Matching and Significance Evaluation of combined Sequence-Structure Motifs in RNA

Carsten Meyer, Robert Giegerich

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

Abstract. The discipline of Algebraic Dynamic Programming is a powerful method to design and implement versatile pattern matching algorithms on sequences; here we consider mixed sequence and secondary structure motifs in RNA. A recurring challenge when designing new pattern matchers is to provide a statistical analysis of pattern significance. We demonstrate that by the use of so-called canonical pattern descriptions, the expected number of hits on a sequence of length n can be computed a priori, using the pattern matcher itself. This provides a systematic way to calibrate the specificity of pattern matching algorithms. The technique is exemplified by examples using IRE and SECIS elements.

1 Motivation and Overview

1.1 Evaluation of Pattern Significance

Significance of standard patterns. A mathematical evaluation of the significance of hits is essential in all pattern matching applications. For simple sequence patterns, the approach of [2], implemented in the tool *Verbumculus*, provides a complete analysis of over- and underrepresented words in a sequence. The E-values computed by the BLAST programs [1] for sequence similarity search give an account of how small the probability is to find a given match by chance. Similarly, the REPuter tool [24] provides significance scores in the analysis of approximate repeats and palindromes in genome data.

Significance analysis of general patterns. Sequence subwords, local similarity and repeats are rather elementary patterns. In searching for, say, regulatory elements in RNA, we need all of the above, and more: We design patterns that adhere to a structural shape, may carry some well defined sequence motifs, but else allow considerable variation in both sequence and structure. When designing such a motif description, a central problem is to balance specificity against variation, or in statistical terms: to analyse $E_m(n, c)$, the expected number of hits of the motif m on a random sequence of length n and base composition c.

Note that $E_m(n, c)$ is a property of the motif, in contrast to a BLAST E-value that rates a particular motif instance. Although an analysis of

statistical significance is highly desirable in all pattern matching applications, it is a separate mathematical effort, and often non-trivial. In fact, for the two complex motifs studied in this paper, pattern matching algorithms have been published without addressing this problem at all [8] [22].

A general approach to calculate probabilities of pattern matches was presented in [29]. The class of patterns is restricted compared to the approach presented below; and Staden calculates the probability of a match achieving a particular integer score value, rather than the expected number of matches. While the basic approach has much in common, the way to reach the goal is quite different. Staden uses the technique of probability generating functions, independent of the pattern search algorithm that may be employed. We advocate a programmer's approach, asking for a programm that can compute $E_m(n, c)$. We show that the motif description can be designed in such a way that the resulting pattern matcher itself can compute $E_m(n, c)$ a priori. The central prerequisite is to provide non-ambiguous motif descriptions within the framework of Algebraic Dynamic Programming (ADP) [15].

1.2 Ambiguity Issues in Dynamic Programming

Dynamic Programming (DP) solves combinatorial optimization problems. It is a classical programming technique throughout computer science [6], and plays a dominant role in computational biology [9, 17]. 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 [3], 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. They are bound to recur with every new design of a pattern matching algorithm based on this implementation technique.

- *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 [25, 27] and parametric [18] solutions.
- *Duplicate solutions:* While there is a general technique to enumerate all solutions to a DP problem (possibly up to some threshold value) [32, 33], 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-factum filtering as a safeguard against duplicate answers are employed e.g. in [34].
- *(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 [23], 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 [20], for structure enumeration [33], energy minimization [36], and base pair maximization [28]. Again, enumerating co-optimal answers will produce duplicates in the latter two cases.

Canonicity and statistical significance. The atomic patterns in RNA motifs are (un-gapped) sequence patterns and base pairings. For both of these, their individual significance can be evaluated when the base composition (the share of nucleotides A,C,G,U) of the data is known. The work presented here is based on two observations:

1. If the motif description takes care that each larger motif is composed from smaller ones in a non-ambiguous way, the significance of a composed motif can be computed from the significance of its constituents.
2. If the pattern matcher implementing the motif search finds each motif exactly once, then this very program can be used for computing $E_m(n, c)$ given only n and c, instead of a specific sequence.

Structure of this paper. In the first part of this paper (Sections 2 and 3) we give a rather short review of the technique of Algebraic Dynamic Programming, and introduce the notion of canonical pattern descriptions in ADP. The former can be found with more detail in [15] and (in tutorial form) in [11]. The latter aspect is studied in more detail in [16]. In the second part (Section 4), we develop canonical pattern descriptions for two RNA motifs, the Iron Responsive Element and the Selenocysteine Insertion Sequence.

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.

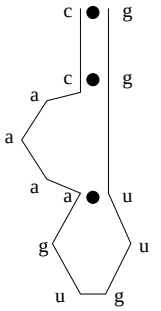
2.1 Basic Notions

Let $\mathcal{A}$ be an alphabet. $\mathcal{A}^*$ denotes the set of finite strings over $\mathcal{A}$, and ++ denotes string concatenation. Throughout this article, $x \in \mathcal{A}^*$ denotes the input string and $|x| = n$. A subword is indicated by its boundaries – $x_{(i,j)}$ denotes $x_{i+1}...x_j$. When T is an arbitrary data type, $[T]$ denotes the data type of all lists with elements from T, and [] denotes the empty list.

An algebra is a set of values and a family of functions over this set. We 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 $\mathcal{I}$.

