Towards a Discipline of Dynamic Programming

Robert Giegerich, Carsten Meyer, Peter Steffen

Faculty of Technology, Bielefeld University
Postfach 10 01 31
33501 Bielefeld, Germany
{robert,cmeyer,psteffen}@techfak.uni-bielefeld.de

Abstract. Dynamic programming is a classic programming technique, applicable in a wide variety of domains, like stochastic systems analysis, operations research, combinatorics of discrete structures, flow problems, parsing ambiguous languages, or biosequence analysis. Yet, heretofore no methodology was available guiding the design of such algorithms. The matrix recurrences that typically describe a dynamic programming algorithm are difficult to construct, error-prone to implement, and almost impossible to debug.
This article introduces an algebraic style of dynamic programming over sequence data. We define its formal framework including a formalization of Bellman's principle. We suggest a language for algorithm design on a convenient level of abstraction. We outline three ways of implementation, including an embedding in a lazy functional language. The workings of the new method are illustrated by a series of examples from diverse areas of computer science.

1 The power and scope of dynamic programming

1.1 Dynamic programming: a world without rules?

Dynamic programming (DP) is one of the classic programming paradigms, introduced even before the term Computer Science was firmly established. When applicable, DP often allows to solve combinatorial optimization problems over a search space of exponential size in polynomial space and time. Bellman's "Principle of Optimality" [Bel57] belongs to the core knowledge we expect from every computer science graduate. Significant work has gone into formally characterizing this principle [Mor82,Mit64], formulating DP in different programming paradigms [Moo99,Cur97] and studying its relation to other general programming methods such as greedy algorithms [BM93].

The scope of DP is enormous. Much of the early work was done in the area of physical state transition systems and operations research [BD62]. Other, simpler examples (more suited for computer science textbooks) are optimal matrix chain multiplication, polygon triangulation, or string comparison. The analysis of molecular sequence data has fostered increased interest in DP. Protein homology search, RNA structure prediction, gene finding, and discovery of regulatory signals in RNA pose string processing problems in unprecedenced variety

and data volume. A recent textbook in biosequence analysis [DEKM98] lists 11 applications of DP in its introductory chapter, and many more in the sequel.

Developing a DP algorithm for an optimization problem over a nontrivial domain has intrinsic difficulties. The choice of objective function and search space are interdependent, and closely tied up with questions of efficiency. Once completed, all DP algorithms are expressed via recurrence relations between tables holding intermediate results. These recurrences provide a very low level of abstraction, and subscript errors are a major nuisance even in published articles. The recurrences are difficult to explain, painful to implement, and almost impossible to debug: A subtle error gives rise to a suboptimal solution every now and then, which can hardly be detected by human inspection. In this situation it is remarkable that neither the literature cited above, nor computer science textbooks [CLR90,Gus97,Meh84,BB88,AHU83,Sed89] provide guidance in the development of DP algorithms.

1.2 The promises of Algebraic Dynamic Programming

Algebraic dynamic programming (ADP) is a new style of dynamic programming and a method for algorithm development, designed to alleviate this situation. It allows to design, reason about, tune and even test DP algorithms on a more abstract level. This is achieved by a restructuring of concerns: Any DP algorithm evaluates a search space of candidate solutions under a scoring scheme and an objective function. The classic DP recurrences reflect the three aspects of search space construction, scoring and choice, and efficiency in an indiscriminable fashion. In the new algebraic approach, these concerns are separated.

The search space is described by a *yield grammar*, which is a tree grammar generating a string language. The ADP developer takes the view that for a given input sequence, "first" the search space is constructed, leading to an enumeration of all candidate solutions. This is a parsing problem, solved by a standard device called a *tabulating yield parser*. The developer can concentrate on the design of the grammar.

Evaluation and choice are captured by an *evaluation algebra*. It is important (and in contrast to traditional presentations of DP algorithms) that this algebra comprises *all* aspects relevant to the intended objective of optimization, but is independent of the description of the search space. The ADP developer takes the view that a "second" phase evaluates the candidates enumerated by the first phase, and makes choices according to some optimality criterion.

Of course, the interleaving of search space construction and evaluation is essential to prevent combinatorial explosion. It is contributed by the ADP method in a way transparent to the developer. By the separate description of search space and evaluation, ADP also produces modular and therefore re-usable algorithm components. More complex problems can be approached with better chance of success, and there is no loss of efficiency compared to ad-hoc approaches. The relief from formulating explicit recurrences brings about a boost in programming productivity, captured by practitioners in the slogan "No subscripts, no errors!".

The ADP approach has emerged recently in the context of biosequence analysis, but it pertains to dynamic programming over sequential data in general. "Sequential data" does not mean we only study string problems – a chain of matrices to be multiplied, for example, is sequential input data in our sense. An informal introduction, written towards the needs of the bioinformatics community, has appeared in [Gie00]. The present article gives a complete account of the foundations of the ADP method, and, almost in the style of a tutorial, shows its application to several classic combinatorial optimization problems in computer science.

Like any methodical work, this article suffers from the dilemma that for the sake of exposition, the problems solved have to be rather simple, such that the impression may arise that methodical guidance is not really required. The ADP method has been applied to several nontrivial problems in the field of biosequence analysis. An early application is a program for aligning recombinant DNA [GKW99], when the ADP theory was just about to emerge. Two recent applications are searching for sequence/structure motifs in DNA or RNA [MG02], and the problem of folding saturated RNA secondary structures, posed by Zuker and Sankoff in [ZS84] and solved in [EG01].

1.3 Overview of this article

In Section 2, we shall review some new and some well known applications of dynamic programming over sequence data, in the form in which they are traditionally presented. This provides a common basis for the subsequent discussion. By the choice of examples, we illustrate the scope of dynamic programming to a certain extent. In particular, we show that (single) sequence analysis and (pairwise) sequence comparison are essentially the same kind of problem when viewed on a more abstract level. The applications studied here will later be reformulated in the style and notation of ADP.

In Section 3 we introduce the formal basis of the algebraic dynamic programming method: Yield grammars and evaluation algebras. We shall argue that these two concepts precisely catch the essence of dynamic programming, at least when applied to sequence data. Furthermore, we introduce a special notation for expressing ADP algorithms. Using this notation an algorithm is completely described on a very abstract level, and can be designed and analysed irrespective of how it is eventually implemented. We discuss efficiency analysis and point to other work concerning techniques to improve efficiency.

In Section 4 we formulate yield grammars and evaluation algebras for the applications described in Section 2. Moreover we show how problem variations can be expressed transparently using the ADP approach.

Section 5 indicates three ways of actually implementing an algorithm once it is written in ADP notation: The first alternative is direct embedding and execution in a functional programming language, the second is manual translation to the abstraction level of an imperative programming language. The third alternative, still under development, is the use of a system which directly compiles ADP notation into C code.

In the conclusion, we discuss the merits of the method presented here, evaluate its scope, and glance on possible extensions of the approach.

2 Dynamic programming, traditional style

In this section we discuss four introductory examples of dynamic programming, solved by recurrences in the traditional style. They will be reformulated in algebraic style in later sections. We begin our series of examples with an algorithmic fable.

2.1 Calif El Mamun's caravan

Once upon a time around the year 800, Calif El Mamun of Bagdad, a son of Harun al Raschid, decided to take his two sons on their first hadj to Mecca. He called for the camel dealer, in order to compose a little caravan for the three pilgrims. Each one needed one riding camel, plus three further ones for baggage and water supplies. The price for the former was five tubes of oil, whereas a baggage camel was to be somewhat cheaper. After a long afternoon of bargaining, the following bill was written in the sand of the court yard:

$$\text{bill}_1 = \qquad (1+2)*(3*4+5) \tag{1}$$

The dealer explained that the $(1+2)$ stood for the calif and his two sons, and there were three baggage camels (4 tubes) and one riding camel (5 tubes) for each pilgrim. Unfortunately, before the total price could be calculated (which took a little longer than today in those days), everyone had to turn away for the evening prayers.

When they returned to business, they found that the wind had erased the parentheses, leaving only numbers and operators:

$$\text{bill}_2 = \qquad 1+2*3*4+5 \tag{2}$$

Such selective erasing seemed mysterious. The Imam was called in. He explained that Allah, while agreeing with the basic constituents of the formula, had been displeased by the placement of the parentheses, which should therefore be re-considered. The camel dealer helpfully jumped up to redraw the formula:

$$\text{bill}_3 = \qquad (1+2)*3*(4+5) \tag{3}$$

The dealer argued that now the parentheses showed beautiful symmetry, pleasant to the eye of any higher being. El Mamun was reluctant to agree. Although not good at numbers, he could not help to suspect that the tour to Mecca had become much more expensive. He called for the scientist.

Al Chwarizmi, a famous mathematician already, was a wise as well as a cautious man. After giving the problem serious consideration, he spoke:

"Honorable Calif, in his infinite wisdom Allah has provided 14 ways to evaluate this formula, with many different outcomes. For example,

$$(1+2)*(3*(4+5)) = 81, \text{while}$$
$$(1+(2*(3*4))+5) = 30.$$

An intermediate outcome would be

$$(1+2)*((3*4)+5) = 51,$$

just another example. As all readings are equal in Allah's eyes, the fate of a good muslim will not depend on the reading adopted, but on the beneficial effects achieved by the choice."

Diplomat he was, with this answer Al Chwarizmi hoped to have passed responsibility back to the Imam. However, Calif El Mamun could think of beneficial effects himself. He contemplated these words of wisdom over night, and in the morning he ruled as follows :

1. The fraudulent camel dealer was to be buried in the sand, next to the formula $(1+2)*3*(4+5)$.
2. The Calif would take possession of all the camels in the dealers stock.
3. Al Chwarizmi was granted 51 tubes of oil for a long-term research project. The task was to find a general method to redraw the parentheses in a formula such that the outcome was either minimized or maximized – depending on whether the Calif was on the buying or on the selling side.
4. Until this research was complete, in any case where parentheses were lacking, multiplication should take priority over addition.

Today, we can still observe the outcome of this episode: El Mamun became a very, very rich man, and his name gave rise to the word "mammon", present in many modern languages and meaning an awful lot of money. Al Chwarizmi did not really solve the problem in full generality, since DP was only developed later in the 1950s by Richard Bellman. But by working on it, he made many important discoveries in elementary arithmetic, and thus he became the father of algorithmics. As a consequence of the problem remaining unsolved, until today multiplication takes priority over addition. This ends our fable.

We now provide a DP solution for El Mamun's problem. Clearly, explicit parentheses add some internal structure to a sequence of numbers and operators. They tell us how subexpressions are grouped together – which are sums, and which are products. Let us number the positions in the text t representing the formula:

$$t = {}_0\,1\,{}_1\;+\;{}_2\,2\,{}_3\;*\;{}_4\,3\,{}_5\;*\;{}_6\,4\,{}_7\;+\;{}_8\,5\,{}_9 \tag{4}$$

such that we can refer to substrings by index pairs: $t(0,9)$ is the complete string t, and $t(2,5)$ is $2*3$. A substring $t(i,j)$ that forms an expression can, in general, be evaluated in many different ways, and we shall record the best value for $t(i,j)$

in a table entry $T(i,j)$. Since addition and multiplication are strictly monotone functions on positive numbers, an overall value $(x+y)$ or $(x*y)$ can only be maximal if both subexpressions x and y are evaluated to their maximal values. So it in fact suffices to record the maximum in each entry. This is our first use of Bellman's Principle, to be formalized later.

More precisely, we define

$$T(i,i+1) = n, \text{if } t(i,i+1) = n \tag{5}$$

$$T(i,j) \quad = \max\{T(i,k) \otimes T(k+1,j) | i < k < j, t(k,k+1) = \otimes\} \tag{6}$$

where $\otimes$ is either $+$ or $*$. Beginning with the shortest subwords of t, we can compute successively all defined table entries.

In $T(0,9)$ we obtain the maximal possible value overall. If, together with $T(i,j)$, we also record the position k that leads to the optimal value, then we can reconstruct the reading of the formula that yields the optimal value. It is clear that El Mamun's minimization problem is solved by simply replacing max by min. Figure 1 gives the results for maximization and minimization of El Mamun's bill.

