

Operadic Gröbner Bases: An Implementation

Vladimir Dotsenko^{1,*} and Mikael Vejdemo-Johansson^{2,**}

¹ Dublin Institute for Advanced Studies and School of Mathematics, Trinity College
Dublin, Ireland

² Department of Mathematics, Stanford University
`mik@stanford.edu`

1 Introduction

In an upcoming paper [1], the first author and Anton Khoroshkin define the concept of a Gröbner basis for finitely presented operads, prove the diamond lemma for these Gröbner bases, and demonstrate that having a quadratic Gröbner basis is equivalent to the existence of a Poincaré-Birkhoff-Witt basis. As demonstrated by Eric Hoffbeck [2], an operad with a PBW basis is Koszul. Thus, out of this emerges an entirely computational framework for proving Koszulness, as well as the possibility to build tools for exploration of operads by means of explicit calculation.

The authors have, in [3], provided a computer implementation of the algorithms specified by Dotsenko and Khoroshkin. At the core of the paper [1] lies the recognition that every symmetric operad can be thought of as a shuffle operad, and forgetting “unnecessary” symmetries does not affect relevant results of linear and homological algebra for operads; therefore, the respective computation may be restricted to the simpler category of shuffle operads without restricting any conclusions drawn.

1.1 Shuffle Operads

For exact definitions, we refer the reader to [1,3]. For the purpose of this short communication, we shall concentrate on a less precise definition of symmetric, and shuffle operads.

Definition 1. A symmetric (resp. shuffle) operad is a collection $O(n)$ of vector spaces, one for each n , together with linear maps

$$\circ_{\sigma} : O(n) \times O(m_1) \times \cdots \times O(m_n) \rightarrow O(m_1 + \cdots + m_n)$$

called composition maps. The composition maps are parametrized by arbitrary permutations in $S_{m_1 + \cdots + m_n}$ (resp. by shuffle permutations of type (m_1, \dots, m_n)) that provide symmetry actions to the operad. These maps are required to fulfill associativity conditions and allow for a unit for the composition.

* 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.

As with so many other things, it is the identification of the *free* objects that we gain both a mental and a computational model for the objects at hand. For operads, the free objects consist of *decorated trees*, with a (rooted) tree being a non-empty connected directed graph T of topological genus 0, with each vertex being equipped with at least one incoming edge and exactly one outgoing edge. We allow for some edges to only have one vertex – the other being ignored for our purposes – call such edges external. Each tree has exactly one external edge: the *root* or *output*, and some number of ingoing edges, called *leaves*.

Given a collection M , we can consider the collection of all trees *decorated by* M , by which we mean trees such that each vertex with n outgoing edges is equipped with some element of $M(n)$. Such a decorated tree, we call a *tree monomial over* M . The collection of the vector spaces spanned by all such tree monomials with exactly n leaves is denoted by $\mathcal{F}_M(n)$, and they form a collection denoted by \mathcal{F}_M .

Composition in the free operad can be defined on basis elements through gluing leaves to roots, and re-arranging the leaves in an admissible way. For symmetric operads, all permutations are allowed, while shuffle operads have a smaller class of admissible permutations: the shuffle permutations, as described in [1]. The re-arranging can either be imagined by allowing branches to cross as the tree is drawn at; or by requiring a planar drawing of each tree, but instead decorating all leaves with integers, and allowing the permutations to act on these integers.

1.2 Gröbner Bases for Operads

We may define divisibility for tree monomials by saying that α divides β if there is some sequence of compositions that starts with α and ends with β . We can produce the equivalent operations to S-polynomials and reductions for the Buchberger algorithm by finding such a dividing sequence $m_{\alpha\beta}$ for a pair of leading tree monomials, according to some tree monomial ordering, and applying $m_{\alpha\beta}$ to all tree monomials in some given free operad element.

This allows us to reproduce the Buchberger algorithm, with trees and non-linear compositional structures instead of the linear monomials known from commutative and non-commutative Gröbner bases. Once we are able to define greatest common divisors and least common multiples for tree monomials, using these $m_{\alpha\beta}$ -operations, the resulting algorithms look very familiar to those working with computational algebra.

Our interest in these Gröbner bases lie in part in their power for the theory of operads: providing computational proof for Koszulness of specific operads, and aiding in the computational exploration of the theory – but also in the way that Gröbner bases and the computational theory of operads encompasses all Gröbner-like theories in one single framework. Commutative and non-commutative polynomial ring Gröbner bases occur as operadic Gröbner bases concentrated in degree 1.

