DISSERTATION

Titel der Dissertation

Grammatical Approaches to Problems in RNA Bioinformatics

Verfasser

Christian Höner zu Siederdissen

angestrebter akademischer Grad

Doktor der Naturwissenschaften (Dr.rer.nat.)

Wien, 2013

Studienkennzahl lt. Studienblatt:	A 091 490
Dissertationsgebiet lt. Studienblatt:	Molekulare Biologie
Betreuerin / Betreuer:	Univ.-Prof Dipl.-Phys. Dr. Ivo L. Hofacker

Contents

1 Of Grammars and Lambdas **1**
1.1 The Past 60 Years 1
1.1.1 RNA Structure Prediction 1
1.1.2 RNA Homology Search 7
1.1.3 Grammatical Adventures 9
1.1.4 Functional Programming Languages 10
1.2 This Thesis 11

2 The Structure of RNA **13**
2.1 The Structure of Nucleotides 14
2.2 RNA Secondary Structure 18
2.3 Scoring Schemes, Optimal Structure, and the Partition Function 19
2.4 RNA Structural Modules 21
2.5 RNA Families 22

3 Formal Grammars and ADP **25**
3.1 Regular and Context-free Grammars 26
3.1.1 Alphabets and Strings 26
3.1.2 Formal Grammars 26
3.1.3 The CYK Parser 29
3.1.4 Syntactic, Structural, and Semantic Ambiguity 31
3.2 Algebraic Dynamic Programming (ADP) 33
3.2.1 Methodology 33
3.2.2 Implementations of the ADP Idea 36

4 Efficient Algorithms in Haskell **39**
4.1 Being Lazy with Class 40
4.1.1 Ad-hoc Polymorphism Using Type Classes 41
4.1.2 Monads 42
4.1.3 Algebraic Data Types 43
4.1.4 Existential Quantification 44
4.2 Deforestation and Fusion 44
4.2.1 Call-pattern Specialization 46

4.2.2 Stream Fusion 47
4.2.3 The `case-of-case` Transformation 50
4.2.4 A Worked Stream Fusion Example 50

5 Semantics and Ambiguity of Stochastic RNA Family Models 53

6 Discriminatory Power of RNA Family Models 79

7 A Folding Algorithm for Extended RNA Secondary Structures 87

8 Sneaking Around *concatMap*:
Efficient Combinators for Dynamic Programming 97

9 Outlook 111
9.1 Generalizing the Language of Grammars 111
9.2 Performance, Automatic Parallelization, and SIMD extensions 112
9.3 Extended Secondary Structures 115
9.4 RNA Family Models 117
9.5 From Functions to Grammars and Back 120

Acknowledgements

First and foremost, thanks to Ivo for not only being a kind boss, but actually letting me explore some of the stranger corners of our scientific field, including pure and lazy λ's (being lazy with class is such a great motto!).

I had some truly great co-authors, of which Ivo, Robert, Peter, and Stephan were involved in the papers presented in this thesis. Thank you Robert for introducing me to Ivo and ongoing cooperation in exploring the grammatical wonders of RNA families. A lot of my later work was and is done in close collaboration with Peter in Leipzig. It is a pleasure working with you and I am looking forward to our next adventures.

I was lucky enough to have been able to work with many other people in the past couple of years. Thanks go to Ronny, Florian, Andrea, Fabian, Andreas, and Christoph. Jan invited me repeatedly to Copenhagen, and working with him, Corinna, Britta, and Christian is a lot of fun. Thank you Sven for being such a friendly Raumteiler. Many thanks also to Jing, Manja, Stefanie, Stefan, Jörg, Irma, Lydia, and Katja for discussions, ongoing work, and future publications, ... and gin (I think).

Many thanks to everybody else at the TBI in Vienna, to the people at Robert's group in Bielefeld, Jan's in Copenhagen, and Peter's in Leipzig.

I am also indebted to all the Haskell people for maintaining such a kind and helpful community. Special thanks to Roman for explaining the intricacies of stream fusion. I also remember the ICFP'12 conference in Copenhagen fondly.

Thanks to the ISCB for providing monetary support after the conclusion of ISMB'11.

Many thanks and my gratitude go to my family Ulrike, Rüdiger, and Annette for always supporting me. I am deeply indebted to you for giving me all the support I needed – even if it was about (sneakily) writing a paper under the Christmas tree.

Abstract

Formal languages and grammars are a classical topic in computer science and a powerful tool to deal with complexity in terms of algorithmic design. In this thesis, four scientific works are presented that aim to solve problems in computational biology. These problems, from the area of RNA bioinformatics, are prediction of RNA secondary structure and the search for homologous sequences of known non-coding (nc-) RNA families. Also, an efficient embedding of these grammars in a functional programming language is presented.

First, two algorithms on structural non-coding RNA families are presented. A structural ncRNA family is an alignment of related RNA sequences together with their consensus structure. From it a stochastic model can be calculated which, in turn, can be used to search for further related sequences on a genome-wide scale. A number of different possibilities exist to produce a stochastic model from a structural alignment. In Semantics and Ambiguity of Stochastic RNA Family Models (Giegerich and Höner zu Siederdissen, 2011) the ramifications of different encodings are discussed.

The algorithm in Discriminatory power of RNA family models (Höner zu Siederdissen and Hofacker, 2010) provides a solution to another problem on ncRNA families. In order to facilitate quality control on ncRNA family libraries, a method is provided to determine whether an RNA family is sufficiently well separated from all other families.

The algorithm on extended RNA secondary structures presented in A folding algorithm for extended RNA secondary structures (Höner zu Siederdissen, Bernhart, Stadler, and Hofacker, 2011) extends the nearest-neighbor secondary structure model toward better reflection of the knowledge gained from RNA tertiary structure. In particular, base pairing beyond the six canonical Watson-Crick pairs is taken into account. Some regions of the RNA with important biological roles contain almost exclusively non-canonical base pairs which can now be predicted in contrast to previous approaches which would model such regions as essentially unstructured.

Finally, in Sneaking Around *concatMap* (Höner zu Siederdissen, 2012), a domain-specific language embedded in the functional programming language Haskell is presented. This embedding allows for simplified algorithmic development on a high level. In particular, this embedded language makes it possible to write and extend the previous algorithms easily, while providing performance close to that of the `C` programming language.

Zusammenfassung

Formale Sprachen und Grammatiken sind ein klassisches Thema in der Informatik und mächtiges Werkzeug um die Komplexität algorithmischen Designs zu beherrschen. In dieser Arbeit werden vier wissenschaftliche Arbeiten zur Lösung von Problemen in der computergestützten Biologie präsentiert. Diese Probleme, ausgewählt aus dem Bereich der RNA-Bioinformatik, sind die Vorhersage von RNA Sekundärstrukturen und die Suche nach homologen Sequenzen bekannter nichtkodierender (nc-) RNA Familien. Ausserdem wird eine effiziente Einbettung dieser Grammatiken in eine funktionale Programmiersprache vorgestellt.

Zuerst werden zwei Algorithmen zu strukturellen RNA Familien präsentiert. Eine strukturelle ncRNA Familie is ein Alignment von verwandten RNA Sequenzen zusammen mit ihrer gemeinsamen Struktur. Sie kann in ein stochastisches Modell verwandelt werden. Solch ein Modell ermöglicht das Auffinden von weiteren verwandten Sequenzen auf genomweiter Ebene. Verschiedene Möglichkeiten existieren um solche Modelle zu erzeugen. In Semantics and Ambiguity of Stochastic RNA Family Models (Giegerich and Höner zu Siederdissen, 2011) werden die Konsequenzen verschiedener Kodierungen diskutiert.

Der Algorithmus in Discriminatory power of RNA family models (Höner zu Siederdissen and Hofacker, 2010) bietet eine Lösung für ein anderes Problem mit ncRNA Familien. Um die Qualität der vorhandenen Familien sicherzustellen wird eine Methode bereitgestellt, die es erlaubt festzustellen ob eine Familie genügend stark von allen anderen Familien differenziert.

Der Algorithmus zu erweiterten RNA Sekundärstrukturen in A folding algorithm for extended RNA secondary structures (Höner zu Siederdissen, Bernhart, Stadler, and Hofacker, 2011) erweitert das nearest-neighbor Sekundärstrukturmodell zu einem das das bekannte Wissen zu RNA Strukturen besser reflektiert. Insbesondere sind Basenpaarungen über das kanonische Watson-Crick Paarungsmodell hinaus möglich. Solch ein erweitertes Modell dient der besseren Vorhersage von Basenpaarungen in Regionen welche als wichtig in der biologischen Rolle vieler RNAs erkannt wurden.

Zuletzt wird eine domänenspezische Sprache, eingebettet in die funktionale Programmiersprache Haskell, in Sneaking Around *concatMap* (Höner zu Siederdissen, 2012) besprochen. Eine solche Einbettung ermöglicht ein einfacheres Entwickeln in einer Hochsprache. Insbesondere ist es möglich die vorherigen Algorithmen einfach zu formulieren und zu erweitern ohne auf `C`-nahe Geschwindigkeit verzichten zu müssen.

Chapter 1

Of Grammars and Lambdas

This thesis is mostly about *grammars*. They are used to provide a more formal treatment of several algorithms from the area of RNA bioinformatics. More specifically prediction of RNA secondary structure and the search for homologs of non-coding RNA. Along the way, additional problems are being tackled, including how to actually implement an algorithm efficiently in two senses: runtime efficiency and programmer time efficiency.

The story begins with some history on relevant topics (Chapter 1.1) followed by a discussion of what this thesis is about (Chapter 1.2). Background information on RNA, nucleotides, and their representations is given in Chapter 2. Chapter 3 is devoted to a small introduction to formal languages. The final background Chapter 4 explores some of the language concepts in the `Haskell` programming language.

In the four publications, advances in RNA bioinformatics are described. Two of these deal with RNA secondary structure prediction (Chapter 7, Höner zu Siederdissen et al. (2011)) and the search for homologs of non-coding RNA (Chapter 5, Giegerich and Höner zu Siederdissen (2011)). The other two publications give answers to two further questions. The calculation of the discriminatory power of a trained ncRNA model can be reduced to the comparison of pairs of stochastic context-free grammars (Chapter 6, Höner zu Siederdissen and Hofacker (2010)), while the final paper describes efficient implementations of grammars in a functional language (Chapter 8, Höner zu Siederdissen (2012)).

The final Chapter 9 yields some conclusions from the work so far and provides hints to future research in a number of directions.

1.1 The Past 60 Years

1.1.1 RNA Structure Prediction

In this thesis, the focus is on the secondary structure of RNA. Chapter 2 provides details on structure, definitions, and abstractions. Here, a history of algorithms in structure prediction is given.

We can recognize at least five major developments in RNA secondary structure prediction that have bearing on the results presented in later chapters. These developments are

roughly in order of appearance, but such an order can not always be maintained or they don't quite fit. Simple base pair counting, for example, is certainly a *non-physics based model* but does not capture the quite involved statistical calculations that went into such models.

First Algorithms and First Scoring Schemes

Tinoco et al. (1971) proposed a "simple method for estimating the secondary structure" (direct quote) of RNA based on a number of small building blocks. Namely hairpin loops, interior loops, bulges, and stacking regions in helices. Each such building block was given a simple score depending on if the block is stabilizing or destabilizing the structure. Loop regions are given negative scores, corresponding to a positive change in free energy, while stacking regions get a positive score. Such a simplification was necessary to keep the base pairing matrix simple enough. This matrix contains the positive base pairing score for each possible base pair. By using both, the base pairing matrix as well as the negative scores from loop region building blocks one could, by hand, find possible structures and sum up their scores.

However, Tinoco et al. (1971) did not make use of advanced computing techniques. The base pairing matrix was simply that, a matrix with positive scores whenever two nucleotides could base pair, and was as such only a visual aid. Once such a scoring scheme was introduced, Nussinov et al. (1978) were able to cast it as a dynamic-programming algorithm. Using a dynamic-programming algorithm has several nice properties. While the number of RNA secondary structures grows exponentially with the length of the RNA sequence (Smith and Waterman, 1981), calculating the full dynamic-programming matrices, from which the optimal structure can be extracted, takes only polynomial time. Granted, dynamic-programming requires scoring functions that obey certain restrictions (namely the *Principle of Optimality* by Bellman (1952)), but both simple additive scoring schemes and simple thermodynamic energy schemes follow those restrictions.

Nussinov et al. (1978) also made use of the planar graph model of secondary structures, wherein pairing nucleotides form edges in the graph together with the edges given by the RNA backbone. By allowing only a single pairing edge in addition to those formed by the backbone, base-pair triplets are forbidden. As the graph is also forced to be outerplanar, pseudo-knots are equally forbidden. In total, Nussinov et al. (1978) set the stage for algorithm designs on RNA secondary structure that is still followed today. Hence, most of the algorithms mentioned below *extend* on the ideas given in said paper.

The Nearest-neighbor Energy Model

Simple base pair counting models (Tinoco et al., 1971) break down if the input sequences become too long to handle. While the algorithm by Nussinov et al. (1978) can handle longer input, it still does not provide accurate predictions. Some improvement was possible by not just counting base pairs but to actually look at free energy changes. Tinoco et al. (1971) and Nussinov et al. (1978) ultimately consider such energy changes. It however soon became possible to consider not just free energies for simple base pairs but for stacking

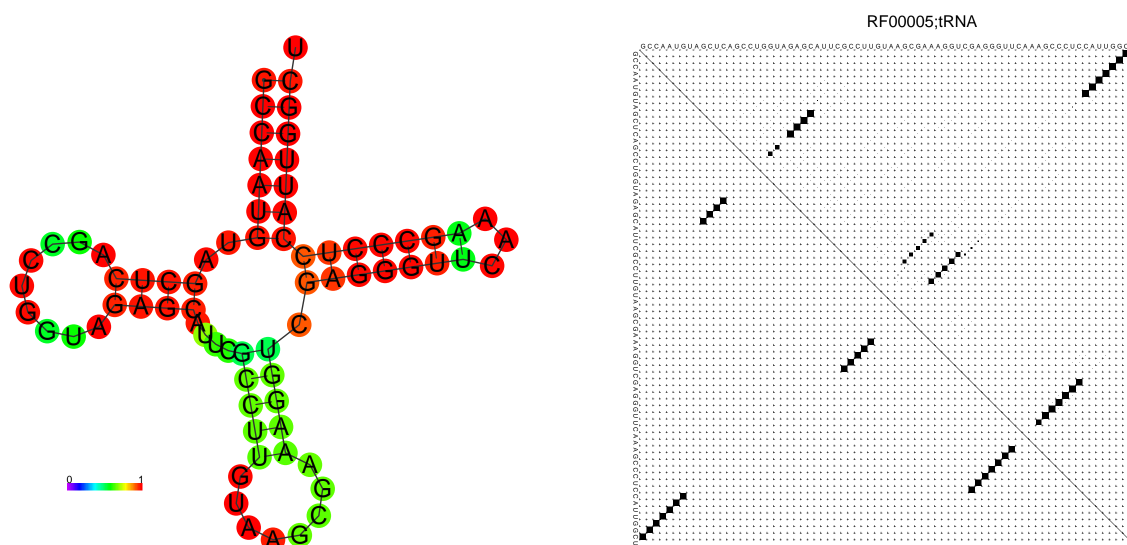

Figure 1.1: *Left:* A `tRNA` minimum-free energy structure. Each nucleotide is colored according to its probability to be paired or unpaired.
Right: McCaskill (1990) - style dotplot. The edges are annotated with the sequence. The upper-triangular matrix shows the probability for each nucleotide to be paired depending on the size of the square filling each cell. The lower triangular matrix gives the minimum-free energy structure. The upper-triangular matrix provides evidence for an alternative structure for one of the stems.
The plots have been produced using the Vienna RNA Websuite (Gruber et al., 2008).

interactions (Tinoco et al., 1973). Stacking interactions were discussed by Nussinov et al. (1978) but not to satisfaction.

Such an algorithmic improvement using both stacking (negative free energies) and loop energies (positive, destabilizing free energies) was finally given by Zuker and Stiegler (1981). This algorithm also began a trend to more and more complex recursions – or grammars in the terminology preferred by us. Compared to the more humble beginnings ten years earlier, calculating the free energy and the folded secondary structure of RNAs of around 500 nucleotides was feasible.

One especially important derivative algorithm was presented by McCaskill (1990). With longer sequences and an exponential number of candidate structures, the structure with minimum free energy might no longer be the best representative of the structure space. McCaskill derived a set of recursive equations together with an outward-inward calculation approach that results in a base pairing *probability matrix* as depicted in Fig. 1.1. Conceptually similar to the base pairing matrix used by Tinoco et al. (1971), the base pairing probability matrix states for a given input sequence the pairing probability of each possible base pair.

Using this matrix it is possible to determine if a certain structure dominates the structure space, or if multiple different structures have non-negligible probability. As with the work by Nussinov et al. (1978), many derivative works would be based on the work by

McCaskill (1990).

The number of wet-lab experiments measuring melting temperatures of small oligonucleotides increased substantially, allowing for increased accuracy of the predicted structures. Works along the lines of Walter et al. (1994); Xia et al. (1998); Mathews et al. (1999) introduced energies for coaxial stacking of otherwise independent helical regions in multi-branched loops, energies for dangling nucleotides next to helices, and further improvements.

The algorithms presented in Hofacker et al. (1994); Lorenz et al. (2011) round off the set of algorithms based on the nearest-neighbor model. They are efficiently implemented, explore novel concepts (for RNA structure folding) like parallel computation of the dynamic programming matrices and provide some derivative results, like RNA structure comparison or the inverse folding problem.

Non-physics Based Models

A second wave of non-physics based models (if we count the simple base pair maximization schemes) was introduced to make use of the availability of multiple alignments of homologous RNA sequences, or RNA families. Such alignments will be taken up again below. Here it is simply assumed that such multiple alignments exist and that they are accurate and annotated with the correct consensus secondary structure.

Given such information, one can count the frequency with which each unique RNA structural feature occurs. Say the number of `GC`-on-`GC` stacks in the number of all stacks, or the relative frequency of multi-branched loops given the choice of continuing a stack, creating a multi-branched loop, an interior loop, or a bulge. In order to collect the frequency of each feature in the ensemble of all features correctly the underlying grammar or recursions need to be structurally unambiguous. The term structurally unambiguous will be taken up again in the following chapters. Here it should suffice to say that a single structure can have only (at most) one way to create it using the algorithm. It is possible to exclude certain structures in the inference process (pseudoknots, for example), but each structural feature should be counted only once.

