Arboretum.hs: Symbolic manipulation for algebras of graphs

EUGEN BRONASCO
Mathematical Sciences,
Chalmers and Gothenburg University, Sweden
(email: bronasco@chalmers.se*)*

JEAN-LUC FALCONE
Department of Computer Science,
University of Geneva, Switzerland
(email: Jean-Luc.Falcone@unige.ch*)*

GILLES VILMART
Section of Mathematics,
University of Geneva, Switzerland
(email: Gilles.Vilmart@unige.ch*)*

Abstract

We design the Arboretum.hs package for symbolic computations with algebras of trees and more general graphs in Haskell. Thanks to the declarative nature of functional programming, the package's implementation closely follows mathematical definitions, making the code intuitive and transparent for users working with algebraic and combinatorial structures.

To assist with current mathematical research, Arboretum.hs supports experimentation by facilitating the introduction of new algebraic operations, as well as providing functionality for rendering trees and forests through LaTeX integration.

Compared to recent imperative implementations in languages such as Julia or Python, Arboretum.hs offers greater flexibility for manipulating and extending tree-based structures. Its use of Haskell enables safe programming and strong compile-time guarantees, serving both as a practical computational tool and a foundation for further research in algebraic combinatorics, beyond the setting of trees usually considered in the implementation of Butcher series, which are a fundamental tool for the analysis of numerical integrators.

Many areas of mathematics make extensive use of graphs to encode algebraic and combinatorial structures. This is evident in non-associative algebra, where trees encode the different ways of bracketing products, and more generally in operad theory, where they naturally represent the composition of operations, see [16, 31]. Similar combinatorial structures arise in rough path theory, as developed in [32] and [22], and in the theory of regularity structures introduced in [26] and further developed in [9]. In numerical analysis, trees play a central role in the study of Butcher series and geometric numerical integration, starting from the work of [11] and further developed in [17, 23, 25, 38]. In these settings, one studies algebras defined on classes of graphs (most commonly rooted trees or forests) and analyzes the algebraic and combinatorial properties that emerge.

Working with such algebras quickly leads to symbolic computations that are intricate and repetitive. Despite their importance, general-purpose software for computations with

algebras of trees and forests remains limited. Existing implementations are often tailored to specific applications or implemented within larger computer algebra systems, where they are developed for narrow purposes and do not provide a unified or easily extensible framework for general computations with trees and forests.

In this paper, we present `Arboretum.hs`, a Haskell package that provides a systematic and rigorous framework for computations involving algebras of graphs. The package is designed with generality and composability as primary goals. It introduces basic abstractions for graded vector spaces and provides tools for defining and manipulating algebras of trees and forests in a way that closely follows their mathematical definitions.

As a concrete example, we implement the pre-Lie grafting algebra of decorated forests, which plays a role in the analysis of numerical integrators. The package also includes tools for generating graphical PDF representations of decorated forests, with support for customization and straightforward extension to more general classes of graphs.

The design of `Arboretum.hs` is guided by three core principles:

(1) Readability: the code mirrors the underlying mathematics as closely as possible, making implementations easy to follow,
(2) Extensibility: new structures and operations can be added without major changes to the existing codebase,
(3) Testing: correctness is emphasized throughout, and the package is structured to make validation of algebraic properties straightforward.

These goals strongly influenced the choice of Haskell as the implementation language. Haskell's syntax and functional style are well suited to expressing algebraic definitions, particularly recursive objects such as trees and forests. Immutability and pure functions simplify reasoning about program behavior, while the static type system allows algebraic structures to be reflected directly in the code and checked at compile time. This stands in contrast to dynamically typed languages such as Python or Matlab, where similar errors often surface only at runtime.

Lazy evaluation is another important feature in this context. It allows computations to be deferred until their results are needed, which is particularly well suited to infinite objects such as Butcher series, which are formal sums of trees used in the analysis of numerical integrators and discussed in detail in Section 2. This enables a natural treatment of infinite sums without explicit truncation, an approach that is considerably more cumbersome in languages such as Python, Julia, or Matlab.

Haskell also emphasizes composability and modularity, making it easy to extend the codebase. In addition, property-based testing tools such as QuickCheck allow algebraic laws to be expressed directly as testable properties, providing a level of automated verification that is harder to achieve in dynamically typed environments.

We emphasize that `Arboretum.hs` prioritizes readability and user experience over raw performance. This choice is deliberate. The package is intended both as a computational tool and as a platform for exploration and experimentation with algebras of graphs. Where performance becomes critical, Haskell's Foreign Function Interface allows integration with high-performance code written in C, Julia, or Python. To reduce the learning barrier, the package avoids advanced Haskell features such as monads or type-level programming, making it more approachable to users without extensive experience in functional programming.

Several related projects exist. A prominent example is `BSeries.jl`[1], which replaces the earlier Python implementation `BSeries.py`[2]. Its focus is on efficient implementations of classical Butcher series techniques and Runge-Kutta methods. Developed with different priorities, `BSeries.jl` primarily targets numerical analysts and well-established structures from numerical analysis. As a result, it is less flexible from an algebraic and combinatorial perspective, and less suited for exploring extensions of tree and forest algebras beyond the classical setting; see [28] for details.

We also mention the master's thesis of [43], which introduces a Python package `pybs`[3] for automating classical computations in the theory of Butcher series. Additionally, [36] present recursive formulas for operations on planar forests, motivated in part by the development of a Haskell package for their computation, and includes a complementary discussion of the benefits of using Haskell in this context.

Several Haskell packages provide complementary tools for algebraic and combinatorial computations. For example, `vector-space`[4] offers generic abstractions for vector spaces and linear operations, `algebraic-graphs`[5] supports compositional construction and manipulation of graphs, and `free-algebras`[6] enables symbolic computation with generic algebraic structures. While powerful in their respective domains, these libraries are either too general to capture the full combinatorial and algebraic structure of trees and forests, or they are not tailored for symbolic computations of operations such as grafting, Grossman–Larson products, or the Connes–Kreimer coproduct. Arboretum.hs addresses this gap by providing a practical, type-safe framework that serves as a flexible platform for research in algebraic combinatorics, rough paths, regularity structures, and numerical analysis.

Most of the work on the `Arboretum.hs` package was carried out during the first author's PhD studies, which focused on the analysis of numerical integrators using algebraic and combinatorial tools, and a discussion of the package is included in first author's PhD thesis [3].

The paper is structured in the following way. Section 1 contains the instructions on how to install and use the package. Section 2 introduces Butcher series, which are the main motivation for the develoment of the package and an important tool in the analysis of numerical integrators. Section 3 presents the necessary tools to represent formal sums of trees and forests and Section 4 describes the implementation of the algebraic and coalgebraic structures over trees and forests.

1 Getting started

The reader is encouraged to explore the examples provided in this paper for a practical understanding of the package's functionality. To begin using `Arboretum.hs`, install the Haskell Tool Stack (Stack) by following the instructions from

`https://haskellstack.org`,

clone the git repository

[1] https://github.com/ranocha/BSeries.jl, [40]
[2] https://github.com/ketch/BSeries, [41]
[3] https://github.com/henriksu/pybs, [44]
[4] https://hackage.haskell.org/package/vector-space
[5] https://hackage.haskell.org/package/algebraic-graphs, [33]
[6] https://hackage.haskell.org/package/free-algebras

https://gitlab.unige.ch/Eugen.Bronasco/arboretum.hs

open a terminal in the root folder and run the command `stack repl`. The initial release version of the package is also availabe at [5]. The package uses a custom extension of the LaTeX package `planarforest` to draw the trees and forests. The original `planarforest` package is available at

https://hmarthinsen.github.io/planarforest/

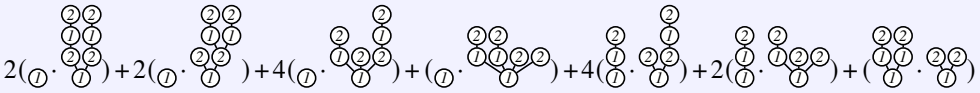
The `display` function is used to display vectors of trees and forests. It generates a PDF file in the root folder called `output.pdf` which contains the visual representation of the vector and attemps to open the file using the default document viewer. The `display` function is used in the examples below to illustrate the results of the computations.

USAGE EXAMPLE 1. *Below you can find some examples of the package usage where we define two forests* `f1` *and* `f2`*, and graft* `f1` *onto* `f2`*. The details can be found in Section 4.3.*

```
>>> f1 = fromBrackets "1[2],1[2]" :: MultiSet (Tree Integer)
>>> display f1
```

```
>>> f2 = fromBrackets "1,1[2,2]" :: MultiSet (Tree Integer)
>>> display f2
```

```
>>> display $ graft f1 f2
```



2 Butcher series

In this section, we introduce Butcher series, introduced in [25] and named B-series to honor the seminal algebraic work of Butcher [11, 12], see also [13, 23]. They are a central tool in the convergence analysis of numerical integrators and in the study of their geometric properties. Butcher series form a rich mathematical framework that connects applied mathematics, in particular numerical analysis, with areas of pure mathematics such as combinatorics, combinatorial algebra, and geometry. Their formulation relies heavily on rooted non-planar trees and forests. Butcher series and their many extensions have been used in the analysis of numerical integrators for Hamiltonian systems on $\mathbb{R}^d$ often used in mechanics, (stochastic) ordinary differential equations on $\mathbb{R}^d$ and on manifolds as well as in the theory of (stochastic) partitioned differential equations. See [8, 9, 17, 19, 22, 23, 38].

In this paper, we restrict our attention to the classical theory of Butcher series which is developed in the context of ordinary differential equations on $\mathbb{R}^d$. Let us consider the following ordinary differential equation which describes the change of a position vector

$y(t) \in \mathbb{R}^d$ in time:

$$\frac{dy(t)}{dt} = f(y(t)), \quad \text{with } y(0) = y_0 \in \mathbb{R}^d, \tag{1}$$

for some given vector $y_0 \in \mathbb{R}^d$ where $f : \mathbb{R}^d \to \mathbb{R}^d$ is a smooth vector field which assigns to every position in $\mathbb{R}^d$ a vector whose direction and magnitude describe the direction and speed of the movement of $y(t)$ as time passes. A solution of (1) is a function $y(t)$ such that the equation (1) is satisfied. It is often the case that an explicit formula for the solution $y(t)$ cannot be found, and, therefore, the only possible way to compute $y(t)$ is by approximating its value using a numerical integrator for $t \in [0, T]$ for some final time $T \in \mathbb{R}$.