	0	1	2	3	4	5	6	7	8	9
0	/	(1,1)		(3,3)		(7,9)		(25,36)		(30,81)
1	/	/								
2	/	/	/	(2,2)		(6,6)		(24,24)		(29,54)
3	/	/	/	/						
4	/	/	/	/	/	(3,3)		(12,12)		(17,27)
5	/	/	/	/	/	/				
6	/	/	/	/	/	/	/	(4,4)		(9,9)
7	/	/	/	/	/	/	/	/		
8	/	/	/	/	/	/	/	/	/	(5,5)
9	/	/	/	/	/	/	/	/	/	/

Fig. 1. Results for the maximization and minimization of El Mamun's bill denoted as tuple (x,y) where x is the minimal value and y the maximal value.

Note that we have left open a few technical points: We have not provided explicit code to compute the table T, which is actually triangular, since i is always smaller than j. Such code has to deal with the fact that an entry remains undefined when $t(i,j)$ is not a syntactically valid expression, like $t(1,4) =$ "+ 2 *". In fact, there are about as many undefined entries as there are defined ones, and we may call this a case of sparse dynamic programming and search for a more clever form of tabulation. Another open point is the possibility of malformed input, like the non-expression "1 + * 2". The implementation shown later will take care of all these cases.

Exercise 1 Al Chwarizmi remarked that there were 14 different ways to evaluate the bill. Develop a dynamic programming algorithm to compute the number of alternative interpretations for a parenthesis-free formula.□

The solution to Exercise 1 closely follows the recurrences just developed, except that there is no maximization or minimization involved. This one is a combinatorial counting problem. Although DP is commonly associated with optimization problems, we see that its scope is actually wider.

2.2 Matrix chain multiplication

A classic dynamic programming example is the matrix chain multiplication problem [CLR90]. Given a chain of matrices $A_1, ..., A_n$, find an optimal placement of parentheses for computing the product $A_1 * ... * A_n$. The placement of parentheses can dramatically reduce the number of scalar multiplications needed. Consider three matrices A_1, A_2, A_3 with dimensions $10 \times 100, 100 \times 5$ and 5×50. Multiplication of $(A_1 * A_2) * A_3$ needs $10*100*5+10*5*50 = 7500$ scalar multiplications, in contrast to $10 * 100 * 50 + 100 * 5 * 50 = 75000$ when choosing $A_1 * (A_2 * A_3)$.

Let M be a $n \times n$ table. Table entry $M(i,j)$ shall hold the minimal number of multiplications needed to calculate $A_i * ... * A_j$. Compared to the previous example, the construction of the search space is a lot easier here since it does not depend on the structure of the input sequence but only on its length. $M(i,j) = 0$ for $i = j$. In any other case there exist $j-i$ possible splittings of the matrix chain $A_i, ..., A_j$ into two parts $(A_i, ..., A_k)$ and $(A_{k+1}, ..., A_j)$. Let (r_i, c_i) be the dimension of matrix A_i. Multiplying the two partial product matrices requires $r_i c_k c_j$ operations. Again we observe Bellman's principle. Only if the partial products have been arranged internally in an optimal fashion, this product can minimize scalar multiplications overall. We order table calculation by increasing subchain length, such that we can look up all the $M(i,k)$ and $M(k+1,j)$ when needed for computing $M(i,j)$. This leads to the following matrix recurrence:

$$\begin{aligned} &\text{for } j = 1 \text{ to } n \text{ do} \\ &\quad \text{for } i = j \text{ to } 1 \text{ do} \\ &\qquad M(i,j) = \begin{cases} 0 & \text{for } i = j \\ \min\{M(i,k) + M(k+1,j) + r_i c_k c_j \mid i \leq k < j\} & \text{for } i < j \end{cases} \end{aligned} \tag{7}$$

Minimization over all possible splittings gives the optimal value for $M(i,j)$.

This example demonstrates that dynamic programming over sequence data is not necessarily limited to (character) strings, but can also be used with sequences of other types, in this case pairs of numeric values.

Exercise 2 Design an algorithm to minimize the size (a) of the largest intermediate matrix that needs to be computed, and (b) of the overall storage needed at any point for intermediate matrices during the computation of the chain product.□

The first part of Exercise 2 closely follows the recurrences developed above, while the latter includes optimizing the evaluation order, and is more complicated.

2.3 Global and local similarity of strings

We continue our series of examples by looking at the comparison of strings. The measurement of similarity or distance of two strings is an important operation applied in several fields, for example spelling correction, textual database retrieval, speech recognition, coding theory or molecular biology.

A common formalization is the string edit model. We measure the similarity of two strings by scoring the different sequences of character deletions (denoted by D), character insertions (denoted by I) and character replacements (denoted by R) that transform one string into the other. If a character is unchanged, we formally model this as a replacement by itself. Thus, an edit operation is applied at each position.

Figure 2 shows some possibilities to transform the string `MISSISSIPPI` into the string `SASSAFRAS` visualized as alignment.

```
MISSI--SSIPPI        MISSISSIPPI-        MISSI---SSIPPI
SASSAFRAS----        ---SASSAFRAS        SASSAFRAS-----

RR  RIIR DDDD        DDD R  RRRRI        RR  RIII DDDDD
```

Fig. 2. Three out of many possible ways to transform the string `MISSISSIPPI` into the string `SASSAFRAS`. Only deletions, insertions, and proper replacements are marked.

A similarity scoring function δ associates a similarity score of 0 with two empty strings, a positive score with two characters that are considered similar, a negative score with two characters that are considered dissimilar. Insertions and deletions also receive negative scores. For strings x of length m and y of length n, we compute the similarity matrix $E_{m,n}$ such that $E(i,j)$ holds the similarity score for the prefixes $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$. $E(m,n)$ holds the overall similarity value of x and y.

E is calculated by the following recurrences:

$$E(0,0) = 0 \tag{8}$$

$$\text{for } i = 0 \text{ to } m-1 \text{ do} \quad E(i+1,0) = E(i,0) + \delta(D(x_{i+1})) \tag{9}$$

$$\text{for } j = 0 \text{ to } n-1 \text{ do} \quad E(0,j+1) = E(0,j) + \delta(I(y_{j+1})) \tag{10}$$

$$\begin{aligned} &\text{for } i = 0 \text{ to } m-1 \text{ do} \\ &\quad \text{for } j = 0 \text{ to } n-1 \text{ do} \\ &\qquad E(i+1,j+1) = max \left\{ \begin{array}{ll} E(i,j+1) & + \delta(D(x_{i+1})) \\ E(i+1,j) & + \delta(I(y_{j+1})) \\ E(i,j) & + \delta(R(x_{i+1}, y_{j+1})) \end{array} \right\} \end{aligned} \tag{11}$$

$$\text{return } E(m,n) \tag{12}$$

The space and time efficiency of these recurrences is $O(mn)$.

Often, rather than computing the (global) similarity of two strings, it is required to search for local similarities within two strings. In molecular sequence analysis, we study DNA sequences, given as strings from four types of nucleotides, or Protein sequences, given as strings from twenty types of amino acids. In DNA, we often have long non-coding regions and small coding regions. If two coding regions are similar, this does not imply that the sequences have a large global similarity. If we investigate a protein with unknown function, we are interested in finding a 'similar' protein with known biological function. In this situation the functionally important sequence parts must be similar while the rest is arbitrary.

Local similarity asks for the best match of a subword of x with a subword of y. The Smith and Waterman algorithm [SW81] needs $O(mn)$ time and space to solve this problem. We compute a matrix $C_{m,n}$ where the entry (i,j) contains the best score for all pairs of suffixes of $x_1 \dots x_i$ and $y_1 \dots y_j$.

$$C(i,j) = max\{score(x',y')|x' \text{ suffix of } x_1 \dots x_i \text{ and } y' \text{ suffix of } y_1 \dots y_j\} \quad (13)$$

Since we look for a local property, it must be possible to match arbitrary subwords of x and y without scoring their dissimilar prefixes and suffixes. In order to reject prefixes with negative scores, all we need to change in comparison to the recurrences for global similarity (see Equations $8-12$) is to fix the first line and column to Zero-values and to add a Zero-value case in the calculation of the entry $C(i+1,j+1)$. This Zero amounts to an empty prefix joining the competition for the best match at each point (i,j). This leads to the following recurrences:

$$\text{If } i = 0 \text{ or } j = 0, \text{ then } C(i,j) = 0. \quad (14)$$

Otherwise,

$$C(i+1,j+1) = max \left\{ \begin{array}{ll} 0 & \\ C(i,j+1) & + \delta(D(x_{i+1})) \\ C(i+1,j) & + \delta(I(y_{j+1})) \\ C(i,j) & + \delta(R(x_{i+1}, y_{j+1})) \end{array} \right\} \quad (15)$$

$$\text{return } \max_{i,j} C(i,j) \quad (16)$$

Equation 16 performs another traversal of table C to obtain the highest score overall.

Exercise 3 Add appropriate for-loops to Equations 14 - 16.□

Exercise 4 Remove maximization in Equation 16. Use instead another table $D_{m,n}$, such that $D(i,j) = \max_{l \leq i, k \leq j} C(i,j)$. Can we compute D within the for-loops controlling Equation 15? □

2.4 Fibonacci numbers and the case of recursion versus tabulation

In this last introductory example, we make our first deviation from the traditional view of dynamic programming. There are many simple cases where the principles of DP are applied without explicit notice. Fibonacci numbers are a famous case in point. They are defined by

$$F(1) = 1 \tag{17}$$
$$F(2) = 1 \tag{18}$$
$$F(i+2) = F(i+1) + F(i) \tag{19}$$

Fibonacci numbers may seem atypical as a DP problem, because there is no optimization criterion. We have already seen (cf. Exercise 1) that optimization is an important, but not an essential part of the dynamic programming paradigm.

Every student of computer science knows that computing F as a recursive function is very inefficient – it takes exactly $2F(n) - 1$ calls to F to compute $F(n)$. Although we really need only the n values $F(1)$ through $F(n-1)$ when computing $F(n)$, each value $F(n-k)$ is calculated not once, but $F(k+1)$ times. The textbook remedy to this inefficiency is strengthening the recursion – define

$$F(n) = Fib(0,1,1), \text{where} \tag{20}$$
$$Fib(a,b,i) = \text{ if } (i = n) \text{ then } b \text{ else } Fib(b, a+b, i+1) \tag{21}$$

Here we shall consider another solution. This one requires no redefinition of F at all, just a change of data type: Consider F as an integer array, whose elements are defined via Equations 17 – 19. In a data driven programming language, its elements will be calculated once when first needed. In an imperative language, since F is data now rather than a function, we need to add explicit control structure – an upper bound for n and a for-loop to actually calculate the array elements in appropriate order.

The lesson here is the observation that a table (matrix, array) over some index domain, defined by recurrences, is mathematically equivalent to a recursive function defined by the very same recurrences. This gives us a first guiding rule for the systematic development of DP algorithms: Think of a DP algorithm as a family of recursive functions over some index domain. Don't worry about tabulation and evaluation order, this can always be added when the design has stabilized.

Exercise 5 Show how this discussion applies to the local similarity problem. Implement table C as a recursive function, and use another function D for the last equation. Does function D require tabulation to be efficient?□

3 Foundations of the algebraic approach to dynamic programming

3.1 Basic terminology

Alphabets. An *alphabet* $\mathcal{A}$ is a finite set of symbols. Sequences of symbols are called strings. ε denotes the empty string, $\mathcal{A}^1 = \mathcal{A}$, $\mathcal{A}^{n+1} = \{ax | a \in \mathcal{A}, x \in \mathcal{A}^n\}$, $\mathcal{A}^+ = \bigcup_{n \geq 1} \mathcal{A}^n$, $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$.

Signatures and algebras. A (single-sorted) signature Σ over some alphabet $\mathcal{A}$ consists of a sort symbol S together with a family of operators. Each operator o has a fixed arity $o : s_1 ... s_{k_o} \rightarrow S$, where each s_i is either S or $\mathcal{A}$. A Σ-algebra $\mathcal{I}$ over $\mathcal{A}$, also called an interpretation, is a set $\mathcal{S}_\mathcal{I}$ of values together with a function o_I for each operator o. Each o_I has type $o_I : (s_1)_I ... (s_{k_o})_I \rightarrow S_I$ where $\mathcal{A}_I = \mathcal{A}$.

A *term algebra* T_Σ arises by interpreting the operators 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, $T_\Sigma(V)$, with $V \subset T_\Sigma(V)$. By convention, operator names are capitalized in the term algebra.

