Algebraic Dynamic Programming for Multiple Context-Free Grammars

Maik Riechert[a,b], Christian Höner zu Siederdissen[b,c,d], Peter F. Stadler[b,c,d,e,f,g,h]

[a] *Fakultät Informatik, Mathematik und Naturwissenschaften, Hochschule für Technik, Wirtschaft und Kultur Leipzig, Gustav-Freytag-Straße 42a, D-04277 Leipzig, Germany.*
[b] *Bioinformatics Group, Department of Computer Science University of Leipzig, Härtelstraße 16-18, D-04107 Leipzig, Germany.*
[c] *Institute for Theoretical Chemistry, University of Vienna, Währingerstraße 17, A-1090 Wien, Austria.*
[d] *Interdisciplinary Center for Bioinformatics, University of Leipzig, Härtelstraße 16-18, D-04107 Leipzig, Germany.*
[e] *Max Planck Institute for Mathematics in the Sciences, Inselstraße 22, D-04103 Leipzig, Germany.*
[f] *Fraunhofer Institut für Zelltherapie und Immunologie, Perlickstraße 1, D-04103 Leipzig, Germany.*
[g] *Center for non-coding RNA in Technology and Health, University of Copenhagen, Grønnegårdsvej 3, DK-1870 Frederiksberg C, Denmark.*
[h] *Santa Fe Institute, 1399 Hyde Park Rd., Santa Fe, NM 87501*

Abstract

We present theoretical foundations, and a practical implementation, that makes the method of Algebraic Dynamic Programming available for Multiple Context-Free Grammars. This allows to formulate optimization problems, where the search space can be described by such grammars, in a concise manner and solutions may be obtained efficiently. This improves on the previous state of the art which required complex code based on hand-written dynamic programming recursions. We apply our method to the RNA pseudoknotted secondary structure prediction problem from computational biology.

Appendix and supporting files available at:
`http://www.bioinf.uni-leipzig.de/Software/gADP/`

Keywords: Multiple Context-Free Grammars, Dynamic Programming, Algebraic Dynamic Programming, RNA Secondary Structure Prediction,

Email addresses: `choener@bioinf.uni-leipzig.de` (Christian Höner zu Siederdissen), `studla@bioinf.uni-leipzig.de` (Peter F. Stadler)

Pseudoknots

2010 MSC: 00-01, 99-00

Contents

1. Introduction

Dynamic programming (DP) is a general algorithmic paradigm that leverages the fact that many complex problems of practical importance can be solved by recursively solving smaller, overlapping, subproblems [15]. In practice, the efficiency of DP algorithms is derived from "memoizing" and combining the solutions of subproblems of a restricted set of subproblems. DP algorithms are particularly prevalent in discrete optimization [13, Chp. 15], with many key applications in bioinformatics.

DP algorithms are usually specified in terms of recursion relations that iteratively fill a multitude of memo-tables that are indexed by sometimes quite complex objects. This makes the implementation of DP recursions and the maintenance of the code a tedious and error prone task [20].

The theory of Algebraic Dynamic Programming (ADP) [21] circumvents these practical difficulties for a restricted class of DP algorithms, namely those that take strings or trees as input. It is based on the insight that the structure of recursion, i.e., the construction of the state space, the evaluation of sub-solutions, and the selection of sub-solutions based on their value can be strictly separated. In ADP, a DP algorithm is completely described by a context free grammar (CFG), an evaluation algebra, and a choice function. This separation confers two key advantages to the practice of programming: (1) The CFG specifies the state space and thus the structure of the recursion without any explicit use of indices. (2) The evaluation algebra can easily be replaced by another one. The possibility to combine evaluation algebras with each other [59] provides extensive flexibility for algorithm design. The same grammar thus can be used to minimize scores, compute partition functions, and enumerate a fixed number of sub-optimal solutions.

As it stands, the ADP framework is essentially restricted to decompositions of the state space that can be captured by CFGs. This is not sufficient, however, to capture several difficult problems in computational biology. We will use here the prediction of pseudoknotted RNA structures as the paradigmatic example. Other important examples that highlight the complicated recursions in practical examples include the simultaneous alignment and folding of RNA (a.k.a. Sankoff's algorithm [53]), implemented e.g. in `foldalign` [24] and `dynalign` [41], and the RNA-RNA interaction problem (RIP [2]). For the latter, equivalent implementations using slightly different recursions have become available [11, 31, 32], each using dozens of 4-dimensional tables to memoize intermediate results. The implementation and testing of such complicated multi-dimensional recursions is a tedious and error-prone process that hampers the systematic exploration of variations of scoring models and search space specification. The three-dimensional structure of an RNA molecule is determined by topological constraints that are determined by the mutual arrangements of the base paired helices, i.e., by its secondary structure [3]. Although most RNAs have simple structures that do not involve crossing base pairs, pseudoknots that violate this simplifying condition are not uncommon [60]. In several cases, pseudoknots are functionally important features that cannot be neglected in a meaningful way, see e.g. [16, 42, 19]. In its most general form, the RNA folding with stacking-based energy functions is NP-complete [1, 40]. The commonly used RNA folding tools (`mfold` [65] and the `Vienna RNA Package` [39]), on the other hand, exclude pseudoknots altogether.

Polynomial-time dynamic programming (DP) algorithms can be devised for a wide variety of restricted classes of pseudoknots. However, most approaches are computationally very demanding, and the design of pseudoknot folding algorithms has been guided decisively by the desire to limit computational cost and to achieve a manageable complexity of the recursions [50]. Consequently, a plethora of different classes of pseudoknotted structures have been considered, see e.g. [12, 52, 10, 49], the references therein, and the book [48]. Since the corresponding folding algorithms have been implemented at different times using different parametrizations of the energy functions it is hard to directly compare them and their performance. On the other hand, a more systematic investigation of alternative structure definitions would require implementations of the corresponding folding algorithms. Due to the complicated structure of the standard energy model this would entail major programming efforts, thus severely limiting such efforts. Already for non-pseudoknotted RNAs that can be modeled by simple context-free grammars, the effort to unify a number of approaches into a common framework required a major effort [51].

Multiple context-free grammars (MCFG) [55] have turned out to be a very natural framework for the RNA folding problem with pseudoknots. In fact, most of the pseudoknot classes can readily be translated to multiple context-free

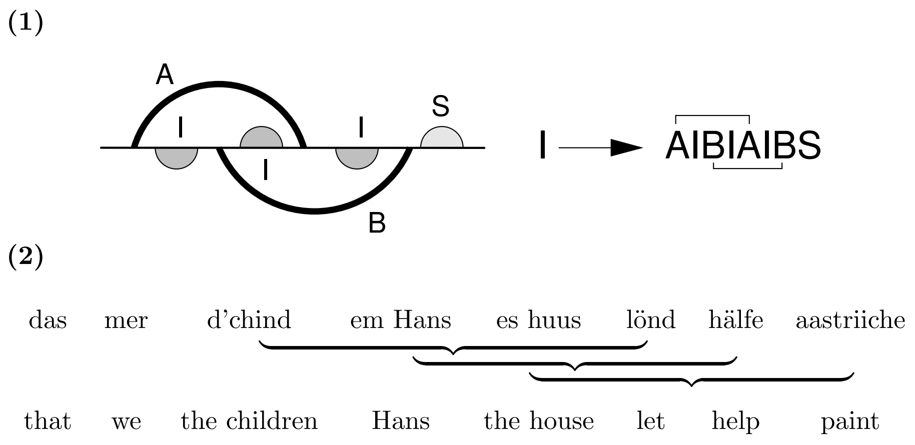

(1)
A
I
I
I
S
B
I → AIBIAIBS
(2)
das mer d'chind em Hans es huus lönd hälfe aastriiche
that we the children Hans the house let help paint

Figure 1: **(1)** Example of a pseudoknot and the MCFG production formulation for type H pseudoknots from the `gfold` grammar [49] as also used in our `GenussFold` example code. Here I represents an arbitrary structure, S a pseudoknot-free secondary structure, while A and B represent the two "one-gap objects" (two-dimensional non-terminals) forming the pseudoknot. The two components of A and B are linked by a bracket to highlight the interleaved structure. **(2)** Crossing dependencies in a Swiss-German sentence [56, 58] show that the language has non-context-free structure. Braces designate semantic dependencies.

grammars (MCFG), see Fig.1(1) for a simple example. In contrast to CFGs, the non-terminal symbols of MCFGs may consist of multiple components that must be expanded in parallel. In this way, it becomes possible to couple separated parts of a derivation and thus to model the crossings inherent in pseudoknotted structures. Reidys et al. [49], for instance, makes explicit use of MCFGs to derive the DP recursions. Stochastic MCFGs were used for RNA already in [35], and a unified view of many pseudoknot classes recently has been established in terms of MCFGs [43], introducing a direct connection between MCFGs and generating functions. Although other grammar models have been proposed, e.g. tree-adjoining grammars [61], and multi-tape S-attribute grammars [37, 38, 64], MCFGs appear to be the most natural choice. A generalization of the framework of the ADP framework [21] from CFG to MCFG thus could overcome the technical difficulties that so far have hampered a more systematic exploration of pseudoknotted structure classes.

In computational linguistics, several distinct classes of "mildly context-sensitive grammar formalisms" have been introduced to capture the syntactic structure of natural language. Among them are, in the order of increasing generality, tree-adjoining grammars [33], coupled context free grammars [30], and MCFGs as well as the equivalent linear context free rewriting systems [62]. Dutch and Swiss-German, for example, have non-context-free structures [56, 58], see Fig. 1(2). MCFGs have been used to model grammars for natural languages explicitly e.g. in [58]. Given the constraints of natural languages, the complexity of their non-context free structure is quite limited in practice.

2. Multiple Context-Free Grammars

Multiple Context-Free Grammars (MCFGs) [55, 54] are a particular type of weakly context-sensitive grammar formalism that still allows for parsing in polynomial time. In contrast to the general case of context-sensitive grammars, MCFGs employ in their rewrite rules only total functions that concatenate constant strings and components of their arguments. The original formulation of MCFGs was modeled after context sensitive grammars and hence emphasizes the rewrite rules. Practical applications, in particular in bioinformatics, emphasize the productions. We therefore start here from a formal definition of a MCFG that is slightly different from its original version.

MCFGs operate on non-terminals that have an interpretation as tuples of strings over an alphabet $\mathcal{A}$ — rather than strings as in the case of CFGs.

An important feature of MCFGs are rewriting functions that recombine the components of different tuples. A particular rewrite function can be defined by specifying which input components from which of its arguments are concatenated in each of its output components. We first deal with a fixed single component of the output. Suppose f has k arguments of dimensions $a_1, a_2, \dots, a_k$ and define the corresponding set of pairs $\mathcal{I} = \{(i, j) \mid 1 \leq i \leq k, 1 \leq j \leq a_k\}$.

The characteristic $[\![f_l]\!], 1 \leq l \leq b$ of a component-wise rewrite function f_l is an ordered list of pairs $[\![f_l]\!] = p_1 \dots p_m \in \mathcal{I}^m$. A pair (i, j) designates the j'th component from the i'th argument, the component f_l then concatenates these input components into the l'th output in the order given by $[\![f_l]\!]$. A rewrite function is then simply a b-tuple of component-wise rewrite functions. The function f_l is therefore uniquely determined by its characteristic $[\![f_l]\!]$. In more precise language, we have

Definition 1. A function $f : ((\mathcal{A}^*)^*)^k \to (\mathcal{A}^*)^b$ is a *rewrite function* of arity $(a_1, \dots, a_k; b) \in \mathbb{N}^* \times \mathbb{N}$ if for each $l \in \{1, \dots, b\}$ there is a list $[\![f_l]\!] \in \mathcal{I}^*$ so that the l'th component $f_l : ((\mathcal{A}^*)^*)^k \to \mathcal{A}^*$ is of the form $x \mapsto \sigma(p_1, x) \dots \sigma(p_{m_l}, x)$, where $\sigma(p_h, x) = (x_i)_j$ with $p_h = (i, j) \in \mathcal{I}$ for $1 \le h \le m_l$.

