A Declarative Approach to the Development of Dynamic Programming Algorithms, Applied to RNA Folding

Robert Giegerich*
Bielefeld University

November 30, 1998

Abstract

A new approach to the systematic development of dynamic programming algorithms is presented and applied to RNA folding. Analyses of the potential foldings of an RNA molecule have mainly been restricted to energy minimization over all feasible secondary structures. In order to develop more specific analyses, we split up the traditional dynamic programming approach into a structure recognition and an evaluation phase. Regular tree grammars are used to describe the recognized class of structures. This allows to analyse for presence (or even absence) of very specific structures in the folding space of an RNA molecule. An EBNF-like notation for tree grammars is developed. It turns into a recognizer by interpreting the EBNF-operators as parser combinators; the polynomial efficiency of dynamic programming is regained by the introduction of tabulating yield parsers.

By abstraction from the result constructors of a recognizer, we obtain an abstract folding space evaluator, described in terms of higher-order functions. It can be instantiated towards any kind of analysis that can be specified by a so-called folding space evaluation algebra. Given such an instantiation, a first order implementation – i.e. a correct and efficient set of dynamic programming recurrencies – can be derived by straightforward mathematical reasoning. Overall, this results in a modular approach – different recognizers and different analyses can be combined freely.

*Postal address: Faculty of Technology, University Bielefeld, Postfach 100131, 33501 Bielefeld, Germany. e-mail: robert@techfak.uni-bielefeld.de

Contents

1 Motivation and Overview

1.1 Statement of problem

Dynamic programming (DP) is probably the most popular programming technique in bioinformatics. It is used for the many variations of sequence comparison and alignment [12, 23, 33], for DNA fragment assembly [1], RNA folding [24, 39], structure comparison [32], gene recognition [8, 34] and a multitude of further applications. For a recent overview of basic and advanced techniques, see [2] and therein [9]. In some research contexts, a large number of such programs must be developed; the *Dynamite* system [5] has been designed to support the *implementation* of sequence comparison algorithms based on dynamic programming. The present article is devoted to the problem of *developing* DP algorithms in the area of RNA secondary structure determination.

1.2 Sketch of results

We show how the development of dynamic programming algorithms can be done in a completely declarative way, using a conceptual separation of a structure recognition and a structure evaluation phase. In our application context, varying classes of RNA secondary structures are described by regular tree grammars over an algebraic data type $\mathcal{FS}$, which represents structures explicitly. We introduce tabulating yield parsers, which recognize structures efficiently. The evaluation phase is specified separately by an $\mathcal{FS}$-algebra. Via abstraction and instantiation we obtain a (higher-order) functional program for the desired analysis. It serves as an executable specification, and as a template from which the typical DP recurrencies can be derived in a systematic way. The explicit structure representation cancels out, the separation of the two phases becomes invisible, and no overhead in terms of efficiency is incurred.

With respect to RNA folding, our method allows to derive efficient DP solutions for a large and well-defined class of analyses, in a systematic way. The method certainly can be adapted to other application domains of DP in bioinformatics (although this has not been studied yet). Aside from the advantages with respect to program correctness and development time that come along with a declarative approach, our method also provides re-use of algorithms. While the derived DP-programs are monolithic, their specifications consists of two separate phases. g tree grammars over $\mathcal{FS}$ and l $\mathcal{FS}$-algebras can be combined to $g * l$ different analyses. This should be beneficial to bioinformatics research that includes substantial explorative programming.

1.3 Related work

The advantages of a declarative approach to biosequence analysis have been explicated most influentially by Searls [29–31]. In his recent review [28], Searls discusses different approaches to the gene prediction problem in the presence of introns. Although gene structure can be described by a context free grammar and hence can be recovered by parsing methods, Searls feels that DP methods such as used in [8, 34] may have an inherent efficiency advantage:

> "Although parsing of context-free languages can be performed with similar efficiency, the problem of examining all parses, where there may be an exponential number of them, may be intractable even so." [28]

This fear of exponential explosion is appropriate as long as we keep the parsing and the examination phase separate. But given the technique developed in this article, the grammar based and the DP approach achieve the same asymptotic effeciency, and should not longer be seen as competing methods. Rather, the declarative approach must be recognized as the explanation of the DP approach on a higher level of abstraction.

Lefebvre has used parsing techniques and attributed context free grammars for RNA folding [20, 21]. This has resulted in a folding program based on energy minimization and parser generation technology. It has been reported to achieve efficiency similar to DP algorithms. Attribute grammars (as used in a simple form by Lefebvre) have taken a long time to get accepted as a declarative description technique in the compiler construction community; in our opinion, they are still too low-level a formalism to be useful as a program development methodology in bioinformatics. By contrast, all the parsing technology required by our approach is completely described in a small section of this article.

The Dynamite system [5] provides a code generation language for dynamic programming recurrencies, while our approach is concerned with their development. Although the system can be of great benefit to the expert, it has not (yet) found the widespread use one might expect. One reason certainly is that the development of correct dynamic programming recurrencies is at least as difficult as their faithful implementation in (say) C. Our approach and the Dynamite system are complementary, although the application domains considered here and in [5] are different.

1.4 Dynamic programming as a programming paradigm

The general idea of dynamic programming is to split a large search space into intermediate stages, whose results are tabulated and re-used many times. This allows to analyse an exponential search space in polynomial time. It takes some creative effort to design the tables of intermediate results, and define the recurrencies that relate their entries. Once the recurrencies are given, the overall algorithm typically consists of a few nested for-loops, in which the recurrencies are embedded. Matrices appear to be the dominant data structure in DP, whereas the individual solutions that make up the search space are not given an explicit representation.

In spite of its importance, and in contrast to paradigms like "Structural Recursion" or "Divide-and-Conquer", "Dynamic Programming" is not a first principle of algorithmics, but rather a composite of three simpler techniques:

1. *structure recognition*, i. e. the construction of the search space of all potential solutions, usually achieved by structural recursion over the input,

2. *evaluation* of individual solutions, usually achieved by structural recursion over the solutions, and a *selection* of solutions based on evaluation results,

3. *tabulation* and re-use of partial evaluation results.

Example 1 Consider the calculation of an optimal pairwise sequence alignment. According to the separation of three techniques, we write a recognizer, which constructs *all* alignments of two sequences, by structural recursion over the two sequences. We write an evaluator, which evaluates each alignment according to a given score function, this time by structural recursion over the alignment, and determines the minimal score. At this point, we would have a declarative approach, directly corresponding to the mathematical definition of an optimal alignment. Since the number of alignments grows exponentially with the length of the sequences, this approach has exponential execution time.

The only, albeit essential contribution that tabulation brings about is that all intermediate results – i.e. the alignments of all pairs of prefixes of the two sequences – are evaluated just once. This guarantes polynomial runtime. In fact, an explicit presentation of the prefix alignments need not even be calculated, since their scores are all that needs to be stored. □

The essence of these observations is that in the final implementation of a dynamic programming algorithm, the logic that lies behind it has been obscured for the sake of efficiency. This may not seem much of a problem for a simple task like pairwise alignment. But in general, should we do program development in such obscurity?

1.5 RNA folding via dynamic programming

RNA folding programs based on energy minimization have been around since the initial work of Zuker [37, 39]. In spite of all their limitations (related to poorly defined energy minima, inability to detect non-planar structures, effects of kinetic folding, switching phenomena in RNA), they have become widely used tools for RNA structure exploration. While there are programs based on simulated annealing [27], genetic algorithms [13], and random helix sampling [22], the classical approach is based on DP. A recent introductory exposition is given in [35]. Two widely used implementations, MFOLD [37] and the VIENNA RNA package [16] offer RNA folding via energy minimization, and further related functionality.

These folding programs can also calculate suboptimal structures, but the possibly interesting structures are drowned in an ocean of structures that differ only trivially. This severely limits the use of folding programs for detailed analysis. For example, Giegerich et al. have recently developed the paRNAss tool for the prediction of conformational switching in RNA [11]. Due to the lack of techniques to test specific hypotheses about the folding space, a sampling procedure was adopted, which is very expensive computationally.

The approach developed here will allow – among other kinds of analyses — to do energy minimization over very specific subclasses of all possible foldings of a given RNA molecule.

1.6 Reading this article

Since our approach is presented here for the first time, we try to give a complete account of both its algorithmic background and its potential applications. The structure of this article supports three ways of reading. In order to address readers both from a biology and a computer science background, our technique is developed gradually, by means of small changes to a realistic example application.[1] However, readers mainly interested in new variants of the RNA folding problem that can be tackled with our approach may skip Section 3 and Section 6. On the other hand, for readers mainly interested in programming methodology, Sections 3 and 6 contain the core of our approach.

2 Tree Grammars for RNA Secondary Structure

2.1 Notation

Let **x** be an RNA sequence given as a list or an array of characters, indexed by 1 through n. Characters are taken from the alphabet $\{a, c, g, u\}$, denoting the four bases adenine, cytosine, guanine and uracil. We adopt the view that indices also denote boundaries between subwords, in the form of $_0x_1x_2...x_n$. This view avoids fiddling with $+1$ or -1, as subwords of **x** are designated

[1] This may lead to some redundancy, but we feel that it is important that all the specifications given here, including the intermediate steps, are complete (and executable).

by their boundaries. If `x = "justice"`, then `"just"` is subword (0,4), and `"ice"` is subword (4,7). The length of subword `(i,j)` is `j - i`, and the concatenation of subwords `(i,j)` and `(j,k)` is subword `(i,k)`.

A convenient notation for higher order functions is essential for our approach. Well-known notations such as the λ-calculus or LISP are inappropriate due to their lack of support for data structures. We therefore borrow some notation from a modern functional language, Haskell [4, 17]. We shall avoid any Haskell notation that goes beyond algebraic data types, lists, and equations defining higher order functions. We shall present the development of non-trivial specifications in successive stages; lines marked `>` are part of a final (executable) specification.

2.2 An algebraic data type representing RNA structures

RNA secondary structure results from hydrogen bonding between base pairs $(c - g)$, $(a - u)$, and $(g - u)$, and from energetically favourable base pair stacking. We use an algebraic data type $\mathcal{FS}$ for the representation of RNA secondary structures. $\mathcal{FS}$ stands for Folding Space, as we use it to represent the potential foldings of a given RNA molecule. A structure generally consists of a list of structural components, where each component is either a single strand (SS), a hairpin loop (HL), a stacking region (SR), a left or right bulge (BL, BR), an internal loop (IL) or a multiloop (ML). By definition, components HL, SR and ML include their closing base pair.

Definition 2 An algebraic data type $\mathcal{FS}$ is defined as follows:

```
> type Base   = Char
> type Region = (Int,Int)

> data  FS  = STRUCT [Component]                                deriving Show
> data  Component =
>           SS  Region                      |
>           HL  Base      Region     Base   |
>           SR  Base      Component  Base   |
>           BL  Region    Component         |
>           BR  Component Region            |
>           IL  Region    Component  Region |
>           ML  Base     [Component] Base                       deriving Show
```

□

Notation: In our notation, constructors and type names (and only these) start with a capital letter. `[ t ]` denotes a list type with element type `t`. Lists are assumed to be predefined. List constructors are `[ ]` (empty list), and `(:)` (attaching an element at the head of a list). The notation `[1,2,3]` is a shorthand for `1:2:3:[ ]`

In the theory of programming, a data type like $\mathcal{FS}$ is called algebraic, because it defines a language of formulas which denote its elements, e.g.

```
> s1 = STRUCT [SS (0,3),
```

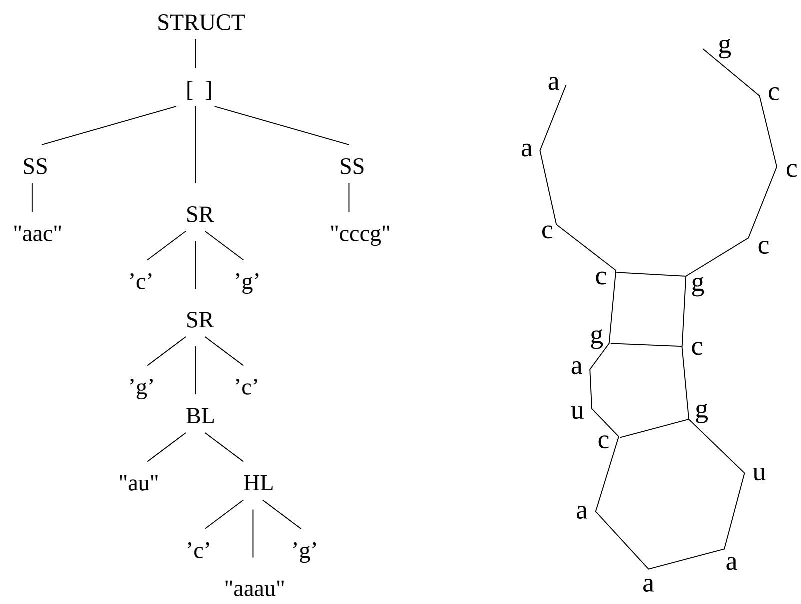

Figure 1: A structure drawing for "aaccgaucaaaugcgcccg"

```
>              SR 'c' (SR 'g'(BL (5,7) (HL 'c' (8,12) 'g')) 'c') 'g',
>              SS (15,19)]
```

Structures can be visualized as trees or drawings as in Figure 1.