Let y_n be an approximation of the solution $y(t_n)$ at time $t_n = nh$, where $h > 0$ is the timestep size. Runge-Kutta methods are a widely used class of numerical integrators which include

$$\begin{aligned}
\text{Forward Euler:} \quad & y_{n+1} = y_n + hf(y_n), \\
\text{Backward Euler:} \quad & y_{n+1} = y_n + hf(y_{n+1}), \\
\text{Implicit midpoint:} \quad & y_{n+1} = y_n + hf\big(\frac{1}{2}(y_n + y_{n+1})\big), \\
\text{Trapezoidal method:} \quad & y_{n+1} = y_n + \frac{h}{2}f(y_n) + \frac{h}{2}f(y_{n+1}),
\end{aligned}$$

as well as the commonly used RK4 method. The idea behind these methods is to compute successive approximations $y_0, y_1, \ldots, y_N$ of the solution at times $t_0, t_1, \ldots, t_N$ with $N = T/h$.

One of the key characteristics of a numerical integrator is its *order of convergence* where a numerical integrator $y_1 := \Phi_h(y_0)$ has order p if

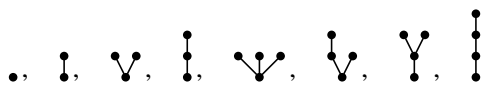
$$|y(T) - \Phi_h^N(y_0)| \approx O(h^p).$$

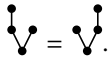
Order of convergence indicates how quickly the numerical solution converges to the exact solution as the timestep size h decreases. For example, forward and backward Euler methods have order 1, implicit midpoint and trapezoidal have order 2, and the RK4 method has order 4.

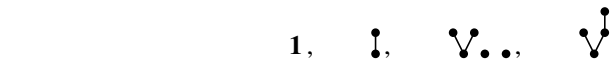
The order of convergence of Runge–Kutta methods is analyzed using Butcher series, whose central idea is to represent the Taylor expansions of both the exact and numerical solutions as formal sums indexed by rooted, non-planar trees. A commutative monomial $\tau_1 \cdots \tau_n$ for some $n \in \mathbb{N}$ is a product of trees $\tau_1, \ldots, \tau_n$ in which the order of trees does not matter.

Definition 2. *A* rooted non-planar tree τ *is a tuple* (r, π) *of a vertex* r *called the* root *of* τ *and a commutative monomial of rooted non-planar trees* $\pi = \tau_1 \cdots \tau_n$ *called the* branches *of* τ.

The set of rooted non-planar trees is denoted by T and the commutative monomials of the rooted non-planar trees are called *rooted non-planar forests* and form the set F. We note that all trees and forests are assumed to be rooted and non-planar and, therefore, we omit writting it from now on. All trees with up to 4 vertices can be found below with roots displayed at the bottom,



We note that the order of branches does not matter, that is,  = . Some examples of forests can be found below,



where **1** denotes the empty forest. We note that $T \subset F$ and that the order of trees does not matter, that is,  = .

The sets T and F are isomorphic through the map $B_v^+ : F \to T$ which constructs a tree $B_v^+(\pi)$ with root v and branches π, for example,

$$B_\bullet^+(\mathbf{1}) = \bullet, \quad B_\bullet^+(\bullet) = \text{}, \quad B_\bullet^+(\bullet\ \bullet) = \text{  }.$$

[15] describes a correspondence between trees and *elementary differentials* which are the terms appearing in Taylor expansions of the exact and numerical solutions. This correspondence is defined in Definition 3.

Definition 3. *Let $\mathbb{F}$ denote the corresondence between trees and elementary differentials defined as*

$$\mathbb{F}(B_\bullet^+(\tau_1 \cdots \tau_n)) := h \sum_{i_1,\dots,i_n=1}^{d} \mathbb{F}(\tau_1)^{i_1} \cdots \mathbb{F}(\tau_n)^{i_n} \partial_{i_1,\dots,i_n} f\,,$$

where v^i denotes i^{th} component of a vector v, ∂_i denotes the partial derivative with respect to the i^{th} argument, and $\partial_{i_1,\dots,i_n} := \partial_{i_1} \cdots \partial_{i_n}$.

For example,

$$\mathbb{F}(\bullet) = hf\,, \quad \mathbb{F}(\text{}) = h^2 \sum_{i=1}^{d} f^i \partial_i f\,, \quad \mathbb{F}(\text{}) = h^4 \sum_{i,j,k=1}^{d} f^i (\partial_i f^j) f^k (\partial_{j,k} f)\,.$$

Using an alternative notation, the values of $\mathbb{F}$ can be written as

$$\mathbb{F}(B_\bullet^+(\tau_1 \cdots \tau_n)) = hf^{(n)}\big(\mathbb{F}(\tau_1), \dots, \mathbb{F}(\tau_n)\big)\,,$$

which is the directional derivative of hf in the directions $\mathbb{F}(\tau_1), \dots, \mathbb{F}(\tau_n)$.

Taylor expansion of both exact and numerical solutions $y(h)$ and $y_1 = \Phi_h(y_0)$ given by Runge-Kutta methods can be expressed using Butcher series denoted by $B(1/\gamma, y_0)$ and $B(a, y_0)$ for sufficiently small timestep h,

$$y(h) = y_0 + \sum_{\tau \in T} \frac{1}{\gamma(\tau)\sigma(\tau)} \mathbb{F}(\tau)(y_0) =: y_0 + B(1/\gamma, y_0)\,, \tag{2}$$

$$\Phi_h(y_0) = y_0 + \sum_{\tau \in T} \frac{a(\tau)}{\sigma(\tau)} \mathbb{F}(\tau)(y_0) =: y_0 + B(a, y_0)\,, \tag{3}$$

where the coefficient map $\sigma : T \to \mathbb{R}$ denotes the symmetry of a tree and is defined as

$$\sigma\big(B_v^+(\tau_1^{k_1} \cdots \tau_n^{k_n})\big) := \prod_{i=1}^{n} k_i! \sigma(\tau_i)\,, \tag{4}$$

where $\tau_1, \dots, \tau_n$ are distinct and $k_1, \dots, k_n$ are the multiplicities of the corresponding trees while the coefficient map $\gamma : T \to \mathbb{R}$ denotes the factorial of a tree and is defined as

$$\gamma(\tau) = \gamma\big(B_v^+(\tau_1 \cdots \tau_n)\big) = |\tau|\Big(\prod_{i=1}^{n} \gamma(\tau_1)\Big)\,,$$

where $|\tau|$ denotes the number of vertices of the tree τ,

$$|B_v^+(\tau_1 \cdots \tau_n)| := 1 + \sum_{i=1}^{n} |\tau_i| \,.$$

Coefficient map $a : T \to \mathbb{R}$ depends on the particular choice of the Runge-Kutta method, see [24] for details. Values of the coefficient maps $\sigma, \gamma : T \to \mathbb{R}$ for all trees up to size 4 can be found in Table 1

Tree τ								
$\sigma(\tau)$	1	1	2	1	6	2	2	1
$\gamma(\tau)$	1	2	3	6	4	8	12	24

Table 1. Symmetry $\sigma(\tau)$ and the factorial $\gamma(\tau)$ for all trees τ of with up to 4 vertices.

The description of the exact and numerical solutions (2) and (3) in terms of Butcher series gives a straightforward way to derive order conditions for Runge-Kutta methods. Indeed, a Runge-Kutta method has order p if and only if the corresponding coefficient map $a : T \to \mathbb{R}$ satisfies $a(\tau) = 1/\gamma(\tau)$ for all trees τ of size $|\tau| \leq p$. This is a major milestone in the study of Runge-Kutta methods obtained in [11] and is the original motivation for the introduction of Butcher series.

2.1 Algebras of forests and composition of Butcher series

Let $y_1 = \Phi_h(y_0)$ and $y_1 = \Psi_h(y_0)$ be two numerical integrators that can be Taylor expanded as Butcher series $B(a, y_0)$ and $B(b, y_0)$, respectively. We compose the integrators Φ_h and Ψ_h by first applying Φ_h to y_0 and then applying Ψ_h to the result, that is, we compute $\Psi_h\big(\Phi_h(y_0)\big)$. In this section, we describe the Butcher series corresponding to the composition,

$$\Psi_h\big(\Phi_h(y_0)\big) = B(b, B(a, y_0)) = B(a * b, y_0) \,,$$

where $a * b : T \to \mathbb{R}$ defines a new coefficient map which is described using the algebraic structures introduced in this section, which include: grafting algebras over trees and forests, symmetric algebra over forests, and combinatorial Hopf algebras of Grossman-Larson and Connes-Kreimer. Implementations of these algebras are discussed in Section 4.

Elementary differentials are vector fields $g : \mathbb{R}^d \to \mathbb{R}^d$ that appear in the Taylor expansion of the exact solution of (1). For each elementary differential $g : \mathbb{R}^d \to \mathbb{R}^d$, there exists a tree $\tau \in T$ such that $\mathbb{F}(\tau) = g$, see Definition 3. The set of elementary differentials is denoted by $\mathbb{F}(T) := \{\mathbb{F}(\tau) \mid \tau \in T\}$. Let $\mathcal{T}$ denote the vector space with the set of trees T as basis, then let $\mathbb{F}(\mathcal{T})$ be the vector space with the set of vector fields $\mathbb{F}(T)$ as basis.

Definition 4. *A* vector space *is a set V and a field of scalars k with two operations: addition $+$ and scalar multiplication $\cdot$ that satisfy the following properties:*

(1) *V is an abelian group under addition,*
(2) *for all $a, b \in V$ and $\lambda \in k$, we have $\lambda \cdot (a + b) = \lambda \cdot a + \lambda \cdot b$,*
(3) *for all $a \in V$ and $\lambda, \mu \in k$, we have $(\lambda + \mu) \cdot a = \lambda \cdot a + \mu \cdot a$ and $(\lambda\mu) \cdot a = \lambda \cdot (\mu \cdot a)$,*
(4) *for all $a \in V$, we have $1 \cdot a = a$.*

A set $\{v_i\}_{i=1}^d$, $v_i \in V$, *is called a* basis *if for each* $v \in V$ *there exists a unique set of coefficients* $\{c_i\}_{i=1}^d$ *with* $c_i \in k$ *such that*

$$v = c_1 v_1 + \cdots + c_d v_d \,.$$

Vector space V *has* dimension d.

The map $\mathbb{F}$ between trees and elementary differentials (Definition 3), relates Butcher series and formal sums of trees,

$$\sum_{\tau\in T} \frac{a(\tau)}{\sigma(\tau)}\tau \quad \overset{\mathbb{F}}{\longmapsto} \quad \sum_{\tau\in T} \frac{a(\tau)}{\sigma(\tau)}\mathbb{F}(\tau) = B(a, y_0)\,.$$

This allows us to replace elementary differentials by the concise and intuitive formalism of trees. This powerful formalism allows us to use the techniques from combinatorics and combinatorial algebra to study Butcher series. We denote the vector space of formal sums of trees by $\overline{T}$.

