A PACKAGE FOR CALCULATING AND MANIPULATING HOPF ALGEBRAS IN MACAULAY2

BY

COLIN ELDRIDGE MARTIN

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF ARTS

Mathematics and Statistics

May, 2019

Winston-Salem, North Carolina

Approved By:

W. Frank Moore, Ph.D., Advisor

Ellen Kirkman, Ph.D., Chair

Sarah Mason, Ph.D.

Acknowledgments

To paraphrase Isaac Newton, we all stand on the shoulders of metaphorical giants here in academia; I was very, very luck to stand on the shoulders of one of the largest (symbolically speaking) men I have ever had the pleasure of learning from, my advisor Dr. Frank Moore. His knowledge and excitement for mathematics are rivaled only by his patience and care as an educator.

I would also thank Dr. Ellen Kirkman and Dr. Sarah Mason for serving on my thesis committee, as well as Dr. Robert Won for starting me on my algebraic journey.

Finally, I would like to thank all my other professors, my classmates, my partner Alena Ross Becker, her cat Thimbly, and the many baristas of Winston-Salem. Between the free flowing advice and occasional free drink, you have all provided me with the emotional, material and academic support that I needed to complete this project.

Table of Contents

List of Figures

Abstract

Colin Eldridge Martin

The study of Hopf algebras is a relatively new field of modern algebra with many applications in other areas. Despite this, there is a marked lack of both introductory literature and software support for mathematicians working with Hopf algebras. Here, we introduce the software `HopfAlgebras.m2`, a package for the `Macaulay2` algebra system designed to perform calculations on Hopf algebras, with a focus on invariant theory.

Chapter 1: Introduction

This document serves to introduce `HopfAlgebras.m2` package, a `Macaulay2` package for making computations over a Hopf algebra or bialgebra. `HopfAlgebras.m2` automates many different tasks over a Hopf algebra or bialgebra, including

- showing the well-definedness of prospective bialgebras and graded bialgebra actions,
- calculating group-like elements of a bialgebra, as well as the dual bialgebra, and
- calculating the antipode of a prospective Hopf algebra.

The package was motivated by a lack of computational algebra packages that are conducive to making such computations "out of the box."

Chapter 2: Background and Motivation

Assume throughout that k is a field. In this chapter we provide the background necessary to discuss the algebraic structures and behaviors that the `HopfAlgebras.m2` package was designed to investigate. We will assume the reader has the mastery of a student who has completed a standard undergraduate abstract algebra sequence. We begin by introducing the tensor product as our *de facto* method for generating new k-vector space objects from old ones. We then discuss the representations of groups and algebras, which ultimately motivates the construction of the objects we are primarily interested in: Hopf algebras and Hopf actions.

2.1 The Tensor Product

Put briefly, the k-tensor product is a convenient way to induce a k-linear map from a k-bilinear one. In fact, it is in some sense the *only* way to induce such a linear map, due to the universal property of k-tensor products.

Definition 2.1. *Let V, W and Y be k-vector spaces. A map $\varphi : V \times W \to Y$ is bilinear if, for all $v, v' \in V$, $w, w' \in W$ and $c \in k$, the following conditions hold.*

$$\varphi(v + v', w) = \varphi(v, w) + \varphi(v', w)$$

$$\varphi(v, w + w') = \varphi(v, w) + \varphi(v, w')$$

$$c\varphi(v, w) = \varphi(cv, w) = \varphi(v, cw).$$

Examples of bilinear maps include inner products on $\mathbb{R}$-vector spaces, dot products on $\mathbb{R}$-vector spaces, and matrix multiplication. As these examples illustrate, bilinear maps are an abstraction of elementary multiplication. We then expect any multiplication rule defined on a vector space to be bilinear. However, bilinear maps are not

module homomorphisms in general. Hence, we must abandon Cartesian products for a new algebraic construction that will "linearize" bilinear maps for us.

Let V, W and Y be k-vector spaces. Take $F(V \times W)$ to be the free vector space on the set $V \times W$ (i.e. F is the free functor). Take $I \subseteq F(V \times W)$ to be the k-span of all vectors of the form

$$\begin{aligned}(v + v', w) - (v, w) + (v', w)\\ (v, w + w') - (v, w) + (v, w')\\ c(v, w) - (cv, w)\\ c(v, w) - (v, cw).\end{aligned}$$

Note that I is the smallest possible kernel of the linear map induced by any k-bilinear map with domain $V \times W$. We may force these relations to be zero by taking the quotient space $F(V \times W)/I$.

Definition 2.2. *Let V, W, $F(V \times W)$ and I to be as above. Then the quotient space $F(V \times W)/I$ is called the tensor product over k of V and W. It is denoted $V \otimes_k W$, or just $V \otimes W$ when k is understood. The coset $\sum_{i=1}^{n} c_i(v_i, w_i) + I$ is denoted $\sum_{i=1}^{n} c_i(v_i \otimes_k w_i)$, where again the k can be omitted depending on context.*

For example, the coset in $\mathbb{C} \otimes \mathbb{C}$ with representative $(i, -i)$ is denoted $i \otimes_{\mathbb{R}} i$. We call elements of $V \otimes W$ that can be written as a single double $v \otimes w$ *simple tensors*. Since the set of all simple tensors span a tensor product, it often suffices to show that a given property holds on the simple tensors; it is important to remember that, in general, an arbitrary element of $V \otimes W$ cannot be written as a simple tensor.

The most important interface for working with tensors is the universal property of tensor products.

Theorem 2.3 (Universal Property of Tensor Products)**.** *Let V, W, $F(V \times W)$ and I be as above. Suppose that ψ is a bilinear map from $V \times W$ to a third k-vector space Y. Then there exists a unique linear transformation $\Psi : V \otimes W \to Y$ such that $\Psi = \psi \circ i$ (where $i : V \times W \to V \otimes W$ by $(v, w) \mapsto v \otimes w$). Equivalently, there exists a unique ψ such that the following diagram commutes.*

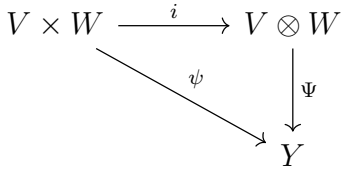


Figure 2.1: The universal property of tensor products

This theorem is particularly useful for defining maps out of a tensor product; we can sidestep the potential quagmire of ensuring well-definedness of a map by defining a bilinear map out of a Cartesian product instead.

In no particular order, here are several other well-known results on the tensor products of vector spaces.

Proposition 2.4. *For all k-vector spaces V, $k \otimes_k V \cong V \otimes_k k \cong V$.*

Proposition 2.5. *Suppose V and W are k-vector spaces with bases given by $\mathcal{B}$ and $\mathcal{C}$ respectively. Then the tensor product $V \otimes W$ has basis $\{b \otimes c \,|\, b \in \mathcal{B}, c \in \mathcal{C}\}$. In particular, if V and W have finite dimension n and m, $V \otimes W$ has dimension nm.*

We also define a tensor product on linear maps:

Definition 2.6. *Let A, B, C and D be k-vector spaces with linear maps $\varphi : A \to B$ and $\psi : C \to D$. Then $\varphi \otimes \psi$ is the linear map*

$$\varphi \otimes \psi : A \otimes C \to B \otimes D$$
$$a \otimes c \mapsto \varphi(a) \otimes \psi(c).$$

Note that the k-tensor product of k-algebras A and B is again a k-algebra, with unit given by $1_A \otimes 1_B$ and multiplication given by $(a \otimes b) \cdot (a' \otimes b') = aa' \otimes bb'$.

2.2 Representations of groups and algebras

Although it has since expanded in scope, the initial focus of `HopfAlgebras.m2` was to calculate invariants of Hopf actions. Since a Hopf action is a particular kind of algebra action, there is obvious utility in discussing algebra representations. Additionally, the representation theory of groups and algebras provides a serendipitously thematic motivation for the Hopf algebra as an algebraic construct: from a representation theory perspective, Hopf algebras are the appropriate abstraction of groups into the category of k-vector spaces.

We begin by defining a group representation.

Definition 2.7. *Let G be a group and let V be a k-vector space. If X is a group homomorphism such that $X : G \to GL_n(V)$, we call X a degree n representation of G.*

Equivalently, we call V a G-module if such an X exists, with action by G on V defined the obvious way: for all $g \in G$ and $v \in V$, $g \cdot v := X(g)v$, which we denote $g \cdot v$ if X is understood. Note that this definition also satisfies the group action axioms, so a group acting linearly on a vector space is the same as a group module is the same as a representation of a group.

One nice property of G-modules over a field k is that the k-tensor of two such G-modules is again a G-module over k. In fact, the category of G-modules over k together with $\otimes_k$ fulfill a stronger categorical condition.

Definition 2.8. *A monoidal category $\mathbf{C}$ is a category equipped with a bifunctor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ that is associative, has a left and right identity, and has some object*

serving as a unit (all up to isomorphism).

Reference [5] for a more rigorous definition; ours is provided sheerly for the sake of motivation.

Example 2.9. The category **k-Vect** (the category of all vector spaces over k) together with the k-tensor functor is a monoidal category. The field k thought of as a one-dimensional vector space over itself serves as a unit, since $k \otimes_k V \cong V \otimes_k k \cong V$ for any $V \in \text{ob}(\textbf{k-Vect})$.

Example 2.10. $\textbf{k}G\textbf{-mod}$, the category of G-modules over k, is also a monoidal category via the same conditions as above. For any two G-modules M and N, G acts on $M \otimes_k N$ diagonally, i.e. for some simple tensor $m \otimes n \in M \otimes N$, $g(m \otimes n) = gm \otimes gn$. Since $M \otimes N$ is spanned by simple tensors, we may extend this action by linearity to get an action on all of $M \otimes N$.

Now we define a k-algebra module.

Definition 2.11. *Let A be a k-algebra. Then a k-vector space V is an A-module if there exists a k-algebra morphism $\varphi : A \to \text{End}_k(V)$. For $a \in A$ and $v \in V$, we denote $\varphi(a)(v)$ as $a \cdot v$ if φ is understood.*

We refer to φ above as a representation of A. Can we extend the monoidal category structure to the category of algebra modules? In general, the answer is no. Suppose that $\varphi : A \to \text{End}_k(V)$ and $\psi : A \to \text{End}_k(W)$ are two representations of A. Clearly, $V \otimes W$ is a k-vector space, but if we were to define the obvious "action" of A on simple tensors of $V \otimes W$

$$a \cdot (v \otimes w) := \varphi(a)(v) \otimes \psi(a)(w)$$

then things fall apart. For instance, this "action" does not preserve scalar multiplication:

$$\begin{aligned}(ca)\cdot(v\otimes w) &= \varphi(ca)(v)\otimes\psi(ca)(w)\\ &= c\varphi(a)(v)\otimes c\psi(v)(w)\\ &= c^2\left(\varphi(a)(v)\otimes\psi(a)(w)\right)\\ &\neq c\left(a\cdot(v\otimes w)\right)\end{aligned}$$

However, $V\otimes W$ *is* an $A\otimes A$ module. This comes directly from the fact that $\varphi\otimes\varphi : A\otimes A\to \mathrm{End}_k(V)\otimes\mathrm{End}_k(W)\cong\mathrm{End}_k(V\otimes W)$ is an algebra homomorphism. Thus, if we can find another algebra homomorphism $\Delta : A\to A\otimes A$ to compose with $\varphi\otimes\varphi$, then we have successfully endowed $V\otimes W$ with an A-module structure. We must also have some notion of a "trivial" representation $\epsilon : A\to k$ that may act as the identity element in our prospective monoidal category. An algebra endowed with this additional structure is a bialgebra, which is the focus of the following chapter.

Chapter 3: Coalgebras and bialgebras

As we concluded in the previous section, for a k-algebra H, the tensor product of H-modules V and W is a k-vector space, but not necessarily an H-module. For this reason, a group algebra needs more structure to be a satisfying abstraction of a group from a representation and category theoretic perspective.

The algebraic structures that are the focus of this chapter often involve maps between tensor products of a vector space. In particular, the coproduct $\Delta : C \to C \otimes_k C$ of a coalgebra C will come up repeatedly. We often use what is known as "Sweedler's notation" to represent arbitrary elements of the tensor product. Normally, if we wanted to discuss the image of some element $c \in C$ under a map Δ like described above, we would leverage the fact that $C \otimes C$ is a span of simple tensors in $C \otimes C$ to write $\Delta(c)$ as the finite sum of simple tensors in $C \otimes C$:

$$\Delta(c) = \sum_{i=1}^{n} a_i \otimes b_i.$$

Sweedler's notation relieves us of the painstaking chore of accounting myriad indices and symbols by doing away with both the indices as well as the symbols "a" and "b:"

$$\Delta(c) = \sum c_{(1)} \otimes c_{(2)}.$$

3.1 Coalgebras

A coalgebra can be thought of as dual to a k-algebra, in that it "reverses the arrows" of the categorical algebra definition. Let us recall the definition of a k-algebra.

Definition 3.1. *A k-algebra A is a triple (A, μ, u) where A is a k-vector space, $\mu : A \otimes A \to A$ and $u : k \to A$ are k-linear maps such that the following diagrams*

commute.

