

Limitations of this definition

For some inputs to the trace, expressions involving the Lambert W-function appear, which complicates attempts to keep the invariant valued in the space of perturbed Gaußians.

5.1 COMPARISON WITH THE MULTIVARIABLE ALEXANDER POLYNOMIAL

Given that the long knot (i.e. one-component) case of this invariant encodes the Alexander Polynomial, it was suspected that the invariant on long links (i.e. multiple components, one of which is long) formed by adding the trace would encode the MVA. However, there are links which the MVA separates which this invariant does not.

On all two-component links with at most 11 crossings (a collection of size 914), the trace map attains 878 distinct values, while the MVA attains only 778. However, the two invariants are incomparable in terms of their strength.

The links L_{5a1} and L_{10a43} are not distinguished by their partial traces, with both returning a value of:

$$\left(\left(\frac{B_1}{B_1^2 t_2 - 2B_1 t_2 + B_1 + t_2} \right)_{(\{1\},\{2\})}, \left(\frac{B_2^{3/2}}{B_2^2 t_1 - 2B_2 t_1 + B_2 + t_1} \right)_{(\{2\},\{1\})} \right) \tag{5.1}$$

The values of these links under the MVA are, however

$$\frac{(B_1-1)(B_2-1)}{\sqrt{B_1}\sqrt{B_2}} \text{ and } -\frac{(B_1-1)(B_2-1)(B_1+B_2-1)(B_2B_1-B_1-B_2)}{B_1^{3/2}B_2^{3/2}} \tag{5.2}$$

respectively.

In the other direction, there are also pairs of links in the same fibre of the MVA which this traced invariant can distinguish. In particular L_{5a1} and L_{7n2} both have the same value under the MVA:

$$\frac{(B_1-1)(B_2-1)}{\sqrt{B_1}\sqrt{B_2}} \tag{5.3}$$

The trace yields the following values (respectively):

$$\left(\left(\frac{B_1}{B_1^2t_2-2B_1t_2+B_1+t_2}\right)_{(\{1\},\{2\})},\left(\frac{B_2^{3/2}}{B_2^2t_1-2B_2t_1+B_2+t_1}\right)_{(\{2\},\{1\})}\right) \tag{5.4}$$

$$\left(\left(\frac{B_1}{B_1^2t_2-2B_1t_2+B_1+t_2}\right)_{(\{1\},\{2\})},\left(\frac{B_2^{5/2}}{B_2^2t_1-2B_2t_1+B_2^2-B_2+t_1+1}\right)_{(\{2\},\{1\})}\right) \tag{5.5}$$

This example also serves to highlight that the information provided by leaving one strand open is not enough to recover the value of a different strand being left open.

5.2 FURTHER WORK

While all other Hopf algebra operations in U are expressed by [BNvdV] as perturbed Gaußians, the form in equation (4.15) does not to conform to the same structure. Further work is needed to either implement this operation into the established framework, or to suitably extend the framework (perhaps with the use of Lambert W-functions).

A CODE

A.1 IMPLEMENTATION OF THE INVARIANT Z

This is a Mathematica™ implementation by Bar-Natan and van der Veen in [BNvdV], modified by the author. We begin by setting some variables, as well as a method for modifying associations.

```
γ = 1; ℏ = 1; $k = 0;
setValue[value_,obj_,coord_]:=Module[
        {b=Association@@obj},
        b[coord] = value; Head[obj]@@Normal@b
]
```

We introduce notation `PG[L, Q, P]` to be interpreted as the Perturbed Gaußian $P\mathrm{e}^{L+Q}$. The function `fromE` serves as a compatibility layer between a former version of the code.

```
toPG[L_, Q_, P_] := PG["L"->L, "Q"->Q, "P"->P]
fromE[e_\[DoubleStruckCapitalE]] := toPG@@e/.
        Subscript[(v:y|b|t|a|x|B|T|η|β|τ|α|ξ|A), i_] -> v[i]
```

We define the Kronecker-δ function next.

```
δ[i_,j_] := If[SameQ[i,j],1,0]
```

Next we introduce helper functions for the reading and manipulating of `PG`-objects:

```
getL[pg_PG] := Lookup[Association@@pg,"L",0]
getQ[pg_PG] := Lookup[Association@@pg,"Q",0]
getP[pg_PG] := Lookup[Association@@pg,"P",1]

setL[L_][pg_PG] := setValue[L, pg, "L"];
setQ[Q_][pg_PG] := setValue[Q, pg, "Q"];
```

```
setP[P_][pg_PG] := setValue[P, pg, "P"];

applyToL[f_][pg_PG] := pg//setL[pg//getL//f]
applyToQ[f_][pg_PG] := pg//setQ[pg//getQ//f]
applyToP[f_][pg_PG] := pg//setP[pg//getP//f]
```

Next is a function CF, which bring objects into canonical form allows us to compare for equality effectively. This is defined by Bar-Natan and van der Veen.

```
CCF[e_] := ExpandDenominator@ExpandNumerator@Together[
        Expand[e] //. E^x_ E^y_ :> E^(x + y) /. E^x_ :>
         ↪  E^CCF[x]
];
CF[sd_SeriesData] := MapAt[CF, sd, 3];
CF[e_] := Module[
        {vs = Union[
                Cases[e, (y|b|t|a|x|η|β|τ|α|ξ)[_], ∞],
                {y, b, t, a, x, η, β, τ, α, ξ}
        ]},
        Total[CoefficientRules[Expand[e], vs] /.
                (ps_ -> c_) :> CCF[c] (Times @@ (vs^ps))
        ]
];
CF[e_PG] := e//applyToL[CF]//applyToQ[CF]//applyToP[CF]
```

We must also define the notion of equality for PG-objects, as well as what it means to multiply them.

```
Congruent[x_, y_, z__] := And[Congruent[x, y], Congruent[y, z]]
PG /: Congruent[pg1_PG, pg2_PG] := And[
        CF[getL@pg1 == getL@pg2],
        CF[getQ@pg1 == getQ@pg2],
        CF[Normal[getP@pg1-getP@pg2] == 0]
]

PG /: pg1_PG * pg2_PG := toPG[
        getL@pg1 + getL@pg2,
        getQ@pg1 + getQ@pg2,
```

```
        getP@pg1 * getP@pg2
]

setEpsilonDegree[k_Integer][pg_PG] :=
 ↪ setP[Series[Normal@getP@pg,{ϵ, 0, k}]][pg]
```

The variables y, b, t, a, and x are paired with their dual variables η, β, τ, α, and ξ. This applies as well when they have subscripts.

```
ddsl2vars = {y, b, t, a, x, z};
ddsl2varsDual = {η, β, τ, α, ξ, ζ};

Evaluate[Dual/@ddsl2vars] = ddsl2varsDual;
Evaluate[Dual/@ddsl2varsDual] = ddsl2vars;
Dual@z = ζ;
Dual@ζ = z;
Dual[u_[i_]]:=Dual[u][i]
```