The `CONTRAfold` algorithm by Do et al. (2006) is quite well-known and has comparative performance to modern physics-based models. `CONTRAfold` is based on a stochastic context-free grammar, more specifically a conditional log-linear model, and uses discriminative statistical methods. The `CONTRAfold` training algorithm effectively learns parameters for the same type of features as used previously in the Turner model, including parameters for the lengths of unpaired regions in different structural features and the sequence-dependency of stacking parameters.

It is also possible to train generative probabilistic models which is generally much simpler and requires fewer resources in terms of runtime. Rivas et al. (2012) have shown that generative probabilistic models of comparable complexity to `RNAfold` and `CONTRAfold` have comparable accuracy. Said work also provides a simple ranking of structural features in terms of their importance toward accurate secondary structure predictions.

Finally, one does not have to choose between either a completely physics-based model or one driven solely by learned parameters. Andronescu et al. (2007); Andronescu (2008);

Andronescu et al. (2010a) proposed an algorithm that starts with initial physics-based parameters for RNA secondary structure which are then driven via different constrained optimization schemes toward better prediction accuracy. One advantage of this approach is that the optimized parameters can still be interpreted as energy contributions of different features. Such energy calculations are of use for certain derived algorithms which model RNAs based on target free energy values – and the optimized parameters can be used in algorithms such as `RNAfold` or `Mfold`.

Derived Algorithms

The availability of modern, efficient physics- and non-physics based algorithms has lead to a wealth of derived algorithms. The basic secondary structure folding algorithm determines the globally optimal structure according to the energy or scoring model. By turning the evaluation part of the algorithm (later to be called algebra in some works) into an inside-outside scheme McCaskill (1990) derived an important variation of the general algorithm wherein the pairing probability of each potential base pair can be calculated.

A conceptually simple (but only in retrospect!) extension of the minimum free energy solution is to provide the option to backtrace all suboptimal solutions within a certain energy band (Wuchty et al., 1999). Backtracing all such suboptimal structures requires a structurally non-ambiguous grammar. That is, there must not be two different back-traces that yield the same secondary structure. While it is in principle possible to use a structurally ambiguous grammar, this would yield multiple parses of the same secondary structure which need to be filtered out in a subsequent step. Such a step is normally prohibitive with larger energy bands as the number of secondary structures increases exponentially. Zuker (1989) uses a different algorithm to generate suboptimal structures that are "not too close" to each other but will possibly not capture all suboptimal structures, leaving part of the search space unexplored (Wuchty et al., 1999).

It is also possible to extend the RNA folding problem to two interacting RNA sequences. Solving the problem of interacting RNA sequences requires the solution of the two folding problems which interact via an alignment procedure. The runtime and memory requirements for the direct solution are quite high ($O(n^3m^3)$ time / $O(n^2m^2)$ space) and the grammar or recursions involved are very complex (Pervouchine, 2004; Alkan et al., 2006). In case not only the minimum free energy structure, but the partition function is to be calculated, the number of non-terminals (the number of structural elements involved in the decomposition) again increases a lot in order to keep the decomposition structurally non-ambiguous. The implementations by Huang et al. (2009) and Chitsaz et al. (2009) both provide fairly involved decompositions.

It is, however, also possible to restrict the problem of interacting RNAs in certain ways. In case only one binding site between the two RNAs is allowed the problem can be solved more efficiently as demonstrated by Mückstein et al. (2006, 2008).

Another direction for extensions of the basic folding algorithm is `RNAalifold` by Hofacker et al. (2002); Bernhart et al. (2008). `RNAalifold` takes a set of aligned sequences and calculates the *consensus structure* based on both, energy contributions from loops and stacks, as well gaps in the alignment. The earlier algorithm (Hofacker et al., 2002) uses

an energy-like score based on covariation, while the later (Bernhart et al., 2008) improves predictions using an advanced model that calculates using RIBOSUM-like matrices. The problem `RNAalifold` aims to solve, in addition, points to questions regarding RNA family models being dealt with later.

An extension and generalization of the idea of `RNAalifold` is provided by `LocARNA` (Will et al., 2007, 2012). While `RNAalifold` finds the consensus structure given an existing alignment, `LocARNA` calculates a structural alignment given two *unaligned* sequences, in the spirit of Sankoff (1985) but with improved asymptotic runtime. In its extension `mLocARNA`, known and putative RNA families can be discovered on a large scale.

Genome-wide scans are possible using modified and extended versions of RNA folding algorithms. This allows, for example, to predict non-coding RNAs in regions of genomes that exhibit some sequence conservation among a number of species. The `RNAz` algorithms (Washietl et al., 2005; Gruber et al., 2010) are designed to work on multiple sequence alignments. By combining information from the predicted consensus secondary structure (using `RNAfold` (Hofacker et al., 1994) and `RNAalifold` (Hofacker et al., 2002)) and per-sequence minimum free energy a support vector machine-based classifier can be used to predict if a local structure is a likely non-coding RNA.

Beyond Canonical Secondary Structures

The notion of secondary structures can be generalized in several different ways. In general RNAs fold into the secondary structure first while tertiary elements follow later on. The reason for this behaviour is the comparatively strong binding in helices, which stabilizes the secondary structure.

Once the secondary structure is formed, additional elements like pseudoknots are formed. A number of computational approaches to pseudoknot prediction exist. These approaches can be classified by the types of pseudoknots that can be predicted. More general classes of pseudoknots, that allow for more complex structures, require algorithms that are computationally more demanding while pseudoknot prediction in general is NP-hard (Lyngsø and Pedersen, 2000).

One could also classify by the method used (i.e. dynamic programming) or parameter model (physics vs. non-physics) but given the general sparsity of thermodynamic information and few known pseudoknotted structures, such distinctions are of less relevance and mostly determined by the underlying secondary structure model. Andronescu et al. (2010b), for example, trained pseudoknot parameters in the same vein as for their improvements on secondary structure energy parameters (Andronescu et al., 2010a).

Different algorithms have been formulated for different classes and complexities of pseudoknots (Akutsu, 2000; Rivas and Eddy, 2000; Cai et al., 2003; Reeder and Giegerich, 2004; Meyer and Miklós, 2007; Sato et al., 2011) with a formal treatment based on topology given in Reidys et al. (2011). The latter treatment classifies pseudoknots by topological genus, provides an algorithm for almost all known naturally occurring pseudoknots, and describes models for pseudoknots of higher genus.

Some structural elements classified here as beyond secondary structure, such as G-quadruplexes (Wong et al., 2009), can actually be implemented in terms of context-free

grammars (Lorenz et al., 2012, 2013) where they are established as a distinct element like stems, bulges, interior loops, or a multi-branched loop. G-quadruplexes are, on the other hand, *not* composed of canonical base pairs and require separately determined energies.

The ultimate goal of structure prediction is to predict the complete tertiary structure up to and including atomic resolution accuracy, complete with potential interactions with other biomolecules. Compared to secondary structure prediction, this goal is much harder. While polynomial-time models (based on dynamic programming) are well established for secondary structures, pseudoknots, and secondary structure interactions, tertiary structure is mostly determined by two different methods. Either a stochastic simulation is run, as established for protein simulations (Das and Baker, 2007; Das et al., 2010), or programs try to insert small building blocks into a scaffold. (Parisien and Major, 2008; Reinharz et al., 2012) A more general overview including 3D prediction performance is given by Cruz et al. (2012).

1.1.2 RNA Homology Search

RNA homology search combines elements of RNA secondary structure, sequence alignment, statistics, and (stochastic) context-free grammars. The over-arching question is, given a number of homologous (related) RNA sequences, can one find novel sequences from different species that belong to this set? `BLAST` (Altschul et al., 1990) provided a first step toward fast genome-wide similarity searches by allowing a heuristic local alignment search to be run over whole genomes. `BLAST` however, is restricted to fairly closely related sequences as only one sequence is used as the query. In order to find more remote homologs, one has to run `BLAST` iteratively with a more and more diverse sequence set.

By using an aligned set of sequences, one can derive a statistical description of the set of these sequences to search with. For families of protein sequences, where long-range interactions are not modelled explicitly, the `HMMer` tools (Eddy, 1998) can be used. These create hidden-Markov models from the input set of sequences and can deal with larger insertions and deletions in the sequences to search for. The `Pfam` database contains 13 672 protein families as of November 2011.

As will be further explained in Sec. 2.5, structural non-coding RNAs conserve structure more than they conserve sequence information. For this reason, base pairing needs to be considered when trying to define a statistical model of a set of aligned homologous RNA sequences, or RNA families. While `HMMer` models can be designed as stochastic regular grammars (Chapter 3), non-coding RNA models require the use of stochastic context-free grammars. The `Infernal` (Eddy and Durbin, 1994; Nawrocki et al., 2009) tools provide programs to both, create RNA family models and search for novel homologs in genomes. The, `Rfam` (Gardner et al., 2011) database collects these models and currently contains over 2 000 models.

Using `Infernal` for model building and search requires an aligned set of RNA sequences, together with their common structure. The question of how to create such a family model is still an active research subject. Different algorithms try to solve the multiple-sequence alignment problem, the problem of estimating a consensus secondary structure, or both at the same time. The `RNAalifold` tool has already been mentioned,

which requires aligned sequences and predicts the common structure. The afore-mentioned LocARNA tool (Will et al., 2007) aims to speed up the Sankoff-style (Sankoff, 1985) simultaneous folding and alignment process to provide structural alignments automatically. More details on the model building process and complications are found in Sec. 2.5.

One interesting approach is to base RNA motif discovery on automatic creation and refinement of RNA family models, in particular the covariance models used by CMfinder (Yao et al., 2006).

In another recent work, Lessa et al. (2011) improved upon the statistical prior information that influences the model construction process to increase the accuracy of non-coding RNA search.

The introduction of Infernal has spawned additional research on heuristic speed improvements regarding stochastic CFG approaches on whole genomes. While it is possible to scan whole genomes with an RNA family model, the process is extremely time-consuming, requiring upwards of hundreds of hours for a single (mammalian) genome and one RNA family. Approaches to speed up the process include BLAST and HMMer pre-filters (Nawrocki et al., 2009; Kolbe and Eddy, 2011), and Query-dependent banding (QDB) (Nawrocki and Eddy, 2007). Using the QDB heuristic, parts of the dynamic-programming matrix are ignored if individual parse trees rooted at the matrix cell(s) have low probability of being part of the final result.

Another possibility is to split a large RNA model into multiple sub-models (Smith, 2009). Multibranched loops increase the asymptotic running time from $O(n^3)$ to $O(n^4)$. If individual helices are being searched, a faster algorithm can be used and the slower multi-branch enabled algorithm is run only for those regions not excluded by this helix-based filter.

In this thesis, two questions are being considered that deal with the construction and search process. While Infernal covariance models can be created from any RNA family, that is a set of structurally aligned RNA sequences, there is no guarantee that said model represents a well-defined family. The Rfam database, for example, contains a number of RNA families that are biologically related but whose sequences have diverged far enough that a single family model does not adequately capture the sequence and structure diversity anymore. Using a dynamic programming method, Höner zu Siederdissen and Hofacker (2010) (Chapter 6) provide a way to calculate how well an RNA family model discriminates against other RNA family models. This allows for quality control of existing and novel families on a large scale, including the whole Rfam database. This quality control measure can be used to provide further information in determining if newly designed families are unique (Lange et al., 2012; Chen and Brown, 2012).

The second work on RNA family models (Giegerich and Höner zu Siederdissen, 2011) (Chapter 5) provides an in-depth evaluation of the context-free grammar underlying Infernal. In particular, different formal semantics of the query to consensus matching are developed, and the semantics used by Rfam models is discussed. The results provide long-term goals for a novel grammar which might be able to capture remote homologs better in the case of larger structure and sequence diversity.

1.1.3 Grammatical Adventures

In the Chomsky hierarchy of grammars (Chomsky, 1956; Grune and Jacobs, 2008, Chapter 2.3), two types of grammars are of particular interest for (RNA) bioinformatics. Regular and context-free grammars (CFGs) combine enough expressive power to formulate and encode a number of algorithms that solve important problems. They are, at the same time, restrictive enough that there exist efficient implementations of these algorithms.

Formal languages have been used for decades to create parsers for artificial languages (Grune and Jacobs, 2008), especially programming languages (Aho et al., 2007). Their stochastic counterparts are used to model natural languages as well. Stochastic context-free languages may be used for speech recognition (Jurafsky et al., 1995), and similarities to the Sankoff algorithm on simultaneous folding and alignment (Sankoff, 1985) can be observed in machine translation (Burkett et al., 2010).

Biological problems are often treated using dynamic programming formalism without explicitly stating connections to a formal grammar. Bompfünewerer et al. (2008) consider structural decomposition schemes for RNA secondary structure in terms of dynamic programming recursions. Similarly, in the `Vienna RNA` package (Lorenz et al., 2011), one speaks of recursions, not of grammars. Nevertheless, a direct correspondence between these recursions and context-free grammars does exist. Rivas et al. (2012) formulate the `RNAfold 2.x` algorithm in terms of a stochastic CFG and train a set of parameters that yield the same prediction accuracy as the physics-based nearest-neighbor energies by Lu et al. (2006).

Alignment, search, and structure prediction are handled using grammars in the book on *Biological sequence analysis* (Durbin et al., 1998). Though the emphasis is on probabilistic models, not on grammars. Algorithms are most often explained in terms of recursions. This is easy to understand if one considers that one of the most used parsers with regard to works discussed here, the CYK parser (Grune and Jacobs, 2008, Chapter 4.2; Durbin et al., 1998, Chapter 10) uses dynamic programming to calculate the optimal parse in stochastic applications, or the correct parse(s) in non-stochastic applications[1].

Another addition to a more formal treatment of dynamic programming algorithms for RNA bioinformatics is Algebraic dynamic programming (ADP) (Giegerich and Meyer, 2002). ADP makes explicit the notion of parse trees. Each parse tree is an exact representation of one of the many possible ways how to parse an input string with a dynamic program written in ADP. This also emphasizes the notion that there are many parse trees for dynamic programs or grammars handling RNA bioinformatics problems. This is because one generally has not only one optimal parse or result but many co- or suboptimal ones, too.

Algebraic dynamic programming will be dealt with again in Chapter 3 as it is used in one way or another in the works described in Chapters 5,6, and 8 though sometimes implicitly.

ADP has evolved since its original inception in the functional programming language Haskell. In general, the performance of Haskell code is noticeably less than code written

[1]In programming language parsing, there should be only one correct parse

in C. One solution to speeding up ADP algorithms was the ADP compiler (Steffen, 2006). As Haskell was not only a barrier in terms of performance but also in terms of adoption by potential users not well-versed in using a functional language, a second domain-specific language for dynamic programs on sequence input was designed. The GAP language and GAP compiler (Sauthoff et al., 2011, 2013) provide a terse language well-suited for RNA bioinformatics problems on sequence data that is compiled directly to C++.

With ADPfusion (Höner zu Siederdissen, 2012) (Chapter 8) the ideas of ADP have come full circle. ADPfusion is again written in Haskell and provides performance very close to C and is typically faster[2] than an equivalent program written in the GAP language and compiled using the GAP compiler.

1.1.4 Functional Programming Languages

Functional programming languages like Lisp (Steele, 1990), Scheme (Dybvig, 2003), or Haskell (Hudak et al., 2007) are not common in bioinformatics (Fourment and Gillings, 2008). Imperative and object-oriented languages like C and C++ are used to code performance-critical algorithms, while scripting languages (Perl, Python) are used to glue different algorithms together. This is, of course, a simplified view. One may write complete programs in Python or glue different programs or algorithms together in C. Functional programming approaches are, however, rarely used. Experience reports (Daniels et al., 2012) describe a number of positive and negative aspects of using such a language.

One aspect, performance, is of major importance. Programs like RNAfold or Infernal are regularly used on a large scale, either with many inputs, input of large size or both. In such cases it is simply not acceptable to pay for the joys of functional programming with an overhead of $\times 100$ in performance compared to C.

Pure, functional languages like Haskell do not need to concern themselves with topics like side effects when compiling code. Being free of side effects means that a function will always return the same output when given the same input and may not perform any task other than calculating the result[3]. This allows aggressive optimizations and code manipulations to take place while guaranteeing that the optimized function (or algorithm) is semantically equivalent to its non-optimized version. A number of advances in recent years (Coutts et al., 2007; Peyton Jones, 2007; Leshchinskiy, 2009) have recently led to the result that Haskell can indeed beat C in performance-critical algorithms (Mainland et al., 2013).

A number of these improvements will be discussed further in Chapter 4 where the main aspects of the Haskell programming language are introduced. These improvements also played a major role in allowing ADPfusion to come close to the performance of C code for dynamic programs in RNA bioinformatics.

[2] this was true at least for the Nussinov78 and RNAfold algorithms in Benasque, 2012 (personal communication with S. Janssen).

[3] an often-used example is that a function in Haskell may not launch missiles while calculating a result as that would be a side effect

1.2 This Thesis

In this thesis four papers are presented that deal, in one way or another, with grammatical approaches to problems in RNA bioinformatics. The set of problems that are either solved or require (better) solutions is growing and diverse as mentioned above in the historical review. Despite this diversity, it is possible to find and name common themes. The commonalities between seemingly disparate problems sometimes require generalizations before shared ideas can be extracted and used to solve additional problems.

Formalising problems and algorithms in a more general and extensively researched language is advisable as this allows inferring solutions in a more general framework. The more general framework used here are formal languages and their grammars. Using formal languages to model problems in RNA bioinformatics is, of course, not new to this thesis.

Sakakibara et al. (1994) and Eddy and Durbin (1994) have employed stochastic context-free grammars (SCFGs) to model structural non-coding RNA families. SCFGs are an appropriate method to model RNA families composed of RNA sequences from different species with the same function. An SCFG can capture the secondary structure induced by base pairing as well as the sequence-based statistical signal. The statistical models produced by tools like `Infernal` can then be used to search for homologous sequences for each RNA family. This makes it possible to annotate newly sequenced genomes.