The *yield* of a tree is the sequence of its leaves, in left-to-right order. The yield function is defined via structural recursion:

```
> yield  (STRUCT es)  = concat (map yield es) where
>   yield (SS r)      = subword r
>   yield (HL a r b)  = [a] ++ subword r ++ [b]
>   yield (SR a e b)  = [a] ++ yield e ++ [b]
>   yield (BL r e)    = subword r ++ yield e
>   yield (BR e r)    = yield e ++ subword r
>   yield (IL l e r)  = subword l ++ yield e ++ subword r
>   yield (ML a es b) = [a] ++ concat (map yield es) ++[b]
```

Notation: This example introduces almost all the Haskell notation we shall need. Function application is generally written `f x y` rather that `f(x,y)`. The symbol `where` opens a local scope (delineated by indentation). `[a]` denotes a list with a single element, `++` appends two lists, `concat` concatenates a list of lists, and `map f l` applies function `f` element-wise to the list `l`. A function `subword` is assumed such that `subword(i,j)` yields the actual character representation of the indicated subword.

Definition 3 Let s be a value of type $\mathcal{FS}$, and x be a sequence of bases. Then s is a structure for x iff $yield(s) = x$. $\square$

While it is clearly possible to represent all planar RNA structures in $\mathcal{FS}$, the converse is not true. There are some values which we would not accept as biologically sensible (sub)structures. Three examples are

```
> s2  = STRUCT [SS (0,2), SS (2,4)]
> e1  = BL (0,2) (BR ( HL 'c' (3,5) 'g' ) (6,7))
> e2  = ML 'a' [SS (1,3), HL 'c' (4,7) 'g', SS (8,10)] 'u'
```

In s2, the two adjacent single strands should be combined into a single one. In e1, the left and adjacent right bulge should combine to an internal loop. Also, the hairpin loop should contain at least three bases. Structure component e2 is not rightfully called a multiloop; as such it should be branching, which requires at least two components which are not single strands. Similar, but correct structures are

```
> s2'  = STRUCT [SS (0,4)]
> e1'  = IL (0,2) (HL 'c' (3,6) 'g') (7,8)
> e2'  = ML 'a' [SS (1,3), HL 'c' (4,7) 'g', SS (8,10), HL 'a' (11,14) 'u'] 'u'
```

We conclude that the algebraic data type $\mathcal{FS}$ is too simple to represent *exactly* the biologically sensible structures. We have to resort to more expressive description techniques.

2.3 A regular tree grammar for all feasible structures

RNA structures (excluding pseudoknots) can be described by context free grammars. Such a description is used in [20, 26, 28]. Structures, then, are the derivation trees resulting from parsing an RNA sequence [20, 21]. Context free grammars clearly define the language of strings derived by the grammar, but the language of derivation trees is implicit. As we shall see, it is much more convenient to use tree grammars instead of string grammars to describe languages of RNA secondary structures.

Definition 4 A *(regular) tree grammar* is a context free grammar, where the righthand sides of the productions are trees. These trees are formed by the constructors of an underlying algebraic data type, and from nonterminal grammar symbols. The nonterminal symbols are only allowed at the leaves. The language $\mathcal{L}(\mathcal{G})$ of a tree grammar $\mathcal{G}$ is the set of all trees (or formulas) of the underlying algebraic data type that can be derived from the axiom. The yield language of a grammar is defined by $\mathcal{Y}(\mathcal{G}) = \{yield(t) \mid t \in \mathcal{L}(\mathcal{G})\}$.
□

The tree grammar $\mathcal{G}_{feasible}$ introduces the following syntactic refinements over the type $\mathcal{FS}$.

1. Lists of components must not contain two adjacent single strands.
2. We distinguish open structure components from those closed by a base pair. We ensure that bulges and internal loops are separated by at least one base pair.
3. We ensure that the component list of a multiloop contains at least two closed substructures.
4. The region of HL must have at least 3 bases.
5. Bases must form legal base pairs with HL, SR, and ML.

Restrictions (1) – (3) are directly modeled by the grammar $\mathcal{G}_{feasible}$. Restrictions (4) and (5) could be modelled within the productions of the grammar, but it would be cumbersome to do so. They are more conveniently expressed by predicates `match` and `minloopsize` to be associated with the corresponding productions.

```
> pair ('a','u') = True
> pair ('u','a') = True
> pair ('u','g') = True
> pair ('c','g') = True
> pair ('g','c') = True
> pair ('g','u') = True
> pair ('u','g') = True
> pair ( x , y ) = False

> match       inp (i,j) = i+1<j && pair (inp!(i+1), inp!(j))
> nomatch     inp (i,j) = not (match inp (i,j))
> minloopsize k   (i,j) = (j-i >= k)
```

Definition 5 The grammar $\mathcal{G}_{feasible}$ is defined as follows:

1. The underlying data type is $\mathcal{FS}$ with its constructors `HL, SR, ML, BL, BR, IL, STRUCT, :, [ ]`.
2. Nonterminal symbols are { `struct`, `components`, `closedcomponents`, `closed`, `open`, `ml_components`, `ml_closedcomponents`, `ml_components1`, `ml_closedcomponents1`}
3. Terminal symbols are {`base`, `region` }
4. The axiom is `struct`.
5. The productions of $\mathcal{G}_{feasible}$ are given in Figure 2.

$\mathcal{L}(\mathcal{G}_{feasible})$ is the language of feasible RNA structures. □

It is easy to verify that `s1` is an element of $\mathcal{L}(\mathcal{G}_{feasible})$, while `s2, e1, e2` are not.

2.4 The descriptive power of regular tree languages

Formal language theory teaches us [6] that languages defined by regular tree grammars share the properties of regular languages — they are closed under union, intersection and complement[2]. For example, we may design a grammar that describes only cloverleaf structures (Section 7.3) – as well as one that comprises all structures *except* cloverleaves. It seems that with regular tree grammars, we have all the descriptive freedom we can think

[2]This is yet another reason not to use string grammars to describe RNA structure. Context free string grammars are required to describe base pairing. But context free languages, as is well known, are neither closed under complement, nor under intersection.

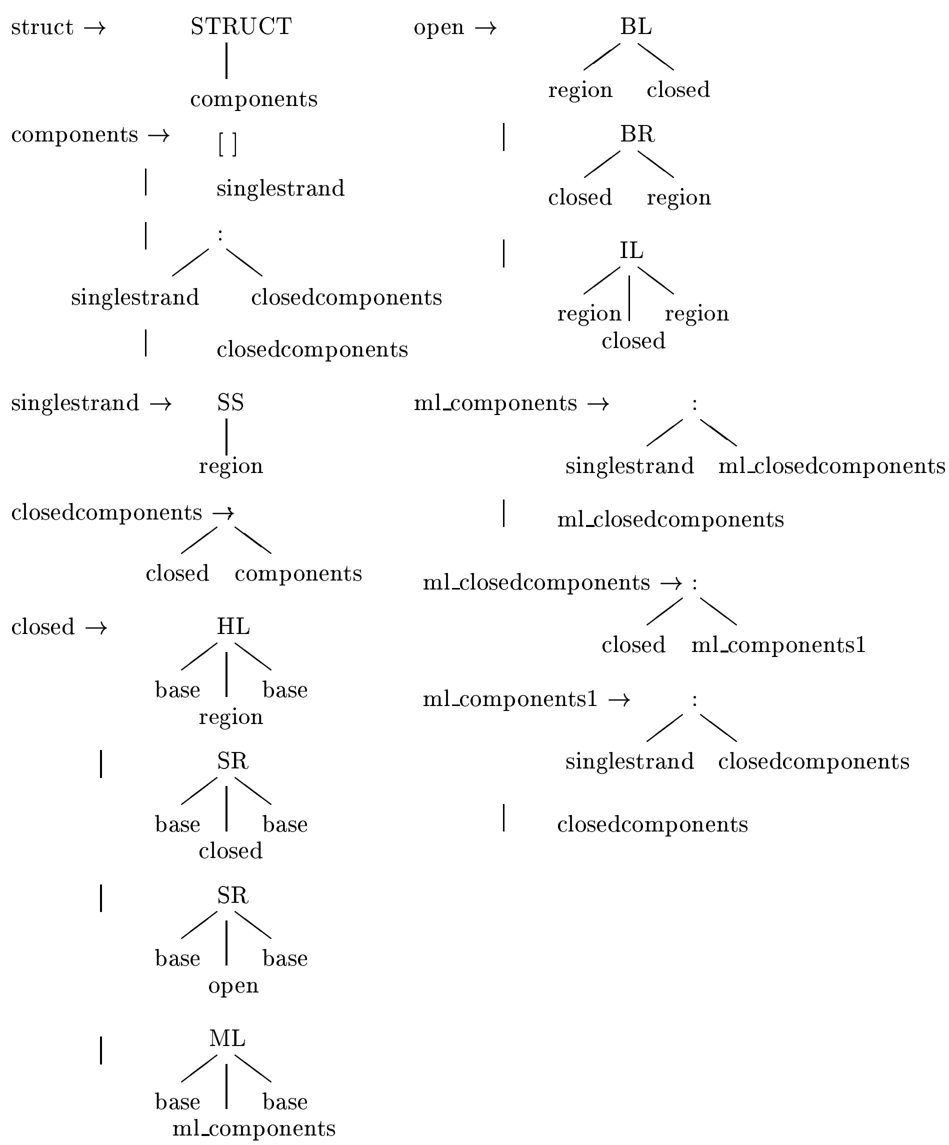

Figure 2: Productions of Grammar $\mathcal{G}_{feasible}$

of. And of course, the more specific our descriptive mechanism is, the more specific questions about RNA structures we may ask.

The standard parsing problem for tree grammars takes a *tree* as its input and determines its derivations according to the grammar. This has applications in compiler code generation, mapping expression trees to machine instructions [3, 10]. However, the efficient parsing technology that has been derived in that context cannot be used here: Our problem at hand takes the *yield of a tree* as its input, and asks for all possible trees for this yield. We call this the *yield parsing problem*.

3 Tabulating Yield Parsers

This section introduces tabulating yield parsers, which efficiently solve the yield parsing problem by dynamic programming. The way in which this is achieved is central to our approach, but readers who are more curious about RNA folding might jump ahead to the next chapter, returning later to the aspects of parser implementation and efficiency.

3.1 An EBNF-style notation for tree grammars

Extended Backus-Naur Form (EBNF) is a convenient notation for context-free string grammars, most widely known for its use in the definition of programming language syntax. EBNF-expressions are built from terminal and nonterminal symbols, juxtaposition, and operators for alternative and iterated constructs. We develop an EBNF-like notation for tree grammars, where we use the operator `~~~` and its variants `+~~` and `~~+` for juxtaposition, `|||` for alternatives, and `<<<` for the application of a tree constructor to its arguments.

For convenience, we assign priorities to these infix operators, highest for `<<<` and lowest for `|||`. Also, `~~~` and its variants associate to the left, while `|||` associates to the right. Nonterminal symbols are written in lower case, while the tree constructors of the underlying data type start with a capital letter. The start symbol of the grammar is prefixed by the word `axiom`.

Let us consider a toy subset of $\mathcal{G}_{feasible}$, essentially two productions for the nonterminal symbol `closed`, plus another one for `open` to make the language non-empty. See Figure 3.

In our notation, these productions are written like this:

```
closed  =  HL <<< base ~~~ region ~~~ base |||
           SR <<< base ~~~  open  ~~~ base

open    =  BR <<< closed ~~~ region
```

Further syntactic restrictions can be associated by a `with`-clause, either with a symbol, or a larger expression on the righthand side. Our example now reads

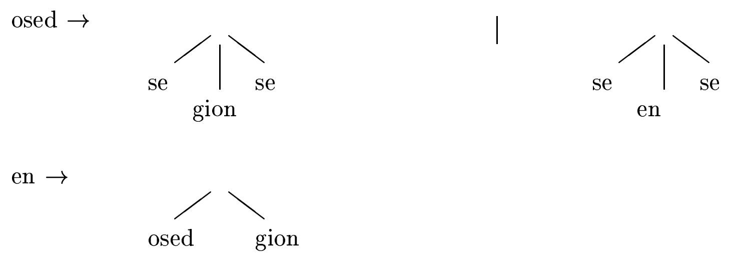

Figure 3: Toy excerpt from $\mathcal{G}_{feasible}$

```
closed  = (HL <<< base ~~~ (region `with` minloopsize 3) ~~~ base |||
           SR <<< base ~~~  open                          ~~~ base)
           `with` basepair

open    = BR <<< closed ~~~ region
```

where `minloopsize` checks the length of the `region`, and `basepair` checks that the enclosing bases can actually pair. Note that this check applies to both alternatives of the production.

We now turn to the problem of recognizing the yields of a tree grammar. At the end of this chapter, the above grammar description itself (with minor notational amendments) will prove to be an efficient yield parser.

