How to Multiply Dynamic Programming Algorithms

Christian Höner zu Siederdissen[1], Ivo L. Hofacker[1−3], and Peter F. Stadler[4,1,3,5−7]

[1] Dept. Theoretical Chemistry, Univ. Vienna, Währingerstr. 17, Wien, Austria
[2] Bioinformatics and Computational Biology research group, University of Vienna, A-1090 Währingerstraße 17, Vienna, Austria
[3] RTH, Univ. Copenhagen, Grønnegårdsvej 3, Frederiksberg C, Denmark
[4] Dept. Computer Science, and Interdisciplinary Center for Bioinformatics, Univ. Leipzig, Härtelstr. 16-18, Leipzig, Germany
[5] MPI Mathematics in the Sciences, Inselstr. 22, Leipzig, Germany
[6] FHI Cell Therapy and Immunology, Perlickstr. 1, Leipzig, Germany
[7] Santa Fe Institute, 1399 Hyde Park Rd., Santa Fe, USAx

Abstract. We develop a theory of algebraic operations over linear grammars that makes it possible to combine simple "atomic" grammars operating on single sequences into complex, multi-dimensional grammars. We demonstrate the utility of this framework by constructing the search spaces of complex alignment problems on multiple input sequences explicitly as algebraic expressions of very simple 1-dimensional grammars. The compiler accompanying our theory makes it easy to experiment with the combination of multiple grammars and different operations. Composite grammars can be written out in LaTeX for documentation and as a guide to implementation of dynamic programming algorithms. An embedding in Haskell as a domain-specific language makes the theory directly accessible to writing and using grammar products without the detour of an external compiler.
`http://www.bioinf.uni-leipzig.de/Software/gramprod/`

Key words: linear grammar, context free grammar, product structure, multiple alignment, Haskell

1 Introduction

The well-known dynamic programming algorithms for the simultaneous alignment of n sequences [1] have a structure that is reminiscent of topological product structures. This is expressed e.g. by the fact that intermediary tables are n-dimensional. Here we explore if and how this intuition can be made precise and operational. To this end we build on the conceptual framework of Algebraic Dynamic Programming (ADP) [2, 3]. In this setting a dynamic programming (DP) algorithm is separated into a context-free grammar (CFG) that generates the search space and an evaluation algebra. In this contribution we will mainly

be concerned with a notion of product grammars to facilitate the construction of the search space.

Before we delve into a more formal presentation, consider the context-free grammar for pairwise sequence alignment with affine gap costs as an example. Gotoh's algorithm [4] uses three non-terminals M, D, I, depending on whether the right end of the alignment is a match state, a gap in the first sequence, or a gap in the second sequence. The corresponding productions are of the form

$$\begin{aligned} M &\to M\binom{u}{v} \mid D\binom{u}{v} \mid I\binom{u}{v} \mid \binom{\varepsilon}{\varepsilon} \\ D &\to M\binom{u}{-} \mid D\binom{u}{\cdot} \mid I\binom{u}{-} \\ I &\to M\binom{-}{v} \mid D\binom{-}{v} \mid I\binom{\cdot}{v} \end{aligned} \tag{1}$$

where u and v denote terminal symbols, '$-$' corresponds to gap opening, while '.' denotes the (differently scored) gap extension. The ε here takes the role of the "sentinel character", i.e., matches the end of the input. Each of the non-terminals reads simultaneously from two separate input tapes. To make this property more transparent in the notation, we write $M \rightsquigarrow \binom{X}{X}$, $D \rightsquigarrow \binom{X}{Y}$, and $I \rightsquigarrow \binom{Y}{X}$. This yields productions such as

$$\binom{X}{X} \to \binom{X}{X}\binom{u}{v} \simeq \binom{Xu}{Xv} \qquad \text{or} \qquad \binom{Y}{X} \to \binom{X}{Y}\binom{-}{v} \simeq \binom{X-}{Yv} \tag{2}$$

Apart from the conspicous absence of $\binom{Y}{Y}$, i.e., alignments ending in an all-gap column, to which we will return later, this notation strongly suggests to consider the 1-dimensional projections of the 2-dimensional productions of Equ. (2), which obviously have the form

$$X \to Xu \mid Yu \mid \varepsilon \qquad \text{and} \qquad Y \to Y. \mid X- \tag{3}$$

This simple grammar either reads a symbol (non-terminal X) or it ignores it (non-terminal Y). Each copy of the "step grammar" (3) operates on its own input tape. The basic idea in this contribution is to consider the dynamic programming algorithms for n-way alignments as an n-fold product of the simple step grammar with itself. To this end we need to solve two problems: First, we need to clarify the precise meaning of the product of CFGs. Since alignment algorithms are naturally expressed as left-linear CFGs we will be content with this special case here. Second, we need to develop a theory for the construction of the evaluation algebra for a product grammar.