Since various exponentials of the lowercase variables frequently appear, we introduce capital variable names to handle various exponentiated forms.

```
U2l = {
        B[i_]^p_. :> E^(-p ℏ γ b[i]), B^p_. :> E^(-p ℏ γ b),
        W[i_]^p_. :> E^(w[i])       , W^p_. :> E^(p w),
        T[i_]^p_. :> E^(-p ℏ t[i])  , T^p_. :> E^(-p ℏ t),
        A[i_]^p_. :> E^(p γ α[i])   , A^p_. :> E^(-p γ α)
};
l2U = {
        E^(c_. b[i_] + d_.) :> B[i]^(-c/(ℏ γ))E^d,
        E^(c_. b + d_.)     :> B^(-c/(ℏ γ))E^d,
        E^(c_. t[i_] + d_.) :> T[i]^(-c/ℏ)E^d,
        E^(c_. t + d_.)     :> T^(-c/ℏ)E^d,
        E^(c_. α[i_] + d_.) :> A[i]^(c/γ)E^d,
        E^(c_. α + d_.)     :> A^(c/γ)E^d,
        E^(c_. w[i_] + d_.) :> W[i]^(c)E^d,
        E^(c_. w + d_.)     :> W^(c)E^d,
        E^expr_             :> E^Expand@expr
};
```

Below the notion of differentiation is defined for expressions which involve both upper- and lower-case variables.

```
DD[f_, b]     := D[f, b   ] - ℏ γ B    D[f, B   ];
DD[f_, b[i_]] := D[f, b[i]] - ℏ γ B[i] D[f, B[i]];

DD[f_, t    ] := D[f, t   ] - ℏ T    D[f, T   ];
DD[f_, t[i_]] := D[f, t[i]] - ℏ T[i] D[f, T[i]];

DD[f_, α    ] := D[f, α   ] + γ A    D[f, A   ];
DD[f_, α[i_]] := D[f, α[i]] + γ A[i] D[f, A[i]];

DD[f_, v_] := D[f, v];
DD[f_, {v_,0}] := f;
DD[f_, {}] := f;
DD[f_, {v_,n_Integer}] := DD[DD[f,v],{v,n-1}];
DD[f_, {l_List, ls___}] := DD[DD[f, l], {ls}];
```

What follows now is the implementation of contraction as introduced in definition 3.4. We begin with the introduction of contractions of (finite) polynomials.

```
collect[sd_SeriesData, ζ_] := MapAt[collect[#, ζ] &, sd, 3];
collect[expr_, ζ_] := Collect[expr, ζ];

Zip[{}][P_] := P;
Zip[ζs_List][Ps_List] := Zip[ζs]/@Ps;
Zip[{ζ_,ζs___}][P_] := (collect[P // Zip[{ζs}],ζ] /.
        f_. ζ^d_. :> DD[f,{Dual[ζ], d}]) /.
        Dual[ζ] -> 0 /.
        ((Dual[ζ] /. {b->B, t->T, α -> A}) -> 1)
```

We define contraction along the variables x and y (here packaged into the matrix `Q`).

```
QZip[ζs_List][pg_PG] := Module[{Q, P, ζ, z, zs, c, ys, ηs, qt,
 ↪  zrule, ζrule},
        zs = Dual/@ζs;
        Q  = pg//getQ;
```

```
        P  = pg//getP;
        c  = CF[Q/.Alternatives@@Union[ζs, zs]->0];
        ys = CF/@Table[D[Q,ζ]/.Alternatives@@zs->0,{ζ,ζs}];
        ηs = CF/@Table[D[Q,z]/.Alternatives@@ζs->0,{z,zs}];
        qt = CF/@#&/@(Inverse@Table[
                δ[z, Dual[ζ]] - D[Q,z,ζ],
                {ζ,ζs},{z,zs}
        ]);
        zrule = Thread[zs -> CF/@(qt . (zs + ys))];
        ζrule = Thread[ζs -> ζs + ηs . qt];
        CF@setQ[c + ηs.qt.ys]@setP[Det[qt] Zip[ζs][P /.
         ↪  Union[zrule, ζrule]]]@pg
]
```

We define contraction along the variables a and b (here packaged into the matrix L).

```
LZip[ζs_List][pg_PG] := Module[
        {
                L, Q, P, ζ, z, zs, Zs, c, ys, ηs, lt,
                zrule, Zrule, ζrule, Q1, EEQ, EQ, U
        },
        zs = Dual/@ζs;
        {L, Q, P} = Through[{getL, getQ, getP}@pg];
        Zs = zs /. {b -> B, t -> T, α -> A};
        c  = CF[L/.Alternatives@@Union[ζs,
         ↪  zs]->0/.Alternatives@@Zs -> 1];
        ys = CF/@Table[D[L,ζ]/.Alternatives@@zs->0,{ζ,ζs}];
        ηs = CF/@Table[D[L,z]/.Alternatives@@ζs->0,{z,zs}];
        lt = CF/@#&/@Inverse@Table[
                δ[z, Dual[ζ]] - D[L,z,ζ],
                {ζ,ζs},{z,zs}
        ];
        zrule = Thread[zs -> CF/@(lt . (zs + ys))];
        Zrule = Join[zrule, zrule /.
                r_Rule :> ( (U = r[[1]] /. {b -> B, t -> T, α
                 ↪  -> A}) ->
                (U /. U2l /. r //. l2U))
```

```
        ];
        \[Zeta]rule = Thread[\[Zeta]s -> \[Zeta]s + \[Eta]s . lt];
        Q1 = Q /. Union[Zrule, ζrule];
        EEQ[ps___] :=
                EEQ[ps] = (
                        CF[E^-Q1 DD[E^Q1,Thread[{zs,{ps}}]] /.
                                {Alternatives@@zs -> 0, Alternatives @@Zs -> 1}]
                );
        CF@toPG[
                c + ηs.lt.ys,
                Q1 /. {Alternatives@@zs -> 0, Alternatives@@Zs -> 1},
                Det[lt] (Zip[ζs][(EQ@@zs) (P /. Union[Zrule,ζrule])] /.
                        Derivative[ps___][EQ][___] :> EEQ[ps] /. _EQ ->1
                )
        ]

]
```

The function Pair combines the above zipping functions into the final contraction map.

```
Pair[{}][L_PG,R_PG] := L R;
Pair[is_List][L_PG,R_PG] := Module[{n},
        Times[
                L /. ((v: b|B|t|T|a|x|y)[#] -> v[n@#]&/@is),
                R /. ((v: β|τ|α|A|ξ|η)[#] -> v[n@#]&/@is)
        ] // LZip[Join@@Table[Through[{β, τ, a}[n@i]],{i, is}]] //
        QZip[Join@@Table[Through[{ξ, y}[n@i]],{i, is}]]
]
```

Our next task is to provide domain and codomain information for the PG-objects. These will be packaged inside a GDO, (Gaußian Differential Operator).

