DIAS-STP-10-02

IMPLEMENTING GRÖBNER BASES FOR OPERADS

VLADIMIR DOTSENKO AND MIKAEL VEJDEMO JOHANSSON

Abstract. We present an implementation of the algorithm for computing Gröbner bases for operads due to the first author and A. Khoroshkin. We discuss the actual algorithms, the choices made for the implementation platform and the data representation, and strengths and weaknesses of our approach.

1. Introduction

1.1. **Summary of results.** In an upcoming paper [4], the first author and Anton Khoroshkin define the concept of a Gröbner basis for finitely presented operads. In that paper, they prove the diamond lemma, and demonstrate that for an operad, having a quadratic Gröbner basis is equivalent to the existence of a Poincaré–Birkhoff–Witt basis. As demonstrated by Eric Hoffbeck [5], an operad with a PBW basis is Koszul. Hence, an implementation of the Gröbner bases algorithm yields, in addition to a framework for exploration of operads by means of explicit calculation, a computer-aided tool for proving Koszulness.

In this paper, we present an implementation of the Gröbner basis algorithm in the Haskell programming language [7]. Being designed with categorical terms, Haskell provides a powerful framework for algorithms like that. What we end up with is a computer sofware package which allows to compute the Gröbner basis for a finitely presented operad, as well as bases and dimensions for components of such an operad.

One of the main goals of this paper is to help mathematicians who want to get familiar with this software package and use it for their needs, including changing some algorithms or adding more functionality.[1] Consequently, this is more of an invitation to experiment with this software than a report on what it is possible to compute. Let us comment briefly on the state of the art regarding computations. While working on the package, we have implemented several well known operads to test the performance. In the case when an operad is PBW, our package captures that right away. This already is a very important achievement: having implemented many different admissible orderings, one can check very fast whether or not an operad is PBW for at least one of them, thus proving the Koszulness in many cases. Note that the PBW property depends a lot not only on the choice of an admissible ordering, but also on the choice of ordering of generators of our operad; for example, for the operad of pre-Lie algebras, depending on the ordering, a Gröbner basis can vary from quadratic to seemingly infinite. On the other hand, for operads that do not have a quadratic Gröbner basis, we encountered subtle performance issues in many cases. For operads having a relatively small finite Gröbner basis, like the fake commutative operad AntiCom [4], the computation easily yields the correct

---

[1]The first author is a living example proving that it is possible; having been introduced to Haskell by the second author in the process of working on this package, he now has enough confidence to not only use the package, but to add new functions as well.

result, while for many other cases, like the pre-Lie operad for a "wrong" ordering, computations with arity 6 and further take enormously long.

The actual implementation is distributed through the HackageDB repository for Haskell software projects at

<center>http://hackage.haskell.org/package/Operads,</center>

software distributed through this repository are available through the automated installation tool `cabal-install`.

The current documentation files are kept online at

<center>http://math.stanford.edu/~mik/operads/.</center>

1.2. **Outline of the paper.** The paper is organized as follows. In Section 2, we recall relevant background information related to operads and Gröbner bases, on one hand, and to types and functions in Haskell, on the other hand. In Section 3, we discuss the way we chose to represent our data in Haskell. In Section 4, we present algorithms used in our implementation. Finally, in the appendix, we list Haskell constructions used throughout the paper.

1.3. **Acknowledgements.** We wish to express our deep gratitude to Eric Hoffbeck and Henrik Strohmayer for both significant assistance in the construction of the software code, and analysis of the techniques we are using. Some of the hairier points of Haskell evaluation has been rendered clear by the helpful assistance of the many members of the `#haskell` IRC channel on the Freenode IRC network.

The first author was supported by an IRCSET research fellowship. The second author was supported by the Office of Naval Research, through grant N00014-08-1-0931.

We are grateful to Jean–Louis Loday and Bruno Vallette who organized the "Operads 2009" meeting in CIRM Luminy, where the work on this project was started. The second author wishes to thank Dublin Institute for Advanced Studies which hosted him as a visitor during the last stage of working on this paper.

2. Overview

For exhaustive information on symmetric operads, we refer the reader to monographs [9] and [10]. Here, we mainly concentrate on shuffle operads and their relationship with symmetric operads, and definitions in the symmetric case are chosen in the way that best suits this approach.

2.1. **Operads.** We denote by Ord the category of nonempty finite ordered sets (with order-preserving bijections as morphisms), and by Fin — the category of nonempty finite sets (with bijections as morphisms). Also, we denote by Vect the category of vector spaces (with linear operators as morphisms; unlike the first two cases, we do not require a map to be invertible).

Definition 1.   (1) A *(nonsymmetric) collection* is a contravariant functor from the category Ord to the category Vect.
   (2) A *symmetric collection* (or an $\mathbb{S}$-*module*) is a contravariant functor from the category Fin to the category Vect.

For either type of collections, we can consider the category whose objects are collections of this type (and morphisms are morphisms of the corresponding functors). The natural forgetful functor $^f\colon$ Ord $\to$ Fin, $I \mapsto I^f$ leads to a forgetful functor ^{f}

from the category of symmetric collections to the category of nonsymmetric ones, $\mathscr{P}^f(I) := \mathscr{P}(I^f)$. For simplicity, we let $\mathscr{P}(k) := \mathscr{P}([k])$.

We use the convention $[k] = \{1, 2, \ldots, k\}$ in this paper.

Definition 2. • Let $\mathscr{P}$ and $\mathscr{Q}$ be two nonsymmetric collections. Define their *shuffle composition* $\mathscr{P} \circ_{sh} \mathscr{Q}$ by the formula

$$(\mathscr{P} \circ_{sh} \mathscr{Q})(I) := \bigoplus_k \mathscr{P}(k) \otimes \left(\bigoplus_{f \colon I \twoheadrightarrow [k]} \mathscr{Q}(f^{-1}(1)) \otimes \ldots \otimes \mathscr{Q}(f^{-1}(k)) \right),$$

where the sum is taken over all shuffling surjections f, that is surjections for which $\min f^{-1}(i) < \min f^{-1}(j)$ whenever $i < j$.
• Let $\mathscr{P}$ and $\mathscr{Q}$ be two symmetric collections. Define their *(symmetric) composition* $\mathscr{P} \circ \mathscr{Q}$ by the formula

$$(\mathscr{P} \circ \mathscr{Q})(I) := \bigoplus_k \mathscr{P}(k) \otimes_{S_k} \left(\bigoplus_{f \colon I \twoheadrightarrow [k]} \mathscr{Q}(f^{-1}(1)) \otimes \ldots \otimes \mathscr{Q}(f^{-1}(k)) \right),$$

where the sum is taken over all surjections f.