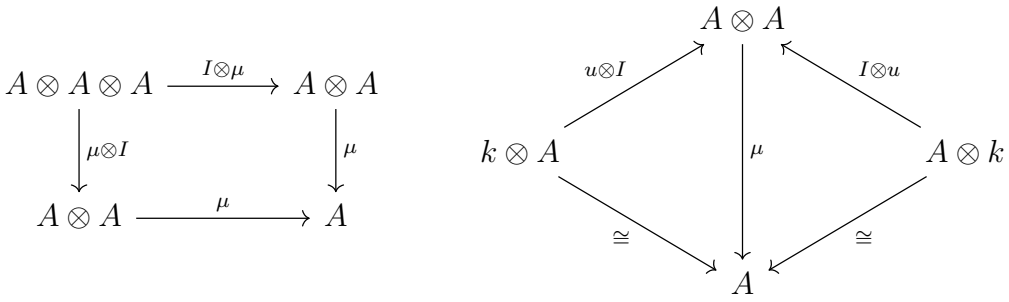


Figure 3.1: Commutative diagrams defining a k-algebra

I above is the identity map, and "$\cong$" is the canonical isomorphism given by $c \otimes v \mapsto cv$. The commutativity of the first diagram is really just associativity of μ, and the second one is the compatibility of scalar multiplication with μ.

With these commutative diagrams, we may now define a coalgebra.

Definition 3.2. *A k-coalgebra is a triple (C, Δ, ε), where C is a k-vector space, $\Delta : C \to C \otimes C$ and $\epsilon : C \to k$ such that the following diagrams commute.*

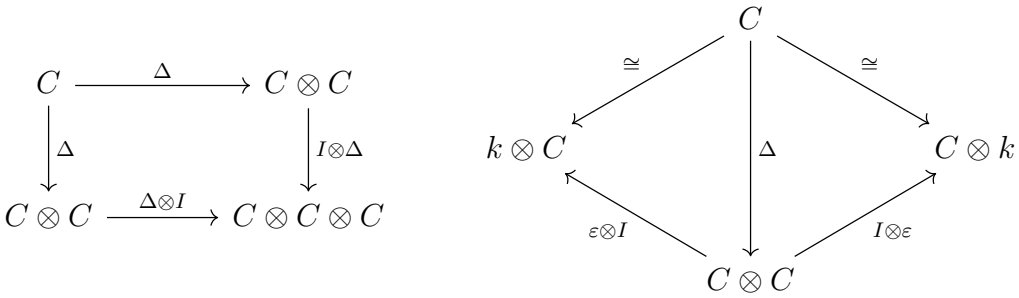


Figure 3.2: Commutative diagrams defining a k-coalgebra

We call ε the "counit" or "augmentation," and Δ is the "coproduct." The property described by the first diagram is called "coassociativity."

Example 3.3. Let V be a k-vector space with basis $\mathcal{B}$. Then we may endow V with a (trivial) coalgebra structure by letting $\Delta : g \mapsto g \otimes_k g$ and $\varepsilon : g \mapsto 1_k$ for all $g \in \mathcal{B}$,

and extending by linearity.

Example 3.4 (Divided power coalgebra). This is the first nontrivial coalgebra introduced in Dascalescu et al [2]. Let V be a k-vector space with basis given by $\{b_i \mid i \in \mathbb{N}\}$. Then (V, Δ, ε) with Δ and ε defined by

$$\Delta(b_m) = \sum_{i=0}^{m} b_i \otimes b_{m-i}$$

$$\varepsilon(b_i) = \begin{cases} 1 & i = 0 \\ 0 & i \neq 0 \end{cases}$$

is a coalgebra.

Coalgebras may have a property that is in some sense "dual" to commutativity property of an algebra. In the language of categories, we say that an algebra A is commutative if the diagram

$$\begin{array}{ccc} A \otimes A & \xrightarrow{\mu} & A \\ \downarrow{\scriptstyle \tau} & \nearrow{\scriptstyle \mu} & \\ A \otimes A & & \end{array}$$

is commutative (τ here is the twist map send simple tensors $a \otimes b$ to $b \otimes a$). We may dualize this diagram to develop the notion of "cocommutativity."

Definition 3.5. *A coalgebra C is cocommutative if the diagram*

$$\begin{array}{ccc} C & \xrightarrow{\Delta} & C \otimes C \\ \downarrow{\scriptstyle \Delta} & \nearrow{\scriptstyle \tau} & \\ C \otimes C & & \end{array}$$

commutes.

Example 3.6. Let H be the noncommutative algebra generated by x, y and z, subject to the relations

$$x^2 = y^2 = 1,\ xy = yx,\ zx = yz,\ zy = xz,\ z^2 = \frac{1}{2}(1 + x + y - xy).$$

Then the maps $\Delta : H \to H \otimes H$ and $\varepsilon : H \to k$ where

$$\Delta(x) = x \otimes x$$

$$\Delta(y) = y \otimes y$$

$$\Delta(z) = \frac{1}{2}(1 \otimes 1 + 1 \otimes x + y \otimes 1 - y \otimes x)(z \otimes z)$$

$$1 = \varepsilon(x) = \varepsilon(y) = \varepsilon(z)$$

give us a coalgebra structure (H, Δ, ε) on H. [3]

The previous example is the coalgebra structure of the so-called Kac-Paljutkin Hopf algebra H_8, which is our perennial example of a noncommutative, noncocommutative Hopf algebra.

We may similarly define coalgebra morphisms by reversing the arrows of the commutative diagrams that define an algebra homomorphism.

Definition 3.7. *Let (A, μ, u) and (B, μ', u') be k-algebras. A morphism of k-algebras A and B is a k-linear map $\varphi : A \to B$ such that the following diagrams commute.*

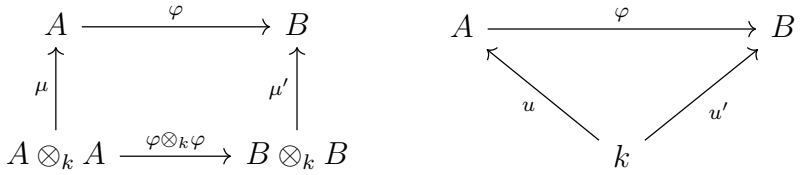


Figure 3.3: Commutative diagram describing k-algebra morphism axioms

Commutivity in the left diagram implies that φ respects algebra multiplication, while commutivity in the right diagram implies that $\varphi(1_A) = 1_B$. As above, we derive a definition of a coalgebra morphism by reversing the arrows of the above commutative diagram.

Definition 3.8. *Let (C, Δ, ε) and $(D, \Delta', \varepsilon')$ be k-coalgebras. Then a k-linear map $\varphi : C \to D$ is a coalgebra homomorphism if the following diagrams commute.*

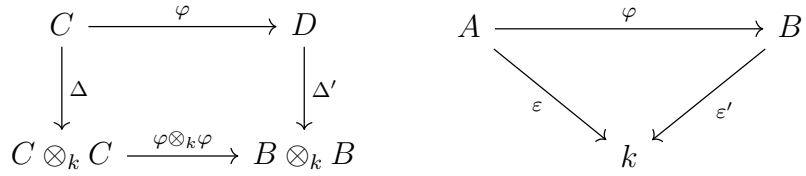


Figure 3.4: Commutative diagram describing k-coalgebra morphism axioms

Recall that for a k-space V, we call the set $\mathrm{Hom}_k(V, k)$ the dual of V, and we denote it V^*. If C is a finite dimensional coalgebra, we may impart an algebra structure on C^*.

Proposition 3.9. *Suppose that (C, Δ, ε) is a coalgebra. Then (C^*, μ, u), with μ and u defined on all $\varphi, \psi \in C^*$ as*

$$\mu(\phi \otimes \psi)(c) = (\phi \otimes \psi) \circ \Delta c = \sum \phi\left(c_{(1)}\right) \psi\left(c_{(2)}\right)$$

$$u(k) = k \cdot \varepsilon(c)$$

respectively, is an algebra. [2]

The map μ as defined above is called the "convolution product" on $\mathrm{Hom}_k(C, k)$, and $\mu(\phi, \psi)$ is usually denoted $\phi * \psi$. We will see a more general definition of the convolution product when we define the antipode later in this chapter.

3.2 Bialgebras

Definition 3.10. *A bialgebra A is a tuple $(A, \mu, u, \Delta, \varepsilon)$ such that (A, μ, u) is a k-algebra, (A, Δ, ε) is a k-coalgebra, and we have that*

1. *Δ and ε are algebra morphisms, and*

2. μ and u are coalgebra morphisms.

In fact, this definition is redundant, as the last two statements above imply one another.

Proposition 3.11. *Let A be a k-vector space such that (A, μ, u) is an algebra and (A, Δ, ε) is a coalgebra. Then Δ and ϵ are algebra morphisms if and only if μ and u are coalgebra morphisms. [2]*

Proof. We apply the categorical definition of an algebra morphism to Δ and ε:

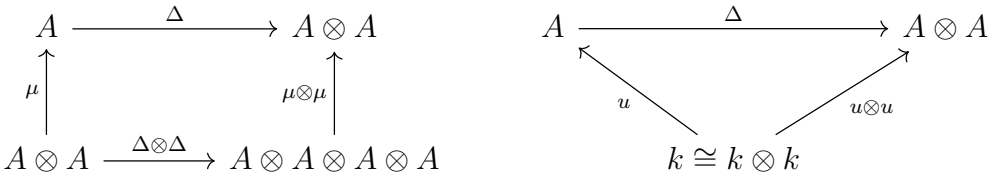


Figure 3.5: Definition of "$\Delta : A \to A \otimes A$ is an algebra morphism."

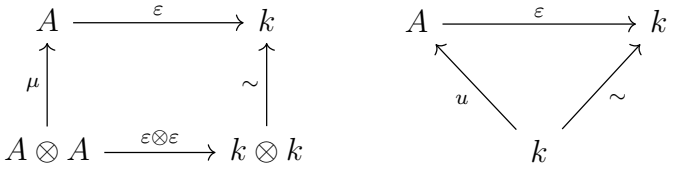


Figure 3.6: Definition of "$\varepsilon : A \to k$ is an algebra morphism."

However, if we may apply the definition of a coalgebra morphism to μ and u, we get identical diagrams:

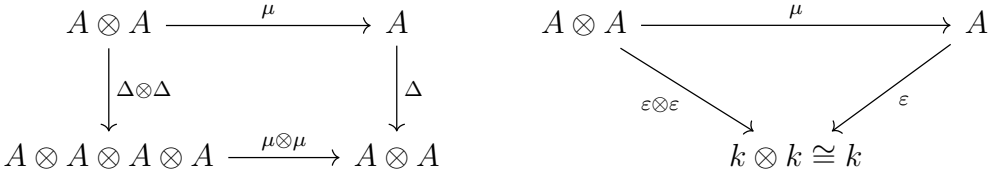


Figure 3.7: Definition of "μ is a coalgebra morphism"

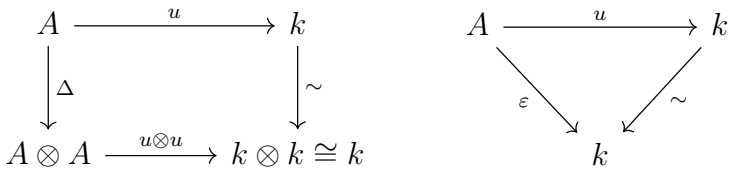


Figure 3.8: Definition of "u is a coalgebra morphism"

Therefore, Δ and ε are algebra homomorphisms if and only if μ and u are coalgebra homomorphisms. □

Example 3.12. The group algebra kG is a bialgebra with $\Delta(g) = g \otimes g$ and $\varepsilon(g) = 1_k$. [2]

Example 3.13. We saw in example 3.7 that the algebra

$$H_8 = k\langle x, y, z\rangle / \langle 1 - x^2, 1 - y^2, \frac{1}{2}(1 + x + y - xy) - z^2, xy - yx, zx - yz, zy - xz\rangle$$

was given a coalgebra structure via the maps Δ and ε:

$$\Delta(x) = x \otimes x$$

$$\Delta(y) = y \otimes y$$

$$\Delta(z) = \frac{1}{2}(1 \otimes 1 + 1 \otimes x + y \otimes 1 - y \otimes x)(z \otimes z)$$

$$1 = \varepsilon(x) = \varepsilon(y) = \varepsilon(z).$$

In fact, Δ and ε are algebra morphisms; therefore H_8 is a bialgebra. [3]

Since most bialgebras we are interested in (including the examples above) have an antipode, we save our other examples for the next chapter.