3.2 Yield parsers and their combinators

Definition 6 The *yield parsing problem* for a tree grammar $\mathcal{G}$ is the following: Given a string y of terminal symbols, find all trees $t \in \mathcal{L}(\mathcal{G})$ such that $yield(t) = y$. $\square$

Combinator parsing [18, 19] is an elegant technique which directly turns a context free grammar into a nondeterministic, top-down parser. We adjust this technique to the yield parsing problem. We need to impose one restriction on $\mathcal{G}$: A grammar is well-formed, if there is no circular chain of productions that does not contribute symbols to the yield. Such a chain would imply an infinite number of parses for some string, and an infinite loop in the parser. These chains can be avoided easily.

A parser is a function that given a subword `(i,j)` of the input `inp`, returns a list of all its parses.[3] If the parse fails, this list is empty. Each parser parses for a specific nonterminal or terminal symbol, hence the result type of the parser varies. In other words, the type `Parser` is polymorphic. In the following type definition, `b` is a parameter standing for an arbitrary element type in the result list of a parser.

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

[3]In contrast to the string parsers in [18], our parsers must completely consume the subword; no unparsed input suffix is returned.

First we define some elementary parsers:

`anyChar` recognizes any subword of length one.

```
> anyChar           :: Array Int Char -> Parser Char
> anyChar inp (i,j) =  [inp!j | i+1==j]
```

Notation: The definition of `anyChar` uses a list comprehension and is to be read as follows: if the subword holds just one Character, `anyChar` returns it as the (only) result; otherwise it returns an empty result list.

Next we define two recognizers for parsing the empty yield into an empty list (of arbitrary element type), and for parsing an arbitrary, non-empty region, returning its subword boundaries as the result:

```
> p_empty         :: Parser [b]
> p_empty (i,j)   =  [[] | i == j]

> p_region        :: Parser (Int,Int)
> p_region (i,j)  =  [(i,j) | i < j]
```

The nonterminals of our grammar will now be interpreted as parsers, the productions serving as their mutually recursive definitions. To achieve this, the operators of our EBNF-like notation are given a meaning as parser combinators, i. e. higher order functions that combine complex parsers from simpler ones. `|||` concatenates result lists of alternative parses, and `<<<` grabs the results of subsequent parsers connected via `~~~` and sucessively "pipes" them into the tree constructor[4].

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

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

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

Notation: The definition of `<<<` relies on the fact that all our multi-argument functions are curried: For example, the three-argument function `SR`, when given a single argument `a`, returns the function `(SR a)` which expects to consume two further arguments , say `r` and `b` to ultimately yield the result `SR a r b`. Correspondingly, the combinator `~~~` assumes that the results of its first argument, the parser `r`, are functions that expect to consume the results from the second argument, parser `q`. The righthand side of `~~~` is defined via a list comprehension, and reads in words: That list of all `f` applied to all `y`, such that `f` results from parser `r` applied to subword `(i,k)`, `y` results from parser `q` applied to subword `(k,j)`, and `k` ranges from `i` through `j`.

The operational meaning of a `with`-clause can also be conveniently defined by turning `with` into a combinator, this time combining a parser with a filter.

[4]Readers familiar with [18] will notice that `<<<` is our version of the `using`- combinator. It avoids the inconvenient tupling of intermediate parser results and (un-)currying tree constructors.

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

Notation: The last equation reads: The parser `with p c` applied to subword `(i,j)` returns the result of `p (i,j)` if the subword `(i,j)` satisfies condition `c`; otherwise it fails.

Finally, the keyword `axiom` of the grammar is interpreted as a function that returns all parses for the startsymbol (`q`) and a prefix of length `n` of the input.

```
> axiom         :: Int -> Parser b -> [b]
> axiom n q     =  q (0,n)
```

With a few changes, the grammar for toy structures now turns into a parser: The grammar name becomes a function, with the input array as its argument. The parser has a little prolog, which determines the length of the input and directs the parsers for terminals to the input. The productions of the grammar remain unchanged.

```
> toy_structure inp = axiom n closed  where
>    (_,n)          = bounds  inp
>    p_base         = anyChar inp
>    basepair       = match   inp

>    closed         = (HL <<< p_base ~~~ (p_region `with` minloopsize 3) ~~~ p_base |||
>                      SR <<< p_base ~~~  open                           ~~~ p_base)
>                     `with` basepair

>    open           = BR <<< closed ~~~ p_region
```

This version of the grammar can be executed as a parser, albeit a highly inefficient one.

3.3 Tabulating (deterministic) yield parsers

In general, a parser like `toy_structures` developed so far will run effeciently only for grammars where the parse proceeds in a deterministic fashion. For ambiguous grammars, the parser will generally take exponential time, as the parse of a subword is repeatedly constructed in the course of different derivations.

The remedy is tabulation: For a nonterminal `X`, we introduce an $(n+1)$ by $(n+1)$ array `X`. Whenever a subword `(i,j)` is parsed into a nonterminal `X`, the results are tabulated as table entry `X!(i,j)`. The operator `!` is used to distinguish the call to a tabulated function — i. e. array indexing — from a normal function call `X(i,j)`.

Alternative derivations then re-use the results of the parse rather than reconstructing it. This is the general idea of the dynamic programming paradigm: By storing and re-using partial results, we invest polynomial space for avoidance of exponential runtime.

In our framework, this step is amazingly simple: we introduce two functions `table` and `p`. `table n q` records the results of parser `q` for all subwords of an input of length `n` in a parser table of suitable size. Conversely, `p t (i,j)` looks up the results stored in parser table `t` for subword `(i,j)`. Note that the expression `p t`, where `t` is a parser table, is itself of type `Parser b`.

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

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

Notation: The function `array` creates an array with the index range as given by its first argument. It initializes the array elements according to the index-value pairs in the second argument. Note that we leave the array undefined for `j<i`.

Some parsers only do a constant amount of work, e.g. for chain productions like `X = Y`, or for `X = c <<< Y`, where the righthand side only applies a unary constructor to the results of some other parser. In such cases, it is not worthwhile to store the results of these parsers in a table. We introduce the following convention: A parser for nonterminal `X` will either be written `p_X`, in which case it will be defined as a parsing function as before. Else, it will be written `p X`, in which case `X` is a parse table for nonterminal `X` and `p` is the table lookup function defined above. The tabulating yield parser for toy structures now takes the form

```
> toy_structure' inp = axiom n (p closed) where
>    (_,n)           = bounds  inp
>    p_base          = anyChar inp
>    basepair        = match   inp
>    tabulated       = table n

>    closed  = tabulated
>             ((HL <<< p_base ~~~ (p_region `with` minloopsize 3) ~~~ p_base |||
>               SR <<< p_base ~~~  p open ~~~ p_base) `with` basepair)

>    open    = tabulated
>             (BR <<< p closed ~~~ p_region)
```

Annotation by the clause `tabulated` and the consistent distinction of tabulated and nontabulated parsers via the `p X` resp. `p_X` convention is all we need to obtain a parser with polynomial efficiency.

3.4 Asymptotic efficiency of tabulating yield parsers

The asymptotic worst-case efficiency of tabulating yield parsers is affected by the size of the input, the size of the output, and the complexity of the grammar. An input of length n has $O(n^2)$ subwords to be parsed. Let s be the number of nonterminals in the tree grammar. Let us assume that for all nonterminals X where the parsing effort for a given subword is not bounded by a constant, the tabulating implementation is chosen in the yield parser. Let there be t tabulated nonterminals. We assume that each element in a parser's result list occupies constant space, using pointers to its components rather than copying them. Let l be the maximal number of elements in the result list of a parser. Well-formedness of the grammar ensures that l is finite.

The space requirement for the parser is $O(t * n^2 * l)$. Hence, the space requirement is polynomial in the size of the input and the grammar. It is linear in the size of the answer. This implies, of course, that we actually need exponential space if there is an exponential number of parses. But note that l turns into a constant when only a yes/no answer or a bounded number of parses is returned. In this case, the overal space requirements reduce to $O(t * n^2)$.

The essential observation about time complexity is that each occurence of the `~~~`-operator in the righthand side increases the the exponent of n by 1. This is because `p ~~~ q` iterates over all possible boundaries between the subwords recognized by either parser. This is necessary for nonterminals that can actually derive yields of unbounded length. On the other hand, for nonterminals which can derive only a fixed-length yield, most of this iteration will be a futile effort, and it can be avoided by modified variants of the `~~~`-combinator. The same trivially holds with respect to terminal symbols. We will show how to implement this idea in Section 3.5.

Let the *width* of a tree production be defined by the number of nonterminals on its righthand side which produce an unbounded yield. Let the width of a tree grammar be the maximal width of its productions. Let w be the width of the grammar, and again let us assume that constant effort is needed to apply a tree constructor when piecing together a result. Then, the worst-case time complexity of a tabulating yield parser is $O(t \ * n^{w+1} * l)$.

In fact, the exponent $w + 1$ can be fixed to 3 at the expense of increasing t. We sketch the general technique by means of an example: Let `u,v,w` be tree parsers, tabulated or not. We give two equivalent definitions for a parser `x` built from `u,v,w` and tree constructor `c`.

```
x = c  <<< u ~~~ v ~~~ w

x = c' <<< p z      ~~~ w  where
    z            = tabulated (combine <<< u ~~~ v)
    c' (u,v) w   = c u v w
    combine u v  = (u,v)
```

Thus, by introducing an auxiliary nonterminal `z`, a production for `z` deriving `u ~~~ v` and using an auxiliary constructor `combine`, and a tabulating parser for `z`, the width of the production for `x` is reduced to 2.

Summing up we may state: A yield parser for a well-formed and width-reduced tree grammar $\mathcal{G}$ is implemented by our technique in $O(t{*}n^2{*}l)$ space and $O(t * n^3 * l)$ time, where t is bounded by the number of nonterminals, n is the length of the input string, and l is the answer size.

3.5 Yield parser combinators for bounded yields

If the length of the yield of a nonterminal `X` is restricted to a known interval, the `~~`-combinator maybe used to restrict the parsing effort to subwords of appropriate length range.

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

The combinators `+~~` and `~~+` are special cases of the `~~~` combinator in another way: they restrict the lefthand (respectively righthand) parser to a single character.

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

Note in accordance with our discussion in section 3.4 that a parser `q +~~ r` has the same asymptotic efficiency as `r` by itself, and likewise for `r ~~+ q`. The parser for toy expressions may now be written more efficiently:

```
> toy_structure'' inp = axiom n (p closed) where
>    (_,n)          = bounds  inp
>    p_base         = anyChar inp
>    basepair       = match   inp
>    tabulated      = table n

>    closed = tabulated
>            ((HL <<< p_base +~~ (p_region `with` minloopsize 3) ~~+ p_base |||
>              SR <<< p_base +~~  p open ~~+ p_base) `with` basepair)

>    open   = tabulated
>            (BR <<< p closed ~~~ p_region)
```

As can be seen from this example, using `~~+` and `+~~` wherever appropriate eliminates many uses of `~~~` that would otherwise call for width reduction.

4 Folding Space Enumerators

4.1 Tree grammar notation: A short summary

For readers who may have skipped Section 3, we summarize the notation that has been developed: Consider the first and third alternative for nonterminal `closed` in grammar $\mathcal{G}_{feasible}$. These tree grammar productions will now be written by means of the combinators `<<<`, `~~~`, and `|||` in the following form:

```
  closed  =  HL <<< base ~~~ region ~~~ base |||
             SR <<< base ~~~  open  ~~~ base
```

<<< denotes application of a tree constructor to its arguments, which are separated by ~~~, and ||| separates alternative righthand sides. Priorities of these operators are conveniently designed, so we do not need to write parentheses in a simple case like this.

Further syntactic restrictions can be associated by a with-clause, either with a symbol, or a larger expression on the righthand side. Our example now reads

```
closed  = (HL <<< base ~~~ (region `with` minloopsize 3) ~~~ base |||
           SR <<< base ~~~  open                         ~~~ base)
           `with` basepair
```

where minloopsize checks the length of the region, and basepair checks that the enclosing bases can actually pair. Given the operational combinator definitions of Section 3, a grammar in this style can be executed as a parser, although a highly inefficient one. We introduce some effciency annotation in the grammar: The clause tabulated indicates that parser results for a given nonterminal should be stored in a table. Whether or not a nonterminal X is tabulated will henceforth be distinguished in our notation writing p X or p_X, respectively. Now our example grammar reads

```
closed  = tabulated
          ((HL <<< p_base ~~~ (p_region `with` minloopsize 3) ~~~ p_base |||
            SR <<< p_base ~~~  p open                         ~~~ p_base)
           `with` basepair)
```

Such annotations do not affect the language described by a grammar, and when designing or first reading of a grammar, they can be ignored. The same holds for combinators +~~, ~~+, ~~, which are equivalent to ~~~ in the declarative semantics of the grammar. Section 3 shows that any tree grammar $\mathcal{G}$ written in this style can be executed as an efficient recognizer of $\mathcal{Y}(\mathcal{G})$. This concludes our notational summary.

