Explaining and Controlling Ambiguity in Dynamic Programming

Robert Giegerich

Faculty of Technology, Bielefeld University
33501 Bielefeld, Germany
robert@techfak.uni-bielefeld.de

Abstract. Ambiguity in dynamic programming arises from two independent sources, the non-uniqueness of optimal solutions and the particular recursion scheme by which the search space is evaluated. Ambiguity, unless explicitly considered, leads to unnecessarily complicated, inflexible, and sometimes even incorrect dynamic programming algorithms. Building upon the recently developed algebraic approach to dynamic programming, we formalize the notions of ambiguity and canonicity. We argue that the use of canonical yield grammars leads to transparent and versatile dynamic programming algorithms. They provide a master copy of recurrences, that can solve all DP problems in a well-defined domain. We demonstrate the advantages of such a systematic approach using problems from the areas of RNA folding and pairwise sequence comparison.

1 Motivation and Overview

1.1 Ambiguity Issues in Dynamic Programming

Dynamic Programming (DP) solves combinatorial optimization problems. It is a classical programming technique throughout computer science [3], and plays a dominant role in computational biology [4, 10]. A typical DP problem spawns a search space of potential solutions in a recursive fashion, from which the final answer is selected according to some criterion of optimality. If an optimal solution can be derived recursively from optimal solutions of subproblems [1], DP can evaluate a search space of exponential size in polynomial time and space.

Sources of Ambiguity. By ambiguity in dynamic programming we refer to the following facts which complicate the understanding and use of DP algorithms:

– *Co-optimal and near-optimal solutions:* It is well known that the "optimal" solution found by a DP algorithm normally is not unique, and

there may be relevant near-optimal solutions. A single, "optimal" answer is often unsatisfactory. Considerable work has been devoted to this problem, producing algorithms providing near-optimal [15, 17] and parametric [11] solutions.
- *Duplicate solutions:* While there is a general technique to enumerate all solutions to a DP problem (possibly up to some threshold value) [21, 22], such enumeration is hampered by the fact that the algorithm may produce the same solution several times – and in fact, this may lead to combinatorial explosion of redundancy. Heuristic enumeration techniques, and post-facto filtering as a safeguard against duplicate answers are employed e.g. in [23].
- *(Non-)canonical solutions:* Often, the search space exhibits additional redundancy in terms of solutions that are represented differently, but are equivalent from a more semantic point of view. Canonization is important in evaluating statistical significance [14], and also in reducing redundancy among near-optimal solutions.

Ambiguity examples. Strings `aaaccttaa` and `aaaggttaa` are aligned below. Alignments (1) and (2) are equivalent under most scoring schemes, while (3) may even be considered a mal-formed alignment, as it shows two deletions separated by an insertion.

```
aaacc--ttaa      aaa--ccttaa      aaac--cttaa
aaa--ggttaa      aaagg--ttaa      aaa-gg-ttaa
    (1)              (2)              (3)
```

In the RNA folding domain, each DP algorithm seems to be a one-trick pony. Different recurrences have been developed for counting or estimating the number of various classes of feasible structures of a sequence of given length [12], for structure enumeration [22], energy minimization [25], and base pair maximization [19]. Again, enumerating co-optimal answers will produce duplicates in the latter two cases. In [4](p. 272) a probabilistic scoring scheme is suggested to find the *most likely* RNA secondary structure – this is a valid idea, but will work correctly only if the underlying recursion scheme considers each feasible structure exactly once.

Our main contributions. The recently developed technique of *algebraic* dynamic programming (ADP), summarized in Section 2, uses yield grammars and evaluation algebras to specify DP algorithms on a rather high level of abstraction. DP algorithms formulated this way *can* have all the ambiguity problems illustrated above. However, the ADP framework also helps to analyse and avoid these problems.

1. In this article, we devise a formal framework to explain and reason about ambiguity in its various forms.
2. Introducing *canonical* yield grammars, we show how to construct a "master copy" of a DP algorithm for a given problem class. This single set of recurrences can correctly and efficiently perform all analyses in this problem class, including optimization, complete enumeration, sampling and statistics.
3. Re-use of the master recurrences for manifold analyses provides a major advantage from a software-engineering point of view, as it enhances not only programming economy, but also program reliability.

