Pair Evaluation Algebras in Dynamic Programming

Robert Giegerich and Peter Steffen

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

Abstract. Dynamic programming solves combinatorial optimization problems by recursive decomposition and tabulation of intermediate results. The recently developed discipline of algebraic dynamic programming (ADP) helps to make program development and implementation in nontrivial applications much more effective. It raises dynamic programming to a declarative level of abstraction, separates the search space definition from its evaluation, and thus yields more reliable and versatile algorithms than the traditional dynamic programming recurrences. Here we extend this discipline by a pairing operation on evaluation algebras, whose clue lies with an asymmetric combination of two different optimization objectives. This leads to a surprising variety of applications without additional programming effort.

1 Introduction

1.1 Motivation

Dynamic Programming is an elementary and widely used programming technique. Introductory textbooks on algorithms usually contain a section devoted to dynamic programming, where simple problems like matrix chain multiplication, polygon triangulation or string comparison are commonly used for the exposition. This programming technique is mainly taught by example. Once designed, all dynamic programming algorithms look kind of similar: They are cast in terms of recurrences between table entries that store solutions to intermediate problems, from which the overall solution is constructed via a more or less sophisticated case analysis. However, the simplicity of these small programming examples is deceiving, as this style of programming provides no abstraction mechanisms, and hence it does not scale up well to more sophisticated problems.

In biological sequence analysis, for example, dynamic programming algorithms are used on a great variety of problems, such as protein homology search, gene structure prediction, motif search, analysis of repetitive genomic elements, RNA secondary structure prediction, or interpretation of data from mass spectrometry [6, 2]. The higher sophistication of these problems is reflected in a large number of recurrences – sometimes filling several pages – using more complicated case distinctions, numerous tables and elaborate scoring schemes.

An *algebraic* style of *dynamic programming* (ADP) has recently been introduced, which allows to formulate dynamic programming algorithms over sequence data on a more convenient level of abstraction [4, 5]. In the ADP approach, the issue of scoring is cast in the form of an *evaluation algebra*, the logical problem decomposition is expressed as a *yield grammar*. Together they constitute a declarative, and notably subscript-free problem specification that transparently reflects the design considerations. Written in a suitable notation, these specifications can be implemented automatically – often more efficiently and always more reliably than hand-programmed DP recurrences.

In this paper, we extend the ADP discipline by one further operator, a product construction on evaluation algebras. Its clue lies with an asymmetric, nested definition of the product of two objective functions, which allows to optimize according to a primary and a secondary objective. Beyond this, when using some non-optimizing algebras, it leads to an unexpected variety of applications, such as backtracing, multiplicity of answers, ambiguity checking, and more.

1.2 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 *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 $o_\mathcal{I}$ for each operator o. Each $o_\mathcal{I}$ has type $o_\mathcal{I} : (s_1)_\mathcal{I} ... (s_{k_o})_\mathcal{I} \rightarrow \mathcal{S}_\mathcal{I}$ where $\mathcal{A}_\mathcal{I} = \mathcal{A}$.

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.

Trees and tree patterns. 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 from Σ 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 [1], with some algebraic flavoring added such that they describe term languages over some signature Σ and some alphabet $\mathcal{A}$.

2 Algebraic Dynamic Programming in a nutshell

ADP is a domain specific language for dynamic programming over sequence data. In ADP, a dynamic programming algorithm is specified by a yield grammar and an evaluation algebra. The grammar defines the search space as a term language, the algebra the scoring of solution candidates. Their interface is a common signature.

Our introduction here must be very condensed. For a complete presentation, including the programming methodology that comes with ADP, the reader is referred to [5] and to the ADP website at
`http://bibiserv.techfak.uni-bielefeld.de/adp/`.

Definition 1. *(Tree grammar over Σ and $\mathcal{A}$)*
A regular tree grammar $\mathcal{G} = (V, Z, P)$ over Σ and $\mathcal{A}$ is given by

- *a set V of nonterminal symbols,*
- *a designated nonterminal symbol Z, called the axiom, and*
- *a set P of productions of the form $v \rightarrow t$, where $v \in V$ and $t \in T_\Sigma(V)$. $v \rightarrow t_1|\dots|t_n$ shall denote the short form for $v \rightarrow t_1, \dots, v \rightarrow t_n$.*

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 $\mathcal{L}(\mathcal{G}) = \mathcal{L}(Z)$.