4.2 Folding space enumerators

A *folding space enumerator* is a function that given an RNA sequence, determines all its structures according to a grammar $\mathcal{G}$. With the notation developed in Section 3, our grammars for RNA structures turn into folding space enumerators. The only matter of choice is which intermediate results to tabulate. For $\mathcal{G}_{feasible}$, we will tabulate for all nonterminals except struct and singlestrand. Parsers for terminal symbols never tabulate.

```
> feasibles inp  = axiom n p_struct where
>   (_,n)        = bounds inp
>   tabulated    = table n
>   basepair     = match inp

>   p_struct     = STRUCT <<< p components
>
>
```

```
>   components   = tabulated
>                     (        p_empty                        |||
>                              p closedcomponents             |||
>                     (:[ ])<<< p_singlestrand                |||
>                     (:)  <<< p_singlestrand ~~~ p closedcomponents)
>   p_singlestrand       = SS <<< p_region
>   closedcomponents     = tabulated
>                          ((:)  <<< p closed ~~~ p components)
>   closed               = tabulated
>                        ( (hairpin ||| stack ||| iloop ||| multiloop)
>                                                   `with` basepair)
>     where hairpin      = HL <<< p_base +~~ (p_region `with` minloopsize 3) ~~+ p_base
>           stack        = SR <<< p_base +~~ p closed        ~~+ p_base
>           iloop        = SR <<< p_base +~~ p open          ~~+ p_base
>           multiloop    = ML <<< p_base +~~ p ml_components ~~+ p_base
>   open                 = tabulated
>                        (bulgeleft ||| bulgeright ||| doublebulge)
>     where bulgeleft    = BL <<< p_region ~~~ p closed
>           bulgeright   = BR <<< p closed ~~~ p_region
>           doublebulge  = IL <<< p_region ~~~ p closed ~~~ p_region
>   ml_components        = tabulated
>                        ( (:) <<< p_singlestrand ~~~ p ml_closedcomponents |||
>                           p ml_closedcomponents)
>   ml_closedcomponents  = tabulated
>                        ( (:) <<< p closed ~~~ p ml_components1)
>   ml_components1       = tabulated
>                        ( (:) <<< p_singlestrand ~~~ p closedcomponents |||
>                           p closedcomponents)
>   p_base               = anyChar inp
```

This tree parser constructs seven tables rather than the two we know e.g. from energy minimization [35]. This is because we collect most detailed information about different substructures. These seven tables encode an exponential answer space, but unless we insist on printing out all the answers, this parser only requires $O(n^2)$ space.

4.3 Uniqueness of structures

By a straightforward induction on the derivation length we can show that grammar $\mathcal{G}_{feasible}$ is non-ambiguous: There is just one derivation for each structure in $\mathcal{L}(\mathcal{G}_{feasible})$. By the very same proof, all structures enumerated by the folding space enumerator are unique.

This is a very useful property of the recognition phase, for example when it comes to enumerating or collecting statistics about suboptimal foldings in an energy minimization context. All the more refined recognizers in later sections will share this property.

5 Abstract RNA Folding Space Evaluators

Recalling our decomposition of dynamic programming into simpler techniques as laid out in Section 1.4, we find that so far we have solved the structure recognition and the tabulation problem. The evaluation phase was trivial - we merely built the recognized structures.

5.1 RNA folding space evaluation algebras

We now direct our attention to the evaluation phase. Our approach is applicable whenever an analysis of RNA folding spaces can be cast in the following framework:

Definition 7 An *RNA folding space evaluation algebra* ($\mathcal{FS}$-algebra for short) is given by

1. a data type `Ans` of answers, representing the results of the analysis,
2. a redefinition of the parsers for terminal symbols, to return results of type `Ans`,
3. a specific evaluation function c for each constructor[5] C of $\mathcal{FS}$, calculating answers for a particular construct from the answers for its components, according to the scheme of structural recursion,
4. a list evaluation function `pp` that summarizes or selects from a list of answers.

By convention, the evaluation functions take the names of the corresponding constructors of $\mathcal{FS}$ in lower case letters, `hl` for `HL`, `sr` for `SR`, and so on. We use `ul` (unit list) and `cons` as abstract names for `(:[ ])` and `(:)`. □

We sketch three examples of $\mathcal{FS}$-algebras:

1. *Energy minimization:* `Ans` is the type of real numbers, representing free energy values. `hl, sr` etc. are the energy rules for hairpin loops, base pair stacking, etc. [35], and `pp` is mimimization over a list of energy values.
2. *Counting (sub)structures:* `Ans` is the type `Int`, representing substructure counts. `hl`, `sr` etc. multiply the structure counts of their constituents, and `pp` is summation of counts for alternative structures. This counting $\mathcal{FS}$-algebra will be fully explicated in the next section.
3. *Structure enumeration:* By reversing our point of view, we find that we have been using an $\mathcal{FS}$-algebra for structure enumeration so far, where `Ans =` $\mathcal{FS}$, `hl`, `sr` etc. are the constructors `HL, SR` etc. The function `pp` is the identity on lists of answers, and hence, has not been explicit in our previous exposition.

[5] This includes auxiliary constructors that may have been introduced by width reduction.

5.2 Abstract evaluators

An *abstract (folding space) evaluator* is a structure space recognizer written in terms of an abstract $\mathcal{FS}$-algebra. We abstract from the answer data type, its constructors, and add the application of the abstract list evaluation function `pp`. The latter necessitates the introduction of a new combinator `(...)` (reminiscent of the dot at the end of each production in EBNF).

```
> (...)              :: Parser b -> ([b] -> [b]) -> Parser b
> (...) q pp (i,j)   =  pp (q (i,j))
```

The abstract evaluator for $\mathcal{G}_{feasible}$ is as follows:

```
> c_feasibles inp  = axiom n p_struct where
>   (_,n)          = bounds inp
>   tabulated      = table n
>   basepair       = match inp

>   p_struct       = struct <<< p components ... pp
>   components     = tabulated
>                       (        p_empty                        |||
>                                p closedcomponents             |||
>                       ul    <<< p_singlestrand                |||
>                       cons  <<< p_singlestrand ~~~ p closedcomponents ... pp)
>   p_singlestrand    = ss <<< p_region ... pp
>   closedcomponents  = tabulated
>                       (cons  <<< p closed ~~~ p components ... pp )
>   closed            = tabulated
>                       (((hairpin ||| stack |||iloop ||| multiloop)
>                                                   `with` basepair) ... pp)
>     where hairpin   = hl <<< p_base +~~ (p_region `with` minloopsize 3) ~~+ p_base
>           stack     = sr <<< p_base +~~ p closed        ~~+ p_base
>           iloop     = sr <<< p_base +~~ p open          ~~+ p_base
>           multiloop = ml <<< p_base +~~ p ml_components ~~+ p_base
>   open                = tabulated
>                         (bulgeleft ||| bulgeright ||| p doublebulge ... pp)
>     where bulgeleft    = bl <<< p_region ~~~ p closed
>           bulgeright   = br <<< p closed ~~~ p_region

>           doublebulge  = tabulated
>                          (il <<< p_region ~~~ p closed ~~~ p_region)
>   ml_components        = tabulated
>                          (cons <<< p_singlestrand ~~~ p ml_closedcomponents |||
>                           p ml_closedcomponents ... pp)
>   ml_closedcomponents  = tabulated
>                          (cons <<< p closed ~~~ p ml_components1 ... pp)
>   ml_components1       = tabulated
>                          (cons <<< p_singlestrand ~~~ p closedcomponents |||
>                           p closedcomponents ... pp )
>   p_base               = (anyChar inp) ... pp
```

The above is an abstract program – there is no indication about the answer data type in these lines. A concrete $\mathcal{FS}$-algebra has yet to be specified by giving a meaning to `hl`, `sr`, etc.

5.3 Instantiating an abstract evaluator

We continue the definition of `c_feasibles` by instantiating it with the counting $\mathcal{FS}$-algebra. This includes four steps:

1. We fix the answer data type to `Int`.
2. The primitive parsers are redefined to yield results in the chosen answer data type:

```
>   anyChar inp (i,j) = if i+1 == j then [1] else [ ]
>   p_empty (i,j)     = if i   == j then [1] else [ ]
>   p_region (i,j)    = if i   <  j then [1] else [ ]
```

3. The evaluation functions are defined to multiply counts of structure constituents:

```
>   ul   x      = x
>   cons x y    = x * y

>   struct x    = x
>   ss    x     = x
>   bl l  x     = l * x
>   br    x l   = x * l
>   hl b1 x b2  = b1 * x * b2
>   sr b1 x b2  = b1 * x * b2
>   il l  x r   = l  * x * r
>   ml b1 x b2  = b1 * x * b2
```

4. The list evaluation function `pp` is defined to sum over the counts in a list, returning a unitary list rather than an integer to conform to the overall scheme[6]. Function `sum` is list summation.

```
>   pp          = addup where addup [ ] = [ ]
>                             addup xs  = [sum xs]
```

This completes the definition for the counting evaluator for $\mathcal{G}_{feasible}$. Computing `c_feasibles x` we obtain the number of feasible secondary structures that an RNA sequence `x` can attain according to the given pairing rules. By a modified definiiton of the `axiom`-clause we may also obtain counts for various types of substructures (hairpins, multiloops), or counts for different regions of the input sequence.

5.4 Peephole optimization

For expository reasons, we have been very systematic in our development of the counting evaluator, refraining from a number of obvious optimizations.

[6] Here is a pitfall to be avoided: The mathematically correct and simpler definition `pp = sum` implies `pp [ ] = [0]`. This leads to an exponential number of zeroes that need to be added and multiplied. While the result remains correct, this would break the polynomial complexity of the counting evaluator.

- In the counting $\mathcal{FS}$-algebra, parsers like `p_base` or `p_region` always yield `[1]` or `[ ]`, such that the evaluation functions actually multiply by 0 or by 1 much of their time. To avoid this, we might write equivalently

  ```
  hl b1 x b2      = x
  ```

  since `b1` and `b2` are always equal to `[1]`. Alternatively, we may introduce parser combinators `-~~` and `~~-` which simply drop the result(s) of their left/right argument parser. They are known as `xthen` and `thenx` combinators in Huttons work on string parsing [18]. This seems more elegant, but also results in a more specific recognizer. Defining a parser like `stack` in the form

  ```
  stack       = sr <<< p_base -~~ p closed ~~- p_base
  sr x        = x
  ```

  means that instantiations of this abstract evaluator in *any* $\mathcal{FS}$-algebra will define a unary function `sr`, and will ignore the results of the parser `p_base`.

- Given that most evaluation function are instantiated in the same way, two parsers may actually have equivalent definitions, and one of them may be substituted by the other.

Such opportunities for optimization can be proved correct thanks to the high level of abstraction of our specification.

6 Deriving Dynamic Programming Recurrencies

Down to earth! Finally, let us translate our evaluators, defined in terms of higher order functions, into a set of (first order) dynamic programming recurrencies, which can be implemented easily in an imperative programming language like FORTRAN or C. This should yield a constant (but significant) factor of speedup, compared to directly executing the specification as a functional program in Haskell.

The derivation of dynamic programming recurrencies from a concrete $\mathcal{FS}$-evaluator proceeds as follows:

- Each tabulating parser q gives rise to a matrix Q with an index range from $(0, 0)$ to (n, n).
- The righthand side of its defining equation yields the defining recurrency for matrix elements $Q!(i, j)$
- Each non-tabulating parser turns into inline code.
- The functions of the $\mathcal{FS}$-algebra, as well as the `with`-clauses in the parsers are also inlined.
- The parser combinators are substituted by their definitions. The resulting code is simplified, leading to one recurrency per tabulated parser.

- The `axiom` function turns into a print statement which outputs the desired result.
- The function `table` yields the for-loop code for the initialization of each matrix separately. All these for-loops must be joined, since the recurrencies are mutually recursive.

The last item needs closer scrutiny. A functional specification inludes no evaluation order aside from the data dependencies. As long as these dependencies are not cirular, they can be evaluated in some order, and a lazy programming language like Haskell will choose such an order. Once we translate the dependencies into an eager (imperative) language, we must explicitly organize the computation such that everything is calculated before it is used. It must be emphasized that our approach yields no general technique for this problem.

In the given case, a suitable evaluation order not only exists, it is also easy to find. For $i \leq j$, all dependencies go from subword (i, j) to interior regions, so we calculate all nontrivial matrix entries inside-out, starting from the main diagonal.

This leads to the following loop code:

$$
\begin{array}{l}
\text{for } j = 0 \text{ to } n-1 \\
\quad\quad \text{for } i = j+1 \text{ to } n \\
\quad\quad\quad\quad \text{for each } M \\
\quad\quad\quad\quad\quad M!(i,j) \quad = \quad 0 \\
\quad \text{for } j = 0 \text{ to } n \\
\quad\quad\quad \text{for } i = j \text{ to } 0 \\
\quad\quad\quad\quad \text{for each } M \\
\quad\quad\quad\quad\quad M!(i,j) \quad = \quad \texttt{<<} \text{ see specific recurrence for } M \texttt{>>}
\end{array}
$$

From this framing code, it follows that the assertion $0 \leq i \leq j \leq n$ may be used for simplification when deriving matrix recurrencies.