Each of these compositions gives a structure of a monoidal category on the category of the corresponding collections. The same definitions can be given if we replace Vect by another symmetric monoidal category. For our purposes, an important replacement for Vect will be the category of finite sets (with arbitrary mappings as morphisms).

Definition 3. (1) A *shuffle operad* is a monoid in the category of nonsymmetric collections with the monoidal structure given by the shuffle composition.
(2) A *symmetric operad* is a monoid in the category of symmetric collections with the monoidal structure given by the (symmetric) composition.

Definition 4. A *shuffle permutation* of the type $(k_1, \ldots, k_n)$ is a permutation in the symmetric group $S_{k_1 + \ldots + k_n}$ which preserves the order of the first k_1 elements, the second k_2 elements,..., the last k_n elements, and satisfies

$$\sigma(1) < \sigma(k_1 + 1) < \sigma(k_1 + k_2 + 1) < \ldots < \sigma(k_1 + \ldots + k_{n-1} + 1).$$

Proposition 1. *The number of shuffle permutations of the type $(k_1, \ldots, k_n)$ is equal to*

$$\frac{k_1 k_2 \cdot \ldots \cdot k_n}{(k_1 + k_2 + \ldots + k_n)(k_2 + \ldots + k_n) \cdot \ldots \cdot k_n} \binom{k_1 + k_2 + \ldots + k_n}{k_1, k_2, \ldots, k_n}.$$

When implementing shuffle permutations, one can use the following simple idea: In a shuffle permutation, the number whose image is $k_1 + \ldots + k_n$ should clearly be the maximal one in its block. Moreover, if this block is of size 1, it should be the last one to comply with the ordering condition on the first elements of blocks. This implies an obvious recursive algorithm to generate a list of shuffle permutations with given sizes of blocks: put the maximal image in the end of each allowed block, and for each such choice list all shuffle permutations where the corresponding block contains one element less than prescribed.

Definition 5.   (1) Let $\mathscr{O}$ be a shuffle operad, $\beta \in \mathscr{O}(n)$, $\alpha_1 \in \mathscr{O}(k_1)$, $\ldots$, $\alpha_n \in \mathscr{O}(k_n)$. Assume that $\sigma \in S_{k_1 + \ldots + k_n}$ is a shuffle permutation of the type $(k_1, \ldots, k_n)$. Denote by B_s, $s = 1, \ldots, n$, the sth block of $[k_1 + \ldots + k_n]$ (on which σ is monotonous). Then we define

$$\beta(\alpha_1, \ldots, \alpha_n)_\sigma = \circ(\beta \otimes \sigma(\alpha_1) \otimes \ldots \otimes \sigma(\alpha_n)) \in \mathscr{O}(k_1 + \ldots + k_n),$$

where $\sigma(\alpha_s)$ is the image of α_s under the isomorphism between $\mathscr{O}(k_s)$ and $\mathscr{O}(\sigma(B_s))$, and $\circ \colon \mathscr{O} \circ_{sh} \mathscr{O} \to \mathscr{O}$ is the monoid product map.

     (2) Let $\mathscr{O}$ be a symmetric operad, $\beta \in \mathscr{O}(n)$, $\alpha_1 \in \mathscr{O}(k_1)$, $\ldots$, $\alpha_n \in \mathscr{O}(k_n)$. Let $\sigma \in S_{k_1 + \ldots + k_n}$ be an arbitrary permutation. Denote by B_s, $s = 1, \ldots, n$, the sth block of $[k_1 + \ldots + k_n]$ (of size k_s). Then we define

$$\beta(\alpha_1, \ldots, \alpha_n)_\sigma = \circ(\beta \otimes \sigma(\alpha_1) \otimes \ldots \otimes \sigma(\alpha_n)) \in \mathscr{O}(k_1 + \ldots + k_n),$$

where $\sigma(\alpha_s)$ is the image of α_s under the isomorphism between $\mathscr{O}(k_s)$ and $\mathscr{O}(\sigma(B_s))$, and $\circ \colon \mathscr{O} \circ \mathscr{O} \to \mathscr{O}$ is the monoid product map.

It turns out that the forgetful functor ^{f} is a monoidal functor between the category of symmetric operads and the category of shuffle operads. Consequently, to study various questions of linear algebra for symmetric operads, it is sufficient to forget the full symmetric structure because the shuffle structure already captures everything. See [4] for more details.

2.2. **Trees.** Assume that we are given a collection of disjoint finite sets $M(n), n \geq 1$. A *(rooted) tree* is a non-empty directed graph T of topological genus 0 for which each vertex has at least one incoming edge and exactly one outgoing edge. We allow for some edges of a tree to be bounded by a vertex at one end only. Such edges are called *external*. Each tree has one outgoing external edge, the *output* or the *root*, and several ingoing external edges, called *leaves*. All vertices of the tree should be decorated by elements of sets from the collection M; for a vertex with n inputs, the element used for the decoration should belong to $M(n)$. Usually, we consider such trees with some additional structure: for a tree with n leaves, we require the leaves to be labelled by $[n]$. For each vertex v of a tree, the edges going in and out of v will be referred to as inputs and outputs at v. A tree with a single vertex is called a *corolla*. There is also a tree with a single input and no vertices called the *degenerate* tree. Trees are originally considered as abstract graphs but to work with them we need some particular representatives that we now are going to describe.

For a tree with labelled leaves, its canonical planar representative is defined as follows. In general, an embedding of a (rooted) tree in the plane is determined by an ordering of inputs for each vertex (the leftmost one being the smallest, the rightmost — the largest). To compare two inputs of a vertex v, we find the minimal leaves that one can reach from v via the corresponding input. The input for which the minimal leaf is smaller is considered to be less than the other one. Planar representatives of decorated trees will be referred to as *tree monomials*. The collection of all tree monomials whose vertices are labelled by the collection M is denoted by $\mathscr{F}_M$. For some constructions, we shall need *nonsymmetric tree monomials*, that is planar trees with decorations of internal vertices but no labelling of leaves (or, when it is needed for compositions, the trivial labelling of leaves in the increasing order from the left to the right suggested by the planar embedding).