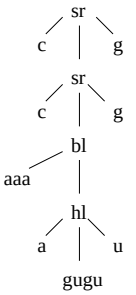
Terms as syntactic objects can be equivalently seen as trees, where each operator is a node, which has its subterms as subtrees. Tree grammars over $\mathcal{T}$ describe specific subsets of the term algebra. A regular tree grammar over $\mathcal{T}$ [5, 14] has a set of nonterminal symbols, a designated axiom symbol, and productions of the form $X \rightarrow t$ 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

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



Hairpin in term representation

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

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

Definition 1 (Evaluation Algebra). *An evaluation algebra is a $\mathcal{T}$-algebra augmented by an objective function h of type $[s] \to [s]$, where s is the base set. We require $h(l)$ to be polynomial in $|l|$. Furthermore, h is called reductive if $|h(l)|$ is bounded by a constant.*

In standard applications, the objective h is minimization or maximization, but, as we shall see, it may also be counting, estimation, or any other kind of synopsis. The reductivity of h makes a DP algorithm run in polynomial rather than in exponential time. Non-reductive choice functions are used e.g. for complete enumeration, where h is the identity function. 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 $y($`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 $\mathcal{Y}(\mathcal{G}) = \mathcal{L}(y(\mathcal{G}))$: Yield languages are context-free.

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

$$y(t) = x.$$

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* $x \in \mathcal{A}^*$ *the set of solutions*

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

This definition precisely describes a class of DP problems over sequences. All DP algorithms in biosequence analysis we have studied so far can be cast in the ADP framework.

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! It is this language in which motif descriptions in Section 4 will be expressed. 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` and `closed`. Terminal symbol `base` denotes an arbitrary base, and `region` a nonempty sequence of bases from the RNA alphabet. The grammar notation is refined further by allowing predicates and the objective function to be associated with nonterminal 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`, which applies to both alternatives, requires that the bounding bases of either closed structure form a feasible base pair. The objective function `h` is attached via the `...`-combinator, indicating that from several alternative closed structures, a synopsis 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 objective 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 objective 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 obtained by the combinator technique of [21]. 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 higher order 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 A Classical DP Algorithm in ADP Notation

Zuker's Algorithm for RNA folding. Zuker and Stiegler [36] 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 [10] has recently reformulated Zuker's recurrences as a yield grammar $\mathcal{G}_{\texttt{zuker81}}$[1]:

```
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.

A gallery of wellknown bioinformatics algorithms (Needleman-Wunsch, Smith-Waterman, Gotoh and more) is found with [11].

[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 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. 1.2 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 $\mathcal{V} = \{R \rightarrow .|..|S, \quad S \rightarrow ...|.S|S.|SS|(S)\}$. Our little hairpin `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

$$k(ol(a, ox(w, b))) = k(ox(ol(a, w), b) \quad (1)$$

$$k(co(u, co(v, w))) = k(co(co(u, v), w)) \quad (2)$$

$$k(ol(u, co(v, w))) = k(co(ol(u, v), w)) \quad (3)$$

$$k(bi(a, u, co(v, w), b)) = k(bi(a, co(u, v), w, b)) \quad (4)$$

$$k(bi(a, ox(u, b), w, c)) = k(bi(a, u, ol(b, w), c)) \quad (5)$$

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 3.5. 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. Even more, trying to evaluate statistical significance in the presence of such redundancy yields a property related to the search algorithm rather than the motif.

3.3 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 [35][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 uses an algebra with several base sets, and an overloaded objective function `h`.

[2] Zuker and Sankoff suggest an even stronger restriction to structures with maximal helices. A solution to this problem is presented in [12].

```
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
    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`). The new operator `ul` constructs a singleton list, while `cons` adds an element to the front of a list. If we identify the two nonterminals and merge their productions, an ADP version of the Wuchty et al. DP recurrences [33] 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.

3.4 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.

3.5 Algebras for Various Analyses

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	Objective Function
Energy minimization	energy values	energy rules for hairpin loops, bulges, stacked pairs, etc.	minimization
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 [10] and [26]. Structure enumeration has been used to generate visualizations of the folding space via RNA-Movies [13]. Structure counts are correct due to canonicity of the grammar, and canonization of structures proves a dramatic reduction of the folding space. The probabilistic estimates for feasible and canonical structures were obtained by the method explained in detail in Section 4. These estimates seem remarkably good: In all our test cases, the number of structures counted was within a factor of 2 of the estimate. The Waterman formula [31] for the number of possible structures for all sequences of length n can also be written as a simple yield grammar, and is included for comparison.

Table 2, showing concrete structure statistics for initial segments of an RNA sequence[3] 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

[3] `"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

Zuker algorithm is redundant, and the majority of the feasible structures enumerated by the non-redundant Wuchty algorithm is non-canonical.

4 Pattern Matching for Regulatory Elements in RNA

Retaining the evaluation algebras and the canonical model, but specializing the grammar we obtain DP algorithms to analyse all classes of structural motifs that can be described by a regular tree grammar. In difference to folding of the complete RNA sequence as described in Section 3.3 we look now for local similarities to a structural motif. By using one statistical and one combinatorial algebra we are able to use the same program to evaluate statistical significance and to perform actual structural pattern searching. This allows to calibrate the specifity of pattern descriptions. Stuctural signals are important in post-transcriptional processing of RNA. We focus on two motifs and formulate canonical grammars for them: The Iron Responsive Elements (*IREs*) and the Selenocysteine Insertion Sequence (*SECIS*).