The four lists' names refer to whether it is a domain or a codomain, and whether the index corresponds to an open strand or a closed one.

```
toGDO[do_List,dc_List,co_List,cc_List,L_,Q_,P_] := GDO[
        "do" -> do,
        "dc" -> dc,
        "co" -> co,
        "cc" -> cc,
        "PG" -> toPG[L, Q, P]
]

toGDO[do_List,dc_List,co_List,cc_List,pg_PG] := GDO[
        "do" -> do,
        "dc" -> dc,
        "co" -> co,
        "cc" -> cc,
        "PG" -> pg
]
```

Next are defined functions for accessing and modifying sub-parts of `GDO`-objects. The last argument of `Lookup` is the default value if nothing is specified. This means that a morphism with empty domain or codomain may be specified as such by omitting that portion of the definition.

```
getDO[gdo_GDO] := Lookup[Association@@gdo, "do", {}]
getDC[gdo_GDO] := Lookup[Association@@gdo, "dc", {}]
getCO[gdo_GDO] := Lookup[Association@@gdo, "co", {}]
getCC[gdo_GDO] := Lookup[Association@@gdo, "cc", {}]

getPG[gdo_GDO] := Lookup[Association@@gdo, "PG", PG[]]

getL[gdo_GDO] := gdo//getPG//getL
getQ[gdo_GDO] := gdo//getPG//getQ
getP[gdo_GDO] := gdo//getPG//getP

setPG[pg_PG][gdo_GDO] := setValue[pg, gdo, "PG"]

setL[L_][gdo_GDO] := setValue[setL[L][gdo//getPG], gdo, "PG"]
```

```
setQ[Q_][gdo_GDO] := setValue[setQ[Q][gdo//getPG], gdo, "PG"]
setP[P_][gdo_GDO] := setValue[setP[P][gdo//getPG], gdo, "PG"]

setDO[do_][gdo_GDO] := setValue[do, gdo, "do"]
setDC[dc_][gdo_GDO] := setValue[dc, gdo, "dc"]
setCO[co_][gdo_GDO] := setValue[co, gdo, "co"]
setCC[cc_][gdo_GDO] := setValue[cc, gdo, "cc"]

applyToDO[f_][gdo_GDO] := gdo//setDO[gdo//getDO//f]
applyToDC[f_][gdo_GDO] := gdo//setDC[gdo//getDC//f]
applyToCO[f_][gdo_GDO] := gdo//setCO[gdo//getCO//f]
applyToCC[f_][gdo_GDO] := gdo//setCC[gdo//getCC//f]

applyToPG[f_][gdo_GDO] := gdo//setPG[gdo//getPG//f]

applyToL[f_][gdo_GDO] := gdo//setL[gdo//getL//f]
applyToQ[f_][gdo_GDO] := gdo//setQ[gdo//getQ//f]
applyToP[f_][gdo_GDO] := gdo//setP[gdo//getP//f]
```

The canonical form function (`CF`) and the contraction mapping (`Pair`) we extend to include `GDO`-objects. Furthermore, on the level of `GDO`-objects we can compose morphisms and keep track of the corresponding domains and codomains, using the left-to-right composition operator "//".

```
CF[e_GDO] := e//
        applyToDO[Union]//
        applyToDC[Union]//
        applyToCO[Union]//
        applyToCC[Union]//
        applyToPG[CF]

Pair[is_List][gdo1_GDO, gdo2_GDO] := GDO[
        "do" -> Union[gdo1//getDO, Complement[gdo2//getDO,
         ↪  is]],
        "dc" -> Union[gdo1//getDC, gdo2//getDC],
        "co" -> Union[gdo2//getCO, Complement[gdo1//getCO,
         ↪  is]],
        "cc" -> Union[gdo1//getCC, gdo2//getCC],
```

```
        "PG" -> Pair[is][gdo1//getPG, gdo2//getPG]
]

gdo1_GDO // gdo2_GDO :=
 ↪  Pair[Intersection[gdo1//getCO,gdo2//getDO]][gdo1,gdo2];
```

We also define notions of equality and multiplication (by concatenation) for GDO's.

```
GDO /: Congruent[gdo1_GDO, gdo2_GDO] := And[
        Sort@*getDO/@Equal[gdo1, gdo2],
        Sort@*getDC/@Equal[gdo1, gdo2],
        Sort@*getCO/@Equal[gdo1, gdo2],
        Sort@*getCC/@Equal[gdo1, gdo2],
        Congruent[gdo1//getPG, gdo2//getPG]
]

GDO /: gdo1_GDO gdo2_GDO := GDO[
        "do" -> Union[gdo1//getDO, gdo2//getDO],
        "dc" -> Union[gdo1//getDC, gdo2//getDC],
        "co" -> Union[gdo1//getCO, gdo2//getCO],
        "cc" -> Union[gdo1//getCC, gdo2//getCC],
        "PG" -> (gdo1//getPG)*(gdo2//getPG)
]
```

For the sake of compatibility with Bar-Natan and van der Veen's program, we introduce several conversion functions between the two notations.

```
setEpsilonDegree[k_Integer][gdo_GDO] :=
        setP[Series[Normal@getP@gdo,{ϵ,0,k}]][gdo]

fromE[Subscript[\[DoubleStruckCapitalE],{do_List,
 ↪  dc_List}->{co_List, cc_List}][
        L_, Q_, P_
]] := toGDO[do, dc, co, cc, fromE[\[DoubleStruckCapitalE][L, Q,
 ↪  P]]]

fromE[Subscript[\[DoubleStruckCapitalE], dom_List->cod_List][
        L_, Q_, P_
```

```
]] := GDO["do" -> dom, "co" -> cod,
        "PG" -> fromE[\[DoubleStruckCapitalE][L, Q, P]]
]
```