Definition 2 (Linear Rewriting Function). A rewriting function f is called linear if each pair $(i, j) \in \mathcal{I}$ occurs at most once in the characteristic $[\![(f_1, \dots, f_b)]\!]$ of f.

Linear rewriting functions thus are those in which each component of each argument appears at most once in the output.

We could strengthen the linearity condition and require that components of rewriting function arguments are used *exactly* once, instead of *at most* once. It was shown in [55, Lemma 2.2] that this does not affect the generative power.

Example 1. The function

$$f\left(\begin{pmatrix} x_{1,1} \\ x_{1,2} \\ x_{1,3} \end{pmatrix}, \begin{pmatrix} x_{2,1} \end{pmatrix}\right) = \begin{pmatrix} x_{2,1} \cdot x_{1,2} \\ x_{1,3} \end{pmatrix}$$

is a linear rewriting function with arity $(3, 1; 2)$ and characteristic

$$[\![f]\!] = \begin{pmatrix} (2,1)(1,2) \\ (1,3) \end{pmatrix}.$$

An example evaluation is as follows:

$$f\left(\begin{pmatrix} ab \\ c \\ abc \end{pmatrix}, \begin{pmatrix} ba \end{pmatrix}\right) = \begin{pmatrix} ba \cdot c \\ abc \end{pmatrix}$$

Definition 3 (Multiple Context-Free Grammar). [cf. 54] An MCFG is a tuple $\mathcal{G} = (V, \mathcal{A}, Z, R, P)$ where V is a finite set of nonterminal symbols, $\mathcal{A}$ a finite set of terminal symbols disjoint from V, $Z \in V$ the start symbol, R a finite set of linear rewriting functions, and P a finite set of productions. Each $v \in V$ has a *dimension* $\dim(v) \geq 1$, where $\dim(Z) = 1$. Productions have the form $v_0 \to f[v_1, \ldots, v_k]$ with $v_i \in V \cup (\mathcal{A}^*)^*$, $0 \leq i \leq k$, and f is a linear rewrite function of arity $(\dim(v_1), \ldots, \dim(v_k); \dim(v_0))$.

Productions of the form $v \to f[]$ with $f[] = \begin{pmatrix} f_1 \\ \vdots \\ f_d \end{pmatrix}$ are written as $v \to \begin{pmatrix} f_1 \\ \vdots \\ f_d \end{pmatrix}$ and are called *terminating productions*. An MCFG with maximal nonterminal dimension k is called a k-MCFG.

Deviating from [54], Definition 3 allows terminals as function arguments in productions and instead prohibits the introduction of terminals in rewriting functions. This change makes reasoning easier and fits better to our extension of ADP. We will see that this modification does not affect the generative power.

Example 2. As an example consider the language $\mathcal{L} = \{\mathtt{a}^i\mathtt{b}^j\mathtt{a}^i\mathtt{b}^j \mid i, j \geq 0\}$ of overlapping palindromes of the form $\mathtt{a}^i\mathtt{b}^j\mathtt{a}^i$ and $\mathtt{b}^j\mathtt{a}^i\mathtt{b}^j$, with $\mathtt{b}^j$ and $\mathtt{a}^i$ missing, respectively. It cannot be expressed by a CFG. The MCFG $\mathcal{G} = (\{Z, A, B\}, \{\mathtt{a}, \mathtt{b}\}, Z, R, P)$ with $\mathcal{L}(\mathcal{G}) = \mathcal{L}$ serves as a running example:

$$\begin{array}{ll}
\dim(Z) = 1 & \dim(A) = \dim(B) = 2 \\
Z \to \mathrm{f}_Z[A, B] & \mathrm{f}_Z[\left(\begin{smallmatrix} A_1 \\ A_2 \end{smallmatrix}\right), \left(\begin{smallmatrix} B_1 \\ B_2 \end{smallmatrix}\right)] = A_1 B_1 A_2 B_2 \\
A \to \mathrm{f}_P[A, \left(\begin{smallmatrix} \mathtt{a} \\ \mathtt{a} \end{smallmatrix}\right)] \mid \left(\begin{smallmatrix} \epsilon \\ \epsilon \end{smallmatrix}\right) & \mathrm{f}_P[\left(\begin{smallmatrix} A_1 \\ A_2 \end{smallmatrix}\right), \left(\begin{smallmatrix} c \\ d \end{smallmatrix}\right)] \quad = \left(\begin{smallmatrix} A_1 c \\ A_2 d \end{smallmatrix}\right) \\
B \to \mathrm{f}_P[B, \left(\begin{smallmatrix} \mathtt{b} \\ \mathtt{b} \end{smallmatrix}\right)] \mid \left(\begin{smallmatrix} \epsilon \\ \epsilon \end{smallmatrix}\right) &
\end{array}$$

The grammar in the original framework corresponding to Example 2 is given in the Appendix for comparison.

Definition 4. [cf. 54] The derivation relation $\overset{*}{\Rightarrow}$ of the MCFG $\mathcal{G}$ is the reflexive, transitive closure of its direct derivation relation. It is defined recursively:

(i) If $v \to a \in P$ with $a \in (\mathcal{A}^*)^{\dim(v)}$ then one writes $v \overset{*}{\Rightarrow} a$.

(ii) If $v_0 \to f[v_1, \ldots, v_k] \in P$ and (a) $v_i \overset{*}{\Rightarrow} a_i$ for $v_i \in V$, or (b) $v_i = a_i$ for $v_i \in (\mathcal{A}^*)^*$ $(1 \leq i \leq k)$, then one writes $v_0 \overset{*}{\Rightarrow} f[a_1, \ldots, a_k]$.

Definition 5. The language of $\mathcal{G}$ is the set $\mathcal{L}(\mathcal{G}) = \{w \in \mathcal{A}^* \mid Z \overset{*}{\Rightarrow} w\}$. A language $\mathcal{L}$ is a multiple context-free language if there is a MCFG $\mathcal{G}$ such that $\mathcal{L} = \mathcal{L}(\mathcal{G})$.

Two grammars $\mathcal{G}_1$ and $\mathcal{G}_2$ are said to be *weakly equivalent* if $\mathcal{L}(\mathcal{G}_1) = \mathcal{L}(\mathcal{G}_2)$. Strong equivalence would, in addition, require semantic equivalence of parse trees [47] and is not relevant for our discussion. An MCFG is called *monotone* if for all rewriting functions the components of a function argument appear in the same order on the right-hand side. It is *binarized* if at most two nonterminals appear in any right-hand side of a production. In analogy to the various normal forms of CFGs one can construct weakly equivalent MCFGs satisfying certain constraints, in particular ϵ-free, monotone, and binarized [34, 7.2].

Theorem 1. *The class of languages produced by the original definition of MCFG is equal to the class of languages produced by the grammar framework of Definition 3.*

Proof. The simple transformation between the notations is given in the Appendix.

Derivation trees are defined in terms of the derivation relation $\overset{*}{\Rightarrow}$ in the following way:

Definition 6. (i) Let $v \to a \in P$ with $a \in (\mathcal{A}^*)^{\dim(v)}$. Then the tree t with the root labelled v and a single child labelled a is a derivation tree of a.

(ii) Let $v_0 \to f[v_1, \ldots, v_k] \in P$. Then the tree t with a single node labelled v_i is a derivation tree of v_i for each $v_i \in (\mathcal{A}^*)^*$ and $1 \leq i \leq k$.

(iii) Let $v_0 \to f[v_1, \ldots, v_k] \in P$ and (a) $v_i \overset{*}{\Rightarrow} a_i$ for $v_i \in V$, or (b) $v_i = a_i$ for $v_i \in (\mathcal{A}^*)^*$ $(1 \leq i \leq k)$, and suppose $t_1, \ldots, t_k$ are derivation trees of $a_1, \ldots, a_k$. Then a tree t with the root labelled $v_0 : f$ and $t_1, \ldots, t_k$ as immediate subtrees (from left to right) is a derivation tree of $f[a_1, \ldots, a_k]$.

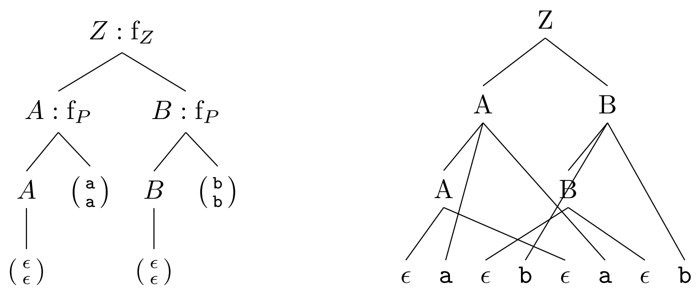
By construction, $w \in \mathcal{L}(\mathcal{G})$ if and only if there is (at least one) derivation tree for w.

It is customary in particular in computational linguistics to rearrange derivation trees in such a way that words (leaf nodes) are shown in the same order in which they appear in the sentences, as in the `NeGra` treebank [57]. As noted in [34], these trees with crossing branches have an interpretation as LCFRS derivation trees. In applications to RNA folding, it also enhances interpretability to draw derivation trees (which, given the appropriate semantics, correspond to RNA structures) on top of the input sequence. A derivation tree t is transformed into an *input-ordered derivation trees* (ioDT) as follows:

1. replace each node labelled with multiple terminal or ϵ symbols with new nodes of single symbols,
2. reorder terminal nodes horizontally according to the rewriting functions such that the input word is displayed from left to right, and
3. remove the function symbols from the node labels.

Conversely, an ioDT can be translated to its corresponding derivation tree by collapsing sibling terminals or ϵ, resp., to a single node. Furthermore, derivation trees are traditionally laid out in a crossing-free manner.

Example 3. In contrast to the derivation tree (left), the ioDT (right) of `abab` makes the crossings immediately visible:



$Z : \mathrm{f}_Z$
$A : \mathrm{f}_P$
$B : \mathrm{f}_P$
A
$\binom{a}{a}$
B
$\binom{b}{b}$
$\binom{\epsilon}{\epsilon}$
$\binom{\epsilon}{\epsilon}$
Z
A
B
A
B
ϵ a ϵ b ϵ a ϵ b

(1)

3. Multiple Context-Free ADP Grammars

In this section we show how to combine MCFGs with the ADP framework in order to solve combinatorial optimization problems for which CFGs do not provide enough expressive power to describe the search space. To differentiate between the existing and our ADP formalism, we refer to the former as "context-free ADP" (CF-ADP) and ours as "multiple context-free ADP" (MCF-ADP). We start with some common terminology, following [21] as far as possible.

Signatures and algebras. A (many-sorted) signature Σ is a tuple (S, F) where S is a finite set of sorts, and F a finite set of function symbols $f : s_1 \times \ldots \times s_n \to s_0$ with $s_i \in S$. A Σ-algebra contains interpretations for each function symbol in F and defines a domain for each sort in S.

Terms and trees. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. A tree containing a designated occurrence of a subtree t is denoted $C\langle \ldots t \ldots \rangle$.

Term algebra. A term algebra T_Σ arises by interpreting the function symbols in Σ as constructors, building bigger terms from smaller ones. When variables from a set V can take the place of arguments to constructors, we speak of a term algebra with variables, $T_\Sigma(V)$.