4.1 Extensions for pattern matching

In order to decribe and evaluate structural patterns we have to extend the ADP notations explained in Sections 2.3 and 2.4 as well as the pattern constructs and the evaluation algebras (see Section 2.2).

Pattern constructs. Table 3 shows a summary of the additional required pattern constructs. For our example motifs these constructs are sufficient. If there are other pattern constructs necessary for the description of further motifs, it is easy to formulate them in the same manner.

Evaluation algebras. A report on the particular pattern instances found, indicating their location in the sequence, their concrete constituents etc. is obtained via the enumeration algebra (see Section 3.5). We formulate a *counting algebra* to calculate the number of occurences of the structural RNA motif and an *expectation algebra* to compute the expected number of appearances of the search pattern. In the *counting algebra* the answer data type is fixed to `Int` and the evaluation functions are defined to multiply

pattern construct	meaning
`unp lb t rb`	(unpaired) The bases `lb` and `rb` cannot form a feasible base pair.
`nwc lb t rb`	(non_watson_crick) The bases `lb` and `rb` form a non-Watson-Crick base pair.
`hl lb us rb`	(hairpin) The bases `lb` and `rb` form a feasible base pair and the hairpin consists of a sequence motif.
`rp lb t rb`	required pair of specific bases `lb` and `rb`
`skip_left _ t`	skips one base at the $5'$ side
`skip_right t _`	skips one base at the $3'$ side
`loop us t _`	Internal loop: The left singlestranded region is a sequence motif. The right singlestranded region is arbitrary.
`lr _ us`	(left region) Pattern consisting of an arbitrary region at the $5'$ end and a specific sequence motif at the $3'$ end (`us`).
`rr us _`	(right region) Pattern consisting of a specific sequence motif at the $5'$ end (`us`) and of an arbitrary region at the $3'$ end.

Table 3. The additional pattern constructs for pattern matching. `t` always stands for the included substructure, and `us` means a specific sequence motif given by a *IUPAC* string.

counts of structure constituents, whereas in the *expectation algebra* the answer data type is `Float` and the evaluation functions are defined to multiply the probabilities of the structure constituents.

The objective function `h` is defined to sum over the elements in a list, returning a unitary list rather than an integer to conform to the overall scheme.

```
h []            = []
h  xs           = [sum xs]
```

In the *counting alegbra* the list elements are structure counts and in the *expectation algebra* expectation values.

Common evaluation functions. The evaluation functions `str`, `ss`, `lr`, `rr`, `skip_left` and `skip_right` are defined identically in both algebras.

```
str t           = t         ss _            = 1
lr _ us         = us        rr us _         = us
skip_left  _ t  = t         skip_right t _  = t
```

Specific evaluation functions. The functions `loop`, `sr`, `unp`, `hl`, `nwc` and `rp` are specific for the different evaluation algebras. The definition of the predicates `basepairing`, `non_watcr_pairing` and `nopairing` also depend on the evaluation strategy[4].

Counting alegbra.

```
loop us t _     = t         sr lb t rb     = t
unp  lb t rb    = t         hl lb us rb    = 1
```

[4] The predicate `iupacmatch`, which also depends on the evaluation algebra, is explained in the next paragraph.

```
rp   lb t rb    = t         nwc lb t rb    = t

basepairing (i,j)       = (i < j) && pair (x!(i+1), (x!j))
non_watcr_pairing (i,j) = (i < j) && not (watcr_pair (x!(i+1), (x!j)))
nopairing (i,j)         = (i < j) && not (pair (x!(i+1), (x!j)))
```

`x` is a global variable referring to an array which contains the input sequence indexed by position. The `basepairing (i,j)` predicate checks if the bases at the positions $i+1$ and j can form a feasible basepair[5], the `non_watcr_pairing` predicate checks if the bases at the positions $i+1$ and j can form any non-Watson-Crick base pair and the `nopairing` predicate checks if the bases at the positions $i+1$ and j cannot form a feasible basepair.

Expectation algebra. In order to define the evaluation functions the base composition of the examined RNA sequence, the base pairing probability[6] and the Watson-Crick base pairing probability[7] are used. For every character of the *IUPAC* code its probability is computed from the base composition and stored in the array `ubasecomp`.

```
loop us t _          = t * product[ubasecomp!u | u <- us]
sr  lb t rb          = t * pair_prob
unp lb t rb          = t * (1 - pair_prob)
nwc lb t rb          = t * (1 - watcr_pair_prob)
hl lb us rb          = pair_prob * product [ubasecomp!u | u <- us]
rp lb t rb           = t * ubasecomp!lb * ubasecomp!rb

nopairing (i,j)         = True
basepairing (i,j)       = True
non_watcr_pairing (i,j) = True
```

The predicates `basepairing`, `non_watcr_pairing` and `nopairing` are defined to deliver always `True`, because the evaluation takes place inside the algebra functions. The *expectation algebra* does (in contrast to the *counting algebra*) not look inside the examined sequence. To evaluate, for example, the probability that the bases at positions $i+1$ and j pair, the algebra uses the `pair_probability` value to obtain the result. The same strategy is used to calculate the probability of two bases not to form a feasible base pair or to form a non-Watson-Crick base pair.