We note that full-fledged n-way DP alignments have exponential running time and hence are of little practical use for large n. Although elaborate divide & conquer strategies have been proposed to prune the search space, see e.g. [1], heuristic approaches that combine pairwise alignments are much more common. Three-way alignments nevertheless are employed in practise in particular when high accuracy is crucial, see e.g. [5–8]. Four-way alignments were recently explored for aligning short words from human language data [9].

2 Algebraic Operations on Grammars

2.1 Notation

A CFG $\mathcal{G} = (N, T, P, s)$ consists of a finite set N of non-terminals, a finite set T of terminals so that $N \cap T = \emptyset$, a set P of productions $x \to \alpha$ where $x \in N$

and $\alpha \in (T \cup N)^*$, and a start symbol $s \in N$. Furthermore, we need the special symbol ϵ denoting the empty string and an "empty production" $\varnothing$. Throughout the body of this contribution we will consider in particular left-linear grammars, i.e., those for which all productions are of the form $A \to Bx$ with $A, B \in N$ and $x \in T$.

The example of Gotoh's algorithm in the introductory section motivates us to introduce algebraic operations on grammars in a more systematic way. As a running example, we will use one of the simplest alignment algorithms. The Needleman-Wunsch algorithm [10] aligns two sequences $x_{1\ldots n}$ and $y_{1\ldots m}$ so that the sum of match and in/del scores is maximized. The basic recursion over the memoization table T reads

$$T_{ij} = \max\left\{T_{i+1,j} + d, T_{i,j+1} + d, T_{i+1,j+1} + m(x_i, y_j), 0_{i=n,j=m}, -\infty\right\} \tag{4}$$

In the recursive scheme, the base case is given by the alignment of two empty substrings "on the right", while the other cases extend the already aligned part of the strings to the left. This slightly unusual variant of the algorithm was chosen to be identical to the grammatical description that follows. The first two cases denote an in/del operation with cost d, while $m(\,.\,,\,.\,)$ scores the (mis)match x_i with y_j.

A two-tape grammar equivalent to the recursion in Equ. (4) is

$$\binom{X}{Y} \to \binom{X}{Y}\binom{a}{\epsilon} \;\Big|\; \binom{X}{Y}\binom{\epsilon}{a} \;\Big|\; \binom{X}{Y}\binom{a}{a} \;\Big|\; \binom{\varepsilon}{\varepsilon} \tag{5}$$

There are several differences between the formulation in Equ. (4) and Equ. (5). The recursive formulation working on the memoization table T does not store the alignment directly but rather the *score* of each partial, optimal alignment. The grammatical description, on the other hand, describes the *search space* of all possible alignments without any notation of scoring. In addition, recursive descriptions usually include explicit annotations for base cases, here the empty alignment. The production rule $\binom{X}{Y} \to \binom{\varepsilon}{\varepsilon}$ has this role in our example. In general, grammatical descriptions abstract away certain implementation details. Some of these will, however, become important when constructing more complex grammars from simpler ones, as we shall see below.

Our task will be to construct the Equ. (5) from even simpler, "atomic" constituents. These grammars are

$$\mathcal{S} = (\{X\}, \{a\}, \{X \to Xa \mid X\}, X) \tag{6}$$

$$\mathcal{N} = (\{X\}, \{\varepsilon\}, \{X \to \varepsilon\}, X) \tag{7}$$

$$\mathcal{L} = (\{X\}, \{\}, \{X \to X\}, X) \tag{8}$$

The grammar $\mathcal{S}$ in Equ. (6) performs a "step". It either reads a single character on the right and recurses on the left, or simply recurses. Note that by itself the rules do not terminate. The grammar $\mathcal{N}$, Equ. (7), matches the empty input (or any empty substring of the input) and immediately terminates. Finally, $\mathcal{L}$ Equ. (8) reproduces the non-terminating loop case already seen in Equ. (6).

Intuitively, we can combine these three components on a single tape as

$$\mathcal{S} + \mathcal{N} - \mathcal{L} = (\{X\}, \{a, \varepsilon\}, \{X \to Xa \mid \varepsilon\}, X) \tag{9}$$

In order to make this intuition precise we need to give a precise meaning to algebraic operations on grammars. In the following we will do this for linear grammars, however with an extension to general CFGs in mind.

Each operator introduced below primarily acts on sets of production rules. They implicitly carry over to the involved sets of terminals and non-terminals in an obvious manner. Two production rules are equivalent if they are isomorphic as in Equ. (13). This is of relevance insofar that it leads to idempotency in one of the operators below, but does not otherwise interfere with parsing[8]. In the following we use the notation P^n to emphasize that the productions operate on n tapes. We will refer to $\dim \mathcal{G} = n$ as the dimension of the grammar.

