Eur. Phys. J. C (2023) 83:636
https://doi.org/10.1140/epjc/s10052-023-11787-2

THE EUROPEAN PHYSICAL JOURNAL C

Regular Article - Theoretical Physics

Functional directed acyclical graphs for scattering amplitudes in perturbation theory

Thorsten Ohl[a]

Institute of Theoretical Physics and Astrophysics, University of Würzburg, Emil-Hilb-Weg 22, 97074 Würzburg, Germany

Received: 6 June 2023 / Accepted: 30 June 2023 / Published online: 18 July 2023

Abstract I describe a mathematical framework for the efficient processing of the very large sets of Feynman diagrams contributing to the scattering of many particles. I reexpress the established numerical methods for the recursive construction of scattering elements as operations on compact abstract data types. This allows efficient perturbative computations in arbitrary models, as long as they can be described by an effective, not necessarily local, Lagrangian.

1 Introduction

The efficient and reliable computation of scattering amplitudes for many particles in a large class of models, both on tree level and including higher order corrections, is a central element of all efforts for analyzing the physics at LHC and possible future colliders.

Since the first release of MADGRAPH [1] about 30 years ago, there has been tremendous progress in the capabilities of the tools that can compute such scattering amplitudes numerically. Replacing sums of Feynman diagrams by recursive numerical evaluation opened the realm of many-legged amplitudes, including loop corrections. In fact, the treatment of QCD corrections has matured so much that tools like MADGRAPH5 [2] are now employed regularly by endusers for LHC physics. Electroweak radiative corrections are starting to become available in user friendly tools and recursive techniques are being applied to loop calculations. At the same time, de facto standard formats like UFO [3,4] allow the specification of almost any physics model that might be of interest in the near and not so near future.

In this paper, I will elaborate a common mathematical structure behind the recursive calculations. The focus is not on the immediate numerical evaluation, but on the elucidation of an algebraic structure that will later be translated into numerical code. This simplifies supporting more general interactions, because purely numerical codes have to make assumptions that can turn out to be hard to relax later. In addition, algebraic expressions can be used to generate more comprehensive tests of models and implementations. They also simplify the automatic generation of the additional expressions needed for subtractions schemes [5].

Finally, at a time when functional programming and strong type systems are moving more and more from academia into the mainstream, it is a useful exercise to reconstruct the mathematical structures in a way that can easily be translated into efficient programs making use of these paradigms. The mathematical structures presented here have not been developed in a vacuum, but are a distillation of commonalities observed in the concrete data structures implemented for the matrix element generator O'MEGA [6] that is part of the WHIZARD event generator [7].

Nothing in the following discussion will be specific to leading order, tree level matrix elements. Exactly the same structures appear when implementing loops using additional legs [8–11] or when adding higher order contributions as terms in an effective action using a skeleton expansion. The translation of the algebraic expressions into robust numeric code calling sophisticated external libraries for loop integrals [12] is much more challenging, of course. However, also here the algebraic step offers more options than a purely numerical approach.

The outline of the paper is as follows: in Sect. 2, I briefly review the recursive techniques used for computing scattering amplitudes for processes with many external particles. This section also serves the purpose of establishing the terminology and notation used in the remaining sections. In Sect. 3, I introduce Directed Acyclical Graphs (DAGs), bundles and their relationships. In Sect. 4, I present an algorithm for efficiently constructing the DAGs representing scattering amplitudes. In Sect. 5, I briefly describe how to gener-

[a] e-mail: ohl@physik.uni-wuerzburg.de (corresponding author)

ate efficient numerical code from DAGs constructed according to the algorithm presented in the previous sections. In Appendix A, I sketch the implementation of DAGs and bundles in O'MEGA [6,7].

2 Scattering amplitudes

It has long been recognized that the textbook representation of scattering amplitudes as a sum of Feynman diagrams becomes very inefficient as the number of external particles rises. Indeed, even though general estimates are hard to derive for realistic models with conserved quantum numbers, analytic formulae for toy models and explicit calculations for specific processes confirm the expectation that the number of tree level Feynman diagrams grows factorially with the number of external particles. If Feynman diagrams with loops are represented by tree diagrams [8–11], each loop adds two more external particles. In addition to requiring prohibitive computational resources, the destructive interferences inherent in gauge theories lead to a loss of precision if too many terms are added. Starting with $2 \to 6$ processes at tree level, the need for a more efficient representation became evident.

In order to simplify the notation in this section, I will cross all scattering amplitudes from $n_{\text{in}} \to n_{\text{out}}$ to $n = n_{\text{in}} + n_{\text{out}} \to 0$. Except for the momentum, I will also suppress all quantum numbers in this introductory section. The treatment of general quantum numbers (spin, flavor, color, etc.) will be the focus of the following sections.

2.1 Recursion

The appropriate building blocks to replace Feynman diagrams turned out to be k-particle matrix elements of fields

$$\phi(\{i_1, i_2, \ldots, i_k\}) = \langle 0|\Phi|p_{i_1}, p_{i_2}, \ldots, p_{i_k}\rangle \tag{1a}$$

which will be referred to as *wavefunctions* or of their associated *currents*

$$j(\{i_1, i_2, \ldots, i_k\}) = \langle 0|J|p_{i_1}, p_{i_2}, \ldots, p_{i_k}\rangle, \tag{1b}$$

as pioneered by Berends and Giele [13]. The set of indices $I = \{i_1, i_2, \ldots, i_k\}$ is a subset of the indices enumerating the external particles or open loop momenta.

Since $I \in 2^{\{1,2,\ldots,n\}}$, the number of possible different wavefunctions or currents grows only as an exponential 2^n instead of a factorial $n! \sim n^n$. Furthermore, both can be computed recursively

$$\phi(I) = \sum_{I_1 \cup I_2 = I} P_I V_{I,I_1,I_2} \phi(I_1)\phi(I_2) \tag{2a}$$

$$j(I) = \sum_{I_1 \cup I_2 = I} V_{I,I_1,I_2} P_{I_1} j(I_1) P_{I_2} j(I_2), \tag{2b}$$

without expanding them into Feynman diagrams, which would reintroduce factorial growth. In (2), P_I denotes a propagator and V_{I,I_1,I_2} a vertex factor for three legs. The generalization to models containing vertices with more than three legs is obvious.

Note that ϕ is just j multiplied by a momentum space propagator. Thus the choice between the two is only a matter of convenience. The rest of the paper will mostly refer to wavefunctions (1a), but all constructions can be repeated trivially for the currents (1b).

2.2 Topologies

There are many ways in which a scattering amplitude $\mathcal{M}$ can be constructed from (1). The first approach observes that the j in (1b) is already amputated. It therefore suffices to set the momentum

$$p_1 = -\sum_{i=2}^{n} p_i \tag{3}$$

on the mass shell of particle 1 to obtain the scattering amplitude using the LSZ prescription

$$\mathcal{M}(\{1, 2, \ldots, n\}) = j(\{2, 3, \ldots, n\}). \tag{4a}$$

This is implemented numerically in HELAC [14,15] and RECOLA [10,11].