Handling of specific sequence motifs. We introduce the `iupac` parser to evaluate fixed sequence patterns.

```
iupac us iupacmatch (i,j) = [us | iupacmatch us [x!k | k <- [i+1 .. j]]]
```

`Iupacmatch` is an auxiliary boolean function in the evaluation algebras. If `iupacmatch` is succesful, the `iupac` parser returns the *IUPAC* string `us`.

[5] Feasible base pairs are the Watson Crick pairs C-G and A-U as well as the Wobble base pair G-U.

[6] This value includes all feasible base pairing possibilities.

[7] This value only includes the Watson Crick pairs A-U and G-C.

In the case of the *counting algebra* `iupacmatch` checks, if `us` matches the sequence part $s_{i+1}..s_j$. In the case of the *expectation algebra* the `iupacmatch` function returns `True`, if `us` has the same length as the considered input interval $s_{i+1}..s_j$. The corresponding functions of the *expectation algebra* calculate the probability of the string `us` according to the base composition of the input sequence[8].

The `iupac` clause is used to define the `fbase` parser, which looks for a specified *IUPAC* character, and to define the `base` parser to search for an arbitrary nucleotide:

```
fbase x = iupac x
base    = fbase "N"
```

Iterative constructs. To conveniently describe helical regions whose sizes are controlled by parameters, we define a `stackscheme`, the `rep` construct and the `upto` construct. The `stackscheme` checks for one base pair of a helical region. The `rep` construct, which parses a given scheme `q` a specified number of times and then continues with `r`, is used to look for stacking regions of specified length. The `upto` construct, which parses a given scheme upto a specified number of times before continuing with `r`, enables us to describe helical regions of flexible size.

```
stackscheme r  = (sr <<< base -~~ r ~~- base) `with` basepairing
rep 0 q r      = r
rep (n+1) q r  = q (rep n q r)
upto 0 q r     = r
upto (n+1) q r = r ||| q (upto n q r)
```

Combinators. We define two new combinators to restrict the input intervals of the lefthand parser `r` respectively of the righthand parser `q` according to a given lower bound `u` and an upper bound `o`. In the case of the ~!~combinator the lefthand parser `r` works only on the intervals starting from `i` with the minimal length `u` upto the maximum length `o` and the righthand parser `q` evaluates the rest of the input interval $i..j$. The ~!!~combinator gives the possibility to define an interval for the righthand parser `q`.

```
(~!~) u o r q (i,j)
 = [x y | k <- [min (i+u) j .. min (i+o) j] , x <- r (i,k) , y <- q (k,j)]

(~!!~) u o r q (i,j)
 = [x y | k <- [max (j-o) i .. max (j-u) i] , x <- r (i,k) , y <- q (k,j)]
```

4.2 The Iron Responsive Element

Iron Responsive Elements (*IREs*) are regulatory signals found for example in the 5'UTR of ferritin-mRNAs [**?**]. Depending on the amount of iron in the cell they effect the translation efficiency of the ferritin-mRNA.

[8] Note again, since the *IUPAC* string comes from the pattern, the expectation algebra need not access the input characters.

An *IRE* is a stem-loop structure consisting of a "CAGUGH" [9] hairpin, a stack of four to six base pairs, an internal loop and another stack of at least two base pairs. The nucleotide next to the four to six base pair stack in the left region of the interior loop must be a *Cytosin*.

One (out of many) secondary structure for sequence `i = "AACCAGGCAA GUGCAGUGCCGCUUUUGGG"` is an *IRE* element. The formula `s` below is an algebraic representation of this secondary structure.

```
s = STR [SS (0,3),(SR 'C'(SR 'A' (IL (5,8) (SR 'A' (SR 'A'(SR
'G' (SR 'U'(HL 'G' "CAGUGC" 'C') 'C') 'G') 'U') 'U') (24,25))
'U') 'G'), SS (27,29)]
```

Fig. 1 shows the term representation of the formula `s` and the *IRE* structure. The structure is divided into different parts. For each variable part of the *IRE* a short description with an abbreviation is given.

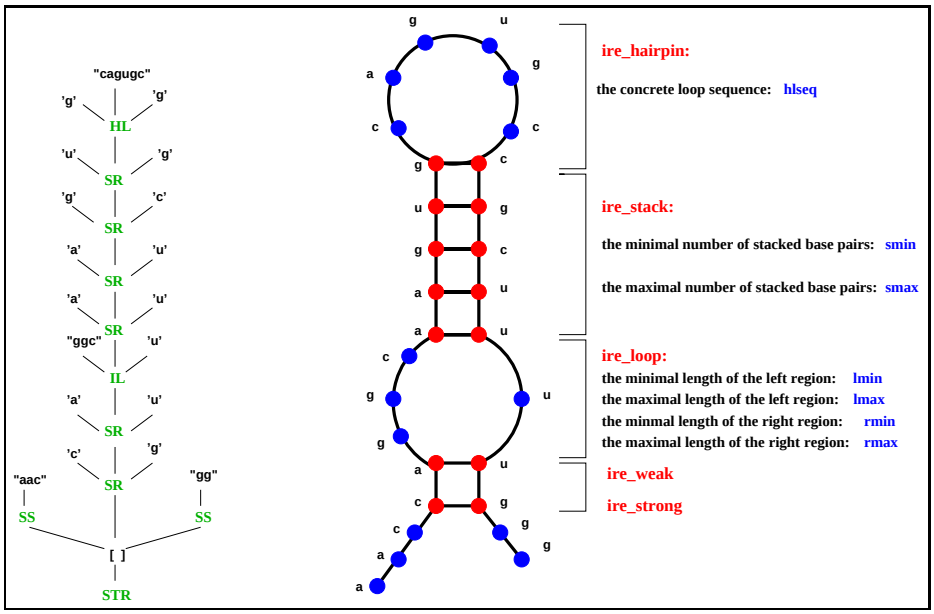


Fig. 1. The *Iron Responsive Element*

4.3 A Canonical Grammar for *IRE* patterns

We now specialize the grammar for canonical structures to get a recognizer of *IREs* in UTRs. Because *IREs* vary between *eukaryotes* and *prokaryotes* [7], we allow a variable description of the *IRE* search pattern using the parameters indicated in the *IRE* graphic (see Fig.1).