2.2 Algebraic Operations on Grammars

The + monoid. The + operator is defined as the union of all production rules of the two grammars:

$$P_1^n + P_2^n = P_1^n \cup P_2^n \tag{10}$$

We enforce explicitly that the + operator requires that the two operand grammars have the same dimensionality. The + operation forms a monoid over the set of production rules. Since the production rules form a set, isomorphic rules collapse to a single rule. The empty set $P^n = \{\}$ is a neutral element and $P^n + P^n = P^n$, i.e., the + monoid is idempotent. Isomorphism on production rules is also symbolic, that is, $X \to X$ is isomorphic to $X \to X$ but not to $\{X \to Y, Y \to X\}$, even though the latter set of two rules reduces to the first. For our example, we have $(X \to Xa \mid X) + (X \to \epsilon) = (X \to Xa \mid X \mid \varepsilon)$.

The − operator. While the + operator unifies two sets of production rules, the − operator acts as a set difference operator

$$P_1^n - P_2^n = \{p \in P_1^n | p \notin P_2^n\} \tag{11}$$

As for +, it requires operands of the same dimensionality. By construction, − is not associative. Thus does not form a semigroup but merely a magma. The empty set of production rules acts as the neutral element on the right. This operator is important to explictly remove production rules that yield infinite derivations. In our example, we need to remove $\{X \to X\}$. With the help of − we can write $(X \to Xa \mid X) - (X \to X) = (X \to Xa)$. We shall see that it is often convenient to "temporarily" introduce productions that later on need to be excluded from the final algorithm.

The ⊗ monoid. The definition of a direct product of left linear grammars lies at the heart of this contribution.

Definition 1. *Let $\mathcal{G}_1 = (N_1, T_1, P_1, s_1)$ and $\mathcal{G}_2 = (N_2, T_2, P_2, s_2)$ be left-linear CFGs, i.e., all productions are of the form $A \to Bx$ or $A \to y$. Their direct product $\mathcal{G}_1 \otimes \mathcal{G}_2$ is the grammar $\mathcal{G} = (N, T, P, s)$ with non-terminals $N = N_1 \times$*

[8] This is not completely true in the context of stochastic linear grammars: replication of a rule in an SCFG that already has duplicated rules requires that we sum over the probabilities for isomorphic rules.

$N_2 \cup N_1 \times \{\epsilon\} \cup \{\epsilon\} \times N_2$, terminals $T = T_1 \times T_2 \cup T_1 \times \{\epsilon\} \cup \{\epsilon\} \times T_2$, the start symbol of the product is $s = \binom{s_1}{s_2}$. The productions are of the forms $\binom{A_1}{A_2} \to \binom{B_1}{B_2}\binom{x_1}{x_2}$, $\binom{A_1}{A_2} \to \binom{B_1}{\epsilon}\binom{x_1}{y_2}$, $\binom{A_1}{A_2} \to \binom{\epsilon}{B_2}\binom{y_1}{x_2}$, $\binom{A_1}{A_2} \to \binom{y_1}{y_2}$, $\binom{A_1}{\epsilon} \to \binom{B_1}{\epsilon}\binom{x_1}{\epsilon}$, $\binom{\epsilon}{A_2} \to \binom{\epsilon}{B_2}\binom{\epsilon}{x_2}$, $\binom{A_1}{\epsilon} \to \binom{y_1}{\epsilon}$, and $\binom{\epsilon}{A_2} \to \binom{\epsilon}{y_2}$ iff $A_1 \to B_1 x_1$ and $A_1 \to y_1$, are productions in P_1 and $A_2 \to B_2 x_2$, $A_2 \to y_2$ are productions in P_2, respectively.

By construction $\mathcal{G}$ is again a left-linear CFG that now operates on two bands. It will be convenient to abuse the notation and write productions of the form $A_i \to y_i$ as $A_i \to \epsilon y_i$. Hence all productions in the product grammar can be written as $\binom{A_1}{A_2} \to \binom{B_1}{B_2}\binom{x_1}{x_2}$ with $A_i, B_i \in N_i \cup \{\epsilon\}$, $x_i \in T_i \cup \{\epsilon\}$ subject to the following conditions: $A_i = \epsilon$ implies $B_i = x_i = \epsilon$, $\binom{A_1}{A_2} \neq \binom{\epsilon}{\epsilon}$, and $\binom{\epsilon}{\epsilon}$ on the r.h.s. is omitted. We will also make use of notation $(A_1 \to B_1 y_1) \otimes (A_2 \to B_2 y_2)$ for the product of two individual productions. By construction, we have

$$\dim(\mathcal{G}_1 \otimes \mathcal{G}_2) = \dim \mathcal{G}_1 + \dim \mathcal{G}_2 \tag{12}$$

We note finally, that the empty string ϵ appearing in the 2-dimensional terminals and non-terminals is not necessarily associated with terminating the reading from the input band(s).

