General Reforestation: Parsing Trees and Forests

Efficient Dynamic Programming on Tree-like Data Structures with ADPfusion

Sarah J. Berkemer
Max Planck Institute for Mathematics in the Sciences
bsarah@bioinf.uni-leipzig.de

Peter F. Stadler and Christian Höner zu Siederdissen
Dept. of Computer Science, Univ. Leipzig
{studla,choener}@bioinf.uni-leipzig.de

Abstract

Where string grammars describe how to generate and parse strings, tree grammars describe how to generate and parse trees. We show how to extend *generalized algebraic dynamic programming* to tree grammars. The resulting dynamic programming algorithms are efficient and provide the complete feature set available to string grammars, including automatic generation of outside parsers, algebra products for efficient backtracking, and abstract algebra on grammars for simplified design of grammars via product structures.

The complete parsing infrastructure is available as an embedded domain-specific language in Haskell.

In addition to the formal framework, we provide implementations for both, tree alignment and tree editing. Both algorithms are in active use in, among others, the area of bioinformatics, where optimization problems on trees are of considerable practical importance.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; I.2.8 [*Problem Solving, Control Methods, and Search*]: Dynamic programming

Keywords algebraic dynamic programming, program fusion, functional programming, tree editing, tree alignment

1. Introduction

Consider two versions of the source of your favorite program: an older working version, and one with new features and bugs. The venerable `diff` tool gives the difference between the two files but makes it very hard to track large-scale changes. One solution is to track changes not (only) in the textual representation but to compare the actual abstract syntax trees (Fluri et al. 2007).

Alignment and editing of trees as well as combinatorial optimization problems on trees appear in diverse areas of computer science and its applications, from software engineering, to image analysis, machine translation, and bioinformatics. Tracking source code changes is a direct application of edit operations on ordered trees. The same problem arises for the comparison of RNA secondary structures in computational biology (Höchsmann 2005; Jiang et al. 2002), which have a natural representation as trees. Analogous editing problems for unordered trees arise in image analysis (Xiao et al. 2011; Zieliński and Iwanowski 2013) and phylogenetics.

Exact solution of the (ordered) tree editing problem is an early application of dynamic programming (Tai 1979; Jiang et al. 1995) alongside with optimization problems defined *on* a single given tree such as the "small parsimony problem" (Fitch 1971; Hartigan 1973). The only conceptual difference between these tree problems and the more familiar problems of string editing is the input data structure.

Our main motivation is that algebraic dynamic programming (ADP) separates the complex tasks of specifying the search space via grammar design, and modelling optimizing evaluation strategies via algebra construction. However, ADP was originally conceived as a framework for developing dynamic programming algorithms on strings (Giegerich and Meyer 2002; Höner zu Siederdissen 2012) only. In this paper, we therefore extend the notion of parsing to tree and forest inputs and thus make the framework of generalized algebraic dynamic programming amenable to the diverse universe of tree problems. Our main contributions are the following:

- A formal notion of parsing for forests that remains close to known notions of parsing for strings. The formal symbols introduced in Sec. 3 are syntactic sugar to disambiguate pattern matching within forests.
- Simplified parsing for *linear* grammars on forests. We draw our worked examples from the set of linear grammars, and show how parsing can be non-ambiguous even without additional symbols.
- We extend the generalized algebraic dynamic programming framework to tree inputs. The extension allows us to work with grammar products, and automatic outside grammar constructions immediately.
- The embedding in Haskell provides parsing combinators that allow for stream fusion to happen. A grammar written in our framework will have competitive performance compared to hand-written designs.

To set the stage we begin with a small introduction to algebraic dynamic programming (Sec. 2), generalize the notation of parsing (Sec. 3), and discuss a number of introductory algorithms on trees (Sec. 4).

The notion of an alignment, as explored in Sec. 5, of two or more inputs requires considerably more care (despite the fact that the Needleman-Wunsch algorithm (Needleman and Wunsch 1970) is one of the most simple dynamic programming algorithms). We have selected tree alignment with linear and affine grammars, as well as tree editing as our fully worked examples. These algorithms are advantageous as examples. They are (i) possibly the most simple two-input tree algorithms, (ii) can be described with linear tree grammars (instead of context-free grammars), (iii) have been described in detail, (iv) are useful in practice, and (v) are non-trivial to implement (especially once inside-outside and affine extensions need to be considered).

We complete the treatment of languages on trees with the introduction of a grammar product for linear tree languages (Sec. 6), the affine cost model for tree alignment (Sec. 7), and automatic derivation of inside-outside grammar combinations for tree languages (Sec. 8).

With the complete theory laid out, we detail the implementation in Haskell in Sec. 9. In Sec. 10 we take a look at other relevant work and sketch directions for future research, followed by a short conclusion (Sec. 11).

2. Algebraic Dynamic Programming

Assuming that a parser describes a problem that has both, optimal substructure and overlapping subproblems, dynamic programming is used according to Bellmans principle. This is the principle we make use of in this work as well.

Algebraic dynamic programming (ADP) (Giegerich et al. 2004) is designed around the idea of higher order programming and starts from the realization that a dynamic programming algorithm can be separated into four parts: (i) signature, (ii) grammar, (iii) algebra, and (iv) memoization. These four components are devised and written separately and combined to form the solution.

One advantage of ADP is that the decoupling of the individual problems makes each individual part much easier to design and implement. Another advantage is that different parts can be re-used easily. This is in particular true for the grammar, which defines the search space. Hence it needs to be specified only once, even though one is typically interested in diverse answers for a given DP problem, say, the optimal solution, all sub-optimal solutions within a range, a count of all possible (sub-)optimal solutions, and many more. These variants are entirely taken care of by the algebra.

The signature in ADP provides a set of function symbols that provide the glue between the grammar and the algebra. More formally, we have a finite alphabet $\mathcal{A}$ of symbols over which finite inputs, including the empty input can be formed. For ADP as designed by Giegerich and colleagues (Giegerich and Meyer 2002) inputs are strings. `ADPfusion` (Höner zu Siederdissen 2012) was designed for string inputs as well. With the advent of *generalized* algebraic dynamic programming (Höner zu Siederdissen et al. 2015a,b; Riechert et al. 2016) inputs were generalized to strings and sets, as well as multiple tapes (or inputs) of those.

Given the finite alphabet $\mathcal{A}$ and a sort symbol S, the complete signature Σ with functions $f_i \in \Sigma$ can be defined. Each $f_i :: t_{i1} \rightarrow \ldots t_{in} \rightarrow S$, with $t_{ik} \in \{\mathcal{A}^+, S, \$, -\}$ accepts one or more arguments and produces a result of type S. One additional function, the *choice*, takes a list of values of type S and produces an aggregate, typically again of type S.

A simple left-linear alignment algorithm that solves the inexact string matching or string edit problem could well have the following signature:

$$\begin{aligned} \text{match}: &\ S \rightarrow \binom{\mathcal{A}}{\mathcal{A}} \rightarrow S \\ \text{indel}: &\ S \rightarrow \binom{\mathcal{A}}{-} \rightarrow S \\ \text{delin}: &\ S \rightarrow \binom{-}{\mathcal{A}} \rightarrow S \\ \text{empty}: &\ \binom{\$}{\$} \rightarrow S \\ \text{choice}: &\ [S] \rightarrow S \end{aligned}$$

Given a signature, we may now devise an algebra implementing the signature. An algebra associates each function symbol in the signature with a concrete function. In this case, a simple maximizing scoring system, where we assume that the functions score and malus are given.

$$\begin{aligned} \text{match} \quad s \quad \binom{u}{v} &= s + \text{score}(u,v) \\ \text{indel} \quad s \quad \binom{u}{-} &= s + \text{malus}(u) \\ \text{delin} \quad s \quad \binom{-}{v} &= s + \text{malus}(v) \\ \text{empty} \quad \binom{\$}{\$} &= 0 \\ \text{choice} \quad xs &= \text{maximum } xs \end{aligned}$$

The grammar defines the space of all possible ways how the given input can be produced. Equivalently, the grammar describes all legal *parses* of the given input. The formal grammars we are interested in are inherently ambiguous, which is in marked contrast to formal grammars for computer languages. The ambiguity we are interested in is, however, natural. Given our alignment problem on strings, there is indeed an exponential number of different ways to align two input strings.

The grammar for aligning two strings can be written thus:

$$\binom{X}{X} \rightarrow \underbrace{\binom{X}{X}\binom{a}{a}}_{match} \mid \underbrace{\binom{X}{X}\binom{a}{-}}_{indel} \mid \underbrace{\binom{X}{X}\binom{-}{a}}_{delin} \mid \underbrace{\binom{\$}{\$}}_{empty} \qquad (1)$$

Without an algebra, this potentially exponential search space can not be constrained but with an optimizing algebra, only the optimal solutions to subproblems need to be retained, of which there is typically only a polynomial number for string problems. Once grammar and algebra are amalgamated, we only need to employ a memoization technique for non-terminals such as $\binom{X}{X}$ to prevent repeated re-calculation of subproblems, and are able to extract the optimal alignment score in polynomial $O(nm)$ time. Backtracking via algebra products (not detailed here) then provides the co-optimal alignments themselves.

3. Generalized Parsing

Parsers are usually thought of as algorithms that take a text as input and convert it into some form of a hierarchical structure, the parse tree. The task of the parser is essentially to determine if and how the input can be derived from the start symbol of a given grammar. Our starting point are context-free languages, i.e., production rules of the form $V \rightarrow \alpha$, where V is a non-terminal and α is some string of terminals and/or non-terminals. We will think of each production rule as associated with a specific parser that recognizes exactly the instantiations of a given production.

There is nothing that prevents us from imagining such parsers to operate on arbitrary data structures. The r.h.s. of the production, α, will in general not be a simple string but a complete description how the data object V is to be divided up into components. In the case of CFGs, V is decomposed into an ordered set of intervals, with the understanding that terminals recognize individual input characters and non-terminals correspond to variable-length intervals. To see how this can be generalized we consider here rooted ordered trees and rooted ordered forests.