The second approach glues the ϕ from (1a) at vertices to obtain the scattering amplitude in the form

$$\mathcal{M}(\{1, 2, \ldots, n\}) = \sum_{I_1 \cup I_2 \cup I_3 = \{1,\ldots,n\}} K_{I_1,I_2,I_3} \phi(I_1)\phi(I_2)\phi(I_3) \tag{4b}$$

with obvious generalizations to models containing vertices with more than three legs. The partitions (I_1, I_2, I_3) of the external particles must be chosen carefully to avoid double counting [6,16] and the *keystones* K correspond to vertex factors. This approach was pioneered for numerical calculations in the standard model by ALPHA [16–18] and is implemented as an algebraic algorithm for arbitrary models in O'MEGA [6,7].

The third approach combines the DAGs at propagators

$$\mathcal{M}(\{1, 2, \ldots, n\}) = \sum_{I \cup I' = \{1,\ldots,n\}} j(I) P_{I,I'} j(I') \tag{4c}$$

instead of vertices as in (4c). It was pioneered by COMIX [19] and OPENLOOPS [8,9]

Algebraically, all expressions (4) will give the same final result, but the number of nodes that need to be evaluated can vary slightly and numerical results will differ due to the different order of evaluation. O'MEGA [6,7] allows to compute the amplitude both as (4a) and as (4b) and confirms these expectations.

While it is impossible to give general estimates for the number of wavefunctions that need to be evaluated in realistic models, one can count them for some examples using O'MEGA. In the standard model, it appears that (4a) requires some 10% fewer evaluations than (4b) in an optimal implementation. One advantage of (4b) and (4c) is that at most $n/2$ of the external momenta appear in the ϕ compared to $n-1$ for (4a). Therefore fewer steps with accumulating floating point errors are required in the recursive evaluation of $\phi(I)$. While this could in principle be a significant advantage in amplitudes with strong gauge cancellations, the difference appears to be small in practice.

The algorithm adding quantum numbers to (1), (2) and (4) described in the following sections is equally applicable for all three variants in (4).

2.3 Evaluation

In the case of a fixed physics model with a moderate number of fields and couplings, such as the standard model, the recursive evaluation (2) can be expressed as an iteration of matrix multiplications [10,11,14–19]. This approach has the advantage that all scattering amplitudes in the supported model can be computed using the same executable, without the need for recompilation.

However, extending this approach to more complicated models, in particular to models that can be specified by endusers in formats like UFO [3,4], is far from trivial [20]. Instead, it is beneficial to represent the recursion relations (2) abstractly by a data structure from which dedicated code can be generated and compiled subsequently, following the pioneering treatment of Feynman diagrams in MADGRAPH [1,2]. This approach has been implemented for the recursive evaluation in [6–9].

This motivates the search for a data structure that represents the recursion relations (2) concisely and can be constructed efficiently from the Feynman rules of a model. The obvious candidate is a finite Directed Acyclical Graph (DAG), that corresponds to the evaluation of an arithmetical expression in which common subexpressions are evaluated only once and later recalled from memory when needed again.

Additional benefits of algebraic manipulations are that it is easier to prune the computation of wavefunctions that are not needed in the final result, that one can target special hardware or dedicated virtual machines [21] that avoid the need for compilation. Formfactors can be restricted to lightlike momenta at compile time [22–24]. Finally, one can optionally instrument the code with numerical checks of Ward and Slavnov–Taylor identities for gauge boson wavefunctions, in order to test matrix element generator, numerical libraries and model descriptions.

3 DAGs and bundles

In this section, I will focus on universal mathematical constructions, not practical algorithms. The discussion of the latter will follow in Sect. 4.

Given a set N of *nodes*, a set E of *edges* and a set $C(N) \subseteq 2^N$ of *children* which is typically the set of subsets of nodes with a limited number of elements, any map from N to the powerset of $E \times C(N)$

$$\Delta : N \to 2^{E\times C(N)} \tag{5}$$

defines a Directed Graph $\mathbf{G} = (N, E, \Delta)$ in the sense described below. The function Δ can be specified completely by the set $\{(n, \Delta(n)) | n \in N\}$ of ordered pairs. This equivalence will be used below to define transformations on DAGs as set theoretical operations that can be implemented efficiently in computer programs. I will often employ the more intuitive notation $\{n \mapsto \Delta(n) | n \in N\}$ or the abbreviated form $\{\delta_n | n \in N\}$. In order to avoid excessive nested superscripts, I will sometimes use the notation $A \to B$ for set B^A of all functions from the set A to the set B.

With wavefunctions as nodes and vertex factors as edges, this definition captures the recursion relations (2) exactly. Note that the map Δ (5) is well defined iff the combination of momenta and other quantum numbers identifies the wavefunctions or currents uniquely.

There are cases where physical quantum numbers are not sufficient to distinguish wavefunctions. For example, if the scattering amplitude is to be expanded in the powers of some coupling constants, these powers can contribute at different levels of the recursive expansion. Therefore a wavefunction can appear more than once with the same momentum and physical quantum numbers. This forces us to add the powers of these coupling constants as unphysical labels that will be combined in the final step (4). Since such labels are later required anyway to disambiguate variable names in the generated numerical code, this adds no additional burden. Such a counting of coupling constants is of course crucial for adding a consistent number of counterterms in calculations involving loops and when adding precomputed loops using a skeleton expansion or effective actions.

The nodes in the preimage of $\emptyset$ under Δ

$$L = \Delta^{-1}(\emptyset) = \{n \in N | \Delta(n) = \emptyset\} \tag{6}$$

are called *leaf nodes* and correspond to the external states in scattering amplitudes. Since the elements of C are sets of elements of N, we can derive from Δ two mutually recursive expansion functions

$$\hat{\Delta} : E \times C(N) \to E \times C(2^{E\times C(N)})$$
$$(e, \{n_i|i \in I\}) \mapsto (e, \left\{\Delta^*(n_i)|i \in I\right\}) \tag{7a}$$

element-by-element and

$$\Delta^*(n) = \begin{cases} \{n\} & \text{for } \Delta(n) = \emptyset \\ \hat{\Delta}(\Delta(n)) & \text{for } \Delta(n) \neq \emptyset. \end{cases} \tag{7b}$$

If $\mathbf{D} = (N, E, \Delta)$ represents an acyclical graph, i.e. a DAG, with a finite number of nodes $|N|$, the functions Δ^* and $\hat{\Delta}$ will reach a fixed point after a finite number of steps. This fixed point consists exclusively of mutually nested sets of leaves. If the image of Δ consists only of singleton sets and $\emptyset$, the fixed point reached from any starting node n corresponds to a tree diagram. Otherwise it corresponds to a forest of tree diagrams, if the elements of the sets are distributed recursively.