To see that $\otimes$ is associative we need to demonstrate that the productions on $(\mathcal{G}_1 \otimes \mathcal{G}_2) \otimes \mathcal{G}_3$ and $\mathcal{G}_1 \otimes (\mathcal{G}_2 \otimes \mathcal{G}_3)$ are isomorphic, i.e.,

$$\left(\binom{x_1}{x_2}\right) \to \left(\binom{\alpha_1}{\alpha_2}\right) \qquad \simeq \qquad \left(\binom{x_1}{x_2}\right) \to \left(\binom{\alpha_1}{\alpha_2}\right) \tag{13}$$

This is most easily seen in the notation with the extra ϵ since in this case the α_i are strings of length 2 that are simply decomposed columnwisely. Hence multiple products are well-defined. Furthermore, permutations of rows are isomorphisms. Thus $\mathcal{G}_1 \otimes \mathcal{G}_2 \simeq \mathcal{G}_2 \otimes \mathcal{G}_1$, i.e. exchanging the order of factors affects the order of the coordinates only. Due to the associativity of $\otimes$, we can safely extend these constructions to more than two factors.

The canonical projection $\pi_i : \mathcal{G}_1 \otimes \mathcal{G}_2 \to \mathcal{G}_i$ is obtained by formally isolating the i-th coordinate and contracting the empty strings ϵ and the empty productions $\varnothing = (\epsilon \to \epsilon)$. Clearly we have $\pi_i(T) = T_i$, $\pi_i(N) = N_i$, $\pi_i(s) = s_i$, and $\pi_i(P) = P_i$. The grammar product $\otimes$ thus has the basic properties of a well-defined product.

Let $\mathrm{lan}(\mathcal{G})$ denote the language generated by G. Note that a "string" in $\mathrm{lan}(\mathcal{G})$ is, by construction, a sequence of terminals, each of which is either of the form $\binom{x_1}{x_2}$ with $x_1 \in T_1$ and $x_2 \in T_2$, or of the form $\binom{x_1}{\epsilon}$ with $x_1 \in T_1$, or of the form $\binom{\epsilon}{x_2}$. Thus $\mathrm{lan}(\mathcal{G}_1 \otimes \mathcal{G}_2)$ consists of alignments of strings $\alpha_i \in \mathcal{G}_i$. To see this, note that each string $\alpha_i \in \mathcal{G}_i$ is generated from s_i using a finite sequence $\wp_i = (p_i^1, p_i^2, \dots)$ of productions. Any partial matching of the $\wp_1$ and $\wp_2$ that preserves the sequential order of the two input sequences gives rise to a sequence of productions $\wp \in P^*$ by matching all unmatched p_i^k with the dummy production $\varnothing$. By construction $\pi_i(\wp) = \wp_i$, i.e., $\wp$ derived an alignment of the input strings β_1 and β_2. Conversely, given a sequence $\wp$ of productions of the product grammar, we know that $\pi_i(\wp)$ is a sequence of productions of $\mathcal{G}_i$; hence it constructs strings in $\mathrm{lan}(\mathcal{G}_i)$. It follows that the product language satisfies

$$\pi_i(\mathrm{lan}(\mathcal{G}_1 \otimes \mathcal{G}_2)) = \mathrm{lan}(\mathcal{G}_i) \tag{14}$$

Similarly, we find that parse trees have a natural alignment structure. Let τ be a parse tree for an input $\beta \in \text{lan}(\mathcal{G}_1 \times \mathcal{G}_2)$. Its interior nodes are labeled by the productions, i.e., pairs of the form $\binom{A_1 \to B_1 x_1}{A_2 \to B_2 x_2}$, $\binom{A_1 \to B_1 x_1}{\varnothing}$, or $\binom{\varnothing}{A_2 \to B_2 x_2}$. The projections $\pi_i(\tau)$ are explained by retaining only the i-th coordinate of the vertex label and contracting all vertices labeled by $\varnothing$ in $\pi_i(\tau)$ yields a valid parse tree for $\pi_i(\beta)$ w.r.t. $\mathcal{G}_i$. Thus τ is a tree alignment of the parse trees for the two input strings.