Recall that we were able to define an algebra structure on C^*, the dual space of a coalgebra C. Now, let B^* be the dual algebra of the coalgebra structure of the bialgebra B, with multiplication and unit as defined in proposition 3.10. Then, if B

is finite dimensional, B^* is a bialgebra as well, with coproduct given by the dual of the product and unit in B:

$$\mu^* : B^* \to (B \otimes B)^* \cong B^* \otimes B^*$$
$$\varphi \mapsto \ (\varphi \circ \mu)$$

Chapter 4: Hopf algebras

Recall that our motivation for constructing a bialgebra was in that we wanted to find a group object over the category of vector spaces that is satisfying from a representation theory perspective. Have we done so? Recall that for B-modules V and W, $V \otimes W$ is automatically an $B \otimes B$ module. If B is a bialgebra, then the coproduct Δ ensures we have an algebra homomorphism to tensor powers of B; therefore, $V \otimes W$ is an B-module, with action by B defined as

$$b(v \otimes w) = \sum b_{(1)}(v) \otimes b_{(2)}(w).$$

In other ways, the bialgebra by itself is lacking as an abstraction of a group. For instance, there is not an analogue to group inverses in general, and there is no guaranteed algebraic structure on the set of group-likes. This is rectified by the one additional piece of structure called the antipode.

Definition 4.1. *A Hopf algebra over the field k is a k-bialgebra B equipped with an additional k-linear map S such that S is the inverse of the identity map in the convolution algebra* $\mathrm{Hom}_k(B, B)$.

For this definition to make sense, we will quickly discuss what is meant by the "convolution algebra" before moving on.

4.1 The convolution algebra $\mathrm{Hom}_k(C, A)$

Let C be a k-colagebra, and let A be a k-algebra. We may place an algebra structure on $\mathrm{Hom}_k(C, A)$, with multiplication defined to be the convolution product as a generalization of the one defined in section 3.1.

Definition 4.2. *Let A be an algebra, and let C be a coalgebra. Suppose f and g are elements of* $\mathrm{Hom}_K(C, A)$. *Then the convolution product $f * g$ is defined for all $c \in C$ as*

$$(f * g)(c) = \mu_A \left(\sum f\left(c_{(1)}\right) \otimes g\left(c_{(2)}\right) \right).$$

Note that when we consider k as an algebra over itself, this reduces to the previous definition.

Proposition 4.3. *Let (C, Δ, ε) be a k-coalgebra, and let (A, μ, u) be a k-algebra. Then the triple* $(\mathrm{Hom}_k(C, A), M, \eta)$, *where $M(f \otimes g) = f * g$ and $\eta = i \circ \varepsilon$, is a k-algebra. [6]*

This proposition begets the following definition:

Definition 4.4. $(\mathrm{Hom}_k(C, A), M, \eta)$ *as defined above is called the convolution algebra on* $\mathrm{Hom}_k(C, A)$.

While the definition requires only that C be a coalgebra and A be an algebra, we only consider the convolution product on k-linear endomorphisms of a bialgebra for the purpose of constructing the antipode. Perhaps non-intuitively, the identity map $I : B \to B$ is *not* the identity in the convolution algebra; the map $u\varepsilon$ is. For example, for $g \in kG$ (given the bialgebra structure in example 3.14), $I * I(g) = g^2 \neq I(g)$, while $I * \varepsilon(g) = g\varepsilon(g) = g$.

4.2 The antipode

Definition 4.5. *For a k-bialgebra B, we call an element S of the convolution algebra* $\mathrm{Hom}_k(B, B)$ *an antipode if S is the inverse of I; that is, if*

$$S * I = I * S = u \circ \varepsilon.$$

Here are a few properties of the antipode. First, we should defend our use of the article "the" when discussing the antipode of a bialgebra:

Proposition 4.6. *If a bialgebra possesses an antipode S, then that antipode is unique.*

Proof. If an element of a k-algebra has an inverse, then that inverse is unique. Hence this result falls directly from I_B being an element of the convolution algebra $\mathrm{Hom}_k(B, B)$. □

S is a k-linear map by definition, but it has additional algebraic structure.

Proposition 4.7. *If a bialgebra B has an antipode S, then S is an algebra antihomomorphism; that is, for $b, c \in B$, $S(bc) = S(c)S(b)$. [2]*

Finally, observe that S is in fact analogous to group inverses in that it must always map a group element to its inverse:

Proposition 4.8. *Let H be a Hopf algebra with group-like element g. Then $S(g) = g^{-1} \in H$. [8]*

Proof. Let H be a Hopf algebra with group-like element g. Since $\Delta g = g \otimes g$, $(S * I)g = \mu(S(g) \otimes I(g)) = S(g)g$. However, recall also that $\varepsilon g = 1$, and that $I * S = u \circ \varepsilon$. Then it must be that $(S * I)g = 1_H$, but then we have that $S(g)g = 1_H$. Therefore, $S(g) = g^{-1}$. □

4.3 Examples

Now that we have defined a Hopf algebra, we present several examples. We have explored the first two already.

Example 4.9 (group algebra kG). We have already seen that for a group G, algebra kG is given a bialgebra structure by $\Delta(g) = g \otimes g$ and $\varepsilon(g) = 1$. The antihomomorphism $S(g) = g^{-1}$ makes kG a Hopf algebra.

Example 4.10 (H_8 Kac-Paljutkin). We have seen already that H_8 is a bialgebra. We now define the antipode S on its generators. Both x and y are group-like, so it must be that $S(x) = x^{-1} = x$ and $S(y) = y^{-1} = y$. In fact, $S(z) = z$ as well, and this is a Hopf algebra. In fact, according to [3], it is the smallest dimensional Hopf algebra that is noncommutative, noncocommutative, and not a "twist" of a Hopf algebra that is.

The reader may have observed that in both examples above, S squares to the identity. We call such a map an "involution." We present some conditions under which S is guaranteed to be an involution, but first we need a definition.

Definition 4.11. *A Hopf algebra H is semisimple if its underlying algebra A is semisimple; in other words, if H is equal as an algebra to a direct sum $\oplus_i A_i$, where A_i is simple.*

Proposition 4.12. *If a Hopf algebra H is commutative, cocommutative or finite dimensional semisimple, then the antipode S of H is involutive. [2]*

The next example demonstrates a family of Hopf algebras having antipodes of order greater than 2.

Example 4.13 (Taft algebra). Let n be an integer greater than 0, and assume that k has a primitive n^{th} root of unity ζ. Take the free algebra $R = k\langle x, c\rangle$ and let I be the ideal of R generated by $c^n - 1$, x^n and $xc - \zeta cx$. Then the algebra $T = R/I$ along with ε and Δ defined a

$$\Delta(c) = c \otimes c,\ \Delta(x) = c \otimes x + x \otimes 1, \text{ and}$$

$$\varepsilon(c) = c,\ \varepsilon(x) = 0$$

is a bialgebra with dimension n^2. Finally, the antihomomorphism S where $S(c) = c^{-1}$ and $S(x) = -c^{-1}x$ makes T a Hopf algebra. The order of S is $2n$. [7]

4.4 Hopf Actions

Our purpose up to this point has been to outline the construction of an object in the category of vector spaces that emulates group actions, namely a Hopf algebra. We shift focus now to exploring actions of Hopf algebras in their own right. This is, after all, the ultimate goal of the software package we are providing background and motivation for.

Definition 4.14. *Let B be a k-bialgebra, and let A be a k-algebra. We say that B acts on A if A is a graded B-module and for all $b \in B$ and all $x, y \in A$,*

$$b(xy) = \sum h_{(1)}(a)h_{(2)}(b) \text{ and } b(1_A) = \varepsilon(h).$$

In other words, B acts on products of A via the coproduct. We call A a B-module algebra.

If B is a Hopf algebra, we call action by B on A a Hopf action; however, the definitions of Hopf action and bialgebra action differ only in whether or not the bialgebra acting on A possesses an antipode; they are mostly interchangeable.

Example 4.15 (Actions of H_8). Let A be the algebra $k\langle u, v\rangle/\langle u^2 + v^2\rangle$. Then H_8 acts on A by

$$x(u) = -u,\ y(u) = u,\ z(u) = v \text{ and}$$

$$x(v) = v,\ y(v) = -v,\ z(v) = u.$$

We now check that the action described above is well defined. In particular, we check that H_8 sends elements of $\langle u^2 + v^2\rangle$ back to $\langle u^2 + v^2\rangle$ (ensuring that $\forall h \in H_8$, $h(0_A) = 0_A$) and that A is an H_8-module, which amounts to checking that the relations of B act as 0 on the span of degree 1 monomials in A.

Proposition 4.16. *Example 4.15 is a valid Hopf action*

Proof. First, we check that the defining ideal of A, $\langle u^2 + v^2 \rangle$, is sent back to itself under action by H_8. Since $\langle u^2 + v^2 \rangle$ is spanned by $u^2 + v^2$, we examine action of H_8 on this element directly. Recall that H_8 acts on degree two monomials via the coproduct. Here are the coproducts of H_8:

$$\Delta(x) = x \otimes x$$

$$\Delta(y) = y \otimes y$$

$$\Delta(z) = \frac{1}{2}(1 \otimes 1 + 1 \otimes x + y \otimes 1 - y \otimes x)(z \otimes z).$$

Since x and y are group-like, it is easy to show that they both act as the identity on $u^2 + v^2$:

$$x(u^2 + v^2) = x(u)^2 + x(v)^2 = u^2 + v^2, \text{ and}$$

$$y(u^2 + v^2) = y(u)^2 + y(v)^2 = u^2 + v^2.$$

The coproduct of z makes this calculation tedious to do by hand. By linearity, $z(u^2 + v^2) = z(u^2) + z(v^2)$, so we will calculate each summand individually.

$$\begin{aligned} z(u^2) &= \frac{1}{2}(z \otimes z + z \otimes xz + yz \otimes z - zy \otimes xz)(u \otimes u) \\ &= \frac{1}{2}(z(u)^2 + z(u)xz(u) + yz(u)z(u) - zy(u)xz(u)) \\ &= \frac{1}{2}(v^2 + v^2 - v^2 + v^2) && = v^2 \end{aligned}$$

$$\begin{aligned} z(v^2) &= \frac{1}{2}(z \otimes z + z \otimes xz + yz \otimes z - zy \otimes xz)(v \otimes v) \\ &= \frac{1}{2}(z(v)^2 + z(v)xz(v) + yz(v)z(v) - zy(v)xz(v)) \\ &= \frac{1}{2}(u^2 - u^2 + u^2 + u^2) && = u^2 \end{aligned}$$

Therefore $z(u^2 + v^2) = u^2 + v^2$, and we have shown that H_8 sends 0_A to 0_A.

Now we check that the defining relations of H_8 act as 0 on A, implying A is a well-defined H_8-module. Recall the relations on H_8: x^2-1, y^2-1, $xy-yx$, $zx-yz$, $zy-xz$, and $z^2-\frac{1}{2}(1+x+y-xy)$. We go through another set of computations to ensure that action by 0_{H_8} is the zero action. These are simple for the most part, but the number of things to be checked in even this relatively small example should indicate to the reader the convenience of have a computer do this work "in the field."

$$(x^2-1)u = u-u = 0, \text{ and } (x^2-1)v = v-v = 0.$$

$$(y^2-1)u = u-u = 0, \text{ and } (y^2-1)v = v-v = 0.$$

$$(xy-yx)u = -u+u = 0, \text{ and } (xy-yx)v = -v+v = 0.$$

$$(zx-yz)u = -v+v = 0, \text{ and } (zx-yz)v = u-u = 0.$$

$$(zy-xz)u = v-v = 0, \text{ and } (zy-xz)v = -u+u = 0.$$

$$\left(z^2-\frac{1}{2}(1+x+y-xy)\right)u = u-\frac{1}{2}(u-u+u+u) = u-u = 0$$

$$\left(z^2-\frac{1}{2}(1+x+y-xy)\right)v = v-\frac{1}{2}(v+v-v+v) = v-v = 0$$

Therefore, A is an H_8-module algebra. □

Finally, we provide one last definition. The original goal of `HopfAlgebras.m2` was to calculate invariants of Hopf actions, so we would be remiss not to discuss them here briefly.

Definition 4.17. *Suppose bialgebra H has an action on A. Then we call the set*

$$\{a \in A \,|\, \forall h \in H, h(a) = \varepsilon(h)a\}$$

the invariant algebra of H in A, which we denote A^H.

This definition demands some explanation, as it is not immediately apparent that A^H is a sub-algebra.

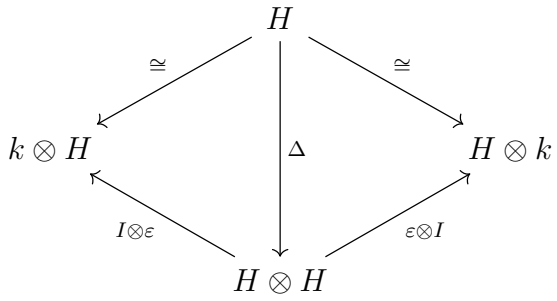
Proposition 4.18. *A^H as defined in definition 4.17 is a subalgebra.*

Proof. First we show that A^H is a k-subspace, and then show that it is also closed under algebra multiplication. Let $a, b \in A^H$, $h \in H$ and $c \in k$.