In this context we have several natural decomposition operations. First, we can decompose a forest F into two or more subforests, i.e., $F \rightarrow F_1 \circ F_2$, $F \rightarrow F_1 \circ F_2 \circ F_3$, etc. In case all trees in F consist of a single node only, we recover strings and concatenation. The analog of a single character is the identification of a non-empty component tree T. A rule of the form $F \rightarrow T \circ F$ separates the leftmost non-empty subtree T from the remainder of the forest. For the decomposition of a tree, however, we have several meaningful operations:

$T \rightarrow x$ the tree consists of single vertex

$T \to x \downarrow F$ separate the root of T from the forest F consisting of the children of T.

$T \to T \llcorner x$ separate the right-most leaf x from the rest of the tree.

$T \to x \lrcorner T$ separate the left-most leaf x from the rest of the tree.

The difference of such rules to CFGs on strings lies in the fact that we now have different ways of concatenating substructures that need to be specified more explicitly. Throughout this contribution we will therefore make the concatenation operators $\circ$, $\llcorner$, $\downarrow$, etc explicit.

4. DP Algorithms on Trees

Let us use the *small parsimony problem* as an example. We are given a rooted tree $\mathcal{T}$ annotated with individual characters taken from a given alphabet $\mathcal{A}$. The task is to find a labeling of interior nodes of $\mathcal{T}$ such that the total number of edges with different labels at the end is minimal. There are two well known dynamic programming solutions to this problem, the algorithm of Fitch (Fitch 1971), later generalized to non-binary trees by Hartigan (Hartigan 1973), and Sankoff's method (Sankoff 1975), which accomodates arbitrary edit costs.

The Fitch/Hartigan version annotates every node u with a subset $\mathcal{V}_u \subseteq \mathcal{A}$ of assignable characters and a score s_u according to the following rules: (i) for each leaf l, $\mathcal{V}_l = \{c_l\}$, the character assigned to it in the input. (ii) For each interior node u, $\mathcal{V}_u$ contains those character(s) that appear with maximum frequency m_u in sets of $\mathcal{V}_v$ assigned to the children v of u and the score $s_u = \sum_{v \in \text{chd}(u)}(s_v + 1) - m_u$, where chd$(u)$ yields the set of children of u. In Sankoff's version, each node is assigned a score $s_u(a)$ for every letter $a \in \mathcal{A}$ according to the rule $s_u(a) = \sum_{v \in \text{chd}(u)} \min_{b \in \mathcal{A}}(s_v(b) + \delta_{ab})$, where δ_{ab} is a user-defined cost of changing characters a to b.

Both the Fitch/Hartigan and the Sankoff algorithm follow a very simple traversal rule on the input tree:

$$T \to x \downarrow F \qquad F \to T \circ F \bigm| \$ \tag{2}$$

where $F \to \$$ terminates at the empty forest. If we know that T is a binary phylogenetic tree as in the original formulation of Fitch's algorithm, then the forest F is either empty or it consists of exactly two trees. In this case the 2nd production becomes $F \to T \circ T \bigm| \$$.

Let us take a moment to make the evaluation algebra and the choice function explicit. The three evaluation functions (from left to right) and choices are:

$$\begin{aligned}
&\texttt{tree} : \text{Maybe } \mathcal{A} \to S \to S\\
&\texttt{forest} : S \to S \to S\\
&\texttt{empty} : \$ \to S\\
&\texttt{choice}_F : \{S\} \to S\\
&\texttt{choice}_T : \{S\} \to S
\end{aligned} \tag{3}$$

Here, `tree` is assumed to have leaves annotated with `Just` the character, which is then parsed, while internal nodes have no annotation (`Nothing`). The implementation for Fitch is then with the type of $S = (s, \mathcal{V})$:

$$\begin{aligned}
&\texttt{tree Nothing}\ s = s\\
&\texttt{tree (Just}\ a)\ s = \{a\}\\
&\texttt{forest}\ l\ r = \texttt{if}\ \ l \cap r \equiv \emptyset\ \ \texttt{then}\ \ l \cup r\ \ \texttt{else}\ \ l \cap r\\
&\texttt{empty}\ \$ = \{\}\\
&\texttt{choice}_F\ \{x\} = x\\
&\texttt{choice}_T\ xs = \text{let}\ frqs = \{(a, \{x \in xs | a \in x\}) | a \in \mathcal{A}\}\\
&\quad m = \max\{r | (\_, r) \in frqs\}\\
&\quad \text{in}\ (\textstyle\sum\{s | (s, \_) \in xs\} + |xs| - m,\\
&\qquad \textstyle\bigcup\{ys | (a, ys) \in frqs, a \equiv m\})
\end{aligned} \tag{4}$$

The evaluation algebra for Fitch's parsimony algorithm requires some discussion. With each tree T we associate a score s_T and a set $\mathcal{V}_T$. On the other hand, each forest comes with the list of the pairs $(s_{T'}, \mathcal{V}_{T'})$ of values that belong to its constituent trees. Different types of decompositions, i.e., different concatenation operators thus also may be associated with different operations of the evaluation algebra and different choice functions. For $\circ$, thus, we simply append the lists of pairs. For $\downarrow$, on the other hand, we have to perform a rather complex operation on this list: First compute, $k(a) := |\{T' | a \in \mathcal{V}_{T'}\}|$ for all $a \in \mathcal{A}$ and $s := \sum_{T'} s_{T'}$ over the entries in the list of length ℓ that evaluates F. Set $k := \max_a k(a)$. Finally we set $\mathcal{V}_{T'} = \mathcal{V}_{\{x\}} \cup \{a | k(a) = k\}$ and $s_{T'} = s + \ell - k$. Here $\mathcal{V}_{\{x\}}$ is the label of a leaf x or the empty set for any interior vertex.

For Sankoff's parsimony algorithm we have a similar situation. The forest F collects the scores of its constituent trees for a given label as a list to which these data are appended stepwise during the $F \to T \circ F$ rule. The $T \to x \downarrow F$ rule then combines these values into a label-dependent score for T.

An interesting example that uses multiple tree-valued nonterminals is the DP solution to the *maximum pairing* or *phylogenetic targeting* problem (Maddison 2000; Arnold and Nunn 2010; Arnold and Stadler 2010). The input is a binary tree $\mathcal{T}$ endowed with a cost matrix D_{pq} for each pair of leaves. The task is to identify a collection of edge-disjoint (and therefore in a binary tree also vertex-disjoint) paths connecting pairs of leaves on $\mathcal{T}$ that minimizes the sum of costs. The key observation for solving this problem is that there are only two situations for each subtree of $\mathcal{T}$: either the path system is enclosed within the subtree, or there is exactly one relevant path leaving the subtree. In the first case we associate the nonterminal T with the subtree, in the second case a nonterminal S. Both T- and S-type subtrees may be leaves. Furthermore, for an S-type tree, exactly one of the two children of its root is also of type S because the path "sticking out from its root" must connect to a unique leaf, while the other child is necessarily of type T. Thus we have the following grammar

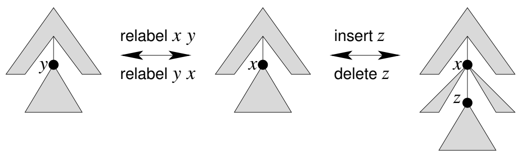
$$\begin{aligned}
T &\to x \downarrow (T \circ T) \bigm| x \downarrow (S \circ S) \bigm| x\\
S &\to x \downarrow (S \circ T) \bigm| x \downarrow (T \circ S) \bigm| x
\end{aligned} \tag{5}$$

Here T also serves as start symbol since there cannot be a path sticking out from the root of the input tree $\mathcal{T}$. While T contains only enclosed paths, S must carry explicit information about the leaf in which the unfinished path ends. This can be taken care of by the evaluation algebra. We assign a value pair (u, s) to each tree where u is the leaf or $\emptyset$ and s is a score. For productions of the form $T \to u$ we use $(\emptyset, 0)$, while $S \to u$ yields the initialization $(i, 0)$. For productions involving the combination of two trees we define $(u', s') * (u'', s'') = (u' * u'', s' + s'' + \phi(u', u''))$ with $u' * u'' = \emptyset$ if either both or none of u' and u'' is $\emptyset$, otherwise $u' * u''$ equals the non-$\emptyset$ argument. $\phi(u', u'') = D_{u',u''}$ if both u' and u'' are leaves and $\phi(u', u'') = 0$ otherwise.

5. Multi-Tape DP for Trees

Pairwise alignment algorithms are conveniently represented in terms of grammars that operate simultaneously on two tapes. The classical Needleman-Wunsch alignment algorithm (Needleman and Wunsch 1970), for instance, has the underlying 2-tape regular language (Höner zu Siederdissen et al. 2015a) that we already encountered in equ.(1). Here we distinguish the termination symbol \$ from the "none"-symbol $-$. Like \$, it emits nothing. As a parsing symbol, however, it succeeds always, while \$ only succeeds on empty substrings. The non-terminal $\binom{X}{X}$ refers to an alignment of prefixes of the input string, while the terminals $\binom{a}{a}$, $\binom{a}{-}$, and $\binom{-}{a}$ correspond to (mis)matches, insertions, and deletions of single characters.

Let us now turn to the analogous problem for forests. There are, in fact, two variants: tree alignment and tree editing. Like string alignments, these problems are based on the three simple edit operations: substitution, insertion, and deletion:



(6)

Naturally, a cost γ^x_y, $\gamma^{\varnothing}_z$, or $\gamma^z_{\varnothing}$, resp., is associated with each edit operation.

5.1 Tree Editing

A *mapping* (Bille 2005) between two ordered forests F_1 to F_2 is a binary relation $M \in V(F_1) \times V(F_2)$ between the vertex sets of the two forests such that for pairs $(x, y), (x', y') \in M$ holds

1. $x = x'$ if and only if $y = y'$. (one-to-one condition)
2. x is an ancestor of x' if and only if y is an ancestor of y'. (ancestor condition)
3. x is to the left of x' if and only if y is to the left of y'. (sibling condition)

The one-to-one condition implies that for each $x \in F_1$ there is a unique "partner" $y \in F_2$, i.e., $(x, y) \in M$, or x has no matching partner at all. With each mapping we can associate the cost

$$\gamma(M) = \sum_{(x,y)\in M} \gamma^x_y + \sum_{y:(x,y)\notin M} \gamma^{\varnothing}_y + \sum_{x:(x,y)\notin M} \gamma^x_{\varnothing} \tag{7}$$

Individual edit operations correspond to "elementary maps". Maps can be composed in a natural manner. Thus every edit script corresponds to a map. Conversely every map can be composed of elementary maps, and thus corresponds to an edit script. Furthermore, the cost of maps is subadditive under composition. As a consequence, minimum cost mappings are equivalent to the minimum cost edit scripts (Tai 1979).