Definition 7 (Tree grammar over Σ [cf. 22, sect. 3.4]). A (regular) tree grammar $\mathcal{G}$ over a signature Σ is defined as a tuple (V, Σ, Z, P), where V is a finite set of nonterminal symbols disjoint from function symbols in Σ, $Z \in V$ is the start symbol, and P a finite set of productions $v \to t$ where $v \in V$ and $t \in T_\Sigma(V)$. All terms t in productions $v \to t$ of the same nonterminal $v \in V$ must have the same sort. The derivation relation for tree grammars is $\Rightarrow^*$, with $C\langle \ldots v \ldots \rangle \Rightarrow C\langle \ldots t \ldots \rangle$ if $v \to t \in P$. The language of a term $t \in T_\Sigma(V)$ is the set $\mathcal{L}(\mathcal{G}, t) = \{t' \in T_\Sigma \mid t \Rightarrow^* t'\}$. The language of $\mathcal{G}$ is the set $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{G}, Z)$.

In the following, signatures with sorts of the form $(\mathcal{A}^*)^d$ with $d \geq 1$ are used. These refer to tuples of terminal sequences and imply that the relevant sorts, function symbols, domains, and interpretations for sequence and tuple construction are part of the signatures and algebras without explicitly including them.

We are now in the position to introduce MCF-ADP formally.

Definition 8 (MCF-ADP Signature). An MCF-ADP signature Σ is a tuple $(\mathcal{A}, \mathcal{S}, F)$ which defines a signature $(\{\mathcal{A}\} \cup \mathcal{S}, F \cup F_{\mathcal{A}})$ where $\mathcal{S}$ is a finite set of result sorts S^j with $j \geq 1$ and $\mathcal{A}$ a finite set of terminal symbols that is formally used as a sort but also implicitly defining a finite set $F_{\mathcal{A}}$ of constant function symbols $a : \emptyset \to \mathcal{A}$ for each $a \in \mathcal{A}$. Each symbol $f \in F$ has the form $f : s_1 \times \ldots \times s_n \to s_0$ with $s_i \in \{(\mathcal{A}^*)^{d_i}\} \cup \mathcal{S}$, $d_i \geq 1$, $1 \leq i \leq n$, and $s_0 \in \mathcal{S}$.

Definition 9 (Rewriting algebra). Let $\Sigma = (\mathcal{A}, \mathcal{S}, F)$ be an MCF-ADP signature. A rewriting algebra $\mathcal{E}_R$ over Σ describes a Σ-algebra and consists of linear rewriting functions for each function symbol in F. Sorts $S^d \in \mathcal{S}, d \geq 1$ are assigned to the domains $(\mathcal{A}^*)^d$. The explicit interpretation of a term $t \in T_\Sigma$ in $\mathcal{E}_R$ is denoted with $\mathcal{E}_R(t)$.

Definition 10 (MCF-ADP grammar over Σ). An MCF-ADP grammar $\mathcal{G}$ over an MCF-ADP signature $\Sigma = (\mathcal{A}, \mathcal{S}, F)$ is defined as a tuple $(V, \mathcal{A}, Z, \mathcal{E}_R, P)$ and describes a tree grammar (V, Σ, Z, P), where $\mathcal{E}_R$ is a rewriting algebra, and the productions in P have the form $v \to t$ with $v \in V$ and $t \in T_\Sigma(V)$, where the sort of t is in $\mathcal{S}$. Each evaluated term $\mathcal{E}_R(t) \in T_\Sigma(v)$ for a given nonterminal $v \in V$ has the same domain $(\mathcal{A}^*)^d$ with $d \geq 1$ where d is the *dimension* of v.

With our definition of MCF-ADP grammars we can use existing MCFGs nearly verbatim and prepare them for solving optimization problems.

Example 4. The grammar from Example 2 is now given as MCF-ADP grammar $\mathcal{G} = (\{Z, A, B\}, \{\mathtt{a}, \mathtt{b}\}, Z, \mathcal{E}_R, P)$ in the following form:

$$\begin{array}{ll}
\dim(Z) = 1 \qquad \dim(A) = \dim(B) = 2 & \\
P: & \mathcal{E}_R: \\
Z \to \mathrm{f}_Z(A, B) & \mathrm{f}_Z[\binom{A_1}{A_2}, \binom{B_1}{B_2}] = A_1 B_1 A_2 B_2 \\
A \to \mathrm{f}_P(A, \binom{\mathtt{a}}{\mathtt{a}}) \mid \mathrm{f}_\epsilon & \mathrm{f}_P[\binom{A_1}{A_2}, \binom{c}{d}] \quad = \binom{A_1 c}{A_2 d} \\
B \to \mathrm{f}_P(B, \binom{\mathtt{b}}{\mathtt{b}}) \mid \mathrm{f}_\epsilon & \mathrm{f}_\epsilon \qquad\qquad\quad = \binom{\epsilon}{\epsilon}
\end{array}$$

with the signature Σ:

$$\begin{aligned} \mathrm{f}_Z &: S^2 \times S^2 && \to S^1 \\ \mathrm{f}_P &: S^2 \times (\mathcal{A}^*)^2 && \to S^2 \\ \mathrm{f}_\epsilon &: \emptyset && \to S^2 \end{aligned}$$

The only change necessary to rephrase example 2, which uses Defn. 3, into example 4, which makes use of Defn. 10, is to transform the terminating productions of the MCFG into constant rewriting functions. These are then used in the productions. The f_ϵ function is an example of such a transformation.

4. Yield Parsing for MCF-ADP Grammars

Given an MCF-ADP grammar $\mathcal{G} = (V, \mathcal{A}, Z, \mathcal{E}_R, P)$ over Σ, then its yield function $\mathrm{y}_\mathcal{G} : T_\Sigma \to (\mathcal{A}^*)^*$ is defined as $\mathrm{y}_\mathcal{G}(t) = \mathcal{E}_R(t)$. The yield language $\mathcal{L}_\mathrm{y}(\mathcal{G}, t)$ of a term $t \in T_\Sigma(V)$ is $\{\mathrm{y}_\mathcal{G}(t') \mid t' \in \mathcal{L}(\mathcal{G}, t)\}$. The yield language of $\mathcal{G}$ is $\mathcal{L}_\mathrm{y}(\mathcal{G}) = \mathcal{L}_\mathrm{y}(\mathcal{G}, Z)$.

It is straightforward to translate MCF-ADP grammars and MCFGs into each other.

Theorem 2. *The class of yield languages of MCF-ADP grammars is equal to the class of languages generated by MCFGs.*

Proof. See Appendix D.

The inverse of generating a yield language is called yield parsing. The yield parser $\mathcal{Q}_\mathcal{G}$ of an MCF-ADP grammar $\mathcal{G}$ computes the search space of all possible yield parses for a given input x:

$$\mathcal{Q}_\mathcal{G}(x) = \{t \in \mathcal{L}(\mathcal{G}) \mid \mathrm{y}_\mathcal{G}(t) = x\} \tag{2}$$

As an example, the input `abab` spawns a search space of one element:

$$\mathcal{Q}_\mathcal{G}(\texttt{abab}) = \{\mathrm{f}_Z(\mathrm{f}_P(\mathrm{f}_\epsilon, \tbinom{\texttt{a}}{\texttt{a}}), \mathrm{f}_P(\mathrm{f}_\epsilon, \tbinom{\texttt{b}}{\texttt{b}}))\}$$

5. Scoring in MCF-ADP

Having defined the search space we need some means of scoring its elements so that we can solve the given dynamic programming problem. In general, this problem need not be an optimization problem.

Instead of rewriting terms with a rewriting algebra, we now evaluate them with the help of an evaluation or scoring algebra. As we need multisets [7] now, we introduce them intuitively and refer to the appendix for a formal definition. A multiset can be understood as a list without an order, or a set in which elements can appear more than once. We use square brackets to differentiate them from sets. The number of times an element is repeated is called its *multiplicity*. If x is an element of a multiset X with multiplicity m, one writes $x \in^m X$. If $m > 0$, one writes $x \in X$. Similar to sets, a multiset-builder notation of the form $[x \mid P(x)]$ exists which works like the list comprehension syntax known from programming languages like Haskell or Python, that is, multiplicities are carried over to the resulting multiset. The multiset sum $\uplus$, also called additive union, sums the multiplicities of two multisets together, equal to list concatenation without an order. The set of multisets over a set S is denoted $\mathcal{M}(S)$.

Definition 11 (Evaluation Algebra [cf. 21, Def. 2]). Let $\Sigma = (\mathcal{A}, \mathcal{S}, F)$ be an MCF-ADP signature and $\Sigma' = (\{\mathcal{A}\} \cup \mathcal{S}, F \cup F_{\mathcal{A}})$ its underlying signature. An evaluation algebra $\mathcal{E}_E$ over Σ is defined as a tuple (S_E, F_E, h_E) where the domain S_E is assigned to all sorts in $\mathcal{S}$, F_E are functions for each function symbol in F, and $\mathrm{h}_E : \mathcal{M}(S_E) \to \mathcal{M}(S_E)$ is an objective function on multisets. The explicit interpretation of a term $t \in T_\Sigma$ in $\mathcal{E}_E$ is denoted $\mathcal{E}_E(t)$.

In dynamic programming, the objective function is typically minimizing or maximizing over all possible candidates. In ADP, a more general view is adopted where the objective function can also be used to calculate the size of the search space, determine the k-best results, enumerate the whole search space, and so on. This is the reason why multisets are used as domain and codomain of the objective function. If minimization was the goal, then the result would be a multiset holding the minimum value as its single element.

Example. Returning to our example, we now define evaluation algebras to solve three problems for the input $x = \texttt{abab}$: (a) counting how many elements the search space contains, (b) constructing compact representations of the elements, and (c) solving the word problem.

(a)	(b)	(c)
$S_E = \mathbb{N}$	$S_E = (\mathcal{A}^*)^*$	$S_E = \{\text{unit}\}$
$\mathrm{h}_E = \text{sum}$	$\mathrm{h}_E = \text{id}$	$\mathrm{h}_E = \text{notempty}$
$\mathrm{f}_Z(A, B) \quad = A$	$\mathrm{f}_Z(\binom{A_1}{A_2}, \binom{B_1}{B_2}) \ = A_1B_1A_2B_2$	$\mathrm{f}_Z(A, B) \quad = \text{unit}$
$\mathrm{f}_P(A, \binom{c}{d}) = A$	$\mathrm{f}_P(\binom{A_1}{A_2}, \binom{\text{“}\texttt{a}\text{”}}{\text{“}\texttt{a}\text{”}}) = \binom{A_1\texttt{[}}{A_2\texttt{]}}$	$\mathrm{f}_P(A, \binom{c}{d}) = \text{unit}$
$\mathrm{f}_\epsilon \qquad\quad = 1$	$\mathrm{f}_P(\binom{A_1}{A_2}, \binom{\text{“}\texttt{b}\text{”}}{\text{“}\texttt{b}\text{”}}) = \binom{A_1\texttt{(}}{A_2\texttt{)}}$	$\mathrm{f}_\epsilon \qquad\quad = \text{unit}$
	$\mathrm{f}_\epsilon \qquad\qquad\qquad = \binom{\epsilon}{\epsilon}$	
$\mathcal{G}(\mathcal{E}_E, x) = [1]$	$\mathcal{G}(\mathcal{E}_E, x) = [\texttt{[(])}]$	$\mathcal{G}(\mathcal{E}_E, x) = [\text{unit}]$

Finally, we have to define the objective function. For the examples above we have $\text{id}(m) = m$,

$$\text{sum}(m) = \begin{cases} \emptyset, & \text{if } m = \emptyset, \\ \left[\sum\limits_{x \in^n m} x \cdot n\right], & \text{else.} \end{cases} \qquad \text{notempty}(m) = \begin{cases} \emptyset, & \text{if } m = \emptyset, \\ [\text{unit}], & \text{else.} \end{cases}$$

Note that an empty multiset is returned for (a) and (b) when the search space has no elements, and for (c) when there is no parse for the input word. While this most general multiset monoid with $\emptyset$ as neutral element is used in standard ADP definitions, it is often possible to use a more specific monoid based on the underlying data type, e.g. $\mathbb{N}$ or $\mathbb{B}$, with corresponding neutral elements like 0 for counting, and `false` for the word problem. In practical implementations of ADP, such as ADPfusion, this optimization is done so that arrays of primitive types can be used to efficiently store partial solutions.