To see that A^H is a subspace, consider $a + cb$. H acts as a linear map, so $h(a + cb) = ha + c(hb)$. We may then apply a and b's status as elements of A^H to convert h to $\varepsilon(h)$:

$$ha + c(hb) = \varepsilon(h)a + c\varepsilon(h)b = \varepsilon(h)(a + cb).$$

Therefore A^H is a subspace of A. We now turn our attention to multiplication. First, recall the commutative diagram describing the relationship between Δ and ε:



We may derive the following computational rule from this diagram

$$h = \sum \varepsilon\left(h_{(1)}\right) h_{(2)} = \sum h_{(2)} \varepsilon\left(h_{(2)}\right)$$

and then apply ε to get

$$\begin{aligned} \varepsilon(h) &= \varepsilon\left(\sum \varepsilon\left(h_{(1)}\right) h_{(2)}\right) \\ &= \sum \varepsilon\left(h_{(1)}\right) \varepsilon\left(h_{(2)}\right). \end{aligned}$$

Now, act on the product ab with h:

$$\begin{aligned} h(ab) &= \Delta(h)(ab) \\ &= \sum \left(h_{(1)}a\right)\left(h_{(2)}b\right) \\ &= \sum \left(\varepsilon\left(h_{(1)}\right)a\right)\left(\varepsilon\left(h_{(2)}\right)b\right) \\ &= ab\sum \varepsilon\left(h_{(1)}\right)\varepsilon\left(h_{(2)}\right) \\ &= \varepsilon(h)ab. \end{aligned}$$

Therefore, A^H is a subalgebra of A.

□

Chapter 5: The code

In this chapter, we introduce `HopfAlgebras.m2` , a `Macaulay2` package for making calculations over a (possibly noncommutative and noncocommutative) bialgebra. The primary directive of `HopfAlgebras.m2` is to calculate invariants of Hopf actions, but it is also able to calculate

- the well-definedness of prospective bialgebras and bialgebra actions,
- the group-like elements of a bialgebra,
- the antipode of a prospective Hopf algebra.

In fact, no calculation in `HopfAlgebras.m2` (save the self-explanatory `findAntipode` command) uses the Hopf algebra structure on a bialgebra. We first give an overview of the `Macaulay2` system and then describe the data structures and algorithm of `HopfAlgebras.m2` with examples and pseudocode.

5.1 Background

The `Macaulay2` software system was developed by Daniel Grayson and Michael Stillman for the purpose of supporting research in commutative algebra and algebraic geometry. Users interface with the system via the `Macaulay2` language, a high-level scripting language with syntax that is intended to resemble the notation already used by mathematicians. [4] Everything in the `Macaulay2` system is an object with a "type" that classifies the object. Most types have ancestor types from which they inherit data. All objects share `Thing` as their most distant class ancestor, and most have `HashTable` as well.

We use HashTables often in HopfAlgebras.m2 to store information about objects, so they are worth some discussion now. Similar to the Dictionary from Python, Macaulay2 stores data in a HashTable by way of what are called "key-value" pairs; certain "values" (objects) may be associated to "keys," other objects. This is useful because it allows us to not only store information, but also associate two objects efficiently. For instance, we store the action of a bialgebra B as a set of key-value pairs, where each key is a basis element of B and the corresponding value is the corresponding endomorphism of the algebra being acted upon. HashTables also allow us to emulate classes of an object-oriented language, which is almost certainly why so many of the objects we use have it as a parent class.

A core feature of Macaulay2 is the use of Gröbner bases to reduce quotient rings and solve systems of equations. While the Macaulay2 engine computes commutative Gröbner bases automatically, our interest lies primarily in working over noncommutative Hopf algebras. The Macaulay2 engine does not (at the time of writing) possess the machinery that allows for calculations over noncommutative rings, so HopfAlgebras.m2 takes advantage of the NCAlgebra.m2 package bundled with Macaulay2. NCAlgebra.m2 provides the types NCRing, NCIdeal and NCQuotientRing, which we use to represent the algebra structure of a bialgebra in our package, as well as the NCMatrix and NCRingMap types that we use to represent linear maps and algebra homomorphisms. NCAlgebra.m2 in turn uses a CLISP package called bergman to compute noncommutative Gröbner bases. If a bialgebra presentation is given with homogeneous relations between the generators, NCAlgebra.m2 can fetch a Gröbner basis directly from bergman. If the defining relations are not homogeneous, then bergman can still attempt to produce a Gröbner basis using an algorithm its documentation refers to as the "jumping rabbit." While we generally found success with this strategy,

the resulting Gröbner basis must be installed into `Macaulay2` by hand. [1]

5.2 Data structures

We now give an overview of how mathematical objects are represented and stored in `HopfAlgebras.m2` .

5.2.1 Tensor products

The first design challenge of `HopfAlgebras.m2` was developing a suitable implementation of the tensor product for the family of `NCRing` types from the `NCAlgebra.m2` package. We required our standard to allow for the automation of tensor product constructions by `Macaulay2` , without sacrificing a great deal of legibility for the end user. Our tensor algorithm writes a presentation that fulfills these requirements using the following isomorphism:

Fact 5.1. *For vector spaces V_1 and V_2 with respective algebra relations R_1 and R_2, we have that*

$$T(V_1 \oplus V_2)/\langle R_1, R_2, C\rangle \cong T(V_1)/R_1 \otimes T(V_2)/R_2,$$

where C is the ideal generated by the additional tensor commutivity relation.

Hence, if A is a k-algebra with a presentation given by generators G_A and relations R_A, and B is similarly a k-algebra with generators G_B and relations R_B, `HopfAlgebras.m2` constructs the tensor product $A \otimes B$ as the quotient algebra

$$k\left\langle G_A \bigcup G_B \right\rangle / \langle R_A, R_B, C\rangle,$$

where, for $x, y \in G_A \bigcup G_B$, C is the ideal generated by elements of the form $xy - yx$ (these are the "additional commutivity relations" of fact 5.1 above). It represents elements of $A \otimes B$ as doubly subscripted sums of generators: the second subscript

denotes which algebra the generator is from (which is equivalent to which "slot" of the simple tensor it sits in), while the first subscript indicates which generator it is (user supplied symbols are replaced with a uniform `xx` symbol to avoid confusion). For example, if A is an algebra generated by x and y, `HopfAlgebras.m2` would represent $xy \otimes yx^2 \in A^{\otimes 2}$ as $\texttt{xx}_{1,1}\texttt{xx}_{2,1}\texttt{xx}_{2,2}\texttt{xx}_{1,2}^2$.

The processes described above are implemented in `HopfAlgebras.m2` via the `tensorProductNoGB` and `createSubscriptRing` methods. `tensorProductNoGB` takes two `NCRings` `A` and `B` over the same field `kk` and returns a third `NCRing` representing their tensor product. It is called "`tensorProductNoGB`" to differentiate it from the `tensorProduct` command of `NCAlgebras`, which constructs a new Gröbner basis instead of inferring one from previously defined Gröbner bases.

The function `tensorProductNoGB` does not work if two generators share a symbol. For instance, if `A` and `B` are two `NCRings` such that `gens A = x, y, z` and `gens B = w,x`, then `tensorProductNoGB(A,B)` will produce an error, even if the symbol `x` means different things in `A` and `B`. Since we use `tensorProductNoGB` almost exclusively to compute tensor powers when defining the coproduct of a bialgebra, we circumvent this problem with the `createSubscriptRing` command, which changes the symbols of an `NCRing` to match the aforementioned double subscripting notation.We should note here that the `createSubscriptRing` command and the double subscripts are primarily for internal bookkeeping in `HopfAlgebras.m2` , and are not in general revealed to the user.

5.2.2 NCBialgebra

The `NCBialgebra` type represents bialgebras and Hopf algebras. Its parent type is `HashTable`. Each `NCAlgebra` saves the following data of a bialgebra that it represents.

- The algebra structure is stored as an `NCRing`.
- The counit is stored as an `ncMap` from the algebra to itself. Since the actual range of a counit is just k, this is really the map $u \circ \varepsilon$, which aids in some internal functions.
- The coproduct maps are stored in a new mutable hashtable as `ncMap`s from a subscripted copy of the `NCRing` to the appropriate subscripted tensor power thereof.
- Finally, the isomorphism from the algebra structure to its first subscripted copy; this is not mathematically interesting, and it is saved only to provide the user a convenient method of accessing the tensor algebra.

An `NCBialgebra` is constructed via the `bialgebra` method, which takes an `NCRing` representing the underlying algebra and two lists representing the images of generators under the coproduct and augmentation respectively. The list of coproduct images deserves further clarification. It is written as a list of lists of sequences using user-defined symbols (as opposed to our doubly subscripted ones). The outermost list is a list of images and inner list contains the simple tensor summands of each image, each of which is represented by a sequence of length 2. For example, the following command constructs Kac-Paljutkin H_8:

```
A = QQ{x,y,z}
I = ncIdeal {x^2 - 1_A, y^2 - 1_A, x*y - y*x, z*x - y*z, z*y - x*z,
```

```
   z^2 - (1/2)*(1_A + x + y - x*y)}
-- we used bergman, and must reassure Macaulay2 that this is a
-- Groebner basis in the below command.
Igb = ncGroebnerBasis(I, InstallGB => true)
B = A/I
H = bialgebra(B,{{(x,x)},{(y,y)},{(1/2*z,z),(1/2*z,x*z),(1/2*y*z,z),
   (-1/2*y*z,x*z)}},{1_QQ,1_QQ,1_QQ})
```

Commands are provided that retrieve the underlying `NCRing`, counit (as an `NCMap`) and inclusion map of an `NCBialgebra`. The command `higherCoprod` is also provided to access coproducts of the bialgebra. Unlike the other "accessor" methods, `higherCoprod` takes an additional integer argument `n`, and returns the `NCMap` from the underlying algebra to the n^{th} tensor power thereof. `bialgebra` only computes and records the coproduct from B to $B \otimes B$; compositions of higher order than that are computed and cached only as they are requested by `higherCoprod`.

5.2.3 BialgebraAction

The `BialgebraAction` type represents the graded action of a bialgebra on a graded algebra. Its parent type is also `HashTable`. Each `BialgebraAction` saves the bialgebra B as an `NCBialgebra B` and the algebra A being acted upon as an `NCRing A`. It saves actions of `B` on `A` as a `HashTable` with basis elements of `B` as keys and the associated actions, represented by `NCMaps` from `A` to `A`, as values. Finally, each `BialgebraAction` saves an extra empty hashtable for recording the actions of `B` on higher degree monomials of `A`; this keeps us from having to make the same computationally expensive calculations over and over again, and significantly boosts the performance of `HopfAlgebras.m2` 's `actOn` command (which we'll discuss in detail later).

The constructor for `BialgebraAction` (aptly named `bialgebraAction`) takes an `NCBialgebra` and a list of the actions of the generators of `B` on `A`, represented again as `NCMaps`. `bialgebraAction` then calls a separate `HopfAlgebras.m2` method, `totalBasis`, to compute a basis of `B`, and records the action of each basis element in the hashtable. `bialgebraAction` infers the `NCRing` being acted on by reading the source of the maps in the provided list.

5.2.4 DualElement and DualBialgebra

The types `DualElement` and `DualBialgebra` are used for working with the dual of a bialgebra. `HopfAlgebras.m2` realizes elements of the dual of some bialgebra B as row vectors of length n, where n is the vector dimension of B. For instance, if V was a four-dimensional bialgebra with basis $\{1, x, y, xy\}$, the dual element that sends xy to 1 and the other basis elements to zero would be recorded as the row vector `[0,0,0,1]`. Hence, the `DualElement` type is really just a wrapper class for 1 by n matrices that also saves the `DualAlgebra` that element belongs to. `DualBialgebra` represents the vector space of linear functionals of some bialgebra. It saves the `NCBialgebra` it is dual to and a dual basis derived from a basis of algebra it is dual to. More importantly, it contains rules for making computations in the dual. If `phi` and `psi` are `DualElements` of the same `NCBialgebra`, for example, `DualBialgebra` tells `Macaulay2` how to interpret input like `phi*psi` (remember that multiplication in B^* is the convolution product), `phi == psi` and even `phi**psi`, the tensor product of linear maps.

A `DualBialgebra` may be constructed via the `dualize` command, which takes a single argument, the `NCBialgebra` being dualized. Objects of the `DualElement` type have several disparate constructors that will be discussed later. The primary

one is called `dualElement`. It takes a `DualBialgebra` and a `Matrix` (the standard `Macaulay2` type representing matrices over a field), and produces a `DualElement` corresponding to that matrix. If the `Matrix` entered is not a $1 \times n$ matrix (where n is the dimension of the `NCBialgebra`), an error is thrown.

5.3 Algorithms

Now, we give overviews of some of the more mathematically interesting methods available in `HopfAlgebras.m2` .

5.3.1 isBialgebra and isWellDefined

The commands `isBialgebra` and `isWellDefined` may be used respectively to check whether an `NCBialgebra` or `BialgebraAction` fulfills the axioms of their respective definitions. While these commands are obviously useful in that they check whether a prospective bialgebra or Hopf action is mathematically sound, we found that they were also very helpful in ensuring that a bialgebra we already knew was well-defined had been entered into `Macaulay2` correctly. We will first describe `isBialgebra`, and then move on to `isWellDefined`.

isBialgebra

Recall that the tuple $(B, \mu, u, \Delta, \varepsilon)$ is a bialgebra if (B, μ, u) is an algebra, (B, Δ, ε) is a coalgebra, and Δ and ε are algebra homomorphisms. To check that an `NCBialgebra` object possesses a valid coalgebra structure, `HopfAlgebras.m2` contains the commands `isCounit` and `isCoassociative`. The package `NCAlgebras` extends the functionality of the `isWellDefined` command included with `Macaulay2` to work on `NCMaps`.