Tree grammars. Terms will be viewed as rooted, ordered, node-labeled trees in the obvious way. All inner nodes carry (non-nullary) operators from Σ, while leaf nodes carry nullary operators or symbols from $\mathcal{A}$. A term/tree with variables is called a *tree pattern.* A tree containing a designated occurrence of a subtree t is denoted $C[...t...]$.

A tree language over Σ is a subset of T_Σ. Tree languages are described by tree grammars, which can be defined in analogy to the Chomsky hierarchy of string grammars. Here we use regular tree grammars originally studied in [Bra69]. In [GS88] they were redefined to specify term languages over some signature. Our further specialization so far lies solely in the distinguished role of $\mathcal{A}$.

Definition 1 (Tree grammar over Σ and $\mathcal{A}$.)
A (regular) tree grammar $\mathcal{G}$ over Σ and $\mathcal{A}$ is given by

- a set V of nonterminal symbols
- a designated nonterminal symbol Ax called the axiom
- a set P of productions of the form $v \rightarrow t$, where $v \in V$ and $t \in T_\Sigma(V)$

The derivation relation for tree grammars is $\rightarrow^*$, with $C[...v...] \rightarrow C[...t...]$ if $v \rightarrow t \in P$. The language of $v \in V$ is $\mathcal{L}(v) = \{t \in T_\Sigma | v \rightarrow^* t\}$, the language of $\mathcal{G}$ is $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Ax)$.□

For convenience, we add a lexical level to the grammar concept, allowing strings from $\mathcal{A}^*$ in place of single symbols. By convention, `achar` denotes an arbitrary character, `char c` a specific character `c`, `string` an arbitrary string and `empty` the empty input. Also for brevity, we allow syntactic conditions associated with righthand sides.

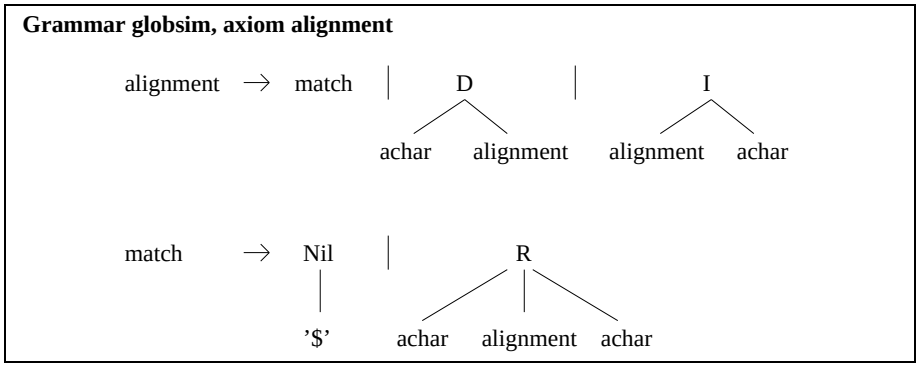

Fig. 3. The tree grammar `globsim` for global similarity (see Section 2.3)

The yield of a tree is normally defined as its sequence of leaf symbols. Here we are only interested in the symbols from $\mathcal{A}^*$; nullary constructors by definition have yield ε. Hence the yield function y on T_Σ is defined by $y(t) = w$, where $w \in \mathcal{A}^*$ is the string of leaf symbols in left to right order.

3.2 Conceptual separation of recognition and evaluation

Any dynamic programming algorithm implicitly constructs a search space from its input. The elements of this search space have been given different names: *policy* in [Bel57], *solution* in [Mor82], *subject under evaluation* in [Gie00]. Since the former two terms have been used ambiguously, and the latter is rather technical, we shall use the term *candidate* for elements of the search space. Each candidate will be evaluated, yielding a *final state*, a *cost*, or a *score*, depending whether you follow [Bel57], [Mor82] or [DEKM98]. We shall use the term *answer* for the result of evaluating a candidate.

Typically, there is an ordering defined on the answer data type. The DP algorithm returns a maximal or minimal answer, and if so desired, also one or all the candidates that evaluate(s) to this answer. Often, the optimal answer is determined first, and a candidate that led to it is reconstructed by backtracking. The candidates themselves do not have an explicit representation during the DP computation. Our goal to separate recognition and evaluation requires an explicit representation of candidates. Figure 4 shows a global similarity candidate.

Imagine that during computing an answer, we did not actually call those functions that perform the evaluation. Instead, we would apply them symbolically, building up a formula that – once evaluated – would yield this answer value. Clearly, this formula itself is a perfect choice of candidate representation:

- The formula represents everything we need to know about the candidate to eventually evaluate it.
- The complete ensemble of such formulas, considered for a specific problem instance, is a precise description of the search space.

```
candidate1 = D 'd' (R 'a' (I (R 'r' (R 'l' (R 'i' (R 'n' (R 'g' (Nil
'$') 'e') 'n') 'i') 'l') 'r') 'i') 'a')
```

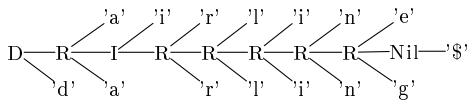

Fig. 4. The term representation of a global similarity candidate `candidate1` for `darling` and `airline`, and the tree representation of this term.

– The idea works for any DP algorithm. After all, when we compute an answer value, we can as well compute it symbolically and record the formula.

To design a DP algorithm, we hence need to specify the language of formulas, the search space spawned by a particular input, and their eventual evaluation.

3.3 Evaluation algebras

Definition 2 (Evaluation algebra.) Let Σ be a signature with sort symbol Ans. A Σ-evaluation algebra is a Σ-algebra augmented with an objective function $h : [Ans] \to [Ans]$, where $[Ans]$ denotes lists over Ans. □

In most DP applications, the purpose of the objective function is minimizing or maximizing over all answers. We take a slightly more general view here. The objective may be to calculate a sample of answers, or all answers within a certain threshold of optimality. It could even be a complete enumeration of answers. We may compute the size of the search space or evaluate it in some statistical fashion, say by averaging over all answers. This is why in general, the objective function will return a list of answers. If maximization was the objective, this list would hold the maximal answer as its only element.

We formulate a signature Π for the global similarity example:

$$\begin{array}{lllll}
Nil: & Ans & & & \to Ans \\
D: & \mathcal{A} & \times Ans & & \to Ans \\
I: & Ans & \times \mathcal{A} & & \to Ans \\
R: & \mathcal{A} & \times Ans & \times \mathcal{A} & \to Ans \\
h: & [Ans] & & & \to [Ans]
\end{array}$$

We formulate two evaluation algebras for signature Π. The algebra `unit` (Figure 5 right) scores each matching character by $+1$, and each character mismatch, deletion or insertion by -1. The algebra `wgap` (Figure 5 left) is a minor generalization of `unit`. It uses two parameter functions `w` and `gap`, that may score (mis)matches and deletions or insertions depending on the concrete characters involved. For example, a phoneticist would choose `w('v','b')` as a (positive) similarity rather than a (negative) mismatch.

```
Ans_wgap  = ℕ                    Ans_unit  = ℕ
wgap = (nil,d,i,r,h)             unit = (nil,d,i,r,h)
 where                            where
nil(a)   = 0                     nil(a)   = 0
d(x,s)   = s + gap(x)            d(x,s)   = s - 1
i(s,y)   = s + gap(y)            i(s,y)   = s - 1
r(a,s,b) = s + w(a,b)            r(a,s,b) = if a==b then s + 1 else s - 1
h([])    = []                    h([])    = []
h (l)    = [maximum(l)]          h (l)    = [maximum(l)]
```

Fig. 5. Algebras `wgap` (left) and `unit` (right)

For term `candidate1` of Figure 4, we obtain $\texttt{candidate1}_{unit}$ `= 2` and $\texttt{candidate1}_{wgap}$ `= gap('d') + w('a','a') + gap('i') + w('r','r') + w('l','l') + w('i','i') + w('n','n') + w('e','g') + 0`.

3.4 Yield grammars

We obtain an explicit and transparent definition of the search space of a given DP problem by a change of view on tree grammars and parsing:

Definition 3 (Yield grammars and yield languages.) Let $\mathcal{G}$ be a tree grammar over Σ and $\mathcal{A}$, and y the yield function. The pair $(\mathcal{G}, y)$ is called a yield grammar. It defines the yield language $\mathcal{L}(\mathcal{G}, y) = y(\mathcal{L}(\mathcal{G}))$. □

Definition 4 (Yield parsing.) Given a yield grammar $(\mathcal{G}, y)$ over $\mathcal{A}$ and $w \in \mathcal{A}^*$, the yield parsing problem is: Find $P_{\mathcal{G}}(w) := \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$.□

The search space spawned by input w is $P_{\mathcal{G}}(w)$. For the similarity example we consider the string $x\$y^{-1}$ as input, where \$ is a separator symbol not occurring elsewhere. In Section 5.1 the relation between single sequence analysis and pairwise sequence comparison is discussed. Yield parsing is the computational engine underlying ADP.

3.5 Algebraic dynamic programming and Bellman's principle

Given that yield parsing traverses the search space, all that is left to do is evaluate candidates in some algebra and apply the objective function.

Definition 5 (Algebraic dynamic programming.)

- An ADP problem is specified by a signature Σ over $\mathcal{A}$, a yield grammar $(\mathcal{G}, y)$ over Σ, and a Σ-evaluation algebra I with objective function h_I.

- An ADP problem instance is posed by a string $w \in \mathcal{A}^*$. The search space it spawns is the set of all its parses, $P_{\mathcal{G}}(w)$.
- Solving an ADP problem is computing

$$h_{\mathcal{I}}\{t_{\mathcal{I}} \mid t \in P_{\mathcal{G}}(w)\}.$$

  in polynomial time and space.

□

So far, there is one essential ingredient missing: efficiency. Since the size of the search space may be exponential in terms of the input size, an ADP problem can be solved in polynomial time and space only under a condition known as Bellman's principle of optimality. In his own words:

> "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." [Bel57]

We formalize this principle:

Definition 6 (Algebraic version of Bellman's principle.) For each k-ary operator f in Σ, and all answer lists $z_1, \ldots, z_k$, the objective function h satisfies

$$\begin{aligned} & h(\ [\ f(x_1, \ldots, x_k) \mid x_1 \leftarrow z_1, \ldots, x_k \leftarrow z_k\ ]\ ) \\ = & h(\ [\ f(x_1, \ldots, x_k) \mid x_1 \leftarrow h(z_1), \ldots, x_k \leftarrow h(z_k)\ ]\ ) \end{aligned}$$

Additionally, the same property holds for the concatenation of answer lists:

$$h(\ z_1 \mathbin{+\!\!+} z_2\ ) = h(\ h(z_1) \mathbin{+\!\!+} h(z_2)\ )$$

□

The practical meaning of the optimality principle is that we may push the application of the objective function inside the computation of subproblems, thus preventing combinatorial explosion. We shall annotate the tree grammar to indicate the cases where h is to be applied.

3.6 ADP notation

For practical programming in the ADP framework, we introduce a simple language. The declarative semantics of this language is simply that it allows to describe signatures, evaluation algebras and yield grammars. The signature Σ is written as an algebraic data type definition in Haskell style. Alike EBNF, the productions of the yield grammar are written as equations. The operator `<<<` is used to denote the application of a tree constructor to its arguments, which are chained via the `~~~`-operator. Operator `|||` separates multiple righthand sides of a nonterminal symbol. Parentheses are used as required for larger trees. The axiom symbol is indicated by the keyword `axiom`, and syntactic conditions may be attached to productions via the keyword `with`. Using this notation, we write the signature Π and the grammar `globsim`:

```
data Alignment = Nil Char          |
                 D  Char Alignment |
                 I  Alignment Char |
                 R  Char Alignment Char

globsim alg = axiom alignment where
  (nil, d, i, r, h) = alg

  alignment  = nil <<< char '$'                          |||
               d   <<< achar  ~~~ alignment              |||
               i   <<<            alignment ~~~ achar    |||
               r   <<< achar  ~~~ alignment ~~~ achar    ... h
```

3.7 Parsing, tabulation and choice

Given a yield grammar and an evaluation algebra, a tabulating yield parser will solve a problem instance as declared in Definition 5. Implementation of yield parsing is explained in detail in Section 5.2. For programming with ADP, we do not really have to know how yield parsing works. Think of it as a family of recursive functions, one for each nonterminal of the grammar. However, the yield parser needs two pieces of information not yet expressed in the grammar: Tabulation and choice.