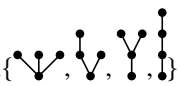
Definition 5. *A vector space* V *is* graded *if* $V = \bigoplus_{n\in\mathbb{N}} V_n$, *that is, each* $v \in V$ *can be decomposed uniquely as*

$$v = \sum_{n\in\mathbb{N}} v_n\,, \quad v_n \in V_n\,,$$

where each $V_n \subset V$ *is a vector subspace and elements of* V_n *are called* homogeneous *elements of degree* n.

The vector space of trees $\mathcal{T}$ and of formal sums of trees $\overline{\mathcal{T}}$ are graded vector spaces with the grading given by the number of vertices. For example,

$\overline{\mathcal{T}}_1 = \mathcal{T}_1 := \mathrm{span}\{\bullet\}$, $\overline{\mathcal{T}}_2 = \mathcal{T}_2 := \mathrm{span}\{$ $\}$,

$\overline{\mathcal{T}}_3 = \mathcal{T}_3 := \mathrm{span}\{$  $\}$, $\overline{\mathcal{T}}_4 = \mathcal{T}_4 := \mathrm{span}\{$  $\}$.

We note that each $\overline{\mathcal{T}}_n$ for $n \in \mathbb{N}$ is finite-dimensional. The implementation of the infinite-dimensional graded vector spaces whose homogeneous subspaces are finite-dimensional is discussed in Section 3.

2.1.1 Grafting algebra of trees

Let $\{e_i\}_{i=1}^d$ be the canonical basis of $\mathbb{R}^d$, that is,

$$e_1 = (1, 0, \ldots, 0)^T\,, \quad e_2 = (0, 1, 0, \ldots, 0)^T\,, \quad \ldots\,, \quad e_d = (0, \ldots, 0, 1)^T\,.$$

Then, a vector field $g : \mathbb{R}^d \to \mathbb{R}^d$ can be written as $g(x) = \sum_{i=1}^d g^i(x) e_i$ for $g^i : \mathbb{R}^d \to \mathbb{R}$. We identify the space of vector fields with the space of first-order differential operators by identifying $\{e_i\}_{i=1}^d$ with $\{\partial_i\}_{i=1}^d$. Given a first-order differential operator $g : \mathbb{R}^d \to \mathbb{R}^d$, its application to $h : \mathbb{R}^d \to \mathbb{R}^d$ is given by

$$g[h](x) = \sum_{i=1}^d g^i(x)(\partial_i h)(x) = \sum_{i,j=1}^d g^i(x)(\partial_i h^j)(x)\partial_j\,.$$

We omit writting x from now on for conciseness. The space of first-order differential operators $\mathbb{F}(\mathcal{T})$ forms a pre-Lie algebra $(\mathbb{F}(\mathcal{T}), -[-])$, that is, for any $g_1, g_2, g_3 \in \mathbb{F}(\mathcal{T})$, we have the following relation,

$$g_1[g_2[g_3]] - g_1[g_2][g_3] = g_2[g_1[g_3]] - g_2[g_1][g_3]\,.$$

Let us define the grafting product $\curvearrowright\colon \mathcal{T} \otimes \mathcal{T} \to \mathcal{T}$ such that

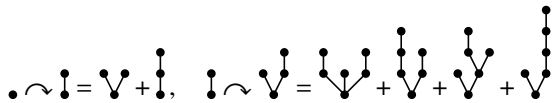
$$\mathbb{F}(\tau \curvearrowright \gamma) = \mathbb{F}(\tau)[\mathbb{F}(\gamma)]\,.$$

Definition 6. *Let $\tau, \gamma \in T$ and let $V(\gamma)$ be the set of vertices of γ. The* grafting product *is defined as*

$$\tau \curvearrowright \gamma := \sum_{v \in V(\gamma)} \tau \curvearrowright_v \gamma\,,$$

where $\tau \curvearrowright_v \gamma$ is the tree obtained by attaching the root of τ to the vertex v of γ.

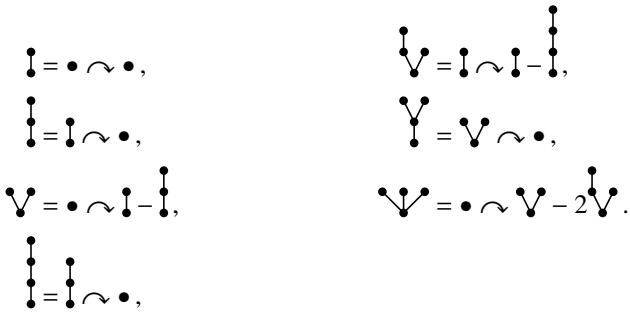
For example,



The algebra $(\mathcal{T}, \curvearrowright)$ is the free pre-Lie algebra with one generator $\{\bullet\}$ as is shown in [16], that is, the relation

$$\tau_1 \curvearrowright (\tau_2 \curvearrowright \tau_3) - (\tau_1 \curvearrowright \tau_2) \curvearrowright \tau_3 = \tau_2 \curvearrowright (\tau_1 \curvearrowright \tau_3) - (\tau_2 \curvearrowright \tau_1) \curvearrowright \tau_3\,, \quad \tau_1, \tau_2, \tau_3 \in \mathcal{T}\,,$$

is the only relation satisfied by the product $\curvearrowright\colon \mathcal{T} \otimes \mathcal{T} \to \mathcal{T}$ and each $\tau \in \mathcal{T}$ can be constructed using only the generator $\bullet$ and the grafting product $\curvearrowright$. For example,



The implementation of $(\mathcal{T}, \curvearrowright)$ is discussed in Section 4.3. The grafting product $\curvearrowright$ over $\mathcal{T}$ is extended to $\overline{\mathcal{T}}$. Moreover, it respects the grading, that is,

$$\mathcal{T}_n \curvearrowright \mathcal{T}_m \subset \mathcal{T}_{n+m}\,.$$

Such products are called *graded* and the algebras they form are called *graded algebras*.

We note that using the chain rule of differentiation and Definition 3 of $\mathbb{F}$, we can express the n^{th} derivative of the exact solution $y(t)$ of (1) at $t = 0$ scaled by h^n as

$$h^n y^{(n)}(0) = \mathbb{F}\Big(\underbrace{\bullet \curvearrowright (\bullet \curvearrowright \cdots (\bullet \curvearrowright \bullet) \cdots)}_{n \text{ times}}\Big)(y_0) = \mathbb{F}\big(\bullet^{\curvearrowright n}\big)(y_0)\,,$$

where $\bullet^{\curvearrowright n}$ is the grafting product of $\bullet$ taken n times assuming right associativity. This allows us to express the Taylor expansion of the exact solution of (1) as

$$y(h) = y_0 + \mathbb{F}\Big(\sum_{n=1} \frac{1}{n!}\, \bullet^{\curvearrowright n}\Big)(y_0) = \mathbb{F}\big(\exp^{\curvearrowright}(\bullet)\big)(y_0)\,,$$

where $\exp^{\curvearrowright}$ denotes the exponential with respect to the grafting product. We obtain the Butcher series representation of the exact solution (2) by grouping the terms.

2.1.2 *Symmetric algebra of forests*

In this subsection, we construct the algebra of high-order differential operators which extends the algebra $(\mathbb{F}(\mathcal{T}), -[-])$ of first-order differential operators. Let V be a vector space spanned by $\{v_i\}_{i\in I}$ for a countable set of indices I and let $S(V)$ be the vector space spanned by commutative monomials $v_1 \cdots v_n$ for $n \in \mathbb{N}$ where the order of $v_1, \dots, v_n$ does not matter. The space $S(V)$ is a graded algebra with the concatenation product,

$$(v_1 \cdots v_n) \cdot (u_1 \cdots u_m) = v_1 \cdots v_n u_1 \cdots u_m \,,$$

and the empty monomial $\mathbf{1}$ as the unit. The grading is given by the number of the elements of V in a monomial, for example, the grading of $v_1 \cdots v_n$ is n. Let us define the deshuffle coproduct $\Delta : S(V) \to S(V) \otimes S(V)$ over the vector space $S(V)$.

Definition 7. *Let the* deshuffle coproduct $\Delta : S(V) \to S(V) \otimes S(V)$ *be defined as*

$$\begin{aligned} \Delta(\mathbf{1}) &= \mathbf{1} \otimes \mathbf{1} \,, \\ \Delta(v) &= v \otimes \mathbf{1} + \mathbf{1} \otimes v \,, \quad v \in V \,, \\ \Delta(a \cdot b) &= \Delta(a) \cdot \Delta(b) \,, \quad a \in S(V) \,, \end{aligned}$$

where $(a_1 \otimes b_1) \cdot (a_2 \otimes b_2) = (a_1 \cdot a_2) \otimes (b_1 \cdot b_2)$ *for* $a_1, a_2, b_1, b_2 \in S(V)$.

Intuitively, the deshuffle coproduct sums over all ways to place the elements of V on the left or the right side of the tensor product while preserving their order. For example,

$$\begin{aligned} \Delta(v_1 v_2 v_3) = {} & v_1 v_2 v_3 \otimes \mathbf{1} + v_2 v_3 \otimes v_1 + v_1 v_3 \otimes v_2 + v_1 v_2 \otimes v_3 \\ & + v_3 \otimes v_1 v_2 + v_2 \otimes v_1 v_3 + v_1 \otimes v_2 v_3 + \mathbf{1} \otimes v_1 v_2 v_3 \,. \end{aligned}$$

The space $S(V)$ is a coalgebra with the deshuffle coproduct $\Delta : S(V) \to S(V) \otimes S(V)$ and counit $\delta_{\mathbf{1}} : S(V) \to \mathbb{R}$ defined as

$$\delta_{\mathbf{1}}(a) := \begin{cases} 1 \,, & a = \mathbf{1} \,, \\ 0 \,, & a \neq \mathbf{1} \,. \end{cases}$$

The deshuffle coalgebra is graded, that is,

$$\Delta(S(V)_n) \subset \sum_{i=0}^{n} S(V)_i \otimes S(V)_{n-i} \,, \quad \delta_{\mathbf{1}}(S(V)_{n\geq 1}) = \{0\} \,.$$

The space $S(V)$ is a combinatorial Hopf algebra $(S(V), \mathbf{1}, \cdot, \delta_{\mathbf{1}}, \Delta)$ called the *symmetric algebra*, see Definition 8. The implementation of the symmetric algebra $S(V)$ is discussed in Section 4.1.

Definition 8. *A graded vector space* V *with* $\dim(V_0) = 1$ *endowed with a graded algebra* $(V, \cdot)$ *with a unit* $\mathbf{1}$ *and coalgebra* (V, Δ) *with counit* $\epsilon : V \to \mathbb{R}$ *such that*

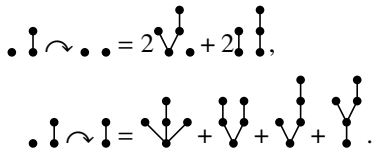
(1) $\epsilon(\mathbf{1}) = 1$,
(2) $\Delta(\mathbf{1}) = \mathbf{1} \otimes \mathbf{1}$,
(3) $\epsilon(a \cdot b) = \epsilon(a) \cdot \epsilon(b)$,
(4) $\Delta(a \cdot b) = \Delta(a) \cdot \Delta(b)$,

is referred to as a combinatorial Hopf algebra $(V, \mathbf{1}, \cdot, \epsilon, \Delta)$. *We recall that* $(a_1 \otimes b_1) \cdot (a_2 \otimes b_2) = (a_1 \cdot a_2) \otimes (b_1 \cdot b_2)$.