Hence, we use `isWellDefined` to check that Δ and ε are valid algebra homomorphisms.

The command `isCoassociative` takes one argument, an `NCBialgebra B`, and returns a Boolean value indicating whether or not `B` is coassociative. Our algorithm constructs the maps $\Delta \otimes I \circ \Delta$ and $I \otimes \Delta \circ \Delta$, and checks to see if they are equivalent. The former composition is the output of `higherCoprod`, so `isCoassociative` performs a similar algorithm on the right, and then compares matrices of the two.

Like `isCoassociative`, the command `isCounit` takes an `NCBialgebra B` as an argument and returns a Boolean value indicating whether or not the counit of `B` satisfies the second commutative diagram in figure 3.2, which is equivalent to showing that the maps $I \otimes \varepsilon \circ \Delta$ and $\Delta \otimes I \circ \Delta$ are both the identity map on B. This is done by simply composing the maps and then checking that the associated matrices have the same values.

These two commands, along with `isWellDefined` applied to Δ and ε, are the meat and bones of `isBialgebra`, which returns true on an `NCBialgebra B` only if `isCoassociative B`, `isCounit B` and `isWellDefined higherCoprod(B,1)` all return true.

isWellDefined

The command `isWellDefined` is extended to work on `BialgebraAction` objects; in a nutshell, it performs the same checks on a prospective bialgebra action that we performed in our defense of example 4.16. Suppose that the `BialgebraAction action` represents the action of an `NCBialgebra B` on an `NCRing A`. Calling `isWellDefined`

on `action` causes `Macaulay2` to run through the action of every generator of `B` on the generators of the defining ideal of `A`, returning false if it does not annihilate the ideal. It then checks that `A` is a `B` module by acting on each generator of `A` with each generator of `B`'s defining ideal. If every action returns `0` again, `isWellDefined` returns true to indicate that `action` is a well-defined bialgebra action.

5.3.2 findGrouplikes

The command `findGrouplikes` allows the user to calculate the space of group-like elements of a bialgebra B, as well as the group-like linear functionals in B^*. We will start by examining what happens when `findGrouplikes` is applied to an `NCBialgebra`, and then cover the dual case. Recall that an element $g \in B$ is group-like if $\Delta g = g \otimes g$ and $\varepsilon(g) = 1_k$. In both the bialgebra and dual bialgebra case, our algorithm's strategy is the same: it finds a basis $\{b_j \mid j = 1, \ldots, n\}$ of B and solves the system of equations induces by the two equalities

$$\Delta\left(\sum_{i=1}^{n} c_i b_i\right) = \left(\sum_{i=1}^{n} c_i b_i\right) \otimes \left(\sum_{i=1}^{n} c_i b_i\right), \text{ and}$$

$$\varepsilon\left(\sum_{i=1}^{n} c_i b_i\right) = 1_k.$$

We deviate here for a moment to discuss our general algorithm for solving systems of equations over an `NCBialgebra B`, as well as why we would want to. Suppose (as above) that we wanted to calculate the set of elements in some bialgebra that satisfy some conditions. We use an internal function called `extendCoeffRing` to add enough variables to the base field to give ourselves an arbitrary element of `B`. For instance, if `B` had dimension 12, we would add 12 commuting variables to the base field. We then impose the desired conditions on our arbitrary element, and use a command from `NCAlgebras` called `sparseCoeffs` to get the resulting relations on

our added variables. We then calculate the ideal generated by the relations, and use the `primaryDecomposition` command from the `PrimaryDecomposition` package to decompose it. Finally, we take the quotient of the base field by each piece of the decomposition to get a solution.

We now walk through a concrete example: the algorithm as it is applied to the Hopf algebra H_8. Suppose H_8 is being represented by the `ncBialgebra B` with underlying `NCAlgebra H`, which in turns sits on top of the field `kk`. When we input the command `findGrouplikes B`, `Macaulay2` calculates a basis `basisH` of `H`, and extends `kk` by adjoining the variables `cc_1` through `cc_8`, since H_8 has dimension 8. The sum of basis elements with coefficients given by the `cc_i`'s is given the name `g`, and we have our arbitrary Hopf element. The computer proceeds to calculate Δ applied to `g`, and subtracts from this the element corresponding to `g`$\otimes$`g`. We set the resulting sum equal to zero, and take the ideal in $H_8 \otimes H_8$ generated by all the relations on `kk[cc_1, ..., cc_8]` that we have just created, as well as the relations we get by setting ε`g` equal to one. This ideal decomposes as the intersection of four ideals, each of which defines a single point. These points correspond to the solutions of our system and hence the grouplikes of H_8: 1, x, y and xy.

5.3.3 findAntipode

The command `findAnitpode` calculates the antipode of a bialgebra, if there is one. We ultimately do this via the algorithm discussed above, but we must first handle the concept of an antihomomorphism. Neither `Macaulay2` nor `NCAlgebra` contains a type that represents an antihomomorphism of algebras, so we store the antipode as a `NCMap` and apply it via a special command, `applyAntihomomorphism`. This method takes an `ncMap` and an element in the domain of that map as arguments. Then it

simply reverses the order of elements in each monomial of the element, and applies the `ncMap` as if it were a homomorphism.

Suppose we apply the `findAntipode` command to an `NCBialgebra B`. The algorithm uses the methods of section 5.3.2 to generate an arbitrary element of an `NCBialgebra` for each generator on `B`. It then constructs an `NCMap S` from `B` to itself, sending each generator to one of the arbitrary elements. `S` is, in a sense, an arbitrary endomorphism of `B`. It then uses a series of compositions and calls to `applyAntihomomorphism` to construct the convolution product of `S` with the identity function on `B`, subtracts the counit of each generator from its image under `S`, and uses `sparseCoeffs` and `primaryDecomposition` to solve the resulting system of equations.

5.3.4 actOn

The command `actOn` is used to compute the action of a bialgebra element on an element of its module algebra. It takes three arguments: the bialgebra element `h` and the algebra element `a` being acted on as `NCRingElement`s, and a `BialgebraAction`. The `actOn` command processes both `h` and `a` into disparate lists of coefficients and monomials. It then feeds each possible combination of monomials derived from `h` and `a` into a separate, internal function called `actOnMonomial`.

Like `actOn`, `actOnMonomial` takes three arguments of the same type. However, `actOnMonomial` will return an error if either of the first two arguments are not monomials (this is a redundancy, as `actOnMonomial` is only called internally). When `actOnMonomial` is called on an `NCBialgebra` monomial `b`, a `NCRing` monomial `x` and a `BialgebraAction action`, it returns a scalar if the `x` is degree 0 and computes

the action as defined in the hashtable of `action` if `x` is degree 1. If `x` has degree greater than one, `actOnMonomial` looks first in the cache on the `BialgebraAction` to see if the image of this action has been computed, and returns it immediately if so. If not, it must compute the action from scratch. It does so recursively, by first splitting `x` into two factors, taking the coproduct of `b`, and then calling itself on each factor of `x` with the corresponding side of the tensor product. This method keeps `Macaulay2` from having to compute higher order coproducts, which we found to be computationally untenable. Once the image of an action is computed, it is stored in the cache of `action`. This is particularly useful given the branching, recursive nature of this algorithm, as we are otherwise making many of the same computations over and over again.

5.3.5 invariantAlgebra

The method `invariantAlgebra` stands on the back of several methods. We introduce and discuss them in reverse order of dependence now.

The command `actionInDegree` takes an `NCRingElement h` of a bialgebra, a `BialgebraAction` representing an action by that bialgebra, and an integer `n`. It returns a `Matrix` representing the action of `h` on the degree `n` component of whatever algebra the `BialgebraAction` is acting on. Recall that the invariants A^H of an H-module algebra A are the $x \in A$ such that for all $h \in H$, $hx = \varepsilon(h)x$. `invariantInDegree` computes a basis of the invariants in degree `n` by finding a basis of the intersection of null space of matrices

`actionInDegree(h,action,n)` - $(\varepsilon$ `h`$)\cdot$`id`

where `h` is a generator of `H` and "id" is the square matrix of rank equal to the dimension of the degree `n` component of the algebra being acted on. It does so by vertically

concatenating the generated matrices and finding one basis.

The commands `invariantAlgebraGens` and `getNextDegInvar` are then used to compute a basis of A^H up to a given degree n. `getNextDegInvar` takes a `BialgebraAction`, a `List` of previously found invariant generators, and an integer `n` as arguments. It calls `invariantInDegree` to get a list of potential generators of the invariant algebra in degree `n`. It then checks to see if any of them can be generated by the generators that have been calculated thus far by considering the quotient of the span of the new generators by the span of the old generators. If any truly new generators are found, `getNextDegInvar` adds them to the list that it was passed, and returns the list; otherwise it does nothing and returns the list it was passed. `invariantAlgebraGens` computes all the invariant algebra generators up to degree n by looping `getNextDegInvar` through the integers from 1 to n, and saves them in the cache of the action. Finally, `invariantAlgebra` wraps everything up neatly by using `invariantAlgebraGens` to build a basis of the invariant algebra and using the `gddkernel` command from `NCAlgebras` to compute the ideal of relations. It then formulates the invariant algebra by taking the quotient of the span of the basis by the ideal of relations. It then caches everything, calculates an `NCMap` into the original algebra, and returns the invariant algebra.