If nothing is said about tabulation, the yield parser may compute partial results many times, quite like our original Fibonacci function. By adding the keyword `"tabulated"`, we indicate that the parser for a particular nonterminal symbol shall make use of tabulation. When a tabulated symbol `v` is used in a righthand side, we write `p v` instead of `v`, indicating that this means a table lookup rather than a recursive call. Using both `tabulated` and `p v` is actually redundant, but facilitates checking the grammar for consistency.

If nothing was said about choice, the parser would not apply the objective function and hence return a list of all answers. By adding `"... h"` to the right-hand side of a production, we indicate that whenever a list of alternative answers has been constructed according to this production, the objective function `h` is to be applied to it.

With these two kinds of annotation, our yield grammar example `globsim` looks like this:

```
globsim alg = axiom (p alignment) where
  (nil, d, i, r, h) = alg

  alignment  = tabulated(
               nil <<< char '$'                          |||
               d   <<< achar  ~~~ p alignment            |||
               i   <<<            p alignment ~~~ achar  |||
               r   <<< achar  ~~~ p alignment ~~~ achar  ... h)
```

3.8 Efficiency analysis of ADP programs

From the viewpoint of programming methodology, it is important that asymptotic efficiency can be analyzed and controlled on the abstract level. This property is a major virtue of ADP – it allows to formulate efficiency tuning as grammar and algebra transformations. Such techniques are described in [GM02]. Here we give only the definition andthe theorem essential for determining the efficiency of an ADP algorithm.

Definition 7 (Width of productions and grammar.) Let t be a tree pattern, and let k be the number of nonterminal or lexical symbols in t whose yield size is not bounded by a constant. We define $width(t) = k - 1$. Let π be a production $v \to t_1 | \ldots | t_r$. $width(\pi) = max\{width(t_1, \ldots, t_r)\}$, and $width(\mathcal{G}) = max\{width(\pi) \mid \pi \text{ production in } \mathcal{G}\}$. □

Theorem 8 Assuming the number of answers is bounded by a constant, the execution time of an ADP algorithm described by tree grammar $\mathcal{G}$ on input w of length n is $O(n^{2+width(\mathcal{G})})$.

Proof: See [GS02] □

3.9 Summary

By means of an evaluation algebra and a yield grammar we can completely specify a dynamic programming algorithm. We can execute it using a yield parser, and analyze its efficiency using Theorem 8. This completes our framework. Let us summarize the key ideas of algebraic dynamic programming:

Phase separation: We conceptually distinguish recognition and evaluation phase.
Term representation: Individual candidates are represented as elements of a term algebra T_Σ; the set of all candidates is described by a tree grammar.
Recognition: The recognition phase constructs the set of candidates arising from a given input string, using a tabulating yield parser.
Evaluation: The evaluation phase interprets these candidates in a concrete Σ-algebra, and applies the objective function to the resulting answers.
Phase amalgamation: To retain efficiency, both phases are amalgamated in a fashion transparent to the programmer.

The virtue of this approach is that the conglomeration of issues criticised above – the traditional recurrences deal with search space construction, evaluation and efficiency concerns in a non-separable way – is resolved by algorithm development on the more abstract level of grammars and algebras.

4 Four example problems and variations, solved with ADP

In this chapter, we shall solve our four introductory problems with ADP. We shall emphasize the systematics of this effort, and in all cases, we shall proceed as follows:

1. We design the signature, representing explicitly all cases that might influence evaluation.
2. We design three or more evaluation algebras:
   - the enumeration algebra, implementing the enumeration of all candidates of a particular problem instance,
   - the counting algebra, computing the size of the search space, in a way much more efficient than by enumeration,
   - one or more scoring algebras, solving our optimization problem.
3. We specify the yield grammar in ADP notation, and apply the ADP program to some simple example data.
4. We formulate some problem variations.

Executable versions of these algorithms can be found at `http://bibiserv.techfak.uni-bielefeld.de/adp`.

4.1 El Mamun's problem

The signature Rather than adding parentheses, our signature `Bill` introduces operators `Add` and `Mult` to make explicit the different possible internal structures of El Mamun's bill.

```
data Bill = Mult  Bill Char Bill |
            Add   Bill Char Bill |
            Ext   Bill Char      |
            Val   Char
```

In the sequel, we consider the three different readings of El Mamun's bill as discussed in Section 2.1:

the original bill:	$(1+2)*((3*4)+5)$
the dealer's reconstruction:	$((1+2)*3)*(4+5)$
El Mamun's favourite:	$1+((2*(3*4))+5)$

Figure 6 shows the term representations of these candidates, and their tree representations.

The evaluation algebras

The enumeration and the counting algebra.

Ans_{enum} = T_{Bill}

```
enum = (val,ext,add,mult,h) where
val      = Val
ext      = Ext
add      = Add
mult     = Mult
h        = id
```

Ans_{count} = $\mathbb{N}$

```
count = (val,ext,add,mult,h) where
val(c)          = 1
ext(n,c)        = 1
add(x,t,y)      = x * y
mult(x,t,y)     = x * y
h([])           = []
h([x_1,...,x_r]) = [x_1 + ··· + x_r]
```

```
original = Mult (Add (Val '1') '+' (Val '2')) '*' (Add (Mult (Val '3')
'*' (Val '4')) '+' (Val '5'))

dealer = Mult (Mult (Add (Val '1') '+' (Val '2')) '*' (Val '3')) '*'
(Add (Val '4') '+' (Val '5'))

elmamun = Add (Val '1') '+' (Add (Mult (Val '2') '*' (Mult (Val '3')
'*' (Val '4'))) '+' (Val '5'))
```

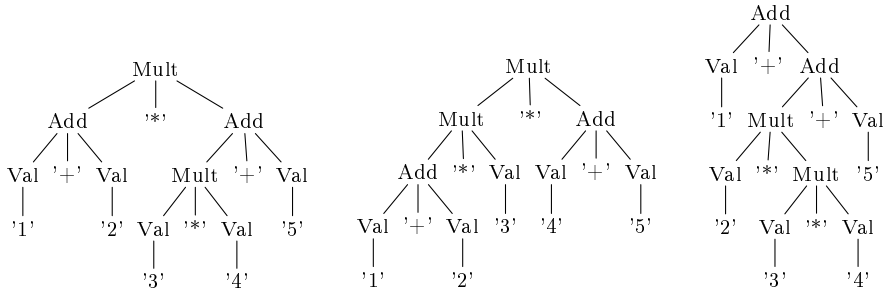

Fig. 6. The term representations of the three candidates for El Mamun's bill and their tree visualizations.

The buyer's and the seller's algebra. The following two scoring algebras use the function `decode` to convert a digit to an integer value. The buyer, of course, seeks to minimize the price, the seller seeks to maximize it.

Ans_{buyer} = $\mathbb{N}$

```
buyer = (val,ext,add,mult,h) where
val(c)      = decode(c)
ext(n,c)    = 10 * n * decode(c)
add(x,t,y)  = x + y
mult(x,t,y) = x * y
h([])       = []
h (l)       = [minimum(l)]
```

Ans_{seller} = $\mathbb{N}$

```
seller = (val,ext,add,mult,h) where
val(c)      = decode(c)
ext(n,c)    = 10 * n * decode(c)
add(x,t,y)  = x + y
mult(x,t,y) = x * y
h([])       = []
h (l)       = [maximum(l)]
```

The yield grammar The yield grammar describes all possible internal readings of El Mamun's formula (and any other such formula).

```
bill alg  = axiom (p formula) where
  (val, ext, add, mult, h) = alg

  formula = tabulated (
            number |||
            add  <<< p formula ~~~ plus  ~~~ p formula |||
            mult <<< p formula ~~~ times ~~~ p formula ... h)
```

```
number = val <<< digit ||| ext <<< number ~~~ digit
digit  = char '0' ||| char '1' ||| char '2' ||| char '3' |||
         char '4' ||| char '5' ||| char '6' ||| char '7' |||
         char '8' ||| char '9'

plus   = char '+'
times  = char '*'
```

Using this grammar, the four algebras, and input `z = "1+2*3*4+5"`, we obtain:

```
bill enum   = [Add (Val '1') '+' (Add (Mult (Val '2') '*' (Mult (Val '3')
 '*' (Val '4'))) '+' (Val '5')), Add (Val '1') '+' (Add (Mult (Mult (Val
 '2') '*' (Val '3')) '*' (Val '4')) '+' (Val '5')), Add (Val '1') '+'
 (Mult (Val '2') '*' (Add (Mult (Val '3') '*' (Val '4')) '+' (Val '5'))),
 ...]
bill count  = [14]
bill buyer  = [30]
bill seller = [81]
```

The first call yields a protocol of the complete search space traversed in all four cases. This is feasible only for small inputs, but is a most helpful testing aid. The second call merely computes the size of the search space - for all `z`, note the invariance `[length(bill enum)] = bill count`. The other two calls solve the problems of minimizing and maximizing the value of the formula.

Problem variation: *A processor allocation problem*

Computation in the days of El Mamun was very slow. A good computing slave took about 2 minutes to perform an addition, and 5 minutes to perform a multiplication. Even then, understanding the value of a number (once read) took practically no time (0 minutes). Fortunately, there were slaves abound, and they could work in parallel as much as the formula permitted. The following algebra selects for the formula that has the shortest computation time:

Ans_{time} = $\mathbb{N}$ (meaning minutes)

```
time = (val,ext,add,mult,h)  where
val(c)      = 0
ext(n,c)    = 0
add(x,t,y)  = max(x,y) + 2
mult(x,t,y) = max(x,y) + 5
h([])       = []
h (l)       = [minimum(l)]
```

Evaluating the three candidates shown in Figure 6 we find computation times between 12 and 14 minutes

```
h[original,dealer,elmamun] =
minimum[12,  12,   14    ] = 12
```

and we find that 12 minutes is actually optimal: `bill time = [12]`.

Exercise 6 Modify the algebra to compute the average computation time of a formula under the assumption that all possible readings are equally likely.□

4.2 Optimal matrix chain multiplication

The signature As in the previous example, we introduce two operators to represent parenthesization of an expression. An expression can consist of a single matrix or of a multiplication of two expressions.

```
data Matrixchain = Mult   Matrixchain Matrixchain |
                   Single (Int, Int)
```

Taking from Section 2.2 our example matrices, $A_1 : 10 \times 100, A_2 : 100 \times 5$ and $A_3 : 5 \times 50$, we get two candidates for matrix chain multiplication. Figure 7 shows the term representation of these candidates and their tree representation.

```
candidate1 = Mult (Single (10,100)) (Mult (Single (100,5)) (Single
(5,50)))

candidate2 = Mult (Mult (Single (10,100)) (Single (100,5))) (Single
(5,50))
```

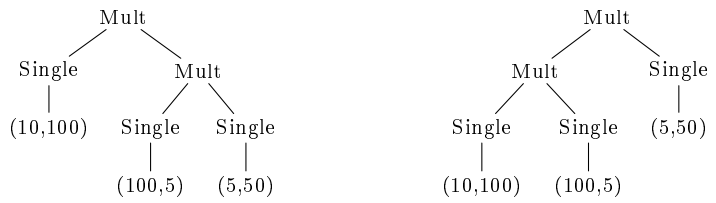

Fig. 7. The term representations of the two candidates for the example matrices and their tree representations.

The evaluation Algebras

The enumeration and the counting algebra.