Definition 12. An *MCF-ADP problem instance* consists of an MCF-ADP grammar $\mathcal{G} = (V, \mathcal{A}, Z, \mathcal{E}_R, P)$ over Σ, an evaluation algebra $\mathcal{E}_E$ over Σ, and an input sequence $x \in (\mathcal{A}^*)^{\dim(Z)}$.

Solving an MCF-ADP problem means computing $\mathcal{G}(\mathcal{E}_E, x) = \mathrm{h}_E[\mathcal{E}_E(t) \mid t \in \mathcal{Q}_{\mathcal{G}}(x)]$.

We remark that algebraic dynamic programming subsumes semiring parsing [23] with the use of evaluation algebras. An ADP parser can be turned into a semiring parser by instantiating the evaluation algebra as parameterized over semirings.

Definition 13. Let $\mathcal{G}$ be a grammar and x an input string. An object x_u is a subproblem of (parsing) x in $\mathcal{G}$ if there is a parse tree t for x and a node u in t so that $x_{t,u}$ is the part of x processed in the subtree of t rooted in u.

The object $x_{t,u}$ can be specified by an ordered set $j(x_{t,u})$ of indices referring to positions in the input string x.

In the most general setting of dynamic programming it is possible to encounter an exponential number of subproblems. A good example is the well-known DP solution of the Traveling Salesman Problem [4]. To obtain polynomial time algorithms it is necessary that the "overlapping subproblems condition" [6] is satisfied.

Definition 14. The grammar $\mathcal{G}$ has the *overlapping subproblem property* if the number of distinct index sets $j(x_{t,u})$ taken over all inputs x with fixed length $|x| = n$, all parse trees t for x in $\mathcal{G}$, and all nodes u in t is polynomially bounded in the input size n.

Theorem 3. *Let $(V, \mathcal{A}, Z, \mathcal{E}_R, P)$ be an MCF-ADP grammar and x an input string of length n. Then there are at most $O(n^{2\max\{\dim(v)|v\in V\}})$ subproblems.*

Proof. Let $(V, \mathcal{A}, Z, \mathcal{E}_R, P)$ be a monotone MCF-ADP grammar. A nonterminal $A \in V$ with $\dim(A) = d$ corresponds to an ordered list of d intervals on the input string x and therefore depends on $2d$ delimiting index positions $i_1 \le i_2 \le \ldots \le i_{2d}$, i.e., for an input of length n there are at most $\mathrm{ms}(n, 2d)$ distinct evaluations of A that need to be computed and possibly stored, where $\mathrm{ms}(n,k) = \sum_{1\le i_1 \le i_2 \le \ldots i_k \le n} 1 = \binom{n+k-1}{k}$ is the multiset coefficient [18, p.38].

For fixed k, we have $\mathrm{ms}(n,k) \in \Theta(n^k) \subseteq O(n^k)$. Allowing non-monotone grammars amounts to dropping the order constraints between the index intervals, i.e., we have to compute at most n^k entries.

The r.h.s. of a production $v \to t \in P$ therefore defines $b = \mathrm{iv}(t)$ intervals with $\mathrm{iv}(f(t_1,\ldots,t_m)) = \sum_{1\le k\le m \wedge t_k \in V} \dim(t_k)$ and hence $b+d$ delimiting index positions. We can safely ignore terminals since they refer to a fixed number of characters at positions completely determined by, say, the preceding nonterminal. Terminals therefore do not affect the scaling of the number of intervals with the input length n. Since each interval corresponding to a nonterminal component on the r.h.s. must be contained in one of the intervals corresponding to A, $2d$ of the interval boundaries on the r.h.s. are pre-determined by the indices on the l.h.s. Thus computing any one of the $O(n^{2d})$ values of A requires not more than $O(n^{b-d})$ different parses so that the total effort for evaluating A is bounded above by $O(n^{b+d})$ steps (sub-parses) and $O(n^{2d})$ memory cells.

Since a MCF-ADP grammar has a finite, input independent number of productions it suffices to take the maximum of $b+d$ over all productions to establish $O(n^{\max\{\mathrm{iv}(t)+\dim(v)|v\to t\in P\}})$ sub-parsing steps and $O(n^{2\max\{\dim(v)|v\in V\}})$ as the total number of parsing steps performed by any MCF-ADP algorithm.

Corollary 1. *Every MCF-ADP grammar satisfies the overlapping subproblem property.*

Each object that appears as solution for one of the subproblems will in general be part of many parses. In order to reduce the actual running time, the parse and evaluation of each particular object ideally is performed only once, while all further calls to the same object just return the evaluation. To this end, parsing results are *memoized*, i.e., tabulated. The actual size of these memo tables depends, as we have seen above, on the specific MCF-ADP grammar and, of course, on the properties of objects to be stored.

To address the latter issue, we have to investigate the contribution of the evaluation algebra. For every individual parsing step we have to estimate the computational effort to combine the k objects returned from the subproblems.

Let $\ell(n)$ be an upper bound on the size of the list of results returned for a subproblem as a function of the input size n. Combining the k lists returned from the subproblems then requires $O(\ell^k)$ effort. The list entries themselves may also be objects whose size scales with n. The effort for an individual parsing step is therefore bounded by $\mu(n) = \ell(n)^{m(P)}\zeta(n)$, where $m(P)$ is the maximum arity of a production and $\zeta(n)$ accounts for the cost of handling large list entries. The upper bound $\mu(n)$ for the computational effort of a single parsing step is therefore completely determined by the evaluation algebra $\mathcal{E}_E$.

Definition 15. An evaluation algebra $\mathcal{E}_E$ is *polynomially-bounded* if the effort $\mu(n)$ of a single parsing step is polynomially bounded.

Summarizing the arguments above, the total effort spent incorporating the contributions of both grammar and algebra is $O(n^{\max\{\mathrm{iv}(t)+\dim(v)|v\to t\in P\}}\,\mu(n))$. The upper bound for the number of required memory is $O(n^{2\max\{\dim(v)|v\in V\}})\,\ell(n)\zeta(n)$.

Example 5. In the case of optimization algorithms or the computation of a partition function, $h_E(\,.\,)$ returns a single scalar, i.e., $\ell(n) = 1$ and $\zeta(n) \in O(1)$, so that $\mu(n) \in O(1)$ since $m(P)$ is independent of n for every given grammar.

On the other hand, if $\mathcal{E}_E$ returns a representation of every possible parse then $\ell(n)$ may grow exponentially with n and $\zeta(n)$ will usually also grow with n. Non-trivial examples are discussed in some detail at the end of this section.

It is well known that the correctness of dynamic programming algorithms in general depends on the scoring model. Bellman's Principle [5] is a sufficient condition. Conceptually, it states that all optimal solutions of a subproblem are composed of optimal solutions of its subproblems. Giegerich and Meyer [20, Defn. 6] gave an algebraic formulation in the context of ADP that is also suitable for our setting. It can be expressed in terms of sets of $\mathcal{E}_E$-parses of the form $z = \{\mathcal{E}_E(t) \mid t \in T\}$ for some $T \subseteq T_\Sigma$. For simplicity of notation we allow $\mathrm{h}_E(z_i) = z_i$ whenever z_i is a terminal evaluation (and thus strictly speaking would have the wrong type for h_E to be applied).

Definition 16. An evaluation algebra $\mathcal{E}_E$ satisfies Bellman's Principle if every function symbol $f \in F_E$ with arity k satisfies for all sets z_i of $\mathcal{E}_E$-parses the following two axioms:

(B1) $\mathrm{h}_E(z_1 \uplus z_2) = \mathrm{h}_E(\mathrm{h}_E(z_1) \uplus \mathrm{h}_E(z_2))$.

(B2) $\mathrm{h}_E[f(x_1, \ldots, x_k) \mid x_1 \in z_1, \ldots, x_k \in z_k]$
$= \mathrm{h}_E[f(x_1, \ldots, x_k) \mid x_1 \in \mathrm{h}_E(z_1), \ldots, x_k \in \mathrm{h}_E(z_k)]$.

Once grammar and algebra have been amalgamated, and nonterminals are tabulated (the exact machinery is transparent here), the problem instance effectively becomes a total memo function bounded over an appropriate index space. To solve a dynamic programming problem in practice, a function `axiom` (in the convention of [21]) calls the memo function for the start symbol to start the recursive computation and tabulation of results. There is at most one memo table for each nonterminal, and for each nonterminal of an MCFG the index space is polynomially bounded.

Theorem 4. *An MCF-ADP problem instance can be solved with polynomial effort if the evaluation algebra $\mathcal{E}_E$ is polynomially bounded and satisfies the Bellmann condition.*

Proof. The Bellman condition ensures that only optimal solutions of subproblems are used to construct the solution as specified in Def. 12. To see this, we assume that f satisfies (B2). Consider all objects (parses) z in a subproblem. Then $h(z)$ is the optimal set of solutions to the subproblem. Now assume there is a $x' \in z \setminus h(z)$, i.e., a non-optimal parse, so that $f(x') \succ f(x)$ for $x \in h(z)$, i.e., $f(x')$ would be preferred over $f(x)$ by the choice function but has not been selected by h. This, however, contradicts (B2). Thus h selects all optimal solutions.

Since the number of subproblems in an MCF-ADP grammar is bounded by a polynomial and the effort to evaluate each subproblem is also polynomial due the assumption that the evaluation algebra is polynomial bounded, the total effort is polynomial.

We close this section with two real-life examples of dynamic programming applications to RNA folding that have non-trivial contribution to running time and memory deriving from the evaluation algebra. Both examples are also of practical interest for MCF-ADP grammars that model RNA structures with pseudoknots or RNA-RNA interactions. The contributions of the scoring algebra would remain unaltered also for more elaborate MCFG models of RNA structures.

Example: Density of States Algorithm

The parameters μ nor ζ will in general not be in $O(1)$, even though this is the case for the most typical case, namely optimization, counting, or the computation of partition functions. In [14], for example, an algorithm for the computation of the density of states with fixed energy increment is described where the parses correspond to histograms counting the number of structures with given energy. The fixed bin width and fundamental properties of the RNA energy model imply that the histograms are of size $O(n)$; hence $O(\ell(n)) = O(n)$, $O(\mu) = O(n^2)$, and $O(\zeta) = O(1)$. Although there are only $O(n^3)$ sub-parses with $O(n^2)$ memoized entries, the total running time is therefore $O(n^5)$ with $O(n^3)$ memory consumption.

Example: Classified Dynamic Programming

The technique of classified dynamic programming sorts parse results into different classes. For each class an optimal solution is then calculated separately. This technique sometimes allows changing only the choice function instead of having to write a class of algorithms, each specific to one of the classes of classified DP. In classified dynamic programming however both ζ and μ depend on the definition of the classes and we cannot *a priori* expect them to be in $O(1)$. After all, each chosen multiset yields a multiset yielding parses for the next production rule.

The `RNAshapes` algorithm [44] provides an alternative to the usual partition-function based ensemble calculations for RNA secondary structure prediction.

It accumulates a set of coarse-grained shape structures, of which there are exponentially many in the length of the input sequence. Each shape collects results for a class of RNA structures, say all hairpin-like or all clover-leaf structures. The number of shapes scales as q^n, where $q > 1$ is a constant that lies in the range of $1.1 - 1.26$ for the different shape classes discussed in [63].

Thus $\ell(n)\zeta(n)$ with $\ell(n) = q^n$ and $\zeta(n) = O(n)$ is incurred as an additional factor for the memory consumption, with ζ determined by the linear-length encoding for each RNA shape as a string. Under the tacit assumption that the `RNAshapes` algorithm has a binarized normal form grammar (which holds only approximately due to the energy evaluation algebra being used), the running time amounts to $O(n^3\mu(n))$ where $\mu(n)$ amounts to a factor of $\ell(n)^2\,\zeta(n)^2$ due to the binary form. This yields a total running time of `RNAshapes` of $O(n^3q^{2n})$.