The problem of minimizing $\gamma(M)$ has a rather obvious dynamic programming solution. For a given forest F we note by $F - x$ the forest obtained by deleting x and $F \setminus T(x)$ is forest obtained from F by deleting with x all descendants of x. Note that $T(x) - x$ is the forest consisting of all trees whose roots are the children of x.

$$D(F_1, F_2) = \min \begin{cases} D(F_1 - v_1, F_2) + \gamma^{v_1}_{\varnothing} \\ D(F_1, F_2 - v_2) + \gamma^{\varnothing}_{v_2} \\ D(T(v_1) - v_1, T(v_2) - v_2) + \gamma^{v_1}_{v_2} \\ \qquad +D(F_1 - T(v_1), F_2 - T(v_2)) \end{cases} \tag{8}$$

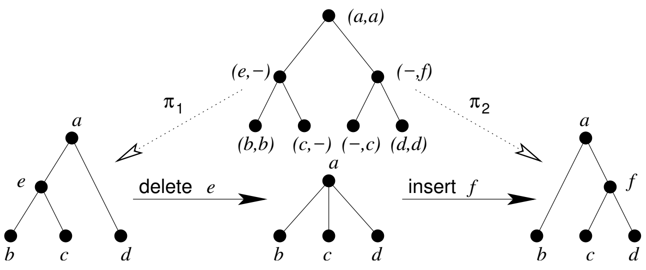


Figure 1: Alignment of two forests F_1 and F_2 and a mapping from F_1 to F_2 that cannot be derived from an alignment.

with $D(\emptyset, \emptyset) = 0$ for two empty forests. A key issue is to implement this algorithm in such a way that only certain classes of subforests need to be evaluated. The corresponding tree editing grammar $\mathcal{E}$ reads

$$\begin{array}{lcl} \binom{T}{T} & \rightarrow & \binom{n}{n} \downarrow \binom{F}{F} \\ \binom{F}{F} & \rightarrow & \binom{F}{F} \circ \binom{T}{T} \;\Big|\; \binom{F}{F} \llcorner \binom{x}{-} \;\Big|\; \binom{F}{F} \llcorner \binom{-}{x} \;\Big|\; \binom{\$}{\$} \end{array} \tag{9}$$

Note that the empty symbol "$-$" acts as neutral element for the concatentation operators, which we take to act component-wise. The grammar is based on the tree editing algorithm of Zhang and Shasha (1989), for which several more efficient implementations exist, see (Dulucq and Tichit 2003) for a detailed analysis of the Zhang-Shasha algorithm.

5.2 Tree Alignments

An alternative way of defining the difference of two forests are *tree alignments* (Jiang et al. 1995). Consider a forest G with vertex labels taken from $(\mathcal{A} \cup \{-\}) \times (\mathcal{A} \cup \{-\})$. Then we obtain restrictions $\pi_1(G)$ and $\pi_2(G)$ by considering only the first or the second coordinate of the labels, respectively, and by then deleting all nodes that are labeled with the gap character $-$ instead of $\varnothing$, see Fig. 1. G is an alignment of the two forests F_1 and F_2 if $F_1 = \pi_1(G)$ and $F_2 = \pi_2(G)$. The cost of the alignment G is the sum of the costs of the label pairs:

$$\gamma(G) = \sum_{(v_1,v_2)\in G} \gamma^{v_1}_{v_2} \tag{10}$$

Every alignment defines a unique mapping, but the converse is not true. The minimum cost alignment is in general more costly than the minimum cost edit script.

We will need a bit of notation. Let F be an ordered forest. By $i : F$ we denote the subforest consisting of the first i trees, while $F : j$ denotes the subforest starting with the $j+1$-th tree. By $F^{\downarrow}$ we denote the forest consisting of the children-trees of the root $v = r_F$ of the first tree in F. $F^{\rightarrow} = F : 1$ is the forest of the right sibling trees of F.

Now consider an alignment A of two forests F_1 and F_2. Let $a = r_A$ be the root of its first tree. We have either:

1. $a = (v_1, v_2)$. Then $v_1 = r_{F_1}$ and $v_2 = r_{F_2}$; $A^{\downarrow}$ is an alignment of $F_1^{\downarrow}$ and $F_2^{\downarrow}$; $A^{\rightarrow}$ is an alignment of $F_1^{\rightarrow}$ and $F_2^{\rightarrow}$.
2. $a = (v_1, -)$. Then $v_1 = r_{F_1}$; for some k, $A^{\downarrow}$ is an alignment of $F_1^{\downarrow}$ and $k : F_2$ and $A^{\rightarrow}$ is an alignment of $F_1^{\rightarrow}$ with $F_2 : k$.
3. $a = (-, v_2)$. Then $v_2 = r_{F_2}$; for some k, $A^{\downarrow}$ is an alignment of $k : F_1$ and $F_2^{\downarrow}$ and $A^{\rightarrow}$ is an alignment of $F_1 : k$ with $F_2^{\rightarrow}$.

These three cases imply the following dynamic programming recursion:

$$S(F_1, F_2) = \min \begin{cases} S(F_1^{\downarrow}, F_2^{\downarrow}) + S(F_1^{\rightarrow}, F_2^{\rightarrow}) + \gamma_{v_2}^{v_1} \\ \min_k S(F_1^{\downarrow}, k : F_2) + S(F_1^{\rightarrow}, F_2 : k) + \gamma_{\varnothing}^{r_{F_1}} \\ \min_k S(k : F_1, F_2^{\downarrow}) + S(F_1 : k, F_2^{\rightarrow}) + \gamma_{r_{F_2}}^{\varnothing} \end{cases} \tag{11}$$

with initial condition $S(\emptyset, \emptyset) = 0$. The formal grammar underlying this recursion is

$$\begin{array}{lllll} \binom{F}{F} & \rightarrow & \binom{T}{T} \circ \binom{F}{F} & \Big| & \binom{\$}{\$} \\ \binom{T}{T} & \rightarrow & \binom{n}{n} \downarrow \binom{F}{F} & \Big| & \binom{-}{n} \downarrow \binom{F}{F} \quad \Big| \quad \binom{n}{-} \downarrow \binom{F}{F} \end{array} \tag{12}$$

It is worth noting that single tape projections of the form $T \rightarrow -{\downarrow}F$ make perfect sense. Since $-$ is a parser that always matches and returns an empty string, which in turn is the neutral element of the concatenator $\downarrow$, this formal production is equivalent to $T \rightarrow F$, i.e., it produces a forest F that happens to consist just of a single tree T.

Höchsmann (2005) describes an efficient variant that makes use of a number of facts that turn this problem into the equivalent of a linear grammar on trees. Trees are separated from their forests from left to right, and forests are always right-maximal. Given a local root node for the tree, and a ternary identifier ({T,F,E}), each forest, tree, and empty forest can be uniquely identified. Trees by the local root, forests by the local root of their left-most tree, and empty forests by the leaves "above" them.

It follows that given a tree with n nodes, one needs no more than $3n$ indices to uniquely identify each substructure. Each production rule gives rise to at most a constant number of parses since the only rule involving more than one non-terminal ($F \rightarrow T \circ F$ for each tape) will remove at most a single tree. The asymptotic running time and space complexity for trees with m and n nodes respectively is then $O(mn)$.

If the nodes of the tree are labelled in pre-order fashion several operations on the forest can be implemented more efficiently by a constant factor. We finally, consider a variant of equ. (12) that distinguishes the match rule $\binom{T}{T} \rightarrow \binom{n}{n}\binom{F}{F}$ with a unique non-terminal $\binom{T}{T}$ on the left-hand side. This rule, which corresponds to the matching of the roots of two substrees, is critical for the calculation of *match probabilities* and will play a major role in Sec. 8. The non-terminals $\binom{T}{Z}$ and $\binom{Z}{T}$ designate insertion and deletion states, respectively.

$$\begin{array}{lll} \binom{F}{F} & \rightarrow & \binom{T}{T} \circ \binom{F}{F} \;\Big|\; \binom{Z}{T} \circ \binom{F}{F} \;\Big|\; \binom{T}{Z} \circ \binom{F}{F} \;\Big|\; \binom{\$}{\$} \\ \binom{T}{T} & \rightarrow & \binom{n}{n} \downarrow \binom{F}{F} \\ \binom{T}{Z} & \rightarrow & \binom{n}{-} \downarrow \binom{F}{F} \\ \binom{Z}{T} & \rightarrow & \binom{-}{n} \downarrow \binom{F}{F} \end{array} \tag{13}$$

6. Tree Alignment and Editing as Products

Multi-tape grammars such as equ. (1) are reminiscent of direct products. This idea has been made precise by Höner zu Siederdissen et al. (2015a) for grammars on strings. In a nutshell, for two sets of productions, the Cartesian set product is formed, and its elements, which are of the form $(A \rightarrow \alpha, B \rightarrow \beta)$ are re-interpreted as 2-tape productions of the form $\binom{A}{B} \rightarrow \binom{\alpha}{\beta}$. For general grammars, the interpretation of the right hand side $\binom{\alpha}{\beta}$ is not clear. For left linear grammars, α and β are of the form Cx, C, or x, where C is a non-terminal and x is terminal. This can always be brought to the form $C'x'$ where C' is either C or $-$ and similarly x' is either x or $-$. Thus $\binom{\alpha}{\beta}$ is of the general form $\binom{C_1}{C_2}\binom{x_1}{x_2}$, where some of the symbols C_i and x_i may equal $-$. This $\otimes$-product is associative and hence the construction generalizes to arbitrary multi-tape grammars.