$Ans_{enum} = T_{Matrixchain}$

```
enum = (single,mult,h) where
single   = Single
mult     = Mult
h        = id
```

$Ans_{count} = \mathbb{N}$

```
count = (single,mult,h) where
single((r,c)) = 1
mult(x,y)     = x * y
h([])         = []
```
$\texttt{h([}x_1, \ldots, x_r\texttt{])} = \texttt{[}x_1 + \cdots + x_r\texttt{]}$

The scoring algebra The algebra for determining the minimal number of scalar multiplications uses a triple (r, m, c) as answer type. (r, c) denotes the dimension of the resulting matrix and m the minimal number of operations needed to calculate it. With this answer type writing down the algebra is simple:

```
Ans_minmult                  = ℕ × ℕ × ℕ

minmult = (single,mult,h) where
single((r,c))                = (r,0,c)
mult((r,m,c),(r',m',c'))     = (r,m+m' + r*c*c',c')
h([])                        = []
h (l)                        = [minimum(l)]
```
[1]

The yield grammar The yield grammar describes all possible combinations of parentheses.

```
matrixmult alg  = axiom (p matrices) where
   (single, mult, h) = alg

   matrices = tabulated (
              single <<< achar                      |||
              mult   <<< p matrices ~~~ p matrices ... h )
```

For input `z = [(10,100),(100,5),(5,50)]` we obtain:

```
matrixmult enum     = [Mult (Single (10,100)) (Mult (Single (100,5))
 (Single (5,50))),Mult (Mult (Single (10,100)) (Single (100,5)))
 (Single (5,50))]
matrixmult count   = [2]
matrixmult minmult = [(10,7500,50)]
```

Problem variation: *Minimizing intermediate storage*

Another interesting exercise is to determine the optimal evaluation order for minimizing the memory usage needed for processing the matrix chain. This is motivated by the fact that memory allocated during calculation can be released in succeeding steps. Consider two matrix chains C_1 and C_2. For multiplying $C_1 * C_2$ we have two possible orders of calculation. When processing C_1 first we have to store the resulting matrix while processing C_2 and then store both results during this multiplication. As a second possibility, we can process C_2 first and store the resulting matrix while calculating C_1. Let $maxloc\ C$ be the biggest memory block allocated during calculation of matrix chain C. Let $loc\ C$ be the size of the resulting matrix. $loc\ A_i = 0$ for all input matrices. The minimal memory usage for processing $C_1 * C_2$ is given by

[1] The objective function considers all three triple elements for minimization. But since `r` and `c` are the same for all candidates for a fixed subchain, only `m` is relevant to this operation.

$$
\begin{aligned}
&maxloc\ C_1\ C_2 = \\
&\qquad min\{max\{maxloc\ C_1, loc\ C_1 + maxloc\ C_2, loc\ C_1 + loc\ C_2 + loc\ C_1C_2\} \\
&\qquad\qquad max\{maxloc\ C_2, loc\ C_2 + maxloc\ C_1, loc\ C_1 + loc\ C_2 + loc\ C_1C_2\}\}
\end{aligned} \tag{22}
$$

This can be expressed by the following algebra:

```
Ans_minmem                  = ℕ × ℕ × ℕ

minmem = (single,mult,h) where
single((r,c))               = (r,0,c)
mult((r,m,c),(r',m',c'))    = (r, minimum
                            [maximum [m,r*c+ m',r*c + r'* c' + r*c'],
                            maximum [m',r'*c'+ m,r*c + r'* c' + r*c']],c')
h([])                       = []
h (l)                       = [minimum(l)]
```

Exercise 7 It is true that this approach determines the minimal memory usage of a given matrix chain problem, but it does not report the responsible candidates of the solutions. Find a simple extension to the grammar which also reports an optimal candidate.□

4.3 Global and local similarity in strings and biosequences

The signature

```
data Alignment = Nil Char                |
                 D  Char Alignment       |
                 I  Alignment Char       |
                 R  Char Alignment Char
```

Figure 8 shows the term representation of a global similarity candidate and its tree representation.

```
candidate1 = D 'd' (R 'a' (I (R 'r' (R 'l' (R 'i' (R 'n' (R 'g' (Nil
'$') 'e') 'n') 'i') 'l') 'r') 'i') 'a')
```

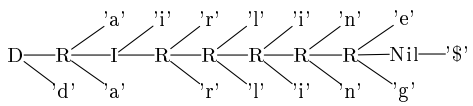

Fig. 8. The term representation of a global similarity candidate `candidate1` for `darling` and `airline` and the tree representation of this term (lying on its side).

The evaluation algebras

The enumeration and the counting algebra

$Ans_{enum} = T_{Alignment}$

```
enum = (nil,d,i,r,h) where
nil      = Nil
d        = D
i        = I
r        = R
h        = id
```

$Ans_{count} = \mathbb{N}$

```
count = (nil,d,i,r,h) where
nil(a)        = 1
d(x,s)        = s
i(s,y)        = s
r(a,s,b)      = s
h([])         = []
h([x_1,...,x_r]) = [x_1 + ··· + x_r]
```

The scoring algebras

$Ans_{wgap} = \mathbb{N}$

```
wgap = (nil,d,i,r,h) where
nil(a)   = 0
d(x,s)   = s + gap(x)
i(s,y)   = s + gap(y)
r(a,s,b) = s + w(a,b)
h([])    = []
h (l)    = [maximum(l)]
```

$Ans_{unit} = \mathbb{N}$

```
unit = (nil,d,i,r,h) where
nil(a)   = 0
d(x,s)   = s - 1
i(s,y)   = s - 1
r(a,s,b) = if a==b then s + 1 else s - 1
h([])    = []
h (l)    = [maximum(l)]
```

The yield grammars

Global similarity The yield grammar describes all possible ways to transform one string into the other by character replacement, deletion and insertion.

```
globsim alg = axiom (p alignment) where
  (nil, d, i, r, h) = alg

  alignment  = tabulated(
               nil <<< char '$'                          |||
               d   <<< achar  ~~~ p alignment            |||
               i   <<<            p alignment ~~~ achar  |||
               r   <<< achar  ~~~ p alignment ~~~ achar  ... h)
```

For input `z = "darling$enilria"` we obtain:

```
globsim enum  = [D 'd' (D 'a' (D 'r' (D 'l' (D 'i' (D 'n' (D 'g' (I (I (I
 (I (I (I (I (Nil '$') 'e') 'n') 'i') 'l') 'r') 'i') 'a')))))))),D 'd' (D
 'a' (D 'r' (D 'l' (D 'i' (D 'n' (I (D 'g' (I (I (I (I (I (I (Nil '$')
 'e') 'n') 'i') 'l') 'r') 'i')) 'a')))))),D 'd' (D 'a' (D 'r' (D 'l' (D
 'i' (D 'n' (I (I (D 'g' (I (I (I (I (I (Nil '$') 'e') 'n') 'i') 'l')
 'r')) 'i') 'a')))))...]
globsim count = [48639]
globsim unit  = [2]
```

Local similarity To formulate the yield grammar for local similarity, we modify the signature. We introduce two new operators `skip_left` and `skip_right` for skipping characters in the beginning of x and y. To allow skipping at the end of x and y, we modify the argument of `Nil` to be an arbitrary string, including the separator symbol.

```
locsim alg = axiom (p loc_align) where
  (nil, d, i, r, h) = alg

  skip_right a b    = a
  skip_left  a b    = b

  loc_align = tabulated (
              p alignment                                    |||
              skip_right <<<           p loc_align ~~~ achar |||
              skip_left  <<< achar ~~~ p loc_align           ... h)

  alignment  = tabulated (
               nil <<< string                          |||
               d   <<< achar ~~~ p alignment           |||
               i   <<<           p alignment ~~~ achar |||
               r   <<< achar ~~~ p alignment ~~~ achar ... h)
```

For input `z = "darling$enilria"` we obtain:

```
locsim enum  = [Nil (0,15),D 'd' (Nil (1,15)),D 'd' (D 'a' (Nil (2,15))),
 D 'd' (D 'a' (D 'r' (Nil (3,15)))), ...]
locsim count = [682662]
locsim unit  = [4]
```

Problem variation: *Affine gap scores*

In the algebras presented so far, succeeding insertions respectively deletions achieve the same score as the same number of single gaps (deletions and insertions). But in order to analyze biological sequence data, it is more adequate to use an affine gap score model. This means to assign an opening cost (`open`) to each gap and an extension cost (`extend`) for each deleted respectively inserted character. This results in a better model favouring few long gaps than having over many short gaps. The use of affine gap scores was introduced in [Got82].

The signature In order to distinguish the opening of a gap and the extension of a gap we have to extend the signature `Alignment`:

```
data Alignment = Nil Char               |
                 D  Char Alignment      |
                 I  Alignment Char      |
                 R  Char Alignment Char |
                 Dx Char Alignment      |
                 Ix Alignment Char
```

Affine gap score algebra

$Ans_{affine} = \mathbb{N}$

```
affine = (nil,d,i,r,dx,ix,h) where
nil(a)    = 0
d(x,s)    = s + open + extend
i(s,y)    = s + open + extend
r(a,s,b)  = s + w(a,b)
dx(x,s)   = s + extend
ix(s,y)   = s + extend
h([])     = []
h (l)     = [maximum(l)]
```

The yield grammar In the modified yield grammar for global similarity, we have to distinguish the opening of a gap and the extension of a gap.

```
affineglobsim alg = axiom (p alignment) where
  (nil, d, i, r, dx, ix, h) = alg

  alignment = tabulated (
               nil <<< char '$'                          |||
               d   <<< achar  ~~~ p xDel                 |||
               i   <<<            p xIns      ~~~ achar |||
               r   <<< achar  ~~~ p alignment ~~~ achar ... h)

  xDel      = tabulated (
              p alignment              |||
              dx <<< achar  ~~~ p xDel ... h )

  xIns      = tabulated (
              p alignment              |||
              ix <<< p xIns ~~~ achar  ... h )
```

To achieve the yield grammar for local alignments using the affine gap score model, the grammar for global alignments has to be modified in the same manner as shown for the simple gap score model.

4.4 Analyses on Fibonacci

The examples we have seen so far work on sequences of input data. Calculating the Fibonacci numbers $F(n)$ with this approach seems inappropriate, since the input consists of a single number here. Of course, n can be encoded in unary as a sequence of length n. But why at all should one want to compute $F(n)$ in such a complicated way? We shall provide a grammar which captures the Fibonacci recursion pattern. Each string of length n gives rise to $F(n)$ parses, and therefore $F(n)$ is computed by the counting algebra. The interesting part here is the use of other evaluation algebras, which can be designed to compute other properties of Fibonacci numbers.

The signature

```
data Fib = F1  Int         |
           F2  Int Int     |
           Fn  Fib         |
           Fn1 Fib Int     |
           Fn2 Fib Int Int
```

The key idea is that an input string[2] of length n gives rise to $F(n)$ different candidates. Operators `F1`, `F2` and `Fn` shall represent the three equations of the original Fibonacci definition. Operators `Fn1` and `Fn2` denote the function calls $F(n-1)$ and $F(n-2)$ – here the input is shortened by 1 respectively 2 elements. Figure 9 shows the term and tree representations of the two candidates for input length $n = 3$.

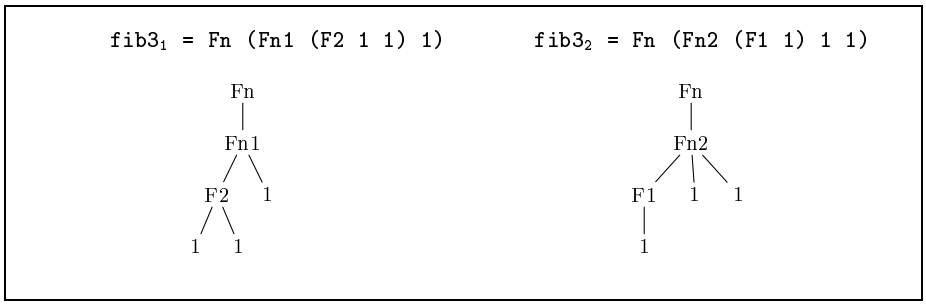

Fig. 9. The term and tree representations of the two candidates for $F(3)$.

Evaluation algebras

The enumeration and the counting algebra.

Ans_{enum} = T_{Fib}

```
enum = (f1,f2,fn,fn1,fn2,h) where
f1       = F1
f2       = F2
fn       = Fn
fn1      = Fn1
fn2      = Fn2
h        = id
```

Ans_{count} = $\mathbb{N}$

```
count = (f1,f2,fn,fn1,fn2,h) where
f1(a1)          = 1
f2(a1,a2)       = 1
fn(x)           = x
fn1(x,a1)       = x
fn2(x,a1,a2)    = x
h([])           = []
h([x_1,...,x_r]) = [x_1 + ··· + x_r]
```

[2] For reasons we will see in the course of this example, we choose input type `Int` here.