It is at this point that we implement the morphisms of the algebra U. Each operation is prepended with a “c” to emphasize that this is a classical algebra, not a quantum deformation.

```
fromLog[l_] := CF@Module[
        {L, l0 = Limit[l, ϵ->0]},
        L = l0 /. (η|y|ξ|x)[_]->0;
        PG[
                "L" -> L,
                "Q" -> l0 - L
        ]/.l2U
]

cΛ = (η[i] + E^(-γ α[i] - ϵ β[i]) η[j]/(1+γ ϵ η[j]ξ[i])) y[k] +
     (β[i] + β[j] + Log[1 + γ ϵ η[j]ξ[i]]/ϵ            ) b[k] +
     (α[i] + α[j] + Log[1 + γ ϵ η[j]ξ[i]]/γ            ) a[k] +
     (ξ[j] + E^(-γ α[j] - ϵ β[j]) ξ[i]/(1+γ ϵ η[j]ξ[i])) x[k];

cm[i_, j_, k_] = GDO["do" -> {i,j}, "co" -> {k}, "PG" ->
 ↪  fromLog[cΛ]];

cη[i_] = GDO["co" -> {i}];
cσ[i_,j_] = GDO["do"->{i},"co"->{j},
        "PG"->fromLog[β[i] b[j] + α[i] a[j] + η[i] y[j] + ξ[i]
         ↪  x[j]]
];
cϵ[i_] = GDO["do" -> {i}];
cΔ[i_, j_, k_] = GDO["do"->{i}, "co"->{j, k},
        "PG" -> fromLog[
                β[i](b[j] + b[k]) +
                α[i](a[j] + a[k]) +
                η[i](y[j] + y[k]) +
                ξ[i](x[j] + x[k])
        ]
```

```
];

sY[i_, j_, k_, l_, m_] = GDO["do"->{i}, "co"->{j, k, l, m},
         "PG" -> fromLog[β[i]b[k] + α[i]a[l] + η[i]y[j] + ξ[i]x[m]]
];

sS[i_] = GDO["do"->{i},"co"->{i},
         "PG"->fromLog[-(β[i] b[i] + α[i] a[i] + η[i] y[i] + ξ[i] x[i])]
];

cS[i_] = sS[i] // sY[i, 1, 2, 3, 4] // cm[4,3, i] // cm[i, 2, i] // cm[i, 1, i];

cR[i_, j_] = GDO[
         "co" -> {i,j},
         "PG" -> toPG[ℏ a[j] b[i], (B[i]-1)/(-b[i]) x[j] y[i], 1]
]

cRi[i_, j_] = GDO[
         "co" -> {i,j},
         "PG" -> toPG[-ℏ a[j] b[i], (B[i]-1)/(B[i] b[i]) x[j] y[i], 1]
]

CC[i_] :=  GDO["co"->{i},"PG"->PG["P"->B[i]^( 1/2)]]
CCi[i_] := GDO["co"->{i},"PG"->PG["P"->B[i]^(-1/2)]]

cKink[i_]  = Module[{k}, cR[i,k] CCi[k] // cm[i, k, i]]
cKinki[i_] = Module[{k}, cRi[i,k] CC[k] // cm[i, k, i]]

cKinkn[0][i_]  = cη[i]
cKinkn[1][i_]  = cKink[i]
cKinkn[-1][i_] = cKinki[i]
```

```
cKinkn[n_Integer][i_] := Module[{j},cKinkn[n-1][i]cKink[j]//cm[i,j,i]]/; n > 1
cKinkn[n_Integer][i_] := Module[{j},cKinkn[n+1][i]cKinki[j]//cm[i,j,i]]/; n < -1

uR[i_, j_]  = Module[{k}, cR[i,j] cKinki[k] // cm[i, k, i]]
uRi[i_, j_] = Module[{k}, cRi[i,j] cKink[k] // cm[i, k, i]]
```

A.2 IMPLEMENTATION OF THE TRACE

Now we implement the trace. We introduce several functions which extract the various coefficients of a `GDO`, so that we may apply equation (4.18). Coefficients are extracted based on whether they belong to the matrix `L` or the matrix `Q`.

```
getConstLCoef::usage = "getConstLCoef[i][gdo] returns the terms in the L-portion of a GDO expression which are not a function of y[i], b[i], a[i], nor x[i]."
getConstLCoef[i_][gdo_] :=
        (SeriesCoefficient[#, {b[i],0,0}]&) @*
        (Coefficient[#, y[i], 0]&) @*
        (Coefficient[#, a[i], 0]&) @*
        (Coefficient[#, x[i], 0]&) @*
        ReplaceAll[U2l] @*
        getL@
        gdo

getConstQCoef::usage = "getConstQCoef[i][gdo] returns the terms in the Q-portion of a GDO expression which are not a function of y[i], b[i], a[i], nor x[i]."
getConstQCoef[i_][gdo_][bb_] :=
        ReplaceAll[{b[i]->bb}] @*
        (Coefficient[#, y[i], 0]&) @*
        (Coefficient[#, a[i], 0]&) @*
        (Coefficient[#, x[i], 0]&) @*
        ReplaceAll[U2l] @*
        getQ@
```

```
        gdo

getyCoef::usage = "getyCoef[i][gdo][b[i]] returns the linear coefficient of y[i] as a function of b[i]."
getyCoef[i_][gdo_][bb_] :=
        ReplaceAll[{b[i]->bb}] @*
        ReplaceAll[U2l] @*
        (Coefficient[#, x[i],0]&) @*
        (Coefficient[#, y[i],1]&) @*
        getQ@
        gdo

getbCoef::usage = "getbCoef[i][gdo] returns the linear coefficient of b[i]."
getbCoef[i_][gdo_] :=
        (SeriesCoefficient[#, {b[i],0,1}]&) @*
        (Coefficient[#, a[i],0]&) @*
        (Coefficient[#, x[i],0]&) @*
        (Coefficient[#, y[i],0]&) @*
        ReplaceAll[U2l] @*
        getL@
        gdo

getPCoef::usage = "getPCoef[i][gdo] returns the perturbation P of a GDO as a function of b[i]."
getPCoef[i_][gdo_][bb_] :=
        ReplaceAll[{b[i]->bb}] @*
        (Coefficient[#, a[i],0]&) @*
        (Coefficient[#, x[i],0]&) @*
        (Coefficient[#, y[i],0]&) @*
        ReplaceAll[U2l] @*
        getP@
        gdo

getaCoef::usage = "getaCoef[i][gdo] returns the linear coefficient of a[i]."
```

```
getaCoef[i_][gdo_] :=
        (SeriesCoefficient[#, {b[i],0,0}]&) @*
        (Coefficient[#, a[i],1]&) @*
        ReplaceAll[U2l] @*
        getL@
        gdo

getxCoef::usage = "getxCoef[i][gdo][b[i]] returns the linear
 ↪  coefficient of x[i] as a function of b[i]."
getxCoef[i_][gdo_][bb_] :=
        ReplaceAll[{b[i]->bb}] @*
        ReplaceAll[U2l] @*
        (Coefficient[#, y[i],0]&) @*
        (Coefficient[#, x[i],1]&) @*
        getQ@
        gdo

getabCoef::usage = "getabCoef[i][gdo] returns the linear
 ↪  coefficient of a[i]b[i]."
getabCoef[i_][gdo_] :=
        (SeriesCoefficient[#,{b[i],0,1}]&) @*
        (Coefficient[#,a[i],1]&) @*
        ReplaceAll[U2l] @*
        getL@
        gdo

getxyCoef::usage = "getxyCoef[i][gdo][b[i]] returns the linear
 ↪  coefficient of x[i]y[i] as a function of b[i]."
getxyCoef[i_][gdo_][bb_] :=
        ReplaceAll[{b[i]->bb}] @*
        ReplaceAll[U2l] @*
        (Coefficient[#,x[i],1]&) @*
        (Coefficient[#,y[i],1]&) @*
        getQ@
        gdo
```

In order to run more efficiently, limits are first computed by direct evaluation, unless such an operation is ill-defined. In such a case, the corresponding series is computed and evaluated at the limit point.

```
safeEval[f_][x_] := Module[{fx, x0},
        If[(fx=Quiet[f[x]]) === Indeterminate,
                Series[f[x0],{x0,x,0}]//Normal,
                fx
        ]
]

closeComponent[i_][gdo_GDO]:=gdo//
        setCO[Complement[gdo//getCO,{i}]]//
        setCC[Union[gdo//getCC,{i}]]
```