As an illustration, consider the DAG $\mathbf{D}$ with the sets

$$N = \{1, 2, \ldots, 8\} \tag{8a}$$
$$E = \emptyset \tag{8b}$$
$$C = \left\{\{n, n'\}|n \neq n' \in N\right\} \tag{8c}$$

and the map

$$\Delta = \big\{1 \mapsto \emptyset,\ 2 \mapsto \emptyset,\ 3 \mapsto \emptyset,\ 4 \mapsto \emptyset,$$
$$5 \mapsto \{\{1, 2\}\},\ 6 \mapsto \{\{5, 3\}\},\ 7 \mapsto \{\{5, 4\}\},$$
$$8 \mapsto \{\{6, 4\}, \{7, 3\}\}\big\}, \tag{8d}$$

where I have not spelled out the unlabeled edges. A quick calculation gives

$$\Delta^*(8) = \{\{\{\{\{\{1, 2\}\}, 3\}\}, 4\},$$
$$\{\{\{\{\{1, 2\}\}, 4\}\}, 3\}\}. \tag{9}$$

This corresponds to the forest consisting of the trees

$$\{\{\{1, 2\}, 3\}, 4\} \tag{10a}$$
$$\{\{\{1, 2\}, 4\}, 3\}\,. \tag{10b}$$

This DAG encodes a stripped down version of the Feynman diagrams for the process $e^+e^-q\bar{q} \to g$, that ignores both the details of the couplings and the contributions of Z and Higgs bosons. Note that the common subdiagram $e^+e^- \to \gamma$ appears only once in the DAG as $5 \mapsto \{\{1, 2\}\}$, but twice in the forest.

A general directed graph can contain cycles and the functions Δ^* and $\hat{\Delta}$ will not reach a fixed point even if $|N| < \infty$. As described in Sect. 4.1, it will however always be possible to equip N with a natural order so that the application of Δ acts strictly decreasing with respect to this order. There can obviously be no cycles and the graph is guaranteed to be a DAG in this case.

If the same node n appears many times in the children, a DAG provides a very efficient encoding of large sets of graphs. The storage and computing time required by typical sets of tree diagrams grows factorially with the number of leaves $|L|$. In contrast, the space and time required for implementing the DAG scales linearly with $|N|$, which only grows exponentially with $|L|$. Using persistent functional data structures [25] instead of mutable arrays to implement the function Δ simplifies the algorithm described below significantly. The additional space and time requirements replace $|N|$ by $|N| \ln |N|$ and turn out not to be important for large $|N|$.

3.1 Constructing DAGs

Using DAGs as a compact representation has only a marginal benefit if their construction requires the generation of all tree diagrams in intermediate steps or if the applications require a full expansion. Fortunately, the sum of Feynman diagrams encoded in the DAG can be evaluated either using the DAG directly or by generating a dedicated numerical code that evaluates each node $n \in N$ only once. As explained in Sect. 4, it turns out that the DAGs representing perturbative scattering amplitudes can be constructed without requiring the construction of the corresponding forest.

For this purpose, I introduce the empty DAG

$$\epsilon = (\emptyset, \emptyset, \emptyset, \emptyset) \tag{11}$$

where $\Delta = \emptyset$ is the function with empty domain and codomain. I also define a function

$$\omega : (N \to E \times 2^{C(N)}) \times \mathcal{D} \to \mathcal{D}$$
$$(n \mapsto (e, c), x) \mapsto \omega_{n\mapsto(e,c)}(\mathbf{D}) \tag{12a}$$

with the function $\omega_{n\mapsto(e,c)}$ that adds a node n together with the mapping $n \mapsto (e, c)$

$$\omega_{n\mapsto(e,c)}(N, E, \Delta)$$
$$= (N \cup \{n\}, E \cup e, \Delta \cup \{n \mapsto (e, c)\})\,, \tag{12b}$$

where e and (e, c) are shorthands for the sets $\{e_i|i \in I\}$ and $\{(e_i, c_i)|i \in I\}$ with $|I|$ elements. In particular, they may be empty to allow inserting a leaf node. In order to avoid ambiguities in the definition of ω, I will require that $n \notin N \wedge n'_i \in N$ in $\omega_{n\mapsto\{(e,\{n'_i|i\in I\})\}}$.

With these definitions, the DAG in (8) is

$$\omega_{8\mapsto\{\{6,4\},\{7,3\}\}}\omega_{7\mapsto\{\{5,8\}\}}\omega_{6\mapsto\{\{5,3\}\}}$$
$$\omega_{5\mapsto\{\{1,2\}\}}\omega_{4\mapsto\emptyset}\omega_{3\mapsto\emptyset}\omega_{2\mapsto\emptyset}\omega_{1\mapsto\emptyset}\epsilon, \tag{13}$$

where the function applications associate to the right, of course. It is obvious that any finite DAG can be constructed by repeated applications of ω.

For the finite DAGs that are the subject of this paper, the function ω can be implemented easily in programming languages that have efficient support for persistent sets and maps (also known as dictionaries) that can grow without a lot of reallocation. Functional programming languages with garbage collection make such implementations particularly straightforward. The domain and codomain of functions like ω (12) are highly structured sets and static type systems allow to verify already at compile time that only matching functions are being composed. Beyond preventing errors, a strict type discipline helps to uncover mathematical structures, such as the ones described in this section. This paper is based on the implementation in the matrix element generator O'MEGA [6] using `ocaml` [26], as described in Appendix A.1.

3.2 Lattices of DAGs

For our purposes, DAGs representing scattering amplitudes for the same external states, categories of DAGs that share the same leaf nodes

$$\mathcal{D}_L = \left\{\mathbf{D} = (N, E, \Delta) | \Delta^{-1}(\emptyset) = L\right\} \tag{14}$$

are the most interesting. Since we describe a DAG as a tuple of sets, there is a natural notion of inclusion for pairs of DAGs in $\mathcal{D}_L$

$$\mathbf{D}' = (N', E', \Delta') \subseteq \mathbf{D} = (N, E, \Delta) \Leftrightarrow \\ N' \subseteq N \wedge E' \subseteq E \wedge \left(\forall n \in N' : \Delta'(n) \subseteq \Delta(n)\right). \tag{15a}$$

It is obvious that this notion of inclusion corresponds to the inclusion of the sets of tree diagrams encoded by the DAGs.

In the same fashion, we can define union and intersection for the DAGs $\mathbf{D}_i = (N_i, E_i, \Delta_i)$

$$\mathbf{D}_1 \cup \mathbf{D}_2 = (N_1 \cup N_2, E_1 \cup E_2, \Delta_1 \cup \Delta_2) \tag{16a}$$

$$\mathbf{D}_1 \cap \mathbf{D}_2 = (N_1 \cap N_2, E_1 \cap E_2, \Delta_1 \cap \Delta_2) \tag{16b}$$

where

$$\Delta_1 \cup \Delta_2 = \{n \mapsto \Delta_1(n) \cup \Delta_2(n) | n \in N_1 \cap N_2\} \\ \cup \{n \mapsto \Delta_1(n) | n \in N_1 \setminus N_2\} \\ \cup \{n \mapsto \Delta_2(n) | n \in N_2 \setminus N_1\} \tag{16c}$$

and in

$$\Delta_1 \cap \Delta_2 = \big\{n \mapsto \Delta_1(n) \cap \Delta_2(n) \big| n \in N_1 \cap N_2 \\ \wedge (\Delta_1(n) \cap \Delta_2(n) \neq \emptyset \vee n \in L)\big\} \tag{16d}$$

I am careful to avoid adding new leaf nodes to the intersection.

From these definitions, it is obvious that $\subseteq$ turns $\mathcal{D}_L$ into a *partially ordered set* and $\cup$ and $\cap$ turn it into a *lattice*. From this point of view, $\mathbf{D}_1 \cup \mathbf{D}_2$ is the least common upper bound of $\mathbf{D}_1$ and $\mathbf{D}_2$, while $\mathbf{D}_1 \cap \mathbf{D}_2$ is their greatest common lower bound. Finally $\mathcal{D}_L$ is bounded from below, with

$$\perp_L = (L, E, \{n \to \emptyset | n \in L\}) \tag{17}$$

as the bottom element.