Normal-form optimizations

Under certain circumstances the performance can be optimized compared to above worst case estimates. W.l.o.g. assume the existence of a production rule $A \to \alpha\beta\gamma$. If the evaluating function e for this rule has linear structure, i.e. for all triples of parses (a, b, c) returned by α, β, or γ respectively, we have that $e(a, b, c) = a \odot b \odot c$, then, $a \odot b \odot c = (a \odot b) \odot c = a \odot (b \odot c)$ from which it follows that either $B \to \alpha\beta$ or $C \to \beta\gamma$ yields immediately reducible parses. Note that reducibility is required. While it is *always* possible to rewrite a grammar in (C)NF, individual production rules will not necessarily allow an optimizing reduction to a single value (or a set of values in case of more complex choice functions). In the case of CFGs, this special structure of the evaluation algebra allows us to replace $\mathcal{G}$ by its CNF. Thus every production has at most two nonterminals on its r.h.s., leading to an $O(n^3)$ time and $O(n^2)$ space requirement. The same ideas can be used to simplify MCFGs. However, there is no guarantee for an absolute performance bound.

6. Implementation

We have implemented MCFGs in two different ways. An earlier prototype is available at `http://adp-multi.ruhoh.com`. Below, we describe a new implementation as an extension of our generalized Algebraic Dynamic Programming (`gADP`) framework [25, 17, 27, 29] which offers superior running time performance. `gADP` is divided into two layers. The first layer is a set of low-level `Haskell` functions that define syntactic and terminal symbols, as well as operators to combine symbols into production rules. It forms the `ADPfusion` [1] library [25]. This implementation strategy provides two advantages: first, the whole of the `Haskell` language is available to the user, and second, `ADPfusion` is open and can be extended by the user. The `ADPfusion` library provides all capabilities needed to write linear and context-free grammars in one or more dimensions. It has been extended here to handle also MCFGs.

MCFGs and their nonterminals with dimension greater one require special handling. In rules, where higher-dimensional nonterminals are interleaved, `split` syntactic variables have been introduced as a new type of object (i.e. for V_1, U_1, V_2, U_2 below that combine to form U and V respectively). These handle the individual dimensions of each nonterminal when the objects are on the right-hand side of a rule. Rule definitions, where nonterminal objects are used in their entirety, are much easier to handle. Such an object is isomorphic to a multi-tape syntactic variable w.r.t. its index type. As a consequence, we were able to make use of the multi-tape extensions of `ADPfusion` introduced in [26, 27].

In order to further simplify development of formal grammars, we provide a second layer in the form of a high-level interface to `ADPfusion`. `gADP` [28, 29], in particular our implementation of a domain-specific language for multi-tape grammars [2], provides an embedded domain-specific language (eDSL) that transparently compiles into efficient low-level code via `ADPfusion`. This eDSL

[1] `http://hackage.haskell.org/package/ADPfusion`

[2] `http://hackage.haskell.org/package/FormalGrammars`

hides most of the low-level plumbing required for (interleaved) nonterminals. Currently, `ADPfusion` and `gADP` allow writing monotone MCFGs. In particular, (interleaved) nonterminals may be part of multi-dimensional symbols.

To illustrate how one implements an MCF-ADP grammar using `gADP` in practice, we provide the `GenussFold` package [3]. `GenussFold` currently provides an implementation of RNA folding with recursive H-type pseudoknots as depicted in Fig. 1(1). The grammar is a simplified version of the one in [49] and can be read as the usual Nussinov grammar with an additional rule for the interleaved nonterminals and rules for each individual two-dimensional nonterminal:

$$
\begin{aligned}
S &\rightarrow \texttt{(}S\texttt{)}S \mid \texttt{.}S \mid \epsilon \\
S &\rightarrow U_1 V_1 U_2 V_2 \\
U &\rightarrow \left(\begin{smallmatrix} S\texttt{(}U_1 \\ U_2 S\texttt{)} \end{smallmatrix}\right) \mid \left(\begin{smallmatrix} \epsilon \\ \epsilon \end{smallmatrix}\right) \\
V &\rightarrow \left(\begin{smallmatrix} S\texttt{[}V_1 \\ V_2 S\texttt{]} \end{smallmatrix}\right) \mid \left(\begin{smallmatrix} \epsilon \\ \epsilon \end{smallmatrix}\right)
\end{aligned}
$$

This grammar closely resembles the one in Appendix A, except that we allow only H-type pseudoknotted structures. The non-standard, yet in bioinformatics commonly used, MCFG notation above is explained in Appendix A as well.

We have also implemented the same algorithm in the `C` programming language to provide a gauge for the relative efficiency of the code generated by `ADPfusion` for these types of grammars. Since we use the `C` version only for performance comparisons, it does not provide backtracking of co-optimal structures, while `GenussFold` provides full backtracking via automated calculation of an algebra product operator analogous to CF-ADP applications.

The `C` version was compiled using `clang/LLVM 3.5` via `clang -O3`. The Haskell version employs `GHC 7.10.1` together with the `LLVM 3.5` backend. We have used `ghc -O2 -fllvm -foptlo-O3` as compiler options.

The running time behaviour is shown in Fig. 2. Except for very short input strings where the Haskell running time dominates, `C` code is roughly fives times faster.

[3]`http://hackage.haskell.org/package/GenussFold`

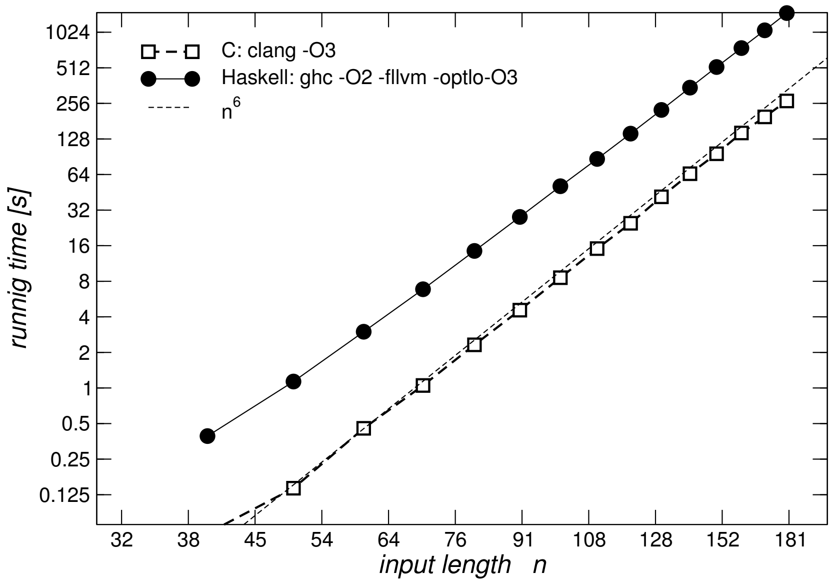


Figure 2: Running time for the $O(n^6)$ recursive pseudoknot grammar. Both the `C` and `Haskell` version make use of the `LLVM` framework for compilation. The `C` version only calculates the optimal score, while the `Haskell` version produces a backtracked dot-bracket string as well. Times are averaged over 5 random sequences of length 40 to 180 in steps of 10 characters.

7. Concluding Remarks

We expand the ADP framework by incorporating the expressive power of MCFGs. Our adaptation is seamless and all concepts known from ADP carry over easily, often without changes, e.g. signatures, tree grammars, yield parsing, evaluation algebras, and Bellman's principle. The core of the generalization from CFG to MCFGs lies in the introduction of rewriting algebras and their use in yield parsing, together with allowing word tuples in several places. As a consequence we can now solve optimization problems whose search spaces cannot be described by CFGs, including the RNA pseudoknotted secondary structure prediction problem.

Our particular implementation in `gADP` also provides additional advanced features. These include the ability to easily combine smaller grammars into a larger, final grammar, via algebraic operations on the grammars themselves and

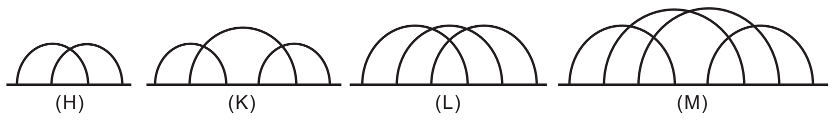


Figure A.3: The four irreducible types of pseudoknots characterize 1-structures. The first two are known as H-type and as kissing hairpin (K), respectively. Each arc in the diagram stands for a collection of nested base pairs.

the possibility to automatically derive outside productions for inside grammars. Since the latter capability is grammar- and implementation-agnostic for single- and multi-tape linear and context-free grammars, it extends to MCFGs as well.

Acknowledgments

We thanks for Johannes Waldmann for his input as supervisor of M.R.'s MSc thesis on the topic and for many critical comments on earlier versions of this manuscript. Thanks to Sarah J. Berkemer for fruitful discussions. CHzS thanks Daenerys.

Appendix A. Example: RNA secondary structure prediction for 1-structures

We describe here in some detail the MCF-ADP formulation of a typical application to RNA structure with pseudoknots. The class of 1-structures is motivated by a classification of RNA pseudoknots in terms of the genus substructures [45, 10, 49]. More precisely, it restricts the types of pseudoknots to irreducible components of topological genus 1, which amounts to the four prototypical crossing diagrams in Fig. A.3 [46, 10, 49]. The `gfold` software implements a dynamic programming algorithm with a realistic energy model for this class of structures [49]. The heuristic `tt2ne` [9] and the Monte Carlo approach `McGenus` [8] address the same class of structures.

For expositional clarity we only describe here a simplified version of the grammar that ignores the distinction between different "loop types" that play a role in realistic energy models for RNAs. Their inclusion expands the grammar to several dozen nonterminals. A naïve description of the class of 1-structures as a grammar was given in [49] in the following form:

$$
\begin{aligned}
&I \to S \mid T\\
&S \to (S)S \mid .S \mid \epsilon\\
&T \to I(T)S\\
&T \to IA_1IB_1IA_2IB_2S\\
&T \to IA_1IB_1IA_2IC_1IB_2IC_2S\\
&T \to IA_1IB_1IC_1IA_2IB_2IC_2S\\
&T \to IA_1IB_1IC_1IA_2ID_1IB_2IC_2ID_2S\\
&\vec{X} \to \binom{(_X IX_1}{X_2I)_X} \mid \binom{(_X}{)_X}
\end{aligned}
$$

Here, the non-terminal I refers to arbitrary RNA structures, S to pseudoknot-free secondary structures, and T to structures with pseudoknots. For the latter, we distinguish the four types of Fig. A.3 (4th to 7th production). For each $X \in \{A, B, C, D\}$ we have distinct terminals $(_X$, $)_X$ that we conceive as different types of opening and closing brackets. We note that this grammar is further transformed in [49] by introducing intermediate non-terminals to reduce the computational complexity. For expositional clarity, we stick to the naïve formulation here. The grammar notation used above is non-standard but has been used several times in the field of bioinformatics – we will call this notation *inlined MCFG*, or *IMCFG*, from now on. While the standard MCFG notation introduced in [55] is based on restricting the allowed form of rewriting functions of generalized context-free grammars, the IMCFG notation is a generalization based on context-free grammars and is mnemonically closer to the structures it represents. While more compact and easier to understand, it is in conflict with our formal integration of MCFGs into the ADP framework. It is, however, simple to transform both notations into each other. Before showing the complete transformed grammar, let us look at a small example.