Now we come to the implementation of the trace map. The current implementation requires that the coefficient of $a_i b_i$ be zero. (See chapter 5 for how this restriction limits computability.)

```
tr::usage = "tr[i] computes the trace of a GDO element on
↪  component i. Current implementation assumes the Subscript[a,
↪  i] Subscript[b, i] term vanishes and $k=0."
tr::nonzeroSigma = "tr[`1`]: Component `1` has writhe: `2`,
↪  expected: 0."
tr[i_][gdo_GDO] := Module[
        {
                cL = getConstLCoef[i][gdo],
                cQ = getConstQCoef[i][gdo],
                βP = getPCoef[i][gdo],
                ηη = getyCoef[i][gdo],
                ββ = getbCoef[i][gdo],
                αα = getaCoef[i][gdo],
                ξξ = getxCoef[i][gdo],
                λ = getxyCoef[i][gdo],
                ta
        },
        ta = (1-Exp[-αα]) z[i];
        expL = cL + αα w[i] + ββ ta;
```

```
        expQ = safeEval[cQ[#] + z[i]ηη[#]ξξ[#]/(1-z[i] λ[#])&][ta];
        expP = safeEval[βP[#]/(1-z[i] λ[#])&][ta];
            CF[(gdo//closeComponent[i]//setL[expL]//setQ[expQ]//setP[expP])//.l2U]
] /; Module[
        {σ = getabCoef[i][gdo]},
        If[σ == 0,
                True,
                Message[tr::nonzeroSigma, i, ToString[σ]]; False
        ]
]
```

Here we introduce some formatting to display the output more aesthetically.

```
Format[gdo_GDO] := Subsuperscript[\[DoubleStruckCapitalE],
        Row[{gdo//getCO, ",", gdo//getCC}],
        Row[{gdo//getDO, ",", gdo//getDC}]
][gdo//getL, gdo//getQ, gdo//getP];
Format[pg_PG] := \[DoubleStruckCapitalE][pg//getL, pg//getQ, pg//getP];

SubscriptFormat[v_] := (Format[v[i_]] := Subscript[v, i]);

SubscriptFormat/@{y,b,t,a,x,z,w,η,β,α,ξ,A,B,T,W};
```

Implementing the full invariant

Now we are in a position to implement the Z invariant to tangles with a closed component. We begin by defining an object representing an isolated strand with arbitrary integer rotation number, `CCn`:

```
CCn[i_][n_Integer]:=Module[{j},
  If[n==0,
    GDO["co"->{i}],
    If[n>0,
      If[n==1,
```

```
        CC[i],
        CC[j]//CCn[i][n-1]//cm[i,j,i]
      ],
      If[n==-1,
        CCi[i],
        CCi[j]//CCn[i][n+1]//cm[i,j,i]
      ]
    ]
  ]
]
```

Since multiplication is associative, we may implement a generalized multiplication which can take any number of arguments. It is also named `cm`, with a first argument given as an ordered list of indices to be concatenated.

```
cm[{}, j_] := cη[j]
cm[{i_}, j_] := cσ[i,j]
cm[{i_, j_}, k_] := cm[i,j,k]
cm[ii_List, k_] := Module[
        {
                i  = First[ii],
                is = Rest[ii],
                j   ,
                js ,
                l
        },
        j  = First[is];
        js = Rest[is];
        cm[i,j,l] // cm[Prepend[js, l], k]
]
```

The function `toGDO` serves as the invariant for the generators of the tangles. We define its value on crossings and on concatenations of elements.

```
toGDO[Xp[i_,j_]] := cR[i,j]
toGDO[Xm[i_,j_]] := cRi[i,j]
toGDO[xs_Strand] := cm[List@@xs, First[xs]]
toGDO[xs_Loop]   := Module[{x = First[xs]}, cm[List@@xs, x]//tr[x]]
```

```

getIndices[RVT[cs_List, _List, _List]] := Sort@Catenate@(List@@@cs)

TerminalQ[cs_List][i_] := MemberQ[Last/@cs,i];
next[cs_List][i_]:=If[TerminalQ[cs][i],
        Nothing,
        Extract[cs,((#/.{c_,j_}->{c,j+1}&)@FirstPosition[i]@cs)]
]

InitialQ[cs_List][i_] := MemberQ[First/@cs,i];
prev[cs_List][i_]:=If[InitialQ[cs][i],
        Nothing,
        Extract[cs,((#/.{c_,j_}->{c,j-1}&)@FirstPosition[i]@cs)]
]
```

To minimize the size of computations, whenever adjacent indices are present in the partial computation, they are to be concatenated before more crossings are introduced.

```
MultiplyAdjacentIndices[{cs_List,calc_GDO}]:=Module[
        { is=getCO[calc]
        , i
        , i2
        },
        i = SelectFirst[is,MemberQ[is,next[cs][#]]&];
        If[Head[i]===Missing,
                {cs,calc},
                i2 = next[cs][i];
                {DeleteCases[cs,i2,2], calc//cm[i,i2,i]}
        ]
]

MultiplyAllAdjacentIndices[{cs_List, calc_GDO}] :=
        FixedPoint[MultiplyAdjacentIndices, {cs, calc}]

generateGDOFromXing[x:_Xp|_Xm,rs_Association]:=Module[
        {p, i,j, in, jn},
```

```
        {i,j} = List@@x;
        {in,jn} = Lookup[rs,{i,j},0];
        toGDO[x]*CCn[p[i]][in]*CCn[p[j]][jn]
         ↪  //cm[p[i],i,i]//cm[p[j],j,j]
]

addRotsToXingFreeStrands[rvt_RVT] := GDO[] * Times @@ (
        CCn[#][Lookup[rvt[[3]], #, 0]] & /@
        First /@ Select[rvt[[1]], Length@# == 1 &]
)
```

Next we implement the framed link invariant `ZFramed`.

```
ZFramedStep[{_List,{},_Association,calc_GDO}]:={{},{},<||>,calc};
ZFramedStep[{cs_List,xs_List,rs_Association,calc_GDO}]:=Module[
        { x=First[xs], xss=Rest[xs]
        , csOut, calcOut
        , new
        },
        new=calc*generateGDOFromXing[x,rs];
        {csOut,calcOut} = MultiplyAllAdjacentIndices[{cs,new}];
        {csOut,xss,rs,calcOut}
]

ZFramed[rvt_RVT] := Last@FixedPoint[ZFramedStep, {Sequence @@
 ↪  rvt,
        addRotsToXingFreeStrands[rvt]}]
ZFramed[L_] := ZFramed[toRVT@L]
```

Finally, when we wish to consider the unframed invariant, we apply the function `Unwrithe`, defined below.

```
Z[rvt_RVT] := Unwrithe@Last@FixedPoint[ZFramedStep, {Sequence
 ↪  @@ rvt, GDO[]}]
Z[L_] := Z[toRVT@L]

combineBySecond[l_List] := mergeWith[Total,#]& /@ GatherBy[l,
 ↪  First];
combineBySecond[lis___] := combineBySecond[Join[lis]]
```

```
mergeWith[f_, l_] := {l[[1, 1]], f@(#[[2]] & /@ l)}

Reindex[RVT[cs_, xs_, rs_]] := Module[
  {
    sf,
    cs2, xs2, rs2,
    repl, repl2
  },
   sf = Flatten[List@@#&/@cs];
   repl = (Thread[sf -> Range[Length[sf]]]);
   repl2 = repl /. {(a_ -> b_) -> ({a, i_} -> {b, i})};
   cs2 = cs /. repl;
   xs2 = xs /. repl;
   rs2 = rs /. repl2;
   RVT[cs2, xs2, rs2]
]

UnwritheComp[i_][gdo_GDO] := Module[
        {n = gdo//getL//SeriesCoefficient[#,{a[i]b[i],0,1}]&,
         ↪  j},
        gdo//(cKinkn[-n][j])//cm[i,j,i]
]

Unwrithe[gdo_GDO]:=(Composition@@(UnwritheComp/@(gdo//getCO)))@gdo

toRVT[L_RVT] := L
```