Compositions of trees are defined as follows. Given a tree β with n leaves, trees α_1, ..., α_n with k_1, ..., k_n leaves respectively, we define the composition $\beta(\alpha_1, \alpha_2, \ldots, \alpha_n)$ as the tree obtained by grafting the tree α_1 to the leaf of β labelled by 1, the tree α_2 with leaf labels shifted by k_1 to the leaf of β labelled by 2, ..., the tree α_n with leaf labels shifted by $k_1 + \ldots + k_{n-1}$ to the leaf of β labelled by n. If, in addition, σ is a shuffle permutation of $[k_1 + \ldots + k_n]$ of the type $(k_1, \ldots, k_n)$, we can define the shuffle composition $\beta(\alpha_1, \alpha_2, \ldots, \alpha_n)_\sigma$; to compute it, we first compute the nonsymmetric composition of our trees and then apply σ to the leaf labels of the resulting tree.

Remark 1. Let us emphasize two practicalities. First of all, the data type for trees chosen for an implementation should be easily adjustable for performing compositions. Second, it is important here that we apply the permutation, not its inverse; usually, action of permutations on functions and mappings uses inverses, however, since operads are contravariant functors, we do not need that. One should be careful, and remember it when implementing the action of permutations.

Proposition 2 ([4]).   (1) *The collection of trees $\mathscr{F}_M$ is closed under shuffle compositions.*
    (2) *Every tree in $\mathscr{F}_M$ can be obtained from corollas by iterated shuffle compositions.*

The collection $\mathscr{F}_M$ is the free shuffle operad generated by the nonsymmetric collection M in the category of finite sets; its linear span is the free shuffle operad generated by the linear span of M in the category of vector spaces.

Here and below by a divisor of a nonsymmetric tree monomial T we mean a nonsymmetric tree monomial T' whose underlying tree is embedded into the underlying tree of T in such a way that the labellings of vertices are the same.

For a tree monomial α with the underlying nonsymmetric monomial T and a divisor T' of T, let us define a tree monomial α' that corresponds to T'. Its vertices are already decorated, so we just need to take care of the leaf labelling. For each leaf l of T', let us consider the smallest leaf of T that can be reached from l. We then number the leaves according to these "smallest descendants": the leaf with the smallest possible descendant gets the label 1, the second smallest — the label 2 etc.

Definition 6. For two tree monomials $\alpha, \beta \in \mathscr{F}_{\mathscr{V}}$, we say that α *is divisible by* β, if there exists a nonsymmetric divisor of α for which the corresponding tree monomial α' is equal to β.

Remark 2. Checking divisibility is important for the Gröbner bases algorithm. Thus, one has to put effort in finding an efficient implementation. Our approach is recursive; it is very much motivated by the choice of the platform. Preliminary experiments suggest that even though Haskell is very clever when it comes to handling recursive algorithms properly, for operads with seemingly infinite Gröbner bases a recursive approach often makes it impossible to handle elements of large arities efficiently enough. Thus, a non-recursive program computing Gröbner bases would be helpful; in particular, it would be very important to find a reasonably fast non-recursive algorithm for divisibility checking, in the spirit of algorithms of Knuth–Morris–Pratt [8] and Boyer–Moore [2] for string divisibility.

Proposition 3 ([4]). *A tree monomial α is divisible by β if and only if α can be obtained from β by iterated shuffle compositions with corollas.*

Assume that α is divisible by β. Take some sequence of shuffle compositions with corollas that produces α from β. This sequence can be applied to any tree monomial with the same number of leaves as β; we abuse the notation a little bit and denote that operation on tree monomials by $m_{\alpha,\beta}$, omitting the reference to the subtree of α which describes the corresponding divisor. This operation depends only on α and the specific subtree corresponding to the divisor β, but not on any particular sequence of compositions that creates α from β, so for the fixed occurrence of β in α, this operation is well defined.

Remark 3. Being able to compute the operations $m_{\alpha,\beta}$ in the most efficient way is crucial for the Gröbner bases algorithm. This means that much thought should be put in the data representation philosophy; keeping the divisibility information in a logical way helps to be efficient. We chose the approach where a divisor is replaced by a "black hole" (or, in other words, by a new corolla); this way the operation $m_{\alpha,\beta}$ corresponds to the simple insertion of a tree in the hole. We explain it in more detail in Section 3.3.3.

Definition 7. A tree monomial γ is called a *common multiple* of two tree monomials α and β, if it is divisible by both α and β. Tree monomials α and β are said to have a *small common multiple*, if they have a common multiple for which the number of vertices of the underlying tree is less than the total number of vertices for α and β, and the embeddings of α and β in this common multiple cover all its vertices.

Remark 4. Computation of small common multiples is one of the most frequently used operations in the Gröbner basis algorithm. Our approach (described in detail below) shares certain similarities with the algorithm that lists all shuffle permutations of the given type, even though is a little bit more sophisticated.

2.3. **Gröbner bases.** In this section, we work with shuffle operads over Vect defined by generators and relations. In other words, we study ideals of the free operad $\mathscr{F}_{\mathscr{V}}$ [4], where $\mathscr{V}$ is a nonsymmetric collection. We assume that $\mathscr{V}$ is endowed with a basis, a collection of sets M. We expect an ideal $\mathscr{I}$ to have an explicit system of generators $\mathscr{R}$ which are called relations of the operad $\mathscr{F}_{\mathscr{V}}/\mathscr{J}$. In the case of shuffle operads, the set of relations $\mathscr{R}$ consists of linear combinations of tree monomials. To apply our methods to the case of symmetric operads, pre-processing of the input data is required: first, the symmetric group action should be used to express operadic monomials in terms of tree monomials, second, relations should generate the corresponding ideal as a shuffle ideal. For that, it is necessary that they form a symmetric subcollection, that is, are closed under the symmetric groups action. For example, if we add to the set of relations the orbit of each relation (possibly adding some redundant relations), this condition is automatically satisfied.

An ordering of tree monomials of $\mathscr{F}_{\mathscr{V}}$ is said to be *admissible*, if it is compatible with the operadic structure, that is, replacing the operations in any shuffle composition with larger operations of the same arities increases the result of the composition. In Section 4.3, we shall describe some admissible orderings. All results below are valid for every admissible ordering of tree monomials.