.1 Code Listing

```
newPackage("HopfAlgebras",
     Headline => "Data types for Hopf algebras",
     Version => "0.9",
     Date => "Nov 27, 2018",
     Authors => {
          {Name => "Colin Martin",
           HomePage => "http://www.google.com",
           Email => "martce17@wfu.edu"},
          {Name => "Frank Moore",
           HomePage => "http://www.math.wfu.edu/Faculty/
              Moore.html",
           Email => "moorewf@wfu.edu"}},
     PackageExports => \{"NCAlgebra"\},
     PackageImports => {"PrimaryDecomposition"},
     AuxiliaryFiles => true,
     DebuggingMode => true,
     CacheExampleOutput =>true
     )

export {"NCBialgebra",
    "BialgebraAction",
    "DualBialgebra",
    "DualElement",
    "ConvolutionElement",
    "bialgebra",
    "getDeltaN",
    "higherCoprod",
    "tensorProductNoGB",
    "isCoassociative",
    "isCounit",
    "isBialgebra",
    "isCocommutative",
    "totalBasis",
    "findLeftIntegral",
    "findGrouplikes",
    "findAntipode",
    "applyAntihomomorphism",
    "bialgebraAction",
    "actOn",
    "actionInDegree",
    "invariantInDegree",
    "invariantAlgebraGens",
```

```
    "invariantAlgebra",
    "RelDegreeLimit",
    "GenDegreeLimit",
    "inclusion",
    "counit",
    "dualCoprod",
    "multMatrix",
    "dualBasis",
    "dualCounit",
    "dualize",
    "dualElement",
    "weightSpaceInDegree",
    "getNextDegWeights",
    "weightSpaceGens"}

protect atlas
protect generatorAction
protect actionCache
protect leftTens
protect rightTens
protect iota1
protect iota2
protect deltas
protect deltaImg
protect coMult
protect mult
protect invarGens
protect invarAlg

NCBialgebra = new Type of HashTable
BialgebraAction = new Type of HashTable
DualBialgebra = new Type of Ring
DualElement = new Type of HashTable
ConvolutionElement = new Type of HashTable

--- we need some functions from NCAlgebra that are not
   exported
debug NCAlgebra

makeMonic = method()
makeMonic NCRingElement := f -> (leadCoefficient f)^(-1)*f
```

```
fastBaseChange = method()
fastBaseChange(NCRing, NCRingElement, RingMap) := (B, f,
   phi) -> (
   -- B should be a NCRing which only differs from the
   -- ring of f in that one may promote elements of its
   -- coefficient ring to that of B.
   if (ring f).generatorSymbols != B.generatorSymbols then
       error "Expected same variables.";
   retVal := new B from {(symbol ring) => B,
       (symbol cache) => new CacheTable from {("isReduced
          ",true)},
       (symbol terms) => new HashTable from apply(pairs f.
          terms, (k,v) -> (new NCMonomial from {(symbol
          monList) => k.monList, (symbol ring) => B}, phi
          v))};
   retVal
)

fastBaseChange (NCRing, NCRingElement) := (B, f) -> (
   phi := map(coefficientRing B, coefficientRing ring f);
   fastBaseChange(B,f,phi)
)

createSubscriptRing = method()
createSubscriptRing (NCRing, ZZ) :=  (A, n) -> (
   -- this ring creates a `copy" of A but with the
      variables
   -- doubly subscripted with last entry n, and the first
      subscript the variable number.

   R := coefficientRing A;
   I := ideal A;
   ambA := ambient A;
   xx := getSymbol "xx";
   newVars := apply(numgens A, i -> xx_(i+1,n));

   ambB := R newVars;
   phi := ncMap(ambB, ambA, gens ambB);
   newI := ncIdeal ((gens I) / phi);
   if class A === NCPolynomialRing then ambB else (
       newI.cache#gb = ncGroebnerBasis((gens (I.cache#gb))
           / phi, InstallGB => true);
```

```
        ambB/newI
    )
)

createSubscriptRing (NCBialgebra, ZZ) := (B,n) ->
   createSubscriptRing(B.ring,n)

tensorProductNoGB = method()
tensorProductNoGB(NCRing, NCRing) := (A1,A2) -> (
   R := coefficientRing A1;
   if coefficientRing A2 =!= R then error "Expected the
      same coefficient ring.";

   A1gens := (gens A1) / baseName;
   A2gens := (gens A2) / baseName;
   newA := R (A1gens | A2gens);
   newA1gens := take(gens newA,numgens A1);
   newA2gens := drop(gens newA,numgens A1);
   phi := ncMap(newA,ambient A1,newA1gens);
   psi := ncMap(newA,ambient A2,newA2gens);
   commRules := flatten apply(newA1gens, x -> apply(
      newA2gens, y -> x*y - y*x));
   A1I := ideal A1;
   A2I := ideal A2;
   J := ncIdeal (((gens A1I) / phi) | ((gens A2I) / psi) |
       commRules);
   gbJGens := ((gens (A1I.cache#gb)) / phi) | ((gens (A2I.
      cache#gb)) / psi) | commRules;
   J.cache#gb = ncGroebnerBasis( gbJGens, InstallGB =>
      true);
   A := newA/J;
   A.cache#(symbol iota1) = ncMap(A,A1,take(gens A,
      numgens A1));
   A.cache#(symbol iota2) = ncMap(A,A2,drop(gens A,
      numgens A1));
   A
)

tensorProductNoGB(NCRingMap,NCRingMap) := (f,g) -> (
   A1 := source f;
   A2 := source g;
   B1 := target f;
```

```
   B2 := target g;
   A12 := tensorProductNoGB(A1,A2);
   B12 := tensorProductNoGB(B1,B2);
   ncMap(B12,A12,((flatten entries matrix f) / B12.cache#
      iota1) | ((flatten entries matrix g) / B12.cache#
      iota2))
)

bialgebra = method()
bialgebra (NCRing, List, List) := (A, deltaOutput,
   augOutput) -> (
   A1 := createSubscriptRing(A,1);
   A2 := createSubscriptRing(A,2);
   tensA := tensorProductNoGB(A1,A2);
   phi1 := ncMap(tensA,A,take(gens tensA, numgens A));
   phi2 := ncMap(tensA,A,drop(gens tensA, numgens A));
   delta := ncMap(tensA, A1, apply(deltaOutput, d -> sum
      apply(d, t -> (phi1(t#0))*(phi2(t#1)))));
   B := new NCBialgebra from {(symbol ring) => A,
                              (symbol counit) => ncMap(A,A
                                 ,augOutput),
                              (symbol deltas) => new
                                 MutableHashTable from
                                 {(1,delta)},
                              (symbol inclusion) => ncMap(
                                 A1,A,gens A1),
                              (symbol leftTens) => ncMap(A
                                 , tensA, gens A | apply(
                                 numgens A, i -> 1_A)),
                              (symbol rightTens) => ncMap(
                                 A, tensA, apply(numgens A
                                 , i -> 1_A) | gens A),
                              (symbol deltaImg) =>
                                 deltaOutput,
                              (symbol cache) => new
                                 CacheTable from {}};
   B
)

ring NCBialgebra := B -> B.ring

counit = method()
```

```
counit NCBialgebra := B -> B.counit

inclusion = method()
inclusion NCBialgebra := B -> B.inclusion

bialgebra NCRing := A -> (
   --- This method constructs a bialgebra using the group
      bialgebra structure
   gensA := gens A;
   protodOutput := apply((pairs gensA), e -> apply(e, i ->
       e#1));
   dOutput := toList apply(protodOutput, e-> {e});
   aOutput := apply(#gensA, i -> 1_A);
   B := bialgebra(A,dOutput,aOutput);
   B
)

getDeltaN = method()
getDeltaN(NCBialgebra,ZZ) := (B,n) -> (
   --- this function returns the n^th iterate of Delta
      from H1 to H1\otimes H2\otimes\cdots\otimes Hn
   --- If it has already been computed, it returns it.  If
       not, the code stashes it (as well as all the
   --- intermediate steps)
   if n <= 0 then error "Expected positive integer";
   if B.deltas#?n then return B.deltas#n;
   --- if here, the map has not yet been built.
   DeltaNMinus1 := getDeltaN(B,n-1);
   Delta1 := getDeltaN(B,1);
   BNPlus1 := createSubscriptRing(B,n+1);
   --- the second factor is cached as the source of the
      inclusion map into B12
   B2 := source ((target B.deltas#1).cache#iota2);
   psi := ncMap(BNPlus1,B2,gens BNPlus1);
   deltaN := tensorProductNoGB(DeltaNMinus1,psi);
   DeltaN := deltaN @@ (ncMap(source deltaN, target Delta1
      , gens source deltaN)) @@ Delta1;
   B.deltas#n = DeltaN;
   DeltaN
)

higherCoprod = method()
```

```
higherCoprod (NCBialgebra, ZZ) := (B, n) -> (
   --- this method returns the nth coproduct as a
      NCRingMap from B to B^{\otimes n+1}
   --- which also involves building the rings in question
   --- it does so recursively
   DeltaN := getDeltaN(B,n);
   DeltaN @@ B.inclusion
)

isCoassociative = method()
isCoassociative NCBialgebra := B -> (
   Delta2L := getDeltaN(B,2);
   delta1 := B.deltas#1;
   B1 := source delta1;
   B2 := source ((target delta1).cache#iota2);
   B3 := source ((target B.deltas#2).cache#iota2);
   B23 := tensorProductNoGB(B2,B3);
   B12 := target delta1;
   psi3 := ncMap(B23,B12,gens B23);
   psi2inv := ncMap(B1,B2,gens B1);
   delta2R := tensorProductNoGB(ncMap(B1,B1,gens B1), psi3
       @@ delta1 @@ psi2inv);
   Delta2R := delta2R @@ (ncMap(source delta2R, target
      delta1, gens source delta2R)) @@ delta1;
   phi := ncMap(target Delta2R, target Delta2L, gens
      target Delta2R);
   matrix Delta2R == phi matrix Delta2L
)

isCounit = method()
isCounit NCBialgebra := B -> (
   H := B.ring;
   H12 := target B.deltas#1;
   mu := ncMap(H,H12, gens H | gens H);
   eps12 := (values B.counit.functionHash) / B.deltas#1 @@
       B.inclusion;
   epsLeft := ncMap(H12,H12, eps12 | drop(gens H12,numgens
       H));
   epsRight := ncMap(H12,H12, take(gens H12,numgens H) |
      eps12); -- requires H.counit to be elements of H!
   idH := ncMap(H,H,gens H);
   epsTimesId := mu @@ epsLeft @@ B.deltas#1 @@ B.
```

```
      inclusion;
   idTimesEps := mu @@ epsRight @@ B.deltas#1 @@ B.
      inclusion;
   (matrix epsTimesId == matrix idH) and (matrix
      idTimesEps == matrix idH)
   --- Checks that (id \otimes counit) \circ \Delta = id
      and (counit \otimes id) \circ \Delta = id.
)

isBialgebra = method()
isBialgebra NCBialgebra := B -> (
  --- isWellDefined on comultiplication
  --- isCounit and isCoassociative
  isWellDefined B.deltas#1 and isCounit B and
     isCoassociative B
)

isCocommutative = method()
isCocommutative NCBialgebra := B -> (
   --- to check cocommutativity only need to check
      generators
   H := B.ring;
   H12 := target B.deltas#1;
   twistMap := ncMap(H12,H12, drop(gens H12,numgens H) |
      take(gens H12,numgens H));
   comultWithTwist := twistMap @@ B.deltas#1 @@ B.
      inclusion;
   comultWithoutTwist := B.deltas#1 @@ B.inclusion;
   (matrix comultWithTwist) == (matrix comultWithoutTwist)
)

totalBasis = method(Options => {DegreeLimit => 10})
totalBasis NCRing := opts -> H -> (
   if H.cache#?basis then return H.cache#basis;
   basisH := ncMatrix{select(apply(opts#DegreeLimit, i ->
      basis(i,H)), m -> m != 0)};
   H.cache#basis = basisH;
   basisH
)

totalBasis NCBialgebra := opts -> B -> totalBasis(B.ring,
   opts)
```

```
coprodImg = method()
coprodImg (List, NCBialgebra) := (eltList,B) -> (
   --- This method takes an NCBialgebra and a list of
      NCBialgebra elements, and returns list of their
      coproducts
   if any(eltList, m -> not instance(m,B.ring)) then error
       "Expected list of ring elements.";
   apply(eltList, m -> apply(terms (B.deltas#1 B.inclusion
       m), t -> (B.leftTens t, (leadCoefficient t)^(-1)*(B
      .rightTens t))))
)

comultMatrix = method(Options => {DegreeLimit => 10})
comultMatrix NCBialgebra := opts -> B -> (
   --- Returns a matrix representation of the coproduct
      with respect to the basis calculated by Macaulay
   if B.cache#?coMult then return B.cache#coMult;
   H := B.ring;
   basisH := flatten entries totalBasis(H,opts);
   coprodBasis := coprodImg(basisH, B);
   comultMat := matrix{apply(coprodBasis, coprod ->
      sum apply(coprod, t -> sparseCoeffs(t#0, Monomials
         => basisH) ** sparseCoeffs(t#1, Monomials =>
         basisH)))};
   B.cache#coMult = comultMat;
   comultMat
)

multMatrix = method(Options => {DegreeLimit => 10})
multMatrix NCBialgebra := opts -> B -> (
   if B.cache#?mult then return B.cache#mult;
   H := B.ring;
   basisH := flatten entries totalBasis(H,opts);
   matrix {apply(basisH, m -> totalLeftMultMap(m, B))}
)

dualCoprod = method(Options => {DegreeLimit => 10})
dualCoprod DualElement := opts -> a -> (
   aMat := a.matrix;
   B := a.ring.bialgebra;
   aMat * multMatrix(B,opts)
```

```
)

totalLeftMultMap = method()
totalLeftMultMap (NCRingElement, NCBialgebra) := (h,B) ->
   (
   basisH := flatten entries totalBasis B;
   output := apply(basisH, f -> h*f);
   sparseCoeffs(output, Monomials => basisH)
)

findLeftIntegral = method(Options => {DegreeLimit => 10})
findLeftIntegral NCBialgebra := opts -> B -> (
   H := B.ring;
   basisH := totalBasis(B,opts);
   n := #(basisH.source);
   kk := coefficientRing H;
   matList := apply(gens H, x -> totalLeftMultMap(x,B)-
      leadCoefficient(B.counit x)*id_(kk^n));
   intCoeff := gens ker matrix apply(matList, m -> {m});
   leftInt := first first entries (basisH*intCoeff);
   leftInt
)

extendCoeffRing = method()
extendCoeffRing (NCBialgebra, ZZ, Symbol) := (B,n,c) -> (
   H := B.ring;
   I := ideal H;
   A := ambient H;
   kk := (A.CoefficientRing)[c_1..c_n];
   newA := kk(H.generatorSymbols);
   toNewA := ncMap(newA,A,gens newA);
   newI := ncIdeal ((I.generators)/toNewA);
   newIgb := ncGroebnerBasis((gens I.cache#gb) / toNewA,
      InstallGB => true);
   newI.cache#gb = newIgb;
   newH := newA/newI;
   toNewH := ncMap(newH,H, gens newH);
   newDeltaOutput := apply(B.deltaImg, l -> apply(l, m ->
      apply(m, n -> toNewH n)));
   bialgebra(newH,newDeltaOutput,(values B.counit#
      functionHash)/toNewH)
   )
```

```
extendCoeffRing2 = method()
extendCoeffRing2 (NCBialgebra, ZZ, Symbol) := (B,n,c) -> (
   H := B.ring;
   I := ideal H;
   A := ambient H;
   kk := (A.CoefficientRing)[c_1..c_n];
   newA := kk(H.generatorSymbols);
   phi := map(kk, coefficientRing A);
   toNewA := f -> fastBaseChange(newA, f, phi);
   newI := ncIdeal ((I.generators)/toNewA);
   newIgb := ncGroebnerBasis((gens I.cache#gb) / toNewA,
      InstallGB => true);
   newI.cache#gb = newIgb;
   newH := newA/newI;
   toNewH := f -> fastBaseChange(newH, f, phi);
   newDeltaOutput := apply(B.deltaImg, l -> apply(l, m ->
      apply(m, n -> toNewH n)));
   bialgebra(newH,newDeltaOutput,(values B.counit#
      functionHash)/toNewH)
   )

addRootsOfUnity = method()
addRootsOfUnity (NCBialgebra, ZZ, Symbol) := (B,n,c) -> (
   H := B.ring;
   I := ideal H;
   A := ambient H;
   kk := (A.CoefficientRing);
   newA := kk(H.generatorSymbols);
   toNewA := ncMap(newA,A,gens newA);
   newI := ncIdeal ((I.generators)/toNewA);
   newIgb := ncGroebnerBasis(newI, InstallGB => true);
   newH := newA/newI;
   toNewH := ncMap(newH,H, gens newH);
   newDeltaOutput := apply(B.deltaImg, l -> apply(l, m ->
      apply(m, n -> toNewH n)));
   bialgebra(newH,newDeltaOutput,(values B.counit#
      functionHash)/toNewH)
)

findGrouplikes = method(Options => {DegreeLimit => 10})
findGrouplikes NCBialgebra := opts -> H -> (
```

```
    c := getSymbol "cc";
    basisH := totalBasis(H,opts);
    Hc := extendCoeffRing(H, #(basisH.source), c);
    basisHc := totalBasis Hc;
    kk := coefficientRing Hc.ring;
    varsAmbient := transpose matrix {gens kk};
    g := first flatten entries (basisHc*varsAmbient);
    deltag := Hc.deltas#1 Hc.inclusion g;
    Hc12 := target Hc.deltas#1;
    Hc1 := source Hc12.cache#iota1;
    Hc2 := source Hc12.cache#iota2;
    phi1 := ncMap(Hc1,Hc.ring,gens Hc1);
    phi2 := ncMap(Hc2,Hc.ring,gens Hc2);
    gtensg := (Hc12.cache#iota1 phi1 g)*(Hc12.cache#iota2
       phi2 g);
    basisHc12 := totalBasis Hc12;
    gpLikeCoeffs := sparseCoeffs(deltag - gtensg, Monomials
        => flatten entries basisHc12);
    J := (ideal gpLikeCoeffs) + ideal ((leadCoefficient Hc.
       counit g) - 1_kk);
    primDec := primaryDecomposition radical J;
    apply(primDec, p -> first flatten entries ((basisH)*
       transpose sub(matrix {gens (kk/p)}, coefficientRing
       H.ring)))
)

applyAntihomomorphism = method()
applyAntihomomorphism (NCRingMap, NCRingElement) := (S, f)
    -> (
    --- apply the function S to the element f as an
       antihomomorphism.
    if ring f =!= source S then error "Expected an element
       of the domain of S.";
    A := ring f;
    matS := flatten entries matrix S;
    genHash := hashTable apply(numgens A, i -> ((A.
       generatorSymbols)#i, matS#i));
    sum apply(pairs (f.terms), (m,c) -> c*(product apply(
       reverse m.monList, x -> genHash#x)))
)

findAntipode = method()
```

```
findAntipode NCBialgebra := H -> (
   B := H.ring;
   basisH := totalBasis H;
   n := #(flatten entries basisH);
   Hbig := extendCoeffRing2(H,n*(numgens B),getSymbol "xxx
      ");
   psi := map(coefficientRing Hbig.ring, coefficientRing H
      .ring);
   phi := f -> fastBaseChange(Hbig.ring, f, psi);
   basisHbig := ncMatrix applyTable(entries basisH, phi);
   coeffRing := coefficientRing Hbig.ring;
   genericAntipodes := apply(numgens B, i -> first flatten
       entries (basisHbig*(transpose (vars coeffRing)_(
      toList((i*n)..((i+1)*n-1))))));
   S := ncMap(Hbig.ring, Hbig.ring, genericAntipodes);
   convolveSId := apply(numgens B, i -> sum apply(Hbig.
      deltaImg#i, p -> applyAntihomomorphism(S,p#0)*(p#1))
       - Hbig.counit (Hbig.ring)_i);
   convCoeffs := sparseCoeffs(convolveSId, Monomials =>
      flatten entries basisHbig);
   antipodeCoeffs := pack((gens (coeffRing/(ideal
      convCoeffs)) / leadCoefficient),n);
   antipodeOutput := apply(numgens B, i -> first flatten
      entries (basisH*(transpose matrix {antipodeCoeffs#i
      })));
   ncMap(H.ring, H.ring, antipodeOutput)
)

----- Bialgebra action code

bialgebraAction = method(Options => {DegreeLimit => 10})
bialgebraAction (NCBialgebra,List) := opts -> (B, mapList)
    -> (
   --- B bialgebra acting on R