What was mostly ignored is the question of the semantics of aligning a part of the genome to an RNA family model – the essential part of the homology search process. In "Semantics and Ambiguity of Stochastic RNA Family Models" Giegerich and Höner zu Siederdissen (2011) (Chapter 5) explore three different semantics for RNA family models. The results therein give a proof that `Infernal`-models (Eddy and Durbin, 1994) are *actually correct* with regards to one of the three semantics, namely the alignment semantics. It is also shown how to construct a formal grammar that implements the trace semantics. Traces have the potential to better capture remote homologs than is possible with the alignment semantics as one trace subsumes many alignments with the same *biological* semantics.

Given the topic of RNA family models, one can consider another problem, namely the quality of RNA family models. The `Rfam` database (Griffiths-Jones et al., 2003; Gardner et al., 2011) contains more than 2200 RNA families (as of Rfam 11.0, August 2012) but no obvious means of quality control. If an RNA family (model) is constructed and then used to search for homologs, there should be some measure of . . . what? It is not possible to determine in silico if a family is correct. It should model actually existing structural RNAs but using the structural alignment alone, this can not be inferred. In "Discriminatory power of RNA family models" (Höner zu Siederdissen and Hofacker, 2010) (Chapter 6) an algorithm is developed that calculates a score, the `Link Score`, for pairs of `Infernal` RNA family models. This score gives, using `Infernal` bit scores, a measure of how similar two RNA families are – viewed through the `Infernal` scoring model. Whenever two families yield a high `Link Score` it is likely that there exist genomic regions where both models will claim to have found a possibly homologous sequence, an event the `Rfam` curators try to prevent (Gardner et al., 2011).

Algorithmically speaking, the `CMCompare` algorithm which calculates the `Link Score` is novel as it provides a way to *align* two stochastic context-free grammars. This is possible because the underlying grammar used by `Infernal` constructs full models from a small set of building blocks.

With "A folding algorithm for extended RNA secondary structures" (Höner zu Siederdissen et al., 2011) (Chapter 7) the focus switches from RNA families to single non-coding RNA sequences. With more features of the global three-dimensional structure and local interactions becoming known, it is appreciated that the canonical rules of RNA base pairing do not describe the whole picture. Individual nucleotides engage in more complex pairing than just a single other nucleotide, especially in regions of the RNA that are active binding sites or show an unusual structure. The `RNAwolf` algorithm provides an efficient polynomial-time algorithm to predict the extended secondary structure of an RNA including non-canonical base pairs and base triplets. The existence of base pair triplets in real structures and the desire to predict them requires the construction of a novel grammar with higher complexity compared to the grammars used by `RNAfold` or `CONTRAfold`.

The work done on these and other problems has led the author to the conclusion that a terse, high-level domain-specific language with a number of special characteristics is required. Resulting code should produce fast programs – ideally within a factor of $\times 2$ or less compared to a well-optimized `C` implementation of the same algorithm. It should also be possible to experiment with new features, not only of the algorithm to implement, but the domain-specific language itself. Finally, the language should be embedded in Haskell as the host language, making it possible to "break out" of the domain-specific language at any point and use features available to the host language. These desiderata have been successfully met in the creation of the `ADPfusion` framework in "Sneaking Around *concatMap*" (Höner zu Siederdissen, 2012) (Chapter 8). In the spirit of Algebraic dynamic programming, terse implementations of the algorithms presented in this thesis are possible and `ADPfusion` can be extended by every user without having to change the `ADPfusion` library itself.

It is to be expected that the individual topics presented here will lead to further developments that join and merge individual ideas. Some of these ideas for future research are presented in Chapter 9 – and some of the ideas have already been implemented.

The three following chapters give short introductions to the topics of the structure of RNA, formal grammars, and efficient algorithms in a functional language.

Chapter 2

The Structure of RNA

Ribonucleic (RNA) and deoxyribonucleic (DNA) acids are long bio-polymers. DNA is the carrier of genetic information for all known organisms (Deonier et al., 2005, p.2)[1]. For decades the *central dogma of molecular biology* has stated that "DNA makes RNA makes protein" as indicated by "probable transfers" according to Crick (1970). Only in recent years has RNA become an object of considerable interest of its own, and not just the messenger. Mattick and Makunin (2006) give an overview of non-coding RNA and its role that "determine[s] most of our complex characteristics, play[s] a significant role in disease and constitute[s] an unexplored world of genetic variation both within and between species."

In this chapter, the focus is on the structure of RNA and its abstractions. In RNA bioinformatics, one can typically deal with abstract models that have little resemblance to the actual physical, atomic structure of the molecules in question. In particular, RNA and DNA are sequences in the computer science sense, and individual nucleotides are characters in an alphabet. In another view, nucleotides are nodes in a graph and RNA secondary structure is represented by edges between those nodes.

While this provides a nice abstraction, it also means a loss of information. In particular, it abstracts away the question of how many edges a node may have – or how many pairings are possible for an individual nucleotide. This question is explored in Chapter 7 with extended secondary structures. The extended structure space aims to more faithfully represent the possible structural conformations of real RNA structures. Some of the less canonical aspects of RNA structure also play a role in designing scanning algorithms for whole-genome non-coding RNA search (Chapter 5). For ncRNA search it seems enough, however, to consider the sixteen possible pairings independent of the exact nucleotide edge involved, and without considering base triplets.

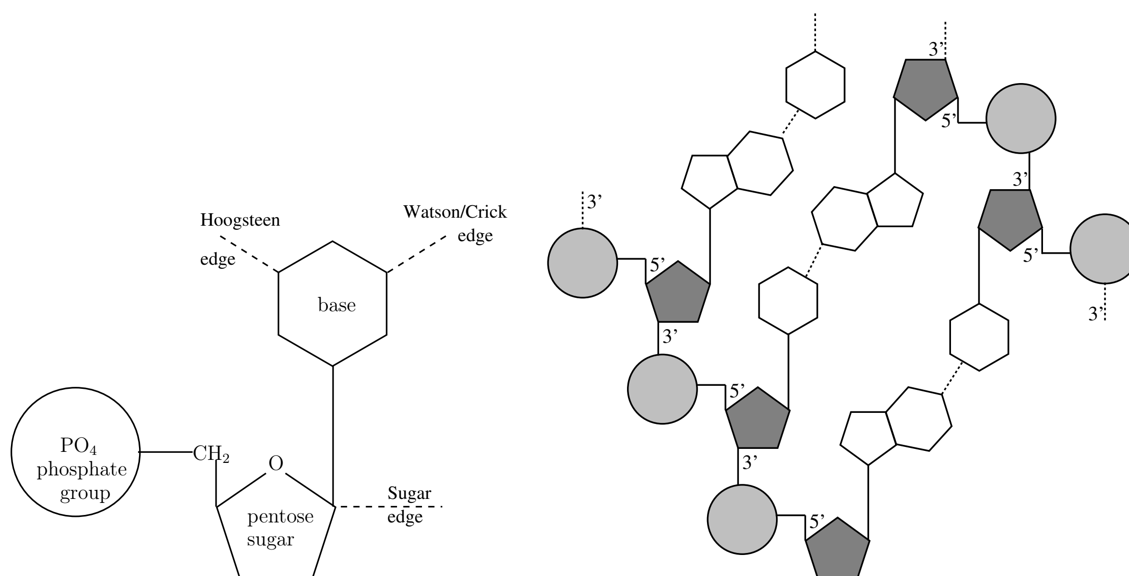

Figure 2.1: *Left:* The three building blocks of nucleotides. The phosphate group and pentose sugar form the backbone of the nucleic acid. The bases (`ACGU` in RNA) are attached to the pentose sugar. Bases can pair with each other. The edges available for pairing are depicted in more detail in Fig 2.2.
Right: Part of an RNA chain. Nucleobases are colored white with sketched Watson/Crick pairing. The pentose sugar is colored grey, the phosphate group in light grey.

2.1 The Structure of Nucleotides

Abstractions of nucleotide molecules play a central role in RNA bioinformatics. Depending on the task, anything from full-atom models for tertiary structure prediction (Das et al., 2010) to single characters in the alphabet {A,C,G,U} (e.g. `RNAfold`, Lorenz et al. (2011)) or {A,C,G,T} (e.g. `Infernal`, Nawrocki et al. (2009)) may be considered.

The algorithms above have in common that they exploit that RNA nucleotides pair with each other. As the focus of the algorithms presented later is on RNA secondary structure, the three-dimensional full-atom structure of RNA nucleotides is not considered and a flat two-dimensional representation as in Fig. 2.2 is enough.

Nucleotides are composed of three distinct building blocks, the phosphate group, the sugar pentose, and the purine or pyrimidine (Campbell and Reece, 2003, pp.97–98), as depicted in Fig. 2.1. The sugar and phosphate groups form the backbone in RNA or DNA chains via covalent bonds, while individual nucleotides may base pair with each other, thereby forming long-range interactions. It is those long-range interactions which are predicted by RNA secondary structure tools.

[1] Yes, we cite basic biological information from a book on *Computational Genome Analysis*, named as such!

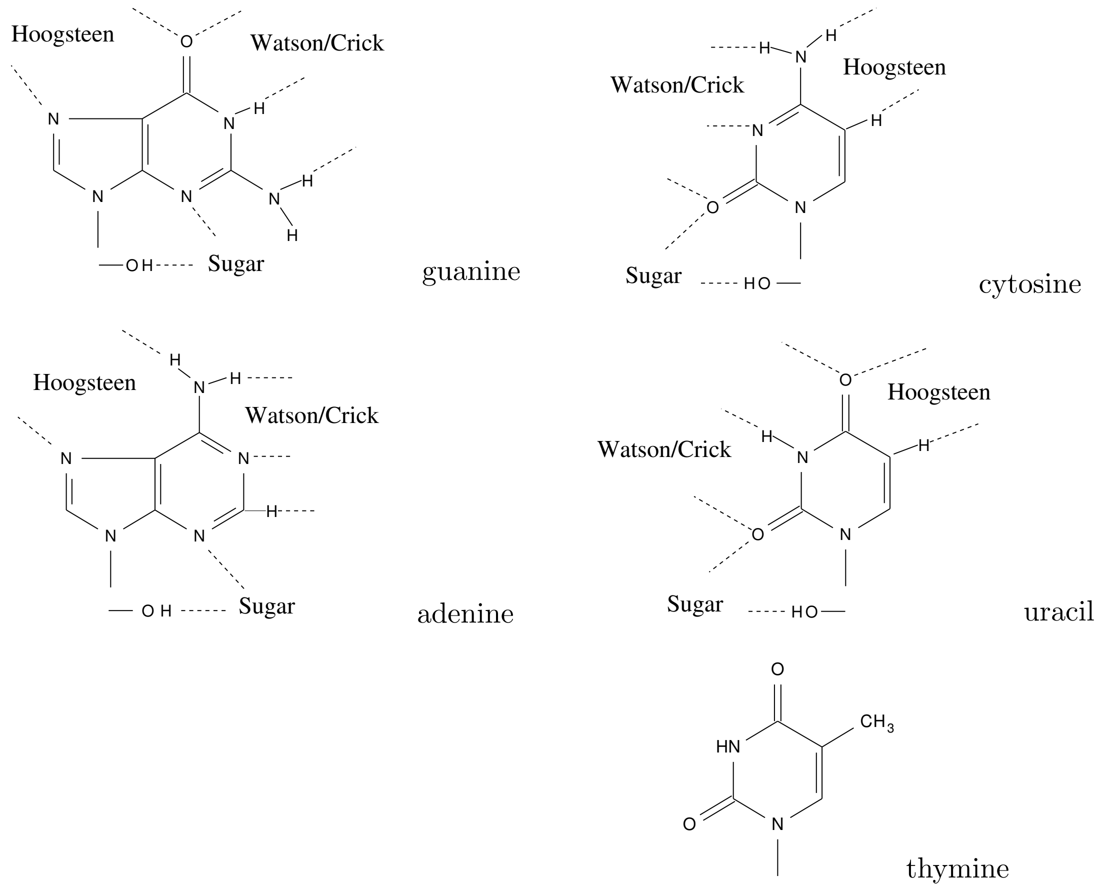

Figure 2.2: The building blocks of RNA and DNA. Cytosine (`C`), uracil (`U`), and thymine (`T`) are pyrimidines, adenine (`A`), and guanine (`G`) are purines. Thymine (`T`) is used in DNA, Uracil in RNA, the other nucleotides are found in both nucleic acids. The three possible binding sites, for the nucleotides found in RNA, are annotated. Watson/Crick pairs engage in up to three hydrogen bonds. For non- `cWW C-G / A-U` pairs consult Leontis et al. (2002) for exact bond locations for each pair and conformation.

type	cWW						tSH	tHS	tsS
pair	G-C	C-G	U-A	A-U	G-U	U-G	G-A	A-G	G-A
count	73 342	68 083	23 606	23 419	10 168	9 644	7 742	6 798	5 121
fraction	0.249	0.231	0.080	0.079	0.035	0.033	0.026	0.023	0.017

type	tWH	csS	tSs	tHW	cSs	csS	cSH	cSs	Rest
pair	U-A	C-A	A-G	A-U	C-A	A-C	G-U	A-C	
count	4 474	3 638	2 863	2 851	2 564	2 109	2 072	1 917	44 302
fraction	0.015	0.012	0.010	0.010	0.009	0.007	0.007	0.007	0.150

Table 2.1: Number of base pairs of each type in the unfiltered (see Sec. 2.1) FR3D derived from the PDB. Pairs are annotated by orientation (cis, trans), pairing edge (Watson-Crick, Sugar, Hoogsteen), and nucleotide (ACGU). The six canonical cWW base pairs make up the bulk of the base pairs. Non-canonical G–A and A–G pairs occur as often as wobble (U–G / G–U) base pairs. Some pairs occur (almost) never (like non-cWW U–U). In total 294 713 base pairs are annotated in the FR3D as of June 2012.

The most well-known type of base pairing,that of the four (thereby canonical) pairings (G-C, C-G, T-A, A-T) in the DNA double helix, is but one of several ways to form a base pair.

In the model and notation introduced by Leontis and Westhof (2001) for RNA base pairing (with U replacing T in RNA), each nucleic base presents three edges for hydrogen bonding: the Watson-Crick, Sugar, and Hoogsteen edge. Of those, the Watson-Crick edges are engaged in six canonical base pairings. The usual canonical C–G, G–C, A–U, and U–A, as well as the two wobble pairings G–U, and U–G. All other pairings, even if one or both edges are of the Watson-Crick type, are non-canonical. Further subdivision occurs by glycosidic bond orientation (cis or trans) and the local strand orientation (antiparallel or parallel). The complete description of a base pair using the pair itself, its bond, and its strand orientation leads to 12 main families (Leontis and Westhof, 2001, Table 1).

The non-Watson-Crick base pairs have received increasing attention in recent years (Leontis and Westhof, 2001; Leontis et al., 2002) due to their importance in intermolecular interactions and in the formation of RNA modules, which act as binding sites or induce compact folding (Leontis and Westhof, 2003).

Leontis et al. (2002) grouped base pairs according to the nucleotides and pairing edges involved. These *isostericity matrices* contain information on which base pairs may be replaced in an RNA structure without disturbing the structure. In particular, the four base pairs C-G, G-C, A-U, and U-A are isosteric in cis-Watson-Crick conformation and may be replaced with each other, while G-U and U-G are almost isosteric.

The importance of the non-canonical base pairs may be observed by determining how often they occur in PDB structures. The PDB (Berman et al., 2000) database contains crystal and NMR structures of bio-molecules. The FR3D database (Sarver et al., 2008) provides base pairing information for RNA structures from the PDB. Using this information, it is possible to create a list of all occurring base pairs, together with the pair type. Of the almost 300 000 base pairs annotated in the FR3D, roughly 1/3 is non-canonical, while

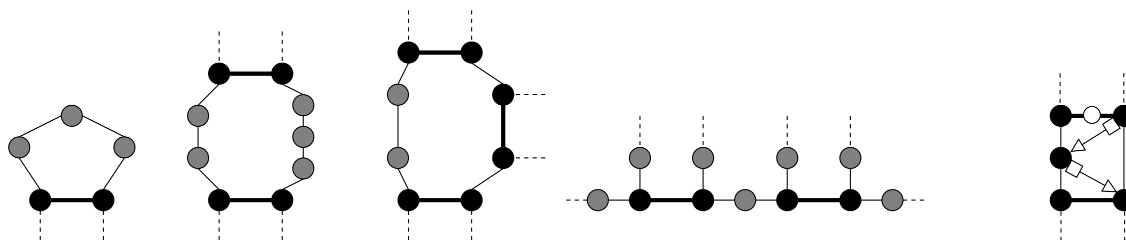

Figure 2.3: The loop types in RNA (extended) secondary structure. Black nodes indicate nucleotides engaged in base pairing, thick black edges indicate the pairing. Grey nodes are nucleotides in unpaired regions. Thin black lines indicate the backbone.
Hairpin loops are a stretch of unpaired nucleotides enclosed by a nucleotide pair.
Interior loops enclose two, possibly empty, stretches of unpaired nucleotides.
Multibranched loops are the join point for three or more helices. They may contain unpaired nucleotides in the loop.
The *exterior loop* joins the independent substructures of an RNA structure.
Extended secondary structures stand apart. Each nucleotide may pair with up to three other nucleotides (one being most common, three happening almost never). Each pair is annotated with the pair type. *Filled* annotations are in *cis* orientation, unfilled (white interior) means *trans* orientation. A *circle* denotes the Watson-Crick edge, a *triangle* the sugar, and a *square* the Hoogsteen edge (notation according to Leontis and Westhof (2001)).

almost 1/2 of the base pairs are cis-Watson-Crick `C-G`/`G-C` base pairs. A closer look at the table raises the question of which base pairs to include in structure prediction programs and which to exclude in order to keep the models simple. The two most often used choices are to include the six most common base pair types, or to allow all sixteen pairs to occur. Both choices omit the specific type of pairing. The former choice actually yields algorithms that consider only Watson-Crick pairings, as the other commonly occurring pairings are all of the non-Watson-Crick type.

Table 2.1 contains an unfiltered histogram of base pairing in the `FR3D` database. One problem when using a database like the `PDB` or `FR3D` is the quality of the data – as with other RNA structure databases as well. For the table all data was taken as is. Considerations regarding each individual RNA source structure are complex, touching crystallization or NMR data generation, cleanup, duplicate removal, and compounds of multiple RNAs and proteins, to name a few. Instead of justifying a particular filter, data is taken as found in the `FR3D` for the table.