Given a pre-Lie product on a vector space V, [39] extend the pre-Lie product to the symmetric algebra $S(V)$. In the case of $V = \mathcal{T}$, this corresponds to the extention of the grafting product $\curvearrowright$ to the space of forests $S(\mathcal{T}) = \mathcal{F}$ spanned by the set of forests F. The grafting product $\curvearrowright\colon \mathcal{F} \otimes \mathcal{F} \to \mathcal{F}$ is defined, for $\tau \in \mathcal{T}$ and $\pi, \eta, \mu \in \mathcal{F}$, as

$$\begin{aligned} \mathbf{1} \curvearrowright \mathbf{1} &= \mathbf{1}, \quad \tau \curvearrowright \mathbf{1} = 0, \\ (\tau \cdot \pi) \curvearrowright \eta &= \tau \curvearrowright (\pi \curvearrowright \eta) - (\tau \curvearrowright \pi) \curvearrowright \eta, \\ \pi \curvearrowright (\eta \cdot \mu) &= \sum_{(\pi)} (\pi_{(1)} \curvearrowright \eta) \cdot (\pi_{(2)} \curvearrowright \mu), \end{aligned} \tag{5}$$

where we use the notation $\Delta(\pi) = \sum_{(\pi)} \pi_{(1)} \otimes \pi_{(2)}$ for the deshuffle coproduct of π, see Definition 7. For example,



Intuitively, the grafting product $\curvearrowright\colon \mathcal{F} \otimes \mathcal{F} \to \mathcal{F}$ sums over all ways to graft each tree of the forest on the left to each vertex of the forest on the right.

The algebra of high-order differential operators $(\mathbb{F}(\mathcal{F}), -[-])$ where $\mathbb{F}(\mathcal{F})$ has $\mathbb{F}(F) := \{\mathbb{F}(\pi) \mid \pi \in F\}$ as basis satisfies

$$\mathbb{F}(\pi \curvearrowright \mu) = \mathbb{F}(\pi)[\mathbb{F}(\mu)] .$$

We note that $\mathbb{F}(B^+(\pi)) = \mathbb{F}(\pi \curvearrowright \bullet) = h\mathbb{F}(\pi)[f]$ where $\pi \in F$. This gives an alternative and purely algebraic definition to the map $\mathbb{F}$ which can be seen as the unique map which maps the generator $\bullet$ to hf and the grafting product $\curvearrowright$ to $-[-]$.

The grafting product $\curvearrowright\colon \mathcal{F} \otimes \mathcal{F} \to \mathcal{F}$ arises naturally in the Taylor expansion of $\mathbb{F}(\tau)(y_0 + \mathbb{F}(\gamma))$ for $\tau, \gamma \in \mathcal{T}$, in particular,

$$\mathbb{F}(\tau)(y_0 + \mathbb{F}(\gamma)) = \mathbb{F}\big(\exp^{\cdot}(\gamma) \curvearrowright \tau\big)(y_0), \tag{6}$$

where $\exp^{\cdot}$ denotes the exponential with respect to the concatenation product. The property (6) is an essential step in the expansion of Runge-Kutta methods as Butcher series (3).

2.1.3 *Grossman-Larson and Connes-Kreimer Hopf algebras of forests*

The grafting algebra $(\mathcal{F}, \curvearrowright)$ is used to study the algebra of differential operators $(\mathbb{F}(\mathcal{F}), -[-])$ where the product $\mathbb{F}(\pi)[\mathbb{F}(\eta)]$ is the differentiation of the differential operator $\mathbb{F}(\eta)$ by the differential operator $\mathbb{F}(\pi)$.

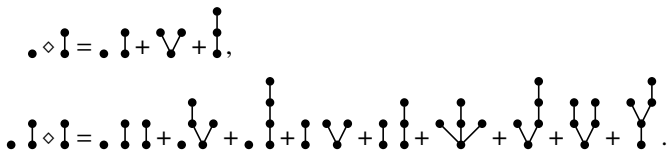
Another natural algebraic structure on $\mathbb{F}(\mathcal{F})$ consists in the composition of differential operators, that is,

$$\mathbb{F}(\pi)\big[\mathbb{F}(\eta)[\phi]\big] = \big(\mathbb{F}(\pi) \circ \mathbb{F}(\eta)\big)[\phi], \tag{7}$$

where ϕ can be a function $\mathbb{R}^d \to \mathbb{R}$, a vector field $\mathbb{R}^d \to \mathbb{R}^d$, or a differential operator. This defines an algebra $(\mathbb{F}(\mathcal{F}), \circ)$ of differential operators. [39] introduce the corresponding algebra over $\mathcal{F}$ which is called the Grossman-Larson algebra with the Grossman-Larson product defined, for $\pi, \eta \in \mathcal{F}$, as

$$\pi \diamond \eta := \sum_{(\pi)} \pi_{(1)} \cdot (\pi_{(2)} \curvearrowright \eta) .$$

Some examples of performing the Grossman-Larson can be found below,



Intuitively, the Grossman-Larson product $\diamond : \mathcal{F} \otimes \mathcal{F} \to \mathcal{F}$ sums over all ways to split the trees of the forest on the left into two groups: the first group is concatenated to the forest on the right, and the second is grafted onto the forest on the right. Grossman-Larson product satisfies, for $\pi, \eta, \mu \in \mathcal{F}$,

$$\pi \curvearrowright (\eta \curvearrowright \mu) = (\pi \diamond \eta) \curvearrowright \mu \,,$$

which implies $\mathbb{F}(\pi \diamond \eta) = \mathbb{F}(\pi) \circ \mathbb{F}(\eta)$ using (7) and gives an alternative definition of the Grossman-Larson product,

$$\pi \diamond \eta = B^-_\bullet \big(\pi \curvearrowright B^+_\bullet(\eta)\big) \,, \quad \text{for } \pi, \eta \in \mathcal{F} \,,$$

where $B^-_\bullet$ removes the root v. Recall that $B^+_\bullet(\eta) = \eta \curvearrowright \bullet$. Grossman-Larson algebra has the empty forsest as the unit and, together with the deshuffle coalgebra, forms the Grossman-Larson combinatorial Hopf algebra $(\mathcal{F}, \diamond, \mathbf{1}, \Delta, \delta_{\mathbf{1}})$ graded by the number of vertices. The implementation of the Grossman-Larson Hopf algebra is discussed in Section 4.3.

Let us define the Connes-Kreimer coproduct $\Delta_{CK} : \mathcal{F} \to \mathcal{F} \otimes \mathcal{F}$.

Definition 9. Connes-Kreimer coproduct $\Delta_{CK} : \mathcal{F} \to \mathcal{F} \otimes \mathcal{F}$ *is defined, for* $\pi, \eta \in F$, *as*

$$\Delta_{CK}(B^+_v(\pi)) := B^+_v(\pi) \otimes \mathbf{1} + \sum_{(\pi)} \pi_{(1)} \otimes B^+_v(\pi_{(2)}) \,,$$
$$\Delta_{CK}(\pi \cdot \eta) := \Delta_{CK}(\pi) \cdot \Delta_{CK}(\eta) \,,$$

and $\Delta_{CK}(\mathbf{1}) = \mathbf{1} \otimes \mathbf{1}$ *where we use the notation* $\Delta_{CK}(\pi) = \sum_{(\pi)} \pi_{(1)} \otimes \pi_{(2)}$ *and*

$$(a_1 \otimes b_1) \cdot (a_2 \otimes b_2) = (a_1 \cdot a_2) \otimes (b_1 \cdot b_2) \,.$$

For example,

$$\Delta_{CK}(\bullet) = \bullet \otimes \mathbf{1} + \mathbf{1} \otimes \bullet \,,$$
$$\Delta_{CK}(\bullet\!\!\bullet) = \bullet\!\!\bullet \otimes \mathbf{1} + \bullet \otimes \bullet + \mathbf{1} \otimes \bullet\!\!\bullet \,,$$
$$\Delta_{CK}(\bullet\!\!\bullet\!\!\bullet) = \bullet\!\!\bullet\!\!\bullet \otimes \mathbf{1} + \bullet\!\!\bullet \otimes \bullet + \bullet \otimes \bullet\!\!\bullet + \mathbf{1} \otimes \bullet\!\!\bullet\!\!\bullet \,,$$
$$\Delta_{CK}(\bullet\!\!\bullet\!\!\bullet\!\!\bullet) = \bullet\!\!\bullet\!\!\bullet\!\!\bullet \otimes \mathbf{1} + \bullet \bullet \otimes \bullet + 2\, \bullet \otimes \bullet\!\!\bullet + \mathbf{1} \otimes \bullet\!\!\bullet\!\!\bullet \,,$$
$$\Delta_{CK}(\bullet\!\!\bullet\!\!\bullet\!\!\bullet\!\!\bullet) = \bullet\!\!\bullet\!\!\bullet\!\!\bullet\!\!\bullet \otimes \mathbf{1} + \bullet \bullet\!\!\bullet \otimes \bullet + \bullet\!\!\bullet \otimes \bullet\!\!\bullet + \bullet \bullet \otimes \bullet\!\!\bullet + \bullet \otimes \bullet\!\!\bullet\!\!\bullet + \bullet \otimes \bullet\!\!\bullet\!\!\bullet + \mathbf{1} \otimes \bullet\!\!\bullet\!\!\bullet\!\!\bullet \,.$$

The vector space $\mathcal{F}$ of forests together with the concatenation product, unit $\mathbf{1}$, and Connes-Kreimer coproduct with the counit $\delta_{\mathbf{1}}$, forms the Connes-Kreimer combinatorial Hopf algebra $(\mathcal{F}, \mathbf{1}, \cdot, \delta_{\mathbf{1}}, \Delta_{CK})$.