The partial trace is what we use to close a subset of the strands in a tangle. It takes the trace of all but one component, then returns the collection of all such ways of leaving one component open. (As described in ??).

```
ptr[L_RVT] := Module[
        {
                ZL = Z[L],
                cod
        },
        cod = getCO@ZL;
```

```
        Table[(Composition@@Table[tr[j],
         ↪ {j,Complement[cod,{i}]}])[ZL], {i,cod}]
]
ptr[L_] := ptr[toRVT[L]]
```

In order to be able to compare GDO's properly, we require a way to canonically represent them. This is achieved by reindexing the strands of the link and selecting one who's resulting invariant comes first in an (arbitrarily-selected) order, in this case the built-in ordering of expressions as defined by Mathematica™.

```
getGDOIndices[gdo_GDO]:=Sort@Catenate@Through[{getDO, getDC,
 ↪ getCO, getCC}@gdo]

isolateVarIndices[i_ -> j_] :=
 ↪ (v:y|b|t|a|x|η|β|α|ξ|A|B|T|w|z|W)[i]->v[j];

ReindexBy[f_][gdo_GDO] := Module[
        {
        replacementRules,
        varIndexFunc,
        repFunc,
        indices = getGDOIndices[gdo]
        },
        replacementRules = Thread[indices->(f/@indices)];
        repFunc = ReplaceAll[replacementRules];
        varIndexFunc =
         ↪ ReplaceAll[Thread[isolateVarIndices[replacementRules]]];
        gdo//applyToPG[varIndexFunc]//
                applyToCO[repFunc]//
                applyToDO[repFunc]//
                applyToDC[repFunc]//
                applyToCC[repFunc]
]

fromAssoc[ass_] := Association[ass][#] &

ReindexToInteger[gdos_List] := Module[
```

```
        {is = getGDOIndices@gdos[[1]], f},
        f = fromAssoc@Thread[is -> Range[Length[is]]];
        ReindexBy[f]/@gdos
]

getReindications[gdos_List] := Module[
        {
                gdosInt = ReindexToInteger[gdos],
                is,
                fs,
                ls
        },
        is = getGDOIndices[gdosInt[[1]]];
        fs = (fromAssoc@*Association@*Thread)/@(is -> # & /@
         ↪  Permutations[is]);
        ls = CF@ReindexBy[#]/@gdosInt&/@fs;
        Sort[Sort/@ls]
]

getCanonicalIndex[gdo_] := First@getReindications@gdo

deleteIndex[i_][expr_] := SeriesCoefficient[expr/.U2l, Sequence
 ↪  @@ ({#[i], 0, 0} & /@ {
        y, b, t, a, x, z, w
})]/.l2U
```

Here we introduce functions to further verify the co-algebra structure of a traced ribbon meta-Hopf algebra. In particular, the counit is responsible for deleting a strand. This has further applications in determining whether the invariants of individual components are contained in those of more complex links.

```
deleteIndexPG[i_][pg_PG] := pg//
        applyToL[deleteIndex[i]]//
        applyToQ[deleteIndex[i]]//
        applyToP[deleteIndex[i]]

deleteLoop[i_][gdo_] := gdo//
```

```
        applyToCC[Complement[#,{i}]&]//
        applyToPG[deleteIndexPG[i]]
```

A.3 IMPLEMENTATION OF ROTATION NUMBER ALGORITHM

Rotational Virtual Tangles (RVTs) for knots

Bar-Natan and van der Veen develop an algorithm to convert a classical long knot into an RVT. As we are interested in links, we must extend this algorithm to include so-called "long links", which we outline below:

```
1. Pass a front over the beginning of the open strand.
2. Progressively absorb the leftmost crossings
        2a. As crossings are absorbed,
            take into account any rotations of arcs.
3. If an arc passes through the front twice, absorb it,
   taking into account any rotations of that arc as a
   result.
```

Extending the algorithm to multiple components

An algorithm to convert a classical knot diagram into an upright knot diagram was implemented by Bar-Natan and van der Veen. Here we generalize the algorithm to convert a classical tangle with one open component to an upright tangle diagram. This generalization allows us to consider tangle diagrams with multiple components.

Lemma A.1. *For each classical tangle with one open component, there exists a unique upright tangle whose unbounded arcs have rotation numbers* 0*.*

Proof. See [BNvdV], Lemma 43. □

This is a Haskell implementation of the algorithm `toRVT` which takes a classical tangle and produces a rotational tangle by computing a compatible choice of rotation numbers for each arc.

We begin with a series of imports of common functions, relating to list manipulations and type-wrangling. The exact details are not too important.

```
{-# LANGUAGE DeriveFunctor #-}
module KnotTheory.PD where
import Data.Maybe (listToMaybe, catMaybes, mapMaybe, fromMaybe, fromJust)
import Data.List (find, groupBy, sortOn, partition, intersect, union, (\\))
import Data.Tuple (swap)
import Data.Function (on)
import Control.Monad ((>=>))
import Control.Arrow ((>>>))
```

Next, we introduce the crossing type, which can be either positive `Xp` or negative `Xm` (using the mnemonic “plus” and “minus”):

```
type Index = Int
data Xing i = Xp i i | Xm i i -- | Xv i i
  deriving (Eq, Show, Functor)
```

We define several functions which extract basic data from a crossing.

```
sign :: (Integral b) => Xing Index -> b
sign (Xp _ _) = 1
sign (Xm _ _) = -1

isPositive :: Xing i -> Bool
isPositive (Xp _ _) = True
isPositive (Xm _ _) = False

isNegative :: Xing i -> Bool
isNegative (Xp _ _) = False
isNegative (Xm _ _) = True

overStrand :: Xing i -> i
overStrand (Xp i _) = i
overStrand (Xm i _) = i

underStrand :: Xing i -> i
underStrand (Xp _ i) = i
underStrand (Xm _ i) = i
```

Next, we introduce the notion of a planar diagram, whose data is comprised of a collection of `Strand`s and `Loop`s (indexed by some type `i`, typically an integer). The `Skeleton` of a planar diagram is defined to be the collection of `Component`s, each of which is either an open `Strand` or a closed `Loop`.

```
type Strand i = [i]
type Loop i = [i]
data Component i = Strand (Strand i) | Loop (Loop i)
  deriving (Eq, Show, Functor)

type Skeleton i = [Component i]
```

Next, we introduce the notion of a `KnotObject`, which has its components labelled by the same type `i`. We further define a function `toRVT` which converts a generic `KnotObject` into an RVT. We call an object a <u>planar diagram</u> (or `PD`) if it has a notion of `Skeleton` and a collection of crossings.

```
class KnotObject k where
  toSX  :: (Ord i) => k i -> SX i
  toRVT :: (Ord i) => k i -> RVT i
  toRVT = toRVT . toSX

class PD k where
  skeleton :: k i -> Skeleton i
  xings :: k i -> [Xing i]