Once a model is considered that more closely relates to the biochemical reality with three possible pairing edges, it also becomes possible to consider a nucleotide pairing with more than one partner. In reality this seems to happen quite frequently in regions of particular importance for activity or structure, namely in RNA structural modules (Sec. 2.4).

2.2 RNA Secondary Structure

The nearest-neighbor model in RNA secondary structure prediction considers loops of different types that each convey a specific energy contribution (Lu et al., 2006). The contribution by each loop is additive, leading to efficient dynamic-programming decomposition schemes (Zuker and Stiegler, 1981; Lorenz et al., 2011). Importantly, it is not the individual base pair that conveys a certain energy (or score for non-physics based models like CONTRAfold (Do et al., 2006)), but the stacking of two pairs on top of each other. In addition, unpaired regions also convey a certain energy. The different loop types are grouped by the number of branches joining the loop with other loops. Each join point is a shared base pair. Overall, the structure should provide a *minimal free energy*, where stacking pairs provide negative free energy, while unpaired regions positive free energy. Examples of the loop types are given in Fig. 2.3.

Hairpin loops are a contiguous region of unpaired nucleotides enclosed by a single base pair. They are connected to a single other loop at the base pair. Hairpins convey a positive energy that increases with the number of unpaired nucleotides, making large unpaired regions increasingly unlikely.

Interior loops are defined by two base pairs, an inner and an outer base pair. In between, two regions of unpaired nucleotides exist. For reasons of efficiency, most folding algorithms cap the number of unpaired nucleotides in the regions at 30. If one of the two regions contains 0 unpaired nucleotides, one speaks of a bulge, while two pairs stacking directly onto each other lead to a stacked pair. Stacked pairs convey negative energy while the other loops convey a penalty or positive energy. Longer regions of tightly stacked pairs lead to helices.

Multibranched loops join three or more helices. In the centre of the well-known cloverleaf shape of the `tRNA` structure is a multibranched loop, with four stems branching out from this central loop as can be seen in Fig. 1.1 (*left*).

Finally, each structured RNA is composed of one or more exterior loops. These loops are not enclosed by further base pairs and create individual self-contained structures in terms of the secondary structure of the RNA.

Many refinements of this general view are possible. First, one can improve the prediction accuracy for multibranched and exterior loops by considering coaxial stacking (Lu et al., 2006). Just as regular stacked pairs convey a beneficial energy term to the total free energy, so does stacking of the helices in multibranched loops. The resulting model is more complex as the optimal stacking for each of the loops needs to be considered.

Other refinements require adding completely new loop types to the model. The model presented in Chapter 7 is an extension of the loop model just described and adds base triplets as a new loop type. In a base triplet three nucleotides are engaged in a total of two base pairings. This requires that at least one non-Watson/Crick edge is engaged in the pairing of the twice-paired nucleotide. Depending on the local structural conformation, the other nucleotides are paired using any of the three possible edges.

Fig. 2.3 gives an example of base triplets in a stem. The figure does not provide an exhaustive enumeration of all possible base triplets (even irrespective of the actual

pair type	base pairs	base triplets	base quadruplets	base quintets (?)
number	261 842	15 288	761	3 (?)
fraction	0.942	0.055	0.003	–

Table 2.2: The *number of pair types* in the `FR3D` database built from `PDB` data.
Base pairs are two nucleotides paired with each other and no other nucleotides. This type of pairing is considered by algorithms like `RNAfold`, `CONTRAfold`, `Infernal`, and many others.
Base triplets are counted when a nucleotide is engaged in pairing with *two* other nucleotides. `RNAwolf` allows pairings of this type.
Base quadruplets saturate the number of possible pairings for a single nucleotide. Their low incidence makes them an object of less interest in predictions.
Base quintets are not considered in the Leontis and Westhof (2001) model. They could be a result of mis-annotations.
The total number, 277 894, is less than the number of base pair interactions as each base triplet accounts for two interactions (quadruplets for three) in Table 2.1.
(Data as of June 2012).

nucleotides involved) as their number is quite large. Figs. 3 and Figs. 4 of Höner zu Siederdissen et al. (2011) (Chapter 7) provide a grammar (see Chapter 3) with all possibilities for interior-loop like base triplets and how the decomposition affects multibranched loops. For interior loops, there are four outer cases for triplets, and two to nine cases how the next inner structure looks like.

As shown in Table 2.2, base triplets do occur in around 5% of the cases, and are often clustered in regions of non-canonical structure in the RNA. These regions, termed RNA structural modules, are explained in more detail in Sec. 2.4. The small number of base quadruplets, one nucleotide paired with three others, makes them hard to predict as not enough statistical information is available.

Another type of extension is the inclusion of `G-quadruplex` elements (Lorenz et al., 2012, 2013). `G-quadruplex` elements are stacks of 4 `G`s each, that lead to very stable local structures. These elements follow an energy model that makes them amenable to inclusion in secondary structure prediction tools.

2.3 Scoring Schemes, Optimal Structure, and the Partition Function

Given an RNA sequence x as input sequence, there are a number of useful calculations on the secondary structure level that can be performed, many of which were introduced in Chapter 1. Here, two essential algorithms are discussed. The notation is the one used by the stochastic context-free grammar community. It might come as a surprise to readers more acquainted with energy-directed folding. From a purely algorithmic standpoint, it is however *only* a question of notation. Energy-directed folding algorithms and energy-based optimization schemes can be implemented without problems in terms of formal grammars – as done in Chapter 7.

Some definitions are required. Given x the input sequence, a structure $S(x)$ is one of

the many structures into which x can fold. Each structure $S(x)$ is a set with elements $s \in S(x)$. The elements s are the loops discussed in Section 2.2. For any pair $s_1, s_2 \in S$ of loops, $s_1 \equiv s_2$ if and only if the nucleotides involved in the loops have the same indices and the structure is exactly the same. This constraint uniquely identifies each loop. The set $\Omega(x)$ is the set of all structures into which x can fold, so that $\forall S(x) : S(x) \in \Omega(x)$. As this text deals mainly with algorithms working on single input (sequences), a more succinct notation is possible. Whenever it is clear from the context, which S and Ω are discussed, their dependence on x is dropped.

For physics-based models like `RNAfold` a function $E(x, s)$ (again, x may be dropped if it is clear which x is meant) calculates the Turner nearest-neighbor energy for the loop s. The energy of a structural decomposition S is denoted by $E(x, S)$.

It follows that that the total energy of a structure S can be easily computed given a loop decomposition:

$$E(x, S) = \sum_{s \in S} E(s).$$

For non-physics based models, the score of a structure is calculated similarly.

An optimal structure with minimum-free energy (mfe) in a physics-based model is found by calculating the energy for each individual structure and choosing the structure with the minimal free energy:

$$\mathrm{mfe}(x) = \operatorname*{argmin}_{S \in \Omega} E(S).$$

Given the exponential number of possible structures S, efficient methods to calculate the minimum-free energy are required. The `CYK` algorithm (Section 3.1.3) allows finding the optimal structure efficiently in polynomial time for energy functions E that follow the nearest-neighbor model.

McCaskill (1990) calculated the partition function for a given sequence x, from which it is possible to calculate the probability with which any two nucleotides are involved in base pairing. The partition function is

$$Z(x) = \sum_{S \in \Omega} \exp(-E(S)/kT)$$

with T the temperature and k Boltzmann's constant. The algorithm calculating Z is called the `Inside` algorithm. The `CYK` parser can be used to calculate Z with minimum operations replaced by sums and sums by products.

Once the partition function has been calculated, the frequency of occurrence for individual structural features can be calculated. The base pair probability matrix in Fig 1.1 (*right*) for example provides the probability for each base pair (i, j) to be part of any structure in Ω. Base pairs with a high probability occur in many of the potential structures of an input sequence.

The algorithm can also be generalized. Given a loop construct $c \in (\bigcup S \in \Omega)$, its frequency of occurrence as a fraction of Z is calculated by

$$Z_c(x) = \sum_{S \in \Omega : \exists c \in S} \exp(-E(S)/kT).$$

The summation is over all structures S of the input x, where c occurs, c itself is one of the possible loops for x. In this way it is possible to calculate how often individual features of the structure space occur by calculating Z_c/Z for all structural features c of interest. For the McCaskill (1990) algorithm, one chooses the pairing of nucleotide i and j, instead of a loop construct.

In order to correctly calculate the partition function for energy-directed folding functions, the underlying algorithm needs[2] to be structurally unambiguous.

Both, the `CYK`- and the `Inside` variants of algorithms on RNA sequences will play a role in the upcoming discussions on ambiguity. This includes Section 3.1.4 with background on ambiguity and in Chapter 5 for semantics and ambiguity of RNA family models.

Ambiguity is also dealt with implicitly in Chapter 7 where a grammar for extended secondary structures is created with careful consideration to make it non-ambiguous.

2.4 RNA Structural Modules

RNA modules are small, highly structured regions of the RNA. They are typically around 20 nucleotides or less in size. Most of the structural modules studied in the literature (Leontis and Westhof, 2003; Theis et al., 2013) are interior-loop like. They are enclosed by an inner and an outer canonical base pair. Between the enclosing base pairs, a number of non-canonical interactions are formed. These interactions include non-Watson/Crick base pairs, and base triplets. The nucleotides can also form locally crossing interactions.

While it is impossible to model all of these complex interactions in an (extended) secondary structure model, it is however possible to encode parts of the structure of these modules. Base triplets and "zig-zag" patterns can be modeled with a nearest-neighbor like model. Such a model is presented in Chapter 7. The aim is to more accurately describe non-canonical regions in predicted RNA secondary structures. Apart from improved structure prediction, it becomes possible to model active parts of RNA and provide more information for tertiary structure prediction. Regions with many non-canonical bindings can also be involved in intermolecular binding. Fig. 1 in Chapter 7 gives an example of an interior loop-like region that is actually a complex non-canonical structure.

Some non-canonical interactions are implicitly included in the well-known Turner parameters (Mathews et al., 1999; Lu et al., 2006; Turner and Mathews, 2010), e.g. in small, tabulated interior loops with loop sizes up to four, as these loops give explicit energies for complete structural modules, albeit without regard to the inner base paired structure. The inherently three-dimensional structure of many larger RNA modules is, however, a problem outside of the realm of secondary structure prediction. Detection of structural modules is mostly done by comparative sequence analysis (Cruz and Westhof, 2011). Prediction of novel modules appears to be a mostly manual process as of yet, though some progress has recently been made (Theis et al., 2013).

[2] otherwise efficient calculations are impossible

```
CCGCCGU.A.GCUCAGCC.CGGG...AGAGCG.C.CCGGC.UGAAGACCGG.GUU.....
CCGGGGU.C.GCCUAGCC.UGGUCA.AGGGCG.C.CGGAC.UCAUAAUCCG.GUC.....
CGGGGGU.G.CCCGAGCCA.GGUCA.AAGGGG.C.AGGGU.UCAGGUCCCU.GU..GGC.
CGGGGGU.G.CCCGAGCCA.GGUCA.AAGGGG.C.AGGGC.UCAAGACCCU.GU..GGC.
CAGGGAU.A.GCCAAGUU.UGGCCA.ACGGCG.C.AGCGU.UCAGGGCGCU.GU..CCC.
CCAAGGU.G.GCAGAGUU.CGGCCC.AACGCA.U.CCGCC.UGCAGAGCGG.AAC.....
GGCCCAU.A.GCUCAGU...GGU...AGAGUG.C.CUCCU.UUGCAAGGAG.GAU.....
GGGCCAU.A.GGGUAGCC.UGGUCU.AUCCUU.U.GGGCU.UUGGGAGCCU.GAG.....
GAGAGCU.G.UCCGAGU...GGUCG.AAGGAG.C.ACGAU.UGGAAAUCGU.GU..AGG.
GGGCCUU.A.GCUCAGCU..GGG...AGAGCG.C.CUGCU.UUGCACGCAG.GAG.....
GCCCGUU.G.GUCAAGC...GGUU..AAGACA.C.CGCCC.UUUCACGGCG.GUA.....
GAGAAGU.A.CUCAAGU...GGCUG.AAGAGG.C.GCCCC.UGCUAAGGGU.GUA.GGU.
<<<<<<....<<<<.............>>>>..<.<<<<........>>>>.>.......

..................GU..CCGG.GGU.UCA.AG..UCCCC.G.CGGCGGC.A
..................UU..CCCG.GGU.UCG.AA..UCCCG.G.CCCCGGC.A
...GUA...G.....GCCUG..CGUG.GGU.UCA.AA..UCCCA.C.CCCCCGC.A
...GUA...G.....GCCUG..CGUG.GGU.UCG.AA..UCCCA.C.CCCCCGC.A
...GUA...G.....GGGUC..CGCA.GGU.UCA.AA..UCCUG.C.UCCCUGC.A
..................CC.CCGCC.GGU.UCA.AA..UCCGG.C.CCUUGGC.U
..................GC..CCUG.GGU.UCG.AA..UCCCA.G.UGGGUCC.A
..................A...CCCC.GGU.UCA.AA..UCCGG.G.UGGCCCC.A
GU.GAA...UA.GCGCCU....CAAG.GGU.UCG.AA..UCCCU.U.GCUCUCC.G
..................GU..CAGC.GGU.UCG.AU..CCCGC.U.AGGCUCC.A
..................A...CACG.GGU.UCG.AA..UCCCG.U.ACGGGUC.A
GU.GUA...A..GCGGCG....CGAG.GGU.UCA.AA..UCCCU.C.CUUCUCC.G
.......................<<<.<<...........>>>>.>.>>>>>>>..
```

Figure 2.4: Part of the `Rfam` `tRNA` family model rendered using Emacs `Ralee` mode (Griffiths-Jones, 2005). Individual RNA sequences have good alignment via sequence identity or compensatory mutations in the helical regions of the common secondary structure (last row). Non-structured regions show a considerable number of mutations, as well as deletions. The TΨC loop and stem (helix and hairpin in the lower block) are particularly well conserved.

2.5 RNA Families

RNA family models are used by `Infernal` in the form of covariance models to parametrize the generic search algorithm for potential homologous sequences of the RNA family (Chapter 5). Via the covariance model representation, it also becomes possible to compare two RNA families with each other. While the result of the comparison needs to be seen through the covariance model lens and can therefore be distorted, it is still possible to infer valuable information (Chapter 6). Finally, RNA family models provide a large source of statistical information for base pairing which can be used to train `RNAwolf` secondary structure folding parameters (Chapter 7).

Due to evolution, individual species' genomes are diverged from other species. It is possible, however, to create phylogenetic trees that describe how closely related one species is to another. In particular, if two sequences (be it protein-coding genes, or in this case DNA-encoded structural non-coding RNAs) from two species have a common ancestor sequence in the tree, assuming the tree is complete enough, the two sequences are homologs. Via sequence alignment (Deonier et al., 2005, p.143) it is possible to calculate

how similar they are and what operations to undertake to transform one sequence into the other.

It is, of course, possible to perform an alignment of any two sequences but with homologous sequences, one may discover similarity of sequences and elucidate structure. The alignment problem itself presents many questions, and is certainly not restricted to RNA, with many advances having been made in natural-language processing (Burkett et al., 2010).

An RNA family is an alignment of many RNA sequences together with a consensus structure. In general it is a very complex task to create a proper structural alignment. The basic algorithm to optimize the structure and alignment of just two ($k = 2$) sequences by Sankoff (1985) requires $O(n^{(3k)}) = O(n^6)$ time and $O(n^{2k}) = O(n^4)$ space and is thus too slow for input of longer length (n large) or the ten or more sequences (larger k) usually found in RNA family databases like `Rfam` (Griffiths-Jones et al., 2003).

`LocARNA` makes use of the idea to only consider significant base pairs, of which there is only a sparse set, to reduce the runtime of a Sankoff-style algorithm to $O(n^4)$. Significant base pairs are selected using the McCaskill (1990) base pair probability matrix for each individual sequence (Will et al., 2007, 2012).

The consensus structure of RNA family models leads back to the previously described base pairing interactions and homology of these sequences. It is to be expected that homologous sequences will not conserve sequence information due to mutation over longer evolutionary time scales but will conserve structure as loss of structure will mean loss of functionality (Leontis et al., 2002). Nucleotides in unpaired, functionally less important regions should mutate rather freely, those base paired will likely retain the same isostericity class for longer evolutionary time scales. As an example of different conservation patterns consider Fig. 2.4 with the more strongly conserved TΨC loop and stem.

From these patterns, one can first deduce the actual consensus structure, and then additional information. Programs like `Infernal` (Eddy and Durbin, 1994; Nawrocki et al., 2009) create stochastic models, called covariance models (CMs), based on a specific grammar. Covariance models can calculate for each substring of a genome how likely it is that said substring belongs to the family. These stochastic models find homologs of the RNA sequences in the family. The same RNA families can be used to train stochastic models based on the grammar described in Chapter 5. This grammar tries to improve the sensitivity of RNA homology search for remote family members. It does so by making explicit the notion of a *trace* in alignments. When a number of deletions or insertions follow each other in an alignment, it is not possible to say in which order they were accumulated and usually there is no reason to. Traces in alignments will be more formally explained in Chapter 5 (II.E).

A final piece of biological information that is important, with regards to how RNA families are handled by `Rfam`, are clans (Gardner et al., 2011). As sequences diverge more and more, a single common model for the sequences can become infeasible due to too much sequence or even structural variation. The approach taken by `Rfam` is to simply create multiple families and group such families in a clan, providing a meta-family.

Chapter 3

Formal Grammars and Algebraic Dynamic Programming

My own "mental model" of DP does not involve grammars.
anonymous referee commenting[1] on "sneaking around *concatMap*"

Dynamic programs can be developed using a variety of notations. Examples in text books like *Introduction to Algorithms* (Cormen et al., 2001) are normally written using explicit recursions. Groups associated with the `Vienna RNA` package tend to prefer a graphical notation (Bompfünewerer et al., 2008) and (tree) grammars have been used as well (Giegerich and Meyer, 2002).

There is no one true notation that fits all problems and requirements. Depending on the problem and the audience – see above quote – a different language might be required. For dynamic programming algorithms on (single) sequences, formal grammars are very convenient in that they partition an input sequence from left to right. Furthermore, this partition can be depicted graphically.