3.3 Mapping and folding DAGs

The most important functions for manipulating DAGs and extracting the information encoded by them are *folds* that perform a nested application of a suitable function for all nodes to a starting value x

$$\Phi_f((N, E, \Delta), x) = f_{\delta_{|N|}} \cdots f_{\delta_2} f_{\delta_1} x, \tag{18}$$

where the elements of $\Delta = \{\delta_{n_1}, \delta_{n_2}, \ldots, \delta_{n_{|N|}}\}$ are arranged in the partial order of the nodes that guarantees acyclicity of the DAG. The only constraint on the function

$$f : (N \to E \times 2^{C(N)}) \times X \to X \\ (\delta, x) \mapsto f_\delta(x) \tag{19}$$

is that the domain and codomain of $f_\delta : X \to X$ must be identical. The computational cost scales with the size of the DAG and not with the size of the forest of tree diagrams described by it.

Used with the constructor ω (12) on the empty DAG, the fold performs a complete copy of any DAG $\mathbf{D}$

$$\Phi_\omega(\mathbf{D}, \epsilon) = \mathbf{D}. \tag{20}$$

Precomposing the first argument of ω in (20) with a function

$$f : (N \to 2^{E \times C}) \to (N \to 2^{E \times C}) \tag{21}$$

in the first argument using the notation

$$(\omega \circ f)_\delta = \lambda_{f(\delta)} \tag{22}$$

maps a DAG $\mathbf{D}$ to a new DAG $\mathbf{D}'$

$$\Phi_{\omega \circ f}(\mathbf{D}, \epsilon) = \mathbf{D}' \tag{23}$$

which can encode a different set of tree graphs.

The precomposition (22) can naturally be extended to functions mapping nodes to sets of nodes

$$f : (N \to 2^{E \times C}) \to 2^{(N \to 2^{E \times C})} \\ \delta \mapsto \{f_1(\delta), \ldots, f_k(\delta)\} \tag{24}$$

as

$$\omega_{f(\delta)} = \omega_{f_k(\delta)} \ldots \omega_{f_1(\delta)} \tag{25}$$

with the identity

$$\omega_\emptyset \mathbf{D} = \mathbf{D} \tag{26}$$

iff the result of f is the empty set $\emptyset$.

Finally, I define a function

$$H : (S, \mathbf{D}) \mapsto \mathbf{D}' \subseteq \mathbf{D} \tag{27}$$

that takes a set $S \subseteq N$ of nodes and a DAG and returns the minimal DAG that contains all the nodes in the set such that the mutually recursive evaluation of the functions Δ^* and $\hat{\Delta}$ from (7) is well defined for the nodes in this set. Intuitively, this corresponds to following all chains of arrows in $\{n \to \Delta(n)|n \in N\}$ from $\mathbf{D}$ that start in S.

3.4 Bundles

I am interested in maps between DAGs that respect certain structures. In order to describe these concisely, I borrow the notion of bundles from topology and differential geometry.

A *bundle* $\mathbf{B} = (X, B, \pi)$ is a triple consisting of a set X, called the *total set*, a set B, called the *base*, and a *projection* $\pi : X \to B$. The preimages $\pi^{-1}(b) \subseteq X$ are called *fibers*. The notation $\pi^{-1} : B \to 2^X$ must of course not be misunderstood as the inverse of π. The fibers are pairwise disjoint and their union

$$X = \bigsqcup_{b\in B} \pi^{-1}(b) \tag{28}$$

reproduces the set X. A *section* is a map $s : B \to X$ for which $\pi \circ s : B \to B$ is the identity. It corresponds to choosing one and only one element from each fiber. This definition generalizes the trivial bundle

$$\mathbf{B}_{\text{trivial}} = (B \times F, B, \pi) \tag{29a}$$

with

$$\pi(b, x) = b \tag{29b}$$

$$\pi^{-1}(b) = (b, F) \tag{29c}$$

where all fibers are trivially isomorphic to F and a section is the parameterized graph $s : B \to B \times F$ of a function $B \to F$.

Bundles formalize equivalence relations on the set X, with the base B as the set of all equivalence classes and π the canonical projection of an element of X to its equivalence class. The composition $\pi^{-1} \circ \pi : X \to 2^X$ maps each element to the set of the members of its equivalence class. Sections correspond to choosing one element from each equivalence class. An illustrative example is equivalence of nodes up to color quantum numbers, where π corresponds to ignoring color. Flavor, coupling constant and loop expansion order can be treated in the same way.

Bundles can be arranged in a sequence

$$B_0 \xleftarrow{\pi_1} B_1 \xleftarrow{\pi_2} B_2 \xleftarrow{\pi_3} \cdots . \tag{30}$$

However, since the preimage π_i^{-1} is not the inverse of the projection π_i, the preimage of a composition of projections is not the composition of the individual preimages, but

$$(\pi_i \circ \pi_{i+1})^{-1}(b) = \cup_{x\in\pi_i^{-1}(b)} \pi_{i+1}^{-1}(x) \tag{31}$$

instead.

As in the case of DAGs, such structures and the operations on them can be implemented for finite sets X straightforwardly in functional programming languages with static type systems and garbage collection (cf. Appendix A.2). In particular, it is efficient to add elements to the set X and update the base B and maps π and π^{-1} immediately. This allows to grow a bundle simultaneously while building a new DAG in order to maintain the relationships to be introduced in Sect. 3.5.

3.5 Projections and preimages of DAGs

Given a DAG $\mathbf{D} = (N, E, \Delta)$, where the set of nodes N is also the total set in a bundle $\mathbf{B} = (N, B, \pi)$, it is natural to ask if there is a canonical DAG $\mathbf{D}' = (B, E', \Delta')$ with the base of $\mathbf{B}$ as its set of nodes.

First, we observe that every section s of $\mathbf{B}$ and map $f : E \to E'$ defines a projected DAG $\mathbf{D}_{s,f} = (B, E', \Delta_{s,f})$ with

$$\begin{aligned} \Delta_{s,f} : B &\to 2^{E'\times C(B)} \\ b &\mapsto \hat{\pi_f}(\Delta(s(b))) \end{aligned} \tag{32a}$$

where $\hat{\pi_f}$ is the distribution of π over the nodes together with the application of f to the edges

$$\hat{\pi_f}(e, \{n_i|i \in I\}) = (f(e), \{\pi(n_i)|i \in I\}). \tag{32b}$$

The formula (32) has to be augmented by the prescription that a b for which $s(b)$ is a leaf node in $\mathbf{D}$ and therefore $\Delta_{s,f}(b) = \emptyset$ is *not* added as a leaf node to $\mathbf{D}_{s,f}$, similar to the definition (16d) of the intersection of two DAGs.

In most cases $f : E \to E'$ will be a simple projection that in our applications will be determined straightforwardly by the two sets of Feynman rules governing the construction of the two DAGs. Therefore we can write $\mathbf{D}_s$ instead of the more explicit $\mathbf{D}_{s,f}$.