Furthermore, one can ask whether a given 2-tape grammar such as equ. (1) can be represented as a product of 1-tape grammars. This is not quite the case. In general it is necessary to add and/or remove certain production rules. To this end, $+$ and $-$ operations that act as union and difference operators on the set of production rules, resp., were introduced by Höner zu Siederdissen et al. (2015a). In general, multitape grammars have the structure $S + N - L$, where S includes the main rules of the grammar (such aligning terminals and non-terminals), N comprises rules dealing with an empty input, and L describes the loops, i.e., the non-productive formal productions that need to be eliminated.

Product Representation for Tree Alignment. The extended tree alignment grammar $\mathcal{I}$ of equ. (13) suggests to consider the following simple operations on a single tape: $(F \rightarrow T \circ F \mid Z \circ F \mid \$)$, $(T \rightarrow n \downarrow F)$, and $(Z \rightarrow F)$. Since the tree alignment grammar is left-linear on both tapes (dimensions), it can be written as a grammar product in the following way (with $R \otimes R = R^2$ for a set of rules R):

$$\begin{aligned} \mathcal{I} = (F \rightarrow T \circ F \mid Z \circ F)^2 - (F \rightarrow Z \circ F)^2 \\ + (T \rightarrow n \downarrow F; Z \rightarrow F)^2 - (Z \rightarrow F)^2 \end{aligned} \tag{14}$$

Here we have used (i) that $-$ is a neutral element for all concatenation operators, and (ii) that the $\otimes$ product is properly defined only for a given concatenation operator. That is we have no idea how a term like $\binom{A}{B} \rightarrow \binom{C \circ D}{x \downarrow F}$ should be interpreted.

Product Representation for Tree Editing. The single tape rules appearing in the tree edit grammar $\mathcal{E}$, equ. (9) are $(F \rightarrow F \circ T \mid F \llcorner x \mid F \mid \$)$ and $(T \rightarrow n \downarrow F)$. The key observation is that the tree decomposition operators traverse the tree in two different *directions*: while $\circ$ and $\llcorner$ proceed to siblings, $\downarrow$ moves down, continuing with the children of the current node. The grammar of the tree editing algorithm is therefore linear only if both directions are considered independently. A product representation that treats the different types of concatenations separately can nevertheless be obtained (again with $R \otimes R = R^2$):

$$\begin{aligned} \mathcal{E} =& (F \rightarrow F \circ T)^2 + (F \rightarrow F \llcorner x \mid F)^2 \\ &+ (T \rightarrow n \downarrow F)^2 - (F \rightarrow F)^2 + (F \rightarrow \$)^2 \end{aligned} \tag{15}$$

7. The Affine Gap Cost Model for Alignments

The simple linear scoring of gaps in alignments as in the original formulation by Needleman and Wunsch (1970) is often a poor model in computational biology. Instead, one typically uses affine gap cost with a large constribution for opening a gap and small contributions for extending the gaps. The sequence alignment problem with affine gap costs was solved by Gotoh (1982). The corresponding formal grammar, in the version used by Höner zu Siederdissen et al. (2015a) reads

$$\begin{array}{lll} M & \rightarrow & M\binom{u}{v} \;\Big|\; D\binom{u}{v} \;\Big|\; I\binom{u}{v} \;\Big|\; \binom{\$}{\$} \\ D & \rightarrow & M\binom{u}{\text{-}} \;\Big|\; D\binom{u}{.} \;\Big|\; I\binom{u}{\text{-}} \\ I & \rightarrow & M\binom{\text{-}}{v} \;\Big|\; D\binom{\text{-}}{v} \;\Big|\; I\binom{.}{v} \end{array} \tag{16}$$

where u and v are terminal symbols, '-' denotes the opening of a gap, and '.' denotes the extension of gap, typically scored differently. Considering only one tape or input dimension, a deletion is denoted by a leading '-' followed by a number of '.' characters, e.g. a sequence '-....'.

For trees, the situation is more complicted. A node in a tree may have siblings as well as children. Once a node has been aligned to an initial gap symbol ('$-$') both its siblings and its children are extending the initial gap. Compared to the three rules for matching, deletion and insertion, we now have to deal with seven different cases. In addition, we explicitly write each non-terminal in such a way as to show the state of each tape. In (Schirmer 2011; Schirmer and Giegerich 2011) seven rules for affine gap costs in forests are formulated based on different modes of scoring: no-gap mode, parent-gap mode and sibling-gap mode. Parent and sibling mode indicate that the preceding node (either parent or sibling node) was considered a deletion. Correspondingly, the non-terminal symbol F denotes a no-gap state, P denotes a parent gap, and G denotes a sibling gap. This means that in P mode a gap was introduced in a node further toward the root, while in G mode a gap was introduced in a sibling. In both modes, an unbroken chain of deletions then follows on that tape.

$$\begin{array}{lcl}
\binom{F}{F} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{F}{G} \;\Big|\; \binom{Z}{T}\circ\binom{G}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{P}{F} & \rightarrow & \binom{T}{T}\circ\binom{P}{F} \;\Big|\; \binom{T}{Z}\circ\binom{P}{G} \;\Big|\; \binom{\tilde{Z}}{T}\circ\binom{P}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{F}{P} & \rightarrow & \binom{T}{T}\circ\binom{F}{P} \;\Big|\; \binom{T}{\tilde{Z}}\circ\binom{F}{P} \;\Big|\; \binom{Z}{T}\circ\binom{G}{P} \;\Big|\; \binom{\$}{\$} \\
\binom{G}{F} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{P}{G} \;\Big|\; \binom{\tilde{Z}}{T}\circ\binom{G}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{F}{G} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{\tilde{Z}}\circ\binom{F}{G} \;\Big|\; \binom{Z}{T}\circ\binom{G}{P} \;\Big|\; \binom{\$}{\$} \\
\binom{P}{G} & \rightarrow & \binom{T}{T}\circ\binom{P}{F} \;\Big|\; \binom{T}{\tilde{Z}}\circ\binom{P}{G} \;\Big|\; \binom{\tilde{Z}}{T}\circ\binom{P}{F} \\
\binom{G}{P} & \rightarrow & \binom{T}{T}\circ\binom{F}{P} \;\Big|\; \binom{T}{\tilde{Z}}\circ\binom{F}{P} \;\Big|\; \binom{\tilde{Z}}{T}\circ\binom{G}{P} \\
\binom{T}{T} & \rightarrow & \binom{n}{n}\downarrow\binom{F}{F} \\
\binom{T}{Z} & \rightarrow & \binom{n}{-}\downarrow\binom{F}{P} \\
\binom{Z}{T} & \rightarrow & \binom{-}{n}\downarrow\binom{P}{F}
\end{array} \tag{17}$$

$$\begin{array}{lcl}
\binom{T}{\tilde{Z}} & \rightarrow & \binom{n}{\cdot}\downarrow\binom{F}{P} \\
\binom{\tilde{Z}}{T} & \rightarrow & \binom{\cdot}{n}\downarrow\binom{P}{F}
\end{array} \tag{18}$$

This grammar supports different scoring functions for parent and sibling gaps. Gap opening and gap extension can be distinguished explicitly by including the two additional rules given in equ. (18). They are useful in particular to produce a more expressive output in the backtracing step.

In most applications, however, there is little reason to distinguish the parent and sibling mode gaps in the scoring function. Omitting also the explicit rules for gap extension, the grammar can be simplified considerably, see also Schirmer (2011); Schirmer and Giegerich (2011). Here, $\binom{F}{F}$ denotes the non-gap mode, whereas the gap-mode is represented by mixed terms. In particular, $\binom{T}{Z}$ and $\binom{Z}{T}$ open gaps, while the remaining mixed terms refer to gap extensions.

$$\begin{array}{lcl}
\binom{F}{F} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{F}{G} \;\Big|\; \binom{Z}{T}\circ\binom{G}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{P}{F} & \rightarrow & \binom{T}{T}\circ\binom{P}{F} \;\Big|\; \binom{T}{Z}\circ\binom{P}{F} \;\Big|\; \binom{Z}{T}\circ\binom{P}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{F}{P} & \rightarrow & \binom{T}{T}\circ\binom{F}{P} \;\Big|\; \binom{T}{Z}\circ\binom{F}{P} \;\Big|\; \binom{Z}{T}\circ\binom{F}{P} \;\Big|\; \binom{\$}{\$} \\
\binom{G}{F} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{F}{G} \;\Big|\; \binom{Z}{T}\circ\binom{G}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{F}{G} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{F}{G} \;\Big|\; \binom{Z}{T}\circ\binom{G}{F} \;\Big|\; \binom{\$}{\$} \\
\binom{T}{T} & \rightarrow & \binom{n}{n}\downarrow\binom{F}{F} \\
\binom{T}{Z} & \rightarrow & \binom{n}{-}\downarrow\binom{F}{P} \\
\binom{Z}{T} & \rightarrow & \binom{-}{n}\downarrow\binom{P}{F}
\end{array} \tag{19}$$

The rules for $\binom{F}{F}$, $\binom{G}{F}$, and $\binom{F}{G}$ produce the same cases on their right hand sides. The difference are the l.h.s. cases, which distinguish between no-gap mode and gap mode, thus between affine extension cost and gap opening cost. Additionally, the rules expressing parent gap modes $\binom{P}{F}$ and $\binom{F}{P}$ are recursively calling themselves. Following these observations, the grammar can be further simplified by summarizing several rules and making gap opening and gap extension costs implicit.