```

The `SX` form of a diagram just contains the `Skeleton` and the `Xing`s (crossings), while the `RVT` form also assigns each arc an integral rotation number.

```
data SX  i = SX  (Skeleton i) [Xing i] deriving (Show, Eq, Functor)
data RVT i = RVT (Skeleton i) [Xing i] [(i,Int)] deriving (Show, Eq, Functor)
```

Given any labelling of the arcs in a diagram, we can re-label the arcs using consecutive whole numbers. This is accomplised with `reindex`:

```
reindex :: (PD k, Functor k, Eq i) => k i -> k Int
reindex k = fmap (fromJust . flip lookup table) k
```

```
  where
    table = zip (skeletonIndices s) [1..]
    s = skeleton k
```

Most importantly, we now declare that a diagram expressed in **SX** form (that is, without any rotation data) may be assigned rotation numbers to each of its arcs in a meaningful way. The bulk of the work is done by `getRotNums`, which is defined farther below. We handle the case where the entire tangle is a single crossingless strand separately.

```
instance KnotObject SX where
  toSX = id
  toRVT k@(SX cs xs) = RVT cs xs rs where
    rs = filter ((/=0) . snd) . mergeBy sum $ getRotNums k f1
    i1 = head . toList $ s
    Just s = find isStrand cs
    f1 = case next i1 (toList s) of
            Just _  -> [(Out,i1)]
            Nothing -> []

instance KnotObject RVT where
  toRVT = id
  toSX (RVT s xs _) = SX s xs

instance PD SX where
  skeleton (SX s _) = s
  xings (SX _ xs)   = xs

instance PD RVT where
  skeleton (RVT s _ _) = s
  xings (RVT _ xs _)   = xs
```

Next, we include a series of functions which answer basic questions about planar diagrams. Note in `rotnum`, if a rotation number is not present in the table of values, it is assumed to be 0.

```
rotnums :: RVT i -> [(i,Int)]
rotnums (RVT _ _ rs) = rs

```

```
rotnum :: (Eq i) => RVT i -> i -> Int
rotnum k i = fromMaybe 0 . lookup i . rotnums $ k

isStrand :: Component i -> Bool
isStrand (Strand _) = True
isStrand _          = False

isLoop :: Component i -> Bool
isLoop (Loop _) = True
isLoop _        = False

toList :: Component i -> [i]
toList (Strand is) = is
toList (Loop is)   = is

skeletonIndices :: Skeleton i -> [i]
skeletonIndices = concatMap toList

involves :: (Eq i) => Xing i -> i -> Bool
x `involves` k = k `elem` [underStrand x, overStrand x]

otherArc :: (Eq i) => Xing i -> i -> Maybe i
otherArc x i
  | i == o     = Just u
  | i == u     = Just o
  | otherwise = Nothing
  where o = overStrand x
        u = underStrand x

next :: (Eq i) => i -> Strand i -> Maybe i
next e = listToMaybe . drop 1 . dropWhile (/= e)

prev :: (Eq i) => i -> Strand i -> Maybe i
prev e = next e . reverse

nextCyc :: (Eq i) => i -> Loop i -> Maybe i
```

```haskell
nextCyc e xs = next e . take (length xs + 1). cycle $ xs

prevCyc :: (Eq i) => i -> Loop i -> Maybe i
prevCyc e xs = prev e . take (length xs + 1). cycle $ xs

isHeadOf :: (Eq i) => i -> [i] -> Bool
x `isHeadOf` ys = x == head ys

isLastOf:: (Eq i) => i -> [i] -> Bool
x `isLastOf` ys = x == last ys

nextComponentIndex :: (Eq i) => i -> Component i -> Maybe i
nextComponentIndex i (Strand is) = next i is
nextComponentIndex i (Loop is) = nextCyc i is

prevComponentIndex :: (Eq i) => i -> Component i -> Maybe i
prevComponentIndex i (Strand is) = prev i is
prevComponentIndex i (Loop is) = prevCyc i is

isHeadOfComponent :: (Eq i) => i -> Component i -> Bool
isHeadOfComponent _ (Loop    _ ) = False
isHeadOfComponent i (Strand is) = i `isHeadOf` is

isLastOfComponent :: (Eq i) => i -> Component i -> Bool
isLastOfComponent _ (Loop    _ ) = False
isLastOfComponent i (Strand is) = i `isLastOf` is

isTerminalOfComponent :: (Eq i) => Component i -> i ->  Bool
isTerminalOfComponent c i = i `isHeadOfComponent` c || i
  ↪  `isLastOfComponent` c

isTerminalIndex :: (Eq i) => Skeleton i -> i ->  Bool
isTerminalIndex cs i = any (`isTerminalOfComponent` i) cs

nextSkeletonIndex :: (Eq i) => Skeleton i -> i -> Maybe i
```

```
nextSkeletonIndex s i = listToMaybe . mapMaybe (nextComponentIndex i) $ s

prevSkeletonIndex :: (Eq i) => Skeleton i -> i -> Maybe i
prevSkeletonIndex s i = listToMaybe . mapMaybe (prevComponentIndex i) $ s
```

In order to obtain all the crossing indices, we must take every combination of the under- and over-strands and their following indices:

```
getXingIndices :: (Eq i) => Skeleton i -> Xing i -> [i]
getXingIndices s x = catMaybes
        [ f a | f <- [id, (>>= nextSkeletonIndex s)], a <- [o, u] ]
        where o = return (overStrand x)
              u = return (underStrand x)

δ :: (Eq a) => a -> a -> Int
δ x y
  | x == y    = 1
  | otherwise = 0

mergeBy :: (Ord i) => ([a] -> b) -> [(i,a)] -> [(i,b)]
mergeBy f = map (wrapIndex f) . groupBy ((==) `on` fst) . sortOn fst
  where
    wrapIndex :: ([a] -> b) -> [(i,a)] -> (i,b)
    wrapIndex g xs@(x:_) = (fst x, g . map snd $ xs)
```

Here we come to the main function, `getRotNums`, for which we have the following requirements (not expressed in the code):

1. The diagram `k` is a $(1, n)$-tangle (a tangle with only one open component)
2. The underlying graph of `k` is a planar.
3. The diagram `k` is a connected.

Only in this case will the function `toRVT` will then output a planar $(1, n)$-rotational virtual tangle which corresponds to a classical (i.e. planar) diagram.

This function involves taking a simple open curve (a Jordan curve passing through infinity) called the **`Front`**, and passing it over arcs in the diagram. This curve is characterized by the arcs it passes through, together with their orientations. Each intersection of the **`Front`** with the diagram provides a different **`View`**, either **`In`** or **`Out`** of the **`Front`** when following the orientation of the intersecting arc.