In this chapter, formal languages and grammars are described with an emphasis on grammars for RNA sequences. This background, especially the introduction to ambiguity, forms the basis for the work in Chapter 5, where different kinds of ambiguity in the context of stochastic RNA family models are considered. Ambiguity in stochastic models can be a serious problem as the parse with the single highest probability might not correspond to the structure with highest probability.

Formal languages are also used to describe all algorithms in Chapters 5–7. For the extended RNA secondary structure algorithm, a graphical notation was chosen, but it is a graphical notation depicting a grammar. Together with the `ADPfusion` framework presented in Chapter 8, formal languages are elevated from the level of a modelling tool to a domain-specific language in which the algorithms presented in this thesis can be written, compiled, and executed.

Section 3.2 introduces the basic concepts of `ADP`. These concepts form part of the `ADPfusion` (Chapter 8) formalism which is a dialect of `ADP`. While programs have to be

[1] Giegerich and Meyer (2002) started a beautiful tradition of quoting referees

ported to `ADPfusion`, the performance gains are considerable, and most of the original theory developed for `ADP` carries through.

3.1 Regular and Context-free Grammars

This introduction to formal grammars is necessarily quite short. Grune and Jacobs (2008) cover many fields, including formal grammars and parsing, as well as including a list of several hundred research articles on all topics with further information.

3.1.1 Alphabets and Strings

An alphabet A is a set of (atomic) characters. The set A^* contains all strings over A, including the empty string, typically denoted ϵ. The characters forming the set A can generally be composed of any finite set. A string is a (possibly empty) sequence of characters.

Some example sets are the set $\{a \dots z\}$ of lower-case `ASCII` characters, the set of all `ASCII` characters (including those that can not be printed), the nucleotide alphabet $\{A, C, G, U\}$ for RNA sequences or $\{A, C, G, T\}$ for DNA sequences. The individual characters can, of course, be compound objects as well. A grammar describing proteins could use an alphabet of the (20+1) amino acids, or the amino acids could be stored in codon form of 64 triplets of DNA sequence characters. The `CMCompare` algorithm (Chapter 6) uses an (implicit) alphabet of the states, or subtrees, of a covariance model (RNA family model).

Generally speaking, the set of characters of the alphabet is defined according to the problem at hand. While the characters are atomic with regards to the alphabet, their encoding can be rather complex.

3.1.2 Formal Grammars

A formal grammar $G = (N, \Sigma, R, S)$ is composed of a set of symbols $(N \cup \Sigma)$ and a set of production rules R. A symbol is either a non-terminal symbol N or a terminal symbol Σ and the intersection of the set of non-terminals and the set of terminals is empty ($N \cap \Sigma = \emptyset$). One of the non-terminal symbols $S \in N$ is designated as the start symbol.

Strictly speaking, the set of terminal symbols Σ forms the alphabet A, though it is common to extend Σ to allow for non-empty strings over A to be considered terminal symbols. It is always possible do define new non-terminal symbols and production rules instead of using the extended notion of terminal symbols.

The set of production rules are rules of the form

$$(N \cup \Sigma)^+ \to (N \cup \Sigma)^*.$$

Each production rule transforms a sequence of terminal and non-terminal symbols into another sequence of terminal and non-terminal symbols. Each of the grammar definitions

given by Chomsky (1956, 1959) restricts the left- (LHS) and right-hand side (RHS) of the production rule.

Below, two of the more restrictive formal grammar types are described in more detail, regular and context-free grammars. These types of grammars have efficient implementations and can express many of the RNA bioinformatics algorithms mentioned in Chapter 1.

General grammars allow any number of non-terminal and terminal symbols to occur on the left- and right-hand side of the production rule, subject to the constraint that at least one non-terminal occurs on the LHS. Both, regular and context-free grammars allow only a single non-terminal symbol on the LHS, and, depending on the grammar type, different symbols on the RHS.

A *stochastic* grammar (N, Σ, R, S, P) extends a grammar with probability annotations $p_i \in P$ for each rule $r_i \in R$. Each rule $r_i = \text{LHS} \rightarrow_{p_i} \text{RHS}$ is extended such that $0 \leq p_i \leq 1$ and the p_k for the same LHS sum to 1.

For many RNA bioinformatics applications, context-free and regular grammars have sufficient expressive power. There are some notable cases, where a context-free grammar is not expressive enough, including the construction of grammars for pseudoknotted RNA structures (Rivas and Eddy, 2000). In this thesis, only context-free grammars and their stochastic extensions are of interest.

Context-free Grammars

Context-free grammars restrict the set of rules R such that the LHS of each rule contains only a single non-terminal. The set of terminals and non-terminals on the RHS is unrestricted. Each rule $r_i \in R$ is of the form $r_i = N \rightarrow (N \cup \Sigma)^*$.

The Sum of Digits Grammar

The sum-of-digits grammar (N, Σ, R, S) from Sec. 2 in Chapter 8 has the following formal description (with S as start symbol):

$$
\begin{aligned}
N =& \{S\} \\
\Sigma =& \{ \texttt{'0'},\ \ldots,\ \texttt{'9'},\ \texttt{'('},\ \texttt{')'} \} \\
R =& \{ S \rightarrow \texttt{'0'} \\
&, S \rightarrow \texttt{'1'} \\
&, S \rightarrow \vdots \\
&, S \rightarrow \texttt{'9'} \\
&, S \rightarrow \texttt{'('}\ S\ \texttt{')'} \\
&, S \rightarrow S\ S \\
&\}
\end{aligned}
$$

It uses a single non-terminal symbol S which is also the start symbol. The ten digits, together with the opening '(' and closing ')' bracket form the set of terminals. The set of

rules R gives all possibilities on how to transform the non-terminal S into a sequence of terminal and non-terminal symbols.

The first ten rules produce a single digit from the non-terminal S. The next-to-last rule brackets the non-terminal S, while the last rule $S \to SS$ splits the non-terminal into two independent parts.

Linear and Regular Grammars

A linear grammar is a context-free grammar with at most one non-terminal on the right-hand side of each rule:

$$N \to \Sigma^* N^{0,1} \Sigma^*.$$

If the non-terminal on the RHS is the left-most symbol, the production rule is left-linear, if it is the right-most symbol, the production rule is right-linear. If the non-terminal is flanked by terminal symbols on the left and the right side (always speaking of the RHS), then the production rule is just linear.

A grammar with completely left-linear or alternatively completely right-linear rules is a (left-) or (right-) regular grammar, while those containing both left- and right-linear rules are linear grammars. Regular grammars are even more restricted than linear grammars but allow for more efficient parsing.

While less powerful than context-free grammars, due to the restriction to at most one non-terminal on the RHS, they can be used to model alignments of sequences, when using multiple tapes or input sequences. Grammars with more than one input sequence, or tape, will be mentioned as a future extension in the conclusion (Chapter 9).

Generating Sequences Belonging to a Grammar

The generation of a sequence according to the rules of a grammar and trying to parse a sequence with a given grammar can be seen as dual operations.

When generating a sequence, one starts with the start symbol S and chooses one of the possible productions. Using the sum-of-digits grammar from above, generation could start with $S \to$ `'('`S`')'`. Again, the non-terminal needs to be replaced: $S \to$ `'('`S`')'` $\to$ `'('`SS`')'` $\to$ `"(("`S`")("`S`"))"` $\to$ `"((1)(3))"`. Or shorter: $S \rightsquigarrow$ `"((1)(3))"`. Here, consecutive terminal characters `'(''3'')'` are represented as a string `"(3)"`.

In case of a stochastic CFG where each rule is given a probability one can stochastically generate sequences belonging to the language defined by the grammar. These sequences are then drawn according to their probability.

For compilers parsing the input text, or RNA secondary structure prediction algorithms trying to determine the secondary structure, the second option, namely parsing, is of more interest.

In the case of parsing, the principle idea is to reverse the arrows from above. Given `"((1)(3))"` one wants to recover the full path leading back to S.

	(	(	1	)	(	3	)	)
(	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$	D
(		$\ddots$	$\ddots$	B_1	$\ddots$	$\ddots$	C	$\ddots$
1			A_1	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$
)				$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$
(					$\ddots$	$\ddots$	B_2	$\ddots$
3						A_2	$\ddots$	$\ddots$
)							$\ddots$	$\ddots$
)								$\ddots$

Table 3.1: Memoization of parsing the input `"((1)(3))"` for the sum-of-digits grammar. Parsing starts at the main diagonal and advances to the upper right corner. The single characters `'1'` and `'3'` in cells A_1 and A_2 are parsed first. All other parses on this diagonal fail as no rule can successfully parse a single opening or closing bracket. Parses B_1 and B_2 succeed next. They parse one opening and one closing bracket around the successful A parses. The rule $S \rightarrow SS$ is parsed in step C. Finally, in step D, a final opening and closing pair of brackets envelop the successful C parse. As the parse at D succeeds at the complete input $(1, L)$, with L the length of the input, the complete parse of the input succeeds.
The parse tree for the input given the sum-of-digits grammar can be recovered using backtracking.

3.1.3 The CYK Parser

For the applications considered here, the `CYK` parser (Grune and Jacobs, 2008, Sec. 4.2) is the most important parser. It parses an input sequence "bottom-to-top". For each individual non-terminal, the `CYK` parser keeps a memoization table that contains the parse result for the substring (i, j), also called the subword (i, j). Starting from the single-character substrings $(i, j), \quad i == j$ all substrings are tried in increasing substring length[2].

The final parse result is memoized in the memoization table for the start symbol S at the index for the full string $(1, L)$, where L is the length of the input.

In Table 3.1 the memoization table for the sum-of-digits grammar and the input `((1)(3))` is given. The memoization table is of upper-triangular form as all subwords (i, j) have the constraint that $i \geq j$. Parsing proceeds from small to large subwords via the main diagonal and consecutive diagonals to the upper-right subword $(1, L)$. For the sum-of-digits grammar, it is only important whether a particular input can be parsed at all. The succeeding subparses A_1, A_2, B_1, B_2, C, and D can, for example, be indicated with a boolean flag.

Other grammars, like the grammars for the `Nussinov78` (Nussinov et al., 1978), `RNAfold` (Lorenz et al., 2011), or `RNAwolf` (Chapter 7) algorithms store counts, ener-

[2] ADP uses slightly different indexing where (i, i) denotes the empty substring

gies, or pseudo-energies (scores in general) in the memoization tables. These grammars will also fill the complete memoization table with scores. An RNA folding grammar will typically succeed for any input, from the empty input to any finite string[3]. What changes is the final score retrieved at the symbol S and subword $(1, L)$.

That is, for RNA grammars, there is an exponential number of succeeding parses, each with a different final score.

Bellman's Principle of Optimality

With the `CYK` parser it is possible to calculate the optimal parse of an input given a grammar and a scoring function, given that certain conditions hold. Bellman's Principle of Optimality (cf. Lew and Mauch (2006) for a general introduction to DP and Bellman's principle) is the driving principle behind dynamic programming. For dynamic programming to work and Bellman's principle to hold, an optimal structure has to have optimal substructures from which the optimal structure is built up.

In terms of the optimality criteria to be used by dynamic programming algorithms, it is required that an optimal solution to an input can be split into small sub-inputs and for each sub-input the optimal solution is calculated recursively. From the optimal solutions for sub-inputs one can then calculate the optimal solution to the whole input.

For the `CYK` parser this is directly required and supported as the optimal solutions to sub-inputs (substrings) are calculated first, followed by ever larger substrings.

In general, all problems considered here conform to Bellman's principle. The nearest-neighbor loop model sums up scores (energies) for individual loops, with the individual summands being independent. Each loop type corresponds to one non-terminal in the RNA folding grammar. The same holds for the `Infernal`-related algorithms. RNA family models do not behave differently in this regard to RNA folding algorithms, except that the grammars have many more non-terminal symbols.

Backtracking in Memoization Tables

For the sum-of-digits grammar it is possible to explicitly store in the "forward" table-filling phase which succeeding rules lead to storing a success value in the memoization table. One can, say, store at D in Table 3.1 that the succeeding parse involved the parse at C via the rule $S \rightarrow (S)$.

For scoring parsers (consider `Nussinov78`, Chapter 8, Sec. 7.1) this is not possible as there is an exponential number of succeeding parses for the input, each with a different score.

Given the production rules and the different scores associated with each production, one can, nevertheless, produce the parse tree for the highest-scoring parse or any number of parses within a certain threshold. Returning the optimal parse yields the structure associated with the best score, as in minimum-free energy structures of `RNAfold` (Lorenz et al., 2011), while returning all parses within a threshold gives complete sub-optimal structures (Wuchty et al., 1999).

[3] given enough resources

Backtracking in these cases works as follows. One starts with the optimal score. The optimal score for the sum-of-digits example (Table 3.1) is given in D. One now tries, successively, all rules for the non-terminal (S) until one matches that succeeds and produces the result D.

In the next step, taking the right-hand side of the succeeding rule ($S \to SS$) it is now required to "backtrack" or step down into the succeeding parts of the rule. Terminal symbols lead to the emission of the matching character, while for non-terminal symbols the backtracking has to be done recursively to determine which rules succeeded in generating the score stored for them.

Backtracking stops once all recursive steps down toward the main diagonal encounter only terminal symbols. The terminal symbols spell out the input sequence while the tree structure of the recursive descent produces the parse tree.

In algorithms like `Nussinov78`, sub-trees are associated with different semantics. The rule $S \to aSb$ (compare to the sum-of-digits rule $S \to (S)!$) denotes base pairing between nucleotides a and b. Depending on the pair (a, b) a different score would have been assigned to the production rule in the forward phase.

The resulting parse trees can be transformed into visual representations. The Vienna dot-bracket strings for RNA secondary structure are such a representation.

If one replaces the terminal digits in the sum-of-digits grammar with a single terminal '`.`' (dot), then the grammar parses RNA secondary structures in dot-bracket format that are non-empty and contain at least one dot between the inner-most pair of brackets.

3.1.4 Syntactic, Structural, and Semantic Ambiguity

Formal grammars can be ambiguous or non-ambiguous. If a grammar is declared non-ambiguous one needs to take into account what kind of ambiguity is considered.

The kind of ambiguity that is easiest to understand is *syntactic ambiguity.* A grammar is syntactically ambiguous if there is more than one parse for a given sequence. A grammar for a programming language should be syntactically non-ambiguous. If that is the case then for any "correct" program that was typed in there is only one abstract syntax tree (AST). An AST is one of the representations of a program before it is turned into assembler. If there were more than one AST for a correct program, each with a different "meaning" (the AST is the meaning of a program), which AST should then be chosen?

The problem is actually a bit less severe than it seems. The sum-of-digits grammar used as a running example in this chapter (and in Chapter 8) is syntactically ambiguous. For certain kinds of algebras (see below) – or evaluations of the AST – ambiguity doesn't matter. Summing up non-negative digits gives the same result in any order of summation. For more complicated algebras, say all four basic mathematical operations, and negative as well as positive numbers in the AST, different parses might easily lead to different meanings.

All useful RNA secondary structure prediction grammars are syntactically ambiguous. For each input sequence there is an exponential number of different structures $S \in \Omega$, each with a different score $E(S)$ (E calculating an energy in case of `RNAfold`; see Section 2.3).

Structural ambiguity, as explored by Dowell and Eddy (2004) for several small RNA secondary structure grammars, is a bit more tricky. A grammar is structurally ambiguous if there exists an input x such that there are two (or more) different parses $S, R \in \Omega$ and both S and R yield the same set of loops. In other words, if S and R are different parse trees but annotate the sequence with the same loops at the same positions, then the grammar is structurally ambiguous.

The sum-of-digits grammar is structurally ambiguous. As an example, consider the input `(1)(2)(3)`. The production rule $S \rightarrow SS$ (with non-terminal S) splits the input in two ways: `(1)  (2)(3)` and `(1)(2)  (3)`. This leads to two different parse trees but the same semantic meaning if digits are to be summed up.

Structural ambiguity may be more severe in other grammars. One version of the `Nussinov78` grammar, used as an example grammar in Chapter 8, has the following production rules (with non-terminal S, start symbol S, and terminals a, b):

$$S \rightarrow \epsilon$$
$$S \rightarrow aS$$
$$S \rightarrow Sa$$
$$S \rightarrow aSb$$
$$S \rightarrow SS$$

The reader may notice the following derivation is possible:
$S \rightsquigarrow SS \rightsquigarrow (S \rightarrow \epsilon)S \rightsquigarrow S \equiv S \rightsquigarrow S$. This gives an infinite number of parse trees due to the existence of the $S \rightarrow \epsilon$ rule. These problems require special care when designing formal grammars.

As Dowell and Eddy (2004) note, structural ambiguity is not relevant for `CYK` algorithms calculating the minimum-free energy for an RNA secondary structure, or calculating the partition function Z for statistical models via the `Inside` algorithm. For complete sub-optimal enumerations (Wuchty et al., 1999) or the partition function calculations for energy-based models (McCaskill, 1990), structural non-ambiguity is an absolute requirement if the algorithms are to remain efficient.

An alternative, ambiguity compensation (Chapter 5), does exist. It is, however, infeasible to use for even small-sized input. As structural ambiguity means that several parses produce the same (secondary) structure, one can, in principle, sum over all parses producing the same structure. Unfortunately, the structure space has exponential size, making this approach impossible.

Semantic ambiguity finally generalizes the structural ambiguity problem and considers (non-) ambiguity under different semantics. For RNA secondary structure the kind of semantic ambiguity discussed is structural ambiguity. RNA family models consider not only secondary structure prediction, but *align* a sequence against a fixed (for each family model) consensus secondary structure. Aligning a sequence against another sequence or against a structure opens up questions of alignment ambiguities – different alignments with

$$
\begin{aligned}
\text{digit} &: & \mathcal{D} \rightarrow & & S \\
\text{bracket} &: & '(' \times S \times ')' \rightarrow & & S \\
\text{split} &: & S \times S \rightarrow & & S \\
\text{h} &: & \{S\} \rightarrow & & \{S\}
\end{aligned}
$$

Figure 3.1: *Signature* for the sum-of-digits problem in Algebraic dynamic programming methodology. Each of the rules is given an explicit name. To simplify notation, the function digit expects a single terminal $d \in \{0, \ldots, 9\} = \mathcal{D} \subset \Sigma$. The symbol S not only denotes the non-terminal symbol in the grammar but also the value domain of the symbol.
The function h is not part of the set of rules of the sum-of-digits grammar but introduces the possibility of an objective function. The objective function accepts a set of parses (rather, their representation in terms of the value domain S) and returns a subset of those.

the same biological meaning or semantics. In Chapter 5, the different possible semantics of aligning a sequence against a family model and their ambiguities are given an in-depth treatment.