The transformation from IMCFG to MCFG notation works by creating a function for each production which then matches the used terminals and nonterminals. For example, the IMCFG rule $S \to A_1 B_1 A_2 B_2$ becomes $S \to \mathrm{f}_{abab}[A, B]$ with $\mathrm{f}_{abab}[\binom{A_1}{A_2}, \binom{B_1}{B_2}] = A_1 B_1 A_2 B_2$. For the reverse direction each rewriting function is inlined into each production where it was used.

The IMCFG grammar above generates so-called dot-bracket notation strings where each such string, e.g. `((..).).`, describes an RNA secondary structure. While this is useful for reasoning about those structures at a language level, it is not the grammar form that is eventually used for solving the optimization problem. Instead we need a grammar which, given an RNA primary structure, generates all possible secondary structures in the form of derivation trees, as it is those trees that are assigned a score and chosen from. The IMCFG grammar above has as input a dot-bracket string and generates exactly one or zero derivation trees, answering the question whether the given secondary structure is generated by the grammar. By using RNA bases (`a`, `g`, `c`, `u`) and pairs (`au`, `cg`, `gu`) instead of dots and brackets, such grammar can easily be made into one that is suitable for optimization in terms of RNA secondary structure prediction.

The transformed grammar suitable for solving the described optimization problem is now given as:

$$\dim(I,S,T,U)=1 \qquad \dim(A,B,C,D,P)=2$$

$$\begin{aligned}
I &\to \mathrm{f}_{simple}(S) \mid \mathrm{f}_{knotted}(T)\\
S &\to \mathrm{f}_{paired}(P,S,S) \mid \mathrm{f}_{unpaired}(U,S) \mid \mathrm{f}_{\epsilon}\\
T &\to \mathrm{f}_{knot}(I,P,T,S) \mid\\
&\qquad \mathrm{f}_{knotH}(I,A,I,B,I,I,S) \mid\\
&\qquad \mathrm{f}_{knotK}(I,A,I,B,I,I,C,I,I,S) \mid\\
&\qquad \mathrm{f}_{knotL}(I,A,I,B,I,C,I,I,I,S) \mid\\
&\qquad \mathrm{f}_{knotM}(I,A,I,B,I,C,I,I,D,I,I,I,S)\\
\vec{X} &\to \mathrm{f}_{stackX}(P,I,X,I) \mid \mathrm{f}_{endstackX}(P)\\
P &\to \mathrm{f}_{pair}(\binom{\mathtt{a}}{\mathtt{u}}) \mid \mathrm{f}_{pair}(\binom{\mathtt{u}}{\mathtt{a}}) \mid \mathrm{f}_{pair}(\binom{\mathtt{c}}{\mathtt{g}}) \mid \mathrm{f}_{pair}(\binom{\mathtt{g}}{\mathtt{c}}) \mid \mathrm{f}_{pair}(\binom{\mathtt{g}}{\mathtt{u}}) \mid \mathrm{f}_{pair}(\binom{\mathtt{u}}{\mathtt{g}})\\
U &\to \mathrm{f}_{base}(\mathtt{a}) \mid \mathrm{f}_{base}(\mathtt{g}) \mid \mathrm{f}_{base}(\mathtt{c}) \mid \mathrm{f}_{base}(\mathtt{u})
\end{aligned}$$

585 where $X \in \{A,B,C,D\}$.

$$\begin{aligned}
&\mathcal{E}_R :\\
&\mathrm{f}_{simple}(S) &&= S\\
&\mathrm{f}_{knotted}(T) &&= T\\
&\mathrm{f}_{paired}(\tbinom{P_1}{P_2}, S^{(1)}, S^{(2)}) &&= P_1 S^{(1)} P_2 S^{(2)}\\
&\mathrm{f}_{unpaired}(U,S) &&= US\\
&\mathrm{f}_{\epsilon} &&= \epsilon\\
&\mathrm{f}_{knot}(I, \tbinom{P_1}{P_2}, T, S) &&= I P_1 T P_2 S\\
&\mathrm{f}_{stackX}(\tbinom{P_1}{P_2}, I^{(1)}, \tbinom{X_1}{X_2}, I^{(2)}) &&= \tbinom{P_1 I^{(1)} X_1}{X_2 I^{(2)} P_2}\\
&\mathrm{f}_{endstackX}(\tbinom{P_1}{P_2}) &&= \tbinom{P_1}{P_2}\\
&\mathrm{f}_{pair}(\tbinom{b_1}{b_2}) &&= \tbinom{b_1}{b_2}\\
&\mathrm{f}_{base}(b) &&= b
\end{aligned}$$

$$\begin{aligned}
&\mathrm{f}_{knotH}(I^{(1)}, \binom{A_1}{A_2}, I^{(2)}, \binom{B_1}{B_2}, I^{(3)}, I^{(4)}, S) \\
&\qquad = I^{(1)} A_1 I^{(2)} B_1 I^{(3)} A_2 I^{(4)} B_2 S \\
&\mathrm{f}_{knotK}(I^{(1)}, \binom{A_1}{A_2}, I^{(2)}, \binom{B_1}{B_2}, I^{(3)}, I^{(4)}, \binom{C_1}{C_2}, I^{(5)}, I^{(6)}, S) \\
&\qquad = I^{(1)} A_1 I^{(2)} B_1 I^{(3)} A_2 I^{(4)} C_1 I^{(5)} B_2 I^{(6)} C_2 S \\
&\mathrm{f}_{knotL}(I^{(1)}, \binom{A_1}{A_2}, I^{(2)}, \binom{B_1}{B_2}, I^{(3)}, \binom{C_1}{C_2}, I^{(4)}, I^{(5)}, I^{(6)}, S) \\
&\qquad = I^{(1)} A_1 I^{(2)} B_1 I^{(3)} C_1 I^{(4)} A_2 I^{(5)} B_2 I^{(6)} C_2 S \\
&\mathrm{f}_{knotM}(I^{(1)}, \binom{A_1}{A_2}, I^{(2)}, \binom{B_1}{B_2}, I^{(3)}, \binom{C_1}{C_2}, I^{(4)}, I^{(5)}, \binom{D_1}{D_2}, I^{(6)}, I^{(7)}, I^{(8)}, S) \\
&\qquad = I^{(1)} A_1 I^{(2)} B_1 I^{(3)} C_1 I^{(4)} A_2 I^{(5)} D_1 I^{(6)} B_2 I^{(7)} C_2 I^{(7)} D_2 S
\end{aligned}$$

While being more verbose, the transformation to a tree grammar also has a convenient side-effect: all productions are now annotated with a meaningful and descriptive name, in the form of the given function name. This is useful when talking about specific grammar productions and assigning an interpretation to them in evaluation algebras.

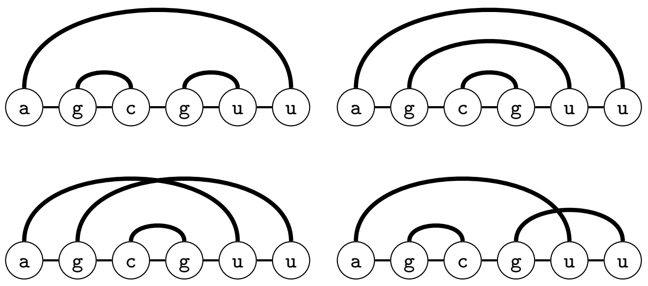
As a simplification of the full energy model from [49] we use base pair counting as the scoring scheme and choose the structure(s) with most base pairs as optimum. This simplification is merely done to ease understanding and keep the example within reasonable length. Before we solve the optimization problem though, let us enumerate the search space for a given primary structure with an evaluation algebra $\mathcal{E}_{DB}$ that returns dot-bracket strings. This algebra has all the functions of the rewriting algebra except for the following:

$$\begin{aligned}
\mathrm{f}_{stackX}(\binom{P_1}{P_2}, I^{(1)}, \binom{X_1}{X_2}, I^{(2)}) &= \binom{(_X I^{(1)} X_1}{X_2 I^{(2)})_X} \\
\mathrm{f}_{endstackX}(\binom{P_1}{P_2}) &= \binom{(_X}{)_X} \\
\mathrm{f}_{pair}(\binom{b_1}{b_2}) &= \binom{(}{)} \\
\mathrm{f}_{base}(b) &= .
\end{aligned}$$

For the primary structure **agcguu** we get:

$$\begin{aligned}\mathcal{G}(\mathcal{E}_{DB}, \texttt{agcguu}) = [\,&\texttt{......,...().,...(.),..()..,.()...,.()().,.()(.),}\\ &\texttt{.(..).,.(()).,.(...),.(.()),.(().),(...).,(.()).,}\\ &\texttt{(().).,(....),(..()),(.().),(()..),(()()),((..)),}\\ &\texttt{((())),([..)],([())],(..[)],(()[)],.(.[)]}\,].\end{aligned}$$

By manual counting we already see what the result of the optimization will be, the maximum number of base pairs of all secondary structures is 3, and the corresponding structures are `(()())`, `((()))`, `([())]`, and `(()[)]`. We can visualize these optimal structures in a more appealing way with Feynman diagrams:



Let us turn to the optimization now. The evaluation algebra $\mathcal{E}_{BP}$ for basepair maximization is given as:

$$\begin{aligned}
&S_{BP} = \mathbb{N}\\
&\mathrm{h}_{BP} = \text{maximum}\\
&\mathrm{f}_{\epsilon} && = 0\\
&\mathrm{f}_{pair}(\left(\begin{smallmatrix}P_1\\P_2\end{smallmatrix}\right)) && = 1\\
&\mathrm{f}_{base}(b) && = 0\\
&\mathrm{f}_{simple}(S) && = S\\
&\mathrm{f}_{knotted}(T) && = T\\
&\mathrm{f}_{paired}(P, S^{(1)}, S^{(2)}) && = P + S^{(1)} + S^{(2)}\\
&\mathrm{f}_{unpaired}(U, S) && = U + S\\
&\mathrm{f}_{knot}(I, P, T, S) && = I + P + T + S\\
&\mathrm{f}_{stackX}(P, I^{(1)}, X, I^{(2)}) && = P + I^{(1)} + X + I^{(2)}\\
&\mathrm{f}_{endstackX}(P) && = P
\end{aligned}$$

and equally for $\mathrm{f}_{knot\{H,K,L,M\}}$ by simple summation of function arguments. The objective function is defined as

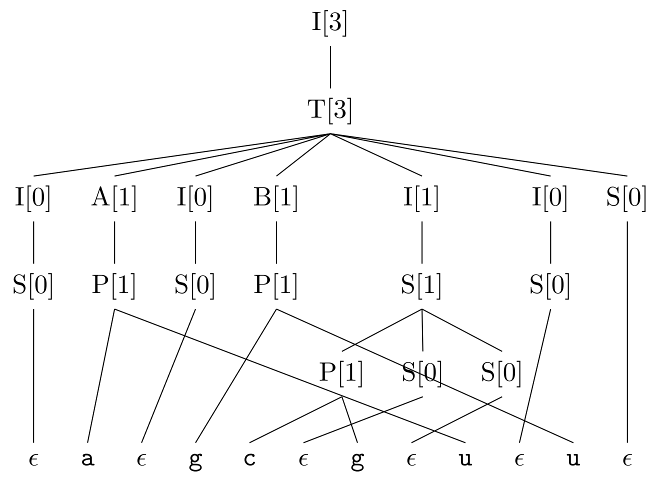
$$\text{maximum}(m) = \begin{cases} \emptyset, & \text{if } m = \emptyset,\\ [\max(m)], & \text{else.} \end{cases}$$

where max determines the maximum of all set elements.

For the primary structure `acuguu` we get:

$$\mathcal{G}(\mathcal{E}_{BP}, \texttt{agcguu}) = [3],$$

matching our expectation. Each search space candidate corresponds to a derivation tree. We can reconstruct the score of a candidate by annotating its tree with the individual scores, here done for the optimum candidate `([()])`:



Coming back to the result, 3, the corresponding RNA secondary structures can be determined by using more complex evaluation algebras. A convenient tool for the construction of those are *algebra products*, further explained in [59].
595 In this case, with the lexicographic algebra product $\mathcal{E}_{BP} * \mathcal{E}_{DB}$ the result would be [(3, `(()())`), (3, `((()))`), (3, `([()])`), (3, `(()[)])`)], that is, a multiset containing tuples with the maximum basepair count and the corresponding secondary structures as dot-bracket strings. We do not describe algebra products further here as our formalism allows their use unchanged.