We consider the inner product $\langle -, - \rangle_\sigma : \mathcal{F} \otimes \mathcal{F} \to \mathbb{R}$ defined, for $\pi, \eta \in \mathcal{F}$ as

$$\langle \pi, \eta \rangle_\sigma := \begin{cases} \sigma(\pi) \,, & \text{if } \pi = \eta \,, \\ 0 \,, & \text{otherwise,} \end{cases}$$

where $\sigma(\pi)$ denotes the symmetry coefficient of the forest π, see (4). We rely on the following relations between the concatenation product and deshuffle coproduct, and Grossman-Larson product and Connes-Kreimer coproduct,

$$\langle \pi \cdot \eta, \mu \rangle_\sigma = \langle \pi \otimes \eta, \Delta(\mu) \rangle_\sigma \,, \quad \langle \pi \diamond \eta, \mu \rangle_\sigma = \langle \pi \otimes \eta, \Delta_{CK}(\mu) \rangle_\sigma \,, \tag{8}$$

for all $\pi, \eta, \mu \in \mathcal{F}$. In other words, concatenation and Grossman-Larson products are adjoints of the deshuffle and Connes-Kreimer coproducts, respectively, with respect to the inner product $\langle -, - \rangle_\sigma$.

Intuitively, this means that the coefficient of a forest $\mu \in F$ in $\pi \diamond \eta$ is equal to the coefficient of $\pi \otimes \eta$ in $\Delta_{CK}(\mu)$ multiplied by the symmetry ratio $\sigma(\pi)\sigma(\eta)/\sigma(\mu)$. For example, the coefficient of  in  is 1, which is equal to the coefficient of  in $\Delta_{CK}($  $)$ multiplied by the symmetry ratio $1/2$.

The relation between Grossman-Larson and Connes-Kreimer Hopf algebras together with the properties of the Grossman-Larson product, (6), (8), and the property

$$a(\pi) = \langle \sum_{\eta \in F} \frac{a(\eta)}{\sigma(\eta)} \eta, \pi \rangle_\sigma$$

are enough to prove that the composition of two numerical integrators $B(a, y_0)$ and $B(b, y_0)$ results in a numerical integrator $B(a * b, y_0)$,

$$B(b, B(a, y_0)) = B(a * b, y_0) \,, \quad \text{with } (a * b)(\tau) = \sum_{(\tau)} a(\tau_{(1)}) b(\tau_{(2)}) \,, \tag{9}$$

where $\Delta_{CK}(\tau) = \sum_{(\tau)} \tau_{(1)} \otimes \tau_{(2)}$. We note that both $a, b : T \to \mathbb{R}$ are coefficient maps defined over trees, while in (9) they are applied to forests $\tau_{(1)}, \tau_{(2)} \in F$, see the type signature of Δ_{CK}. We extend the coefficient maps a, b to forests by $a(\pi \cdot \eta) = a(\pi)a(\eta)$ and $a(\mathbf{1}) = b(\mathbf{1}) = 1$. Some values for the coefficient map $a * b$ can be found below,

$$\begin{aligned}
(a * b)(\bullet) &= a(\bullet) + b(\bullet) \,, \\
(a * b)(\bullet\!\!\bullet) &= a(\bullet\!\!\bullet) + a(\bullet)b(\bullet) + b(\bullet\!\!\bullet) \,, \\
(a * b)(\bullet\!\!\bullet\!\!\bullet) &= a(\bullet\!\!\bullet\!\!\bullet) + a(\bullet\!\!\bullet)b(\bullet) + a(\bullet)b(\bullet\!\!\bullet) + b(\bullet\!\!\bullet\!\!\bullet) \,, \\
(a * b)(\bullet\!\!\bullet\!\!\bullet\!\!\bullet) &= a(\bullet\!\!\bullet\!\!\bullet\!\!\bullet) + a(\bullet)^2 b(\bullet) + 2a(\bullet)b(\bullet\!\!\bullet) + b(\bullet\!\!\bullet\!\!\bullet\!\!\bullet) \,.
\end{aligned}$$

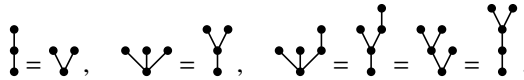
The implementation of the Connes-Kreimer Hopf algebra is discussed in Section 4.3.2.

2.2 Extensions of Butcher series

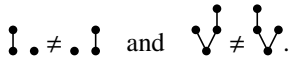
We have presented a concise introduction to the theory of classical Butcher series for the analysis of numerical integrators of ordinary differential equations. From a computational point of view, this is the simplest setting, involving only basic algebraic structures and rooted non-planar trees.

More refined questions in numerical analysis quickly lead to substantially richer variants of the Butcher series formalism. In particular, long time and geometric analysis of numerical integrators gives rise to the Calaque–Ebrahimi-Fard–Manchon Hopf algebra introduced in [14], whose coproduct poses serious computational challenges already in the simplest case that we consider here.

When Butcher series are applied to Hamiltonian systems found in mechanics, one is led to series indexed by non-rooted trees, where we forget which vertex is the root. See *Chapter IX.9.2* in [23] and [2]. For example,



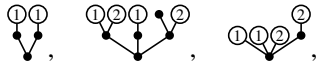
The extension of the Butcher series formalism to ordinary differential equations on manifolds leads to Lie-Butcher series and to Hopf algebras defined on planar trees and forests, where the order of branches in a tree and the order of trees in a forest matter. For instance,



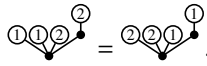
See [34] and [37].

In the stochastic setting, the theory of Butcher series introduces bicolored trees and forests, where one color represents noise, as well as exotic trees and forests in which vertices of a given color are paired. Examples of grafted trees include

Examples of exotic trees are



where the numbering of white vertices encodes the pairing, and the specific choice of labels is irrelevant. For example,

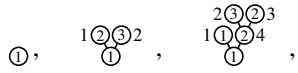


See [10, 29, 42].

Additional geometric constraints on numerical integrators, such as volume preservation, lead to the introduction of aromatic Butcher series in [18] and [27]. In this setting, trees and forests are replaced by aromatic trees and forests, which allow cycles. Some examples are

See also [1, 35].

Finally, the application of the Butcher series formalism to partitioned and stochastic partitioned differential equations leads to trees and forests with decorations on both vertices and edges. For example,



where the label outside a vertex decorates the edge under the vertex. See [9].

These extensions can also be combined, leading to highly complex combinatorial objects. As an illustration, [30] study exotic aromatic trees and forests with stolons (two horizontal edges connecting two roots). Even listing all such forests up to a small size quickly becomes infeasible by hand. For example, the complete list of all connected exotic aromatic forests with stolons of size up to 3 is already nontrivial and is found in Table 1.

In [7], this framework is further generalized by allowing white vertices to carry incoming edges. This modification leads to a dramatic increase in combinatorial complexity and,

consequently, in the number of order conditions: already at order 2, one obtains a system of 93 conditions. Fortunately, an alternative approach was developed that circumvents the need to work explicitly with these conditions.

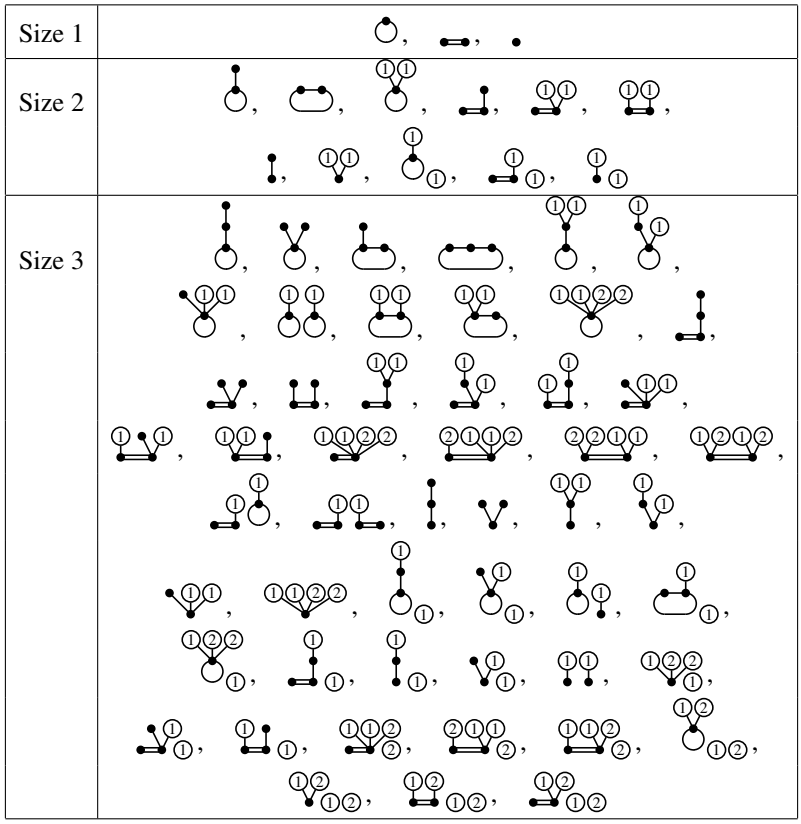


Fig. 1. All connected exotic aromatic forests with stolons up to size 3.

2.3 Recursive formulas suitable for implementation

We observe that Definition 6 of the grafting product on trees, together with its extension to forests given in (5), is not ideally suited for implementation. The grafting product on trees is formulated in a combinatorial manner, whereas a recursive definition would lead to a clearer structure and a more natural realization in Haskell. Furthermore, the extension to forests is computationally inefficient, as it involves the generation of intermediate terms in the computation of $(\tau \cdot \pi) \curvearrowright \eta$ that cancel out and do not contribute to the final result.

To address these issues, we introduce Definition 10, which presents a recursive definition of the grafting product on forests. This formulation is based on recursion via the Grossman–Larson product and avoids the unnecessary generation of intermediate terms, leading to a clearer and more efficient implementation.

Definition 10. *Let* grafting *of a forest and an empty forest be defined as,*

$$\mathbf{1} \curvearrowright \mathbf{1} = \mathbf{1}\,, \quad \pi \curvearrowright \mathbf{1} = 0\,, \quad \mathbf{1} \curvearrowright \pi = \pi\,, \quad \text{for } \pi \in F, \pi \neq \mathbf{1}\,.$$

Grafting of a forest on a tree is defined as,

$$\pi \curvearrowright B_r^+(\eta) = B_r^+(\pi \diamond \eta)\,, \quad \text{for } \pi, \eta \in F\,.$$

Grafting of a forest on a forest is defined as,

$$\pi \curvearrowright (\tau \cdot \eta) = \sum_{(\pi)} (\pi_{(1)} \curvearrowright \tau) \cdot (\pi_{(2)} \curvearrowright \eta), \quad \text{for } \tau \in T, \pi, \eta \in F\,.$$

where $\Delta(\pi) = \sum_{(\pi)} \pi_{(1)} \otimes \pi_{(2)}$ *denotes the deshuffle coproduct.*

3 Implementation of graded vector spaces

As the core objective of this package is to implement and manipulate algebras of graphs, we first define the fundamental structure on which these algebras are built: graded vector spaces. This section provides the necessary tools for the implementation of the grafting product on decorated forests discussed in Section 4. We start with Assumption 11.