Definition 8. For an element λ of the free operad $\mathscr{F}_{\mathscr{V}}$, the tree monomial α is said to be its *leading monomial*, if it is the maximal monomial that occurs in the expansion of λ with a nonzero coefficient (notation: $\mathrm{lm}(\lambda) = \alpha$). This nonzero coefficient (the leading coefficient of λ) is denoted by c_λ.

Remark 5. Whereas in the previous section we only worked with trees, from now on we use linear combinations of trees. Thus, it is important to implement working with tree polynomials, that is, linear combinations of tree monomials. The main requirement is that obtaining the leading monomial and the leading coefficient, being the most frequently used operations on polynomials, should be easy.

Definition 9. Assume that f and g are two elements of $\mathscr{F}_{\mathscr{V}}$ for which the leading monomial of f is divisible by the leading monomial of g. The element

$$r_g(f) := f - \frac{c_f}{c_g} m_{\mathrm{lm}(f),\mathrm{lm}(g)}(g),$$

is called the *reduction of f modulo g*.

Definition 10. Assume that f and g are two elements of $\mathscr{F}_{\mathscr{V}}$ whose leading monomials have a small common multiple γ. We have

$$m_{\gamma,\mathrm{lm}(f)}(\mathrm{lm}(f)) = \gamma = m_{\gamma,\mathrm{lm}(g)}(\mathrm{lm}(g)).$$

The element

$$s_\gamma(f,g) := m_{\gamma,\mathrm{lm}(f)}(f) - \frac{c_f}{c_g} m_{\gamma,\mathrm{lm}(g)}(g),$$

is called the *S-polynomial of f and g* (corresponding to the common multiple γ; note that there can be several different small common multiples).

In most classical literature on Gröbner bases, reductions are thought of as something different from S-polynomials. In our definition, S-polynomials include the reductions as a particular case. However, as we shall see below, the Buchberger's algorithm makes use of reductions on their own, and so they cannot be eliminated from the story.

Definition 11. Let $\mathscr{J}$ be an ideal of the free operad. $\mathscr{G}$ is called a *Gröbner basis* of $\mathscr{J}$, if for every $f \in \mathscr{J}$ the leading monomial of f is divisible by the leading monomial of some element of $\mathscr{G}$.

The main fact about Gröbner bases that makes them so useful is that the tree monomials that are not divisible by leading monomials of a Gröbner basis form a basis for the quotient of the free operad modulo $\mathscr{J}$. Thus, knowing a Gröbner basis for the defining relations of an operad allows to obtain important information about this operad.

Recall the Buchberger's algorithm for operads [4]. Its input is a set $\mathscr{R}$ of relations between elements of the free shuffle operad $\mathscr{F}_{\mathscr{V}}$. It repeatedly applies the following step:

> **Step of the Buchberger's algorithm:** Compute all pairwise S-polynomials of elements of $\mathscr{R}$. Reduce all these elements modulo $\mathscr{R}$ until they cannot be reduced further. Extend $\mathscr{R}$ by joining these reductions to it. If there are no new elements joined, terminate. (If there are new elements, the step is repeated for the newly obtained set $\mathscr{R}$.)

2.4. **Haskell.** Haskell [7] is a purely functional programming language with a powerful type system. Programming in Haskell has a declarative feel to it, in the sense that functions are defined by declaring equations for function evaluation — the equations are then used by the compiler with the first matching equation in the source code used for every application in the source code.

We have built the implementation we are discussing in Haskell, and will use occasional source code excerpts for illustration through the paper. See the appendix for the list of all the Haskell-specific functions in use in our code examples.

2.4.1. *Types.* Haskell depends on a strong adherence to its type system. Hence, any entity in the language possesses a type. There are types that are complete in themselves — such as **Bool**, **Int**, and types that are assembled from component types — such as the type [a] for lists containing elements of the type a. Functions, too, are first class citizens of the language, with a function taking input of type a and returning output of type b having type a $->$ b.

Real power in the Haskell treatment of data types appears with the freedom to declare your own types. The most complete way to do this is with the **data** declaration. This allows, easily, for both record and union types; where a record contains one value of each of the specified values, and a union contains one value out of the specified values.

As an example, similar to the datatype for trees that we will discuss at length in Section 3.1, a rooted tree is either a leaf node, or a root node with a list of subtrees. The arity of the root node will be precisely the length of the list of subtrees. And a node of arity 0 could be considered a leaf.

Thus, we may define a tree data type using the declaration

data Tree = Leaf | Node [Tree]

Here, Tree is the resulting data type, and Leaf and Node are constructors for elements of the data type. A typical tree might look like:

Node [Node [Leaf, Leaf], Node [Leaf, Leaf, Leaf]]

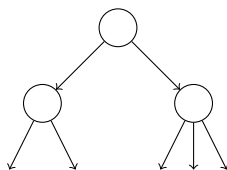The corresponding tree shape is shown in Figure 1.

FIGURE 1. A tree shape

This data type has been defined completely using the union type: a tree is either a leaf, or a node carrying a list of subtrees. We can extend the type using the record type construction into something that can carry labels both on leaves and vertices, making it usable to represent the decorated trees we use for representing operads.

Thus, we can introduce two *type variables*, to make the resulting tree type versatile, and define a tree type that takes labels of any type for the nodes, and labels of any type — independent of the node type — for the leaves by:

data Tree a b = Leaf a | Node b [Tree]

Here, an element of the type Tree is either a leaf, equipped with a value of type a, or a node, equipped with a value of type b as well as a list of subtrees.

Hence, an example of type Tree **Int Char** would be:

```
Node  b  [ Node  c  [ Leaf  '1 ',  Leaf  '3 '] ,
         Node  a  [ Leaf  '2 ',  Leaf  '4 ',  Leaf  '5 ']]
```

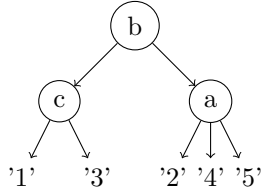The corresponding decorated tree is shown in Figure 2.