600 Appendix B. Multisets

A multiset [7] is a collection of objects. It generalizes the concept of sets by allowing elements to appear more than once. A multiset over a set S can be formally defined as a function from S to $\mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \dots\}$. A finite multiset f is such function with only finitely many x such that $f(x) > 0$. The
605 notation $[e_1, \dots, e_n]$ is used to distinguish multisets from sets. The number of times an element occurs in a multiset is called its *multiplicity*. A set can be seen as a multiset where all multiplicities are at most one. If e is an element of a multiset f with multiplicity m, one writes $e \in^m f$. If $m > 0$, one writes $e \in f$.

The cardinality $|f|$ of a multiset is the sum of all multiplicities. The union $f \cup g$ of two multisets is the multiset $(f \cup g)(x) = \max\{f(x), g(x)\}$, the additive union $f \uplus g$ is the multiset $(f \uplus g)(x) = f(x) + g(x)$. The set of multisets over a set S is denoted $\mathcal{M}(S) = \{f \mid f : S \to \mathbb{N}\}$.

As for sets, a builder notation for multisets is introduced. We could find only one reference where such a notation is both used and an attempt was made to formally define its interpretation using mathematical logic [36]. In other cases such notation is used without reference or by referring to list comprehension syntax common in functional programming, implicitly ignoring the list order. Here, we base our notation and interpretation on [36] but extend it slightly to match the intuitive interpretation from list comprehensions. The multiset-builder notation has the form $[x \mid \exists y : P(x,y)]$ where P is a predicate and the multiplicity of x in the resulting multiset is $|[y \mid P(x,y)]|$. For an arbitrary multiset f (or a set seen as a multiset) and $\hat{f} = [y \mid y \in f \wedge P(y)]$ it holds that $\forall y \exists m : y \in^m f \leftrightarrow y \in^m \hat{f}$, that is, the multiplicities of the input multiset carry over to the resulting multiset. The original interpretation in [36] is based on sets being the only type of input, that is, the multiplicity of x was simply defined as $|\{y \mid P(x,y)\}|$. In our case, and similar to list comprehensions, we need multisets as input too. With the original interpretation, we would lose the multiplicity information of the input multisets. Let's look at some examples:

$$M_1 = \{1,2,3\}, M_2 = [1,3,3], M_3 = [2,2]$$

$$\begin{aligned}
[x \bmod 2 \mid x \in M_1] &= [0,1,1] \\
[(x,y) \mid x \in M_2, y \in M_3] &= [(1,2),(1,2),(3,2),(3,2),(3,2),(3,2)]
\end{aligned}$$

With the original interpretation in [36], the results would have been:

$$\begin{aligned}
[x \bmod 2 \mid x \in M_1] &= [0,1,1] \\
[(x,y) \mid x \in M_2, y \in M_3] &= [(1,2),(3,2)]
\end{aligned}$$

Appendix C. Alternative MCFG Definition

We first cite the original MCFG definition (modulo some grammar and symbol adaptations), and then show which changes we applied.

Definition 17. [54] An MCFG is a tuple $\mathcal{G} = (V, \mathcal{A}, Z, R, P)$ where V is a finite set of nonterminal symbols, $\mathcal{A}$ a finite set of terminal symbols disjoint from V, $Z \in V$ the start symbol, R a finite set of rewriting functions, and P a finite set of productions. Each $v \in V$ has a *dimension* $\dim(v) \geq 1$, where $\dim(Z) = 1$. Productions have the form $v_0 \to f[v_1, \dots, v_k]$ with $v_i \in V$, $0 \leq i \leq k$, and $f : (\mathcal{A}^*)^{\dim(v_1)} \times \dots \times (\mathcal{A}^*)^{\dim(v_k)} \to (\mathcal{A}^*)^{\dim(v_0)} \in R$. Productions of the form $v \to f[]$ with $f[] = \begin{pmatrix} f_1 \\ \vdots \\ f_d \end{pmatrix}$ are written as $v \to \begin{pmatrix} f_1 \\ \vdots \\ f_d \end{pmatrix}$ and are called *terminating productions*. Each rewriting function $f \in R$ must satisfy the following condition

(F) Let $\overline{x_i} = (x_{i1}, \dots, x_{i\dim(v_i)})$ denote the ith argument of f for $1 \leq i \leq k$. The hth component of the function value for $1 \leq h \leq \dim(v_0)$, denoted by $f^{[h]}$, is defined as

$$f^{[h]}[\overline{x_1}, \dots, \overline{x_k}] = \beta_{h0} z_{h1} \beta_{h1} z_{h2} \dots z_{hv_h} \beta_{hv_h} \qquad (*)$$

where $\beta_{hl} \in \mathcal{A}^*$, $0 \leq l \leq \dim(v_h)$, and $z_{hl} \in \{x_{ij} \mid 1 \leq i \leq k, 1 \leq j \leq \dim(v_i)\}$ for $1 \leq l \leq \dim(v_h)$. The total number of occurrences of x_{ij} in the right-hand sides of (*) from $h = 1$ through $\dim(v_0)$ is at most one.

Definition 18. [54] The derivation relation $\overset{*}{\Rightarrow}$ of the MCFG $\mathcal{G}$ is defined recursively:

(i) If $v \to a \in P$ with $a \in (\mathcal{A}^*)^{\dim(v)}$ then one writes $v \overset{*}{\Rightarrow} a$.

(ii) If $v_0 \to f[v_1, \dots, v_k] \in P$ and $v_i \overset{*}{\Rightarrow} a_i$ $(1 \leq i \leq k)$, then one writes $v_0 \overset{*}{\Rightarrow} f[a_1, \dots, a_k]$.

In this contribution a modified definition of MCFGs is used. The modified version allows terminals as function arguments in productions and instead disallows the introduction of terminals in rewriting functions. The derivation relation is adapted accordingly to allow terminals as function arguments.

In detail, in our MCFG definition, productions have the form $v_0 \to f[v_1, \ldots, v_k]$ with $v_i \in V \cup (\mathcal{A}^*)^*$, $0 \leq i \leq k$ and rewrite functions have the property that the resulting components of an application is the concatenation of components of its arguments only. The terminals that are "created" by the rewrite rule in Seki's version therefore are already upon input in our variant, i.e., they appear explicitly in the production. Our rewrite function merely "moves" each terminal to its place in the output of the rewrite function. Elimination of the emission of terminals in rewrite functions amounts to replacing equ.(*) in condition F by

$$f^{[h]}[\overline{x_1}, \ldots, \overline{x_k}] = z_{h1} z_{h2} \ldots z_{hv_h} \tag{*}$$

Condition (F) thus becomes just a slightly different way of expressing Definition 1 and Definition 2. Our rephrasing of MCFGs is therefore weakly equivalent to Seki's original definition.

The following shows an equivalent of the example MCFG grammar when using the original definition:

$$\begin{aligned}
&Z \to \mathrm{f}_Z[A, B] && \mathrm{f}_Z[\left(\begin{smallmatrix}A_1\\A_2\end{smallmatrix}\right), \left(\begin{smallmatrix}B_1\\B_2\end{smallmatrix}\right)] = A_1 B_1 A_2 B_2\\
&A \to \mathrm{f}_A[A] \mid \left(\begin{smallmatrix}\epsilon\\\epsilon\end{smallmatrix}\right) && \mathrm{f}_A[\left(\begin{smallmatrix}A_1\\A_2\end{smallmatrix}\right)] \qquad = \left(\begin{smallmatrix}A_1\mathtt{a}\\A_2\mathtt{a}\end{smallmatrix}\right)\\
&B \to \mathrm{f}_B[B] \mid \left(\begin{smallmatrix}\epsilon\\\epsilon\end{smallmatrix}\right) && \mathrm{f}_B[\left(\begin{smallmatrix}B_1\\B_2\end{smallmatrix}\right)] \qquad = \left(\begin{smallmatrix}B_1\mathtt{b}\\B_2\mathtt{b}\end{smallmatrix}\right)
\end{aligned}$$

The terminals moved into the rewriting functions and one additional rewriting function had to be defined to handle the `a` and `b` terminal symbols separately, as the rewriting functions cannot be parameterized over terminals.

When going the reverse way one simply replaces the terminals in the rewriting functions with parameters and adds those to the productions:

$$\begin{aligned}
&Z \to \mathrm{f}_Z[A, B] && \mathrm{f}_Z[\left(\begin{smallmatrix}A_1\\A_2\end{smallmatrix}\right), \left(\begin{smallmatrix}B_1\\B_2\end{smallmatrix}\right)] = A_1 B_1 A_2 B_2\\
&A \to \mathrm{f}_A[A, \left(\begin{smallmatrix}\mathtt{a}\\\mathtt{a}\end{smallmatrix}\right)] \mid \left(\begin{smallmatrix}\epsilon\\\epsilon\end{smallmatrix}\right) && \mathrm{f}_A[\left(\begin{smallmatrix}A_1\\A_2\end{smallmatrix}\right), \left(\begin{smallmatrix}c\\d\end{smallmatrix}\right)] \quad = \left(\begin{smallmatrix}A_1 c\\A_2 d\end{smallmatrix}\right)\\
&B \to \mathrm{f}_B[B, \left(\begin{smallmatrix}\mathtt{b}\\\mathtt{b}\end{smallmatrix}\right)] \mid \left(\begin{smallmatrix}\epsilon\\\epsilon\end{smallmatrix}\right) && \mathrm{f}_B[\left(\begin{smallmatrix}B_1\\B_2\end{smallmatrix}\right), \left(\begin{smallmatrix}c\\d\end{smallmatrix}\right)] \quad = \left(\begin{smallmatrix}B_1 c\\B_2 d\end{smallmatrix}\right)
\end{aligned}$$

If desired, the two now semantically identical rewriting functions f_A and f_B can be replaced by a single one, which would produce the example MCFG as used in this work.

This leads us to the following conclusion:

Theorem 5. *The class of languages produced by Seki's original definition of MCFGs is equal to the class produced by our modified definition.*

Appendix D. MCF-ADP grammar yield languages class

In this section we show that MCF-ADP grammar yield languages are multiple context-free languages, and *vice versa*. We restrict ourselves to MCF-ADP grammars where the start symbol is one-dimensional as formal language hierarchies are typically related to word languages and do not know the concepts of tuples of words.

Each MCF-ADP grammar $\mathcal{G}$ is trivially transformed to an MCFG $\mathcal{G}'$ – the functions of the rewriting algebra become the rewriting functions. By construction, $\mathcal{L}_{\mathrm{y}}(\mathcal{G}) = \mathcal{L}(\mathcal{G}')$, for each derivation in $\mathcal{G}$ there is one in $\mathcal{G}'$, and vice versa. This means that MCF-ADP grammar yield languages are multiple context-free languages. As we will see, the reverse is also true.

Each MCFG $\mathcal{G}'$ can be transformed into an MCF-ADP grammar $\mathcal{G}$ by using the identity function as rewriting function for terminating productions, that is, $V \to \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix}$ becomes $V \to \mathrm{id}_d(\begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix})$ with $\mathrm{id}_d(\begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix}) = \begin{pmatrix} w_1 \\ \vdots \\ w_d \end{pmatrix}$, while all other productions and rewriting functions are reused unchanged. Again, by construction, $\mathcal{L}(\mathcal{G}') = \mathcal{L}_{\mathrm{y}}(\mathcal{G})$. So, multiple context-free languages are MCF-ADP grammar yield languages. We conclude:

The class of yield languages of MCF-ADP grammars is equal to the class of languages generated by MCFGs.