For convenience, we add a lexical level to the grammar concept, allowing strings from $\mathcal{A}^*$ in place of single symbols. $L = \{char, string, empty\}$ is the set of lexical symbols. By convention, $\mathcal{L}(char) = \mathcal{A}$, $\mathcal{L}(string) = \mathcal{A}^*$, and $\mathcal{L}(empty) = \{\varepsilon\}$.

The yield function y on the trees in T_Σ is defined by $y(a) = a$ for $a \in \mathcal{A}$, and $y(f(x_1, \dots, x_n)) = y(x_1) \dots y(x_n)$, for $f \in \Sigma$ and $n \geq 0$. Note that nullary constructors by definition have yield ε, hence $y(t)$ is the string of leaf symbols from $\mathcal{A}$ in left to right order.

We shall also allow conditional productions, where a simple predicate must be satisfied by the yield string derived.

Definition 2. *(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 $\mathcal{L}(\mathcal{G}, y) = \{y(t) | t \in \mathcal{L}(\mathcal{G})\}$.*

Definition 3. *(Yield parsing) Given a yield grammar $(\mathcal{G}, y)$ over $\mathcal{A}$ and a sequence $w \in \mathcal{A}^*$, the yield parsing problem is to find $P_\mathcal{G}(w) = \{t \in \mathcal{L}(\mathcal{G}) | y(t) = w\}$.*

Note that the input string w is "parsed" into trees $t \in \mathcal{L}(\mathcal{G})$, each of which in turn has a tree parse according to the tree grammar $\mathcal{G}$. These tree parses must exist – they ensure that each candidate t corresponds to a proper problem decomposition – but otherwise, they are irrelevant and will play no part in the sequel. The candidate trees t, however, represent the search space spanned by a particular problem instance, and will be subject to scoring and choice under our optimization objective.

Definition 4. *(Evaluation algebra) Let Σ be a signature with sort symbol Ans. A Σ-evaluation algebra $\mathcal{I}$ is a Σ-algebra augmented with an objective function $h_\mathcal{I} : \mathcal{L}(Ans_\mathcal{I}) \rightarrow \mathcal{L}(Ans_\mathcal{I})$, where $\mathcal{L}(Ans_\mathcal{I})$ denotes the set of lists with elements from $Ans_\mathcal{I}$.*

Given that yield parsing constructs the search space for a given input, all that is left to do is to evaluate the candidates in a given algebra, and make our choice via the objective function $h_{\mathcal{I}}$.

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* $\mathcal{I}$ *with objective function* $h_{\mathcal{I}}$*.*
- *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 with respect to* $|w|$*.*

Bellman's Principle[1], when satisfied, allows the following implementation of tree parsing: As the trees that constitute the search space are constructed in a bottom up fashion, rather than building them explicitly as terms in T_{Σ}, for each constructor C the evaluation function $C_{\mathcal{I}}$ is called. Thus, the tree parser computes not trees, but answer values. To reduce their number (and thus to avoid exponential explosion) the objective function may be applied at an intermediate step where a list of alternative answers has been computed. Due to Bellman's Principle, the recursive intermediate applications of the choice function do not affect the final result.

In this paper, the reader is asked to take it for granted that the tree parsing sketched here can be implemented efficiently. ADP comes with an ASCII notation for yield grammars and evaluation algebras, which is either embedded in Haskell or directly translated to C. In the examples at the aforementioned website we explicitly annotate productions to the results of which the choice function is to be applied, but for our presentation here the reader may assume that it is applied wherever appropriate.

3 RNA secondary structure prediction

We need an example with a certain sophistication to illustrate well the variety of applications we have in mind. The following is a simplified version of the RNA structure analysis problem that plays an important role in biosequence analysis.