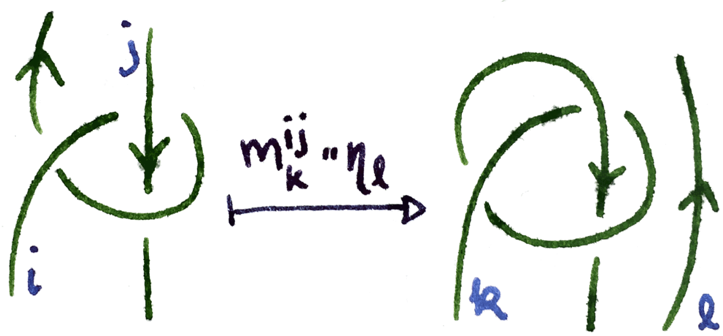

Figure A.1: A tangle with a front passing over it. The portion of the tangle below the front is in upright form.

```
type Front i = [View i]
type View  i = (Dir, i)
```

We obtain the rotation numbers by successively passing the front across new crossings (achieved by `advanceFront`), keeping track of the rotation numbers of arcs which have already passed by the front. Once the front has passed across every crossing, all the rotation numbers have been computed.

Next, we define `converge`, which iterates a function until a fixed point is achieved.

```
converge :: (Eq a) => (a -> a) -> a -> a
converge f x
        | x == x'    = x
        | otherwise = converge f x'
        where x' = f x
```

The function `convergeT` wraps `converge` in monadic transformations. In our context, the monad will be used to keep track of rotation numbers of the arcs.

```
convergeT :: (Monad m, Eq (m a)) => (a -> m a) -> a -> m a
convergeT f = return >>> converge (>>= f)
```

The implementation of `getRotNums` takes a front and advances it along a diagram until no more changes occur.

```
getRotNums :: (Eq i) => SX i -> Front i -> [(i,Int)]
getRotNums k = convergeT (advanceFront k) >>> fst
```

When advancing the `Front`, we start by absorbing arcs that intersect with the front twice until the leftmost `View` no longer connects directly back to the `Front`. At this point, we can absorb a crossing into the front.

```
advanceFront :: (Eq i) => SX i -> Front i -> ([(i,Int)], Front i)
advanceFront k = convergeT (absorbArc k) >=> absorbXing k
```

We next check for the case where the leftmost arc connects back to the `Front`. If it is pointing `Out` (and therefore connects back `In` further to the right), we adjust the rotation number of the arc by -1. Otherwise, we leave both the `Front` and the rotation numbers unchanged.

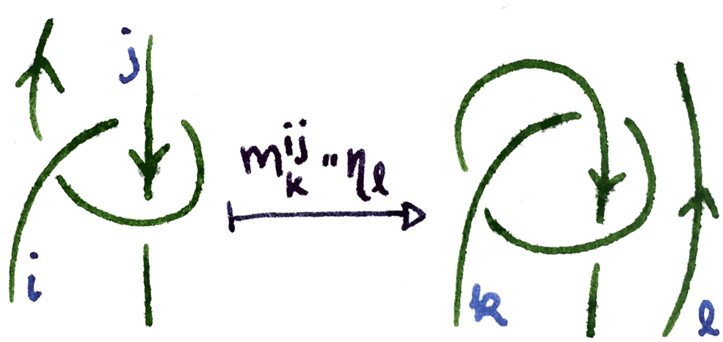

Figure A.2: Example of absorbing an arc which intersects the front multiple times. The rotation number of the arc depends on its orientation.

```
absorbArc :: (Eq i) => SX i -> Front i -> ([(i,Int)],Front i)
absorbArc k []      = return []
```

```
absorbArc k f@(f1:fs) = case fs1 of
        ( In,i):_ -> (return (i,-1), fss)
        (Out,i):_ -> return fss          -- No new rotation numbers
        []        -> return f
        where (fs1,fss) = partition (((==) `on` snd) f1) fs
```

Our goal is to repeat this operation until we get a fixed point, which is encoded in `absorbArcs`:

```
absorbArcs :: (Eq i) => SX i -> Front i -> ([(i,Int)],Front i)
absorbArcs k = convergeT (absorbArc k)
```

Absorb a crossing involves expanding one's view at an arc from looking at a crossing to all the views one gets when looking in every direction at the crossing (namely, to the left, along the arc, and to the right). The function `absorbXing` performs this task on the leftmost `View` on the `Front`. The transverse strand receives a positive rotation number if it moves from left to right. The arc receiving the rotation depends on how the crossing is oriented.

```
absorbXing :: (Eq i) => SX i -> Front i -> ([(i,Int)],Front i)
absorbXing _ [] = return []
absorbXing k (f:fs) = (rs,newFront++fs) where
        newFront = catMaybes [l, a, r]
        l = lookLeft k f
        a = lookAlong k f
        r = lookRight k f
        rs = case (l,f,r) of
                (Just (In,i), (Out,_),_           ) -> [(i,1)]
                (_          , (In ,_),Just (Out, j)) -> [(j,1)]
                _                                    -> [     ]

data Dir = In | Out
  deriving (Eq, Show)
```

The following functions take a `View`, returning the `View` one has when looking in the corresponding direction. Since it is possible for the resulting gaze to be merely the boundary, it is possible for these functions to return `Nothing`.

```haskell
lookAlong :: (Eq i, PD k) => k i -> View i -> Maybe (View i)
lookAlong k (d, i) = case d of
        Out -> sequence (Out, nextSkeletonIndex s i)
        In  -> sequence (In , prevSkeletonIndex s i)
        where s = skeleton k

lookSide :: (Eq i, PD k) => Bool -> k i -> View i -> Maybe (View i)
lookSide isLeft k di@(Out,i) = do
        x <- findNextXing k di
        j <- otherArc x i
        if isLeft == ((underStrand x == i) == isPositive x)
        then return (In, j)
        else sequence (Out, nextSkeletonIndex (skeleton k) j)
lookSide isLeft k (In,i) =
        sequence (Out, prevSkeletonIndex (skeleton k) i) >>=
        lookSide (not isLeft) k

lookLeft  :: (Eq i, PD k) => k i -> View i -> Maybe (View i)
lookLeft = lookSide True

lookRight :: (Eq i, PD k) => k i -> View i -> Maybe (View i)
lookRight = lookSide False

findNextXing :: (Eq i, PD k) => k i -> View i -> Maybe (Xing i)
findNextXing k (Out,i) = find (`involves` i) $ xings k
findNextXing k (In ,i) = do
  i' <- prevSkeletonIndex (skeleton k) i
  find (`involves` i') $ xings k
```

BIBLIOGRAPHY

[BNS] Dror Bar-Natan and Sam Selmani, Meta-monoids, meta-bicrossed products, and the alexander polynomial, no. 10, 1350058, Publisher: World Scientific Publishing Co.

[BNvdV] Dror Bar-Natan and Roland van der Veen, Perturbed gaussian generating functions for universal knot invariants, no. arXiv:2109.02057.

[ES] Pavel I. Etingof and Olivier Schiffmann, Lectures on quantum groups, International Press.

[Maj] Shahn Majid, A quantum groups primer, London Mathematical Society Lecture Note Series, Cambridge University Press.

COLOPHON

This thesis was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić.

The style was inspired by Robert Bringhurst's seminal book on typography The Elements of Typographic Style.