FIGURE 2. A decorated tree

2.4.2. *Functions.* The second important part of understanding the way Haskell works is the functions. A function is defined by its type, and by what it makes to the input arguments it takes. Haskell views a function with several input parameters as a function taking one value and returning a function expecting one less parameter. Thus (+) is a function that when applied to the value 2 will return a function (2+) that in turn increases its parameter by 2.

A function specification in Haskell has two components. First off is the (optional) type declaration. An example, taken from our source code:

```
operadicBuchberger  ::
        (Ord  a ,  Show  a ,  TreeOrdering  t ,  Fractional  n) =>
        [ OperadElement  a  n  t ]  −> [ OperadElement  a  n  t ]
```

This type declaration alone tells us a number of things about the function, and the parameters it takes and returns. First comes the name of the function: operadicBuchberger. It is the top level function to run the Buchberger algorithm on a set of operadic relations. Next is the :: — signifying that a type declaration follows.

Following the name and the ::, we give an (optional) list of assumptions on the type variables involved in constructing all types of the type declaration: we need a to be a type that can be sorted and printed, we need t to be a TreeOrdering, i.e. an implementation of the monomial ordering algorithms we use. Finally, we expect n to be a numeric type implementing a field.

Following the expectations follows the symbol =>, signifying the start of the actual type declaration. And we read off that the function has type [OperadElement a n t] −> [OperadElement a n t], or in other words that operadicBuchberger takes a list of operad elements with a certain type of vertex labels (signifying the operations in the free operad), a certain type of coefficients and a certain monomial ordering.

After the type declaration, a sequence of equational declarations follow, containing the bulk of the function definition. As a function is called, these equations are processed in the order they are stated until one is found such that the left hand side of the equation matches the parameters submitted to the call. Once a match is found, the code on the right hand side is executed and the result is returned as the value of the function call.

Again, an example may be in order. We can write a function for the Tree type we described above that allow us to recognize leaves:

```
isLeaf :: Tree a b -> Bool
isLeaf (Leaf leaflabel) = True
isLeaf (Node nodelabel subtrees) = False
```

3. Internal representations

Data representations and algorithms go hand in hand. A good algorithm will suggest a data representation that makes the steps of that algorithm easier; a good data representation will make the algorithms handling the data obvious and efficient from the storage methods. We have tried to find representations for the data types required for the Gröbner basis algorithms that will make the subtasks we have identified easy to implement efficiently.

We discuss representation for decorated trees in Section 3.1. The elements of a free operad in the category Vect are formal linear combinations of decorated trees, and the representation of these is discussed in Section 3.2. Next up is the special type trickery needed to represent the *black hole* trees first introduced in Remark 3 on page 6. We introduce two coproduct types useable for tagging vertices of trees while preserving some or all of the vertex tags in Section 3.3.1. This way, we can designate a corolla a black hole, or an embedding point in a small common multiple. We discuss the small common multiple structures in Section 3.3.2 and the black hole tagging in Section 3.3.3. Finally, we discuss the representation we use for permutations in Section 3.4.

3.1. **Decorated trees.** We recall that the free operad is built with trees decorated at the corollas with elements of the generating collection and with leaves decorated with an ordered set.

While we expect the trees representing the basis of a free operad to have integer leaf labels, some of the lower level tasks are easier if we can also represent trees with other types of leaf labels. Hence, we will define one underlying tree type, PreDecoratedTree, and derive another tree type, DecoratedTree representing the basis elements of free operads.

Hence, we will build our software with the decorated tree as our fundamental building block. We represent trees using a data type that encodes internal vertices and leaves as different, allowing each to carry a label. This guides us to the data type declaration

```
data (Ord a, Show a) =>
    PreDecoratedTree a b =
        DTLeaf !b |
        DTVertex {
            vertexType :: !a,
            subTrees :: ![PreDecoratedTree a b]}
        deriving (Eq, Ord, Read, Show)
```

This is essentially the same as the tree type we discussed at the end of 2.4.1. It is decorated with more expectations, and some Haskell idioms to automatically generate functionality. Hence, the **deriving** clause will make the tree type automatically allow equality checks, sorting and methods to serialize and deserialize

the data into strings. The sorting induced by the **deriving** clause, however, is not in general a monomial ordering, and we introduce further types in the code to introduce admissible monomial ordering.

Furthermore, the vertexType and subTrees clauses automatically generate functions that allow us to extract the node label and the list of subtrees from a corolla.

Since we occasionally, but not very often, will feel a need to decorate the leaves with something different from integers, we define our tree type as a different type from the type we use with the users in the end. We define a type synonym DecoratedTree a that stands for PreDecoratedTree a **Int**, so that the user should only ever have to see DecoratedTree occuring.

We also define a number of utility functions for operations on these trees: methods to apply a function to each node label, and to apply a function to each leaf label, as well as functions to easily construct corollas and leaves, and to easily determine the arity of a corolla and the total number of leaves of a tree.

One pattern that reoccurs a lot here is the basic tree recursion shape. We give, here, as an example, the code to apply a function to each vertex label:

```
vertexMap ::
    (Ord a, Show a, Ord b, Show b) =>
    (a -> b) -> PreDecoratedTree a c -> PreDecoratedTree b c
vertexMap f (DTLeaf i) = DTLeaf i
vertexMap f (DTVertex t ts) =
    DTVertex (f t) (map (vertexMap f) ts)
```

There is some boiler plate — the type variables a and b have to match the assumptions needed to build a tree. But then the code just states that applying f to the node labels if you encounter a leaf just returns the leaf unchanged. If, however, you encounter a vertex, you apply the function to the label and construct a new vertex with the new label and with the results from applying the function recursively to all subtrees.

This structure faithfully reproduces planar rooted trees with leaf and node labels. Using the ordering on the leaf labels, we can represent any symmetric node-labeled tree this way. Using only the tree monomials from the free shuffle operad, finally, means placing restrictions on the permutations the leaf labels are allowed to display. Specifically, at any vertex, the minimal leaves of all its subtrees need to occur in sorted order in its list; this is exactly the choice of a planar representative from Section 2.2. Hence, in Figure 3, the left tree is a valid tree monomial, whereas the right tree is not. The points where the assumption fails are denoted by filled circles.

3.2. **Operad elements.** Recall that an element of the free operad in the category of vector spaces over a field is a formal linear combination of decorated trees. To represent operad elements, thus, we need to be able to represent formal linear combinations — sorted according to the appropriate monomial ordering.