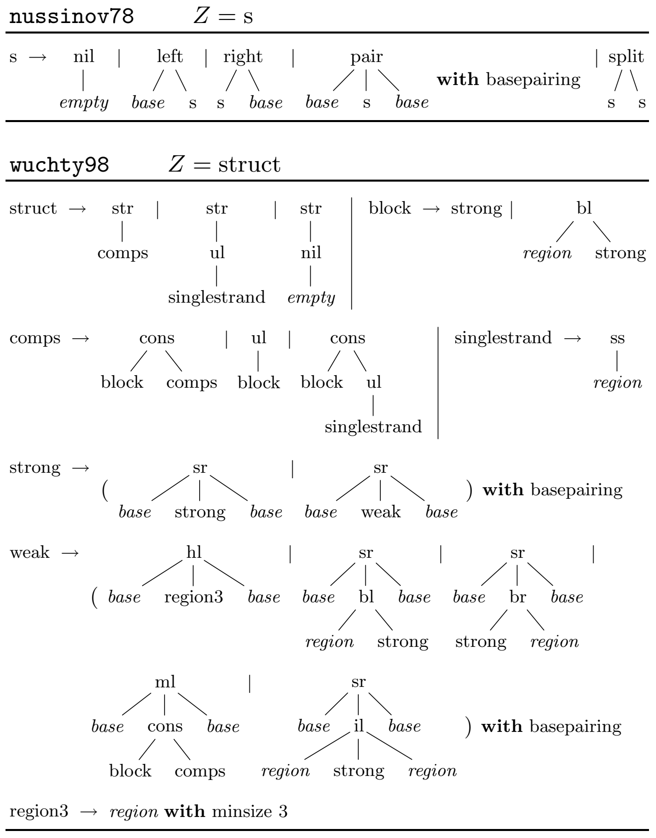
All genetic information in living organisms is encoded in long chain molecules. DNA is the storage form of genetic information, its shape being the double helix discovered by Watson and Crick. Mathematically, the human genome is a string of length 3×10^9 over a four letter alphabet. RNA is the active form of genetic information. It is transcribed from a segment of the DNA as a chain of *bases* or *nucleotides* A, C, G and U, denoting Adenine, Cytosine, Guanine, and Uracil. Some bases can form base pairs by hydrogen bonds: G–C, A–U and also G–U. RNA is typically single stranded, and by folding back onto itself, it forms the structure essential for its biological function. Structure formation is driven by

[1] See [5] for the formulation of Bellman's Principle in the ADP framework.

the forces of hydrogen bonding between base pairs, and energetically favorable stacking of base pairs in a helical pattern similar to DNA. While today the prediction of RNA 3D structure is inaccessible to computational methods, its *secondary structure*, given by the set of paired bases can be predicted quite reliably. Mathematically, RNA secondary structures are approximate palindromes that can be nested recursively.

In RNA structure prediction, our input sequence is a string over $\{A, C, G, U\}$. The lexical symbols *char* and *string* are renamed to *base* and *region*. The predicate *basepairing* checks whether the two bases mentioned in a production can actually form a base pair.

The first approach to RNA structure prediction was based on the idea of maximizing the number of base pairs [8]. Figure 1 (top) shows the grammar `nussinov78` which implements the algorithm of [8], with the evaluation algebra designed for maximizing the number of base pairs.



nussinov78 Z = s
s → nil | left | right | pair with basepairing | split
empty base s s base base s base s s
wuchty98 Z = struct
struct → str | str | str
comps ul nil singlestrand empty
block → strong | bl
region strong
comps → cons | ul | cons
block comps block block ul singlestrand
singlestrand → ss
region
strong → (sr | sr) with basepairing
base strong base base weak base
weak → (hl | sr | sr |
base region3 base base bl base base br base
region strong strong region
ml | sr) with basepairing
base cons base base il base
block comps region strong region
region3 → region with minsize 3

Fig. 1. Yield grammars `nussinov78` and `wuchty98`. Terminal symbols in italics.

Note that the case analysis in the Nussinov algorithm is redundant – even the base string "A" is assigned the two structures `Left('A', Nil)` and `Right(Nil, 'A')`, which actually denote the same shape.

Base pair maximization ignores the favorable energy contributions from base pair stacking, as well as the unfavorable contributions from loops. A non-redundant algorithm based on energy minimization was presented by Wuchty et al. [9]. Figure 1 (bottom) shows the grammar `wuchty98`. Here the signature has 8 operators, each one modeling a particular structure element, plus the list constructors (`nil, ul, cons`) to collect sequences of components in a unique way. This grammar, because of its non-redundancy, can also be used to study combinatorics, such as the expected number of feasible structures of a particular sequence of length n.