The dependence of this projection on the section s is not satisfactory. However, the DAG

$$\Pi(\mathbf{D}) = \bigcup_{s\in S(\mathbf{B})} \mathbf{D}_s, \tag{33}$$

where $S(\mathbf{B})$ denotes the set of all sections of the bundle $\mathbf{B}$, is well defined and will be shown to suit our needs. Observe that the union is the correct universal construction for our applications, because the additional quantum numbers in N lead to more selection rules. These selection rules are the reason for the dependency of $\mathbf{D}_s$ on s. The DAG corresponding to the more basic set of nodes B should therefore be the combination of all possibilities. As an example consider the scattering of two scalars without and with flavor. Without flavor, there will be s-, $t-$ and u-channel diagrams. With a conserved flavor, only one of them will remain.

Note however, that this construction does *not* guarantee that the set of nodes of the DAG $\Pi(\mathbf{D})$ is actually the full base B of the bundle $\mathbf{B}$. We must therefore demand in addition compatibility of DAG and bundle, by requiring that the diagram

$$\begin{tikzcd} B & N \arrow[l, "\pi"'] \\ \Pi(\mathbf{D}) \arrow[u, "\nu"] & \mathbf{D} \arrow[l, "\Pi"'] \arrow[u, "\nu"'] \end{tikzcd} \tag{34}$$

commutes. The function ν in the commuting square (34) just extracts the set of nodes from a DAG

$$\nu(N, E, \Delta) = N. \tag{35}$$

The objects in the commuting square (34) can be understood as a combination of a pair of DAGs and a bundle, which I will call a *fibered DAG*. In programs, nodes can be added to the DAG $\mathbf{D}$ and the bundle in concert such that the relationship (34) is maintained.

An immediate benefit of such an universal construction of the projection is that it provides a corresponding preimage Π^{-1} which maps DAGs with the base B as nodes to all DAGs with the set N as nodes. The maps in the preimage can be written

$$\begin{aligned} \Delta^{s,f} : N &\to 2^{E\times C(N)} \\ n &\mapsto \hat{s}^f(\Delta(\pi(n))) \end{aligned} \tag{36}$$

where $\hat{s}^f$ is to be understood as the distribution of s over the nodes together with the application of f to the edges. Unfortunately, in contrast to (32), there will not be a single function $f : E \to E'$. Instead, we must allow that $\hat{s}^f$ maps into the powerset $2^{E'\times C(N)}$ instead of $E' \times C(N)$. In addition, the image of f will depend, via the Feynman rules, on the nodes appearing as children.

Since the resulting notation would be unnecessarily cumbersome, I will refrain from making the nature of f in (36) explicit as a function by specifying its domain and writing out all of its arguments. Nevertheless, the discussion of the example in Sect. 4.2 will demonstrate how a set of Feynman rules defines the maps $\Delta^{s,f}$ unambiguously.

In this picture, the application of Feynman rules amounts to choosing a particular element of the preimage Π^{-1}. It would however be extremely wasteful to construct the preimage first and to throw away all but one of its elements later. In Sect. 4.2, I will describe an algorithm that can be used to construct the desired element directly.

So far, I have assumed that the DAGs are selected by Feynman rules that are local to each vertex in the case of Feynman diagrams or to each element δ_i of the map Δ in our DAGs individually. There are however important exceptions. The most important is provided by loop expansions. There it is required for consistency that counterterms are inserted a fixed number of times in Feynman diagrams. Such conditions on complete Feynman diagrams do not translate immediately to the DAGs, whose components can enter the scattering amplitudes (4) more than once. Fortunately, this problem can be solved by introducing additional unphysical labels representing loop orders to the physical labels of the nodes and to select the required combinations of wavefunctions in (4) at the end, as will be described in Sect. 4.4. The same applies to selecting fixed orders in the perturbative expansions, as required for comparing to many results from the literature.

We call two DAGs $\mathbf{D}_1 = (N_1, E_1, \Delta_1)$ and $\mathbf{D}_2 = (N_2, E_2, \Delta_2)$ equivalent with respect to a pair of bundles $\mathbf{B}_1 = (N_1, B, \pi_1)$ and $\mathbf{B}_2 = (N_2, B, \pi_2)$ with the same base B iff there is a common projected DAG $\mathbf{D}$

$$\Pi_1(\mathbf{D}_1) = \mathbf{D} = \Pi_2(\mathbf{D}_2). \tag{37}$$

In this case $\mathbf{D}_1$ and $\mathbf{D}_2$ can be viewed as refinements of the same basic DAG $\mathbf{D}$. This notion of equivalence generalizes the notion of *topological equivalence* for diagrams, where two diagrams are considered equivalent if they agree after stripping off all quantum numbers. With the new notion of equivalence, we can say that the sets of Feynman diagrams encoded in a DAG are equivalent up to flavor or upto color.

Using the basic commuting square (34), we can immediately extend the bundle complex (30) to include the corresponding DAGs

$$\begin{tikzcd} B_0 & B_1 \arrow[l, "\pi_1"'] & B_2 \arrow[l, "\pi_2"'] & \cdots \arrow[l, "\pi_3"'] \\ \mathbf{D}_0 \arrow[u, "\nu"] & \mathbf{D}_1 \arrow[l, "\Pi_1"'] \arrow[u, "\nu"] & \mathbf{D}_2 \arrow[l, "\Pi_2"'] \arrow[u, "\nu"] & \cdots \arrow[l, "\Pi_3"'] \end{tikzcd} . \tag{38}$$

In our applications, this complex does not continue further to the left, because for each number of leaf nodes there is a natural leftmost nontrivial DAG $\mathbf{D}_\mathrm{P}$, described in Sect. 4.1 below.

In the following Sect. 4 I will describe how to use Feynman rules to walk the lower row of (38) to construct a DAG for a scattering amplitude efficiently in stages.

4 DAGs from Feynman rules

In principle, it is possible to construct the DAG encoding all Feynman diagrams in a single step.

First one adds leaf nodes for external states, labeled by all quantum numbers (momentum, spin/polarization, flavor, color, …). Which states are to be included here depends on the choice of algorithm, as has been discussed in Sect. 2.2.

Then one uses the Feynman rules of the model to add all nodes where the node and its children correspond to an allowed vertex. This proceeds iteratively: in the first step all subsets of the leaf nodes appear as children. In the following steps subsets of all nodes, including the leaf nodes appear as children subject to the constraint that no leaf node appears twice if the DAG is expanded recursively with the functions Δ^* and $\hat{\Delta}$ from (7). This iteration will terminate after a finite number of steps when all leaf nodes have been combined in all possible ways. While this algorithm inserts nodes that will not appear in the scattering amplitude the function H (27) can be used to harvest the minimal DAG.

This is a workable approach, but it is neither the most efficient nor particularly maintainable in actual code. Since the nodes are labeled by all quantum numbers, handling them all at once requires the construction of many nodes that will not appear in the final result. Adding quantum numbers in several stages instead allows us to use the constraints from earlier simpler stages to avoid in later stages the construction of many more complicated nodes that will never be used. While not relevant for the final numerical code, experience with early versions of O'Mega [6,7] revealed that the latter approach requires noticeably less time and memory for constructing the code.

Breaking up the construction of the DAG into several stages also simplifies the implementation of each stage and allows separate testing and swapping of different implementations. Finally, applications often need access to projected DAGs as described in Sect. 3.5 anyway. A prominent example is the construction of phase space parameterizations that only refer to kinematical information, such as propagators and masses.