Assumption 11. *We consider graded vector spaces* $V = \bigoplus_{n\in\mathbb{N}} V_n$ *(see Definitions 4 and 5) where all homogeneous subspaces* V_n *are finite-dimensional.*

Assumption 11 allows us to manipulate formal series as long as the manipulations act nicely with respect to the grading. A vector $v \in V$ can be decomposed as

$$v = \sum_{n\in\mathbb{N}} v_n\,, \quad \text{with } v_n \in V_n\,,$$

where V_n is the vector space of homogeneous elements with grading n. Each v_n is a linear combination of basis elements $\{b_i\}_{i=1}^{d_n}$ with $\dim(V_n) = d_n$, that is,

$$v_n = \sum_{i=1}^{d_n} c_i b_i\,, \quad \text{for some } c_i \in \mathbb{R}\,.$$

This representation of a vector $v \in V$ is implemented in Implementation 12.

Implementation 12. *We define the elements of a vector space below.*

```
data ScalarProduct k b = SP
    { scalar :: k
    , basisElement :: b
    }
data Sum k b = S Integer (ScalarProduct k b) (Sum k b) | Zero
data Vector k b = V (Sum k b) (Vector k b) | Empty
```

where

(1) `ScalarProduct k b` *is a pair of a scalar* `k` *and a basis element* `b`*,*
(2) `Sum k b` *is a recursively defined list (assumed finite) of scalar products with basis elements of the same grading which is stored in the first argument of* `S`*,*
(3) `Vector k b` *is a recursively defined list (possibly infinite) of sums.*

The basis, denoted as `b`, must be an instance of the `Eq` and `Graded` type classes. An instance of `Eq` ensures that we can determine whether two elements of type `b` are equal, which is a fundamental requirement for most types. To be an instance of the `Graded` type class, a type `b` must implement a function `grading :: b -> Integer`. The definition of the `Graded` type class is given below.

Implementation 13. *`Graded` type class.*

```
class Graded b where
    grading :: b -> Integer
```

Common types like `Integer` and `Char` have instances of the `Graded` class, where each integer or character is assigned a grading of 1. For composite structures such as lists, 2-tuples, and multisets, the grading is defined as the sum of the gradings of their elements.

The scalar field, `k`, is required to be an instance of both the `Num` and `Eq` type classes. An instance of `Num` enables arithmetic operations on elements of type `k`, and most numeric types provide this functionality.

3.1 Constructing vectors

Let us detail how the `ScalarProduct k b`, `Sum k b`, and `Vector k b` defined in Implementation 12 are constructed. A scalar product `ScalarProduct k b` and a homogeneous sum `Sum k b` can be constructed using binary operations `(*^)` and `(+:)` as can be seen in Usage Example 14.

Usage Example 14. *We use `:type` to confirm the types of the expressions.*

```
>>> :type 1 *^ 'x'
1 *^ 'x' :: Num k => ScalarProduct k Char
>>> :type 1*^'x' +: 2*^'y' +: Zero
1*^'x' +: 2*^'y' +: Zero
  :: (Eq k, Num k) => Sum k Char
```

A vector is constructed from data types that have an instance of the `IsVector` type class, which includes all types listed in Implementation 12. The definiton of the `IsVector` type class is shown below.

Implementation 15. *The `IsVector` type class leverages `TypeFamilies`, specifically `VectorScalar` and `VectorBasis`, which map the type `v` to its associated scalar and basis types.*

```
class
    ( Eq v
    , Eq (VectorBasis v)
    , Graded (VectorBasis v)
    , Eq (VectorScalar v)
    , Num (VectorScalar v)
    )
    => IsVector v where
    type VectorScalar v
    type VectorBasis v
    vector :: v -> Vector (VectorScalar v) (VectorBasis v)
```

We note that `IsVector` *is also a* constraint synonym *since it includes a list of common constraints in its definition.*

As an example, the instance of `IsVector` for the type `ScalarProduct k b` is presented below.

IMPLEMENTATION 16. *The instance for* `ScalarProduct k b` *converts a scalar product into a sum and then uses the instance for* `Sum k b` *to obtain the corresponding vector.*

```
instance (Num k, Eq k, Eq b, Graded b)
        => IsVector (ScalarProduct k b) where
    type VectorScalar (ScalarProduct k b) = k
    type VectorBasis (ScalarProduct k b) = b
    vector = vector . (+: Zero)
```

It is recommended to implement an instance of `IsVector` for each type `b` used as a basis in `Vector k b`. The function `vector` from the `IsVector` class can be used to construct finite vectors.

USAGE EXAMPLE 17. *Construct vectors from a scalar product and a sum:*

```
>>> :type vector (1 *^ 'x')
vector (1 *^ 'x')
  :: (Num k, Eq k) => Vector k Char
>>> :type vector (1*^'x' +: 2*^'y' +: Zero)
vector (1*^'x' +: 2*^'y' +: Zero)
  :: (Num k, Eq k) => Vector k Char
```

To construct a finite vector from a finite list of scalar products with varying gradings, use

```
vectorFromList :: [ScalarProduct k b] -> Vector k b.
```

For an infinite vector (formal series), constructed from a potentially infinite list of scalar products with non-decreasing gradings, use

```
vectorFromNonDecList : [ScalarProduct k b] -> Vector k b.
```

USAGE EXAMPLE 18. *Use of the* `vectorFromList` *function to construct a vector from a finite list of scalar products with varying gradings:*

```
>>> :type vectorFromList [3 *^ "x", 2 *^ "xy", 1 *^ "xyz"]
vectorFromList [3 *^ "x", 2 *^ "xy", 1 *^ "xyz"]
  :: (Num k, Eq k) => Vector k String
```

and of the `vectorFromNonDecList` *function to construct a vector from a possibly infinite list of scalar products with non-decreasing gradings:*

```
>>> :type vectorFromNonDecList [i *^ replicate i 'x' | i <- [1..]]
vectorFromNonDecList [i *^ replicate i 'x' | i <- [1..]]
  :: Vector Int [Char]
```

where `takeV n` *extracts the first n terms of a vector.*

3.2 Displaying vectors

The package includes the `display` function which generates a PDF file with LaTeX representation of `Vector k b`, see Implementation 19.

Implementation 19. *Implementation of* `display` *function*

```
display
    :: ( IsVector v
       , Texifiable (VectorScalar v)
       , Texifiable (VectorBasis v)
       )
    => v
    -> IO ()
display v = printPdf $ " $ " ++ texify (vector v) ++ " $ "
```

where `printPdf` *produces a PDF file with the LaTeX code generated by the* `texify` *function from the* `Texifiable` *class.*

The function `display` assumes the basis `b` in `Vector k b` has instance of the `Texifiable` class whose implementation is not detailed here. The `Texifiable` class provides a method `texify :: a -> String` that converts an element of type `a` into its corresponding LaTeX representation.

Usage Example 20. *Use of the* `display` *function*

```
>>> display $ vectorFromList
      [3 *^ "x", 2 *^ "xy", 1 *^ "xyz"]
```

$3x + 2(x \cdot y) + (x \cdot y \cdot z)$

```
>>> display $ takeV 5 $ vectorFromNonDecList
      [i *^ replicate i 'x' | i <- [1..]]
```

$x + 2(x \cdot x) + 3(x \cdot x \cdot x) + 4(x \cdot x \cdot x \cdot a) + 5(x \cdot x \cdot x \cdot x \cdot x)$
where `takeV n` *extracts the first n terms of a vector.*

3.3 Linear and bilinear functions

Defining linear maps on vector spaces is a common operation in linear algebra and it is often done by starting with a map defined on the basis of the vector space which is then extended linearly to the entire vector space. To extend a map `f :: b -> v` defined on the basis `b` of `Vector k b` to the entire vector space `Vector k b`, we use the `linear` function. It is important to note that `v` must be a type which has an instance of `IsVector`.

Usage Example 21. *Extending a map defined on the basis* `Char` *to a linear map on* `Vector k Char`*:*

```
>>> f x = case x of { 'x' -> 1 *^ 'a'; 'y' -> 2 *^ 'b'; 'z' -> 3 *^ 'c' }
>>> v = vector $ 1 *^ 'x' +: 2 *^ 'y' +: 3 *^ 'z' +: Zero
>>> display $ linear f v
```

$a + 4b + 9c$

Analogously, products over vector spaces can be defined by starting with a map on the basis elements and extending it bilinearly to the entire vector space. This is achieved using the `bilinear` function.

USAGE EXAMPLE 22. *We use bilinear below:*

```
>>> f x y = 1 *^ [x, y]
>>> v1 = vector
        $ 1 *^ 'x' +: 2 *^ 'y' +: 3 *^ 'z' +: Zero
>>> v2 = vector
        $ 5 *^ 'x' +: 7 *^ 'y' +: 11 *^ 'z' +: Zero
>>> display $ bilinear f v1 v2
```

$5(x \cdot x) + 10(y \cdot x) + 7(x \cdot y) + 15(z \cdot x) + 14(y \cdot y) + 11(x \cdot z) + 21(z \cdot y) + 22(y \cdot z) + 33(z \cdot z)$

4 Implementation of algebras of graphs

In this section, we endow the graded vector spaces whose implementation is discussed in Section 3 with algebraic structures introduced in Section 2. In particular, we discuss the symmetric algebra in Section 4.1 and the two different approaches in the implementation of algebras of graphs:

(1) Defining a general graph and subsequently restricting the type of graphs we accept by imposing additional structure,
(2) Building the specific graphs of interest directly from the ground up.

Each approach comes with its own set of advantages and drawbacks. The first approach is more flexible and can accommodate a broader variety of graphs, but it may introduce additional complexity. The second, more specialized approach, is often simpler to implement, although it can require more upfront work to define the specific graphs.

We illustrate the first approach by implementing the grafting algebra of graphs in Section 4.2. We then focus on the second approach and implement the grafting algebra of trees, forests, as well as the Grossman-Larson algebra in Section 4.3. We finish the section with the implementation of the Connes-Kreimer coalgebra.

4.1 Symmetric algebra

When the basis `b` of a vector space `Vector k b` forms a monoid, we can define a product on the vector space `Vector k b` by extending the product of the monoid to the entire space bilinearly. This turns `Vector k b` into an algebra with a `Num` instance which allows us to use familiar arithmetic notation for addition and the product. The symmetric algebra introduced in Section 2.1 is implemented as `Vector k (MultiSet b)`.

USAGE EXAMPLE 23. *We use the + and ∗ operations below for the symmetric algebra* `Vector k (MultiSet b)`.

```
>>> [x,y,z,a,b,c] = ["x", "y", "z", "a", "b", "c"]
>>> v1 = vector $ 1 *^ x +: 2 *^ y +: 3 *^ z +: Zero
>>> v2 = vector $ 5 *^ a +: 7 *^ b +: 11 *^ c +: Zero
>>> display $ v1 + v2
```

$x + 2y + 3z + 5a + 7b + 11c$

```
>>> display $ v1 * v2
```

$5(a \cdot x) + 10(a \cdot y) + 7(b \cdot x) + 15(a \cdot z) + 14(b \cdot y) + 11(c \cdot x) + 21(b \cdot z) + 22(c \cdot y) + 33(c \cdot z)$

We define the deshuffle coproduct by following the Definition 7.