The grammar $\mathcal{G}_{feasible}$ requires the following seven matrices: *Components*, *Closedcomponents*, *Closed*, *Open*, *Ml−components*, *Ml−closedcomponents*, *Ml − components1*. Their index range is $i \in [0, n], j \in [0, n]$, and their element type is Integer. We shall identify $[x]$ with x and $[\,]$ with 0.

First we derive the code to be inlined for some elementary parsers:

$$
\begin{array}{rcl}
\texttt{p\_base}(i,j) & = & \text{if } i+1=j \text{ then } 1 \text{ else } 0 \\
\texttt{p\_region}(i,j) & = & \text{if } i<j \text{ then } 1 \text{ else } 0 \\
\texttt{(p\_region 'with' minloopsize 3)}(i,j) & = & \text{if } j-i \geq 3 \text{ then } (\text{if } i<j \text{ then } 1 \text{ else } 0) \text{ else } 0 \\
& = & \text{if } j-i \geq 3 \text{ then } 1 \text{ else } 0 \\
\texttt{p\_singlestrand}(i,j) & = & \text{if } i<j \text{ then } 1 \text{ else } 0 \\
\texttt{p\_struct}(i,j) & = & \mathit{Components}!(i,j)
\end{array}
$$

The above equations define expressions to be substituted for the non-tabulating parsers. We now derive the recurrencies for the matrix *Closed*. First, we derive and simplify code for its auxiliary parsers `hairpin`, `stack`, `iloop`, and `multiloop`.

$$
\begin{array}{rcl}
\texttt{hairpin!}(i,j) & = & (\texttt{hl<<<p\_base+\~{}\~{}p\_region `with` minloopsize 3 \~{}\~{}+p\_base})(i,j) \\
 & = & \texttt{p\_base}(i,i+1) * (\text{if } (j-1)-(i+1) \geq 3 \text{ then } 1 \text{ else } 0) \\
 & & * \ \texttt{p\_base}(j-1,j) \\
 & = & 1 * (\text{if } j-i \geq 5 \text{ then } 1 \text{ else } 0) * 1 \\
 & = & \text{if } j-i \geq 5 \text{ then } 1 \text{ else } 0 \\
\texttt{stack!}(i,j) & = & \texttt{p\_base}(i,i+1) * \mathit{Closed}!(i+1,j-1) * \ \texttt{p\_base}(j-1,j) \\
 & = & \mathit{Closed}!(i+1,j-1) \\
\texttt{iloop!}(i,j) & = & \texttt{p\_base}(i,i+1) * \mathit{Open}!(i+1,j-1) * \ \texttt{p\_base}(j-1,j) \\
 & = & \text{Open}!(i+1,j-1) \\
\texttt{multiloop!}(i,j) & = & \texttt{p\_base}(i,i+1) * \mathit{Ml\_components}!(i+1,j-1) * \ \texttt{p\_base}(j-1,j) \\
 & = & \mathit{Ml\_components}!(i+1,j-1)
\end{array}
$$

Finally, we obtain the overall recurrence for matrix $Closed$:

$$
\begin{array}{rcl}
\mathit{Closed}!(i,j) & = & \text{if pair}(\text{inp}!i+1, \text{inp}!j) \text{ then} \\
 & & \ \text{addup}[\ \texttt{hairpin}(i,j),\ \texttt{stack}(i,j),\ \texttt{iloop}(i,j),\ \texttt{multiloop}(i,j)] \\
 & & \text{else } 0
\end{array}
$$

which simplifies to

$$
\begin{array}{rcl}
\mathit{Closed}!(i,j) & = & 0, \ \text{if not pair}(\text{inp}!(i+1), \ \text{inp}!j) \\
 & = & \text{if } j-i \geq 5 \text{ then } 1 \text{ else } 0 \\
 & & + \ \mathit{Closed}!(i+1,j-1) + \mathit{Open}!(i+1,j-1) \\
 & & + \ \mathit{Ml\_components}!(i+1,j-1), \ \text{otherwise}
\end{array}
$$

It is easy to prove that $Closed!(i,i) = 0$ for all i, which is useful in the further derivations.

For the other six matrices, we derive the following recurrencies:

$$Components!(i,j) = 1 + \sum_{k=i}^{j} Closedcomponents!(k,j)$$

$$Closedcomponents!(i,j) = \sum_{k=i+1}^{j} Closed!(i,k) * Components!(k,j)$$

$$Open!(i,j) = \sum_{k=i+1}^{j-1} Closed!(k,j) + \sum_{k=i+1}^{j-1} Closed!(i,k) + \sum_{k=i+2}^{j-1} \sum_{k'=i+1}^{k-1} Closed!(k',k)$$

$$Ml\_components!(i,j) = \sum_{k=i}^{j} Ml\_closedcomponents!(k,j)$$

$$Ml\_closedcomponents!(i,j) = \sum_{k=i+1}^{j} Closed!(i,k) * Ml\_components1!(k,j)$$

$$Ml\_components1!(i,j) = \sum_{k=i}^{j} Closedcomponents!(k,j)$$

The overall result is given by

$$\begin{aligned} \texttt{count\_feasibles inp} &= \text{print } \texttt{p\_struct}(0,\text{n}) \\ &= \text{print } Components!(0,\text{n}) \end{aligned}$$

There is still one fly in the ointment – the equation defining matrix *Open* contains a quadratic term arising from `doublebulge`. This renders the time complexity $O(n^4)$ for the overall computation. Section 3.4 provides a technique to avoid this. We apply a width reduction transformation to the tree grammar production for `doublebulge`. The original definition

```
doublebulge = tabulated (il <<< p_region ~~~ p closed ~~~ p_region)
```

is rewritten by means of an auxiliary tabulating parser `p illeft`, and an extension of the counting algebra by functions `combine` and `il'`.

```
doublebulge   = tabulated ( il'     <<< p illeft ~~~ p_region) where
  illeft      = tabulated ( combine <<< p_region ~~~ p closed)
  combine x y = x * y
  il' x z     = x * z
```

Thus, we have added an additional parser table `illeft` and thereby reduced the width of the overall grammar to 2. Correspondingly, another matrix *Illeft* needs to be calculated, yielding the improved recurrencies.

$$Doublebulge!(i,j) = \sum_{k=i+1}^{j-1} Illeft!(i,k)$$

$$Illeft!(i,j) = \sum_{k=i+1}^{j-1} Closed!(k,j)$$

It is clear that the analysis in Section 3.4 still applies, but the efficiency of the imperative version of the counting evaluator can also be read directly from the recurrencies and the shape of the matrices. Alike its functional counterpart, it executes in $O(t * n^2)$ space and $O(t * n^3)$ time, where $t = 8$.

7 Applications

In this section, we will develop dynamic programming solutions for several problems in the area of RNA folding. We hope to demonstrate to the reader that with the technique introduced in the previous sections, we may find such solutions in a systematic way, and as a matter of hours. We claim that – unless we are extraordinarily gifted mathematicians and programmers – such solutions otherwise take days to develop and weeks to debug.

We shall consider three applications of our programming method:

1. We take a look at the Waterman estimate of all feasible structures of an RNA sequence of length n. This problem is a simplification of what we have seen so far.
2. We will then proceed to grammars more sophisticated than $\mathcal{G}_{feasible}$, which describe more specific (and hence much smaller) languages of structures.
3. We finally show how a motif search program can be constructed with our method.

7.1 Variations of the Waterman estimate

A well-known upper estimate for the number of feasible foldings of an RNA molecule of n residues was proposed and analysed by Waterman and Smith [35, 36]. The Waterman estimate assumes a minimal loop size of 1, and more importantly, abstracts from the concrete primary sequence and the rules of base pairing, assuming that arbitrary two bases can form a base pair. Thus, the Waterman estimate is a function of n only, lending itself to a deeper mathematical analysis [36].

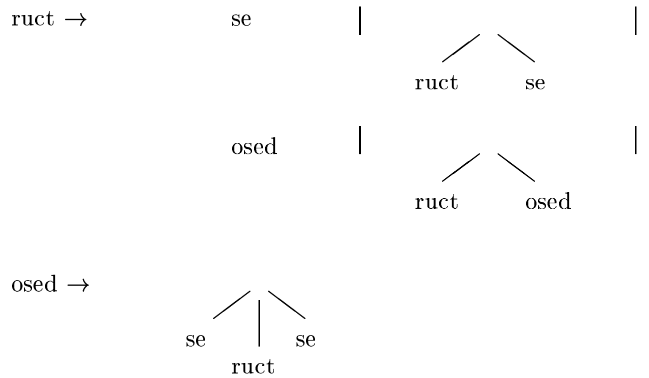

Figure 4: Grammar $G_{estimate}$

A simple grammar to derive the Waterman recurrencies Our analogue of the Waterman approach is a tree grammar that does not care about base pairing and minimal loop sizes. By slight abuse of our earlier interpretation, we use the constructor `BR` for anything composed from two substructures. Atomic structures are either a single base, or three adjacent bases forming a minimal hairpin. Larger structures arise by either adding an unpaired base to a smaller one, or a base that pairs with a partner base somewhere inside the smaller structure. This is reflected in the simple grammar given in Figure 4. The resulting abstract evaluator, instantiated by the counting algebra, is easily obtained by our method.

```
> waterman_estimate inp  = axiom n (p structs) where
>   axiom n q        = q (0,n)
>   (_,n)            = bounds inp
>   tabulated        = table n
>   (~~~)            = (~~) (1,n-1) (1,n-1)

>   structs          = tabulated (
>                         p_base                     |||
>                         br <<< p structs ~~+ p_base |||
>                         p_closed                   |||
>                         br <<< p structs ~~~ p_closed  ... pp)

>   p_closed         =  (sr <<< p_base +~~
>                                (p structs`with` minloopsize 1) ~~+
>                                 p_base) ... pp

>   p_base           = anyChar inp

>   anyChar _ (i,j)  = if i+1 == j then [1] else [ ]
>   sr x y z         = x*y*z
>   br x y           = x*y

>   pp [ ]           = [ ]
>   pp xs            = [sum xs]
```

Since the grammar is left (and right) recursive, the `~~~`-combinator has been slightly redefined, using our `~~`-combinator for bounded yields, as introduced in Section 3.5. Here it requires that each of its argument parsers recognizes at least one symbol. Note that the parsers never inspect the input, except to determine its length n. This property results from the way we define `anyChar`, and is inherited by all other parsers. Hence, the counting evaluator implemented by this grammar is a function of n only.[7]

Applying our deriviation technique of Section 6, we obtain the following recurrencies:

$$\begin{aligned}\texttt{closed}(i,j) = \;& \text{if } j-i<3 \text{ then } 0 \\ & \text{else } \mathit{Structs}!(i+1,j-1)\end{aligned}$$

$$\begin{aligned}\mathit{Structs}!(i,j) = \;& (\text{if } i+1=j \text{ then } 1 \text{ else } 0)+ \\ & (\text{if } i<j \text{ then } \mathit{Structs}!(i,j-1)+\texttt{p\_base}(j-1,j) \text{ else } 0)+ \\ & \texttt{closed}\ (i,j)+ \\ & \textstyle\sum_{k=i+1}^{j-1} \mathit{Structs}!(i,k) * \ \texttt{closed}(k,j)\end{aligned}$$

We substitute for `closed`, use $\texttt{closed}(j-1,j)=0$, and sort out the starting cases:

$$\begin{aligned}\mathit{Structs}!(i,i) &= 0 \\ \mathit{Structs}!(i,i+1) &= 1 \\ \mathit{Structs}!(i,j) &= \mathit{Structs}!(i,j-1)+\mathit{Structs}!(i+1,j-1)+ \\ & \quad \sum_{k=i+1}^{j-2} \mathit{Structs}!(i,k) * \mathit{Structs}!(k+1,j-1) \text{ for } j \geq i+2\end{aligned}$$

By induction on $j-i$, we observe that $\mathit{Structs}!(i,j)=\mathit{Structs}!(i+x,j+x)$ for any $x \geq 0$. Thus we may adjust the latter equation to

$$\begin{aligned}\mathit{Structs}!(i,j) &= \mathit{Structs}!(i,j-1)+\mathit{Structs}!(i,j-2)+ \\ &= \sum_{k=i+1}^{j-2} \mathit{Structs}!(i,k) * \mathit{Structs}!(i,i+j-k-2)\end{aligned}$$

Being interested only in $\mathit{Structs}!(0,n)$, we set $S(m)=\mathit{Structs}!(0,n)$ and obtain

$$\begin{aligned}S(0) &= 0 \\ S(1) &= 1 \\ S(m+1) &= S(m)+S(m-1)+\sum_{k=1}^{m-1} S(k) * S(m-k-1)\end{aligned}$$

This is Waterman's estimate as presented in [35].

[7] This can be verified by a simple exercise: Calculate `waterman_estimate` for an input string of n characters, all of which are undefined.