3.2 Algebraic Dynamic Programming (ADP)

Algebraic dynamic programming (Giegerich and Meyer, 2002; Giegerich et al., 2002, 2004) provides formal methodology and a framework for dynamic programming over sequence data. This short introduction necessarily repeats and combines parts of Chapter 5 (II.B) and Chapter 8 (3). Most of the notation and definitions originally given by the above authors carry through to the `ADPfusion` implementation. As `ADPfusion` is a dialect of `ADP`, programs have to be adapted before they can make use of the new framework. The gains are, however, substantial as shown in the benchmarks in Chapter 8.

3.2.1 Methodology

On the formal side, Algebraic dynamic programming (ADP) considers the different aspects of a dynamic program individually, thereby simplifying the design process. An algebraic dynamic program consists of a *signature*, a *grammar*, an *algebra*, and a concept of *memoization*. Implementations then provide the machinery to combine these parts into a single algorithm.

The *signature* can be seen as an interface, a supply of function symbols, or an ordered set of typed functions. The signature for the sum-of-digits example from above is given in Fig. 3.1. Each of the production rules of the grammar is given a type in the signature. The signature thereby explicitly fixes the function types to be used to evaluate parses induced by the production rules. Note that function signatures are to be read RHS-to-LHS compared to production rules. The production rule $S \rightarrow SS$ yields the type

```
sum-of-digits (digit,bracket,split,h) = s where
  s = digit   <<< aDigit                        |||
      bracket <<< char '(' -~~ s ~~- char ')'   |||
      split   <<<          s ~~~ s              ... h
```

Figure 3.2: The sum-of-digits grammar in ADP notation. `aDigit` is a parser that parses a single digit, producing nothing otherwise. The `char` parser parses the single character it has been given. The non-terminal `s` is recursively defined in this grammar. Combinators guide the correct deconstruction of the input into substrings. The `(-~~)` and `(~~-)` combinators remove the left- or right-most character using the given parser. The `(~~~)` combinator provides all possibilities of splitting a string into two substrings at a common split point. Evaluation functions (`digit,bracket,split`) are applied to the terminal and non-terminal symbols using the `(<<<)` combinator. Different production rules are combined using the `(|||)` combinator. Selection of the optimal parse is done by applying the objective function `h` using the `(...)` combinator.

split : $S \times S \to S$. This is quite natural as in CYK parsing the parsing flow in each rule is RHS-to-LHS.

Grammars are very similar to formal grammars in notation but are augmented with additional symbols, or combinators, used to compose the grammar. Each grammar is composed of a set of production rules, where one of the signature functions is applied to the symbols forming the right-hand side using the `(<<<)` combinator function. Production rules for the same non-terminal are combined using the `(|||)` combinator which concatenates the parses for the two rules. A list of parses is reduced to the optimal choice using the objective function `h` from the signature applied to the list of parses using the `(...)` combinator. The explicit notion of an objective function that may be changed, together with the possibility of changing the other signature functions, makes it possible to use one grammar with many evaluation methods. This means that the grammar only describes the search space of all possible parses, while the evaluation of individual parses is delegated to algebras. The sum-of-digits grammar written using ADP notation is shown in Fig. 3.2.

An *algebra* is an actual set of functions describing an optimization procedure for a grammar that conforms to the signature. Continuing with the sum-of-digits example, Fig. 3.3 gives two algebras that produce both, different results and optimization according to different criteria. There are two types of functions. The objective function `h` takes a list of parses and keeps parses according to the stated criterion, e.g. just the maximal parse(s). All other functions describe the semantics of individual production rules. That is, say, rule $S \to$ `'0'`|...|`'9'` is the production rule[4] for a single character but the algebra function `digit` in the `sumDigits` algebra provides a way to parse a single character `'0'` ... `'9'` to produce the number 0 ... 9. Depending on the algebra, the value domain changes as well. `sumDigits` is an algebra with the `Int` value domain and sums up digits. `treeDigits` on the other hand produces `ParseTrees`, actual parse tree representations devoid of any semantic meaning.

Finally, *memoization* provides a way to reduce the running time for the parser from

[4]using shorthand notation for ten rules $S \to$ `'0'` to $S \to$ `'9'`

```
sumDigits :: Signature
sumDigits = (digit,bracket,split,h) where
  digit       d     = parseDigit d
  bracket '(' s ')' = s
  split    l     r  = l+r
  h xs              = [maximum xs]

data ParseTree
  = Digit Char
  | Bracket Char ParseTree Char
  | Split ParseTree ParseTree

treeDigits :: Signature
treeDigits = (digit,bracket,split,h) where
  digit       d     = Digit d
  bracket '(' s ')' = Bracket '(' s ')'
  split    l     r  = Split l r
  h xs              = xs
```

Figure 3.3: Two possible algebras for the sum-of-digits grammar. An algebra specifies how individual parses are to be evaluated (`digit,bracket,split`) and how to select an optimal parse (`h`).
The `sumDigits` algebra collects individual digits, parses the digit characters into numbers, and sums them up. The second algebra is a unique feature of ADP. In ADP it is easy to return all parses independent of their evaluation in the form of a parse tree representation, which is done with `treeDigits`.

one that is typically exponential to one that is polynomial. For memoization to be correct in the setting of an optimizing `CYK`-style parse, Bellman's principle needs to hold. In case that it does hold, it is necessary to memoize for each non-terminal the optimal parse(s) for each substring (or subword (i, j)) that has already been parsed. As each subword is parsed many times during a `CYK` parser run, it pays off to store just the parse result instead of having to create the complete parse tree starting at each non-terminal and each substring index.

Parse Trees and Bellman's Principle

The ADP idea of explicit parse trees seems minor at first, considering that the `CYK` algorithm recovers parse trees during backtracking quite easily. The importance of the idea becomes clear if one considers what happens if the objective function `h` is replaced by the identity function that just returns all parses. In that case, running the grammar and algebra combination on an input produces all parse tree representations explicitly. A parse tree representation is a parse tree, where the production rule evaluation functions have been applied to all tree nodes. For the `sumDigits` algebra, this is the sum of all digits, while for `treeDigits` it is an explicit tree representation.

If Bellman's principle holds for a certain problem, like sum-of-digits, it does not matter if the objective function is applied to the list of all parses, or if individual sub-parses are pruned. At least it does not matter for calculating the optimal parse. It *does* matter in terms of performance. In this way, ADP makes application of Bellman's principle explicit. This allows (for small inputs) checking if the principle holds for a certain grammar and algebra combination. Comparing the result of applying the objective function interleaved with sub-parses or to all possible parses should yield the same set of final results.

3.2.2 Implementations of the ADP Idea

The section above on methodology is silent on an actual implementation. This is because there are currently at least three main branches of the ADP idea. The original implementation of ADP in Haskell, a new language (`GAP language`) and compiler (`GAP compiler`) implemented in `C++`, and `ADPfusion` (Chapter 8) which is again written and embedded in Haskell.

The initial work on ADP was done in Haskell (Giegerich and Meyer, 2002; Giegerich et al., 2002, 2004). The choice of Haskell allowed for a terse description, elements of which can be found in Fig. 3.2 and Fig. 3.3. Haskell is popular as a language for embedding domain-specific languages like ADP, as it is possible to write parsers and define novel syntax (symbol names, infix notation symbols) easily. One problem of ADP in Haskell was the overhead in running time and memory consumption and the choice of a rather obscure host language.

A first step toward better run times was the introduction of an ADP-to-`C` compiler (Steffen, 2006). The compiler took ADP as an abstract language (as opposed to ADP embedded in Haskell – the abstract ADP language and its embedding just happened to have the same syntax) and turned grammars and algebras into efficient `C` loop constructs.

A number of problems with (Haskell-) ADP remained. Errors were hard to decipher for non-specialists, an abundance of infix operators was required, and sub-optimal performance in some cases; to name a few (Sauthoff et al., 2011). One solution to these problems is a novel language, the `GAP language` and a compiler for this language, the `GAP compiler`. The `GAP` language is a domain-specific language with `Java`-like syntax and not embedded in any particular host language. It comes with its own compiler and supporting library. It has a feature set that makes it possible to write many dynamic-programming algorithms, especially some of those discussed in Chapter 1, without having to use a mainstream programming language. In addition, the `GAP` environment produces programs whose performance characteristics are competitive to those of writing directly in, say `C` (Sauthoff, 2011; Sauthoff et al., 2011, 2013).

One problem with a language like `GAP` that provide their own compiler is that it is, in general, hard to integrate new features. One either needs to extend the compiler or an interface to a foreign language needs to be available.

`ADPfusion` is another approach to implement ADP. The first version, described in Chapter 8 is very close in style to the original ADP work, but offers substantially better performance even though both approaches are embedded in Haskell. The choice of embedding `ADPfusion` directly in Haskell makes it possible to treat it as what it is – plain Haskell. The main benefit is that whenever a feature is not present in the domain-specific language one can easily escape to the host language without having to wait for the developer of the domain-specific language to implement the missing feature. Experimentation with new features and ideas is promoted as they are simple to integrate. Performance is also acceptable and almost on par with direct implementations in `C`. No comparison with `GAP` programs was made at the time of writing the paper in Chapter 8. Some preliminary results presented in Chapter 9 indicate that `ADPfusion` is indeed faster than `GAP`-based implementations. The newest version also removes the need for combinators like `(-~~)`. This points out that the `ADPfusion` framework can serve as a backend for different domain-specific languages to provide the performance benefits of fusion.

The work presented in Chapter 8 makes it possible to (re-) write, the new grammar for stochastic RNA family models (Chapter 5), and the `RNAwolf` algorithm (Chapter 7) in a high-level style directly as a context-free grammar using `ADPfusion`. The (heuristic) tree alignment of pairs of RNA family models using `CMCompare` explained in Chapter 6 requires further advances of the fusion approach.

Chapter 4

Efficient Algorithms in a Functional Language

This Chapter is the third part of the triad of biological background (Chapter 2), grammatical background (Chapter 3), and efficient implementation in a functional language (here). Its main purpose is to provide the necessary background to understand the design of the `ADPfusion` library (Chapter 8).

In recent years it has become apparent that functional programming is not only an efficient tool where efficiency is the time required to *implement* an algorithm, but that functional programs can also be competitive with the runtime performance of equivalent programs written in `C`, `Fortran`, and `C++`.

All algorithms described in Chapters 5–7 were implemented in the functional programming language Haskell. The primary reason was to provide a working prototype in a short amount of programmer time. The presented algorithms do have clearly valid use cases where a “slow prototype” is not enough. The `CMCompare` based comparison of RNA family models has an obvious application in comparing all pairwise combinations of `Rfam` models with each other. For the newest version of the `Rfam` database this requires more than 2.4 million comparisons. The `RNAwolf` algorithm for extended secondary structures should be usable in the same way as `RNAfold` is being used. This includes producing secondary structure predictions for a large set of input sequences.

The techniques discussed in this chapter are used to help the programmer write efficient code in Haskell. The key idea is that the programmer continues to write high-level code that promotes thinking about the algorithmic problem to be solved. The compiler will *automatically* and invisible to the programmer translate the (inefficient) high-level code into efficient low-level constructs that are semantically *equivalent*. This guarantees that the optimized code behaves in the same way as the programmer-written code. In this regard the most significant point is that *all* of the optimizations discussed in Section 4.2 are automated and require no user intervention. The techniques presented in Section 4.2 provide such automation for list- and vector-based code and are used for `RNAwolf` (Chapter 7), while `ADPfusion` (Chapter 8) extends this automation to *full dynamic programming algorithms* and is based on the ideas discussed here.

One can go a step further, in that the programmer does not even have to *know* how these optimization methods work. While this goal has not been reached quite yet for a number of corner cases, the typical user can write the functional-language analogs of loop constructs – namely list-based processing and recursive functions – and expect good performance.

4.1 Being Lazy with Class

Why Haskell? Computational biology is dominated by imperative, low-level (`C`, `C++`) or scripting high-level (`Perl`, `Python`, `R`) languages (Fourment and Gillings, 2008). Haskell provides a curious combination of high-level programming (up to and including features found in theorem provers) and high performance (Keller et al., 2010). The result are programs that are more terse with less lines of code than a comparative, say `C`, program, with a tendency for fewer bugs.

The main features of Haskell are it's design to be a pure, lazy, functional programming language. In "Being Lazy with Class" Hudak et al. (2007) describe the roots and history of the language, but also the design decisions that were made. For the user these features yield a style of programming rather different from what is known in the imperative world.

Purity is the absence of side-effects. A function in Haskell resembles the notion of a mathematical (total or partial) function. The behavior of a Haskell function is *completely* determined by its input variables and will always produce the same output given the same input. Purity in itself leads to programs that are more easy to reason about as there is no hidden state that can influence how functions behave.

The absence of effects can also be rather inconvenient. Input and output are impossible and purely functional data structures (Okasaki, 1999) and algorithms (Bird, 2010) are sometimes quite different from established ones (Cormen et al., 2001), or have worse asymptotic runtime, though laziness can help (Bird et al., 1997). The idea of a *monad*, further described below, will alleviate *all* of these problems in one way or another.

Non-strictness, call by need, or laziness is the evaluation at the last possible moment, exactly when a value is needed. This feature tends to be controversial. It makes it hard to reason about programs in terms of space (Hudak et al., 2007; Okasaki, 1999) as well as running times. Combined with I/O, scarcity of computer resources (like handles) can become a problem. Programs do become more compositional on the level of individual functions, however, offsetting performance problems with more possibilities to combine and compose existing code (Hughes, 1989).

A functional programming language treats functions as first class citizens and puts "the function" in the center of attention, not "a procedure" or "an object". Functions as first class citizens means that (partially applied) functions can be stored in data structures just like values, can be arguments to other functions, and, in general, there is no difference between handling a value or a function. Except, of course, that at some point functions are applied yielding a result. This leads to the next difference between functional and imperative programming. Imperative programming is much like a set of orders that are handled from top to bottom (with the occasional branch). Functional programming puts

much less emphasis on explicitly ordering code and more on evaluating individual functions. The compiler (or interpreter) has to figure out how to translate function evaluations into ordered machine code.

4.1.1 Ad-hoc Polymorphism Using Type Classes

Type classes are the Haskell solution to overloading functions. A function doing the same thing for different types, be it an equality test, summation, printing, or anything else, should have the same name. In Haskell, type classes capture functions that are polymorphic over the data types they are working on. This allows using the same function symbol (i.e. (==) for equality, (+) for addition) for `Int`s and `Double`s which are different types. Type class instances can be written for user-defined data types as well, sometimes even automatically derived for a core set of type classes.

The type class `Num`, for example, defines the `(+)` function to be of the type `(+)::a→a→a`, where mathematical notation would be $(+) :: \mathcal{A} \times \mathcal{A} \to \mathcal{A}$. The Haskell notation exposes two peculiarities: (i) addition is possible only for two values of the same type, as there is no implicit casting between types; (ii) Haskell functions can be partially applied: `(+1) :: Int→Int` with the `(+)` operator applied to one argument giving the "increment by one" function.

Generic programming (Lämmel and Peyton Jones, 2003, 2005; Chakravarty et al., 2009; Hinze et al., 2007) extends this concept further. A special type class is instanciated (automatically by the compiler!) for user-defined data types and functions can be written generically on top of the generic representation.

Modern Haskell allows associated data types for type classes. Consider the type class function for addition `(+)::a→a→a`. The type of the function restricts the input types to be of the same type as the resulting one. With an associated type, one could reasonably write `(+)::a→b→AddType a b` with the type `AddType` itself being a function of the input types `a` and `b`. One such instance could be `(+)::Int→Double→AddType Int Double` with `AddType Int Double =Double`. The actual (+) function than reads:

```
type instance AddType Int Double = Double
(+) :: Int → Double → AddType Int Double
(+) a b = plusDouble (fromIntegral a) b
```

The notation `type instance AddType a b = ...` declares what the resulting type of applying `AddType` to two types is. The implementation uses the machine instruction `plusDouble` to add two doubles. The `a::Int` is converted to a double using `fromIntegral`. Note that the user now only needs to write `1 +1.34`. The implementation of (+), the conversion, and the use of the machine instruction are hidden.

This arithmetic example, as well as several others, are discussed by Kiselyov et al. (2010) in an introduction to *type families*, basically functions on the type level.

Type classes with associated data types are used in a number of high-performance libraries (Leshchinskiy, 2009; Keller et al., 2010) including the `ADPfusion` library (Höner zu Siederdissen, 2012) for high-performance dynamic programming, mainly in computational biology, which forms part of this thesis (Chapter 8).

As an example for how type classes and type families simplify design of a high performance library consider a linear algebra package. In such a package, the efficient computation of matrix products is a standard feature. For best performance, the two operands need to have a different memory layout.

The actual matrix operations, like additions and products, are defined in a type class, just like addition for scalars was defined above. This gives the user a single interface independent of the internal implementation.

The choice of internal implementation can be guided by the types of the operations. Using type families it is possible for the compiler to calculate the best representation and encode this statically in the operand types. The product operations for matrices can then be designed so that one matrix is in column- the other in row- major layout, improving performance.

While this example simplifies reality somewhat, type-guided auto-adaptive representations are a powerful feature in the design of algorithms that relieve the user of such a library from the burden of having to understand the intricacies of the underlying numerics and hardware.

4.1.2 Monads

When filling memoization tables for dynamic programming, the order in which individual cells are to be filled must be given. There are two solutions, one may use laziness and let the program figure out the order by itself or give an explicit order in a monad. The former approach is more elegant, the latter a lot faster.

Monads (Moggi, 1989) in functional programming languages are a very general concept, and thus here only one specific aspect of their use is being considered. As stated above, lazy evaluation and high performance are generally at odds with each other. The known approaches to an implementation of lazy behaviour are particularly problematic for numeric code as used for linear algebra or dynamic programming where memoization tables have to be filled.