2 A Short Review of Algebraic Dynamic Programming

ADP introduces a *conceptual* splitting of a DP algorithm into a recognition and an evaluation phase. A *yield grammar* is used to specify the recognition phase (i. e. the search space of the optimization problem). A particular parsing technique turns the grammar directly into an efficient dynamic programming scheme. The evaluation phase is specified by an *evaluation algebra*, and each grammar can be combined with a variety of algebras to solve different problems over the same data domain, for which heretofore DP recurrences had to be developed independently.

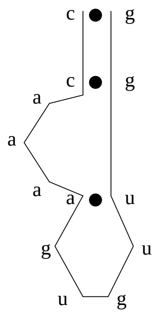
2.1 Basic Notions

Let $\mathcal{A}$ be an alphabet. $\mathcal{A}^*$ denotes the set of finite strings over $\mathcal{A}$, and ++ denotes string concatenation. Throughout this article, $x, y \in \mathcal{A}^*$ denote input strings to various problems, and $|x| = n$. A subword is indicated by its boundaries – $x_{(i,j)}$ denotes $x_{i+1}...x_j$.

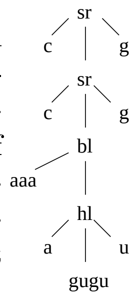
An algebra is a set of values and a family of functions over this set. We shall allow that these functions take additional arguments from $\mathcal{A}^*$. An algebraic data type $\mathcal{T}$ is a type name and a family of typed function symbols, also called operators. It introduces a language of (well-typed) formulas, called the term algebra. An algebra that provides a function for each operator in $\mathcal{T}$ is a $\mathcal{T}$-algebra. The interpretation $t_{\mathcal{I}}$ of a term t in a $\mathcal{T}$-algebra $\mathcal{I}$ is obtained by substituting the corresponding function of the algebra for each operator. Thus, $t_{\mathcal{I}}$ evaluates to a value in the base set of $\mathcal{I}$.

Terms as syntactic objects can be equivalently seen as trees, where each operator is a node, which has its subterms as subtrees. Tree grammars over $\mathcal{T}$ describe specific subsets of the term algebra. A regular tree grammar over $\mathcal{T}$ [2, 7] has a set of nonterminal symbols, a designated axiom symbol, and productions of the form $X \to t$ where X is a nonterminal symbol, and t is a tree pattern, i.e. a tree over $\mathcal{T}$ which may have nonterminals in leaf positions.

2.2 ADP – the Declarative Level



Simple hairpin structure



Hairpin in term representation

In the sequel, we assume that $\mathcal{T}$ is some fixed data type, and $\mathcal{A}$ a fixed alphabet. As a running example, let $\mathcal{A} = \{$`a, c, g, u`$\}$, representing the four bases in RNA, and let $\mathcal{T}$ consist of the operators `sr, hl, bl, br, il`, representing structural elements in RNA: stacking regions, hairpin loops, bulges on the left and right side, and internal loops. Feasible base pairs are `a - u, g - c, g - u`. The little hairpin denoted by the term

```
s = sr 'c' (sr 'c' (bl "aaa" (hl 'a' "gugu" 'u')) 'g') 'g'
```

is one of many possible 2D structures of the RNA sequence `ccaaaaguguugg`.

Definition 1 (Evaluation Algebra). *An evaluation algebra is a $\mathcal{T}$-algebra augmented by a choice function h. If l is a list of values of the algebra's value set, then $h(l)$ is a sublist thereof. We require h to be polynomial in $|l|$. Furthermore, h is called reductive if $|h(l)|$ is bounded by a constant.*

The choice function is a standard part of our evaluation algebras because we shall deal with optimization problems. Typically, h will be minimization or maximization, but, as we shall see, it may also be used for counting, estimation, or some other kind of synopsis. Non-reductive choice functions are used e.g. for complete enumeration. The hairpin `s` evaluates to 3 in the *basepair algebra*, and (naturally) to 1 in the *counting algebra*:

```
basepair_alg = (sr,hl,bl,br,il,h)
 where sr _ x _ = x+1
       hl _ x _ = 1
       bl _ x   = x
       br   x _ = x
       il _ x _ = x
       h        = maximum
```

```
counting_alg = (sr,hl,bl,br,il,h)
 where sr _ x _ = x
       hl _ x _ = 1
       bl _ x   = x
       br   x _ = x
       il _ x _ = x
       h        = sum
```

Definition 2 (Yield Grammar). *A yield grammar $(\mathcal{G}, y)$ is given by*

– *an underlying algebraic datatype $\mathcal{T}$, and alphabet $\mathcal{A}$,*

– *a homomorphism $y : \mathcal{T} \to \mathcal{A}^*$ called the yield function,*
– *a regular tree grammar $\mathcal{G}$ over $\mathcal{T}$.*

$\mathcal{L}(\mathcal{G})$ denotes the tree language derived from the axiom, and $\mathcal{Y}(\mathcal{G}) := \{y(t) | \ t \in \mathcal{L}(\mathcal{G})\}$ is the yield language of $\mathcal{G}$.