The direct product $\otimes$ forms a monoid on grammars with arbitrary dimensions since

$$P_1^m \otimes P_2^n = \{(p_1 \otimes p_2)^{m+n} | p_1^m \in P_1^m, p_2^n \in P_2^n\}\,, \tag{15}$$

where $p_1 \otimes p_2$ is explained in Def. 1. The neutral element of the $\otimes$ monoid is the zero-dimensional grammar which has one production rule $\epsilon^0 \to \epsilon^0$ that neither reads nor writes anything as it does not operate on a tape. Albeit rather artificial at first glance, it is useful to have a neutral element available. For our example, we have

$$\begin{aligned} & (X \to Xa|X) \otimes (X \to Xa|X) \\ = \quad & \tbinom{X}{X} \to \tbinom{X}{X}\tbinom{a}{a} \mid \tbinom{X}{X}\tbinom{a}{\epsilon} \mid \tbinom{X}{X}\tbinom{\epsilon}{a} \mid \tbinom{X}{X} \end{aligned} \tag{16}$$

This grammar contains the 2-dimensional loop rule $\binom{X}{X} \to \binom{X}{X}$, derived from $(X \to X) \otimes (X \to X)$ that eventually needs to be eliminated. To this end, it will be convenient to consider yet another operation on productions.

The structure-preserving power $*$ For any k-dimensional grammar $\mathcal{G}$ and any natural number $n \in \mathbb{Z}$, $\mathcal{G} * n$ denotes the $k \times n$-dimensional grammar with the same structure. Each k-dimensional (terminal or non-terminal) symbol $\binom{s_1}{s_k}$ is transformed to an $k \times n$-dimensional symbol $\binom{\binom{s_1}{s_k}}{\binom{s_1}{s_k}}$. Note that for a grammar with a single production rule we have $G \otimes G \equiv G * 2$.

For our example grammar, this operation is useful as short-hand for both Equ. 7 and Equ. 8. In the case of linear grammars, the $*$ operator is mostly useful as shorthand to expand singleton grammars. It is worth noting, however, that a number of algorithms, notably [11], in computational biology work on multiple tapes with a grammar structure equal to their one-dimensional cousins. In particular, the Sankoff algorithm [11] is a variant of the Nussinov algorithm extended to two tapes.

We can now construct the full Needleman-Wunsch alignment grammar from the much simpler 1-dimensional constituents of Eqns.(6–8) in the following way:

$$\mathcal{NW} = \mathcal{G} \otimes \mathcal{G} + \mathcal{N} * 2 - \mathcal{L} * 2\,, \tag{17}$$

Written in terms of the productions only, this can be rephrased as

$$\begin{aligned} & (X \to Xa|X) \otimes (X \to Xa|X) + (X \to \epsilon) * 2 - (X \to X) * 2 \\ = \quad & \tbinom{X}{X} \to \tbinom{X}{X}\tbinom{a}{a} \mid \tbinom{X}{X}\tbinom{a}{\epsilon} \mid \tbinom{X}{X}\tbinom{\epsilon}{a} \mid \tbinom{\varepsilon}{\varepsilon} \end{aligned} \tag{18}$$

Note that we have used here a distinct symbol ε to highlight the termination case deriving from $\mathcal{N}$. Since our construction of the Needleman-Wunsch grammar is

based on well-defined algebraic operations we can readily use the same approach to construct much more complex alignment algorithms. Before we proceed, however, we need to address the technical issue of loop rules.

2.3 Grammars with Loops

In Equ. (17) we explicitly added a terminating base case $X \to \varepsilon$ and removed a production rule with infinite derivations $X \to X$. Why do we insist on performing this operation explicitly instead of modifying the definition of the direct product $\otimes$ accordingly?

The main reason lies in performance considerations. An "intelligent" product operator would first need to determine which rules have infinite derivations. For linear grammars with only one non-terminal a rule is not infinite if a single terminal (except ϵ) is present. ϵ rules are also fine, as long as only the empty word case $X \to \epsilon$ is present. Productions of the form $\{X \to Y, Y \to X\}$, however need to be followed up to a depth of the number of production rules present. For context-free grammars, the complexity will increase further, as now multiple non-terminals may exist on the right-hand side. For both convenience and efficiency (by a constant factor), it does not seem to be desirable to transform the grammar into Chomsky normal form. The second problem is the need for rewriting. In the case of $\{X \to Y, Y \to X\}$, rewriting yields $X \to X$ by inserting the rules for Y wherever Y is used. More complicated grammars might quite easily require major rewrites before all loop cases can be removed.

Finally, using looping productions can be conceptually useful during construction. In case of Equ. 6, we either want to read a character in a "step" $X \to Xa$ or perform an in/del with a "stand" $X \to X$. The direct product of Equ. (6) then yields all possibilities of stepping or standing on two (or more) tapes. Of these cases we only want to remove the case where all tapes "stand". This case is quite easily determined as Equ. 8 and just needs to be scaled (with $*$) to the correct dimension and subtracted from the complete grammar.