The yield grammar

```
fibonacci alg  = axiom (p fib) where
  (f1, f2, fn, fn1, fn2, h) = alg

  fib = tabulated(
         f1 <<< achar                        |||
         f2 <<< achar ~~~ achar              |||
         fn <<< fibn                        `with`    minsize 3)

  fibn = fn1 <<< p fib ~~~ achar             |||
         fn2 <<< p fib ~~~ achar ~~~ achar ... h
```

Note that this example makes use of the syntactic predicate `minsize 3` which guarantees that the alternative `fn <<< fibn` is only considered for an input of at least 3. Otherwise we would obtain two possible derivations for `fib 2`.
For input `z = [1,1,1]` we obtain:

```
fibonacci enum  = [Fn (Fn1 (F2 1 1) 1),Fn (Fn2 (F1 1) 1 1)]
fibonacci count = [2]
```

Problem variations: *Properties of Fibonacci*

In the rest of this section, we show how ADP can be of some use in combinatorics. It can be used to compute certain properties or to test a certain hypothesis. Of course, it does not do inductive poofs, nor can it provide a closed formula for a combinatorial problem.

In Section 2.4 we made two statements on the properties of F as a recursive function: It takes exactly $2F(n) - 1$ calls to compute $F(n)$, and each value $F(n-k)$, with $n - k > 1$, is calculated $F(k+1)$ times. Giving an algebra for testing the first statement only needs a slight modification to the above one:

```
Ans_calls          = ℕ

calls = (f1,f2,fn,fn1,fn2,h) where
f1(a1)             = 1
f2(a1,a2)          = 1
fn(x)              = 1 + x
fn1(x,a1)          = x
fn2(x,a1,a2)       = x
h([])              = []
h([x_1,...,x_r])   = [x_1 + ··· + x_r]
```

For the second statement we need to introduce some more extensive changes to the algebra, both in type of the answer value, the definition of the evaluation functions, and our interpretation of the input sequence. The two parameters n and k shall be represented as a sequence of length n, with each element the numeric value $m = n - k$.

```
Ans_calc                          = ℕ × ℕ × ℕ   (calculations of F(m), n, m)
calc = (f1,f2,fn,fn1,fn2,h) where
f1(m)                             = if m==1 then (1,1,m) else (0,1,m)
f2(m,m')                          = if m==2 then (1,2,m) else (0,2,m)
fn((c,n,m))                       = if n==m then (c+1,n,m) else (c,n,m)
fn1((c,n,m),m')                   = (c,n+1,m)
fn2((c,n,m),m',m'')               = (c,n+2,m)
h([])                             = []
h([(c_1, n_1, m_1), ..., (c_r, n_r, m_r)]) = [(c_1 + ... + c_r, n_1, m_1)]
```

5 Three ways to implement ADP

5.1 Unifying single sequence analysis and pairwise sequence comparison

We have been considering two kinds of problems: In El Mamun's and in the matrix chain problem, the task was to recover an internal structure in a single sequence x. A candidate t for x has $yield(t) = x$. In the similarity problem, we are comparing two sequences x and y. We saw that here a candidate t for inputs x, y has $yield(t) = xy^{-1}$. If we choose to include a separator symbol \$ between x and the reverse of y, we have $yield(t) = x\$y^{-1}$. This is a matter of convenience. To unify both cases, in the sequel we assume we have a single input z, where either $z = x$ or $z = xy^{-1}$, or $z = x\$y^{-1}$. We assume that $z, m = |x|, n = |y|, l = |z|$ are known and represented by global variables. Thus, in the pairwise case, even when we do not use the separator symbol, we know the boundary between x and y^{-1} in z.

Since z is global, a subword $z_{i+1}, ..., z_j$ of z is simply represented by the subscript pair `(i,j)`. Note that i marks the subscript position *before* the first character of subword (i, j). This convention allows to use k as the common boundary of adjacent subwords when splitting (i, j) into (i, k) and (k, j).

The DP tables storing intermediate results for a subword (i, j) are triangular, since we always have $i \leq j$. In the pairwise case, the different parts A, B and C of the triangle, as indicated in Figure 10, have a different meaning. Entries $A_{i,j}$ are derived from the subword (i, j) of z, which is $x_{i+1}...x_j$. Entries $C_{i,j}$ are derived from the subword (i, j) of z, which is $y_{m+n+2-j}...y_{m+n+1-i}$. Entries $B_{i,j}$ are derived from $x_{i+1}...x_m$ and $y_{m+n+2-j}...y_n$. Depending on the problem at hand, only a part of A, B or C may need to be calculated; the global similarity problem, for example, only needs part B. We shall describe the implementation for the general case, however.

5.2 Embedding in Haskell

An algorithm written in ADP notation can be directly executed as a Haskell program. Of course, this requires that the functions of the evaluation algebra

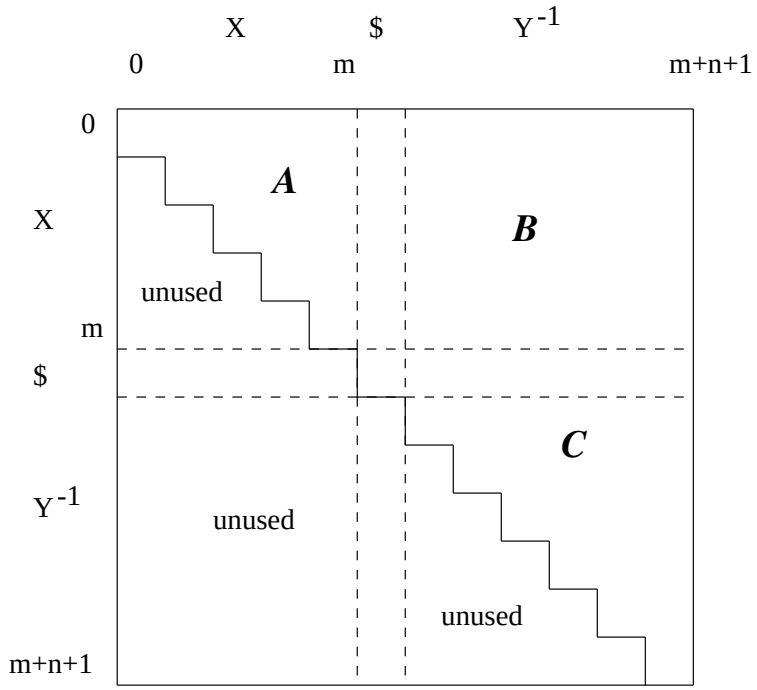

Fig. 10. Compartments of the triangular DP tables

are coded in Haskell. This smooth embedding is achieved by the technique of parser combinators [Hut92], which essentially turn the grammar into a parser. We introduce suitable combinator definitions for yield parsing, and add tabulation.

Generally, a parser is a function that, given a subword of the input, returns a list of all its parses.

Lexical parsers The lexical parser `achar` recognizes any single character except the separator symbol. Parser `string` recognizes a (possibly empty) subword. Specific characters or symbols are recognized by `char` and `symbol`. Parser `empty` recognizes the empty subword.

```
> type Subword  = (Int,Int)
> type Parser b = Subword -> [b]

> empty        :: Parser ()
> empty  (i,j) =  [() | i == j]

> achar        :: Parser Char
> achar  (i,j) =  [z!j | i+1 == j, z!j /= '$']
```

```
> char         :: Char -> Parser Char
> char c (i,j) =  [c | i+1 == j, z!j == c]

> string       :: Parser Subword
> string (i,j) =  [(i,j) | i <= j]

> symbol       :: String -> Parser Subword
> symbol s (i,j) = [(i,j)| and [z!(i+k) == s!!(k-1) | k <-[1..(j-i)]]
```

Nonterminal parsers The nonterminal symbols are interpreted as parsers, with the productions serving as their mutually recursive definitions. Each right-hand side is an expression that combines parsers using the parser combinators.

Parser combinators The operators introduced in the ADP notation are now defined as parser combinators: `|||` concatenates result lists of alternative parses, and `<<<` grabs the results of subsequent parsers connected via `~~~` and successively "pipes" them into the tree constructor. Combinator `...` applies the objective function to a list of answers.

```
> infixr 6 |||
> (|||)           :: Parser b -> Parser b -> Parser b
> (|||) r q (i,j) = r (i,j) ++ q (i,j)

> infix  8 <<<
> (<<<)           :: (b -> c) -> Parser b -> Parser c
> (<<<) f q (i,j) =  map f (q (i,j))

> infixl 7 ~~~
> (~~~)           :: Parser (b -> c) -> Parser b -> Parser c
> (~~~) r q (i,j) =  [f y | k <- [i..j], f <- r (i,k), y <- q (k,j)]

> infix  5 ...
> (...)           :: Parser b -> ([b] -> [b]) -> Parser b
> (...) r h (i,j) = h (r (i,j))
```

Note that the operator priorities are defined such that an expression `f <<< a ~~~ b ~~~ c` is read as `((f <<< a) ~~~ b) ~~~ c`. This makes use of curried functions: the results of parser `f <<< a` are calls to `f` with (only) the first argument fixed.

The operational meaning of a `with`-clause can be defined by turning `with` into a combinator, this time combining a parser with a filter. Finally, the keyword `axiom` of the grammar is interpreted as a function that returns all parses for the specified nonterminal symbol and the complete input.

```
> type Filter    =  (Int, Int) -> Bool
> with           :: Parser b -> Filter -> Parser b
> with q c (i,j) =  if c (i,j) then q (i,j)  else []
```

```
> axiom          :: Int -> Parser b -> [b]
> axiom ax       =  ax (0,l)
```

When a parser is called with the enumeration algebra – i.e. the functions applied are actually tree constructors, and the objective function is the identity function –, then it behaves like a proper yield parser and generates a list of trees according to Definition 4. However, called with some other evaluation algebra, it computes any desired type of answer.

Tabulation Adding tabulation is merely a change of data type, replacing a recursive function by a recursively defined table. We use a general scheme for this purpose: The function `table` records the results of a parser `p` for all subwords of an input of size `n`. The function `p` does table lookup. Note the essential invariance `p (table n f) = f`. Therefore, if `r` is a tabulated parser, then `p r` can be used as a parser function. This is another use of curried functions, which allows to freely combine tabulated and non-tabulated parsers in the grammar. The keyword `tabulated` is now defined as `table` bound to the global variable l, the length of the input.

```
> type Parsetable b = Array (Int,Int) [b]

> table     :: Int -> Parser b -> Parsetable b
> table n q =  array ((0,0),(n,n))
>                  [((i,j),q (i,j)) | i<- [0..n], j<- [i..n]]

> tabulated = table l

> p         :: Parsetable b -> Parser b
> p t (i,j) =  if i <= j then t!(i,j) else []
```

Removing futile computations Consider the production `a = f <<< a ~~~ char`. Our definition of the `~~~` combinator splits subword (i, j) in all possible ways, including empty subwords on either side. Obviously, `achar`, which recognizes a single character, has a fixed yield size of 1, leaving the subword $(i, j-1)$ for the yield of nonterminal symbol `a`. In this case, iteration over all splits of (i, j) into (i, k) and (k, j) is mathematically correct, but a futile effort. The only successful split can be $(i, j-1)$ and $(j-1, j)$. What is worse, since the production is left-recursive, the last split considered without need is (i, j) and (j, j), which leads to infinite recursion.

Both situations are avoided by using specializations of the `~~~` combinator that are aware of bounded yield sizes and avoid unneccessary splits. For the case of splitting of a single character, we use `-~~` and `~~-`, while the fully general case of an arbitrary, but known yield size limit is treated by the `~~` combinator.

```
> infixl 7 ~~,-~~ , ~~-

> (-~~) q r (i,j) = [x y | i<j, x <- q (i,i+1), y <- r (i+1,j)]
> (~~-) q r (i,j) = [x y | i<j, x <- q (i,j-1), y <- r (j-1,j)]


> (~~) :: (Int,Int) -> (Int,Int)
>      -> Parser (b -> c) -> Parser b -> Parser c
> (~~) (l,u) (l',u') r q (i,j)
>      = [x y | k <- [max (i+l) (j-u') .. min (i+u) (j-l')],
>               x <- r (i,k), y <- q (k,j)]
```

These combinators are used in asymptotic efficiency tuning via width reduction as described in [GM02]. Using these special cases, our global similarity grammar is now written in the form

```
> globsim alg = axiom (p alignment) where
>   (nil, d, i, r, h) = alg

>   alignment  = tabulated(
>                nil <<< char '$'                            |||
>                d   <<< achar  -~~ p alignment              |||
>                i   <<<            p alignment ~~- achar    |||
>                r   <<< achar  -~~ p alignment ~~- achar    ... h)
```

which now, as a functional program, has the appropriate efficiency of $\mathcal{O}(mn)$.