Here, two examples are given where an explicit ordering is important. The first is `I/O`. An interactive program will typically interleave user input with output. In this case it is important that each output is provided only after the corresponding input has been given by the user. Prior to the use of monads, `I/O` in Haskell was rather unsatisfying (Hudak et al., 2007, Sec. 7).

More important toward a correct implementation of dynamic programming calculations is the ability to order the way in which memoization tables are to be filled. Consider the memoization table example in Table 3.1. Upon requesting the result D stored at cell $(1, L)$, a lazy evaluation scheme has to recursively calculate all table cells. The `GHC` compiler knows how to do this automatically but this comes with a large runtime cost. This cost can be anything from $\times 1.25$ up to $\times 60$ or more (for ADP programs).

Instead of letting the compiler figure out how to lazily calculate the values for memoization tables, monads allow the user to specify the exact order in which tables are to be filled. The burden of filling the table is now on the user; he has to make sure that

A_1 and A_2 are actually calculated before B_1 and B_2 in Table 3.1. In return however, the performance can be close to what is possible in `C`.

The reader can rightfully ask why to use Haskell at all if the convenience of using high-level programming comes with the cost of performance or vice versa. This is where the monad concept comes in handy. As an abstraction mechanism, it allows the programmer to specify how tables should be filled. In the case of RNA secondary structure dynamic programming tables, this order is from the main diagonal towards the upper-right corner of the dynamic programming matrices.

Once specified, the correct order of filling is guaranteed by the abstraction mechanism (the monad) and this specification, made concrete in form of a function, can be re-used. The work of figuring out how to fill memoization tables thus needs only to be spent once for a whole class of algorithms that require a certain way in which to calculate results.

4.1.3 Algebraic Data Types

Algebraic data types compose (simpler) data types. They can both collect different data types thereby providing product types (a struct in the language of `C`), and offer choice within a finite set as a sum type (a union in `C`).

A very simple example of a sum type is the `Bool` data type with data constructors `True` and `False`: `data Bool =True | False`. Here, `data Bool` defines the boolean data type, and `True` or `False` are the two possible variants.

A more complex data type, `Step` (Listing 4.1), provides three data constructors similar to what would be offered by a `C` union construct. A `Step` is either `Done`, `Yield`'s something, or `Skip`s a step. The `Step` data type is parametrized over `a` and `s`. The parameters have no inherent meaning – the programmer chooses which type to give the parameters.

In the explanation of stream fusion below, `Step` encodes a single step in a *loop*. In this encoding, `s` encodes the current *index*, while `a` encodes the state of some computation given the current index. This can, for example, be the result of accessing an array given the current index.

Listing 4.1: The `Step` data type (Coutts et al., 2007)

```
data Step a s = Done | Yield a s | Skip s
```

Data constructors like `Done`, `Yield`, and `Skip` can be pattern-matched on to determine which of a set of constructors from a single data type is the argument. The variable `step` below is of type `Step`. A piece of code like the one below is the Haskell version of branching. Depending on the variant of `step`, the code will branch into one of the three cases `function_1` to `function_3`. As both `Skip` and `Yield` carry data around (`Skip` the next index, `Yield` the next index and some current value), these values are handed over to `function_2` and `function_3`.

```
case step of
  Done      → function_1
  Yield a s → function_2 a s
  Skip    s → function_3   s
```

4.1.4 Existential Quantification

In Haskell all functions and type constructors are typed, either implicitly letting the compiler figure out the types (for functions), or explicitly by the user. A number of extensions to the Haskell type system exist that allow expressing more complex types.

One of these abstractions is existential quantification which allows the programmer to implement the concept of an *abstract* data type. Abstract data types encapsulate a specific *implementation* and only expose an *interface*. One example of this is the `Stream` data type defined and used below that has a curious type:
`data Stream a = ∃ s . Stream (s →Step a s) s.`

Below, `Streams` are used to encode a concept of "functional loops", a `Stream` provides zero to many values of type `a`. If one wants to encode $(1 + \cdots + 10)$ in Haskell, one possibility is to create a stream that produces the numbers 1 to 10 (with `a` being `Int` in this case) and providing a function that can sum up all values in such a stream. This will, in fact, be shown below.

The implementation of a `Stream` that produces the values $1 \ldots 10$ involves the actual values which need to be accessible and are exposed as the type `a`. It also involves a "step" or "loop" construct which requires an index or seed `s`. Every time a new step is taken (`s → Step a s`) one may create a new value and also advance the index. The actual implementation of the index should be abstracted away as there is no need to expose it. By declaring ∃ `s`, the only thing stream functions, or the programmer, can do is use the current index `s` to calculate the next index `s`, nothing more.

Given that each stream-transforming function defined below may only manipulate the stream state using exactly this interface, it becomes impossible[1] for the programmer to accidentally manipulate the stream state in a way that creates bugs.

4.2 Deforestation and Fusion

In Sec. 4.1 some basics of the Haskell language were introduced. This section is devoted to *automatic* optimization techniques, with the emphasis being on automatic. These techniques form the core functionality that enables the `ADPfusion` library to generate optimized code from high-level code provided by the user. Both, `RNAwolf` and `MC-Fold-DP` use such optimization techniques in a more low-level form, and a desire to combine high-performance code with high-level programming led to `ADPfusion`.

In general, these techiques are deforestation (Wadler, 1990; Gill et al., 1993), stream fusion (Coutts et al., 2007), and acid rain (Hinze et al., 2011). As deforestation and acid rain allude to, the goal is to remove intermediate tree structures that are being created whenever functions are composed in Haskell.

More explicit, whenever functions operating on data structures are composed, each function creates a new modified copy of the structure being manipulated. Each of these modified copies requires time and memory space to be created, and normally, only the last copy is retained as the result.

[1] not quite, but it becomes very hard to shoot yourself in the foot

If the intermediate copies are not required, they shouldn't be created in the first place. And thanks to purity it can be guaranteed that no side effects can be unintentionally lost by removing intermediate structures (but cf. Bird (2010), Cha. 21 on *Hylomorphisms and nexuses* for variants where retaining intermediates is useful).

As said, the key approach to faster algorithms is the removal of intermediate, temporary data structures. Taking the introductory example by Wadler (1990), we want to create the list of numbers $1 \ldots n$, square each number, and sum them up. In Haskell notation:

Listing 4.2: Wadler'1990 list-based

```
sum (map square [1..n])
```

The notation `[1..n]` is syntactic sugar, producing a Haskell linked list of type `[Int]` with all integers ranging from `1` to `n`. The empty list would be written as `[]`. `map square` considers each element of the list and applies the function `square`. Finally, `sum` deconstructs the list and sums up all elements.

The problem, as pointed out by Wadler, is that in a strict language both `[1..n]` and `map square` create temporary lists each of size $O(n)$. In case of lazy evaluation, the space complexity will be $O(1)$, but the individual list cells themselves still need to be created and destroyed.

The key idea is that the program 4.2 can be transformed automatically and transparently (invisible) to the user into a version allocating *no* list cells *at all*. This automatically transformed version is shown in Listing 4.3 and requires just three registers instead of having to create $O(n)$ list cells.

The compiler-created version `sumMapSquare` makes use of a locally defined function `go`[2] which is defined using the `where` keyword. `go acc cur` uses an `acc`umular to to store the sum of squared values up to the `cur`rent value. If the current value exceeds the integer up to which to sum (`n`), the current `acc`umulator value is returned. Otherwise, `go` is called recursively. The `acc`umulator is increased by the `square` of the `cur`rent value and the `cur`rent value is increased by one.

Listing 4.3: Wadler'1990 transformed

```
sumMapSquare n = go 0 1 where
  go acc cur = if cur > n
                 then acc
                 else go (acc + square cur) (cur + 1)
```

Stream fusion on lists (Coutts et al., 2007) and arrays (Leshchinskiy, 2009) allows the programmer to express more programs using the high-level style of listing 4.2 that the compiler can *automatically* transform into tail-recursive calls of the type shown in listing 4.3 without user intervention apart from having to include the stream fusion library.

The algorithms presented in Chapter 7 make use of stream fusion on the level of lists and vectors. `ADPfusion` (Chapter 8) extends the fusion paradigm to *automatic optimization of complete dynamic programming algorithms*. Updates for the algorithm presented

[2] the use of `go` is Haskell culture

in Chapter 6 are planned.

We follow with two additional techniques that are made use of in ADPfusion: call-pattern specialization and stream fusion itself. Stream fusion is the technique explicitly used, but it in turn relies on call-pattern specialization.

Hence an introduction to the most important ideas in this topic is in order.

4.2.1 Call-pattern Specialization

Call-pattern specialization (Peyton Jones, 2007) or constructor specialization is an optimization that replaces function calls with constructor patterns with specialized functions. For each constructor (not their arguments) a specialized function is created. This optimization can reduce the need for temporarily created constructors, thereby improving the runtime performance of programs.

Consider the type constructor `Maybe a` which contains either `Just a` or `Nothing`. This data type captures the concept of a value of type `a` being either available or not. Similar to how in `C` a pointer can be a null pointer, pointing to nothing, or pointing to a value of a certain type.

Listing 4.4: Call-pattern specialization

```
data Maybe a = Just a |  Nothing

f :: Maybe Int → Int
f (Just x) = 10 + x
f Nothing  = 0

g = f (Just 10)
h = f Nothing
```

The calculations in `g` and `h` first create a `Maybe` value and then pass this value to the function `f`. `f` then inspects both the data constructors `Just` or `Nothing` and then performs a calculation.

The pattern matches performed by `f` are known during compile time, making it possible to create two versions of `f` without the programmer having to know this (he is writing the code above).

Listing 4.5: Call-pattern specialization

```
f_Just :: Int → Int, f_Nothing :: Int
f_Just x = 10 + x
f_Nothing = 0
g = f_Just 10
h = f_Nothing
```

Further optimization of `f_Just, f_Nothing` will inline both calls, making both `g` and `h` constant in this case. The *real* use of call-pattern specialization is in the optimization of stream fusion as well as the optimization of more generic functions in Haskell. Peyton Jones (2007) gives a number of examples, including some in the stream fusion library (Coutts et al., 2007).

For us, the importance of call-pattern specialization is that conditional branches can be encoded using different data constructors, which are individually transformed into recursive calls that are able to call each other. This pattern of conditional branches is captured by the `Step` data type defined in Listing 4.6

4.2.2 Stream Fusion

Stream fusion (Coutts et al., 2007) is a short-cut fusion system for lists aimed at eliminating intermediate data structures without user-intervention. While the ideas of stream fusion have been extended to arrays (with the `vector`[3] library) (Leshchinskiy, 2009), only fusion of lists will be discussed using the implementation by Coutts et al. (2007). For a general introduction, the original implementation is simpler, as it explicitly transforms list-based operations, while the `vector` package transforms operations on arrays, requiring additional thoughts on array size management and an underlying monadic interface.

Steps and Streams

The basic building blocks of stream fusion are the `Step`, and the `Stream` data types. The `Step` data type has already been used as an example in Sec. 4.1.3, while the `Stream` data type has been seen in Sec. 4.1.4. Both data types where introduced in the original stream fusion paper by Coutts et al. (2007).

To recapitulate, a `Stream` encodes a "functional loop" that will, in each step, produce a value of type `a`. This stream of values is what the user wants to "loop" or iterate over. In dynamic programming it will be data extracted from individual cells in a dynamic programming matrix.

In each step of the "functional loop", a `Step` data type is created. The three variants, `Done`, `Yield`, and `Skip` provide the different branching possibilities. With `Done` a loop is terminated, similar to a **`return`** statement[4] in `C`. When the current stream step produces a `Yield`, an element of type `a` is provided. Coming back to Wadler's summation example (Listing 4.2), a `Yield` produces the integers 1 to n. `Skip` provides the ability to filter out stream elements that are undesired. Both `Yield` and `Skip` carry along the next seed (or index) `s`, but where `Yield` provides an element `a`, `Skip` just continues to the next loop step. In terms of `C`, this is equivalent to `C`'s **`continue`**.

Listing 4.6: The `Step` and `Stream` data types (Coutts et al., 2007)

```
data Step a s = Done | Yield a s | Skip s
data Stream a = ∃ s. Stream (s → Step a s) s
```

Another way to look at `Step` and `Stream` is to consider the combination of `fold`s and `unfold`s. A `fold`, like `sum` ($\sum$), takes a list of elements and produces from this list a single result element. A `Stream` unfolds a single initial seed index `s` into a list – or stream – which can then be folded over.

[3] `http://hackage.haskell.org/package/vector`

[4] it is unfortunate that Haskell uses **`return`** as well, but in Haskell **`return`** has a *completely* different meaning

Conversion to and from Lists

Two simple stream functions capture this concept of `unfold` and `fold`. The stream function `stream` take a list `zs :: [a]` (`zs` of type list of `a`s) and provides a `Stream` that produces exactly the elements in the list. The unfolding function `next` takes the seed – in this case the list provided by the user – and examines it. If the list is empty (`next []`), the stream produces a `Done` step. If the list is non-empty (`next (x:xs)`), it is cut into a head `x` and a tail `xs`. The head is provided as the stream element to yield, while the tail becomes the next seed.

Note that neither `stream` nor `next` are actually recursive. The definition says how to produce one step given a seed. The only recursive functions are folding functions like `unstream`.

The `unstream` function folds a stream to produce a single result. This single result is the list of elements created using the different combined stream functions. `unstream` takes the `Stream` apart to extract the `next` function which can produce a single step in the functional loop using the `Step` data type. It also requires the initial seed `z`. Via the locally defined `go`, each current seed or index is pattern-matched on. In case the current seed produces `Done` the loop terminates with an empty list `[]` as the last element. `Skip`ping directly calls `go` recursively with the next seed `t`, while `Yield x t` produces an `x` in the head position and appends (using `(:)`) the tail of the list recursively via `go`.

It follows that given a finite list `xs` the equality `xs` ≡ `unstream (stream xs)` holds. From this equality one can conclude that functions operating on finite lists can be replaced by stream fusion functions that are more efficient than their list counterparts. As already stated, this can be done by the user switching from the standard list library to the stream fusion drop-in replacement.

Listing 4.7: `stream` and `unstream` (Coutts et al., 2007)

```
stream :: [a] → Stream a
stream zs = Stream next zs where
  next []     = Done
  next (x:xs) = Yield x xs

unstream :: Stream a → [a]
unstream (Stream next z) = go z where
  go s = case s of
    Done      → []
    Skip t    → go t
    Yield x t → x : go t
```

In order to replicate Wadler's example (Listing 4.2), three stream fusion functions are required. A sum over integers, a function to map the `square` function, and a way to create consecutive integers from m to n. Providing all three functions makes it possible to not use `stream`/`unstream` at all. For Wadler's example this is not required as the compiler can already produce the efficient version of Listing 4.3. The `ADPfusion` library requires a more complex fusion system not possible with plain linked lists.

Mapping a Function over a Stream

The first stream fusion function, `map` takes a function (`f::a→b`) from type `a` to type `b` and a `Stream` of `as` and produces a `Stream` of `bs`. The original `map` function was only defined for lists. The version presented here is the stream equivalent. By keeping the names the same, existing code needs only import the stream fusion library, instead of the library for plain linked lists, to enable fusion.

Note that `map` is not recursive. It inspects only one `Step` using the locally created `next` function. If a `Yield` is produced, it changes `x::a` to `(f x)::b`, but otherwise does not change the stream. `Steps` can be created from the seeds `s` only using `sfun s`. This is the only way in which the seed can be inspected.

Listing 4.8: `map` (Coutts et al., 2007)

```
map :: (a→b) → Stream a → Stream b
map f (Stream sfun z) = Stream next z where
  next s = case sfun s of
    Done       → Done
    Skip t     → Skip t
    Yield x t  → Yield (f x) t
```

Folding a Stream into a Single Value: Summation

What remains is to sum up all the arguments. `sum`, like `unstream`, is recursive and consumes the whole stream. Note that using example Listing 4.2, the `next` function in `sum` is actually combined of a number of functions applied one after another, as exercised below in the worked stream fusion example in Sec. 4.2.4.

The inner workings of `sum` are almost the same as those of `unstream`. In this case, a stream of things that can be summed up is reduced to a single number. The recursive `go` function carries an `accumulator` that is initialized with 0, and the initial stream seed `z`. Depending on the step produced by the current seed in `go`, one of three branches is taken. In case of `Done` the final `accumulator` value is returned. If the current step is to be `Skip`ped, `go` is called recursively with the current accumulator and the next seed `t`. Only in case of a `Yield` is the yield value `x` added to the accumulator. `go` is then called recursively with the new accumulator value and the next seed.

Listing 4.9: `sum`

```
sum :: Stream a → a
sum (Stream next z) = go 0 z where
  go acc s = case next s of
    Done       → acc
    Skip t     → go acc t
    Yield x t  → go (acc+x) t
```

Creating a Stream of Integers

Finally, in order to avoid list comprehensions (terms of the form `[1..n]`), a generator function is required. This function creates a stream equivalent to `stream [1..n]`. As `ADPfusion` (Chapter 8, Höner zu Siederdissen (2012)) does not use `stream`/`unstream` but creates streams directly, a discussion of the syntactic sugar provided by list comprehensions can be avoided.

The `enumFromTo` generator of Listing 4.10 creates a stream that yields each `Int` from `from` to `to` and then terminates the stream. The initial seed is the lower bound `from` of the increasing list of integers. The local function `next` takes the current seed and compares it to the upper bound. While the current value is still within the bounds ($from \leq cur \leq to$) the current value is wrapped in a `Yield` and the seed is set to $cur + 1$. If the condition does not hold, a `Done` step is generated.

Listing 4.10: `enumFromTo` generator function

```
enumFromTo :: Int → Int → Stream Int
enumFromTo from to = Stream next from where
  next cur = case (cur ≤ to) of
    True  → Yield cur (cur+1)
    False → Done
```

4.2.3 The `case-of-case` Transformation

The optimizations below makes extensive use of the `case-of-case` transform (Peyton Jones and Santos, 1998). This transform simplifies nested branches. Branches of this type occur very often during the simplifications performed in stream fusion.

Consider the nested `case` statement below, where the outer branch variable is calculated via the inner `case`.

```
case
  case x of
            A → X
            B → Y
of
   X → x
   Y → y
```

This code can be transformed into the equivalent but simpler

```
case x of
          A → x
          B → y
```

4.2.4 A Worked Stream Fusion Example

In this worked example, the steps taken by the compiler to transform high-level code into efficient code are shown. Starting from `map square (enumFromTo m n)` written by the user, the compiler is able to calculate the same efficient code as shown in Listing 4.3.