   --- mapList is a list of either NCRingMaps or matrices
      over the coefficient ring of R
   --- that define the action of the generators of B on
      the generators of R
   if length(mapList) =!= numgens B.ring then error "
      Action definition incomplete";
   if any(mapList, f -> class f =!= NCRingMap) then error
      "Expected a list of NCRingMaps.";
```

```
    aRing := source first mapList;
    basisB := flatten apply(opts#DegreeLimit, i -> flatten
       entries basis(i,B.ring));
    genActionHash := new HashTable from apply(numgens B.
       ring, i -> ((B.ring.generatorSymbols)#i, mapList#i))
       ;
    myAtlas := new HashTable from {(first basisB,ncMap(
       aRing,aRing,gens aRing))} |
                                    apply(drop(basisB,1), m
                                       -> (m, product apply
                                       ((first first pairs (
                                       m.terms)).monList, a
                                       -> genActionHash#a)))
                                       ;
    actCache := new HashTable from apply(basisB, m-> (m,new
         MutableHashTable from {}));
    A := new BialgebraAction from {(symbol bialgebra) => B,
                                    (symbol ring) => aRing,
                                    (symbol generatorAction)
                                        => genActionHash,
                                    (symbol atlas) =>
                                       myAtlas,
                                    (symbol actionCache) =>
                                       actCache,
                                    (symbol cache) => new
                                       CacheTable from {}};
    A
)

ambient BialgebraAction := action -> (
    --- lifts the action of action.bialgebra from action.
       ring to its ambient tensor algebra.
    K := action.ring;
    L := ambient K;
    phi := ncMap(L, K, gens L);
    bialgebraAction(action.bialgebra, L, apply(action.
       bialgebra.ring.generatorSymbols, var -> phi @@ (
       ambient action.generatorAction#var)))
)

isWellDefined BialgebraAction := action -> (
    --- is the ideal sent to itself under the Hopf action?
```

```
    ambAction1 := ambient action;
    defIdeal := ideal action.ring;
    gbDefIdeal := ncGroebnerBasis defIdeal;
    isIdealPreserved := all(gens (action.bialgebra.ring), x
        -> all(gens defIdeal, f -> (actOn(x,f,ambAction1) %
        gbDefIdeal) == 0));
    --- is the action on the degree 1 part of action.ring a
        module over action.bialgebra?
    defIdealH := ideal (action.bialgebra.ring);
    genMatrices := apply(gens action.bialgebra.ring, x -> (
       action.atlas#x)_1);
    isHModule := all(gens defIdealH, f -> matrixSub(f,
       genMatrices) == 0);
    isIdealPreserved and isHModule
)

matrixSub = method()
matrixSub (NCRingElement, List) := (f, mats) -> (
    if numgens ring f != #mats then error "Expected a
       matrix for each generator of the ring of the first
       input.";
    A := ring f;
    genHash := hashTable apply(numgens A, i -> ((A.
       generatorSymbols)#i, mats#i));
    sum apply(pairs (f.terms), (m,c) -> c*(product apply(m.
       monList, x -> genHash#x)))
)

actOnDegreeOne = method()
actOnDegreeOne (NCRingElement, NCRingElement,
   BialgebraAction) := (h,x,action) -> (
    if degree x != 1 then error "Wrong function called for
       action on elements of degree greater than one.";
    hTerms := terms h;
    hCoeffs := hTerms / leadCoefficient;
    hMons := hTerms / leadMonomial;
    sum apply(#hTerms, i -> (hCoeffs#i)*((action#atlas)#(
       hMons#i) x))
)

factorMonomial = method()
factorMonomial NCRingElement := m -> (
```

```
    monList := (first first pairs (m.terms)).monList;
    halfOf := floor(#monList/2);
    (putInRing(take(monList,halfOf),ring m, 1), putInRing(
        drop(monList,halfOf),ring m, 1))
)


actOnQuadMon = method() -- Is this unnessisary at this
    point?
actOnQuadMon (NCRingElement, NCRingElement,
    BialgebraAction) := (h,m,action) -> (
    if degree m != 2 then error "Wrong function called for
        action on elements of degree not two.";
    if #(terms m) != 1 then error "Wrong function called
        for action on non-monomial element.";
    H := action.bialgebra;
    mCoeff := leadCoefficient(m);
    mFactor := factorMonomial(m);
    dh := H.deltas#1 H.inclusion h;
    dhTerms := terms dh;
    dhCoeffs := dhTerms / leadCoefficient;
    dhMons := dhTerms / leadMonomial;
    dhMonsL := apply(dhMons, i-> H.leftTens(i));
    dhMonsR := apply(dhMons, i-> H.rightTens(i));
    sum(apply(#dhTerms, i -> (dhCoeffs#i)*actOnDegreeOne(
        dhMonsL#i,mFactor#0,action)*actOnDegreeOne(dhMonsR#i
        ,mFactor#1,action)))*mCoeff
)

actOn = method()
actOn (NCRingElement, NCRingElement, BialgebraAction) := (
    h,m,action) -> (
    --- employ actOnMonomial to handle sums of elements
        from hopf algebra and ring, as well as coefficients
        from ring
    mTerms := terms m;
    mCoeffs := mTerms / leadCoefficient;
    mMons := mTerms / leadMonomial;
    hTerms := terms h;
    hCoeffs := hTerms / leadCoefficient;
    hMons := hTerms / leadMonomial;
    sum flatten apply(#hCoeffs, i -> apply(#mCoeffs, j ->
```

```
      hCoeffs#i*mCoeffs#j*actOnMonomial(hMons#i,mMons#j,
      action)))
)

actOnMonomial = method()
actOnMonomial (NCRingElement, NCRingElement,
   BialgebraAction) := (h,m,action) -> (
   --- assumes f is a monomial in the basis given by
      Macaulay2 induced by the
   --- Groebner bases computed previously.
   --- this function will be recursive based on the "
      tensor length" of f.
   --- we want to restrict h to a basis element of H
   H := action.bialgebra;
   if #(terms m) != 1 then error "Wrong function called
      for action on non-monomial element.";
   if (degree m) === 1 then return actOnDegreeOne(h,m,
      action);
   if (degree m) === 0 then return (leadCoefficient H.
      counit(h))*m;
   if ((action.actionCache)#h)#?m then return ((action.
      actionCache)#h)#m;
   mFactor := factorMonomial(m);
   dh := H.deltas#1 H.inclusion h;
   dhTerms := terms dh;
   dhCoeffs := dhTerms / leadCoefficient;
   dhMons := dhTerms / leadMonomial;
   dhMonsL := apply(dhMons, i-> H.leftTens(i));
   dhMonsR := apply(dhMons, i-> H.rightTens(i));
   val:=sum(apply(#dhTerms, i -> (dhCoeffs#i)*
      actOnMonomial(dhMonsL#i,mFactor#0,action)*
      actOnMonomial(dhMonsR#i,mFactor#1,action)));
   ((action.actionCache)#h)#m = val;
   val
)

actionInDegree = method()
actionInDegree (NCRingElement, BialgebraAction, ZZ) := (h,
   action,n) -> (
   K := action.ring;
   basisK := flatten entries basis(n,K);
   actOnBasis := apply(basisK, m -> actOn(h,m,action));
```

```
    sparseCoeffs(actOnBasis,Monomials=>basisK)
)


invariantInDegree = method(Options => {Strategy => 0})
invariantInDegree (BialgebraAction, ZZ) := opts -> (action
    , n) -> (
    --- This method computes invariants using the
        intersection
    --- of kernels trick
    if opts#Strategy != 0 then return invariantInDegreeAlt(
        action,n);
    B := action.bialgebra;
    H := B.ring;
    K := action.ring;
    basisK := basis(n,K);
    kk := coefficientRing K;
    matList := apply(gens H, x -> actionInDegree(x,action,n
        ) - leadCoefficient(B.counit x)*id_(kk^(#(basisK.
        source))));
    pruneKer := prune ker matrix apply(matList, m -> {m});
    invCoeffs := gens image pruneKer.cache.pruningMap;
    flatten entries (basisK*invCoeffs)
)


invariantInDegreeAlt = method()
invariantInDegreeAlt (BialgebraAction, ZZ) := (action, n)
    -> (
    --- This method computes invariants using (left)
        integrals
    H := action.bialgebra;
    A := action.ring;
    int :=findLeftIntegral(H);
    basisA := basis(n,A);
    mat := sparseCoeffs(apply(flatten entries basisA, a ->
        actOn(int,a,action)),Monomials=>flatten entries
        basisA);
    columnBasis  := mingens image mat;
    flatten entries (basisA*columnBasis)
)
```

```
getNextDegInvar = method()
getNextDegInvar (BialgebraAction,List,ZZ) := (action,
   gensSoFar, n) -> (
   H := action.bialgebra;
   R := action.ring;
   B := H.ring;
   c := getSymbol "cc";
   kk := coefficientRing B;
   KK := kk{c_1..c_#gensSoFar};
   phi := ncMap(R,KK,gensSoFar);
   setWeights(KK,apply(gensSoFar, n-> degree n));
   potentialGens := invariantInDegree(action, n);
   if potentialGens == {} then return {};
   potentialGensCoeffs := sparseCoeffs(potentialGens,
      Monomials=> flatten entries basis(n,R));
   newGensQuot := prune ((image potentialGensCoeffs)/(
      image phi_n));
   if newGensQuot == 0 then return {};
   newGens := (flatten entries (basis(n,R)*(gens image
      newGensQuot.cache.pruningMap))) / makeMonic;
   newGens
)


invariantAlgebraGens = method((Options => {DegreeLimit =>
   10}))
invariantAlgebraGens BialgebraAction := opts -> action ->
   (
   if action.cache#?invarGens then return action.cache.
      invarGens;
   i := 1;
   gensSoFar := {};
   while #gensSoFar < 1 and i <= opts#DegreeLimit do (
      gensSoFar = invariantInDegree(action,i); i = i+1);
   if i == opts#DegreeLimit then return gensSoFar;
   while i <= opts#DegreeLimit do (
       gensSoFar = gensSoFar|(getNextDegInvar(action,
          gensSoFar,i));
       i = i+1;
       );
   action.cache.invarGens = gensSoFar;
   gensSoFar
```

```
)

invariantAlgebra = method(Options => {GenDegreeLimit =>
   10, RelDegreeLimit => infinity, InstallGB => false})
invariantAlgebra (BialgebraAction,Symbol) := opts -> (
   action,c) -> (
   if action.cache#?invarAlg then return action.cache.
      invarAlg;
   relDegLimit := opts#RelDegreeLimit;
   if relDegLimit == infinity then relDegLimit = 2*opts#
      GenDegreeLimit;
   invAlgGens := invariantAlgebraGens(action, DegreeLimit
      => opts#GenDegreeLimit);
   kk := coefficientRing action.ring;
   if invAlgGens == {} then return kk;
   KK := kk{c_1..c_#invAlgGens};
   setWeights(KK, invAlgGens/degree);
   phi := ncMap(action.ring, KK, invAlgGens);
   kerGens := gddKernel(relDegLimit, phi);
   I := ncIdeal(kerGens);
   Igb := ncGroebnerBasis(I, InstallGB => opts#InstallGB);
   R := if kerGens == {} then KK else KK/I;
   R.cache#(symbol presentation) = ncMap(action.ring, R,
      invAlgGens);
   R.cache#(symbol GenDegreeLimit) = opts#GenDegreeLimit;
   R.cache#(symbol RelDegreeLimit) = relDegLimit;
   action.cache#(symbol invarAlg) = R;
   R
)

--- Dual structure
new DualBialgebra from List := (DualBialgebra, inits) ->
   new DualBialgebra of DualElement from new HashTable
   from inits

dualize = method(Options => {DegreeLimit => 10})
dualize NCBialgebra := opts -> B -> (
   basisA := dualBasis(B,opts);
   A := new DualBialgebra from {(symbol bialgebra) => B,
                                (symbol ring) => null,
                                (symbol basis) => basisA,
                                (symbol cache) => new
```

```
                                        CacheTable from {}};

    A + A := (phi,psi) -> dualElement(A,(phi.matrix + psi.
       matrix));
    A * A := (phi,psi) -> dualProd(phi,psi);
    A ** A := (phi,psi) -> (
        R := ring (phi.matrix);
        zeroDegree := degree (1_R);
        temp := (phi.matrix)**(psi.matrix);
        map(R^{zeroDegree},R^(toList((#basisA)^2:zeroDegree
           )),entries temp)
    );
    A == A := (phi,psi) -> (phi.ring == psi.ring) and (phi.
       matrix == psi.matrix);
    A ^ ZZ := (phi,n) -> product apply(n, i-> phi);
    k := coefficientRing ((A.bialgebra).ring);
    k*A := (c,phi) -> dualElement(A,(c*(phi.matrix)));
    A*k := (phi,c) -> dualElement(A,((phi.matrix)*c));
    ZZ*A := (c,phi) -> promote(c,k)*phi;
    A*ZZ := (phi,c) -> phi*(promote(c,k));
    QQ*A := (c,phi) -> promote(c,k)*phi;
    A*QQ := (phi,c) -> phi*(promote(c,k));
    A
)

DualBialgebra == DualBialgebra := (A,B) -> (A.bialgebra
   === B.bialgebra) and (#(A.basis) == #(B.basis));

dualElement = method(Options => {DegreeLimit => 10})
dualElement (DualBialgebra, Matrix) := opts -> (dualB,mat)
    -> (
    if (numgens target mat) != 1 then error "Expected row
       vector";
    B := dualB.bialgebra;
    basisB := flatten entries totalBasis(B,opts);
    if (numgens source mat) != #basisB then error "Row
       vector source does not match bialgebra";
    new dualB from hashTable {(symbol matrix) => mat,
                              (symbol ring) => dualB}
)

net DualElement := de -> net de.matrix
```

```
ncMap DualElement := opts -> phi -> (

   H := ((phi.ring).bialgebra).ring;
   kk := coefficientRing H;
   basisH := flatten entries totalBasis H;
   gensH := gens H;
   gensAddress := apply(gensH, h -> position(basisH, j ->
      h == j));
   gensImage := apply(flatten entries (sub(phi.
      matrix_gensAddress,kk)), c -> c_H);
   ncMap(H, H, gensImage / (i -> if i === 0 then i_H else
      i))
)

dualBasis = method(Options => {DegreeLimit => 10})
dualBasis NCRing := opts -> H -> (
   --- This method takes an NCRing, computes a basis and
      outputs a basis of the dual,
   --- considered as row vectors.
   basisH := flatten entries totalBasis(H,opts);
   kk := coefficientRing H;
   apply(#basisH, i -> (id_(kk^(#basisH)))^{i})
)

dualBasis NCBialgebra := opts -> B -> dualBasis(B.ring,
   opts)

dualCounit = method()
dualCounit DualElement := f -> (f.matrix)_(0,0)

dualUnit = method(Options => {DegreeLimit => 10})
dualUnit DualBialgebra := opts -> B -> (
   --- This method is the counit of a bialgebra,
      considered as a function
   basisB := flatten entries totalBasis(B,opts);
   eps := B.counit;
   imgList := apply(basisB, x -> leadCoefficient eps x);
   matrix{imgList}
)

dualProd = method()
```

```
dualProd (DualElement,DualElement) := (phi,psi) -> (
   a := phi.matrix;
   b := psi.matrix;
   if (phi.ring != psi.ring) then error "Expected elements
       from the same dual space.";
   comultMat := comultMatrix (phi.ring).bialgebra;
   prodMat := (a**b)*comultMat;
   dualElement(phi.ring,prodMat)
)

--- oneDimensionalRepresentations
findGrouplikes DualBialgebra := opts -> H2 -> (
   basisH := totalBasis(H2.bialgebra,opts);
   n := #(flatten entries basisH);
   cc := getSymbol "cc";
   bigH := extendCoeffRing(H2.bialgebra,n,cc);
   R := coefficientRing (bigH.ring);
   bigH2 := dualize bigH;
   phi := ncMap(bigH.ring,H2.bialgebra.ring,gens bigH.ring
      );
   basisBigH := phi basisH;
   gen1 := dualElement(bigH2, vars R);
   temp := dualCoprod gen1 - (gen1.matrix ** gen1.matrix);
   grouplikeList := primaryDecomposition radical (ideal
      sub(temp,R) + ideal sub(dualCounit gen1 - 1_R, R));
   highDegreePolys := select(flatten (grouplikeList / gens
       / entries / flatten), f -> first degree f > 1);
   if any(highDegreePolys, f -> first degree f > 1) then
      << "Warning: The following polynomials did not
         factor completely: " << highDegreePolys << "." <<
          endl;
   grouplikeList = select(grouplikeList, f -> #(flatten
      entries basis (R/f)) == 1);
   oneDimRepRowVecs := apply(grouplikeList, p ->
      dualElement(H2,sub(matrix entries sub(gen1.matrix, R
      /p),R)));
   oneDimRepRowVecs
)

weightSpaceInDegree = method()
weightSpaceInDegree (BialgebraAction, ZZ, DualElement) :=
   (action, n, phi) -> (
```

```
    B := action.bialgebra;
    phiMap := ncMap phi;
    if not isWellDefined phiMap then error "Provided linear
        functional is not a valid representation.";
    H := B.ring;
    K := action.ring;
    basisK := basis(n,K);
    kk := coefficientRing K;
    matList := apply(gens H, x -> actionInDegree(x,action,n
       ) - leadCoefficient(phiMap x)*id_(kk^(#(basisK.
       source))));
    pruneKer := prune ker matrix apply(matList, m -> {m});
    invCoeffs := gens image pruneKer.cache.pruningMap;
    wtGens := (flatten entries (basisK*invCoeffs)) /
       makeMonic;
    wtGens
)

getNextDegWeights = method()
getNextDegWeights (BialgebraAction,List,ZZ, DualElement)
    := (action, gensSoFar, n, phi) -> (
    H := action.bialgebra;
    R := action.ring;
    B := H.ring;
    d := getSymbol "dd";
    invAlg := invariantAlgebra(action,d);
    basesOfDeg := apply(gensSoFar/degree, d -> (flatten
       entries basis(n-d, invAlg))/(invAlg.cache.
       presentation));
    spanningInDeg := flatten apply(basesOfDeg,gensSoFar, (l
       ,f) -> apply(l, g -> f*g));
    potentialGens := weightSpaceInDegree(action, n, phi);
    if potentialGens == {} then return {};
    potentialGensCoeffs := sparseCoeffs(potentialGens,
       Monomials=> flatten entries basis(n,R));
    if spanningInDeg == {} then return potentialGens;
    sparseSpanningInDeg := sparseCoeffs(spanningInDeg,
       Monomials=> flatten entries basis(n,R));
    newGensQuot := prune ((image potentialGensCoeffs)/(
       image sparseSpanningInDeg));
    if newGensQuot == 0 then return {};
    newGens := (flatten entries (basis(n,R)*(gens image
```

```
      newGensQuot.cache.pruningMap))) / makeMonic;
   newGens
)

weightSpaceGens = method((Options => {DegreeLimit => 10}))
weightSpaceGens (BialgebraAction,DualElement) := opts -> (
   action,phi) -> (
   i := 0;
   gensSoFar := {};
   while #gensSoFar < 1 and i <= opts#DegreeLimit do (
      gensSoFar = weightSpaceInDegree(action,i,phi); i = i
      +1);
   if i == opts#DegreeLimit then return gensSoFar;
   while i <= opts#DegreeLimit do (
       gensSoFar = gensSoFar|(getNextDegWeights(action,
          gensSoFar,i,phi));
       i = i+1;
       );
   gensSoFar
)

end
```