2.4 Implementation

We have implemented a small compiler for our grammar product formalism with three output targets. First, we generate LaTeX output. This supports researchers in the development of complex, multiple dimensional linear grammars, facilitates the comparison with the intended model for an elaborate alignment-like algorithm. It assists implementation of the grammar in the users' programming language of choice as the mathematical description of the recurrences reduces the chance that a production rule or recursion is simply forgotten.

In addition, we directly target the functional programming language Haskell [12]. It is possible to emit a Haskell module prototype which then needs to be extended with user-defined evaluation (scoring) algebras. This mode mirrors the LaTeX output. Advanced users may make use of TemplateHaskell [13] to *directly embed* our domain-specific language as a proper extension of Haskell itself. Both Haskell-based approaches ultimately make use of stream fusion optimizations

[14] by way of the `ADPfusion` [15] framework that produces efficient code for dynamic programming algorithms.

Currently, the emitted Haskell code for non-trivial applications is slower than optimized `C` by a factor of two [15]. Recent additions to the compiler infrastructure [16], which provide instruction-level parallelism, will reduce this factor further. As `ADPfusion` is built on top of the `Repa` [17] library for CPU-level parallelism, we can expect improvements in this regard to be available for our dynamic programming algorithms in the near future.

3 Applications

```
Grammar: DNA
F{i} -> stay <<< F{i}   c c c
F{i} -> rf1  <<< F{i+1} c c
F{i} -> rf2  <<< F{i+2} c
F{i} -> del  <<< F{i}
//
Grammar: DNAdone
F{i} -> nil <<< empty
//
Grammar: DNAstand
F{i} -> del <<< F{i}
//
```

```
Grammar: PRO
P -> amino <<< P a
P -> del   <<< P
//
Grammar: PROdone
P -> nil <<< empty
//
Grammar: PROstand
P -> del <<< P
//
```

```
Product: DnaPro
DNA >< PRO
+ DNAdone >< PROdone
- DNAstand >< PROstand
//
```

Fig. 1. Atomic grammars for the DNA-Protein alignment example. **(I)** Nucleotides are read in triplets (three nucleotides each). The `DNA` grammar switches between reading frames. `DNAdone` and `DNAstand` handle the termining and looping case. **(II)** The `PRO`tein grammar works similarly, but reads only a single amino acid at a time. The expansion of the `DNA` grammar is more complicated, as the indexed non-terminal symbol F expands to three different non-terminals corresponding to the three possible reading frames. **(III)** The grammar product of `DNA` and `PRO`tein without the looping case "`stand`" and with the terminating case "`done`". In code, `><` represents the direct product ($\otimes$). The resulting 24-production rule grammar is shown in the Supplemental Material together with an extended description.

In this section we discuss one elaborate and practically relevant example where a grammar product of two simple grammars yields a complex result grammar. The alignment of two sequences of the same type is typically simplified due to mirrored operations. Recalling the alignment grammar from above, we speak of in/del operations as an insertion in one sequence may just as well be described as a deletion in the other sequence. In addition, it does not matter which sequence is bound to which input tape.

The alignment of a protein sequence to a DNA sequence is, however, more involved. In Fig. 1 we summarize this more elaborate example. The DNA sequence is read in one of three reading frames (RFs), and a deletion or insertion

does not yield a "simple" in/del but also a frame shift. This more advanced treatment of DNA characters in triplets is due to the *translation* of DNA into protein in steps of three nucleotides, the "codons" of the genetic code. In Fig. 1 frame shifts (with scoring functions `rf1, rf2`) are allowed only at high cost as they change the transcription of following protein characters completely. Staying within a frame is very cheap, even if this involves the deletion of three characters (`del`).

The protein grammar, on the other hand, has the same simple structure as our previous atomic components of the alignment grammar. Here, we indeed only read a single amino acid, or handle a deletion.

The complexity of the DNA-protein alignment stems from the fact that we need to "align" the different frame shifting possibilities in the DNA input while matching zero to three nucleotides to zero or one amino acid in the protein input. In addition, once a frame shift has occurred all following alignments of three nucleotides against one amino acid are scored in the new reading frame until another frame shift occurs or the alignment is completed.

Our framework simplifies the complexity of designing this algorithm considerably. While the *combined* grammar is highly complex, the individual grammars are rather simple. As already mentioned, the protein "stepping grammar" is one of the simplest possible ones. The DNA grammar is more complex as we need to handle stepping and frame shifts in all three reading frames. But considering that we allow indexed non-terminals and calculations on these indices (modulo 3 in the frame shift case), even the frame shift grammar has only four rules, just twice as much as the simplest stepping grammar.