5.3 Deriving explicit recurrences

In the previous section we showed how to embed an ADP algorithm smoothly in a functional language. Although there exist efficient implementations of Haskell, it still seems desirable to derive an imperative version of the algorithm. The sheer amount of data volume present in most dynamic programming domains and an easy integration in existing systems are two of the reasons. The classic approach in dynamic programming is to implement the imperative version of the algorithm starting from the matrix recurrences derived by experience or intuition. In this section we show how to derive the recurrences in a systematic way from an algorithm in ADP notation.

Each tabulated production will result in a matrix recurrence relation. The definitions of non tabulated productions can be inserted directly at the occurrences of the corresponding nonterminal symbols in the grammar. In the following, we assume that all productions are tabulated.

Translation patterns The matrix recurrences for a grammar $\mathcal{G}$ can be derived by the following translation patterns starting with $\mathcal{C}[\![\mathcal{G}]\!]$ in Pattern 23. We use list comprehension notation, in analogy to set notation: $[f(x,y)|x \in xs, y \in ys]$ denotes the list of all values $f(x,y)$ such that x is from list xs and y from

list ys. To distinguish a parser call $q(i,j)$ from a semantically equivalent table lookup, we denote the latter by $q!(i,j)$. The function pair $(low(p), up(p))$ shall provide the yield size of a tree pattern p and is defined by $(low(p), up(p)) = (\inf_{q \in \mathcal{L}(p)} |q|, \sup_{q \in \mathcal{L}(p)} |q|)$ if $\mathcal{L}(p) \neq \emptyset$, and $(low(p), up(p)) = (\infty, 0)$ otherwise.

$$\mathcal{C}[\![\texttt{grammar alg} \ = \ \texttt{axiom p where v}_1 = \texttt{q}_1 \ldots \texttt{v}_\texttt{m} = \texttt{q}_\texttt{m}]\!] = \tag{23}$$

$$\begin{array}{l}
\quad \text{for } j = 0 \text{ to } l \\
\qquad \text{for } i = 0 \text{ to } j \\
\qquad\quad v_1!(i,j) = \mathcal{C}[\![\texttt{q}_1]\!](i,j) \\
\qquad\quad \vdots \\
\qquad\quad v_m!(i,j) = \mathcal{C}[\![\texttt{q}_\texttt{m}]\!](i,j) \\
\quad \text{return } p!(0,l)
\end{array}$$

$$\mathcal{C}[\![\texttt{q} \ \ldots \texttt{h}]\!](i,j) \quad = h(\mathcal{C}[\![\texttt{q}]\!](i,j)) \tag{24}$$

$$\mathcal{C}[\![\texttt{q}_1 \ \texttt{|||} \ \ldots \ \texttt{|||} \ \texttt{q}_\texttt{r}]\!](i,j) \quad = \mathcal{C}[\![\texttt{q}_1]\!](i,j) \ ++ \ldots ++ \mathcal{C}[\![\texttt{q}_\texttt{r}]\!](i,j) \tag{25}$$

$$\begin{array}{ll}
\mathcal{C}[\![\texttt{t <<< q}_1 \ \texttt{~~~} \ \ldots \ \texttt{~~~} \ \texttt{q}_\texttt{r}]\!](i,j) & = \\
\quad [t(p_1, \ldots, p_r) | p_1 \in \mathcal{C}[\![\texttt{q}_1]\!](i,k_1), \ldots, p_r \in \mathcal{C}[\![\texttt{q}_\texttt{r}]\!](k_{r-1}, j)] & \\
\quad \text{for } k_1, \ldots, k_{r-1}, \quad \text{such that } \ k_0 = i, \ k_r = j, & \\
\qquad max(k_{l-1} + low(q_l), k_{l+1} - up(q_{l+1})) \leq k_l \leq & \\
\qquad min(k_{l-1} + up(q_l), k_{l+1} - low(q_{l+1})) &
\end{array} \tag{26}$$

$$\mathcal{C}[\![\texttt{q with c}]\!](i,j) \quad = \text{if } c(i,j) \text{ then } \mathcal{C}[\![\texttt{q}]\!](i,j) \text{ else } [\,] \tag{27}$$

$$\mathcal{C}[\![\texttt{v}]\!](i,j) \quad = v!(i,j) \qquad \text{for } v \in V \tag{28}$$

$$\mathcal{C}[\![\texttt{t}]\!](i,j) \quad = \mathcal{T}[\![\texttt{t}]\!](i,j) \qquad \text{for } t \text{ terminal} \tag{29}$$

In Pattern 26, note the direct correspondence to the definition of the ~~ combinator in Section 5.2.

Translation patterns $\mathcal{T}[\![\texttt{w}]\!]$ for terminal symbols must be chosen according to their respective semantics. We give three examples:

$$\begin{array}{ll}
\mathcal{T}[\![\texttt{char c}]\!](i,j) & = \text{if } i+1 \equiv j \wedge z_j \equiv c \text{ then } [c] \text{ else } [\,] \\
\mathcal{T}[\![\texttt{achar}]\!](i,j) & = \text{if } i+1 \equiv j \wedge z_j \neq \text{'\$'} \text{ then } [z_j] \text{ else } [\,] \\
\mathcal{T}[\![\texttt{string}]\!](i,j) & = \text{if } i \leq j \text{ then } [(i,j)] \text{ else } [\,]
\end{array}$$

Example We demonstrate the translation for the global similarity example of Section 4.3:

```
globsim alg = axiom alignment where
  (nil, d, i, r, h) = alg

  alignment  = nil <<< char '$'                          |||
               d   <<< achar ~~~ alignment               |||
               i   <<<           alignment ~~~ achar     |||
               r   <<< achar ~~~ alignment ~~~ achar     ... h
```

Applying Pattern 23 to this grammar provides the framework of the control structure:

$$
\begin{array}{l}
\text{for } j = 0 \text{ to } l \\
\quad \text{for } i = 0 \text{ to } j \\
\qquad alignment!(i,j) = \mathcal{C}[\![\texttt{nil <<< ...}]\!](i,j) \\
\text{return } alignment!(0,l)
\end{array}
$$

Starting with $alignment!(i,j) = \mathcal{C}[\![\texttt{nil <<< ...}]\!](i,j)$ we apply Patterns 24 and 25 to the righthand side of the production:

$$
\begin{array}{ll}
alignment!(i,j) = h(& \\
\quad \mathcal{C}[\![\texttt{nil <<< char '\$'}]\!](i,j) \mathbin{++} & (30) \\
\quad \mathcal{C}[\![\texttt{d <<< achar ~~~ alignment}]\!](i,j) \mathbin{++} & (31) \\
\quad \mathcal{C}[\![\texttt{i <<< alignment ~~~ achar}]\!](i,j) \mathbin{++} & (32) \\
\quad \mathcal{C}[\![\texttt{r <<< achar ~~~ alignment ~~~ achar}]\!](i,j)) & (33)
\end{array}
$$

The resulting four expressions can be translated separately according to Pattern 26. Expression 30 translates to:

$$
\begin{array}{l}
[nil(p_1) | p_1 \in \mathcal{T}[\![\texttt{char '\$'}]\!](i,j)] \\
= \text{if } i+1 \equiv j \wedge z_j \equiv \text{'\$'} \text{ then } [nil(\text{'\$'})] \text{ else } [\,]
\end{array}
$$

Translation of Expressions 31 – 33 makes use of the yield size functions *low* and *up*. Table 1 shows their values for the expressions needed in this example. The constant yield sizes of the terminal symbols can be taken directly from the corresponding parser definitions. For nonterminal symbols and arbitrary expressions this needs a deeper analysis of the grammer. This is detailed in [GS02]. Despite the risk of ending with suboptimal code for the resulting matrix recurrences, a yield size $(0, \infty)$ is always a safe approximation.

x	$(low(x), up(x))$
`char c`	$(1,1)$
`achar`	$(1,1)$
`alignment`	$(1,\infty)$

Table 1. Yield sizes needed for alignment example

Proceeding with Expression 31 leads to the following calculation:

$$
\begin{aligned}
&\mathcal{C}[\![\texttt{d <<< achar ~~~ alignment}]\!](i,j) \\
&= [d(p_1,p_2) | p_1 \in \mathcal{T}[\![\texttt{achar}]\!](i,k_1), p_2 \in \mathcal{C}[\![\texttt{alignment}]\!](k_1,j)] \\
&\qquad \text{for } k_1 \text{ such that} \\
&\qquad\qquad max(i + low(achar), j - up(alignment)) \leq k_1 \leq \\
&\qquad\qquad min(i + up(achar), j - low(alignment))
\end{aligned}
$$

With yield sizes $(1,1)$ and $(1,\infty)$ for `achar` and `alignment` the loop variable k_1 simplifies to a constant $k_1 = i+1$ and the condition $i+2 \leq j$:

$$
\begin{aligned}
&[d(p_1,p_2) | i+2 \leq j, p_1 \in \mathcal{T}[\![\texttt{achar}]\!](i,i+1), p_2 \in \mathcal{C}[\![\texttt{alignment}]\!](i+1,j)] \\
&= [d(p_1,p_2) | i+2 \leq j \wedge z_{i+1} \neq \text{'\$'}, p_1 \in [z_{i+1}], p_2 \in alignment!(i+1,j)] \\
&= [d(z_{i+1},p_2) | i+2 \leq j \wedge z_{i+1} \neq \text{'\$'}, p_2 \in alignment!(i+1,j)]
\end{aligned}
$$

Translating Expressions 32 and 33 in the same way we arrive at the following recurrence relation for the matrix *alignment*:

$$
\begin{aligned}
&alignment!(i,j) = h(\\
&\quad \text{if } i+1 \equiv j \wedge z_j \equiv \text{'\$'} \text{ then } [nil(\text{'\$'})] \text{ else } [\,] \mathbin{++} \\
&\quad [d(z_{i+1},p_2) | i+2 \leq j \wedge z_{i+1} \neq \text{'\$'}, p_2 \in alignment!(i+1,j)] \mathbin{++} \\
&\quad [i(p_1,z_j) | i+2 \leq j \wedge z_j \neq \text{'\$'}, p_1 \in alignment!(i,j-1)] \mathbin{++} \\
&\quad [r(z_{i+1},p_2,z_j) | i+3 \leq j \wedge z_{i+1} \neq \text{'\$'} \wedge z_j \neq \text{'\$'}, \\
&\qquad\qquad p_2 \in alignment!(i+1,j-1)])
\end{aligned}
\tag{34}
$$

The explicit recurrences derived so far can be used together with code implementing the functions of an arbitrary evaluation algebra. If this code is simple, it can be inlined, which often allows further simplification of the recurrences.