Some of the stages described in the following subsections will be performed in a particular order, while the order of others can be interchanged easily.

4.1 Momenta

An element of the set N_{P} of nodes in the first DAG $\mathbf{D}_{\mathrm{P}} = (N_{\mathrm{P}}, \emptyset, \Delta_{\mathrm{P}})$ to be constructed is uniquely labeled by a subset of the powerset $2^{\{1,2,\ldots,n\}}$ of labels for the external momenta and the edges are unlabeled. The leaf nodes are the elements $n(\{i\})$ of N_{P} and the action of the map Δ_{P} is given by

$$n(I) \mapsto \Big\{ (\emptyset, \{n(I_i) | 1 \leq i \leq k\}) \Big| 2 \leq k \leq l-1 \\ \wedge \cup_{i=1}^{k} I_i = I \wedge I_i \neq \emptyset \Big\} \tag{39}$$

where l is the maximum number of legs of the vertices in the model. Obviously, we can order the nodes $n(I)$ according to the number of elements of I to prove that there are no cycles in $\mathbf{D}_{\mathrm{P}}$.

In case of (4a), we only need the elements of $2^{\{2,\ldots,n\}}$ as labels. In the cases (4b) and (4c), only labels with at most $n/2$ elements are needed. Finally, the function H (27) is applied to construct the minimal DAG required for evaluating one of the expressions (4).

4.2 Flavors and Lorentz structures

In the next stage, the momenta of the leaf nodes of $\mathbf{D}_{\mathrm{P}}$ are combined with the flavor quantum numbers of the corresponding external state. The resulting leaf nodes form the starting point of a new DAG $\mathbf{D}_{\mathrm{F}} = (N_{\mathrm{F}}, V_{\mathrm{F}}, \Delta_{\mathrm{F}})$ and bundle $\mathbf{B}_{\mathrm{F}} = (N_{\mathrm{F}}, N_{\mathrm{P}}, \pi_{\mathrm{F}})$. The edges V_{F} are vertex factors consisting of coupling constants, Lorentz tensors and Dirac matrices.

Using a fold Φ of $\mathbf{D}_{\mathrm{P}}$ using the constructor ω of $\mathbf{D}_{\mathrm{F}}$ with precomposition (23) that maintains the fibration (34) will ensure that the nodes of $\mathbf{D}_{\mathrm{P}}$ are visited in the correct order of growing label sets. The function f that is precomposed to ω in (23) acts on each element

$$n(I) \mapsto (\emptyset, \{n(I_i) | 1 \leq i \leq k\}) \tag{40}$$

of the map Δ_{P} as follows: since the $n(I_i) \in N_{\mathrm{P}}$ have been processed, they are elements of the base of the growing bundle $\mathbf{B}_{\mathrm{F}}$. Therefore, the fibers $\pi_{\mathrm{F}}^{-1}(n(I_i))$ are already complete and we can compute their cartesian product

$$\Gamma = \pi_{\mathrm{F}}^{-1}(n(I_1)) \times \pi_{\mathrm{F}}^{-1}(n(I_2)) \times \cdots . \tag{41}$$

We then use the Feynman rules to select all elements of Γ that can be combined with another flavor to obtain a valid vertex. This defines a function $\Gamma \mapsto 2^{V_{\mathrm{F}}}$. For each of the resulting flavors, a new node labeled by I and this flavor is added to $\mathbf{D}_{\mathrm{F}}$ and $\mathbf{B}_{\mathrm{F}}$ together with the corresponding vertex factors and elements of Γ as edges and children, maintaining the fibration (34).

This algorithm has been implemented in O'Mega [6,7] and is completely independent of the kind of Feynman rules. It can accommodate both hardcoded rules and rules derived from a UFO file [3,4]. The only potential performance bottleneck is the efficient matching of vertices to the elements of a Γ representing a large number of children. For vertices with few legs, this is not a practical issue, but care has to be taken for vertices with many legs where the factorial growth of the number of permutations might be felt.

Once the flavors have been assigned, it is known which fermion lines contribute in the computation of each node. This information must also be added to the node in order to be able to assign the correct sign to interfering contributions in (4) later. Special care must be taken if the model contains Majorana fermions [27–29].

By construction, after the fold is complete, the new DAG $\mathbf{D}_\mathrm{F}$ encodes all the information needed to compute the scattering amplitude for the leaf nodes in a theory without color, using one of the formulae (4). Some nodes in $\mathbf{D}_\mathrm{F}$ might not be needed due to conserved quantum numbers. Therefore the function H (27) from $\mathbf{D}_\mathrm{F}$ is applied again to construct the minimal DAG required to evaluate one of the expressions (4).

4.3 Colors

Since the color representation depends on the flavor, the assignment of color quantum numbers in the construction of the DAG $\mathbf{D}_\mathrm{C} = (N_\mathrm{C}, V_\mathrm{C}, \Delta_\mathrm{C})$ naturally comes after the construction of $\mathbf{D}_\mathrm{F}$.

We can now follow the steps of the previous stage, as described in Sect. 4.2, word for word, only replacing the subscripts (F, P) by (C, F). The implementation in O'MEGA uses the realization of the color flow basis described in [30], but, except for the labeling of the nodes in N_C, the form of the vertices in V_C and the Feynman rules to be used, the algorithm is completely independent of the representation of the color algebra.

Having the color information available algebraically allows to compute color factors and color correlators [5] analytically.

4.4 Coupling orders

As already mentioned in Sect. 3.5, there are cases where it is important that the Feynman diagrams encoded by the DAG contain certain coupling constants with fixed powers. The most important examples are the counterterms and the terms of an effective action in a loop expansion. Also the inclusion of self energy type terms will not terminate in a DAG, unless a finite maximum expansion order is prescribed.

For practical purposes it is sometimes also important to compute only a part of a scattering amplitude corresponding to fixed powers of couplings. Such results are often available in the literature from Feynman diagram based calculations and a comparison for the purpose of validation is only possible if the DAG based calculation can select exactly the same contributions.

A priori, this conflicts with the representations (4) of scattering amplitudes as DAGs, since the wavefunctions or currents will have accumulated different powers of couplings that will be mixed by (4).

Fortunately, there is a simple solution. For example, in the case of (4b) we can write

$$\mathcal{M}_o(\{1,2,\ldots,n\}) = \sum_{\substack{I_1\cup I_2\cup I_3=\{1,\ldots,n\}\\ o_1+o_2+o_3=o}} K_{I_1,I_2,I_3}\phi_{o_1}(I_1)\phi_{o_1}(I_2)\phi_{o_1}(I_3) \tag{42}$$

to compute the scattering amplitude at the coupling order o. The only change required is that the wavefunctions have to keep track of the coupling orders accumulated in their recursive computation. Since the powers of the couplings are additive, we never have to add the wavefunctions or currents that exceed the requested order to the DAG.

This necessitates augmenting the set of labels of the nodes by unphysical "quantum numbers" corresponding to the coupling orders. It can be implemented easily, as long as the number of coupling orders to be tracked remains moderate.

4.5 Skeleton expansion

If we are using DAGs to efficiently implement a skeleton expansion, the remarks in Sect. 4.4 apply word for word by replacing "coupling order" by "loop order".