The homomorphism condition means that $y(Cx_1...x_n) = y(x_1)$++...++$y(x_n)$ for any operator C of $\mathcal{T}$. For the hairpin `s`, we have $y(\texttt{s})$ = `ccaaaaguguugg`. By virtue of the homomorphism property, we may apply the yield function to the righthand sides of the productions in the tree grammar. In this way, we obtain a context free grammar $y(\mathcal{G})$ such that $\mathcal{Y}(\mathcal{G}) = \mathcal{L}(y(\mathcal{G}))$.

Definition 3 (Yield Parsing). *The yield parsing problem of $(\mathcal{G}, y)$ is to compute for a given $s \in \mathcal{A}^*$ the set of all $t \in \mathcal{T}$ such that*

$$y(t) = s.$$

Definition 4 (Algebraic Dynamic Programming). *Let $\mathcal{I}$ be a $\mathcal{T}$-algebra with a reductive choice function $h_\mathcal{I}$. Algebraic Dynamic Programming is computing for given $s \in \mathcal{A}^*$ the set of solutions*

$$h_\mathcal{I}\{t_\mathcal{I} \mid y(t) = s\} \text{ in polynomial time and space.}$$

This definition precisely describes a class of DP problems over sequences. All biosequence analysis problems we have studied so far fall under the ADP framework. Outside the realm of sequences, DP is also done over trees, dags, and graphs. It is open whether the concept of a yield grammar can be generalized to accommodate these domains. A detailed discussion of the scope of ADP is beyond the space limits of this short review.

2.3 ADP – the Notation

Once a problem has been specified by a yield grammar and an evaluation algebra, the ADP approach provides a systematic transition to an efficient DP algorithm that solves the problem. To achieve this, we introduce a notation for yield grammars that is both human readable and — executable! In ADP notation, yield grammars are written in the form

```
hairpin   = axiom struct where
   struct = open ||| closed
   open   = bl <<< region ~~~ closed |||
            br <<< closed ~~~ region |||
            il <<< region ~~~ closed ~~~ region
```

The grammar `hairpin` has axiom `struct` and further nonterminal symbols `open, closed`. `base` denotes an arbitrary base, and `region` a non-empty sequence of bases from the RNA alphabet. The grammar notation is refined further by allowing predicates and the choice function to be associated with nonterminals symbols and productions:

```
closed =
  ((hl <<< base ~~~ (region `with` minsize 3) ~~~ base           |||
    sr <<< base ~~~ (closed ||| open) ~~~ base) `with` basepair) ... h
```

This production uses two predicates: `minsize k` requires a yield of minimal length `k`, and `basepair` applies to both alternatives, requiring that the bounding bases of either closed structure form a feasible base pair. The choice function `h` is attached via the `...`-combinator, indicating that from several alternative closed structures, a selection according to `h` is imposed.

In the syntactic view of yield grammars, we interpret the operators `hl, sr,...` in the term algebra. They merely construct terms or trees representing hairpins. In this view, the choice function `h` has little use and should be assumed to be the identity function. However, in a more semantic view, we see `hl, sr,...` as functions of some evaluation algebra. Then, the "trees" generated by the grammar are actually formulas that can be evaluated. In this view, the grammar is a mechanism to generate a set of values, and it makes sense to apply the algebra's choice function to select (say) a maximal one.

2.4 ADP – the Implementation Level

We now solve the yield parsing problem. A nondeterministic, top-down parser for a context-free grammar is easily obtained by the combinator technique of [13]. This idea is adapted to yield grammars. A yield parser pN for nonterminal N takes a subword (i,j) of x as its argument and returns the set $pN(i,j) = \{t|y(t) = x_{(i,j)}\}$. Technically, it returns a list; when the list is empty, we say that the parser fails. Where the operators of $\mathcal{T}$ take strings from $\mathcal{A}^*$ as their arguments, suitable parsers must be provided.