$Ans_{\texttt{enum}} = T_{\Sigma}$

```
enum = (str, ..., h) where
str(s)                = Str s
ss((i,j))             = Ss (i,j)
hl(a,(i,j),b)         = Hl a (i,j) b
sr(a,s,b)             = Sr a s b
bl((i,j),s)           = Bl (i,j) s
br(s,(i,j))           = Br s (i,j)
il((i,j),s,(i',j')) = Il (i,j) s (i',j')
ml(a,s,b)             = Ml a s b
nil((i,j))            = Nil (i,j)
cons(s,s')            = Cons s s'
ul(s)                 = Ul s
h([s_1,...,s_r])      = [s_1,...,s_r]
```

$Ans_{\texttt{pretty}} = \{(,),.\}^*$

```
pretty = (str, ..., h) where
str(s)                = s
ss((i,j))             = dots(j-i)
hl(a,(i,j),b)         = "("++dots(j-i)++")"
sr(a,s,b)             = "("++s++")"
bl((i,j),s)           = dots(j-i)++s
br(s,(i,j))           = s++dots(j-i)
il((i,j),s,(i',j')) = dots(j-i)++s++
                        dots(j'-i')
ml(a,s,b)             = "("++s++")"
nil((i,j))            = ""
cons(s,s')            = s++s'
ul(s)                 = s
h([s_1,...,s_r])      = [s_1,...,s_r]
```

$Ans_{\texttt{bpmax}} = \mathbb{N}$

```
bpmax = (str, ..., h) where
str(s)                = s
ss((i,j))             = 0
hl(a,(i,j),b)         = 1
sr(a,s,b)             = s + 1
bl((i,j),s)           = s
br(s,(i,j))           = s
il((i,j),s,(i',j')) = s
ml(a,s,b)             = s + 1
nil((i,j))            = 0
cons(s,s')            = s + s'
ul(s)                 = s
h([s_1,...,s_r])      = [ max_{1≤i≤r} s_i]
```

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

```
count = (str, ..., h) where
str(s)                = s
ss((i,j))             = 1
hl(a,(i,j),b)         = 1
sr(a,s,b)             = s
bl((i,j),s)           = s
br(s,(i,j))           = s
il((i,j),s,(i',j')) = s
ml(a,s,b)             = s
nil((i,j))            = 1
cons(s,s')            = s * s'
ul(s)                 = s
h([s_1,...,s_r])      = [s_1 + ··· + s_r]
```

Fig. 2. Four evaluation algebras for grammar `wuchty98`. Arguments `a` and `b` denote bases, `(i,j)` represents the input subword $x_{i+1} \ldots x_j$, and `s` denotes answer values. Function `dots(r)` in algebra `pretty` yields a string of `r` dots (`'.'`).

This algorithm is widely used for structure prediction via energy minimization. Unfortunately, the thermodynamic model is too elaborate to be presented here, and we will stick with base pair maximization as our optimization objective for the sake of this presentation. Figure 2 shows four evaluation algebras that we will use with grammar `wuchty98`. We illustrate their use via the following examples, where `g(a,x)` denotes the application of grammar `g` and algebra `a` to input `x`, as defined in Definition 5. Appendix A shows all results for an example sequence.

`wuchty98(enum,x)`: the enumeration algebra `enum` yields unevaluated terms. Since the choice function is identity, this call enumerates all candidates in the search space spanned by `x`. This is mainly used in program debugging, as it visualizes the search space actually traversed by our program.

`wuchty98(pretty,x)`: the pretty-printing algebra `pretty` yields a string representation of the same structures as the above, but in the widely used notation `"..(((...)).)"`, where pairing bases are indicated by matching brackets.

`wuchty98(bpmax,x)`: the base pair maximization algebra is `bpmax`, such that this call yields the maximal number of base pairs that a structure for `x` can attain. Here the choice function is maximization, and it can be easily shown to satisfy Bellman's Principle. Similarly for grammar `nussinov78`.

`wuchty98(count,x)`: the counting algebra is `count`. Its choice function is summation, and $t_{count} = 1$ for all candidates t. However, the evaluation functions are written in such a way that they satisfy Bellman's Principle. Thus, `[length(wuchty98(enum,x))] == wuchty98(count,x)`, where the righthand side is polynomial to compute, while the lefthand side typically is exponential due to the large number of answers.

4 Pair evaluation algebras

We now create an algebra of evaluation algebras by introducing a product operation `***` that joins two evaluation algebras into a single one.