References

[1] Akutsu, T., 2000. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. Discr. Appl. Math. 104, 45–62.

[2] Alkan, C., Karakoc, E., Nadeau, J., Sahinalp, S., Zhang, K., 2006. RNA-RNA interaction prediction and antisense RNA target search. J. Comput. Biol. 13, 267–282.

[3] Bailor, M. H., Sun, X., Al-Hashimi, H. M., 2010. Topology links RNA secondary structure with global conformation, dynamics, and adaptation. Science 327, 202–206.

[4] Bellman, R., 1962. Dynamic programming treatment of the travelling salesman problem. J. ACM 9, 61–63.

[5] Bellman, R. E., 1957. Dynamic Programming. Princeton University Press.

[6] Bird, R., 2010. Pearls of Functional Algorithm Design. Cambridge University Press.

[7] Blizard, W. D., 1988. Multiset theory. Notre Dame Journal of formal logic 30 (1), 36–66.

[8] Bon, M., Micheletti, C., Orland, H., 2013. McGenus: a Monte Carlo algorithm to predict RNA secondary structures with pseudoknots. Nucleic Acids Res. 41, 1895–1900.

[9] Bon, M., Orland, H., 2011. TT2NE: a novel algorithm to predict RNA secondary structures with pseudoknots. Nucleic Acids Res. 39, e93.

[10] Bon, M., Vernizzi, G., Orland, H., Zee, A., 2008. Topological classification of RNA structures. J. Mol. Biol. 379, 900–911.

[11] Chitsaz, H., Salari, R., Sahinalp, S. C., Backofen, R., 2009. A partition function algorithm for interacting nucleic acid strands. Bioinformatics 25, i365–i373.

[12] Condon, A., Davy, B., Rastegari, B., Zhao, S., Tarrant, F., 2004. Classifying RNA pseudoknotted structures. Theor. Comp. Sci. 320, 35–50.

[13] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., 2001. Introduction to Algorithms. MIT Press.

[14] Cupal, J., Flamm, C., Renner, A., Stadler, P. F., 1997. Density of states, metastable states, and saddle points. Exploring the energy landscape of an RNA molecule. In: Gaasterland, T., Karp, P., Karplus, K., Ouzounis, C., Sander, C., Valencia, A. (Eds.), Proceedings of the ISMB-97. AAAI Press, Menlo Park, CA, pp. 88–91.

[15] Dasgupta, S., Papadimitriou, C., Vasirani, U., 2006. Algorithms.

[16] Doudna, J. A., Cech, T. R., 2002. The chemical repertoire of natural ribozymes. Nature 418, 222–228.

[17] Farmer, A., Höner zu Siederdissen, C., Gill, A., 2014. The HERMIT in the stream: fusing stream fusion's concatMap. In: Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation. ACM, pp. 97–108.

[18] Feller, V., 1950. An Introduction to Probability Theory and Its Applications: Volume One. John Wiley & Sons.

[19] Giedroc, D. P., Cornish, P. V., 2009. Frameshifting RNA pseudoknots: structure and mechanism. Virus Res. 139, 193–208.

[20] Giegerich, R., Meyer, C., 2002. Algebraic dynamic programming. In: Algebraic Methodology And Software Technology. Vol. 2422. Springer, pp. 243–257.

[21] Giegerich, R., Meyer, C., Steffen, P., 2004. A discipline of dynamic programming over sequence data. Sci. Computer Prog. 51, 215–263.

[22] Giegerich, R., Schmal, K., 1988. Code selection techniques: Pattern matching, tree parsing, and inversion of derivors. In: Proceedings of the 2nd European Symposium on Programming. Springer, Heidelberg, pp. 247–268.

[23] Goodman, J., 1999. Semiring parsing. Computational Linguistics 25 (4), 573–605.

[24] Gorodkin, J., Heyer, L. J., Stormo, G. D., 1997. Finding the most significant common sequence and structure motifs in a set of RNA sequences. Nucleic Acids Res. 25, 3724–3732.

[25] Höner zu Siederdissen, C., 2012. Sneaking Around concatMap: Efficient Combinators for Dynamic Programming. In: Proceedings of the 17th ACM SIGPLAN international conference on Functional programming. ICFP '12. ACM, New York, NY, USA, pp. 215–226.

[26] Höner zu Siederdissen, C., Hofacker, I. L., Stadler, P. F., 2013. How to Multiply Dynamic Programming Algorithms. In: Brazilian Symposium on Bioinformatics (BSB 2013). Vol. 8213 of Lecture Notes in Bioinformatics. Springer, Heidelberg, pp. 82–93.

[27] Höner zu Siederdissen, C., Hofacker, I. L., Stadler, P. F., 2014. Product Grammars for Alignment and Folding. IEEE/ACM Transactions on Computational Biology and Bioinformatics 99 (PrePrints), 1.

[28] Höner zu Siederdissen, C., Prohaska, S. J., Stadler, P. F., 2014. Dynamic Programming for Set Data Types. In: Brazilian Sympositum on Bioinformatics (BSB 2014). Vol. 8826 of Lecture Notes in Bioinformatics. Springer, Heidelberg.

[29] Höner zu Siederdissen, C., Prohaska, S. J., Stadler, P. F., 2015. Algebraic dynamic programming over general data structures. preliminarily accepted (BMC Bioinformatics).

[30] Hotz, G., Pitsch, G., 1996. On parsing coupled-context-free languages. Theor. Comp. Sci. 161, 205–233.

[31] Huang, F. W. D., Qin, J., Reidys, C. M., Stadler, P. F., 2009. Partition function and base pairing probabilities for RNA-RNA interaction prediction. Bioinformatics 25, 2646–2654.

[32] Huang, F. W. D., Qin, J., Reidys, C. M., Stadler, P. F., 2010. Target prediction and a statistical sampling algorithm for RNA-RNA interaction. Bioinformatics 26, 175–181.

[33] Joshi, A. K., 1985. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In: Dowty, D. R., Karttunen, L., Zwicky, A. M. (Eds.), Natural Language Parsing. Cambridge University Press, pp. 206–250.

[34] Kallmeyer, L., 2010. Parsing Beyond Context-Free Grammars. Springer.

[35] Kato, Y., Seki, H., Kasami, T., 2007. RNA pseudoknotted structure prediction using stochastic multiple context-free grammar. Information and Media Technologies 2, 79–88.

[36] Kotowski, J., Bry, F., Eisinger, N., 2011. A potpourri of reason maintenance methods, unpublished manuscript.

[37] Lefebvre, F., 1995. An optimized parsing algorithm well suited to RNA folding. Proc Int Conf Intell Syst Mol Biol 3, 222–230.

[38] Lefebvre, F., 1996. A grammar-based unification of several alignment and folding algorithms. Proc Int Conf Intell Syst Mol Biol 4, 143–154.

[39] Lorenz, R., Bernhart, S. H., Höner zu Siederdissen, C., Tafer, H., Flamm, C., Stadler, P. F., Hofacker, I. L., 2011. ViennaRNA Package 2.0. Alg. Mol. Biol. 6, 26.

[40] Lyngsø, R. B., Pedersen, C. N., 2000. RNA pseudoknot prediction in energy-based models. J. Comp. Biol. 7, 409–427.

[41] Mathews, D. H., Turner, D. H., 2002. Dynalign: An algorithm for finding secondary structures common to two RNA sequences. J. Mol. Biol. 317, 191–203.

[42] Namy, O., Moran, S. J., Stuart, D. I., Gilbert, R. J. C., Brierley, I., 2006. A mechanical explanation of RNA pseudoknot function in programmed ribosomal frameshifting. Nature 441, 244–247.

[43] Nebel, M. E., Weinberg, F., 2012. Algebraic and combinatorial properties of common RNA pseudoknot classes with applications. J Comput Biol. 19, 1134–1150.

[44] of RNA, A. S., 2004. Giegerich, robert and voß, björn and rehmsmeier, marc. Nucleic Acids Res. 32, 4843–4851.

[45] Orland, H., Zee, A., 2002. RNA folding and large n matrix theory. Nuclear Physics B 620, 456–476.

[46] Pillsbury, M., Orland, H., Zee, A., 2005. Steepest descent calculation of RNA pseudoknots. Phys. Rev. E 72, 011911.

[47] Reghizzi, S. C., Breveglieri, L., Morzenti, A., 2009. Formal Languages and Compilation. Springer.

[48] Reidys, C., 2011. Combinatorial Computational Biology of RNA: Pseudoknots and Neutral Networks. Springer, New York.

[49] Reidys, C. M., Huang, F. W. D., Andersen, J. E., Penner, R. C., Stadler, P. F., Nebel, M. E., 2011. Topology and prediction of RNA pseudoknots. Bioinformatics 27, 1076–1085, addendum in: Bioinformatics 28:300 (2012).

[50] Rivas, E., Eddy, S. R., 1999. A dynamic programming algorithm for RNA structure prediction including pseudoknots. J. Mol. Biol. 285, 2053–2068.

[51] Rivas, E., Lang, R., Eddy, S. R., 2012. A range of complex probabilistic models for RNA secondary structure prediction that includes the nearest-neighbor model and more. RNA 18, 193–212.

[52] Rødland, E. A., 2006. Pseudoknots in RNA secondary structures: Representation, enumeration, and prevalence. J. Comp. Biol. 13, 1197–1213.

[53] Sankoff, D., 1985. Simultaneous solution of the RNA folding, alignment, and proto-sequence problems. SIAM J. Appl. Math. 45, 810–825.

[54] Seki, H., Kato, Y., 2008. On the generative power of multiple context-free grammars and macro grammars. IEICE Trans. Inf. Syst. E91-D, 209–221.

[55] Seki, H., Matsumura, T., Fujii, M., Kasami, T., 1991. On multiple context free grammars. Theor. Comp. Sci. 88, 191–229.

[56] Shieber, S. M., 1985. Evidence against the context-freeness of natural language. Linguistics and Philosophy 8, 333–343.

[57] Skut, W., Krenn, B., Brants, T., Uszkoreit, H., 1997. An annotation scheme for free word order languages. In: Proceedings of the 5th Conference on Applied Natural Language Processing. Assoc. Comp. Linguistics, Stroudsburg, PA, pp. 88–95.

[58] Stabler, E. P., 2004. Varieties of crossing dependencies: structure dependence and mild context sensitivity. Cognitive Science 28 (5), 699–720.

[59] Steffen, P., Giegerich, R., 2005. Versatile dynamic programming using pair algebras. BMC Bioinformatics 6, 224.

[60] Taufer, M., Licon, A., Araiza, R., Mireles, D., van Batenburg, F. H. D., Gultyaev, A., Leung, M.-Y., 2009. `PseudoBase++`: an extension of `PseudoBase` for easy searching, formatting and visualization of pseudoknots. Nucleic Acids Res. 37, D127–D135.

[61] Uemura Y., Hasegawa, A., Kobayashi, S., Yokomori, T., 1999. Tree adjoining grammars for RNA structure prediction. Theor. Comp. Sci. 210, 277–303.

[62] Vijay-Shanker, K., Weir, D. J., Joshi, A. K., 1987. Characterizing structural descriptions produced by various grammatical formalisms. In: Proceedings of the 25th Annual Meeting of the Association for Computational Linguistics (ACL). pp. 104–111.

[63] Voß, B., Giegerich, R., Rehmsmeier, M., 2006. Complete probabilistic analysis of RNA shapes. BMC biology 4 (1), 5.

[64] Waldispühl, J., Behzadi, B., Steyaert, J.-M., 2002. An approximate matching algorithm for finding (sub-)optimal sequences in S-attributed grammars. Bioinformatics 18, 250–259.

[65] Zuker, M., 1989. On finding all suboptimal foldings of an RNA molecule. Science 244, 48–52.