The grammar itself is turned into a parser by defining the combinators as higher-order functions which compose complex parsers from simpler ones. For the sake of completeness, definitions are given here, but space does not allow a thorough discussion. We use list comprehension notation borrowed from the functional programming language *Haskell*.

```
  (r ||| q) (i,j)    = r(i,j) ++ q(i,j)
```

```
(f <<< q) (i,j)     = [f z | z <- q(i,j)]
(r ~~~ q) (i,j)     = [f y | k <- [i+1..j-1], f <- r(i,k), y <- q(k,j)]
(r ... h) (i,j)     = h(p(i,j))
axiom q             = q(0,n)
(r `with` w) (i,j) = if w(i,j) then r(i,j) else []
```

Note that the `axiom`- and the `with`-clause are also defined as functions applied to parsers. With these definitions, a grammar like `hairpin` is now an executable yield parser, albeit of miserable efficiency: There may be an exponential number of parses, and any subparse is constructed many times. This is alleviated by tabulating the parser functions. Let `p` be a table indexing function and `tabulated` be a tabulation function such that

```
p (tabulated f) (i,j) = f(i,j), or equivalently
p (tabulated f)       = f
```

With this convention, a grammar may be annotated for efficiency, replacing parsers by tables. Choosing to tabulate the parser for nonterminal `closed`, grammar `hairpin` now reads

```
hairpin   = axiom struct where
   struct =        open                  ||| p closed
   open   = bl <<< region   ~~~ p closed |||
            br <<< p closed ~~~ region   |||
            il <<< region   ~~~ p closed ~~~ region
   closed = tabulated (
    ((hl <<< base ~~~ (region `with` minsize 3) ~~~ base |||
      sr <<< base ~~~ (p closed ||| open) ~~~ base) `with` basepair) ... h)
```

Such annotation does not affect the meaning of the grammar, nor that of the parser. It only affects the parser's efficiency: The parser now uses dynamic programming. In general, the parser consists of a family of recursively defined tables and functions. Substituting the definitions of the combinators and the functions of a specific evaluation algebra, the annotated grammar simplifies to a set of recurrences as we traditionally see it in dynamic programming.

2.5 Two Classical DP Algorithms in ADP Notation

Zuker's Algorithm for RNA folding. Zuker and Stiegler [25] gave a DP algorithm for determining the minimal free energy structure of an RNA molecule under the nearest neighbour model. The model and the algorithm have been elaborated considerably since then, but for lack of space, we base our discussion on the original description. Evers [5] has recently reformulated Zuker's recurrences as a yield grammar $\mathcal{G}_{\texttt{zuker81}}$[1]:

[1] This example shows actually executable ADP code, and contains a few refinements not explained in Section 2. The variants of the ~~~-operator are all equivalent in

```
zuker81 algebra inp                 = axiom struct where
  (str,hl,bi,sr,bl,br,il,ol,ox,co,h) = algebra

              -- nonterminals v and w are Zuker's tables V and W.
  struct      = str <<< p w
  v           = tabulated (
      ((hairpin ||| twoedged ||| bifurcation) `with` basepair)     ... h)
  hairpin     = hl <<< base -~~ (region `with` minsize 3) ~~- base
  bifurcation = bi <<< base -~~ p w ~-~ p w ~~- base              ... h
  twoedged    = stack ||| bulgeleft ||| bulgeright ||| interior   ... h
  stack       = sr <<< base -~~            p v            ~~- base
  bulgeleft   = bl <<< base -~~ region ~-~ p v            ~~- base
  bulgeright  = br <<< base -~~            p v ~-~ region ~~- base
  interior    = il <<< base -~~ region ~-~ p v ~-~ region ~~- base

  w = tabulated ( openleft ||| openright ||| p v ||| connected    ... h)
  openleft  = ol <<<   base -~~ p w
  openright = ox <<< p w    ~~-   base
  connected = co <<< p w    ~-~ p w ... h
```

This grammar uses two essential nonterminals, `v` and `w`; the others are introduced to reflect Zuker's case analysis. It is quite instructive to reformulate classical DP algorithms in the uniform ADP framework. Making explicit the grammar behind the algorithm helps to clarify properties relating to ambiguity as well as efficiency.