IMPLEMENTATION 24. *Implementation of the deshuffle coproduct.*

```
deshuffle
    :: MultiSet b -> Vector Integer (MultiSet b, MultiSet b)
deshuffle =
    product . map (\b -> vector (empty, b') + vector (b', empty))
  where
    b' = singleton b
```

The algebra and coalgebra structure of `Vector k (MultiSet b)` form the symmetric algebra of the vector space `Vector k b`, see Section 2.1.2.

4.2 Grafting of graphs

In this section, we discuss the definition and implementation of graphs, along with the grafting operation. We provide an example of implementing an algebra of graphs, starting with a general graph and imposing only the minimal required structure to define the product of interest, which is the grafting product in our case.

DEFINITION 25. *A* graph *g is a set of vertices $V(g)$ together with a set of directed edges $E(g)$. An edge e is defined by its source $s(e) \in V(g)$ and target $t(e) \in V(g)$. A* rooted *graph has a marked vertex called a root.*

We denote the set of graphs by G and the set of rooted graphs by G_R. The implemention of Definition 25 can be found in Implementation 26.

IMPLEMENTATION 26. *A graph is any type* `g` *that is an instance of the* `Graph` *type class. The* `Graph` *type class provides functions for constructing graphs, adding an edge to a graph, and adding graphs together by taking the unions of the sets of vertices and edges.*

```
class Graph g where
    type Vertex g

    singleton :: Vertex g -> g
```

```
    edges :: g -> MultiSet (Vertex g, Vertex g)
    vertices :: g -> MultiSet (Vertex g)
    addEdge :: (Vertex g, Vertex g) -> g -> g
    addGraph :: (Graph g0, Vertex g ~ Vertex g0) => g0 -> g -> g

class (Graph g) => RootedGraph g where
    root :: g -> Vertex g
```

A straightforward implementation of a graph which uses integers as vertices is presented in Implementation 27.

IMPLEMENTATION 27. *A simple implementation of a graph with integers as vertices. Finally, we define the* `Rooted` *type and its* `RootedGraph` *instance.*

```
data IntegerGraph
    = IG (MultiSet Integer) (MultiSet (Integer, Integer))

instance Graph IntegerGraph where
    type Edge IntegerGraph = (Integer, Integer)

    singleton v = IG (singleton v) empty
    edges (IG _ es) = es
    vertices (IG vs _) = vs
    addGraph g (IG vs es) =
        IG (vertices g `union` vs) (edges g `union` es)
    addEdge e (IG vs es) =
        IG vs (e `insert` es)

data Rooted g = R (Vertex g) g

instance (Graph g) => RootedGraph (Rooted g) where
    root (R r _) = r
```

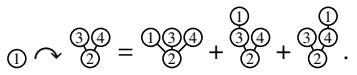
We define a helper function `integerGraph` used to construct an `IntegerGraph` from a list of vertices and edges. We also define the helper function `rooted` which constructs a rooted graph `Rooted g` from a graph `g` by choosing a vertex as a root. We do no discuss the implementations of these helper functions for brevity.

USAGE EXAMPLE 28. *An example of a graph:*

```
>>> g = integerGraph [1,2,3] [(1,1),(2,1),(2,3)]
>>> g
IntegerGraph(V=[1,2,3], E=[(1,1),(2,1),(2,3)])
>>> rg = rooted g 1
>>> rg
RootedIntegerGraph(V=[1,2,3], E=[(1,1),(2,1),(2,3)], R=1)
```

We note that since we don't assume any structure on the graph, we can't display it in a more visually appealing way.

We are now ready to define and implement a generalization of the grafting product. We rely on Definition 6 and define the grafting of a rooted graph $g_r \in G_R$ onto a graph $g \in G$ as a sum of all ways to attach the root of g_r to a vertex of g. For example,



The implementation of the grafting product as defined in Definition 6 is presented below.

IMPLEMENTATION 29. *Implementation of the grafting product.*

```
graftGraph
    :: ( Eq g2, Graded g2, Graph g2
       , RootedGraph g1, Vertex g1 ~ Vertex g2
       )
    => g1 -> g2 -> Vector Integer g2
graftGraph rg1 g2 =
    vectorFromList $
        map ((1 *^) . graftGraphTo rg1 g2) $
            toList $
                vertices g2

graftGraphTo
    :: ( RootedGraph g1
       , Graph g2, Vertex g1 ~ Vertex g2
       )
    => g1 -> g2 -> Vertex g2 -> g2
graftGraphTo rg1 g2 v
    = addGraph rg1 $ addEdge (root rg1, v) g2
```

We use the `bilinear` function to extend `graftGraph` from the basis sets to the corresponding vector spaces to obtain the grafting product. The map `vectorFromList` constructs a vector as discussed in Section 3.

USAGE EXAMPLE 30. *Example of the grafting product of graphs.*

```
>>> rg1 = rooted (integerGraph [1] []) 1
>>> rg2 = rooted (integerGraph [2] []) 2
>>> g1 = integerGraph [3,4] [(4,3)]
>>> g2 = integerGraph [5] [(5,5)]
>>> bilinear graftGraph
      (vectorFromList [1 *^ rg1, 2 *^ rg2])
      (vectorFromList [3 *^ g1, 4 *^ g2])
(4 *^ IntegerGraph(V=[1,5], E=[(1,5),(5,5)])
      + 8 *^ IntegerGraph(V=[2,5], E=[(2,5),(5,5)]))_2
      + (3 *^ IntegerGraph(V=[1,3,4], E=[(1,3),(4,3)])
```

```
    + 3 *^ IntegerGraph(V=[1,3,4], E=[(1,4),(4,3)])
    + 6 *^ IntegerGraph(V=[2,3,4], E=[(2,3),(4,3)])
    + 6 *^ IntegerGraph(V=[2,3,4], E=[(2,4),(4,3)]))_3
```

We note that `rg1` *and* `rg2` *are rooted graphs while* `g1` *and* `g2` *are non-rooted.*

We recall that the technique presented in this subsection defines a general graph before imposing the structure necessary to define the grafting product. While this approach is flexible and can accommodate a wide variety of graphs, it may also introduce additional complexity and reduce readability.

4.3 Grafting of forests

In this section, we take an alternative approach by constructing the graphs we are interested in from the ground up. This approach allows us to leverage their properties to implement efficient formulas for operations and ensures that the structure we aim to preserve is maintained through these operations which is guaranteed by Haskell's strong and flexible type system. However, this approach requires a more detailed understanding of the graph's structure compared to the one discussed in Section 4.2.

We will explore the definition and implementation of decorated forests, along with grafting, Grossman-Larson products, and Connes-Kreimer coproduct. We start by implementing trees following Definition 2. A tree, in this case, is any type `t` which has an instance of `IsTree`.

Implementation 31. *The* `IsDecorated` *type class is used to define the decoration of the tree, while the* `IsTree` *type class is used to define the root, children, and construction of the tree.*

```
class IsDecorated a where
    type Decoration a

class (IsDecorated t) => IsTree t where
    root :: t -> Decoration t
    branches :: t -> MultiSet t
    buildTree :: Decoration t -> MultiSet t -> t
```

An implementation of non-planar trees `Tree d` is presented in Implementation 32 with forests being represented as multisets of non-planar trees `MultiSet (Tree d)`.

Implementation 32. *We define the* `Tree` *data type which is an instance of the* `IsDecorated` *and* `IsTree` *type classes. The* `Tree` *data type represents a tree with a root and a multiset of branches.*

```
data Tree d = T
    { nonplanarRoot :: d
    , nonplanarBranches :: MultiSet (Tree d)
    }

instance IsDecorated (Tree d) where
    type Decoration (Tree d) = d
```

```
instance IsTree (Tree d) where
    root = nonplanarRoot
    branches = toList . nonplanarChildren

    buildTree r = T r . fromList
```

To facilitate the construction of forests from a textual representation, we provide the `fromBrackets` function, which is part of the `HasBracketNotation` type class. This function takes a single string representing a forest in bracket notation and converts it into the corresponding Haskell forest type. For example, given a forest encoded as a string in bracket notation, the expression

```
fromBrackets "1[2,3],4[5[6]],7" :: MultiSet (Tree Integer)
```

produces a forest represented as a multiset of `Tree Integer`.

4.3.1 *Grafting and Grossman-Larson products*

Implementation 33 provides the implementation of the grafting and Grossman-Larson products of forests using the Definition 10.

IMPLEMENTATION 33. *To facilitate future extension of the algebras, we define the* `CanGraft` *and* `CanGrossmanLarson` *type classes for graphs that support grafting and Grossman-Larson products.*

```
class (IsVector a) => CanGraft a where
    graft
        :: a -> a -> Vector (VectorScalar a) (VectorBasis a)

class (IsVector a) => CanGrossmanLarson a where
    grossmanLarson
        :: a -> a -> Vector (VectorScalar a) (VectorBasis a)

instance
    ( IsTree t
    , IsVector t
    ) => CanGraft (MultiSet t)
    where
    graft empty empty = vector empty
    graft _ empty = vector Zero
    graft empty f2 = vector f2
    graft f1 f2 = case (toList f2) of
        [t]    -> linear (singleton . buildTree (root t))
                      $ grossmanLarson f $ branches t
        t : ts -> linear perTerm $ deshuffle f1
      where
```

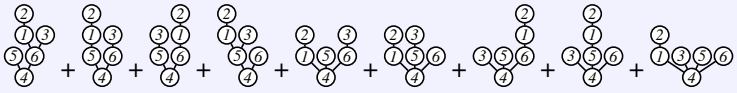
```
        perTerm (f11, f12)
            = graft f11 (singleton t) * graft f12 ts

instance
    ( IsTree t
    , IsVector t
    )
    => CanGrossmanLarson (MultiSet t)
    where
    grossmanLarson f1 f2 = linear perTerm $ deshuffle f1
      where
        perTerm (f11, f12)
            = vector f11 * graft f12 f2
```

An example of the use of forests can be found in Usage Example 1 in the *Getting Started* section of the paper as well as in Usage Example 34.

USAGE EXAMPLE 34. *We define the trees*  *and*  *and compute their grafting product.*

```
>>> f1 = fromBrackets "1[2],3" :: MultiSet (Tree Integer)
>>> f2 = fromBrackets "4[5,6]" :: MultiSet (Tree Integer)
>>> display $ graft f1 f2
```



4.3.2 *Connes-Kreimer coproduct*

We implement Connes-Kreimer coproduct by following Definition 9.