The Waterman estimate lends itself to mathematical analysis, and explicates the general exponential growth pattern of the number of feasible structures depending on the sequence length. It is not a tight upper bound, however, under the usual set of base pairing rules. For a sequence x of length 3, it accounts for structures `"..."`, `"()."`, `".()"`, and `"(.)"`, where paired bases are indicated by parentheses. There is no sequence `"xyz"` which can achieve the latter three of these. This would require base pairings `x-y`, `y-z` as well as `x-z`, which is impossible. Thus, the Waterman estimate accounts for certain impossible structures, and their number shares the general exponential pattern of growth of the overall formula.

There are two ways to go from here: We can make probabilistic assumptions on base pair matching, obtaining a probabilistic version of the Waterman estimate. We may also add syntactic restrictions to the grammar, to be satisfied by a given sequence, obtaining the count of substructures for a concrete sequence – a formula also given in [38].

A more realistic estimate Given the base composition of a sequence x and set of base pairing rules, it is simple to determine the probability that two independently chosen residues x_i and x_j can form a base pair. The contribution of the rule for nonterminal `closed` is now weighted with this probability. We also return to a minimal loopsize of 3.

```
> prob_waterman_estimate basecomp inp  = axiom n (p structs) where

>   axiom n q        = q (0,n)
>   (_,n)            = bounds inp
>   tabulated        = table n
>   (~~~)            = (~~) (1,n-1) (1,n-1)

>   structs           = tabulated
>                      (p_base                     |||
>                       br <<< p structs ~~+ p_base |||
>                       p_closed                   |||
>                       br <<< p structs ~~~ p_closed  ... pp)

>   p_closed         =  (sr <<< p_base +~~
>                              (p structs`with` minloopsize 3) ~~+
>                               p_base) ... pp

>   p_base           = anyChar inp

>   anyChar _ (i,j)= if i+1 == j then [1] else [ ]
>   sr x y z         = x*y*z* (pair_probability basecomp)
>   br x y           = x*y
>   pp [ ]           = [ ]
>   pp xs            = [sum xs]
```

As a minor contrast to the treatment in [15], our formula takes into account the base composition for the calculation of base pair probabilities. See Section 7.4 for comparing some results from these estimates.

Counting, revisited In [36] Waterman and Smith give a formula for $N^k_{i,j}(x)$, the number of structures formed by exactly k base pairs for subword (i,j) of a *given* RNA sequence x. Summing up $N^k_{0,n}(x)$ over all k, one obtains a count of all feasible structures of the complete sequence x. Zuker and Sankoff [38] and also Hofacker et al. [15] derive their formula directly from the Waterman estimate, as we shall do here.

We obtain a concrete counting formula from the Waterman estimate by including in its grammar checks for minimal loop size and base pairing. All the rest is familiar from $\mathcal{G}_{feasible}$.

```
> waterman_count inp = axiom n (p structs) where
>   axiom n q        = q (0,n)
>   (_,n)            = bounds inp
>   tabulated        = table n
>   basepair         = match inp
>   (~~~)            = (~~) (1,n-1) (1,n-1)

>   structs          = tabulated (
>                       p_base                    |||
>                       br <<< p structs ~~+ p_base |||
>                       p_closed                  |||
>                       br <<< p structs ~~~ p_closed  ... pp)

>   p_closed         =  (sr <<< p_base +~~
>                              (p structs`with` minloopsize 3) ~~+
>                               p_base) `with` basepair ... pp

>   p_base           = anyChar inp

>   anyChar _ (i,j)= if i+1 == j then [1] else [ ]

>   sr x y z         = x*y*z
>   br x y           = x*y
>   bl x y           = x*y
>   pp [ ]           = [ ]
>   pp xs            = [sum xs]
```

Of course, for an arbitrary RNA sequence `x`, the equation `waterman_count x = count_feasibles x` holds, and the two programs can be used to validate each other. Both run with the same time efficiency, but `count_feasibles` uses a few more tables. One virtue of `count_feasibles` is that it allows to report counts also for specific substructures (e.g. multiloops), which are not distinguished from other closed substructures by `waterman_count`. The other virtue of $\mathcal{G}_{feasible}$ is that it allows to direct our attention to more specific (and hence smaller) sets of structures. This is what we built upon in the next section, when we consider canonical and saturated structures.

Waterman and Smith's analysis [36] has been considerably extended by Hofacker et al. [15], where a large number of recurrencies is developed and analysed. Their analysis starts from the Waterman estimate, and gives recurrencies for numbers of stacks, loops, external bases, and so on. In our terminology, all these formulas use simplistic grammars like the one of the

Waterman estimate as their recognizers. Recasting this work in terms of an $\mathcal{FS}$-algebra, together with the grammars introduced in the next section, would give us Hofacker's recursions for canonical and saturated structures – for free.

7.2 Canonical and saturated structures

Canonical structures Tree grammars provide a simple and quite convenient formalism for fine-tuning a tree language describing RNA structures. We demonstrate this first by defining a more restrictive language of *canonical* structures. Let us say that a structure is canonical if it has no isolated base pair. In other words, we impose the rule that base pairs must enjoy stacking, otherwise they will not form. This is motivated by free energy considerations.

Refining grammar $\mathcal{G}_{feasible}$ into $\mathcal{G}_{canonical}$, we split nonterminal `closed` into two variants: `stack` derives structures which are (strongly) `closed` by at least two stacked base pairs, while `closed` derives those structures which are (weakly) closed by a single base pair. Weakly closed structures may now only exist inside strongly closed ones, but must not occur as components in a multiloop.

The productions of grammar $\mathcal{G}_{canonical}$ are given in Figure 3. This is the folding space enumerator for the canonical structures:

```
> canonicals inp = axiom n p_struct  where
>   (_,n)        = bounds inp
>   tabulated          = table n
>   basepair     = match inp
>   p_struct     = STRUCT <<< p components
>   components   = tabulated
>                       (        p_empty                        |||
>                                p closedcomponents             |||
>                       (:[ ])<<< p_singlestrand                |||
>                       (:)   <<< p_singlestrand ~~~ p closedcomponents)

>   p_singlestrand       = SS <<< p_region
>   closedcomponents     = tabulated
>                        ((:)  <<< p stack ~~~ p components)

>   stack                = tabulated
>                        ((SR <<< p_base +~~ ( p stack ||| p closed) ~~+ p_base
>                                                ) `with` basepair)

>   closed               = tabulated
>                        ( (hairpin ||| iloop ||| multiloop)
>                                                 `with` basepair)
>     where hairpin      = HL <<< p_base +~~ (p_region `with` minloopsize 3) ~~+ p_base
>           iloop        = SR <<< p_base +~~ p open          ~~+ p_base
>           multiloop    = ML <<< p_base +~~ p ml_components ~~+ p_base

>   open                 = tabulated
>                        (bulgeleft ||| bulgeright ||| doublebulge)
>     where bulgeleft    = BL <<< p_region ~~~ (p closed ||| p stack)
```

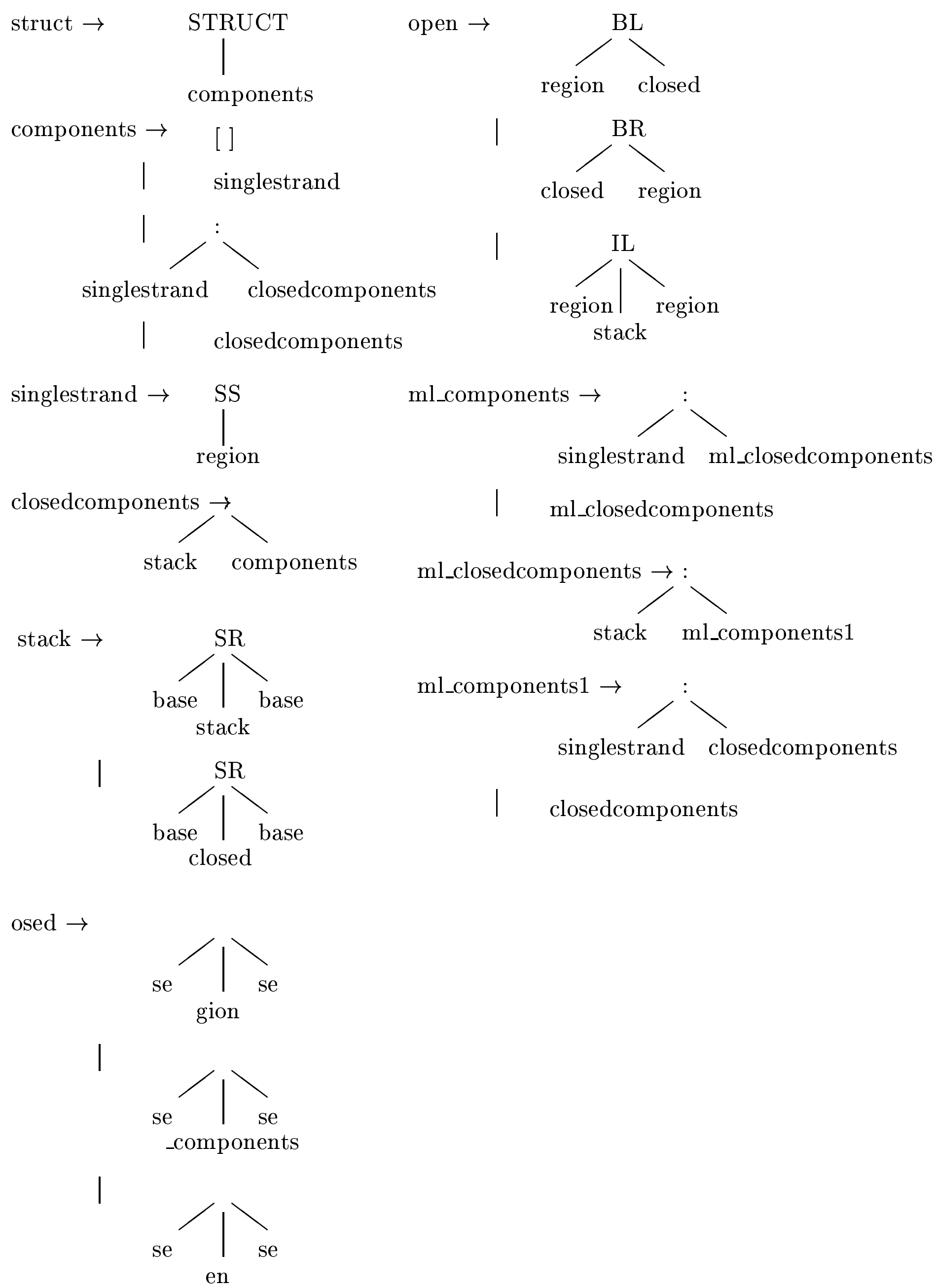

Figure 5: Productions of grammar $\mathcal{G}_{canonical}$

```
>          bulgeright   = BR <<< (p closed ||| p stack) ~~~ p_region
>          doublebulge  = IL <<< p_region ~~~ (p closed ||| p stack) ~~~ p_region

>   ml_components        = tabulated
>                        ( (:) <<< p_singlestrand ~~~ p ml_closedcomponents |||
>                           p ml_closedcomponents)

>   ml_closedcomponents  = tabulated
>                        ( (:) <<< p stack ~~~ p ml_components1)

>   ml_components1       = tabulated
>                        ( (:) <<< p_singlestrand ~~~ p closedcomponents |||
>                           p closedcomponents)

>   p_base               = anyChar inp
```

The abstract evaluator for canonical structures, and its instantiation with the counting $\mathcal{FS}$-algebra, is obtained in the same manner as with feasible structures.

Saturated structures We call a canonical RNA structure *saturated*, when all stacks maximally extend in either direction. This is a much stronger restriction than canonicity, but less frequently applicable. Still, we hope it significantly reduces further the number of structures under consideration.

A first approach at an enumerator of saturated structures works as follows: To check the maximality condition, this parser introduces two private combinators: `suchthat` filters the results of a parser, while `within` checks a condition on the context around a given subword.