Representing this computationally in Haskell is a three-step process. First, we represent monomial orderings. Then, we represent trees equipped with a monomial ordering. Finally, we use the Data.Map Haskell standard library implementation to represent a partially defined function taking decorated trees to coefficient values.

These last functions become equivalent with formal linear combinations once we define an arithmetic on them:
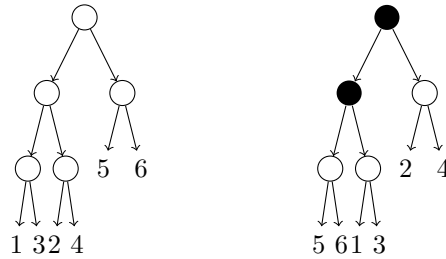
FIGURE 3. A tree monomial, and a decorated tree which is not a tree monomial

$$(f + g)(\alpha) = f(\alpha) + g(\alpha)$$
$$(c \cdot f)(\alpha) = c \cdot f(\alpha)$$

This equivalence is given by, for a function f, forming the formal linear combination $\sum f(\alpha)\alpha$. The reverse is given by taking a formal linear combination $\sum c_\alpha \alpha$ and forming the function $f : \alpha \mapsto c_\alpha$.

Monomial orderings are represented as *empty types* — constructors without values other than having distinct constructors. These types, then, are made to implement *type classes* — the Haskell way to do polymorphism. Implementing a type class means that the type class defines specific functions, and the type class implementation defines the implementation of these functions, in a separate manner for each type that implements the type class. We pair trees with orderings using the record type facility — a tree with an ordering is a tree paired with an ordering. A number of easy conversion functions between ordered and non-ordered trees make interfacing with this layering easier.

As for the partial function definition, the datatype Data.Map works for finite such definitions by way of a lookup table: an entity of type Data.Map is a search tree that can be queried for the value associated to a particular tree, and that works, internally, by maintaining a balanced binary search tree. In particular, this makes the use of Data.Map very dependent on an efficient implementation of the monomial ordering methods, and one early adjustement we decided on was to overlay a thin, encapsulating module around Data.Map that would cache the relevant information needed to perform the most common monomial orderings.

This last point is worth elaborating on. We found in early implementations that storing decorated trees in a binary search tree, and having monomial orderings depending on a significant number of tree traversals in order to construct the ordering invariants, lead to an extraordinary amount of tree traversals. In our first working implementation, we found that over 60% of the computation time was spent traversing trees for comparisons triggered by the use of Data.Map for storage. Every operation on such operad elements would incur many tree comparisons, each of which would incur several tree traversals. To deal with this, we wrote a wrapper around the storage type that would perform the tree traversals for the orderings described in Section 4.3 once at the creation of an operad element, and store this with the tree. Subsequent interactions with this particular element would use this cached value for all comparisons — until the point a new element was constructed,

as a modification of the previous one, at which point the comparison values would be recomputed. Using the wrapper, the proportion spent on tree comparisons and the related building block functions dropped to less than 5%.

3.3. **Trees with holes and tagged nodes.** Recall that for two tree monomials α and β such that α is divisible by β, it is possible to define the function $m_{\alpha,\beta}$ that reconstructs the surroundings of β in α; this function is applicable to any other operad element of the correct arity.

An algorithm for finding divisors of trees, as the Gröbner basis algorithms makes heavy use of, would need an efficient way to represent the data needed for such a reconstruction. We decided to do this by representing holes punched in trees by corollas with a specific marking.

Similar to this idea is the data representation we found to be efficient to represent a small common multiple of two trees in such a manner that the divisor data for both trees is easily reconstructed: such a small common multiple will have one of the trees dividing it at the root, and the other somewhere in the tree. We found a natural way, in Haskell, using union data types, to represent a tree with a single vertex marked.

3.3.1. *An aside on data types and labels.* The union data type construction in Haskell has a standard library implementation, with quite a bit of predefined functionality: **Either** a b. This defines for us constructors **Left** and **Right** carrying values of types a and b respectively.

One special case of the **Either** type is when one of the two types is empty. This case has been given a name of its own: **Maybe** a and comes with new constructors — **Just** taking the place of **Left** and **Nothing** taking the place of **Right**.

These type constructions turn out to be exactly what we need to signify marked nodes and removed nodes.

3.3.2. *Trees with marked nodes.* We generate, in order to mark some of the nodes of our tree, a new tree from the old one with changed node labels. Instead of labelling our tree with some type a, we now label them with **Either** a a. This has the effect of increasing the amount of information carried by each node — in addition to the original node information it now also carries a binary choice for each node: is it a **Left** or a **Right** instance of the label type?

Hence, we can in our code for generating small common multiples return a tree labeled in **Either** a a, and making sure it only contains one single node labeled **Right** — namely the point of attachment for the second tree. We know that one of the two trees has to be rooted at the root of the small common multiple — otherwise it would not cover all vertices.

The algorithm we use to find small common multiples, elaborated on in Section 4.1, has the following basic structure. In order to find small common multiples of α_1 and α_2 we go through the steps:

(1) To find small common multiples of α_1 and α_2 sharing the common root, traverse both trees, checking compatibility at each step and whenever one tree yields a leaf, attach the remaining subtree of the other tree. Tag the root of the returned small common multiple as **Right** label.

(2) Recurse through vertices v of α_2, applying the previous step to find small common multiples of α_1 and the subtree α_2^v of α_2 rooted at v. For each

such common multiple γ, form a new tree by taking α_2 and replacing α_2^v with γ.

As a result, the only point where a node is tagged with the **Right** is when a rooted common multiple is found, and the recursion ensures that the rest of the tree is rebuilt so that α_2 is embedded with a shared root with the common multiple.

In order to find all small common multiples, we need to perform this algorithm once again with the trees interchanged, so that we may find small common multiples with α_1 embedded at the root.

3.3.3. *Trees with holes.* As for the divisor reconstruction, we have some embedding of the tree monomial β into the tree α, and we want to retain the information of the entire tree excepting the part that corresponds to β.

The way we do this is to collapse the embedded copy of β into a single corolla of the correct arity, and keeping the rest of the tree intact. This corolla, then, is marked — forgetting any original corolla type markings — to signify that it forms an embedding point.