To this end we write F for the non-gap mode, R for the parent-gap mode and Q for sibling-gap mode. Instead of adding the gap costs in the explicit cases for gaps, e.g. $\binom{T}{Z}$ and $\binom{Z}{T}$, affine gap costs are now added in the rules $(R \rightarrow T \circ R \;\big|\; Z \circ R)$ and $(Q \rightarrow T \circ Q \;\big|\; Z \circ Q)$ whereas gap opening costs are applied for $(F \rightarrow Z \circ Q)$. No costs are added for the rule $(Q \rightarrow T \circ F)$. As the algorithm applies the scoring for each rule on both tapes at the same time, distinguishing between different tapes is not needed anymore as soon as the cases appear on both tapes symmetrically. Thus, the grammar can be simplified and scoring is applied implicitly such that we only distinguish between no-gap mode, gap-opening mode and gap extension:

$$\begin{array}{lcl}
\binom{F}{F} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{Q}{Q} \;\Big|\; \binom{Z}{T}\circ\binom{Q}{Q} \;\Big|\; \binom{\$}{\$} \\
\binom{Q}{Q} & \rightarrow & \binom{T}{T}\circ\binom{F}{F} \;\Big|\; \binom{T}{Z}\circ\binom{Q}{Q} \;\Big|\; \binom{Z}{T}\circ\binom{Q}{Q} \;\Big|\; \binom{\$}{\$} \\
\binom{R}{R} & \rightarrow & \binom{T}{T}\circ\binom{R}{R} \;\Big|\; \binom{T}{Z}\circ\binom{R}{R} \;\Big|\; \binom{Z}{T}\circ\binom{R}{R} \;\Big|\; \binom{\$}{\$} \\
\binom{T}{T} & \rightarrow & \binom{n}{n}\downarrow\binom{F}{F} \\
\binom{T}{Z} & \rightarrow & \binom{n}{-}\downarrow\binom{R}{R} \\
\binom{Z}{T} & \rightarrow & \binom{-}{n}\downarrow\binom{R}{R}
\end{array} \tag{20}$$

8. Inside and Outside Grammars

An inside parser will readily produce two kinds of results. Of course a globally optimal solution, say the alignment distance between two trees, can be obtained. Alternatively, the partition function $Z = \sum_{\omega} e^{s(\omega)/T}$ can be computed. Here, the sum runs over all configurations ω, $s(\omega)$ is the score of ω and T is a scaling temperature. For $T \rightarrow 0$, Z just counts the number of optimal solutions, for $T \rightarrow \infty$, all conformations are treated equally. The partition function Z thus provides access to a probabilistic model. This view plays a key role in practical applications.

While we typically cannot enumerate all possible states because of the exponentially large size of the search space, it is often possible to describe a polynomial number of subproblems that provide salient information about the solutions. For instance, consider the alignment of the two trees in Fig. 2. Here we can ask for the probability of two nodes being matched with each other over all possible alignments. To do this, we would need to know the partition function Z' of the alignment problem restricted to a single prescribed match; the desired probability is then simply Z'/Z. The tool by which we can make such statements in a principled manner is the combination of the *Inside* grammar together with an *Outside* grammar. Conceptually Outside grammars describe decompositions, i.e., parses of the complements of Inside objects. Restricted problems thus are specified by fixed subdivision of the input into a part that is treated by the Inside and the complementary part that is treated by the Outside part.

Outside grammars are typically more complex, having both more rules and a more complex index space, which makes them a bit of a challenge to construct by hand. Because they operate on suitably defined complements of the inside objects, however, there is a completely generic construction of outside grammars (Höner zu Siederdissen et al. 2015b). It suffices to write down the rules for index space transformation once for each symbol. The machinery then constructs the correct algorithm. The use of automatic construction has the added benefit that the evaluation algebra used for the inside case can be re-used for the outside case, as both grammars share the same signature. Hence they have not only completely isomorphic types, but also the same semantics of the functions is used for evaluation of each parse.

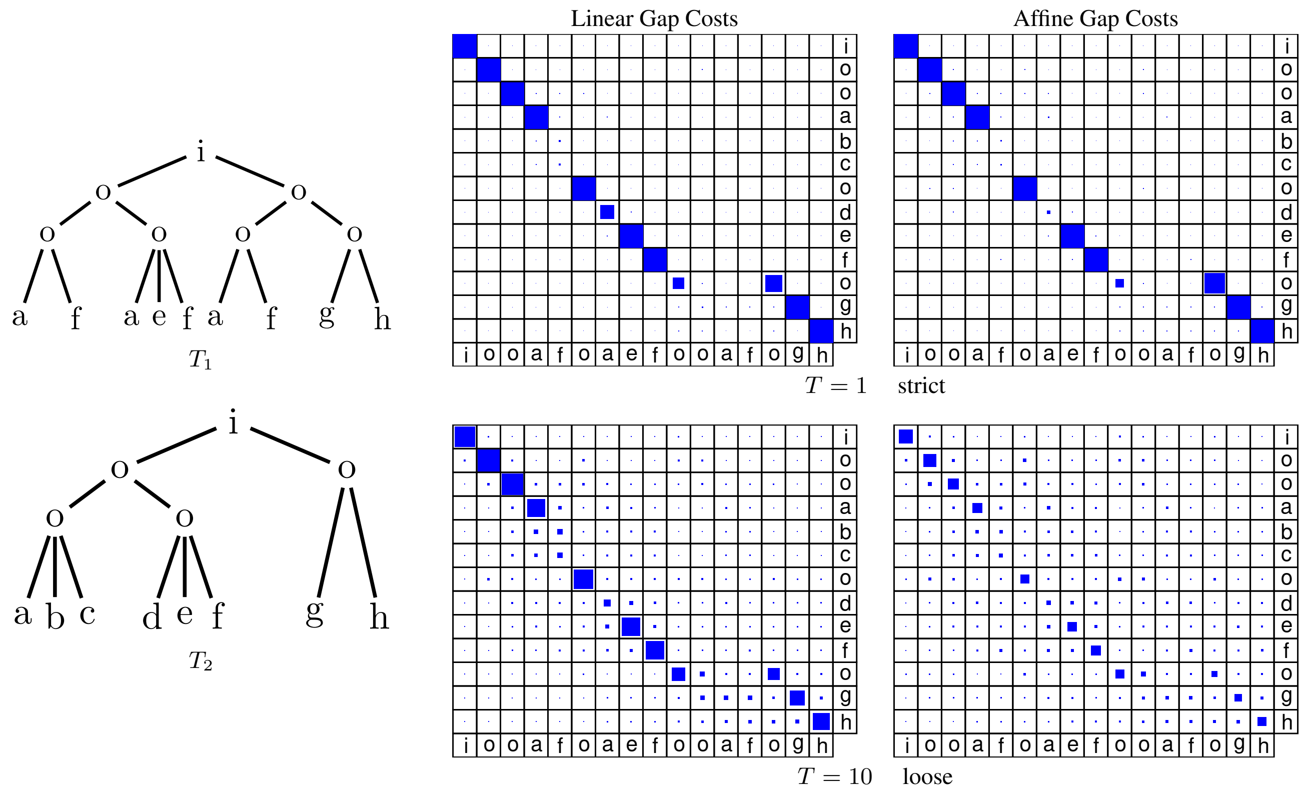


Figure 2: Match probabilities for alignments of the trees T_1 and T_2. The inner vertices are denoted by o, the root is i, and all leaves are labeled by different letters. Results are compared for linear (l.h.s. column) and affine gaps costs (r.h.s. column), and for two different choices of the scaling temperature T. The size of the squares scales logarithmically with the probability as $1/(1 - \log p)$.

Below, we give the outside grammar (with start symbol $\binom{F^*}{F^*}$, and outside "epsilon" symbols σ – given that σ in an outside grammar terminates with *full* input, not the empty input) for the simple linear-cost tree alignment problem (Eq. 13) and combine inside and outside grammar to yield match probabilities.

$$\begin{aligned}
\binom{F^*}{F^*} &\rightarrow \binom{n}{n}\downarrow\binom{T^*}{T^*} \;\Big|\; \binom{n}{-}\downarrow\binom{T^*}{Z^*} \;\Big|\; \binom{-}{n}\downarrow\binom{Z^*}{T^*} \\
&\;\Big|\; \binom{T}{T}\circ\binom{F^*}{F^*} \;\Big|\; \binom{T}{Z}\circ\binom{F^*}{F^*} \;\Big|\; \binom{Z}{T}\circ\binom{F^*}{F^*} \\
&\;\Big|\; \binom{\sigma}{\sigma} \\
\binom{T^*}{T^*} &\rightarrow \binom{F^*}{F^*}\circ\binom{F}{F} \\
\binom{T^*}{Z^*} &\rightarrow \binom{F^*}{F^*}\circ\binom{F}{F} \\
\binom{Z^*}{T^*} &\rightarrow \binom{F^*}{F^*}\circ\binom{F}{F}
\end{aligned} \tag{21}$$

The probability that nodes i in T_1 and j in T_2 are matched in any given alignment is now simply

$$\binom{T}{T}_{ij}\binom{T^*}{T^*}_{ij}/\binom{F}{F}_{0,0}$$

which again can be expressed as a grammar: $P \rightarrow_p \binom{T}{T}\binom{T^*}{T^*}$ with evaluation function $p = \lambda i.\lambda o \rightarrow io/z$, with $z = \binom{F}{F}_{0,0}$. This is why we separated out $\binom{T}{T}$ in the construction of the alignment grammar, as $\binom{T}{T}$ holds the total accumulated inside weight with a match (i, j), while $\binom{T^*}{T^*}$ holds the total accumulated outside weight where the *next* match will be at (i, j).

Inside-Outside for Affine Gap Costs. The combined Inside-Outside algorithm with an affine gap cost model can be implemented in complete analogy to the linear model. One designs the inside grammar and the outside grammar is constructed automatically. This yields an algorithm that computes the match probabilites using the affine gap cost model. The example in Fig. 2 shows the effect of the scaling temperature T.

Due to the small size of the inputs, alignment with linear and affine gap costs produces similar results. Depending on the temperature, the alignment with the single highest probability mass will dominate (T small) or many sub-optimal solutions will show up with significant probability (T large). For higher temperatures, the cost of opening a gap becomes less pronounced yielding a much less constrained probability space than for low temperatures – or the linear model at high temperatures.

9. Implementation

Now that all required theory is in place, we turn toward implementation. During the exposition we make several simplifications that are not present in the actual implementation. Several type class names have been shortened to fit the double-column layout. The new names have been selected to be easily recognized in the actual code, such as `class Elem` instead of `class Element`. We also removed all references to monadic parametrization. The whole framework works for any monad, giving a type class `class (Monad m) => MkStream m x i` below. While this is powerful in practice, it is not required in the exposition. As such, the `MkStream` type class will be presented as `MkStream x i`.