We directly specify the enumerator for saturated structures.

```
> saturated inp  = axiom n p_struct where
>   suchthat r f (i,j) = [x |x<-r (i,j),f x]
>   (_,n)              = bounds inp
>   tabulated          = table n
>   basepair           = match inp
>   nobasepair (i,j)   = j - i < 3   || not (basepair (i,j))
>   within r f (i,j)   = [x | 0 == i || j == n || f (i-1,j+1), x <- r (i,j)]

>   p_struct     = STRUCT <<< p components
>   components   = tabulated
>                       (        p_empty                        |||
>                                p closedcomponents             |||
>                       (:[ ]) <<< p_singlestrand               |||
>                       ((:)   <<< p_singlestrand ~~~ p closedcomponents)
>                               `suchthat` maximize )
>   p_singlestrand       = SS <<< p_region
>   closedcomponents     = tabulated
>                        ((:)  <<< p stack ~~~ p components)
>   stack                = tabulated
>                        ((SR <<< p_base +~~ ( p stack ||| p closed) ~~+ p_base)
>                          `with` basepair)
>   closed               = tabulated
>                        ( (hairpin |||iloop ||| multiloop)
```

```
>                                                    `with` basepair)
>     where hairpin      = HL <<< p_base +~~ ((p_region `with` minloopsize 3)
>                                                    `with` nobasepair) ~~+ p_base
>           iloop        = SR <<< p_base +~~ p open          ~~+ p_base
>           multiloop    = ML <<< p_base +~~ p ml_components ~~+ p_base
>   open                 = tabulated
>                          (bulgeleft ||| bulgeright ||| doublebulge)
>     where bulgeleft    = BL <<< p_region ~~~ (p closed ||| p stack)
>           bulgeright   = BR <<< (p closed ||| p stack) ~~~ p_region
>           doublebulge = (IL <<< p_region
>                                 ~~~   (p closed ||| p stack)`within` nobasepair
>                                 ~~~ p_region) `with` nobasepair
>   ml_components        = tabulated
>                          ( ((:) <<< p_singlestrand ~~~ p ml_closedcomponents)
>                                                       `suchthat` maximize |||
>                             p ml_closedcomponents)
>   ml_closedcomponents  = tabulated
>                          ( (:) <<< p stack ~~~ p ml_components1)
>   ml_components1       = tabulated
>                          ( ((:) <<< p_singlestrand ~~~ p closedcomponents)
>                                                        `suchthat` maximize |||
>                             p closedcomponents)
>   p_base               = anyChar inp

>   maximize (SS (i,j): c: SS (i',j'):es) = not (pair (inp!j,inp!(i'+1)))
>   maximize es                           = True
```

While this enumerator looks very similar to the previous one, there is an important difference: The private combinator `suchthat` applies a predicate to a result structure. It is used to check maximality of a stacking region *only* where embedded between two single strands. This presents a problem when instantiating the abstract evaluator in the counting $\mathcal{FS}$-algebra. What should be the argument type of the predicate `maximize`? In fact - the grammar needs to be rewritten to alleviate this situation (and this is the challenge of this example).

The task is to encode *in the grammar* the pattern matching on result structures, that is done by the above definition of `maximize`. Productions for component lists must be refined such that there is a distinguished point where a closed component is embedded between two single strands. In this context, a structure is rejected if the stacking region of the closed components can be extended by base pairing into the adjacent single strands. Note that at least one structure which has this (and possibly further) stack extension remains in the set of recognized structures.

Component lists of the overall structure will now be generated by two nonterminals, `ucomp` and `rcomp`. `rcomp` generates component lists in the left context of a single strand; the others are generated by `ucomp`. Two further nonterminals `sr` and `cu` arise from a width reduction transformation.

Structural components of a multiloop are handled in the same way, with the traditional restriction that a multiloop must contain at least two stacking regions. Multiloop elements are now derived by nonterminal `ml_comps`, which has a very complex righthand side. With width reduction, this leads to 15 nonterminals, for which a specific mnemonic is introduced below.

```
> saturated' inp  = axiom n p_struct where
>   (_,n)        = bounds inp
>   tabulated          = table n
>   basepair     = match inp
>   nobasepair (i,j)   = j - i < 3   || not (basepair (i,j))
>   within r f (i,j)    = [x | 0 == i || j == n || f (i-1,j+1), x <- r (i,j)]

>   p_struct     = STRUCT <<< p components
>   components   = ucomps
>   ucomps       = tabulated
>                       (        p_empty                        |||
>                       (:)  <<< p stack        ~~~ p ucomps    |||
>                       (:)  <<< p_singlestrand ~~~ p rcomps       )
>   rcomps       = tabulated
>                       (        p_empty                                 |||
>                       (:[ ])<<<(p stack)                               |||
>                       (:)   <<< (p stack `within` nobasepair) ~~~ p sr |||
>                       (:)   <<<  p stack                      ~~~ p cu    )
>   sr           = tabulated
>                      ((:) <<<  p_singlestrand ~~~ p rcomps)
>   cu           = tabulated
>                       ((:) <<<  p stack        ~~~ p ucomps)
>   p_singlestrand       = SS <<< p_region
>   stack                = tabulated
>                          ((SR <<< p_base +~~ ( p stack ||| p closed) ~~+ p_base)
>                          `with` basepair)
>   closed               = tabulated
>                         ((hairpin |||iloop ||| multiloop) `with` basepair)     where
>           hairpin      =  HL <<< p_base +~~ ((p_region `with` minloopsize 3)
>                                                    `with` nobasepair) ~~+ p_base
>           iloop        =  SR <<< p_base +~~ p open          ~~+ p_base
>           multiloop    =  ML <<< p_base +~~ p_ml_comps ~~+ p_base
>   open                 = tabulated
>                          (bulgeleft ||| bulgeright ||| doublebulge)          where
>           bulgeleft    =  BL <<< p_region ~~~ (p closed ||| p stack)
>           bulgeright   =  BR <<< (p closed ||| p stack) ~~~ p_region
>           doublebulge  = (IL <<< p_region
>                              ~~~ (p closed ||| p stack) `within` nobasepair
>                              ~~~ p_region) `with` nobasepair
>   p_ml_comps            =            p   sccu ||| p  ccu |||
>                           p  csmsr ||| p  csccu ||| p  csc |||
>                           p smsmsr ||| p smsccu ||| p smsc

    Mnemonic used: s = singlestrand,
                   c = stack,
                   m = stack with maximality restriction
                   r = rcomps
                   u = ucomps

>   p_m                  = p stack `within` nobasepair
>   ccu                  = tabulated ( (:) <<< p stack        ~~~ p cu    )
>   sccu                 = tabulated ( (:) <<< p_singlestrand ~~~ p ccu   )
>   csc                  = tabulated ( (:) <<< p stack        ~~~ p sc    )
>   sc                   = tabulated ( (:) <<< p_singlestrand ~~~ ((:[ ]) <<< p stack) )
```

```
>   csccu              = tabulated ( (:) <<< p stack        ~~~ p sccu  )
>   csmsr              = tabulated ( (:) <<< p stack        ~~~ p smsr  )
>   smsr               = tabulated ( (:) <<< p_singlestrand ~~~ p msr   )
>   msr                = tabulated ( (:) <<< p_m            ~~~ p sr    )
>   smsc               = tabulated ( (:) <<< p_singlestrand ~~~ p msc   )
>   msc                = tabulated ( (:) <<< p_m            ~~~ p sc    )
>   smsccu             = tabulated ( (:) <<< p_singlestrand ~~~ p msccu )
>   msccu              = tabulated ( (:) <<< p_m            ~~~ p sccu  )
>   smsmsr             = tabulated ( (:) <<< p_singlestrand ~~~ p msmsr )
>   msmsr              = tabulated ( (:) <<< p_m            ~~~ p smsr  )

>   p_base             = anyChar inp
```

The grammar has grown considerably more complex. Our reward is that we eliminated the use of the `suchthat`-combinator. Now structures are constructed as they are recognized, but never inspected a posteriori. We may now obtain the abstract evaluator and instantiate it by the counting $\mathcal{FS}$-algebra. This works as before, so it is not shown here.

Even with saturated structures, the number of structures that are possible for a given RNA of length `n` remains exponential in `n`. So we rarely want to see the complete list of all answers of a call to `saturated`. See Section 7.4 for some concrete data. The enumerator `saturated` has been used to create some RNA-movies [7], visualizing the complete (saturated) folding space of a very short RNA.

7.3 Recognition of structural motifs

In the previous examples, we have defined tree grammars that describe essentially all possible structures, subject to slightly different restrictions. Just as well, we may design a grammar to recognize just one motif (a particular substructure), or a set of such motifs. This may be a single hairpin of a certain size, a group of neighbouring hairpins, or a very complex structure. We may write the grammar to allow a certain amount of variation. All those points of variation will vary independently – this means, that with the grammar-based approach, we cannot implement the equivalent of an edit-distance like notion of similarity.

Our model case will be the cloverleaf motif, as it occurs with transfer RNA. We start from the folding space enumerator for saturated structures (first version), and modify it in several respects:

1. We only recognize cloverleaves, in arbitrary positions in the overall sequence, embedded between (possibly empty) single strands. This affects the production for the new nonterminal symbol `clovers`, and adds a new nonterminal symbol `outerss` for singlestrands which may be empty.

2. Cloverleaves are multiloops with k stacking regions branching from them, where $k \geq 3$ is a parameter. The number of branches of a mul-

tiloop `m` is checked by the predicate `arms k m`. This check is included in the production for `clovers` via the `suchthat`-combinator.

3. We disallow further multiloops inside the arms of the cloverleaf. Hence, nonterminal `multiloop` is deleted from the righthand side of the production for nonterminal `closed`.

4. To avoid uninteresting structures, let us impose an upper bound s on the length of bulges and hairpin loops, where $s \geq 3$ is a parameter. This leads to a new nonterminal `rregion` for size-restricted regions.

With these decisions in mind, it is straightforward to write down an enumerator for saturated cloverleaf structures.

```
> sat_cloverleaves k s inp  = axiom n p_struct where
>   (_,n)              = bounds inp
>   tabulated          = table n
>   basepair           = match inp
>   nobasepair (i,j)   = j - i < 3   || not (basepair (i,j))
>   within r f (i,j)    = [x | 0 == i || j == n || f (i-1,j+1), x <- r (i,j)]
>   suchthat r f (i,j) = [x |x<-r (i,j),f x]

>   p_struct             = STRUCT <<< p_clovers
>   p_clovers            = list <<< p_outerss                     ~~~
>                                  (p_multiloop `within` nobasepair)
>                                               `suchthat` arms k ~~~
>                                   p_outerss  where
>     list x y z         = [x,y,z]
>   components   = tabulated
>                       (        p_empty                         |||
>                                p closedcomponents              |||
>                       (:[ ]) <<< p_singlestrand                |||
>                       ((:)   <<< p_singlestrand ~~~ p closedcomponents)
>                                `suchthat` maximize )
>   p_rregion          = p_region `with` maxsize s
>   p_outerss (i,j)    = [SS (i,j)|i <= j]
>   p_singlestrand       = SS <<< p_region
>   closedcomponents     = tabulated
>                        ((:)  <<< p stack ~~~ p components)
>   stack                = tabulated
>                        ((SR <<< p_base +~~ ( p stack ||| p closed) ~~+ p_base
>                                                      ) `with` basepair)
>   closed               = tabulated
>                        ( (hairpin |||iloop) `with` basepair)
>     where hairpin      = HL <<< p_base +~~ ((p_rregion `with` minloopsize 3)
>                                                        `with` nobasepair) ~~+ p_base
>           iloop        = SR <<< p_base +~~ p open          ~~+ p_base
>   p_multiloop          = (SR <<< p_base +~~
>                                  ((ML <<< p_base +~~ p ml_components ~~+ p_base)
>                                  `with` basepair ) ~~+
>                                  p_base) `with` basepair
>   open                 = tabulated
>                        (bulgeleft ||| bulgeright ||| doublebulge)
>     where bulgeleft    = BL <<< p_rregion ~~~ (p closed ||| p stack)
```

```
>          bulgeright   = BR <<< (p closed ||| p stack) ~~~ p_rregion
>          doublebulge = (IL <<< p_rregion
>                             ~~~   (p closed ||| p stack)`within` nobasepair
>                             ~~~ p_rregion) `with` nobasepair
>   ml_components        = tabulated
>                        ( ((:) <<< p_singlestrand ~~~ p ml_closedcomponents)
>                                                   `suchthat` maximize |||
>                          p ml_closedcomponents)
>   ml_closedcomponents  = tabulated
>                        ( (:) <<< p stack ~~~ p ml_components1)
>   ml_components1       = tabulated
>                        ( ((:) <<< p_singlestrand ~~~ p closedcomponents)
>                                                    `suchthat` maximize |||
>                          p closedcomponents)
>   p_base               = anyChar inp

>   maximize (SS (i,j): c: SS (i',j'):es) = not (pair (inp!j,inp!(i'+1)))
>   maximize es                           = True

>   maxsize s (i,j)          = j-i <= s
>   arms k (SR a s b)        = arms k s
>   arms k (ML a cs b)       = arms cs k where
>     arms [ ] n             = (n==0)
>     arms (SS x : ys)     n = arms ys n
>     arms (SR a x b : ys) n = arms ys (n-1)
```

For expository reasons, this is not the most efficient version of the cloverleaf enumerator. Width reduction should be applied to the productions for nonterminals `clovers` and `doublebulge` in order to achieve the usual asymptotic runtime of $O(n^3)$ also for the recognition of cloverleaves.

7.4 Examples from Applications

This paper is devoted to programming methodology. Applications that have been done using our approach will be reported elsewhere. However, some data shall be included here, to illustrate the outcome of the analyses developed in earlier sections.

The stepwise growth of the structure counts in Figure 6 is remarkable, and good correspondence of the probabilistic estimate with the number of feasible structures is not self-understood. These phenomena need to be investigated further.