[9] We use the *IUPAC*-notation. The last nucleotide must not pair with the Cytosine in the first position of the loop.

```
IRE alg lmin lmax rmin rmax smin smax hlseq inp = axiom (p lcomps) where
  (str,ss,hl,sr,lr,skip_left,skip_right,loop,h) = alg
  (~!+~)        = lmin ~!~  lmax
  (~!!+~)       = rmin ~!!~ rmax
  lcomps        = tabulated (
    str <<<  (skip_left <<<  base -~~ p lcomps ||| p rcomps      ... h))
  rcomps        = tabulated (
    skip_right <<<  p rcomps ~~- base ||| p ire_strong           ... h)
  ire_strong    = tabulated (
    (sr <<< base -~~ ire_weak ~~- base)  with‘ basepairing)
  ire_weak      = sr <<< base -~~ p ire_loop  ~~- base)
                                              ‘with‘ basepairing
  usinglestrand = ss <<< uregion
  ire_loop      = tabulated ((loop <<< (lr <<< usinglestrand  ~~-
    (fbase "C")) ~!+~ p ire_stack  ~!!+~  usinglestrand)         ... h)
  stackscheme r = (sr <<< base -~~ r ~~- base) ‘with‘ basepairing
  ire_stack     = tabulated ((upto (smax-smin) stackscheme
    (rep (smin-1) stackscheme ire_hairpin))                      ... h)
  ire_hairpin   = (hl <<< base -~~ (iupac hlseq) ~~- base)
                                              ‘with‘ basepairing
```

The parsers `lcomps` and `rcomps` process singlestranded regions adjacent to the *IRE*. The parsers `ire_strong`, `ire_weak`, `ire_loop`, `ire_stack` and `ire_hairpin` recognize *IRE* components as indicated in the *IRE* graphic (see Fig.1). The `~!+~` and the `~!!+~` combinators bind the loop size parameters to the combinators `~!~` and `~!!~` as explained in section 4.1 to allow only loops with the desired sizes. Using the `fbase` clause we insure that the left region of the internal loop ends with a *Cytosin*. The `rep` and the `upto` constructs enables us to describe an `ire_stack` consisting of at least `smin` and at most `smax` base pairs. It should be mentioned that the last base pair of the stack is found by the `ire_hairpin` parser, which checks for the desired hairpin sequence `hlseq` by using the `iupac` clause.

The recognition of *IREs* requires $O(n^2)$ space and $O(n^2 + m)$ time, where n is the length of the input and m the number of matches requested.

To ensure canonicity of the *IRE* grammar, two productions (`lcomps` and `rcomps`) must be used to ensure that bases at the $5'$ and $3'$ end cannot be skipped in different, but equivalent orders. For the rest of the grammar, canonicity follows from Theorem 2 (see Section 3.3). Hence, the pattern matcher defined by this grammar can compute matches, match counts as well as expected match numbers.

4.4 The Selenocysteine Insertion Sequence

The *SECIS* element [4] [22] is found in 3' UTR of mRNAs which encode for proteins containing the aminoacid selenocysteine. Selenocysteine is encoded by `UGA`, which normally functions as a stop codon. The regulatory structure is necessary for incorporation of selenocysteine at an UGA codon.

The *SECIS* element is a stem-loop structure consisting of a hairpin of 10 to 24 nucletides, a helix of 13 base pairs, an internal loop and another stack of at least four base pairs [30]. There can occur mismatched bases and bulges inside the long helical region. The hairpin loop contains a "`AA`" sequence motif within the first seven nucleotides from the $5'$ end. The four base pairs at the $5'$ end of the long helix are a special component called *Quartet*. It consists of four non-Watson-Crick base pairs. Some of those nucleotides are fixed. The $5'$ part of the internal loop is three to seven nucleotides long and the nucleotide next to the *Quartet* is an *Adenosin*. The $3'$ part of the internal loop is four to nine nucleotides long. Fig. 2 shows the details of the structure.