Definition 6. *(Product operation on evaluation algebras) Let M and N be evaluation algebras over Σ. Their product M***N is an evaluation algebra over Σ and has the functions $f_{M,N}((m_1, n_1)...(m_k, n_k)) = (f_M(m_1, ..., m_k), f_N(n_1, ..., n_k))$ for each f in Σ, and the choice function $h_{M,N}([(m_1, n_1)...(m_k, n_k)]) = [(l, r)|l \in L, r \in h_N([r|(l, r) \leftarrow [(m_1, n_1)...(m_k, n_k)], l \in L])]$ where $L = h_M([m_1, ..., m_k])$.*

Above, $\in$ denotes set membership and hence ignores duplicates[2], while $\leftarrow$ denotes list membership and respects duplicates. Our first observation is that this definition preserves identity and ordering:

Theorem 1. *(1) For any algebras M and N, and answer list x, $(id_M{*}{*}{*}id_N)(x)$ is a permutation of x. (2) If h_M and h_N minimize wrt. some order relations $\leq_M$ and $\leq_N$, then $h_{M,N}$ minimizes wrt. the lexicographic ordering $(\leq_M, \leq_N)$. (3) If both M and N minimize and satisfy Bellman's Principle, then so does M***N.*

Proof. (1) According to Def. 6, the elements of x are merely re-grouped according to their first component. For this to hold, it is essential that duplicate entries in the first component are ignored. (2) follows directly from Def. 6. (3) In the case of minimization, Bellman's Principle is equivalent to (strict) monotonicity of f_M and f_N with respect to $\leq_M$ and $\leq_N$, and this carries over to the combined functions (trivially) and the lexicographic ordering (because of (2)). $\square$

In the above proof, *strict* monotonicity is required only if we ask for multiple optimal, or the k best, solutions rather than a single optimal one [7].

[2] This may require some extra effort in the implementation, but when a choice function does not produce duplicates anyway, it comes for free.

Theorem 1 essentially says that `***` behaves as expected in the case of optimizing evaluation algebras. This is very useful, but not too surprising. The interesting situations are when `***` is used with algebras that do not do optimization, like `enum`, `count`, and `pretty`. Applications of pair algebras are subject to the following

Proof scheme: *The declarative semantics of (i.e. the problem specified by)* $\mathcal{G}(M\texttt{***}N, x)$ *is given by Definition 5. Its operational semantics (the tabulating yield parser) is correct only if* $M\texttt{***}N$ *satisfies Bellman's Principle. This requires an individual proof unless covered by Theorem 1.*

That a proof is required in general is witnessed by the fact that, for example, `wuchty98(count***count,x)` delivers no meaningful result.

With this in mind, we now turn to applications of pair algebras. Appendix A shows all results for an example RNA sequence.

Application 1: Backtracing and co-optimal solutions Often, we want not only the optimal answer value, but also a candidate which achieves the optimum. We may ask if there are several such candidates. If yes, we may want to see them all, maybe even including some near-optimal candidates. They can be retrieved if we store a table of intermediate answers and backtrace through the optimizing decisions made. This backtracing can become quite intricate to program if we ask for more than one candidate. There are simpler ways to answer these questions:

`wuchty98(bpmax***count,x)` computes the optimal number of base pairs, together with the number of candidate structures which achieve it.

`wuchty98(bpmax***enum,x)` computes the optimal number of base pairs, together with all structures for `x` that achieve this maximum, in their representation as terms from T_Σ.

`wuchty98(bpmax***pretty,x)` does the same as the previous call, producing the string representation of structures.

It is a nontrivial consequence of Definition 6 that the above pair algebras in fact give multiple co-optimal solutions. Should only a single one be desired, we would use `enum` or `pretty` with a choice function h that retains only one (arbitrary) element. Note that our replacement of backtracing by a "forward" computation does not affect asymptotic efficiency.