This marking, in turn, is achieved by changing the type of all labels from the type a to the type **Maybe** a. That way, the part of the tree that needs to stay intact is marked with **Just** label for what previously was marked with label, and the corolla holding the position of the hole is marked with **Nothing**, distinguishing it from the other nodes.

For this reason, we introduce the type alias Embedding a, defined to be of the type DecoratedTree (**Maybe** a). Hence, the division algorithm described in Section 4.2 will take the two trees α and β as parameters, and return an embedding of the shape of α with a subtree isomorphic to β taken out. See Figure 4 for an example.
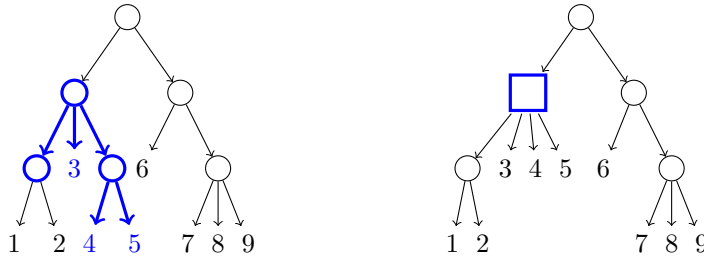

FIGURE 4. Taking away a subtree results in a tree with a hole

The reconstruction algorithm, on the other hand, takes a shape representing some embedding of β in α, and a new tree monomial γ of the same arity as β, and returns the corresponding tree $m_{\alpha,\beta}(\gamma)$.

3.3.4. *On data types not used.* There is an alternative way to construct the trees with a single hole, and the trees with a single tag, that gives abstract guarantees that the resulting tree will have exactly one hole, or tag. The key here is the approach taken with zippers [6] or with differentiating data types [1]. The derivation of the resulting data type guarantees that the new tree has exactly one hole, which currently is done by further tree-walks.

As an example, the resulting data type for our choice of tree representation, for the trees with holes would be:

```
data TreeHole a b = Leaf a
          | Node b [Tree a b]
          | HoleParent b [Tree a b] (TreeHole a b) [Tree a b]
          | Hole [Tree a b]
```

3.4. **Permutations.** Since the most common use for permutations in this project is to label leaves of trees, and to reorder subtrees for composition, we have decided to store our permutations as lists of images. This choice is reinforced by the lack of need for compositions and decompositions of permutations.

This representation yields a simple method to reorder a list of objects in the order specified by a permutation — an operation we have reason to perform often in the code, for instance in order to decorate leaves of a labelled tree according to their integer decorations: we pair off the elements we want to reorder with the image list. Then we sort the pairs, giving priority to the comparison of the image indices. Stripping off the indices, finally, gives us the reordered list of elements. This is a code idiom we have used at several points in the code base.

4. Algorithms

With the data structures we use settled in Section 3, we now turn to the algorithms that implement the core components of the Buchberger algorithm from Section 2.3. To compute S-polynomials, we have to find all small common multiples of leading terms (together with occurences the divisors in these small common multiples), and then compute the actual S-polynomials. The latter requires many reconstructions of trees via insertions of lower terms in the black hole. After that, we compute reductions, which requres divisibility checks and many reconstructions again. We discuss algorithms for all these key steps below. In Section 4.1, we adapt the idea for finding block permutations to give us an efficient algorithm for finding small common multiples. In Section 4.2, we meet the division algorithm, creating the black hole trees, and the reconstruction algorithm, re-inserting a tree in the black hole. Finally, in Section 4.3 we discuss the family of monomial orderings that the software package implements.

4.1. **Finding small common multiples.** An algorithm that lists all small common multiples of two given trees α_1 and α_2 consists of several steps. If we forget all leaf labels of a tree monomial, we end up with a planar tree with labelled vertices. We earlier called such trees nonsymmetric tree monomials; they form a basis in the free nonsymmetric operad generated by the same nonsymmetric collection as our free shuffle operad. Describing all small common multiples is naturally split into two steps: forgetting about leaf labels and finding a nonsymmetric small common multiple, and then acquiring all possible leaf labellings of the resulting trees.

The first step is more or less trivial: small common multiples of two nonsymmetric tree monomials are superpositions of the trees for which all labels of vertices agree with each other. Thus, to list all such small common multiples, we should go through all ways to identify the root vertex of one of the trees with a vertex of another tree, and check that there are no contradictions between the successors of these two vertices. This can be easily done recursively.

The second step is a bit more tricky, but still requires a straightforward recursive algorithm. To recover all admissible leaf labellings giving a common multiple α, we have to solve the problem of finding all possible linear orders on a poset of a special

type. The elements of that poset are leaves of the nonsymmetric tree monomial, and we say that $a < b$ if there exist two vertices u and v of the nonsymmetric tree monomial such that

- a is the smallest leaf reachable from u and b is the smallest leaf reachable from v;
- u and v are leaves of the occurence of α_i in α (where i is either 1 or 2), and $u < v$ in the ordered leaf set of α_i.

A labelling of the leaves of α makes it a small common multiple if and only if it extends the above ordering to a linear ordering. We shall recover all such labellings recursively. Our poset essentially consists of two intertwined and intersecting linear orders, and the maximal element of the labelling set should label the maximal elements of one of the two orders (under the additional condition that it cannot occur as a non-maximal element in the other linear order). For each such option, we are left with a similar problem where the size of the labelling set is one less, so we can use recursion.

4.2. **Divisibility and reconstructions.** Finding all embeddings (as a divisor) of β into α can be easily reduced to finding out whether or not β is embedded into α in such a way that they share the common root. If we know how to solve this problem, we should just solve it for all subtrees α' of α rooted at various vertices. To solve this problem, it suffices to recurse down the tree checking that the same holds for all subtrees, and then check that the total leaf orderings match. At this stage, to check that the orderings match, one can look at the planar orders of leaves and compare them; this appears to be the best way to do it for our recursive algorithm.

Specifically, if we try to find an embedding of β into α sharing the common root, we first verify that the root vertices share the label and arity. If this is the case, we can pair up the subtrees β_i of β and the subtrees of α_i of α, and then find rooted embeddings of each β_i in each α_i. If all these succeed, we expect to get as a result from each a tree m_{α_i,β_i} for each pair with a hole punched out at the root corresponding to a subtree looking like β_i.