Bibliography

[1] Jörgen Backelin. bergman, a system for computations in commutative and purely non-commutative graded algebra. Available at `http://servus.math.su.se/bergman/`.

[2] Sorin Dascalescu, Constantin Nastasescu, and Serban Raianu. *Hopf algebra: An introduction*. CRC Press, 2000.

[3] Luigi Ferraro, Ellen Kirkman, W Frank Moore, and Robert Won. Three infinite families of reflection hopf algebras. *arXiv preprint arXiv:1810.12935*, 2018.

[4] Daniel R. Grayson and Michael E. Stillman. Macaulay2, a software system for research in algebraic geometry. Available at `http://www.math.uiuc.edu/Macaulay2/`.

[5] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[6] David E Radford. *Hopf algebras*, volume 49. World Scientific, 2012.

[7] Earl J Taft. The order of the antipode of finite-dimensional hopf algebra. *Proceedings of the National Academy of Sciences*, 68(11):2631–2633, 1971.

[8] William C Waterhouse. Antipodes and group-likes in finite hopf algebras. *Journal of Algebra*, 37(2):290–295, 1975.

.2 Curriculum Vita

Colin Martin

181 Camelot Crest
Sylva, North Carolina
United States of America
+1 (828) 508 0340
martce17@wfu.edu
colin-martin-88921a181
@ColinMa45563703

Aut viam inveniam aut faciam.

Education

2011-2015 **Bachelor of Arts in Mathematics**, *Warren Wilson College*, Swannanoa, North Carolina, *3.58*.
Minor: Traditional Music

2017–2019 **Master of Arts in Mathematics and Statistics**, *Wake Forest University*, Winston-Salem, NC, *3.42*.

Master thesis

title *A Package for Calculating and Manipulating Hopf Algebras in Macaulay2*

supervisors Dr. W. Frank Moore

description The study of Hopf algebras is a relatively new field of modern algebra with many applications in other areas. Despite this, there is a marked lack of software support for mathematicians working with Hopf algebras. Here, we introduce the software HopfAlgebras.m2, a package for the Macaulay2 algebra system designed to perform calculations on Hopf algebras, with a focus on invariant theory.

Experience

2017-2019 **Teaching Assistant**, *Wake Forest University Department of Mathematics and Statistics*, Winston-Salem, NC.
Ran weekly study sessions in linear algebra and calculus, graded papers and performed quality one-on-one tutoring.

2015-2018 **Lead Youth Mentor**, *YMCA of WNC*, Asheville, NC.
Provided academic support to underprivileged students in Buncombe county school systems.

2012-2015 **Math Crew**, *Warren Wilson College*, Swannanoa, NC.
Ran nightly study sessions on calculus, linear algebra and statistics, graded, maintained math departmental website and performed other clerical duties.