The resulting 24-production rule grammar is easily calculated in our frame work. We emphasize that it is very easily extend this grammar to allow for, say, an alignment of two DNA sequences with two protein sequences. This grammar can be *calculated* at basically no additional cost but would pose a daunting task if implemented by hand. An extended description of this grammar, together with a depiction of the 24 production rules can be found in the Supplemental Material[9].

4 Discussion

Summary We have presented a formal, abstract algebra on linear grammars. This algebra provides operations to create complex, multi-tape grammars from simple, single-tape atomic ones. More informally, we have created a method and implementation to "multiply" dynamic programming algorithms. We also provide a compiler framework that makes the grammars readily available for actual deployment with good performance of the resulting code.

The products of linear grammars, despite the simplicity of individual grammars, give rise to many often-used and powerful algorithms where word-like objects are aligned with each other. We have restricted ourselves to a problem

[9] url:where

from the realm of computational biology, as the alignment of DNA and protein sequences provides a good example of the emerging complexity of algorithmic alignment, especially when the words to be aligned have differing internal structure – in the example case the possibility of a frame shift in the DNA sequence.

Future Work This work also leads to a number of questions to be answered in the future. We should investigate the actual performance of our automatically generated grammar implementations versus hand-written code, but this is mostly a question delegated to the underlying `ADPfusion` framework. We prefer a separation of concerns: *grammar products* emphasize algebraic operations, the user need not be concerned with low-level implementation details.

We have restricted ourselves to linear grammars, as the next class of formal grammars, context-free grammars, requires us to give a good definition of the direct product on production rules with more than one non-terminal symbols.

The direct product implicitly introduces dependencies that couple the input bands. Consider the product of productions $(X \to Xa) \otimes (X \to X)$. There are two possibilities how the right-hand side can be interpreted:

$$\binom{X}{X} \to \binom{X}{X}\binom{a}{\epsilon} \tag{19}$$

$$\binom{X}{X} \to \binom{X}{\epsilon}\binom{a}{X} \tag{20}$$

Aligning the two non-terminals to form a new non-terminal as in Equ. (19) is equivalent to a dependence statement. All possible derivations of $\binom{X}{X}$ are considered and both tapes are coupled.

The situation is quite different for the production rule given in Equ. (20). Since $\binom{X}{\epsilon}$ aligns a substring with the empty string, we basically decouple the two tapes. Furthermore, we formally have constructed a non-terminal $\binom{a}{X}$ that "mixes" non-terminals and terminals on different tapes. In Definition 1 we have avoided this complication by restricting ourselves to linear grammars, where constructions akin to Equ. (20) can always be avoided. When attempting to generalize the framework to arbitrary CFGs, however, this is not possible anymore.

Consider, for example the CFG $\mathcal{A} = \{\{S\}, \{x\}, \{S \to Sx \mid SS\}, S\}$. Even if we give precendence to matching up non-terminals, $\mathcal{A} \otimes \mathcal{A}$ has productions of the form

$$\binom{S}{S} \to \binom{SS}{Sx} \simeq \binom{S}{S}\binom{S}{x} \tag{21}$$

where $\binom{S}{x}$ is neither a terminal nor a non-terminal according to Def. 1. One possibility to deal with this issue is to expand the set of non-terminals to $N = (N_1 \times N_2) \cup (N_1 \times T_2) \cup (T_1 \times N_2)$ and to add productions of the form $\binom{x}{A} \to \binom{x}{\alpha}$ if $x \in T_1$ and $A \to \alpha \in \mathcal{P}_2$ as well as $\binom{A}{x} \to \binom{\alpha}{x}$ if $x \in T_2$ and $A \to \alpha \in \mathcal{P}_1$. The intuition here is that we can have a terminal produced in one factor, while the other factor still presents a non-terminal. Further derivations then can affect only the factor with the non-terminal, while the terminal in the other factor must remain untouched.

A second complication arises e.g. in the following example: $\mathcal{B} = \{\{S\}, \{x\}, \{S \to x \mid SS\}, S\}$. In $\mathcal{B} \times \mathcal{B}$ we now obtain productions of the form $\binom{S}{S} \to \binom{SS}{x}$. A useful resolution in this case is to re-interpret these as

$$\binom{S}{S} \to \binom{SS}{x} \simeq \binom{S}{\epsilon}\binom{S}{x} \mid \binom{S}{x}\binom{S}{\epsilon} \tag{22}$$

i.e., to allow for all alignments of the r.h.s. in the productions. The explicit use of ϵ suggests an alternative extension of terminal and non-terminal symbols sets, respectively. Setting $N = N_1 \times N_2 \cup N_1 \times \{\epsilon\} \cup \{\epsilon\} \times N_2$ and $T = T_1 \times T_2 \cup T_1 \times \{\epsilon\} \cup \{\epsilon\} \times T_2$. In this setting, we would re-interpret the production of Equ. (21) in the following way:

$$\binom{S}{S} \to \binom{SS}{Sx} \simeq \binom{S}{S}\binom{S}{\epsilon}\binom{\epsilon}{x} \mid \binom{S}{S}\binom{\epsilon}{x}\binom{S}{\epsilon} \tag{23}$$