The Needleman-Wunsch Algorithm of 1970. The Needleman-Wunsch algorithm for pairwise sequence comparison [18] is based on a particularly simple yield grammar with a single nonterminal symbol `alignment`, terminals `xbase, ybase, region, empty`, and the algebra represented the five operators `replace, delete, insert, nil, h`. When sequences x and y are to be aligned, the input to this parser is x++y^{-1}.

```
 nw_alignment algebra x y              =  axiom alignment  where
     (replace, delete, insert, nil, h) =  algebra
     alignment =  tabulated (
                  replace <<<   xbase      ~~~ p alignment ~~~ ybase |||
                  delete  <<<   region     ~~~ p alignment           |||
                  insert  <<< p alignment  ~~~   region              |||
                  nil     ><<   empty                                ... h)
```

the declarative view, but operationally they are special cases with a more efficient implementation. E.g., `~~-` is used when the righthand parser accepts a single base.

3 Ambiguity and Canonicity

3.1 Formalizing Ambiguity and Canonicity

Remember that a context-free grammar $\mathcal{G}$ is ambiguous, if there are different leftmost derivations for some $x \in \mathcal{L}(\mathcal{G})$.

Definition 5 (Yield Grammar Ambiguity). *A tree grammar $\mathcal{G}$ is ambiguous if there are different leftmost derivations for some tree $t \in \mathcal{L}(\mathcal{G})$. A yield grammar $(\mathcal{G}, y)$ is ambiguous, if $\mathcal{G}$ is ambiguous, otherwise it is unambiguous. A yield grammar $(\mathcal{G}, y)$ is strictly unambiguous, if it is unambiguous and y is injective.*

Strict unambiguity means that for each $s \in \mathcal{A}^*$, we have at most one $t \in \mathcal{L}(\mathcal{G})$ such that $y(t) = s$. Hence, we do not have an optimization problem at all. Strictly unambiguous yield grammars play no part in dynamic programming.

Canonicity means that all solutions from which we want to choose an optimal one have a *unique* representation in the search space. For example, alignments as shown in Sect. 1.1 could be canonized by requiring that deletions are arranged always before adjacent insertions. To formalize canonicity, we must introduce a canonical model as the point of reference.

Definition 6 (Canonical Models and Canonical Yield Grammars). *Let $\mathcal{K}$ be a set, the canonical model. Let k be a mapping from $\mathcal{L}(\mathcal{G})$ to $\mathcal{K}$. A yield grammar $(\mathcal{G}, y)$ is canonical w.r.t. $\mathcal{K}$ and k if it is unambiguous and the mapping k is bijective. A DP algorithm is canonical w.r.t. $\mathcal{K}$ and k, if the underlying yield grammar is canonical w.r.t. $\mathcal{K}$ and k.*

The canonical model may exist merely in the mind of the algorithm designer, but preferably, it should be formulated explicitly, together with the mapping k.

3.2 Analysing Canonicity

We show that the Zuker algorithm is not canonical. A canonical model for RNA secondary structures would be sets of properly nested base pairs. Such a model is too remote from the tree-like representation of RNA structures. The Vienna notation, encoding a structure as a string of dots and properly nested parentheses, however, proves to be very convenient. It can be formally defined as $\mathcal{L}(\mathcal{V})$, using the string grammar $\mathcal{V} = \{R \to .|..|S, \quad S \to ...|.S|S.|SS|(S)\}$. Our little hairpin $\mathtt{s}$ would be denoted by the pair ("((...(....)))","ccaaaaguguugg"). The mapping k from Zuker's underlying data type $\mathcal{Z}$ to $\mathcal{L}(\mathcal{V})$ is defined via

$$\begin{aligned} k(bi(a,u,v,b) &= \texttt{"("}\ \texttt{++}\ k(u)\ \texttt{++}\ k(v)\ \texttt{++}\ \texttt{")"} \\ k(ol(a,v) &= \texttt{"."}\ \texttt{++}\ k(v) \\ k(co(u,v) &= k(u)\ \texttt{++}\ k(v) \\ k(ox(u,b) &= k(u)\ \texttt{++}\ \texttt{"."} \end{aligned}$$

Further equations are omitted, as these suffice to prove the equalities below.