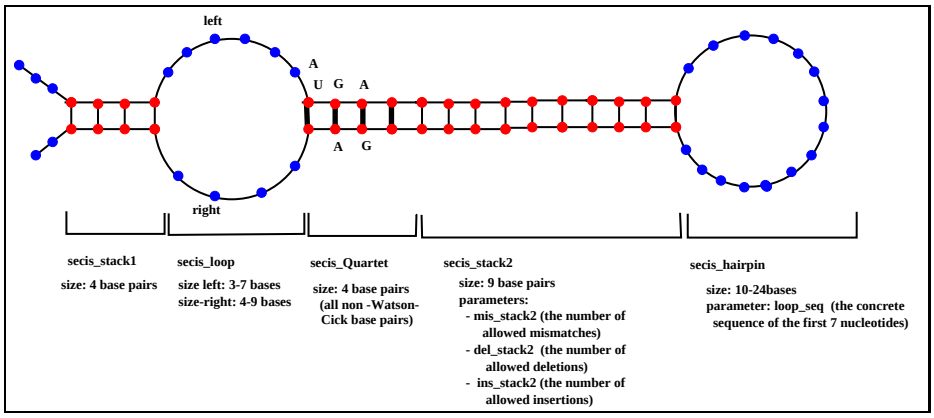


Fig. 2. The *Selenocysteine Insertion Sequence*. Only the fixed nucleotides are labeled. All the others are undetermined but must allow the claimed base pairs. The feasible base pairs are symbolized by a regular black line, the non-Watson-Crick base pairs in the *Quartet* are symbolized by a wide black line. The structure is divided into different parts. For each part a short description is given. For variable parts the parameters are listed with an abbreviation.

4.5 A Canonical Grammar for *SECIS* patterns

As in section 4.3 we specialize the grammar for canonical structures to get a recognizer of the *SECIS* element. In order to formulate a grammar for the *SECIS* element we have to modify the `rep` construct such that mismatches and bulges are allowed inside helical regions[10]. Bulges are modelled by insertions and deletions. It is possible to specify the number of mismatches, insertions and deletions for the long helical region (`secis_stack2`). Further we allow to specify the first seven nucleotides of the hairpin loop (`loop_seq`).

```
SECIS alg mis_stack2 del_stack2 ins_stack2 loop_seq inp = axiom (p lcomps)
```

[10] The formal definition is not shown.

```
  where
(str,ss,hl,sr,st,unp,nwc,lr,rr,skip_left,skip_right,loop,h) = alg
(~!+~)  = 3 ~!~  7
(~!!+~) = 4 ~!!~ 9
(~!++~) = 7 ~!~ 7
lcomps           = tabulated (
       str <<< ((skip_left <<<  base -~~ p lcomps ||| p rcomps) ... h))
rcomps           = tabulated (
       (skip_right <<< p rcomps ~~- base ||| secis_stack1)      ... h)
stackscheme r    = (sr <<< base -~~ r ~~- base)  `with` basepairing
mismatch r       = (unp <<< base -~~ r ~~- base) `with` nopairing
deletion r       = skip_left  <<< base -~~ r
insertion r      = skip_right <<< r ~~- base
secis_stack1     = tabulated (
 (rep 4 stackscheme 0 mismatch 0 deletion 0 insertion (p secis_loop)))
usinglestrand    = ss <<< uregion
secis_loop       = tabulated (
  (loop <<< (lr <<< usinglestrand   ~~- (fbase "A"))   ~!+~
            (secis_quartet)  ~!!+~  usinglestrand)              ... h)
secis_quartet    =
   (rp <<< fbase "U" -~~ secis_quartet2 ~~- fbase "U"  |||
    rp <<< fbase "U" -~~ secis_quartet2 ~~- fbase "C") |||
    rp <<< fbase "U" -~~ secis_quartet2 ~~- fbase "G")          ... h
secis_quartet2   = rp <<< fbase "G" -~~ secis_quartet3 ~~- fbase "A"
secis_quartet3   = rp <<< fbase "A" -~~ secis_quartet4 ~~- fbase "G"
secis_quartet4   = (nwc <<< base -~~ secis_stack2 ~~- base)
                                              `with` non_watcr_pairing
secis_stack2     = tabulated (
  (rep 7 stackscheme mis_stack2 mismatch del_stack2 deletion
                     ins_stack2 insertion secis_stack2end)      ... h)
secis_stack2end  = (sr <<< base -~~ secis_hairpin ~~- base)
                                                 `with` basepairing
secis_hairpin    =
((hl <<< base -~~ (hairpin `with` minloopsize 10 `with` maxloopsize 24)
                   ~~- base)  `with` basepairing)               ... h
             where
     hairpin  = rr <<< (iupac loop_seq) ~!++~ usinglestrand
```

The grammar for the *SECIS* element is similar to the *IRE* grammar. The parsers `secis_stack1`, `secis_loop`, `secis_quartet`, `secis_stack2` and `secis_hairpin` recognize *SECIS* components as indicated in the *SECIS* graphic (see Fig.2) and the parsers `lcomps` and `rcomps` process singlestranded regions on the lefthand and the righthand side of the *SECIS* element. Beside the stackscheme we define three other schemes to look for mismatches, deletions and insertions. These schemes are used to construct the long helical region with a specified number of mismatches, bulges and deletions. In order to avoid isolated base pairs the last two pairs are described separately, one inside the `secis_hairpin` parser and the other one in the `secis_stack2end` parser. Otherwise the helical region could end with an insertion, with a deletion or with a mismatch followed by just one base pair. The four non-Watson-Crick base pairs of the *Quartet* component are described with a separate parser for each pair, where the

`fbase` construct allows to check for the required bases. The `~!+~`, the `~!++~` and the `~!!+~` combinators allow only loops with the sizes mentioned in the pattern description by using the combinators explained in section 4.1.

The recognition of *SECIS* elements requires $O(n^2)$ space and $O(n^2 + m)$ time, where n is the length of the input and m the number of matches requested. Canonicity is ensured by the same reasoning as applied to the *IRE* grammar.