It is interesting to see where the variation among the saturated structures lies. As it is most cumbersome to scan through a large number of related structures by individual inspection, the tool RNA-Movies has been developed [7]. Among other forms of usage, it allows to visualize the folding space of a given RNA in the form of an animated structure drawing, creating the impression of a molecule exploring its own folding space. The folding space enumerators developed in this article have been used to create several such movie scripts, which will be included in the software distribution of RNA-Movies.

length	Waterman estimate	feasible structs	canon. structs	satur. structs	probabilistic estimate
1	1	1	1	1	1.0
2	1	1	1	1	1.0
3	2	1	1	1	1.0
4	4	1	1	1	1.0
5	8	1	1	1	1.16
6	17	3	1	1	1.83333
7	37	3	1	1	2.52936
8	82	5	1	1	3.42676
9	185	7	1	1	4.55595
10	423	9	1	1	5.98722
11	978	10	1	1	7.80433
12	2283	13	1	1	10.1062
13	5373	23	1	1	19.931
14	12735	50	4	4	47.1037
15	30372	106	11	6	100.822
16	72832	117	11	6	139.23
17	175502	141	11	6	191.08
18	424748	166	11	6	260.745
19	1032004	192	11	6	353.922
20	2516347	390	11	6	510.607
21	6155441	1195	12	7	1189.46
22	15101701	2080	125	68	1685.49
23	37150472	3802	130	71	3651.9
24	91618049	7743	377	215	7569.13
25	226460893	16343	886	493	15160.5
26	560954047	18417	886	493	20991.9
27	1392251012	57708	2067	903	47999.3
28	- out -	100026	8809	3554	74371.5
29	- of -	157473	8963	3559	114574.0
30	- integer -	235025	9249	3639	175550.0
31	- range -	392769	9405	3661	373024.0
32	- - -	959584	24756	8711	797314.0
33	- - -	1571253	63033	14666	1.23524e+06
34	- - -	2316077	64958	14946	1.90449e+06
35	- - -	3964084	66341	15069	4.1482e+06
36	- - -	6258425	66341	15069	6.5104e+06
37	- - -	12414737	70721	16406	1.32957e+07
38	- - -	18828142	281723	66001	2.06058e+07
39	- - -	36275414	293556	68088	4.10018e+07
40	- - -	55611266	682743	158433	6.28615e+07

Figure 6: Structure counts for initial segments of an RNA sequence from *neurospora crassi*, `"gaccauacccacuggaaaacucggg-aucccguccgcucuccca..."`.

8 Conclusions and Future Work

Let us summarize what has been achieved, and then, look at the consequences of this work.

Development of dynamic programming algorithms We have shown that DP algorithms can be derived systematically by introducing a separation between a structure recognition and a structure evaluation phase. This general idea certainly applies to many, if not all problems traditionally tackled by dynamic programming. Pairwise alignment under various cost functions [35] is a simple case in point (mostly an pedagogical exercise), a more challenging one is RNA structure comparison based on the tree edit distance following [32].

In the problem domain of RNA folding, regular tree grammars for structure description, tabulating yield parsers for structure recognition, and folding space evaluation algebras for structure evaluation allow a complex analysis problem to be solved in a completely declarative fashion. The virtues of declarative programming are well known – better understanding due to high level of abstraction, fast development, and high reliability[8] even including techniques for formal validation of desired properties. The recurrencies ultimately derived can be formally cross-checked equation by equation, and the resulting C program may be checked against the functional program directly executing the specification.

Programming economy The conceptual separation of recognition and evaluation phases leads to a modular approach to algorithm development. A particular tree grammar can be combined with different $\mathcal{FS}$-algebras, and vice versa. From $g+l$ specification components (g grammars and l algebras), we can derive $g * l$ analyses.

Exploring RNA secondary structure This paper concentrating on the aspects of program development, we have not reported about specific analyses of RNA folding spaces. Our counting evaluators can certainly be used to collect statistics about frequency of substructures, and it is interesting to see how the numbers diverge for feasible, canonical, saturated, and maybe even more restricted classes of structures. Note that all kinds of statistics about substructures can be extracted from the recognizer by a straightforward change of the meaning of the `axiom`-clause.

---

[8] Practitioners of dynamic programming will have noticed with relief that our approach avoids subscript errors by avoiding subscripts. Subscript errors are always a nuisance, but a particular problem in DP approaches. They may result in occasional suboptimal results that are unlikely ever to be noticed. In algorithm development with our method, all the control structure is taken care of by the tree parser combinators. Only in the final phase when the dynamic prgramming recurrencies are derived by hand, subscript errors may sneek in. This calls for automating the procedure.

Efficiency Our formal method involves no trade-off in terms of ultimate program efficiency. On the contrary, the systematic use of the width reduction transformation on tree grammars garantees cubic complexity in cases where without such guidance, such efficiency might be difficult to achieve.

From formal method to automation? Could we fully automate the derivation of the imperative program from the declarative specification? Certainly to a large extend, if only a straightforward implementation is desired. But note that the derivation sometimes involves the application of partial program evaluation or of mathematical laws (about summation, minimization, or laws holding for the specific $\mathcal{FS}$-algebra). So, a fully automatic tool producing *good* code for the loop bodies would not be trivial. Also, automation would certainly restrict the flexibility of the approach, which must be seen as one of its major virtues.

And there is also a more fundamental obstacle to complete automation. The functional specification indicates no particular evaluation order for the matrix elements. In general, a recognizer may not even terminate. Our matrix recurrencies embedded in for-loops, on the other hand, always terminate. Thus, a general method to determine evaluation order would implicitly solve the Halting Problem. It will always require some human ingenuity and theorem proving to find an evaluation order for the imperative implementation.

Extensions of the method Our method can be extended in several ways; we only list a few:

Annotated nonterminals sometimes allow to write smaller grammars. This leads to combinators that handle parsers with extra arguments, and to three dimensional tables in the DP recurrencies. The size in the extra dimension corresponds to the value range of the annotation.

Yield parsers with extra arguments, say a table with data from mapping experiments, may focus the analysis towards structures compatible with positional constraints on base pairing. This extension fairly simple, since it is quite independent of the rest of our approach.

Heterogenous fs-algebras arise naturally when different matrices store different types of information. There will be various instances of the list evaluation function `pp` (most generally, one for each production of the tree grammar), but else, nothing changes.

Cross-products of fs-algebras allow two combine two analyses with a single recognition phase. This becomes intricate (only) if the results of the two analyses mutually depend on one another.

Future work Partly with the approach as presented, partly using the extensions sketched above, the following problems should be studied:

Collect *statistics* about frequencies of substructures. Can they be used for a functional classification of an unknown RNA sequence?

Incorporate *experimental data* to give an account of (the number of) succinctly different structures compatible with these data.

For a given set of *structural motifs* (hammerhead, cloverleaf, prominent hairpins) find the energetically best instance of this motiv in the folding space of a given RNA.

Folding away from a given motif means to find the energetically best structure that does *not* exhibit this motif. This relies on the closure properties of regular tree languages under complement (cf. Section 2.4) and may lead to a large grammar, but seems straightforward otherwise.

A more intelligent way to *sample the folding space* of a given RNA would greatly enhance the efficiency of the paRNAss [11] tool for the prediction of conformational switches.

Pseudoknots are very important in catalytic RNA, and it is an obstacle to the use of folding programs that pseudoknots cannot be detected due to computational complexity reasons. A DP approach (of complexity $O(n^6)$) is currently developed by Rivas and Eddy [25]. Their approach yields the most complex DP recurrencies this author has encountered in bioinformatics. It remains a challenge to find out if our approach can contribute to solving the pseudoknot problem.

Acknowledgement Michael Zuker encouraged me to seek ways to formulate more specific folding space analyses. Gerhard Steger gave advice on the differences between alternative RNA secondary structures that actually matter from the biological point of view. Stefan Kurtz provided numerous valuable hints on the algorithmics, helping to simplify the presentation of tabulating yield parsers considerably. However, the final incentive for this work I owe to Gene Myers, who asked me to write down just one more set of dynamic programming recurrencies.

References

[1] E. L. Anson and G. W. Myers. Realigner: A program for refining DNA sequence multi-alignments. In *1st Conference on Computational Molecular Biology*, pages 9–16, 1997.

[2] Alberto Apostolico and Ziv Galil. *Pattern Matching Algorithms.* Oxford University Press, 1997.

[3] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargettable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[4] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 2nd edition edition, 1998.

[5] E. Birney and R. Durbin. Dynamite: A flexible code generating language for dynamic programming methods. In *5th International Conference on Intelligent Systems for Molecular Biology*, pages 56–64, 1997.

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

[7] Dirk Evers and Robert Giegerich. RNA movies: Visualizing RNA secondary structure spaces. *Bioinformatics*, 1999. To appear.

[8] M. S. Gelfand and M. A. Roytberg. A dynamic programming approach for predicting the exon-intron structure. *Biosystems*, 30:173–182, 1993.

[9] Raffaele Giancarlo. *Dynamic Programming: Special Cases*, chapter 7 in Apostolico, Galil: Pattern Matching Algorithms, pages 201–236. Oxford University Press, 1997.

[10] R. Giegerich. Code Selection by Inversion of Order-Sorted Derivors. *Theor. Comput. Sci.*, 73:177–211, 1990.

[11] R. Giegerich, D. Haase, and M. Rehmsmeier. Prediction and visualization of structural switches in RNA. In *Proceedings 1999 Pacific Symposium on Biocomputing,*, 1999. to appear.

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

[13] A.P. Gultyaev, F.D.H. van Batenburg, and C.W.A. Pleij. Dynamic competition between alternative structures in viroid RNAs simulated by an RNA folding algorithm. *J. Mol. Biol.*, 276:43–55, 1998.

[14] D. Gusfield, editor. *Proceedings of the Fifth Annual Symposium on Combinatorial Pattern Matching, Asilomar, California, June 1994*. Lecture Notes in Computer Science **807**, Springer Verlag, 1994.

[15] I. L. Hofacker, P. Schuster, and P. F. Stadler. Combinatorics of rna secondary structures. *Discr. Appl. Math*, 89:177–207, 1999.

[16] I.L. Hofacker, W. Fontana, P. Stadler, L.S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte Chemie*, 125:167–188, 1994.

[17] J. Hudak, J. Fasel, and J. Peterson. A gentle introduction to Haskell. Technical Report YALEU/DCS/RR-901, Yale University, 1996.

[18] G. Hutton. Higher Order Functions for Parsing. *Journal of Functional Programming*, 3(2):323–343, 1992.

[19] Graham Hutton and Erik Meijer. Monadic parsing in haskell. *JFP*, 8(4), 1998.

[20] F. Lefebvre. An optimized parsing algorithm well suited to RNA folding. In *Proc. of the Third Conference on Intelligent Systems for Molecular Biology, ISMB 95*, pages 222–230. AAAI Press, Menlo Park, CA, USA, 1995.

[21] Fabrice Lefebvre. A grammar-based unification of several alignment and folding algorithms. In *Proceedings 4th ISMB*, pages 143–154. AAAI Press, 1996.

[22] Wu Ju Li and Jia Jin Wu. Prediction of RNA secondary structure based on helical regions distribution. *Bioinformatics*, 14(8):700–706, 1998.

[23] 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.

[24] R. Nussinov, G. Pieczenik, J.R. Griggs, and D.J. Kleitman. Algorithms for loop matchings. *SIAM J. Appl. Math.*, 35:68–82, 1978.

[25] E. Rivas and S. Eddy. A dynamic programming algorithm for rna pseudoknots. In *Poster Presentation, ISMB98*, 1998.

[26] Y. Sakakibara, M. Brown, R. Hughey, I.S. Mian, K. Sjölander, R.C. Underwood, and D. Haussler. Recent Methods for RNA Modeling Using Stochastic Context-Free Grammars. In [14], pages 289–306, 1994.

[27] M. Schmitz and G. Steger. Description of RNA folding by "simulated annealing". *Journal of Molecular Biology*, 255:254–266, 1996.

[28] D. B. Searls. Linguistic approaches to biological sequences. *CABIOS*, 13(4):333–344, 1997.

[29] D. B. Searls and K. P. Murphy. Automata-theoretic models of mutation and alignment. In *Proc. of the Third Conference on Intelligent Systems for Molecular Biology, ISMB 95*, pages 341–349. AAAI Press, Menlo Park, CA, USA, 1995.

[30] D.B. Searls. Representing genetic information with formal grammars. In *Proceedings of the 1988 National Conference of the American Association for Artificial Intelligence*, pages 386–391, 1988.

[31] D.B. Searls. Investigating the linguistics of dna with definite clause grammars. In *Proceedings of the North American Conference on Logic Programming*, pages 189–208. MIT Press, 1989.

[32] B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *CABIOS*, 6(4):309–318, 1990.

[33] T. F. Smith and M. S. Waterman. Comparison of biosequences. *Adv. Appl. Math.*, 2:482–489, 1981.

[34] E. E. Snyder and G. D. Stormo. Identification of coding regions in genomic dna: an application of dynamic programming and neural networks. *Nucleic Acids Res.*, 21:607–613, 1993.

[35] M. S. Waterman. *Introduction to Computational Biology. Maps, Sequences and Genomes.* Chapman & Hall, London, UK, 1995.

[36] M. S. Waterman and T. F. Smith. RNA secondary structure: A complete mathematical analysis. *Math. Biosci.*, 41:257–266, 1978.

[37] M. Zuker. On finding all suboptimal foldings of an RNA molecule. *Science*, 244:48–52, 1989.

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

[39] 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.