Theorem 1. *The Zuker DP algorithm for RNA folding is not canonical with respect to feasible RNA structures.*

Proof. We observe the equalities

$$k(ol(a, ox(w,b))) = k(ox(ol(a,w),b) \tag{1}$$
$$k(co(u, co(v,w))) = k(co(co(u,v),w)) \tag{2}$$
$$k(ol(u, co(v,w))) = k(co(ol(u,v),w)) \tag{3}$$
$$k(bi(a,u,co(v,w),b)) = k(bi(a,co(u,v),w,b)) \tag{4}$$
$$k(bi(a,ox(u,b),w,c)) = k(bi(a,u,ol(b,w),c)) \tag{5}$$

Either one of these proves that k is not injective.

While equalities (1) and (2) are quite obvious and easy to avoid, (3) – (5) are more subtle, and there may be more such equalities.

The degree of redundancy incurred by the non-canonical grammar is demonstrated in Section 4.4. Such redundancy is *not* an efficiency problem, as the asymptotic efficiency of a DP algorithm is not affected. However, it makes it impossible to use the same recurrences for other purposes, say for the enumeration of all suboptimal solutions. This explains why Zuker's algorithm employs an incomplete heuristics when enumerating suboptimal foldings.

4 Master Recurrences for RNA Folding

4.1 A Canonical Grammar for Feasible RNA Structures

In [8] a data type $\mathcal{FS}$ is given together with a grammar $\mathcal{G}_f$ of all feasible structures. $\mathcal{FS}$ extends $\mathcal{T}$ as used above by operators `ss` and `ml` representing single stranded and multiloop structures, plus `cons` and `ul` for constructing component lists. It is easy to show by induction that

$\mathcal{L}(\mathcal{G}_f) \subset \mathcal{FS}$, and there is a canonical mapping $k : \mathcal{L}(\mathcal{G}_f) \to \mathcal{L}(\mathcal{V})$. Another grammar for feasible structures is implicitly given by the recurrences developed in [22]. These recurrences are designed for canonicity, since the authors seek a complete and non-redundant enumeration of suboptimal structures. We do not show either grammar here as we plan to go one step further, which will provide a significant reduction in the number of structures to be considered.

4.2 A Canonical Grammar for Canonical RNA Secondary Structures

Although the energy model permits structures of minimal free energy with isolated (unstacked) base pairs, there are good biophysical arguments to consider such structures unrealistic. As already noted by Zuker and Sankoff in [24][2], removing such redundant structures from the search space is the key to obtaining more significant near-optimal solutions.

Definition 7. *An RNA structure without isolated base pairs is canonical.*

The canonical model suiting this definition is defined as $\mathcal{L}(\mathcal{W}) \times \mathcal{A}^*$ using the string grammar $\mathcal{W} = \{R \to \epsilon|.|..|S, \quad S \to ...|.S|S.|SS|((P)), \quad P \to S|(P)\}$. $(d, s) \in \mathcal{K}$ is subject to the restriction that bases in s can pair as indicated by matching parentheses in$d$. The following grammar $\mathcal{G}_c$ for canonical RNA structures is based on the data type $\mathcal{FS}$. It uses an algebra with several base sets, and an overloaded choice function **h**.

```
canonicals alg x = axiom struct where
  (str,ss,hl,sr,bl,br,il,ml,nil,cons,ul,h,) = alg
  singlestrand = ss <<< region
  struct = str <<< p comps                        |||
           str <<< (ul  <<< singlestrand)         |||
           str <<< (nil ><<  empty)                         ... h
  comps  = tabulated (cons  <<< p block ~~~ p comps  |||
                      ul    <<< p block              |||
                      cons  <<< p block ~~~ (ul <<< singlestrand) ... h)
  block  = tabulated (p strong ||| bl <<< region ~~~ p strong     ... h)

  strong = tabulated (((sr <<< base -~~ ( p strong ||| p weak)    ~~- base)
                                             `with` basepair) ... h)
  weak   = tabulated (((hairpin ||| leftB ||| rightB ||| iloop ||| multiloop)
                                             `with` basepair) ... h)
    where
```

[2] Zuker and Sankoff suggest an even stronger restriction to structures with maximal helices. Such recurrences are within the scope of the ADP approach [5], but well beyond the space limits of this article.

```
hairpin    = hl <<< base -~~ (region `with` minsize 3)          ~~- base
leftB      = sr <<< base -~~ (bl   <<< region     ~~~ p strong) ~~- base
rightB     = sr <<< base -~~ (br   <<< p strong   ~~~ region)   ~~- base
multiloop  = ml <<< base -~~ (cons <<< p  block   ~~~ p comps)  ~~- base
iloop      = sr <<< base -~~ (il   <<< region     ~~~ p strong
                                                  ~~~ region)   ~~- base
```

The grammar distinguishes substructures closed by a single base pair (`weak`) from those closed by at least two stacked pairs (`strong`). If we identify these two nonterminals and merge their productions, an ADP version of the Wuchty et al. DP recurrences [22] is obtained. Note how the grammar takes care that single strands and closed components alternate in multiloops, and that multiloops contain at least two branches.