4.6 Results

Significance calibration. The number of occurences of the search pattern in a sequence, which is computed by using the counting algebra, is only meaningful if it lies significantly above the estimated number of hits, which is calculated by using the expectation algebra. By choice of the parameters, the specificity of the pattern search can be calibrated. For example decreasing of the allowed loop sizes or increasing of the desired number of stacked base pairs result in a higher specifity of the search pattern. Increasing of the number of allowed mismatches and bulges in stacked regions for example leads to a lower specifity. Running the recognizer m with the *expectation algebra* on an arbitrary input of length n and base composition c, it computes the expectation value $E_m(n, c)$.

Searching IREs. In order to test our *IRE* pattern matcher we use two different random sequences of length 1000 (drawn from an identically distributed basecomposition) and two human sequences that contain *IREs*:

- the 5' UTR of the human mRNA for the ferritin heavy chain (containing one *IRE* element, GenBank accesion number: D28463, length: 208 nucleotides)
- the 3' UTR of the human mRNA for the transferrin receptor (containing four *IRE* elements, GenBank accession number: X01060, length: 2464 nucleotides)

For our queries we vary the parameters for the `ire_haiprin` sequence (`hlseq`) and the parameters for the allowed length of the `ire_stack` (`smin` and `smax`). But we fix the parameters for the `ire_loop` so that the lefthand singlestranded region (see Figure 1) is one nucleotide long (`lmin` = 1, `lmax` = 1) and the righthand singlestranded region is empty (`rmin` = 0, `lmax` = 0). Table 4 summarizes the results.

Interpretation of the results for random sequences. When we use a very unspecific motif description, we get some hits in the random sequences. By looking at the corresponding E-values it becomes obvious, that those hits are not meaningful, because the expected value is close to the observed number of hits. With a more specific motif description, we do not observe any hits in the random sequences.

		random_seq1 $n = 1000$		random_seq2 $n = 1000$		ferritin $n = 208$		transferrin $n = 2464$	
hl_seq	stack sizes	E-value	count	E-value	count	E-value	count	E-value	count
nnnnnn	3 - 8 bp	2.874	3	2.877	2	0.558	1	7.054	11
nnnnnn	4 - 6 bp	1.01726	1	1.02096	0	0.17563	1	2.62773	9
cagugh	3 - 8 bp	0.00208	0	0.00209	0	0.00019	1	0.00390	4
cagugh	4 - 6 bp	0.00074	0	0.00074	0	0.00006	1	0.00145	4

Table 4. The results for the two random sequences and for the two human UTR sequences.

Interpretation of the results for the human UTR sequences. With a very unspecific motif description we observe random hits in addition to the true positives. Again the number of random hits alone is closely related to the expectation values. By increasing the specifity of the motif description we find meaningful hits only.

Searching SECIS-elements We use the same two different random sequences of length 1000 as for testing the *IRE* pattern matcher. Further we use a drosophila melanogaster and a mouse sequence that both contain *SECIS*-elements:

- Drosophila melanogaster mRNA for selenophosphate synthetase 2 (GenBank accession number: AJ278068, length: 1341 nucleotides)
- Mus musculus (house mouse) mRNA for selenoprotein P (GenBank accession number: X99807, length: 2075 nucleotides)

Deliberately, we use a rather unspecific version of the *SECIS* pattern: We do not look for specific hairpin loop sequences in our queries[11]. We vary the number of allowed mismatches (`mis_stack2`), deletions (`del_stack2`) and insertions (`ins_stack2`). Table 5 summarizes the results.

			random_seq1 $n = 1000$		random_seq2 $n = 1000$		Drosophila $n = 1341$		Mus musculus $n = 2075$	
mis	del	ins	E-value	count	E-value	count	E-value	count	E-value	count
0	0	0	0.00015	0	0.00017	0	0.00033	0	0.00025	4
1	1	1	0.01771	0	0.02019	0	0.03959	0	0.03195	10
2	2	2	0.22700	0	0.25720	0	0.50305	1	0.42538	14

Table 5. Results for the *SECIS* element.

Interpretation of the results for the SECIS *pattern.* The columns of *random_seq1* and *random_seq2* show the effect of relaxing the stack stringency. Allowing six mismatches and gaps makes the pattern practically

[11] That means the hairpin loop sequence is always arbitrary (`"nnnnnnn"`).

useless for searching large data sets. Note that the drosophila *SECIS* element is only found at this level of (un)specificity. This indicates that the hairpin loop information is truely necessary; furthermore, it suggests that it may be necessary to construct organism specific *SECIS* patterns.

5 Conclusion

The application range of the approach presented here is limited only by the expressive power of tree grammars, or in more conventional terms, the power of DP over sequences. Using our ADP-based approach a new pattern matching algorithm can be designed and tested within a few hours. Its efficiency is high enough for systematic testing of hypotheses. For screening large data sets, a more efficient version in `C` can be derived systematically by the method explained in [15].

However, the problem is not yet solved to satisfaction in the case of self-overlapping patterns. Consider the hairpin-loop in *SECIS*, described there by `"nnnnnnn"`. Assume we did allow `"aa"` anywhere in the loop sequence. Then, we would observe two distinct hits in `"ccaaacc"`. Mathematically, this is correct, since the pattern matches in two different ways. But from the practical point of view, such overlapping hits of the same pattern should be counted as a single hit. While this is not a problem for implementing the counting algebra, the proper statistics of self-overlapping patterns present an open problem and is the subject of current research.