We represent an ordered forest as a set of vectors:

```
data TreeOrder = Pre | Post
data Forest (p :: TreeOrder) a where
  Forest
    { label     :: Vector a
    , parent    :: Vector Int
    , children  :: Vector (UVector Int)
    , lsib      :: Vector Int
    , rsib      :: Vector Int
    , roots     :: Vector Int
    } -> Forest p a
```

Each node has a unique index in the range $0 \ldots |\text{label}| - 1$, and is associated with a label of type `a`, a parent, a vector of children, and its left and right sibling. In addition, we store a vector of roots for our forest. The `Vector` data type is assumed to have an index operator `(!)` with constant time random access.

Forests are also typed as either being in pre-order or post-order sorting of the nodes. The linear grammars presented beforehand make direct use of the given ordering and we can easily guarantee that only inputs of the correct type are being handled. The underlying machinery for tree alignments is somewhat more complex and our choice for an example implementation in this paper. The implementation for tree edit grammars is available in the sources.

Tree Alignment. The tree alignment grammar is a right-linear tree grammar which admits a running time and space optimization to linear complexity for each tape. The grammar itself has been presented in Sec. 5.2. Here, we discriminate between three types of sub-structures within a forest

```
data TF = T | F | E
  deriving (Enum,Bounded)
```

A forest can be empty (`E`) at any node k, be a single tree `T` with local root k, or be the subforest `F` starting with the left-most tree rooted at k and be right-maximal. We can thus uniquely identify sub-structures with the index structure

`data TreeIxR a t = TreeIxR (Forest Pre a) Int TF`

for right-linear tree indices, including the above-mentioned `Forest` structure. The phantom type `t` allows us to tag the index structure as being used in an inside (`I`) or outside (`O`) context.

Since the `Forest` structure is static and not modified by each parsing symbol, it is beneficial to separate out the static parts. `ADPfusion` provides a data family for running indices `RunIx i` to this end. With a new structure,

`data instance RunIx (TreeIxR (Forest Pre a) I) = Ix Int TF`

we capture the variable parts for inside grammars.

Not only do we not have to rely on the compiler to lift the static `Forest Pre a` out of the loops but we also can implement a variant for outside indices.

```
data instance RunIx (TreeIxR (Forest Pre a) O)
  = Ix Int TF Int TF
```

Outside index structures need to carry around a pair of indices. One `(Int,TF)` pair captures index movement for inside tables and terminal symbols in the outside grammar, the other pair captures index movement for outside tables.

Regular shape-polymorphic arrays in Haskell (Keller et al. 2010) provide inductive tuples. An inductive tuple `(Z:.Int:.Int)` is isomorphic to an index in $\mathbb{Z}^2$, with `Z` representing dimension 0, and `(:.)` constructs inductive tuples. A type class `Shape` provides, among others, a function `toIndex` of the index type to a linear index in $\mathbb{Z}$. We make use of an analogous type class (whose full definition we elude) that requires us to implement both a `linearIndex` and a `size` function. The linear index with largest (u, v) and current (k, t) index into the forest

```
linearIndex (TreeIxR _ u v) (TreeIxR _ k t)
  = (fromEnum v + 1) * k + fromEnum t
```

and size function

`size (TreeIxR _ u v) = (fromEnum v + 1) * (u+1)`

together allow us to define inductive tuples with our specialized index structures as well. `linearIndex` and `size` are defined for the tuple constructor `(a :. b)` and combine both functions for `a` and `b` in such a way as to allow us to write grammars for *any* fixed dimension.

Parsing with Terminal Symbols. In a production rule $L \rightarrow tR$, terminal symbols, denoted t perform the actual parsing, while non-terminals (L, R) provide recursion points (and memoization of intermediate results).

For tree grammars, three terminal symbols are required. The \$-parser is successful only on empty substructures, the deletion symbol ($-$) performs *no* parse on the given tape, while only the `Node` parser performs actual work. The first two parsers can be constructed easily, given a node parser. Hence, we give only its construction here.

First, we need a data constructor

```
data Node r x where
  Node (Vector x -> Int -> r) (Vector x) -> Node r x
```

that allows us to define a function from an input vector with elements of type x to parses of type r, and the actual input vector of type x. While quite often $x \sim r$, this generalization allows to provide additional context for the element at a given index if necessary.

We can now bind the labels for all nodes of given input forest f to a node terminal:

`let node frst = Node (!) (label frst)`

where `(!)` is the index operator into the label vector.

Construction of a Parser for a Terminal Symbol. Before we can construct a parser for a rule like $L \rightarrow tR$ we require a bit of machinery in place. First, we decorate the rule with an evaluation function (f) from the interface (see Sec. 2), thus turning the rule into $L \rightarrow_f tR$. The left-hand side will not play a role in the construction of the parser as it will only be bound to the result of the parse.

For the right-hand side we need to be able to create a stream of parses. Streams enable us to use the powerful stream-fusion framework (Coutts et al. 2007). We will denote streams with angled brackets `<x>` akin to the usual list notation of `[x]` in Haskell.

The elements x of our parse streams are somewhat complicated. The element type class

```
class Elem x i where
  data Elm x i :: *
  type Arg x   :: *
  getArg :: Elm x i -> Arg x
  getIdx :: Elm x i -> RunIx i
```

allows us to capture the important structure for each symbol in a parse. For the above terminal symbol `Node` we have

```
instance (Elem ls i) => Elem (ls , Node r x) i where
  data Elm (ls,Node r x) i = ENd r (RunIx i) (Elm ls i)
  type Arg (ls,Node r x)   = (Arg ls , r)
  getArg (ENd x _ ls)      = (getArg ls , r)
  getIdx (ENd _ i _ )      = i
```

Each `Elem` instance is defined with variable left partial parses (`ls`) and variable index (`i`) as all index-specific information is encoded in `RunIx` instances.

Given a partial parse of all elements to the left of the node symbol in `ls`, we extend the `Elm` structure for `ls` inductively with the structure for a `Node`. The Haskell compiler is able to take a stream of `Elm` structures, i.e. `<Elm x>` and erase all type constructors during optimization. In particular, the data family constructors like `ENd` are erased, as well as the running index constructors, say `Ix Int TF`. This means that

```
Elm (ls, Node r x)
    (RunIx (TreeIxR (Forest Pre a) t))
```

has a run time representation in a stream fusion stream that is isomorphic to
`(r,(Int,TF),ls)`
and can be further unboxed, leading to efficient, tight loops. In case forest labels are unboxable `r` will be unboxed as well. The recursive content in `ls` receives the same treatment, leading to total erasure of all constructors.

The `MkStream` type class does the actual work of turning a right-hand side into a stream of elements. Its definition is simple enough:

```
class MkStream x i where
  mkStream :: x -> StaticVar -> i -> <Elm x i>
```

Streams are to be generated for symbols `x` and some index structure `i`. Depending on the position of `x` in a production rule, it might be considered *static* – having no right neighbor, or variable. In $L \rightarrow D \circ R$ we have D in variable and R in static position.
`data StaticVar = Static | Var`
takes care of this below.

We distinguish two types of elements x for which we need to construct streams. Non-terminals always produce a single value for an index i, independent of the the dimensionality of i. Terminal symbols need to deconstruct the dimension of i and produce independent values (or parses) for each dimension or input tape.

Multi-tape Terminal Symbols. For terminal symbols, we introduce yet another inductive tuple, `(:|)` with zero-dimensional symbol `M`. We can now construct a terminal that parses a node on each tape via
`M:|Node (!) i1:|Node (!) i2`.
The corresponding `MkStream` instance hands of work to another type class `TermStream` for tape-wise handling of each terminal:

```
instance (TermStream (ts:|t) => MkStream (ls , ts:|t) i
  where mkStream (ls , ts:|t) c i
          = fromTermStream
          . termStream (ts:|t) c i
          . prepareTermStream
          $ mkStream ls i
```

The required packing and unpacking is done by `fromTermStream` and `prepareTermStream`, and the actual handling is done via `TermStream` with the help of a type family to capture each parse result.

```
type family   TArg x :: *
type instance TArg M = Z
type instance TArg (ts:|t) = TermArg ts :. TermArg t

class TermStream t s i where
  termStream :: t -> StaticVar -> i
             -> <(s,Z,Z)> -> <(s,i,(TArg t))>
```

Now, we can actually implement the functionality for a stream of `Node` parses on any given tape for an inside (`I`) tree grammar:

```
instance TermStream (ts :| Node r x) s (is:.TreeIxR a I)
where
  termStream (ts:|Node f xs) (_both) (is:.TreeIxR frst i tfe)
  = map (\(s, ii, ee) ->
      let Ix l _ = getIndex s P
          P = Proxy :: Proxy (RunIx (is:.TreeIxR a I))
          l' = l+1
          ltf' = if null (children frst ! l) then E else F
      in  (s, (ii:.Ix l' ltf') (ee:.f xs l)))
  . termStream ts is
  . staticCheck (tfe == T)
```

We are given the current partial multi-tape terminal state `(s,ii,ee)` from the left symbol (`s`), partial tape index for dimensions 0 to the current dimension $k-1$ (`ii`), and parses from dimension 0 up to $k-1$ (`ee`).

Then we extract the node index l via `getIndex`. `getIndex` makes use of the inductive type-level structure of the index `(is:.TreeIxR a I)` to extract exactly the right index for the current tape. Since all computations inside `getIndex` are done on the type level, we have two benefits: (i) it is incredibly hard to confuse two different tapes because we never match explicitly on the structure `is` of the inductive index structure for the previous $k-1$ dimensions. (ii) the recursive traversal of the index structure `(is:.TreeIxR a I)` is a type-level operation. The runtime representation is just a set of unboxed parameters of the loop functions. This means that `getIndex` is a runtime "no-op" since there is no structure to traverse.

For a forest in pre-order, the first child of the current node l is just $l+1$. In case of a node without children, we set this to be empty (`E`), and otherwise we have a forest starting at $l+1$. We then extend the partial index `ii` to include the index for this tape, and analogously extend the partial parse `ee` with this parse.