We now specify the canonical mapping $k : \mathcal{L}(\mathcal{G}_c) \to \mathcal{L}(\mathcal{W})$

```
k (str cs)       = k'' cs
k (hl b1 r b2)   = "(" ++ k' r    ++ ")"
k (sr b1 s b2)   = "(" ++ k' s    ++ ")"
k (ml b1 cs b2)  = "(" ++ k'' cs  ++ ")"
k (bl r s)       = k' r ++ k  s
k (br s r)       = k s  ++ k' r
k (il r1 s r2)   = k' r1 ++ k s ++ k r2
k (ss r)         = k' r
k' r             = ['.' | b <- r]    -- a sequence of |r| dots
k'' cs           = concat (map k cs) -- concatenating (k c) for all c in cs
```

Theorem 2. *Grammar $\mathcal{G}_c$ is a canonical yield grammar for canonical RNA secondary structures.*

Proof. We have to show that (a) the grammar $\mathcal{G}_c$ is unambiguous, and (b) the mapping k is bijective. (a) is shown by induction on the derivations of the grammar. For (b), injectivity of k is shown by structural recursion, while surjectivity uses a string grammar $k(\mathcal{G}_c)$ (in analogy to $y(\mathcal{G})$ in Sect. 3) to show that $\mathcal{L}(\mathcal{W}) \subset \mathcal{L}(k(\mathcal{G}_c))$. Details are omitted.

4.3 Efficiency

A canonical grammar, whether encoded in ADP or in conventional matrix recurrences, may require some extra tables compared to its non-canonical counterpart, in order to keep more structures distinct. In our RNA example, the non-canonical grammar `zuker81` uses 2 tables, while the canonical grammar `canonicals` uses 4. This is the price for the added versatility.

4.4 Applications

Due to Theorem 2 we know that the DP algorithm $\mathcal{G}_c$ considers each canonical RNA structure exactly once. Hence it can serve as a "master copy" of *all* DP algorithms which can be formulated as a $\mathcal{FS}$-algebra.

Simple evaluation algebras. The analyses in Table 1 can be defined each in a few lines: Energy minimization for canonical structures has been de-

Purpose	Value Domain	Interpretation of Operators	Choice Function
Energy minimization	energy values	energy rules for hairpin loops, bulges, stacked pairs, etc.	minimi-zation
Structure enumeration	trees in data type $\mathcal{T}$	tree constructors `HL, IL, SR,` etc.	identity function
Structure counting	Integers	multiply counts of substructures	summation
Structure count estimation	Reals	multiply counts by pairing prob.	summation

Table 1. Different analyses based on $\mathcal{G}_c$

signed and implemented in [5] and [16]. Structure enumeration has been used to generate visualizations of the folding space via RNA-Movies [6]. Structure counts are correct due to canonicity of the grammar, and canonization of structures proves a dramatic reduction of the folding space. A probabilistic estimate for feasible structures obtained in this manner[3] was already shown in [8] to be remarkably accurate. Accuracy is confirmed for the estimate of canonical structures supplied here. The Waterman formula [20] for the number of possible structures for all sequences of length n can also be written as a simple yield grammar.

Some Observations. A few of the observations made by applying these evaluation algebras are summarized in Table 2, showing structure statistics for initial segments of an RNA sequence[4] from *neurospora crassi*. n denotes sequence length, and the columns list structures counted, estimated, or evaluated by the various algorithms. These figures also indicate that the majority of structures accounted for by Waterman's formula do not exist in the folding space of a *given* sequence, the majority of structures considered by the Zuker algorithm is redundant, and the majority of the feasible structures enumerated by the non-redundant Wuchty algorithm is non-canonical.

Combined analyses. Since all algebras share the same grammar, a general construction is available [9] that forms the cross product of two algebras. Everything is mechanic except the combined choice function. This means

[3] Equivalently based on the canonical grammar for feasible structures or on the special recurrences given in [24], but modified to reflect base composition.