Apart from questions on how to extend algebraic operations on grammars from linear to context-free grammars, we also need to consider scoring algebras for such products. We anticipate that in many cases, a scoring algebra can be expressed as a form of product itself where the two scoring functions (one for each grammar) are themselves combined in some well-defined form. One possibility is the use of a folding operation to combine scores for subsets of the individual dimensions. It then follows that given two algebras $\mathcal{A}_{G_1}$ and $\mathcal{A}_{G_2}$ for grammars G_1 and G_2 we should be able to define an operation $\mathcal{A}_{G_1} \otimes_\tau \mathcal{A}_{G_2}$ which generates appropriate algebras from algebras for atomic grammars. As long as τ has some structure similar to a fold or another operation on subsets of the dimensions (of the grammars) involved, appropriate products can be automatically defined. This becomes especially useful as we want to define `ADP`-like [18] grammar-products as well, to explore the rich space of combined algebras on grammars constructed from algebraic operations on atomic grammars.

Another avenue of future research is the question of semantic ambiguity of the resulting grammars. Simple products of the same grammar yield ambiguous alignments on sequences of in-dels. This problem is typically dealt with a good grammar design that explicitly allows only one order of successive insertions and deletions on multiple tapes. Automatic dis-ambiguiation is probably complicated but would further simplify the creation of complex multi-tape grammars.

Acknowledgements.

This work was funded, in part, by the Austrian FWF, project "SFB F43 RNA regulation of the transcriptome". CHzS thanks Jing, Katja, Lydia, and Nancy (and gin, as well as a mad man in a box).

References

1. Lipman, D.J., Altschul, S.F., Kececioglu, J.D.: A tool for multiple sequence alignment. Proc. Natl. Acad. Sci. USA **86**(12) (1989) 4412–4415
2. Giegerich, R., Meyer, C.: Algebraic Dynamic Programming. In: Lecture Notes In Computer Science. Volume 2422. Springer-Verlag (2002) 349–364

3. Giegerich, R., Meyer, C., Steffen, P.: A Discipline of Dynamic Programming over Sequence Data. Science of Computer Programming **51**(3) (2004) 215–263
4. Gotoh, O.: An improved algorithm for matching biological sequences. J. Mol. Biol. **162** (1982) 705–708
5. Gotoh, O.: Alignment of three biological sequences with an efficient traceback procedure. J. theor. Biol. **121** (1986) 327–337
6. Dewey, T.G.: A sequence alignment algorithm with an arbitrary gap penalty function. J. Comp. Biol. **8** (2001) 177–190
7. Konagurthu, A.S., Whisstock, J., Stuckey, P.J.: Progressive multiple alignment using sequence triplet optimization and three-residue exchange costs. J. Bioinf. and Comp. Biol. **2** (2004) 719–745
8. Kruspe, M., Stadler, P.F.: Progressive multiple sequence alignments from triplets. BMC Bioinformatics **8** (2007) 254
9. Steiner, L., Stadler, P.F., Cysouw, M.: A pipeline for computational historical linguistics. Language Dynamics & Change **1** (2011) 89–127
10. Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology **48**(3) (1970) 443–453
11. Sankoff, D.: Simultaneous solution of the RNA folding, alignment and protosequence problems. SIAM Journal on Applied Mathematics (1985) 810–825
12. The GHC Team: The Glasgow Haskell Compiler (GHC). `http://www.haskell.org/ghc/` (1989–2013)
13. Sheard, T., Jones, S.P.: Template Meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell, ACM (2002) 1–16
14. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream Fusion: From Lists to Streams to Nothing at All. In: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. ICFP'07, ACM (2007) 315–326
15. Höner zu Siederdissen, C.: Sneaking around concatMap: efficient combinators for dynamic programming. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. ICFP '12, New York, NY, USA, ACM (2012) 215–226
16. Mainland, G., Leshchinskiy, R., Jones, S.P., Marlow, S.: Exploiting vector instructions with generalized stream fusion. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming (accepted). (2013)
17. Keller, G., Chakravarty, M.M., Leshchinskiy, R., Peyton Jones, S., Lippmeier, B.: Regular, Shape-polymorphic, Parallel Arrays in Haskell. In: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming. ICFP'10, ACM (2010) 261–272
18. Steffen, P., Giegerich, R.: Versatile and declarative dynamic programming using pair algebras. BMC bioinformatics **6**(1) (2005) 224