6 Acknowledgement

We thank Thomas Töller for providing informations about the *IRE* and the *SECIS* patterns as well as the sequences for testing our programms.

References

1. S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, **215**(3):403–10, 1990.
2. A. Apostolico, M. E. Bock, S. Lonardi, and X. Xu. Efficient detection of unusual words. *Journal of Computational Biology*, 7(1/2):71–94, 2000.
3. R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.
4. M.J. Berry, L Banu, Y. Chen, S.J. Mandel, J.D. Kieffer, J.W. Harney, and P.R. Larsen. Recognition of UGA as a selenocystein codon in type I deiodinase requires sequences in the 3' untranslated region. *Nature*, **353**:273–276, 1991.
5. W. S. Brainerd. Tree generating regular systems. *Information and Control*, 14:217–231, 1969.
6. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.
7. T. Dandekar, K. Beyer, P. Bork, M-R. Kenealy, K. Pantopoulos, M. Hentze, V. Sonntag-Buck, G. Flouriot, F. Gannon, W. Keller, and S. Schreiber. Systematic genomic screening and analysis of mRNA in untranslated regions and mRNA precursors: combining experimental and computational approaches. *Bioinformatics*, 14(3):271–278, 1998.

8. T. Dandekar and M.W. Hentze. Finding the Hairpin in the Haystack: Searching for RNA Motifs. *Trends Genet.*, **11**(2):45–50, 1995.
9. R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998.
10. D. Evers. *RNA Folding via Algebraic Dynamic Programming.* Bielefeld University, 2001. Forthcoming Dissertation.
11. D. Evers and R. Giegerich. Systematic dynamic programming in bioinformatics. In *Intelligent Systems for Molecular Biology (Tutorial Notes).* AAAI Press, Menlo Park, CA, USA, 2000.
12. D. Evers and R. Giegerich. Reducing the conformation space in RNA structure prediction. Submitted., 2001.
13. Dirk Evers and Robert Giegerich. RNA Movies: Visualizing RNA Secondary Structure Spaces. *Bioinformatics*, 15(1):32–37, 1999.
14. R. Giegerich. Code Selection by Inversion of Order-Sorted Derivors. *Theor. Comput. Sci.*, 73:177–211, 1990.
15. R. Giegerich. A Systematic Approach to Dynamic Programming in Bioinformatics. *Bioinformatics*, 16:665–677, 2000.
16. R. Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proc. Combinatorial Pattern Matching*, pages 46–59. Springer Verlag, 2000.
17. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
18. D. Gusfield, K. Balasubramanian, and D. Naor. Parametric Optimization of Sequence Alignment. *Algorithmica*, 12:312–326, 1994.
19. M.W. Hentze and L.C. Kühn. Molecular control of vertebrate iron metabolism: mRNA-based regulatory circuits operated by iron, nitric oxide, and oxidative stress. *Proc. Natl. Sci. USA*, 93:8175–8182, 1996.
20. I. L. Hofacker, P. Schuster, and P. F. Stadler. Combinatorics of RNA secondary structures. *Discr. Appl. Math*, 89:177–207, 1999.
21. G. Hutton. Higher Order Functions for Parsing. *Journal of Functional Programming*, **3**(2):323–343, 1992.
22. G. Kryukov, M. Kryukov, and V. Gladyshev. New Mammalian Selenocysteine-containing Proteins Identified with an Algorithm That Searches for Selenocysteine Insertion Sequence Elements. *Cabios*, **274**(48):33888–33897, 1999.
23. S. Kurtz and G. W. Myers. Estimating the Probability of Approximate Matches. In *Proceedings Combinatorial Pattern Matching*, pages 52–64, 1997.
24. S. Kurtz, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Computation and visualization of degenerate repeats in complete genomes. In *Proc. Intelligent Systems for Molecular Biology*, pages 228–238. AAAI Press, Menlo Park, CA, USA, 2000.
25. H.T. Mevissen and M. Vingron. Quantifying the Local Reliability of a Sequence Alignment. *Prot. Eng.*, 9(2), 1996.
26. C. Meyer. Lazy Evaluation of Recurrences in Dynamic Programming, 1999. Diploma Thesis, Bielefeld University (in German).
27. D. Naor and D. Brutlag. On Near-Optimal Alignments in Biological Sequences. *J. Comp. Biol.*, 1:349–366, 1994.
28. R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.
29. R. Staden. Methods for calculating the probabilities of finding patterns in sequences. *Cabios*, **5**(2):89–96, 1989.
30. R. Walczak, E. Westhof, P. Carbon, and A. Krol. A novel RNA structural motif in the selenocysteine insertion element of eukaryotic selenoprotein mRNAs. *RNA*, **2**:367–379, 1996.
31. M. S. Waterman and T. F. Smith. RNA secondary structure: A complete mathematical analysis. *Math. Biosci.*, 41:257–266, 1978.

32. M.S. Waterman and T.H. Byers. A dynamic programming algorithm to find all solutions in a neighborhood of the optimum. *Mathematical Biosciences*, 77:179–188, 1985.
33. 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.
34. M. Zuker. On Finding all Suboptimal Foldings of an RNA Molecule. *Science*, 244:48–52, 1989.
35. M. Zuker and S. Sankoff. RNA secondary structures and their prediction. *Bull. Math. Biol.*, 46:591–621, 1984.
36. 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.