Non-Terminals. Streams are generated differently for non-terminals. A non-terminal captures the parsing state for a complete (multi-dimensional) index, not for individual tapes. The `ITbl i x` data type captures memoizing behaviour for indices of type i, and memoized elements of type x. One may consider this data type to behave essentially like a memoizing function of type $i \rightarrow x$. We can capture this behaviour thus:

```
instance (Elem ls i) => Elem (ls, ITbl i x) i where
  data Elm (ls, ITbl i x) i = EIt x (RunIx i) (Elm ls i)
  type Arg (ls, ITbl i x)   = (Arg ls, r)
  getArg (EIt x _ ls)       = (getArg ls, r)
  getIdx (EIt _ i _ )       = i

instance MkStream (ls, ITbl (is:.i) x) (is:.i) where
  mkStream (ls, ITbl t f) ix
    = map ((s,tt,ii) -> ElmITbl (t!tt) ii s)
    . addIndexDense ix
    $ mkStream ls ix
```

Again, we require the use of an additional type class capturing `addIndexDense`. In this case, this makes it possible to provide n different ways on how to memoize (via `ITbl` for dense structures, `IRec` if no memoization is needed, etc) with m different index types using just $n+m$ instances instead of $n \times m$.

```
class AddIndexDense i where
  addIndexDense :: i -> <s Z> -> <s i>
```

Indexing into non-terminals can be quite involved however, and tree structures are no exception. A production rule $L \rightarrow D \circ R$ splits a forest into the tree-like element D that is explored further downwards from its local root, and the remaining right forest R. Both D and R can be in various states. These states are

```
data TFsize s = EpsFull TF s | FullEps s | OneRem s
              | OneEps s | Finis
```

In the implementation below, the optimizer makes use of constructor specialization (Peyton Jones 2007) to erase the `TFsize` constructors, while they allow us to provide certain guarantees of having captured all possible ways on how to partition a forest.

Excluding `Finis`, which denotes that no more parses are possible, we consider four major split operations:

`EpsFull` denotes an empty tree D meaning that the "left-most" element to be aligned to another subforest will actually be empty. This will later on induce a deletion on the current tape, if an empty D needs to be aligned to a non-empty structure.

`FullEps` will assign the complete remaining subforest to D, which will (obviously) not be a tree but a forest. While no tree (on another tape) can be aligned to a forest, the subforest below a tree on another tape can be aligned to this forest.

`OneRem` splits off the left-most tree to be bound to D, with the remaining siblings (the tail) being bound to R.

Finally, `OneEps` takes care of single trees masquerading as forests during a sequence of deletions on another tape.

All the logic of fixing the indices happens in the variable case for the D non-terminal. The static case for R just needs to extract the index and disallow further nont-terminals from accessing any subforest as all subforests are right-maximal.

```
instance AddIndexDense (is:.TreeIxR a I) where
  addIndexDense (vs:.Static) (is:.TreeIxR _ _ _)
  = map go . addIndexDense vs is where
    go (s,tt,ii) =
      let t = getIndex s P
          P = Proxy :: Proxy (RunIx (is:.TreeIxR a))
      in (s, tt:.t, ii:.Ix maxBound E)
-- continued below
```

In the variable case, we need to take care of all possible variations. The use of `flatten` to implement "nested loops" in Haskell follows in the same way as in (Höner zu Siederdissen 2012). `Done` and `Yield` are stream fusion step constructors (Coutts et al. 2007) that are explicit *only* in `flatten`.

```
-- continued
addIndexDense (vs:.Variable) (is:.TreeIxR f j tj)
= flatten mk step . addIndexDense vs is where
mk = return . EpsFull jj
-- forests
step (EpsFull E (s,t,i))
  = Yield (s, t:.TreeIxR f j E, i:.Ix j E) Finis
step (EpsFull F (s,t,i))
  = let Ix k _ = getIndex s P
        P = Proxy :: Proxy (RunIx (is:.TreeIxR a))
  in Yield (s, t:.TreeIxR f k E, i:.Ix k F) (FullEps (s,t,i))
step (FullEps (s,t,i))
  = let Ix k _ = getIndex s P
        P = Proxy :: Proxy (RunIx (is:.TreeIxR a))
        u = maxBound
  in Yield (s, t:.TreeIxR f k F, i:.Ix u E) (OneRem (s,t,i))
step (OneRem (s,t,i))
  = let Ix k _ = getIndex P
        l = rightSibling f k
        P = Proxy :: Proxy (RunIx (is:.TreeIxR a))
  in Yield (s, t:.TreeIxR f k T, i:.Ix l F) Finis
-- trees
step (EpsFull T (s,t,i))
  = let Ix k _ = getIndex P
        P = Proxy :: Proxy (RunIx (is:.TreeIxR a))
  in Yield (s,tt:.TreeIxR f k E,ii:.Ix k T) (OneEps (s,t,i))
step (OneEps (s,t,i))
  = let Ix k _ = getIndex P
        P = Proxy :: Proxy (RunIx (is:.TreeIxR a))
  in Yield (s,tt:.TreeIxR f k T,ii:.Ix k E) Finis

rightSibling :: Forest -> Int -> Int
rightSibling f k = rsib f ! k
```

This finally concludes the machinery necessary to extend `ADPfusion` to parse forest structures. The full implementation includes an extension to Outside grammars as well, in order to implement the algorithms as described in Sec. 8. We will not describe the full implementation for outside-style algorithms here as they require some effort. The source library [1] has extensive annotations in the comments that describe all cases that need to be considered.

The entire low-level implementation given above can remain transparent to a user who just wants to implement grammars on tree and forest structures since the implementation works seamlessly together with the embedded domain-specific languages `ADPfusion` (Höner zu Siederdissen 2012) and *its* abstraction that uses a quasiquotation (Mainland 2007) mechanism to hide most of this more intermediate-level library.

[1] `ADPfusionForest.tgz` (post-review: full url)

10. Related and further work

Here we discuss a small set of related ideas and how they intersect with our work. Due to the diversity of uses for trees, we necessarily have to be quite brief.

ICOREs. Giegerich and Touzet (2014) developed a system of inverse coupled rewrite systems (ICOREs) which unify dynamic programming on sequences and trees. Rules written in this system bear a certain semblance to our system. As of early 2016, we are not aware of an implementation, which makes comparisons beyond the formalism somewhat difficult. Given that with the present work we introduce not only a formal definition of parsing on tree-like structures but also *working machinery* with a thin layer on top that represents the domain-specific language, we point out the possibility of using this exact machinery to actually implement ICOREs.

Performance Improvements and Code Simplification. Library writers who want to extend this framework to new types of algorithms (say for unordered trees) have to deal with stream fusion primitives. Most of these are benign, but the `flatten` function, which provides the crucial possibility of fusion for `concatMap`-like behaviour and thus nested looping, breaks the stream fusion abstractions. One seemingly has to decide between readable code (`concatMap`) or performance (`flatten`). Farmer et al. (2014) proposed a mechanism to translate calls to `concatMap` into calls to `flatten`. The introduction of reliable rewrite rules into the GHC Haskell compiler would ameliorate this problem for basically all problems that involve nested loops and apply stream fusion.

Sparsification. The editing algorithm as proposed by Zhang and Shasha (1989) allows for a simple sparsification scheme, presented in the paper as operating only on certain *key roots*. Sparsification here prunes from the search space only illegal computations, and is not a heuristic. With a suitable index structure, the only change required for users of this framework would be an exchange of the imported module (for said index structure) while grammar and algebra remain the same.

These problems do, in general, also admit different decomposition strategies (Demaine et al. 2007) that positively influence the asymptotics, while still being optimal.

These approaches for running time and space improvements are only some of the possible sparsifying schemes. They prune *impossible* structures and do do not modify the candidate space since impossible structures lead to no parses. For string-based problems, a number of schemes exist which prune the actual candidate space and remove those candidates which are in some sense suboptimal. This approach tends to be successful for more complex problems, like the simultaneous alignment and folding of nucleotide sequences (Sankoff 1985) which has $O(n^3m^3)$ running time and $O(n^2m^2)$ space requirements. Good sparsification schemes (Will et al. 2015) can be very successful in reducing these asymptotics (here to a quadratic running time).

Generic approaches to sparsification would be very much appreciated to allow running more complex tree algorithms on large data.

Refinement Types and Dependent Types. Languages on strings are inherently "one"-dimensional. A string allows removing characters from the left or right end or splitting in two substrings. Grammars on trees on the other hand have depth, or a second dimension. This increased complexity is currently being countered by reliance on the strong type system in Haskell (cf. Sec. 9). With the move towards grammars on more complex data types (and index types) it will become necessary to provide even stronger guarantees on the type level.

Two recent developments will be helpful in this regard. *Liquid Haskell* (Vazou et al. 2014) provides refinement types which allow checking additional constraints on the indices as they are being manipulated. This should allow us to check conditions such as rules that loop during parsing, or create index out of bounds conditions.

The second development is the continued addition of features usually found in dependently typed programming languages (Weirich et al. 2013). Again, the goal is to check index manipulations during compilation. Given the inherent non-linearity of the underlying index structure, it should prove interesting to consider if there is way to encode the required index changes with a tree-type analogue of type-level integral numbers (which would be sufficient for string grammars).

Machine Translation. Parse trees of sentences in human languages are hugely important in machine translation. The combination of the Inside-Outside algorithm with statistical learning (for example the EM algorithm (Dempster et al. 1977)) provides the basis for efficient parameter estimation and many variants exist (Gildea 2003; Eisner 2003; Och et al. 2004). One should note that the Inside-Outside algorithm as seen in the context of expectation-maximization is used to calculate the required rule probabilities, and thereby part of EM. The work by (Höner zu Siederdissen et al. 2015b) takes a somewhat more generic view of the derivation of outside rules that happens to coincide with what is required for EM.

Computational Linguistics. Current machine translation systems internally use trees to translate sentences of different languages (Eisner 2003). Within the SMULTRON project (Volk et al. 2015), phrases of the book *Sophie's world* in English, German and Swedish were parsed and mutually aligned using tree alignments. Fig. 3 (center) shows one of the phrases for English and German. Using our tree alignment algorithm including the outside grammar, our system is able to align syntax trees of different languages.