4.6 Multiple amplitudes

In practical applications [7], it is usually necessary to compute scattering amplitudes for the same external momenta, but more than one combination of flavors and colors at the same time. These flavor and color combinations often overlap pairwise and the sets of leaf nodes will also overlap, i.e. $L_1 \cap L_2 \neq \emptyset$. In this case, it is efficient to combine the corresponding DAGs $\mathbf{D}_{L_1}$ and $\mathbf{D}_{L_2}$ into a single DAG and to compute the scattering amplitudes from this DAG in order to reuse nodes from the part of the DAG build on $L_1 \cap L_2$. For this purpose, we can generalize the union defined in (16c) to a map

$$\cup : \mathcal{D}_{L_1} \times \mathcal{D}_{L_2} \to \mathcal{D}_{L_1\cup L_2} \tag{43}$$

in an obvious way.

5 Code generation

The example (8) can be translated directly into, e.g. Fortran, as

```
  w1  = phi(p1)
  w2  = phi(p2)
  w3  = phi(p3)
  w4  = phi(p4)
  p5  = p1 + p2
  w5  = prop(p5)*g*w1*w2
  p6  = p5 + p3
  w6  = prop(p6)*g*w5*w3
  p7  = p5 + p4
  w7  = prop(p7)*g*w5*w4
  p8  = p6 + p4
! p8  = p7 + p3
  w8  = prop(p7)*(g*w6*w4 + g*w7*w3)
```

where `pn` and `wn` denote fourmomenta and wavefunctions, respectively. `phi()` computes external wavefunctions, `prop()` propagators and `g` is a coupling constant. Using overloaded operators $+$, $-$ and $*$ allows to write similarly concise and readable code for realistic models with standard model quantum numbers. In the case of more general models, functions implementing the vertex factors can be generated from UFO files [3,4].

Identically structured code can be emitted as bytecode for a virtual machine that realizes the operators as basic instructions [21]. The improving memory bandwidth for graphical processing units even allows to start targeting GPUs for interesting examples.

As already mentioned in the introduction, the generation of robust numerical code is much more challenging if the DAG encodes diagrams that contain loops. The problem has been solved for the standard model [8–11]. The structures described in the paper will help with the task of extending this approach to general models.

6 Conclusions

I have described the algebraic structures that organize recursive calculations in perturbative quantum field theory without the need to expand intermediate expressions into Feynman diagrams. In functional programming languages, these algebraic structures translate directly into data structures. In a second step, these data structures are translated to efficient numerical code for any programming language or hardware target required.

This algebraic approach adds flexibility over purely numeric implementations tied to specific models and computing targets. It allows for more extensive consistency checks and paves the way for more challenging applications.

Acknowledgements I thank Wolfgang Kilian, Jürgen Reuter and the other members of the WHIZARD team for the decades long productive collaboration. This work is supported by the German Federal Ministry for Education and Research (BMBF) under contract no. 05H21WWCAA.

Data Availability Statement This manuscript has no associated data or the data will not be deposited. [Authors' comment: As a theoretical work, this manuscript has no associated data.]

Funded by SCOAP³. SCOAP³ supports the goals of the International Year of Basic Sciences for Sustainable Development.

Appendix A: Implementation

A.1 DAGs

Here is the relevant subset of the `ocaml` signature [26] of the DAG module in O'MEGA [6], implementing the functions from Sects. 3.1 and 3.3. For flexibility, this module is implemented as a functor application on the types `node`, `edge` and `children`, corresponding to N, E and $C(N)$ respectively

```
module type DAG = sig
  type node
  type edge
  type children
  type t
  val empty : t
  val add_node : node -> t -> t
  val add_offspring :
    node -> edge * children -> t -> t
  val fold_nodes :
    (node -> 'a -> 'a) -> t -> 'a -> 'a
  val fold :
    (node -> edge * children -> 'a -> 'a)
    -> t -> 'a -> 'a
  val harvest : t -> node -> t -> t
end
```

Here `type` declares an abstract data type and `val` declares values and functions, the latter just being values in a functional programming language. The type `'a` is polymorphic. The actual signature in O'MEGA contains additional convenience functions that can be build from the functions presented here.

Note that this implementation breaks the function ω (12) into products of functions ω^0 and ω^1, with

$$\omega_{n\mapsto\emptyset} = \omega^0_n \tag{A1a}$$

$$\omega_{n\mapsto\{(e_1,c_1),\ldots,(e_k,c_k)\}} = \prod_{i=1}^{k} \omega^1_{n\mapsto(e_i,c_i)} \,. \tag{A1b}$$

The function ω^0 (called `add_node` here) can be used to construct $\perp_L \in \mathcal{D}_L$ from $\epsilon \in \mathcal{D}_\emptyset$, while the action of ω^1 (called `add_offspring` here), does not leave the category $\mathcal{D}_L$. This provides a better interface for programming, but the ω used in the main part of the paper allowed a more concise writeup of the mathematical structures in Sect. 3.

Correspondingly, the fold Φ from (18) is broken into `fold_nodes` processing all nodes and `fold` processing all $N \to E \times C$ mappings element-by-element. The `ocaml` equivalent of (20) is then

```
  let leaves' =
    fold_nodes add_node dag empty in
  fold add_offspring dag leaves'
```

Note that this gives up some generality, because the Φ from (18) could process the sets of $E \times C$ as a whole and not only element-by-element. However, this interface is more straightforward and is better tailored to our applications.

The function `harvest` implements H (27). In particular, `harvest dag n dag'` finds the subset of the DAG `dag` that is reachable from the node `n` and adds it to the DAG `dag'`. This way, applications can compute a minimal DAG for further processing.

Since the construction of the DAG $\mathbf{D}_\mathrm{P}$ (cf. Sect. 4.1) is very simple, it had been combined with the construction of $\mathbf{D}_\mathrm{F}$ (cf. Sect. 4.2) in O'MEGA [6] before the structures described in this paper were elaborated. However, the separation of the remaining stages described in Sect. 4 forms the backbone of the current version of O'MEGA.

A.2 Bundles

Here is the signature of the `Bundle` module in O'MEGA [6]. Again a functor is applied to the types `elt`, `base` and the function `pi`, corresponding to X, B and π respectively

```
module type Bundle = sig
  type elt
  type base
  val pi : elt -> base
  type t
  val empty : t
  val add : t -> elt -> t
  val inv_pi : t -> base -> fiber
  val base : t -> base list
end
```

The semantics of the functions is evident from the discussion of bundles in Sect. 3.4. Note that π is universal for all bundles with this type, while π^{-1} depends on the elements added to the bundle previously.