Inlining evaluation algebras We demonstrate inlining by means of the count algebra and the unit cost algebra introduced in Section 4.3:

Ans_{count} = $\mathbb{N}$

```
count = (nil,d,i,r,h) where
nil(x)        = 1
d(x,s)        = s
i(s,y)        = s
r(a,s,b)      = s
h([])         = []
h([x_1,...,x_r]) = [x_1 + ··· + x_r]
```

Ans_{unit} = $\mathbb{N}$

```
unit = (nil,d,i,r,h) where
nil(x)   = 0
d(x,s)   = s - 1
i(s,y)   = s - 1
r(a,s,b) = if a==b then s + 1 else s - 1
h([])    = []
h (l)    = [maximum(l)]
```

For the counting algebra this results in the following recurrence for the matrix *alignment*:

$$
\begin{aligned}
&\mathit{alignment}!(i,j) = \\
&\quad (\text{if } i+1 \equiv j \wedge z_j \equiv \text{'\$'} \text{ then } 1 \text{ else } 0) + \\
&\quad (\text{if } i+2 \leq j \wedge z_{i+1} \neq \text{'\$'} \text{ then } \mathit{alignment}!(i+1,j) \text{ else } 0) + \\
&\quad (\text{if } i+2 \leq j \wedge z_j \neq \text{'\$'} \text{ then } \mathit{alignment}!(i,j-1) \text{ else } 0) + \\
&\quad (\text{if } i+3 \leq j \wedge z_{i+1} \neq \text{'\$'} \wedge z_j \neq \text{'\$'} \text{ then } \mathit{alignment}!(i+1,j-1) \text{ else } 0)
\end{aligned}
$$

And for the unit cost algebra:

$$
\begin{aligned}
&\mathit{alignment}!(i,j) = \max(\\
&\quad (\text{if } i+1 \equiv j \wedge z_j \equiv \text{'\$'} \text{ then } [0] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_{i+1} \neq \text{'\$'} \text{ then } [\mathit{alignment}!(i+1,j)-1] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_j \neq \text{'\$'} \text{ then } [\mathit{alignment}!(i,j-1)-1] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+3 \leq j \wedge z_{i+1} \neq \text{'\$'} \wedge z_j \neq \text{'\$'} \text{ then } [\text{if } z_{i+1} \equiv z_j \\
&\qquad\qquad\qquad\qquad \text{then } \mathit{alignment}!(i+1,j-1)+1 \\
&\qquad\qquad\qquad\qquad \text{else } \ \mathit{alignment}!(i+1,j-1)-1] \text{ else } [\,]))
\end{aligned}
$$

Solving dependencies Consider the example of local similarity shown in Section 4.3. By adding the production

```
loc_align = alignment                                        |||
            skip_right <<<           loc_align ~~~ achar     |||
            skip_left  <<< achar ~~~ loc_align               ... h
```

we extended the algorithm for global similarity to an algorithm for local similarity. Following the translation scheme of the last paragraphs we derive the matrix recurrence for *loc_align*:

$$
\begin{aligned}
&\mathit{loc_align}!(i,j) = \max(\\
&\quad [\mathit{alignment}!(i,j)] \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_j \neq \text{'\$'} \text{ then } [\mathit{loc_align}!(i,j-1)] \text{ else } [\,]) \mathbin{++} \\
&\quad (\text{if } i+2 \leq j \wedge z_{i+1} \neq \text{'\$'} \text{ then } [\mathit{loc_align}!(i+1,j)] \text{ else } [\,]))
\end{aligned}
$$

The dependency between $\mathit{loc_align}!(i,j)$ and $\mathit{alignment}!(i,j)$ leads us to a new issue not present in the functional version of the algorithm. Functional languages are data-driven, so in the functional prototype of the algorithm the computational model of the programming language guarantees that all computations are made on demand. Since we cannot assume this in an imperative setting, we have to find a suitable ordering of calculation, so that all dependencies are solved and all values are calculated before they are used. For the small example shown here this is an easy task. But consider an algorithm with about 20 productions and various dependencies between them. Finding a suitable order of calculation is a strenuous and error-prone venture. Solving this problem is one of the tasks of the compiler described in the next section.

5.4 Compiling ADP notation to C

In the previous section we showed how to derive the traditional recurrences in a systematic way, as an intermediate step towards an implementation of an ADP algorithm in an imperative programming language, such as C. The C program can be tested systematically against the Haskell prototype, a procedure that guarantees much higher reliability than ad-hoc testing. Still, the main difficulties with this approach are twofold: It proves to be time consuming to produce by hand a C program equivalent to the Haskell prototype. Furthermore, for sake of efficiency, developers are tempted to perform ad-hoc yield size analysis and use special combinators in the prototype. This introduces through the backdoor the possibility of subscript errors otherwise banned by the ADP approach. The compiler currently under development eliminates both problems.

Aside from parsing the ADP program and producing C code, the core of the compiler implements yield size and dependency analysis, and performs the translation steps described in the previous section. With respect to the evaluation algebra we follow the strategy that simple arithmetic functions are inlined, while others must be provided as native C functions. Compiler options provide a simplified translation in the case where the evaluation algebra computes scalar answers rather than lists. As an example, the code produced for the grammar `globsim` is shown in Figure 11.

We also added a *source-to-source option* to the compiler, reproducing ADP input with all `~~~` operators replaced by variants bound to exact yield sizes. Hence, the user of the prototype is no longer committed to delicate tuning efforts.

6 Conclusion

6.1 Virtues of ADP

What has been achieved with respect to our goal of making dynamic programming a more systematic effort? First of all, we can give clear methodical guidance for the development of DP recurrences.

1. Design the signature Σ, representing explicitly all cases that might influence evaluation of candidates.
2. Design evaluation algebras, at least the enumeration, counting and one scoring algebra, the latter describing the objective of the application.
3. Design the grammar defining the search space.
4. Improve efficiency by grammar and algebra transformations (see below).
5. Test the design using search space enumeration and plausibility checking via the counting algebra.
6. Experiment with alternative algorithmic ideas using the functional prototype.
7. When the algorithm design has stabilized and has been successfully tested, generate the C implementation and validate it against the functional prototype.

```
void calc_alignment(int i, int j)
{
   struct t_result  *v[8];

   if ((j-i) == 1) { if (z[j] == '$') {
         v[0] = allocMem(0);
      } else { v[0] = NULL; };
   } else { v[0] = NULL; };                    /* nil x   = 0  */

   if ((j-i) >= 2) { if (z[i+1] != '$') {
         v[1] = allocMem(alignment[i+1][j] + gap(z[i+1]));
      } else { v[1] = NULL; };
   } else { v[1] = NULL; };           /* d x s   = s + gap(x)  */

   if ((j-i) >= 2) { if (z[j] != '$') {
         v[2] = allocMem(alignment[i][j-1] + gap(z[j]));
      } else { v[2] = NULL; };
   } else { v[2] = NULL; };           /* i s y   = s + gap(y)  */

   if ((j-i) >= 3) { if ((z[i+1] != '$') && (z[j] != '$')) {
         v[3] = allocMem(alignment[i+1][j-1] + w(z[i+1], z[j]));
      } else { v[3] = NULL; };
   } else { v[3] = NULL; };           /* r a s b = s + w(a,b)  */

   v[4] = append(v[2], v[3]);                            /* ||| */
   v[5] = append(v[1], v[4]);                            /* ||| */
   v[6] = append(v[0], v[5]);                            /* ||| */
   v[7] = maximum(v[6]);              /* h x     = [maximum x] */

   freemem_result(v[6]);
   alignment[i][j] = (*v[7]).value;
   freemem_result(v[7]);
};

void mainloop()
{
   int i; int j;

   for (j=0; j<=n; j++)
      for (i=j; i>=0; i--)
         calc_alignment(i, j);
   printAxiom(alignment[0][n]);
};
```

Fig. 11. C-Code produced by the ADP compiler for grammar `globsim` with algebra `wgap`

Compared to the classic description of DP algorithms, formulated exclusively by matrix recurrences, we have achieved several improvements:

An ADP specification is *more abstract* than the traditional recurrences. Separation between search space construction and evaluation is perfect. Tree grammars and evaluation algebras can be combined in a modular way, and the relationships between problem variants can be explained clearly. With a little experience, it becomes easy to judge whether a new problem variation affects the algebra, the grammar, or both.

The ADP specification is also *more complete*: DP algorithms in the literature often claim to be parametric with respect to the scoring function, while the initialization equations are considered part of the search algorithm [DEKM98]. In ADP, it becomes clear that initialization semantically is the evaluation of empty candidates, and is specified within the algebra.

Dynamic programmers have discovered many tricks that improve the efficiency of a DP algorithm. In the ADP framework, many such *tricks turn into techniques*, which can be formalized as transformation schemes, taught, and reused. This aspect is elaborated in [GM02].

Our formalization of Bellman's principle is *more general* than commonly seen. Objectives like complete enumeration or statistical evaluation of the search space now fall under the framework. If maximization is the objective, our criterion implies Morin's formalization (strict monotonicity) [Mor82] as a special case.

The ADP specification is *more reliable*. The absence of subscripts excludes a large class of errors that are traditionally hard to find. Furthermore, the functional language prototype allows systematic testing of alternative algorithmic ideas, and validation of the ultimate imperative program code.

6.2 Scope and limitations of ADP

The scope of a programming method can hardly be defined formally. We think that ADP as presented here is applicable to all DP problems over sequential data. ("Sequential" here does not mean we are restricted to problems on textual strings, as witnessed by our example of optimal matrix chain multiplication.) This claim of scope results from the observation that our initial idea - computing symbolically the formulas that evaluate to answers - is universally applicable.

This does not mean that our simple ADP notation is sufficient to express all the ingenuity that practitioners of dynamic programming may exhibit. Yet, many ideas that we have seen integrate smoothly into the ADP framework. Some algorithms compute different types of informations from different subproblems. This leads to many-sorted evaluation algebras, with one objective function per sort, but otherwise no change. More difficult are the recurrences that use precomputed information to reduce the number of nested for-loops. In the functional prototype, this can be accommodated by providing combinators with extra arguments, but the resulting programs can no longer be interpreted as a regular tree grammar.

6.3 Future work

Moving on from sequences to more structured data like trees or two dimensional images, the new technical problem is to provide a suitable tabulation method. To this end, we currently study local similarity problems on trees.

Another interesting extension in the realm of sequential data is to consider language intersection and complement. We may allow productions with *and* and *butnot* operators, written $u \to v$ &&& z and $u \to v$ \\\ z. The former means that an input string can be reduced to u if it can be reduced to both v and z. The latter means that it can be reduced to u if it can be reduced to v, but not to z. While this clearly leads out of the realm of context free yield languages, it is easy to implement in the parser, and quite useful in describing complicated pattern matching applications.

Returning to the ADP method as presented here, we end with pointing out an open problem that can be studied systematically now for the first time. In many DP applications, particularly in biosequence analysis where data size is large, space rather than time is the limiting resource. Algorithm designers therefore try to minimize the number of tables used by the algorithm. With ADP, we can first describe the algorithm by the yield grammar, and then decide which parsers for the nonterminals of the grammar must be tabulated to prevent combinatorial explosion, and which may be implemented as functions or inline code. These decisions are not independent. If one parser is tabulated, another one need not be. Given the grammar, is there an algorithm to determine the minimal number of required tables?

7 Acknowledgements

We are grateful to Marc Rehmsmeier for a careful reading of this manuscript.

References

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, USA, 1983.

[BB88] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice.* Prentice-Hall, 1988.

[BD62] R. Bellman and S.E. Dreyfus. *Applied Dynamic Programming.* Princeton University Press, 1962.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[BM93] R. S. Bird and O. de Moor. From dynamic programming to greedy algorithms. In B. Moeller, editor, *State-of-the-Art Seminar on Formal Program Development.* Springer LNCS 755, 1993.

[Bra69] W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Cur97] S. Curtis. Dynamic programming: A different perspective. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 1–23. Chapman & Hall, London, U.K., 1997.

[DEKM98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

[EG01] D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. In *German Conference on Bioinformatics*, 2001.

[Gie00] R. Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. *Bioinformatics*, 16:665–677, 2000.

[GKW99] R. Giegerich, S. Kurtz, and G. F. Weiller. An algebraic dynamic programming approach to the analysis of recombinant DNA sequences. In *Proc. of the First Workshop on Algorithmic Aspects of Advanced Programming Languages*, pages 77–88, 1999.

[GM02] R. Giegerich and C. Meyer. Algebraic dynamic programming. In *9th International Conference on Algebraic Methodology And Software Technology (AMAST)*, 2002. To appear.

[Got82] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[GS88] R. Giegerich and K. Schmal. Code selection techniques: Pattern matching, tree parsing and inversion of derivors. In *Proc. European Symposium on Programming 1988*, pages 247–268. Springer LNCS 300, 1988.

[GS02] R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative paradigm. In E.A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. Springer LNCS 2386, 2002.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[Hut92] G. Hutton. Higher order functions for parsing. *Journal of Functional Programming*, 3(2):323–343, 1992.

[Meh84] K. Mehlhorn. *Data structures and algorithms*. Springer Verlag, 1984.

[MG02] C. Meyer and R. Giegerich. Matching and Significance Evaluation of Combined Sequence-Structure Motifs in RNA. *Z.Phys.Chem.*, 216:193–216, 2002.

[Mit64] L. Mitten. Composition principles for the synthesis of optimal multi-stage processes. *Operations Research*, 12:610–619, 1964.

[Moo99] O. de Moor. Dynamic Programming as a Software Component. In M. Mastorakis, editor, *Proceedings of CSCC, July 4-8, Athens*. WSES Press, 1999.

[Mor82] T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[Sed89] R. Sedgewick. *Algorithms*. Addison-Wesley, 2nd edition, 1989.

[SW81] T. F. Smith and M. S. Waterman. The identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[ZS84] M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.