One difficulty here is the calculation of word similarity in a meaningful way before tree alignment is performed. This is outside the score of the current paper and for reasons of simplicity we have labelled two leaves with the same symbol if the corresponding words have the same meaning, cf. Fig. 3 (left). The match probabilities for the alignment of the English and German sentence are shown on the right of Fig. 3. As the structure and composition of sentences differ in both languages, only certain parts align well. Nevertheless feasible alternative matchings have a much higher probability than the background, e.g. the three inner nodes labeled with VP in the English sentence each have significant probability to match with the one inner VP-labeled node in the German phrase.

Even though phrase productions differ for different languages, the probability mass assigned to the correct nodes is significant. This could be further improved by the introduction of dynamic programming algorithms over *unordered* trees. We simulate the effect by manually reordering the parse tree for the German sentence to more closely match the sentence in English. The result is given in the rightmost plot of Fig. 3. The two leaves labelled c can now be aligned together with the other highly probable matches, which is not possible given the ordered trees. The resulting maximum a-posteriori alignment carries an even higher total probability mass than the direct alignment made possible.

Multiple Context-Free Languages. The small example given above gives clear evidence that alignments (or edit maps for that matter) between ordered trees are not sufficient for all problems. One solution is the introduction of the class of unordered trees. The resulting search space is, however, much larger than for ordered trees. A more modest extension still provides for polynomial-time algorithms, but is more expressive. Multiple context-free languages (MCFGs) (Seki et al. 1991) allow crossings to occur in the derivation tree which is not possible for context-free grammars. An extension of `ADPfusion` for MCFGs has recently been developed (Riechert et al. 2016).

11. Conclusion

We provide sound theoretical foundations for generalized parsing beyond strings. This theoretical foundation forms a natural extension of abstract algebras on grammars. Together, this extended theory simplifies the design of advanced dynamic programming solutions. The theoretical foundations are accompanied by a practical and efficient implementation for parsing forest structures in the generalized `ADPfusion` framework. We provide implementations of two foundational algorithms, namely tree editing and tree alignment, that may serve as a guide on how to use and extend this library.

We point out that our extension of `ADPfusion` is assuming an "open world". It is possible to further extend both the underlying machinery to accommodate novel input data structures, and to and to design algorithms on top of this library. In addition, the whole system is completely embedded in Haskell so that the whole language is available for use at any time.

Acknowledgments

SJB thanks Nancy Retzlaff for an introductory lesson into computational linguistics and available data sets.

References

C. Arnold and C. L. Nunn. Phylogenetic targeting of research effort in evolutionary biology. *Amer. Nat.*, 176:601–612, 2010.

C. Arnold and P. F. Stadler. Polynomial algorithms for the maximal pairing problem: efficient phylogenetic targeting on arbitrary trees. *Alg. Mol. Biol.*, 5:25, 2010.

P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337:217–239, 2005.

D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 315–326. ACM, 2007.

E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Automata, languages and programming*, pages 146–157. Springer, 2007.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the royal statistical society. Series B (methodological)*, pages 1–38, 1977.

S. Dulucq and L. Tichit. RNA secondary structure comparison: exact analysis of the Zhang-Shasha tree edit algorithm. *Theoretical Computer Science*, 306(1):471–484, 2003.

J. Eisner. Learning non-isomorphic tree mappings for machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 2*, pages 205–208. Association for Computational Linguistics, 2003.

A. Farmer, C. Höner zu Siederdissen, and A. Gill. The HERMIT in the stream: fusing stream fusion's concatMap. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation*, pages 97–108. ACM, 2014. doi: 10.1145/2543728.2543736. URL `http://hackage.haskell.org/package/hermit`.

W. M. Fitch. Towards defining the course of evolution: minimum change for a specific tree topology. *Syst Zool*, 20:406–416, 1971.

B. Fluri, M. Würsch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

	german	english
a	Sie	They
b	hatten	had
c	unterhalten	discussing
d	Roboter	robots
e	über	-
x	-	been
y	sich	-

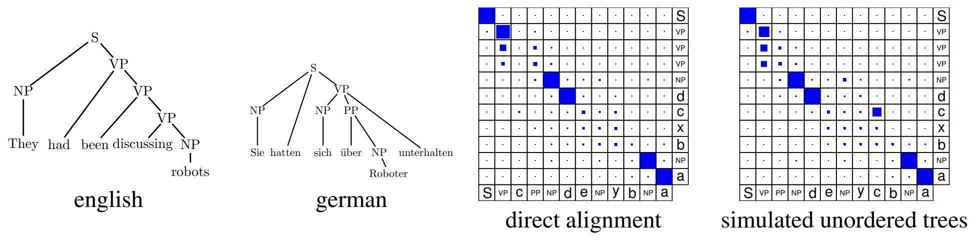

S
VP
NP
VP
VP
NP
They
had
been
discussing
robots
english
S
VP
NP
NP
PP
NP
Sie
hatten
sich
über
unterhalten
Roboter
german
S VP VP VP NP d c x b NP a
S VP c PP NP d e NP y b NP a
direct alignment
S VP VP VP NP d c x b NP a
S VP PP NP d e NP y c b NP a
simulated unordered trees

Figure 3: Parse trees for the sentence "They had been discussing robots" from *Sophie's world* in English and German. **Left:** Assignment of words to identifiers. Since the phrase productions differ, some words do not have an expression in the other language. **Center:** Parse trees for the English and German sentence based on the tree parse of the SMULTRON project. Inner nodes denote types of phrase productions. **Right:** alignment probabilities using tree alignment for the two parse trees. The direct alignment plot is the result of aligning the two ordered trees. If unordered trees are simulated via explicit movement of the bottom c node, the alignment quality is improved.

R. Giegerich and C. Meyer. Algebraic Dynamic Programming. In *Algebraic Methodology And Software Technology*, volume 2422, pages 243–257. Springer, 2002.

R. Giegerich and H. Touzet. Modeling Dynamic Programming Problems over Sequences and Trees with Inverse Coupled Rewrite Systems. *Algorithms*, pages 62–144, 2014.

R. Giegerich, C. Meyer, and P. Steffen. A Discipline of Dynamic Programming over Sequence Data. *Science of Computer Programming*, 51(3): 215–263, 2004.

D. Gildea. Loosely tree-based alignment for machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 80–87. Association for Computational Linguistics, 2003.

O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

J. A. Hartigan. Minimum mutation fits to a given tree. *Biometrics*, 29: 53–65, 1973.

M. Höchsmann. *The tree alignment model: algorithms, implementations and applications for the analysis of RNA secondary structures*. PhD thesis, Technische Fakultät, Universität Bielefeld, 2005.

C. Höner zu Siederdissen. Sneaking around concatMap: efficient combinators for dynamic programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming ICFP'12*, volume 47 (9) of *ACM SIGPLAN Notices*, pages 215–226, New York, NY, 2012. ACM. doi: 10.1145/2364527.2364559.

C. Höner zu Siederdissen, I. L. Hofacker, and P. F. Stadler. Product Grammars for Alignment and Folding. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 12(3):507–519, 2015a. ISSN 1545-5963. doi: 10.1109/TCBB.2014.2326155. URL `http://www.bioinf.uni-leipzig.de/Software/gADP/`.

C. Höner zu Siederdissen, S. J. Prohaska, and P. F. Stadler. Algebraic dynamic programming over general data structures. *BMC Bioinformatics*, 16, 2015b. doi: 10.1186/1471-2105-16-S19-S2.

T. Jiang, L. Wang, and K. Zhang. Alignment of treesan alternative to tree edit. *Theoretical Computer Science*, 143(1):137–148, 1995.

T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of computational biology*, 9(2):371–388, 2002.

G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 261–272. ACM, 2010.

W. P. Maddison. Testing character correlation using pairwise comparisons on a phylogeny. *J. Theor. Biol.*, 202:195–204, 2000.

G. Mainland. Why It's Nice to be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

F. J. Och, D. Gildea, S. Khudanpur, A. Sarkar, K. Yamada, A. M. Fraser, S. Kumar, L. Shen, D. Smith, K. Eng, et al. A smorgasbord of features for statistical machine translation. In *HLT-NAACL*, pages 161–168, 2004.

S. Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 327–337. ACM, 2007.

M. Riechert, C. Höner zu Siederdissen, and P. F. Stadler. Algebraic dynamic programming for multiple context-free languages. *submitted*, 2016.

D. Sankoff. Minimal mutation trees of sequences. *SIAM J. Appl Math.*, 28: 35–42, 1975.

D. Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics*, pages 810–825, 1985.

S. Schirmer. *Comparing forests*. PhD thesis, Bielefeld University, 2011.

S. Schirmer and R. Giegerich. Forest alignment with affine gaps and anchors. In *Combinatorial Pattern Matching*, pages 104–117. Springer, 2011.

H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context free grammars. *Theor. Comp. Sci.*, 88:191–229, 1991.

K. Tai. The tree-to-tree correction problem. *J. ACM*, 26:422–433, 1979.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In M. W. Bailey, R. Balasubramonian, A. Davis, and S. Adve, editors, *ICFP'14*, volume 49 (9) of *ACM SIGPLAN Notices*, pages 269–282, New York, NY, 2014. ACM.

M. Volk, A. Ghring, A. Rios, T. Marek, and Y. Samuelsson. SMULTRON (version 4.0) The Stockholm MULtilingual parallel TReebank, 2015.

S. Weirich, J. Hsu, and R. A. Eisenberg. Towards dependently typed haskell: System fc with kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP*, volume 13. Citeseer, 2013.

S. Will, C. Otto, M. Miladi, M. Möhl, and R. Backofen. Sparse: Quadratic time simultaneous alignment and folding of rnas without sequence-based heuristics. *Bioinformatics*, 31(15):2489–2496, 2015.

H. Xiao, M. Zhang, A. Mosig, and H. W. Leong. Dynamic programming algorithms for efficiently computing cosegmentations between biological images. In T. M. Przytycka and M.-F. Sagot, editors, *Algorithms in Bioinformatics WABI'11*, volume 6833 of *Lect. Notes Comp. Sci.*, pages 339–350, New York, 2011. Springer.

K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J Computing*, 18:1245–1262, 1989.

B. Zieliński and M. Iwanowski. Binary image comparison with use of tree-based approach. In R. S. Choraś, editor, *Image Processing and Communications Challenges 4*, volume 184 of *Adv. Intelligent Systems Comput.*, pages 171–177, New York, 2013. Springer.