References

1. T. Stelzer, W.F. Long, Automatic generation of tree level helicity amplitudes. Comput. Phys. Commun. **81**, 357–371 (1994). https://doi.org/10.1016/0010-4655(94)90084-1. arXiv:hep-ph/9401258
2. J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, T. Stelzer, MadGraph 5: going beyond. JHEP **06**, 128 (2011). https://doi.org/10.1007/JHEP06(2011)128. arXiv:1106.0522 [hep-ph]
3. C. Degrande, C. Duhr, B. Fuks, D. Grellscheid, O. Mattelaer, T. Reiter, UFO—the universal FeynRules output. Comput. Phys. Commun. **183**, 1201–1214 (2012). https://doi.org/10.1016/j.cpc.2012.01.022. arXiv:1108.2040 [hep-ph]
4. L. Darmé et al., UFO 2.0—the Universal Feynman Output format (2023). https://doi.org/10.1140/epjc/s10052-023-11780-9. arXiv:2304.09883 [hep-ph]
5. S. Catani, M.H. Seymour, A general algorithm for calculating jet cross-sections in NLO QCD. Nucl. Phys. B **485**, 291–419 (1997). [Erratum: Nucl. Phys. B 510, 503–504 (1998)]. https://doi.org/10.1016/S0550-3213(96)00589-5. arXiv:hep-ph/9605323
6. M. Moretti, T. Ohl, J. Reuter, O'Mega: An Optimizing Matrix Element GenerAtor. In: Proceedings of the 2nd Workshop of the 2nd Joint ECFA/DESY Study on Physics and Detectors for a Linear Electron Positron Collider, pp. 1981–2009 (2001). arXiv:hep-ph/0102195
7. W. Kilian, T. Ohl, J. Reuter, WHIZARD: simulating multi-particle processes at LHC and ILC. Eur. Phys. J. C **71**, 1742 (2011). https://doi.org/10.1140/epjc/s10052-011-1742-y. arXiv:0708.4233 [hep-ph]
8. F. Cascioli, P. Maierhofer, S. Pozzorini, Scattering amplitudes with open loops. Phys. Rev. Lett. **108**, 111601 (2012). https://doi.org/10.1103/PhysRevLett.108.111601. arXiv:1111.5206 [hep-ph]
9. F. Buccioni, J.N. Lang, J.M. Lindert, P. Maierhöfer, S. Pozzorini, H. Zhang, M.F. Zoller, OpenLoops 2. Eur. Phys. J. C **79**(10), 866 (2019). https://doi.org/10.1140/epjc/s10052-019-7306-2. arXiv:1907.13071 [hep-ph]
10. S. Actis, A. Denner, L. Hofer, A. Scharf, S. Uccirati, Recursive generation of one-loop amplitudes in the standard model. JHEP **04**, 037 (2013). https://doi.org/10.1007/JHEP04(2013)037. arXiv:1211.6316 [hep-ph]
11. S. Actis, A. Denner, L. Hofer, J.N. Lang, A. Scharf, S. Uccirati, RECOLA: REcursive Computation of One-Loop Amplitudes. Comput. Phys. Commun. **214**, 140–173 (2017). https://doi.org/10.1016/j.cpc.2017.01.004. arXiv:1605.01090 [hep-ph]
12. A. Denner, S. Dittmaier, L. Hofer, COLLIER: a fortran-based Complex One-Loop LIbrary in Extended Regularizations. Comput. Phys. Commun. **212**, 220–238 (2017). https://doi.org/10.1016/j.cpc.2016.10.013. arXiv:1604.06792 [hep-ph]
13. F.A. Berends, W.T. Giele, Recursive calculations for processes with n gluons. Nucl. Phys. B **306**, 759–808 (1988). https://doi.org/10.1016/0550-3213(88)90442-7
14. A. Kanaki, C.G. Papadopoulos, HELAC: a package to compute electroweak helicity amplitudes. Comput. Phys. Commun. **132**, 306–315 (2000). https://doi.org/10.1016/S0010-4655(00)00151-X. arXiv:hep-ph/0002082
15. A. Cafarella, C.G. Papadopoulos, M. Worek, Helac-Phegas: a generator for all parton level processes. Comput. Phys. Commun. **180**, 1941–1955 (2009). https://doi.org/10.1016/j.cpc.2009.04.023. arXiv:0710.2427 [hep-ph]
16. F. Caravaglios, M. Moretti, An algorithm to compute born scattering amplitudes without Feynman graphs. Phys. Lett. B **358**, 332–338 (1995). https://doi.org/10.1016/0370-2693(95)00971-M. arXiv:hep-ph/9507237
17. F. Caravaglios, M. Moretti, e^+e^- into four fermions $+\gamma$ with ALPHA. Z. Phys. C **74**, 291–296 (1997). https://doi.org/10.1007/s002880050390. arXiv:hep-ph/9604316
18. F. Caravaglios, M.L. Mangano, M. Moretti, R. Pittau, A new approach to multijet calculations in hadron collisions. Nucl. Phys. B **539**, 215–232 (1999). https://doi.org/10.1016/S0550-3213(98)00739-1. arXiv:hep-ph/9807570
19. T. Gleisberg, S. Hoeche, Comix, a new matrix element generator. JHEP **12**, 039 (2008). https://doi.org/10.1088/1126-6708/2008/12/039. arXiv:0808.3674 [hep-ph]
20. A. Denner, J.N. Lang, S. Uccirati, Recola2: REcursive Computation of One-Loop Amplitudes 2. Comput. Phys. Commun. **224**, 346–361 (2018). https://doi.org/10.1016/j.cpc.2017.11.013. arXiv:1711.07388 [hep-ph]
21. B. Chokoufe. Nejad, T. Ohl, J. Reuter, Simple, parallel virtual machines for extreme computations. Comput. Phys. Com-

mun. **196**, 58–69 (2015). https://doi.org/10.1016/j.cpc.2015.05.015. arXiv:1411.3834 [physics.comp-ph]

22. A. Alboteanu, W. Kilian, J. Reuter, Resonances and unitarity in weak boson scattering at the LHC. JHEP **11**, 010 (2008). https://doi.org/10.1088/1126-6708/2008/11/010. arXiv:0806.4145 [hep-ph]
23. W. Kilian, T. Ohl, J. Reuter, M. Sekulla, High-energy vector boson scattering after the Higgs discovery. Phys. Rev. D **91**, 096007 (2015). https://doi.org/10.1103/PhysRevD.91.096007. arXiv:1408.6207 [hep-ph]
24. W. Kilian, T. Ohl, J. Reuter, M. Sekulla, Resonances at the LHC beyond the Higgs boson: the scalar/tensor case. Phys. Rev. D **93**(3), 036004 (2016). https://doi.org/10.1103/PhysRevD.93.036004. arXiv:1511.00022 [hep-ph]
25. C. Okasaki, *Purely Functional Data Structures* (Cambridge University Press, New York, 1998)
26. X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, K. Sivaramakrishnan, J. Vouillon, *The OCaml System, Release 5.0. Documentation and User's Manual* (Institut National de Recherche en Informatique et en Automatique, 2022). https://ocaml.org/manual/
27. A. Denner, H. Eck, O. Hahn, J. Küblbeck, Compact Feynman rules for Majorana fermions. Phys. Lett. B **291**, 278–280 (1992). https://doi.org/10.1016/0370-2693(92)91045-B
28. J. Reuter, Supersymmetry of scattering amplitudes and green functions in perturbation theory. PhD Thesis, Technische Universität Darmstadt, Germany (2002). arXiv:hep-th/0212154
29. T. Ohl, J. Reuter, Clockwork SUSY: supersymmetric ward and Slavnov–Taylor identities at work in green's functions and scattering amplitudes. Eur. Phys. J. C **30**, 525–536 (2003). https://doi.org/10.1140/epjc/s2003-01301-7. arXiv:hep-th/0212224
30. W. Kilian, T. Ohl, J. Reuter, C. Speckner, QCD in the color-flow representation. JHEP **10**, 022 (2012). https://doi.org/10.1007/JHEP10(2012)022. arXiv:1206.3700 [hep-ph]