At this point, we need to patch things up. The root nodes match, and the subtrees have already found embeddings. Checking the leaf orders, we then need to merge all the subtrees with holes into a tree with hole that gets returned up to earlier recursion levels. This is done by simply creating a new hole vertex, and then attaching all subtrees of the hole subtrees, as shown in Figure 5.
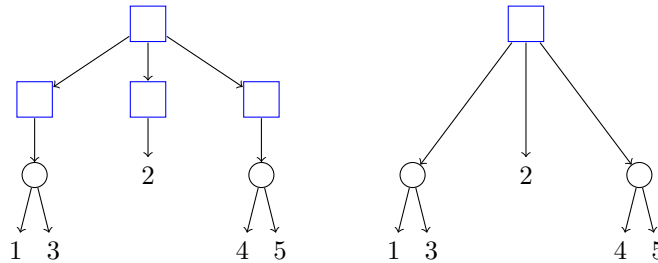

FIGURE 5. Merging subtrees with holes

Once we find an embedding of β, we store this embedding as a tree monomial obtained from α by collapsing the occurrence of β into a single vertex. To reconstruct α from that, we insert β in the "hole" in such a way that the leaves of β match the outputs of the "hole" (order-wise).

Inserting β into the hole, specifically, means that we replace the leaves of β with the subtrees of the hole in the tree with the hole, in the order specified by the labels of the leaves of β, and then replace the hole and all its subtrees in the full tree with a hole by this extended β, as shown in Figure 6.
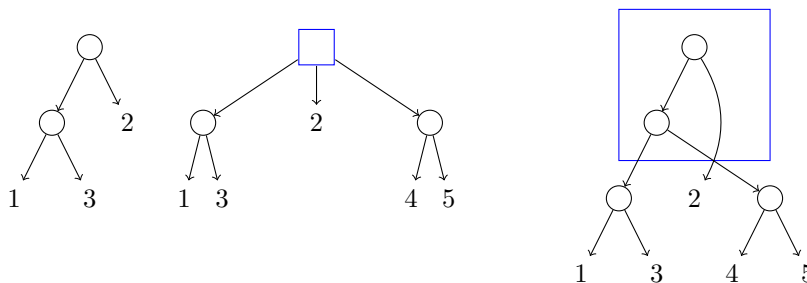

FIGURE 6. Inserting a tree in a hole

4.3. **Monomial orderings.** Let us describe some admissible orderings. As one can see, each definition will be either immediate to implement because of the storage types we use or straightforward recursive.

Let α be a tree monomial with n inputs in the free operad $\mathscr{F}_M$. We associate to α a sequence $(a_1, a_2, \ldots, a_n)$ of n words in the alphabet M, and a permutation $g \in S_n$ as follows. For each leaf i of α, there exists a unique path from the root to i. The word a_i is the word composed, from left to right, of the labels of the vertices of this path, starting from the root vertex. The permutation g lists the labels of leaves in the order determined by the planar structure (from left to right).

Now, to compare two tree monomials we always compare their arities first. If the arities are equal, there are several different options of how to proceed. Sequences of words can be compared lexicographically using either the degree-lexicographic ordering of words, or the reverse degree-lexicographic ordering (either the longer word is greater, or vice versa; for words of the same length the comparison is lexicographic). Permutations can be compared in the lexicographic or reverse lexicographic order. Also, the result depends on what we compare first, the permutations or the sequences of words. This gives rise to eight candidates for an ordering; we name these candidates **PathPerm**, **RPathPerm**, **PathRPerm**, **RPathRPerm**, **PermPath** etc. (the names are self-explanatory).

Proposition 4 ([3]). *All the above orderings are admissible.*

In fact, to compare words one may use any admissible ordering of the monomial basis of the free algebra, for example, the lexicographic ordering, or the reverse lexicographic one: the resulting ordering of tree monomials will be admissible as well.

Appendix: Haskell constructions used

Bool: The boolean truth values type. Has values **True** and **False**.

Int: The bounded integer type. Has values, on a 32 bit machine, from the interval $[-2147483648, 2147483647]$.

Char: The single character type.

[a]: The type of lists of elements of type **a**.

a → b: The type of a function from a type **a** to a type **b**.

data: The declaration of a new data type.

Ord: The type class that defines the ordering functions `<`, `>`, `<=`, `>=`, and `compare`.

Eq: The type class that defines the equality testing function (`==`).

Show: The type class that defines the serialization function **show**.

Fractional: The type class that defines a type to implement a field.

(::): The syntax element indicating a type declaration.

(⇒): The syntax element delimiting type assumptions from the type declaration.

(!): When occurring in a type declaration, forces strictness in the corresponding part.

(|): When occurring in a type declaration, delimiting the union type components. Hence, a type declared as **data** T = A **Int** | B **Char** is either an **Int** with the constructor A, or a **Char** with the constructor B.

TreeOrdering: A type class created by our code carrying information about the chosen monomial order.

OperadElement: The type created by our code representing, internally, a linear combination of tree monomials with associated monomial orderings.

(+): The addition function.

deriving: Used in a **data** declaration. It will automatically generate implementations of the type classes listed.

map: A higher order function that applies another function to every element in a list.

References

[1] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani. ∂ for Data: Differentiating Data Structures. *Fundamenta Informaticae*, 65(1-2):28, 2004.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762–772, 1977.

[3] Vladimir Dotsenko. Freeness theorems for operads via Gröbner bases. *arXiv:0907.4958*.

[4] Vladimir Dotsenko and Anton Khoroshkin. Gröbner bases for operads. *Duke Math. Journal*, to appear.

[5] Eric Hoffbeck. A Poincaré–Birkhoff–Witt criterion for Koszul operads. *Manuscripta Math.*, to appear.

[6] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

[7] S. P Jones, editor. *Haskell 98 language and libraries: the revised report*. Cambridge Univ Pr, 2003.

[8] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[9] Jean-Louis Loday and Bruno Vallette. Algebraic operads. In preparation.

[10] Martin Markl, Steve Shnider, and Jim Stasheff. *Operads in algebra, topology and physics*, volume 96 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 2002.

Dublin Institute for Advanced Studies, 10 Burlington Road, Dublin 4, Ireland and School of Mathematics, Trinity College, Dublin 2, Ireland
E-mail address: `vdots@maths.tcd.ie`

Department of Mathematics, Stanford CA 94305, USA
E-mail address: `mik@stanford.edu`