IMPLEMENTATION 35. *To facilitate future extension of the Connes-Kreimer coalgebra, we define the* `CanConnesKreimer` *type class for graphs that support Connes-Kreimer coproduct.*

```
class (IsVector a) => CanConnesKreimer a where
    connesKreimer
        :: a -> Vector (VectorScalar a)
                       (VectorBasis a, VectorBasis a)

instance
    ( Ord t
    , IsVector t
    , IsTree t
    ) => CanConnesKreimer (MS.MultiSet t) where
    connesKreimer ms = case (toList ms) of
        []  -> vector (empty, empty)
        [t] -> vector (singleton t, empty)
```

```
            + (linear perTerm
                $ connesKreimer $ fromList $ branches t)
      where
        perTerm (f1, f2) = (f1, singleton
                                  $ buildTree r $ toList f2)
        r = root t
    f  -> product
            $ map (connesKreimer . singleton) f
```

5 Conclusion

In this work, we have presented `Arboretum.hs` as a flexible and type-safe framework for symbolic computations with algebras of trees and forests built on top of graded vector spaces, designed to closely mirror their mathematical structure while remaining accessible and extensible. In contrast to existing software such as `BSeries.jl`, which focuses on efficient implementations of classical Butcher series built from classical rooted trees and is primarily tailored to established applications in numerical analysis, `Arboretum.hs` adopts a broader and more exploratory perspective, accommodating more general combinatorial structures and algebraic operations beyond the classical setting. Moreover, while `BSeries.jl` benefits from the performance and ecosystem of Julia, it does not leverage a static type system to encode algebraic invariants, leaving certain classes of errors to be detected only at runtime; by contrast, Haskell's strong and expressive type system allows `Arboretum.hs` to reflect algebraic structures directly in the code and enforce correctness properties at compile time. As a result, the package provides a robust environment not only for computation but also for experimentation and verification, making it a useful tool for researchers working at the interface of algebra, combinatorics, and applications.

For the sake of exposition, we have focused on the implementation of the pre-Lie grafting algebra of decorated forests within the `Arboretum.hs` package. However, the design principles and modular structure of the package allow for straightforward extensions to other algebraic structures and operations on graphs. In particular, the package includes the following implementations that were not discussed in detail in the main text:

(1) Shuffle-deconcatenation Hopf algebra,
(2) Planar forests and trees together with grafting, Grossman-Larson products, Connes-Kreimer coproduct, and other operations for which recursive formulas suitable for implementation were derived in [36],
(3) Aromatic forests and trees together with the grafting, Grossman-Larson, substitution products, as well as the divergence operation. See [6, 21].
(4) Syntactic trees which correspond to the mathematical concept of operads or computer science concept of abstract syntax trees.

Planned features include:

(1) Implementation of exotic forests and the corresponding products and operations, see [4, 30],
(2) Implementation the algorithm for the generation of order conditions for invariant measure sampling of overdamped Langevin dynamics, see [4, 30].

(3) Integration of the package `Operads`[7] which computes Gröbner basis for operads (see [20]) into the `Arboretum.hs` package.

We believe this package will serve as a useful tool for both practitioners and theoretical researchers working at the interface of algebra, combinatorics, and applications.

Acknowledgements

EB acknowledges the support of the Knut and Alice Wallenberg Foundation (grant number KAW 2023.0433). EB and GV acknowledge the support of the Swiss National Science Foundation, projects No 200020_214819, No. 200020_192129 and No. 10009199.

References

[1] Bogfjellmo, G. (2019) Algebraic structure of aromatic B-series. *Journal of Computational Dynamics*. **6**(2), 199–222.

[2] Bogfjellmo, G., Curry, C. & Manchon, D. (2017) Hamiltonian B-series and a Lie algebra of non-rooted trees. *Numerische Mathematik*. **135**(1), 97–112.

[3] Bronasco, E. (2025) *Algebraic structures and numerical methods for invariant measure sampling of Langevin dynamics*. Thesis. University of Geneva.

[4] Bronasco, E. (2025) Exotic B-Series and S-Series: Algebraic Structures and Order Conditions for Invariant Measure Sampling. *Foundations of Computational Mathematics*. **25**(1), 271–301.

[5] Bronasco, E. (2026) arboretum.hs. https://doi.org/10.5281/zenodo.19737034.

[6] Bronasco, E. & Busnot Laurent, A. (2026) Hopf algebra structures for the backward error analysis of ergodic stochastic differential equations. *Numerische Mathematik*.

[7] Bronasco, E., Leimkuhler, B., Phillips, D. & Vilmart, G. (2025) Efficient Langevin sampling with position-dependent diffusion. *submitted for publication*. (arXiv:2501.02943).

[8] Brouder, Ch. (2000) Runge–Kutta methods and renormalization. *The European Physical Journal C - Particles and Fields*. **12**(3), 521–534.

[9] Bruned, Y., Hairer, M. & Zambotti, L. (2019) Algebraic renormalisation of regularity structures. *Inventiones mathematicae*. **215**(3), 1039–1156.

[10] Burrage, K. & Burrage, P. M. (2000) Order Conditions of Stochastic Runge–Kutta Methods by B-Series. *SIAM Journal on Numerical Analysis*. **38**(5), 1626–1646.

[11] Butcher, J. C. (1963) Coefficients for the study of Runge-Kutta integration processes. *Journal of the Australian Mathematical Society*. **3**(2), 185–201.

[12] Butcher, J. C. (1972) An algebraic theory of integration methods. *Math. Comp.* **26**, 79–106.

[13] Butcher, J. C. (2021) *B-Series: Algebraic Analysis of Numerical Methods*. vol. 55 of *Springer Series in Computational Mathematics*. Springer International Publishing. Cham.

[14] Calaque, D., Ebrahimi-Fard, K. & Manchon, D. (2011) Two interacting Hopf algebras of trees. *Advances in Applied Mathematics*. **47**(2), 282–308.

[15] Cayley, A. (1857) On the theory of the analytical forms called trees. *Philosophical Magazine*. **13**, 172–176.

[16] Chapoton, F. & Livernet, M. (2001) Pre-Lie algebras and the rooted trees operad. *International Mathematics Research Notices*. **2001**(8), 395–408.

[17] Chartier, P., Hairer, E. & Vilmart, G. (2010) Algebraic Structures of B-series. *Foundations of Computational Mathematics*. **10**(4), 407–427.

[18] Chartier, P. & Murua, A. (2007) Preserving first integrals and volume forms of additively split systems. *IMA Journal of Numerical Analysis*. **27**(2), 381–405.

[19] Connes, A. & Kreimer, D. (1998) Hopf Algebras, Renormalization and Noncommutative Geometry. *Communications in Mathematical Physics*. **199**(1), 203–242.

[20] Dotsenko, V. & Khoroshkin, A. (2010) Gröbner bases for operads. *Duke Mathematical Journal*. **153**(2), 363–396.

[7] https://hackage.haskell.org/package/Operads

[21] Fløystad, G., Manchon, D. & Munthe-Kaas, H. Z. (2021) The universal pre-Lie–Rinehart algebras of aromatic trees. Geometric and Harmonic Analysis on Homogeneous Spaces and Applications. Springer International Publishing. pp. 137–159.
[22] Gubinelli, M. (2010) Ramification of rough paths. *Journal of Differential Equations*. **248**(4), 693–721.
[23] Hairer, E., Lubich, C. & Wanner, G. (2010) *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. Springer Berlin Heidelberg.
[24] Hairer, E., Nørsett, S. P. & Wanner, G. (1993) *Solving Ordinary Differential Equations I*. vol. 8 of *Springer Series in Computational Mathematics*. Springer. Berlin, Heidelberg.
[25] Hairer, E. & Wanner, G. (1974) On the Butcher group and general multi-value methods. *Computing*. **13**(1), 1–15.
[26] Hairer, M. (2014) A theory of regularity structures. *Inventiones mathematicae*. **198**(2), 269–504.
[27] Iserles, A., Quispel, G. & Tse, P. (2007) B-Series methods cannot be volume-preserving. *BIT Numerical Mathematics*. **47**(2), 351–378.
[28] Ketcheson, D. I. & Ranocha, H. (2023) Computing with B-series. *ACM Transactions on Mathematical Software*. **49**(2).
[29] Laurent, A. & Vilmart, G. (2020) Exotic aromatic B-series for the study of long time integrators for a class of ergodic SDEs. *Mathematics of Computation*. **89**(321), 169–202.
[30] Laurent, A. & Vilmart, G. (2022) Order conditions for sampling the invariant measure of ergodic stochastic differential equations on manifolds. *Foundations of Computational Mathematics*. **22**(3), 649–695.
[31] Loday, J.-L. & Vallette, B. (2012) *Algebraic Operads*. vol. 346 of *Grundlehren Der Mathematischen Wissenschaften*. Springer. Berlin, Heidelberg.
[32] Lyons, T. J. (1998) Differential equations driven by rough signals. *Revista Matemática Iberoamericana*. **14**(2), 215–310.
[33] Mokhov, A. (2017) Algebraic graphs with class (functional pearl). Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. New York, NY, USA. Association for Computing Machinery. pp. 2–13.
[34] Munthe-Kaas, H. (1995) Lie-Butcher theory for Runge-Kutta methods. *BIT Numerical Mathematics*. **35**(4), 572–587.
[35] Munthe-Kaas, H. & Verdier, O. (2016) Aromatic Butcher Series. *Foundations of Computational Mathematics*. **16**(1), 183–215.
[36] Munthe-Kaas, H. Z. & Føllesdal, K. K. (2018) Lie–Butcher Series, Geometry, Algebra and Computation. Discrete Mechanics, Geometric Integration and Lie– Butcher Series. Cham. Springer International Publishing. pp. 71–113.
[37] Munthe-Kaas, H. Z. & Wright, W. M. (2006) On the Hopf Algebraic Structure of Lie Group Integrators.
[38] Murua, A. (2006) The Hopf Algebra of Rooted Trees, Free Lie Algebras, and Lie Series. *Foundations of Computational Mathematics*. **6**(4), 387–426.
[39] Oudom, J.-M. & Guin, D. (2008) On the Lie enveloping algebra of a pre-Lie algebra. *Journal of K-Theory*. **2**(1), 147–167.
[40] Ranocha, H. & Ketcheson, D. I. (2021) BSeries.jl: Computing with B-series in Julia. https://github.com/ranocha/BSeries.jl.
[41] Ranocha, H. & Ketcheson, D. I. (2021) Bseries.py. https://github.com/ketch/BSeries.
[42] Rößler, A. (2006) Rooted Tree Analysis for Order Conditions of Stochastic Runge-Kutta Methods for the Weak Approximation of Stochastic Differential Equations. *Stochastic Analysis and Applications*. **24**(1), 97–134.
[43] Sundklakk, H. S. (2015) *A library for computing with trees and B-Series*. Master's thesis. NTNU. Supervisor: B. Owren.
[44] Sundklakk, H. S. (2015) pybs. https://github.com/henriksu/pybs.