Application 2: Testing ambiguity Dynamic programming algorithms can often be written in a simpler way if we do not care whether the same solution is considered many times during the optimization. This does not affect the overall optimum. A dynamic programming algorithm is then called redundant or ambiguous. In such a case, the computation of a list of near-optimal solutions is cumbersome, as it contains duplicates whose number often has an exponential growth pattern. Also, search space statistics become problematic – for example, the counting algebra speaks about the algorithm rather than the problem space, as it counts considered, but not necessarily distinct solutions. Yield grammars with a suitable probabilistic evaluation algebra implement stochastic context free grammars. The frequently used statistical scoring schemes, when trying to find the answer of maximal probability (the Viterbi algorithm, cf. [2]), are fooled by the presence of redundant solutions. In principle, it is clear how to control ambiguity [3]. One needs to show unambiguity of the *tree* grammar[3] in the language theoretic sense, and the existence of an injective mapping from T_{Σ} to a canonical model of the search space. However, the proofs involved are not trivial. On the practical side, one would like to implement a check for ambiguity in the implementation of the ADP approach, but this is rendered futile by the following observation:

Theorem 2. *Non-redundancy in dynamic programming is formally undecidable.*

Proof. For lack of space, we can only sketch the idea of the proof. Ambiguity of context free language is a well-known undecidable problem. For an arbitrary context free grammar, we may construct an ADP problem where the context free language serves as the canonical model, and show that the language is unambiguous if and only if the ADP problem is non-redundant. $\square$

Given this situation, we turn to the possibility of testing for (non-)redundancy. The required homomorphism from the search space to the canonical model may be coded as another evaluation algebra. This is, for example, the case with `pretty`. A pragmatic approach to this question of ambiguity is then to test

`wuchty98(pretty***count,x)` on a number of inputs `x`. If any count larger than 1 shows up in the results, we have found a case of ambiguity. Clearly, this test can be automated.

Application 3: Classification of candidates A `shape` algebra is a version of `pretty` that maps structures onto more abstract shapes. This allows to analyze the number of possible shapes, the size of their membership, and the (near-)optimality of members. Let `bpmax(k)` be `bpmax` with a choice function that retains the best `k` answers (without duplicates).

`wuchty98(shape***count,x)` computes all the shapes in the search space spanned by `x`, and the number of structures that map onto each shape.

`wuchty98(bpmax(k)***shape,x)` computes the best `k` base pair scores, together with their candidate's shapes.

`wuchty98(bpmax(k)***(shape***count),x)` computes base pairs and shapes as above, plus the number of structures that achieve this number of base pairs in the given shape.

`wuchty98(shape***bpmax,x)` computes for each shape the maximum number of base pairs among all structures of this shape.

5 Conclusion

We hope to have demonstrated that the evaluation algebra product as introduced here adds a significant amount of flexibility to dynamic programming. The mathematical properties of `***` are not yet fully explored. Moreover, Definition 6 is not without alternatives. One might consider to make in M***N the

[3] Not the *yield* grammar – it is always ambiguous, else we did not have an optimization problem.

results of M available to the choice function h_N. This leads to parameterized evaluation algebras and is a challenging subject for further study.

A Results for examples

The following table shows the application of grammar `wuchty98` with different algebras on input `x = cgggauaccacu`.

Algebra	Result
`enum`	`[Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u'))),Str (Ul (Bl (0,2) (Sr ....]`
`pretty`	`[".((.......))","..((......))",".((....))...", "..((...))...","............"]`
`bpmax`	`[2]`
`count`	`[5]`
`count***count`	`[(1,1)]`
`bpmax***count`	`[(2,4)]`
`bpmax***enum`	`[(2,Str (Ul (Bl (0,1) (Sr 'g' (Hl 'g' (3,10) 'c') 'u')))),(2,Str (Ul (Bl (0,2) ....]`
`bpmax***pretty`	`[(2,".((.......))"),(2,"..((......))"), (2,".((....))..."),(2,"..((...))...")]`
`pretty***count`	`[(".((.......))",1),("..((......))",1), (".((....))...",1),("..((...))...",1), ("............",1)]`
`shape***count`	`[("_[_]",2),("_[_]_",2),("_",1)]`
`bpmax(5)***shape`	`[(2,"_[_]"),(2,"_[_]_"),(0,"_")]`
`bpmax(5)***(shape***count)`	`[(2,("_[_]",2)),(2,("_[_]_",2)),(0,("_",1))]`
`shape***bpmax`	`[("_[_]",2),("_[_]_",2),("_",0)]`

References

1. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
2. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.
3. R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer LNCS 1848, 2000.
4. R. Giegerich. A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, 16:665–677, 2000.
5. R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, 2004.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.
7. T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.
8. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.
9. S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1999.