[4] `"gaccauacccacuggaaaacucgggaucccguccgcucuccca..."`.

n	Waterman formula	Zuker algorithm	Probabilistic estimate feasibles	Feasible structs.	Canonical structs.	Probabilistic estimate canonicals
5	8	0	1.16	1	1	1.00
10	423	12	5.98	9	1	1.34
15	30372	544	100.82	106	7	5.82
20	2516347	38160	510.60	390	7	9.02
25	226460893	2428352	15160.50	16343	72	71.37
30	21511212261	229202163	175550.00	235025	244	233.80

Table 2. Some structure statistics collected via the algebras listed in Table 1

we can, for example, return an optimal solution together with the total number of co-optimal solutions.

Structural Motifs. Retaining the evaluation algebras and the canonical model, but specializing the grammar, we obtain the above analyses for all classes of structural motifs that can be described by a regular tree grammar.

Application to pairwise sequence comparison. In many situations, confidence in the answer computed by a sequence alignment algorithm could be substantiated by reporting the number of (different) co-optimal answers, accompanied by some measure of the diversity within the co-optimal answer space. This, again, requires canonization, which can be achieved by our approach. For lack of space, we refer the reader to the "alignment ambiguity awareness suite" in [9], Chapter 3.

5 Conclusion

We have provided a framework for reasoning about properties of DP algorithms related to ambiguity. We hope to have shown that canonical yield grammars are a useful concept both in theory – understanding the properties of DP algorithms – and in practice – building reliable and versatile DP algorithms more quickly. We expect to apply this approach to further problem domains, as we are working to explore the full scope of algebraic dynamic programming.

References

1. R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.
2. W.S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
3. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

4. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.
5. D. Evers. *RNA Folding via Algebraic Dynamic Programming.* Bielefeld University, 2000. Forthcoming Dissertation.
6. D. Evers and R. Giegerich. RNA Movies: Visualizing RNA Secondary Structure Spaces. *Bioinformatics*, 15(1):32–37, 1999.
7. R. Giegerich. Code Selection by Inversion of Order-Sorted Derivors. *Theor. Comput. Sci.*, 73:177–211, 1990.
8. R. Giegerich. A declarative approach to the development of dynamic programming algorithms, applied to RNA folding. Report 98–02, Technische Fakultät, Universität Bielefeld, 1998.
9. R. Giegerich. Towards a discipline of dynamic programming in bioinformatics. Parts 1 and 2: Sequence comparison and RNA folding. Report 99–05, Technische Fakultät, Universität Bielefeld, 1999. (Lecture Notes).
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
11. D. Gusfield, K. Balasubramanian, and D. Naor. Parametric Optimization of Sequence Alignment. *Algorithmica*, 12:312–326, 1994.
12. I.L. Hofacker, P. Schuster, and P.F. Stadler. Combinatorics of rna secondary structures. *Discr. Appl. Math*, 89:177–207, 1999.
13. G. Hutton. Higher Order Functions for Parsing. *Journal of Functional Programming*, **3**(2):323–343, 1992.
14. S. Kurtz and G. W. Myers. Estimating the Probability of Approximate Matches. In *Proceedings Combinatorial Pattern Matching*, pages 52–64, 1997.
15. H.T. Mevissen and M. Vingron. Quantifying the Local Reliability of a Sequence Alignment. *Prot. Eng.*, 9(2), 1996.
16. C. Meyer. Lazy Auswertung von Rekurrenzen der Dynamischen Programmierung, 1999. Diploma Thesis, Bielefeld University, (in German).
17. D. Naor and D. Brutlag. On Near-Optimal Alignments in Biological Sequences. *J. Comp. Biol.*, 1:349–366, 1994.
18. S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.
19. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.
20. M. S. Waterman and T. F. Smith. RNA secondary structure: A complete mathematical analysis. *Math. Biosci.*, 41:257–266, 1978.
21. M.S. Waterman and T.H. Byers. A dynamic programming algorithm to find all solutions in a neighborhood of the optimum. *Math. Biosci.*, 77:179–188, 1985.
22. S. Wuchty, I. Fontana, W.and Hofacker, and P. Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1998.
23. M. Zuker. On Finding all Suboptimal Foldings of an RNA Molecule. *Science*, 244:48–52, 1989.
24. M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.
25. M. Zuker and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Res.*, 9(1):133–148, 1981.