2 An Example

To illustrate the concepts and techniques, consider the algebraic theory that stipulates a single binary operation $*$, and requires of this the rules $(a*b)*c = -a*(b*c)$

and $(a * c) * b = a * (b * c)$. This universal algebra is captured by the operad defined as a quotient of the free shuffle operad on a single generator \vee by the ideal generated by the two tree polynomials on the left. These give rise to an S-polynomial (using the PathPerm ordering described in [3]) – the one consisting of only the monomial on the right; which will not reduce further.



```
% ghci -cpp Math.Operad
*Math.Operad> let v = corolla 2 [1,2]
*Math.Operad> let [g1t1,g1t2,g2t2] =
  [shuffleCompose 1 [1,2,3] v v,
   shuffleCompose 2 [1,2,3] v v,
   shuffleCompose 1 [1,3,2] v v]
*Math.Operad> let ac =
  [(oet g1t1) + (oet g1t2), (oet g2t2) - (oet g1t2)]
  :: [OperadElement Integer Rational PathPerm]
*Math.Operad> let acGB = operadicBuchberger ac
*Math.Operad> length acGB
3
*Math.Operad> putStrLn $ pp acGB
[
+1 % 1*m2(m2(1,3),2)
+(-1) % 1*m2(1,m2(2,3)),

+1 % 1*m2(m2(1,2),3)
+1 % 1*m2(1,m2(2,3)),

+2 % 1*m2(1,m2(2,m2(3,4))),
]
```

Fig. 1. Example session: computing the Gröbner basis of the operad that controls anti-commuting associative algebras: $m(m(a,b),c) = -m(a,m(b,c))$ and $m(m(a,c),b) = m(a,m(b,c))$. The symbol % denotes an element of \mathbb{Q} , so that $a\%b = \frac{a}{b}$. The first two clusters in the output are the original generators, and the third is the non-trivial S-polynomial added by the computation.

3 Implementing Gröbner Bases for Operads

For the implementation of the Buchberger algorithm, we chose to work in the programming language Haskell.¹ [4] For a variety of reasons, including the relative ease with which new datatypes can be constructed, and mathematical thoughts can be all but transliterated into the programming language itself, this choice made the production of a first implementation easy and even pleasant.

¹ A working version of our software package can be found at the Hackage repository <http://hackage.haskell.org/package/Operads>

However, there are plenty of issues with the implementation, and its platform. The platform choice makes the package uninviting to the casual user, and hard to integrate into other mathematical software systems. The relatively specialized programming paradigm is unusual enough that the language itself forms an additional barrier to entry, and both optimization, debugging and code analysis are made more difficult by the sometimes highly unintuitive way that the declarative programs transform into machine code.

At the core of the representation of an operad is the tree; and thus the ease with which Haskell represents trees helped making the implementation work easy; with a definition somewhat similar to

```
data Tree = Leaf | Node [Tree]
```

equipped with appropriate functionality and decorations, implementing the arithmetic of free operads became straightforward. However, it is with the trees that the core of the difficulty implementing these algorithms lies as well: in our profiling experiments, the real deep time sinks have invariably been functions that traverse the trees in order to determine their value – primarily the monomial ordering functions that get called repeatedly by every single operation that modifies a tree polynomial. Introducing caching to the polynomial storage type helps, but even so, tree traversals take up most of the time.

Hence, any other implementation will have to be carefully done to capture the tree structures in a way that is amenable to manipulation while still efficient in handling.

We would like to call for further implementations; for the modification of this implementation into something more accessible, more usable, faster, leaner, and better.

References

1. Dotsenko, V., Khoroshkin, A.: Gröbner bases for operads. *Duke Math. Journal* 153(2), 363–396 (2010)
2. Hoffbeck, E.: A Poincaré–Birkhoff–Witt criterion for Koszul operads. *Manuscripta Mathematica* 131, 87–110 (2010)
3. Dotsenko, V., Vejdemo-Johansson, M.: Implementing Gröbner bases for operads. *Séminaires et Congrès* (2009) (to appear)
4. Jones, S.P. (ed.): *Haskell 98 language and libraries: the revised report*. Cambridge Univ. Pr., Cambridge (2003)