In Listing 4.11, the first compilation step in transforming `sum (map square (enumFromTo m n))` is shown (where $m == 1$ would give the original sum over `[1..n]`, each squared) The compiler inlines (copies) all function bodies from `enumFromTo`, `map`, and `sum`. The composed functions are combined into one larger block. The Haskell-style comment "`-- comment`" denotes the origin of each local function.

Some simplification is achieved by ignoring the `Skip` constructor. As the example does not make use of `Skip`, nothing is lost and the example code is a bit more clear. Furthermore, the same optimizations as for `Done` and `Yield` would be performed.

Listing 4.11: optimization of `sum (map square (enumFromTo m n))`

```
sumMapSqure m n = go 0 m where
    next_enum from = case (from ≤ n) of  -- enumFromTo "next"
      True  → Yield from (from+1)
      False → Done
    next_map s = case next_enum s of      -- map "next"
      Done       → Done
      Yield x s’ → Yield (square x) s’
    go acc s = case next_map s of         -- sum "go"
      Done       → acc
      Yield x s’ → go (acc+x) s’
```

In the first transformation step, the compiler inlines `next_enum` into `next_map`. The following `case-of-case` construct can then be further optimized. Notice that all constructors (`True`, `False`, `Done`, and `Yield`) are statically known.

```
sumMapSqure m n = go 0 m where
    next_map s =
      case
        case (s ≤ n) of
          True  → Yield s (s+1)
          False → Done
      of
        Done       → Done
        Yield x s’ → Yield (square x) s’
    go acc s = case next_map s of
      Done       → acc
      Yield x s’ → go (acc+x) s’
```

A `case-of-case` optimization is performed.

```
sumMapSqure m n = go 0 m where
    next_map s = case (s ≤ n) of
      True  → Yield (square s) (s+1)
      False → Done
    go acc s = case next_map s of
      Done       → acc
      Yield x s’ → go (acc+x) s’
```

The resulting code for `next_map` is inlined, and again a `case-of-case` transformation is performed.

```
sumMapSqure m n = go 0 m where
    go acc s =
      case
        case (s ≤ n) of
          True  → Yield (square s) (s+1)
          False → Done
      of
        Done       → acc
        Yield x s' → go (acc+x) s'
```

The final code is optimal. `go` is recursive, accumulating in `acc` the sum of values from `m` to `n`. The final `case` expression will be transformed into efficient assembler or LLVM (low-level virtual machine (Terei and Chakravarty, 2010; Lattner and Adve, 2004)) code.

```
sumMapSqure m n = go 0 m where
    go acc s = case (s ≤ n) of
      True  → go (acc + square s) (s+1)
      False → acc
```

At this point, it should be clear that even very complex list-like code can be transformed into efficient final code by the compiler applying a set of transformations often enough. For more complex user code, like `ADPfusion` dynamic programs, it would become infeasible for the programmer to apply these transformations manually. The compiler, however, only needs to iterate the set of rules given here to finally produce efficient code.

The combinators developed in the `ADPfusion` library are all written in the style used in `map` (Listing 4.8). `RNAwolf` as of this work is based on the `vector` library. This allows for partial fusion. The success with `Nussinov78` and `RNAfold` using `ADPfusion` means that complex algorithms can be written using `ADPfusion`. Ongoing work using these ideas is presented in the concluding Chapter 9.

Chapter 5

Semantics and Ambiguity of Stochastic RNA Family Models

Robert Giegerich and Christian Höner zu Siederdissen.
Semantics and Ambiguity of Stochastic RNA Family Models.
IEEE/ACM Transactions on Computational Biology and Bioinformatics. 2011. 8 (2). 499–516.

RG designed the study. CHzS wrote the CMC covariance model compiler. Both authors participated in writing the paper.

© 2011 IEEE. Reprinted, with permission, from Giegerich and Höner zu Siederdissen, Semantics and Ambiguity of Stochastic RNA Family Models, 2011 (published online 3. Feb. 2010)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

Semantics and Ambiguity of Stochastic RNA Family Models

Robert Giegerich and Christian Höner zu Siederdissen

Abstract

Stochastic models such as hidden Markov models or stochastic context free grammars can fail to return the correct, maximum likelihood solution in the case of semantic ambiguity. This problem arises when the algorithm implementing the model inspects the same solution in different guises. It is a difficult problem in the sense that proving semantic non-ambiguity has been shown to be algorithmically undecidable, while compensating for it (by coalescing scores of equivalent solutions) has been shown to be NP-hard. For stochastic context free grammars modeling RNA secondary structure, it has been shown that the distortion of results can be quite severe. Much less is known about the case when stochastic context free grammars model the matching of a query sequence to an implicit consensus structure for an RNA family.

We find that three different, meaningful semantics can be associated with the matching of a query against the model – a structural, an alignment, and a trace semantics. Rfam models correctly implement the alignment semantics, and are ambiguous with respect to the other two semantics, which are more abstract. We show how provably correct models can be generated for the trace semantics. For approaches where such a proof is not possible, we present an automated pipeline to check *post factum* for ambiguity of the generated models.

We propose that both the structure and the trace semantics are worth-while concepts for further study, possibly better suited to capture remotely related family members.

Index Terms

RNA secondary structure, RNA family models, covariance models, semantic ambiguity.

I. Introduction

A. Background: Semantics and ambiguity in stochastic modeling

Stochastic models: Stochastic models are powerful and widely used techniques in computational biology. In this article, we study covariance models implemented by stochastic context free grammars (SCFGs), which include hidden Markov models (HMMs) as a subclass. Let us start our discussion with this simpler model.

An important application of HMMs in biosequence analysis is the modeling of protein families. There, aligned protein sequences are processed into family models implemented as HMMs using the HMMer package [7] and stored in the Pfam data base [2]. Running a query sequence against a model returns a score that indicates the likelihood that the query belongs to the sequence family. Scanning a long sequence with the model reveals those regions that most likely share an evolutionary relationship with the model family.

An application of similar importance is the modeling of structural RNA families. Models are generated with the tool *Infernal* [8], [15] and accessed via the Rfam data base [9]. Here, an SCFG implements a covariance model of RNA sequences that share a consensus secondary structure. A "parse" of a query sequence with the model grammar shows how well it matches the family sequences, accounting for sequence as well as structure conservation.

HMMs and SCFGs use quite a different nomenclature. Nevertheless, mathematically, HMMs are a subclass of SCFGs – those cases where the context-free grammar underlying the SCFG belongs to the

Robert Giegerich is with the Center of Biotechnology and the Faculty of Technology at Bielefeld University, D-33615 Bielefeld, Germany; robert@techfak.uni-bielefeld.de
Christian Höner zu Siederdissen is with the Institute for Theoretical Chemistry, University of Vienna, Währingerstraße 17, 1090 Vienna, Austria; choener@tbi.univie.ac.at 0000

Digital Object Indentifier 10.1109/TCBB.2010.12 1545-5963/10/$26.00 © 2010 IEEE

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

subclass of regular grammars. Where the SCFG literature [8], [18] uses the terminology of formal language theory, such as grammars and parses, the HMM literature prefers a terminology of transition rules and paths. The CYK algorithm, which returns the highest scoring parse according to a SCFG, is a generalization of the Viterbi algorithm, which returns the highest scoring transition path for an HMM. In this article, we will build on the established SCFG terminology, because it makes the theory more general and also because our immediate practical interest lies with covariance models as used in Rfam.

Modeling semantic ambiguity: The problem of semantic ambiguity has been recently addressed in a series of papers. Giegerich [10] pointed out the problem and suggested a suitable formalization: the parse trees constructed by a SCFG parser represent some real-world objects of interest, for example alternative secondary structures for an RNA sequence. If some of these parses actually represent the same object, we have a case of semantic ambiguity. By specifying an explicit mapping of parses to a canonical (unique) representation of our objects of interest, it may be possible to prove presence or absence of ambiguity. The use of a canonical representation appears to be a necessary extension to the standard framework of stochastic modeling, in order to deal with ambiguity in a systematic manner. It plays the role of associating a precise semantics to the parse trees (namely, the structures they represent), and coding this meaning *within* the model is the key to tackling it computationally. The term "semantic" ambiguity that we use in this article catches this fact, and discerns it from syntactic ambiguity as studied in formal language theory. In our case, syntactic ambiguity only means that a grammar can specify several *different* structures for a given sequence, which is a good thing rather than a problem in combinatorial optimization. Note that in textbooks covering SCFGs [1], [6], the pitfall of semantic ambiguity has not yet been paid attention to, and the most likely parse is taken for granted to indicate the most likely structure.

Ambiguity – does it really matter: Dowell and Eddy [5] approached the ambiguity issue from the pragmatic side and investigated whether it really matters in practice. They compiled a number of plausibility arguments, why one might hope that the most likely parse somehow points to the most likely structure, even in the presence of ambiguity. But then, they refuted such hopes: For two ambiguous grammars, they tested how often the most likely parse returned by the SCFG was different from the most likely structure. For one grammar (G1), the result was wrong for 20% of all tested sequences. For the other grammar (G2), which was a refinement of G1 for the sake of better parameter training, the result was wrong even for 98%. Dowell and Eddy provided a first empirical test for the presence of ambiguity, and continued studying parameter estimation for several alternative, non-ambiguous grammars.

Algorithmic undecidability of semantic ambiguity: The idea of ambiguity checking was further worked out by Reeder et al. [16]. They gave a proof that, in general, presence or absence of semantic ambiguity is formally undecidable. However, they contributed a series of further techniques for ambiguity checking, where the most powerful one involves translation of the SCFG into a context-free grammar generating the canonical representation introduced in [10]. Then, a semi-decision procedure such as a parser generator may be able to demonstrate presence or prove absence of ambiguity in many relevant cases. The simple unambiguous grammars studied in [5] were proved unambiguous in this mechanized fashion. Moreover, the rather sophisticated grammar designed by Voss et al. for probabilistic shape analysis [21] could also be proved non-ambiguous in a similar way. The study by Reeder et al. [16] also indicated some techniques of avoiding ambiguity. However, there are cases where the expressiveness of the model – the capability of adapting the parameters of the model to a training set – may suggest to prefer a semantically ambiguous grammar.

Algorithmic infeasibility of ambiguity compensation: Can we still obtain the desired result when the grammar is ambiguous? Such a case was studied in the HMM literature by Brejova et al. [4] under the name "path labeling problem". In HMM modeling, the model itself often is more refined than the final result. For example, the gene structure of a sequence can be indicated by a labeling of residues by E (exon) or I (intron) states. Yeast, for example, has two classes of introns, "short" and "long". The stochastic model, in order to capture the length distribution of introns, requires several states to model intronic residues. Therefore, several transition paths through the model may differ in their points of transition between intronic states, while they lead to the same path labeling and hence, indicate the same

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

gene structure. Here, the path labeling constitutes the canonical mapping: Paths are equivalent when they have the same labeling, and the HMM is semantically ambiguous when this happens. Brejova et al. then studied what we call ambiguity compensation: can the algorithm be modified such that all scores of paths with the same labeling accumulate? Their main result was that, in general, this problem is NP-hard, and hence, computationally infeasible. This does not rule out that ambiguity compensation may be practical in restricted cases, but in general, we are better advised to avoid semantic ambiguity altogether.

Unresolved questions: There are three questions that were not addressed: (1) Dowell and Eddy studied semantic ambiguity in principle, but worked with rather small example grammars. Grammars such as those underlying Rfam are much larger, and they do not simply assign a structure to an RNA sequence, but they also relate it to the family model. It is unclear how the semantics of a model should be defined. (2) While methods for ambiguity checking in formal language theory have been advanced recently [3], the step from a large, tool-generated SCFG to the context-free grammar suitable for ambiguity checking is still left open. (3) Are the models used in practice actually semantically unambiguous, and if so, based on which semantics? These are the questions we will address.

B. Contributions of this article

This article provides a theoretical and a software-technical contribution, and their application to Rfam models.

On the theory side, we formally define three alternative semantics for covariance models for RNA families – a *structure*, a *trace*, and an *alignment semantics*. All three of them have a well-defined biological meaning, which is interesting to implement. Whether or not a particular grammar is in fact semantically ambiguous depends, of course, on the chosen semantics. We show how provably non-ambiguous models with respect to the trace semantics can be constructed.

On the technical side, we provide an automated pipeline that accepts a grammar G, a canonical representation mapping (written in a particular style), and produces a grammar $\widehat{G}$ which is *syntactically* ambiguous if and only if G is *semantically* ambiguous. Connecting this pipeline to a (syntactic) ambiguity checker for context-free grammars, this automates *semantic* ambiguity checking as far as its intrinsic undecidability allows for it.

In the application, we apply our formalism to Rfam models. We find that Rfam models faithfully implement the alignment semantics, although their description in the literature at one point suggests a structure semantics. With respect to the structure and the trace semantics, they are ambiguous. In the conclusion, we argue that both the structure and the trace semantics are worth further study, because they are more abstract and may be better suited to capture remotely related family members.

The article is organized as follows: In Section II we review what is known about semantics and ambiguity of simple SCFGs as used for structure prediction, about ambiguity checking, and ambiguity compensation. In Section III we turn to family model grammars and find that there are three alternative ways to define their semantics. In Section IV we describe precisely a new algorithm of model generation for the trace semantics and prove its correctness (i.e. non-ambiguity of the generated models). In Section V we describe a software for upward compilation and ambiguity checking of Rfam models. This pipeline is applied in Section VI. We conclude with a discussion of open research questions which arise from our findings.

II. A SUMMARY OF SEMANTIC AMBIGUITY THEORY

In this section, we review known results on the problem of semantic ambiguity. The only new contribution in this section is that the method for ambiguity checking suggested in [16] has now been automated. Along with this review, we introduce the concepts and the formalism to be further developed subsequently.

A. SCFGs and their semantic ambiguity

Context-free grammars: Given an alphabet $\mathcal{A}$ of symbols, $\mathcal{A}^*$ denotes the set of all strings of symbols from $\mathcal{A}$, including the empty string ϵ. A *context-free grammar* G is a formal system that generates a

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

$$G1: \qquad S \to \varepsilon \mid aS \mid Sa \mid aSb \mid SS \qquad\qquad G5: \qquad S \to \varepsilon \mid aS \mid aSbS$$

Fig. 1. Grammars G1 and G5 taken from [5]. S is the axiom and only nonterminal symbol in either grammar. a and b denote arbitrary bases out of $\{a, c, g, u\}$, as SCFGs allow non-standard base pairs (albeit with low probability). Hence, a rule like $S \to aSb$ is a shorthand for 16 different rules.

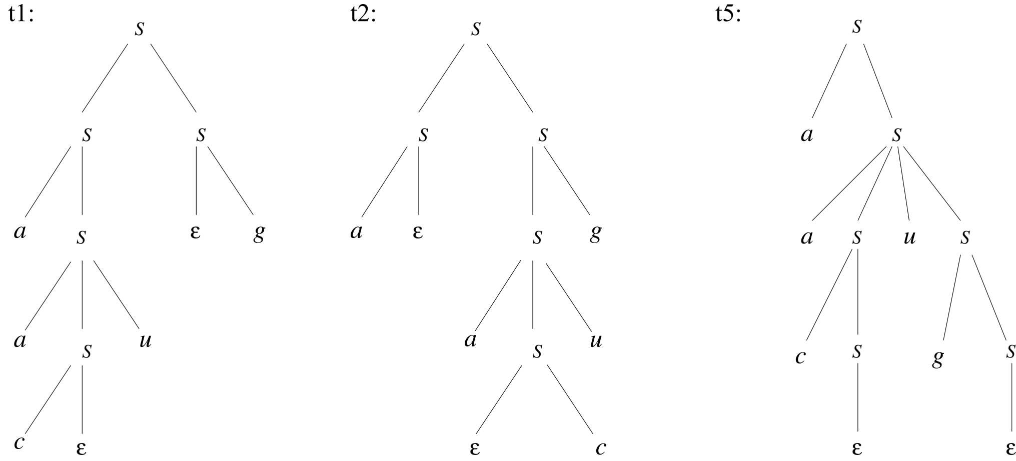

Fig. 2. Three derivation trees for the sequence `aacug`. $t1$ and $t2$ are derived with $G1$, $t5$ is derived with $G5$.

language of strings over $\mathcal{A}$. It uses a set V of *nonterminal symbols*, one of which is designated as the *axiom*. Its *derivation rules (productions)* have the form $X \to \alpha$, where $X \in V$ and $\alpha \in (V \cup \mathcal{A})^*$. A derivation of a terminal string $w \in \mathcal{A}^*$ starts from the axiom symbol, and in each step, replaces one of the nonterminal symbols in the emerging string according to one of the productions: $xXy \to x\alpha y$ may be a chosen transition when $X \to \alpha$ is a production of G. Such a derivation can be represented uniquely in the form of a tree, and by reversing the angle of view (from generating a string from the axiom to reducing a given string towards the axiom), this tree is also called a parse tree. Two grammars are shown in Fig. 1 , and three such parse trees are shown in Fig. 2. A grammar is *(syntactically) ambiguous* if there is a string that has at least two different parse trees. It is a classical result of formal language theory [12] that syntactic ambiguity of context-free grammars is formally undecidable. This means, there is no algorithm that can decide presence or absence of ambiguity *for all* context-free grammars. However, there are semi-decision procedures that return either YES, NO or MAYBE, which have proved quite powerful in practice [3].

Stochastic CFGs: A *stochastic* context-free grammar augments each production rule with a transition probability, such that the probabilities assigned with alternative rules for the same nonterminal symbol sum up to 1. For rules which simply generate a terminal symbol, the associated probability is called emission probability. We do not distinguish these two types of probabilities here. In a derivation, the probabilities of all applied rules multiply. In such a way, a parse tree t of string x assigns a probability $P(t, x)$ with x. The CYK algorithm, given x, computes the parse $t_{opt}(x) = argmax_t\{P(t, x) \mid t \text{ parse for } x\}$.

SCFG semantics: When modeling RNA structure, the *semantics* $\mathcal{S}_{\mathcal{SCFG}}$ of an SCFG G is defined as follows: Each parse tree t according to G associates an RNA secondary structure $\mathcal{S}_{\mathcal{SCFG}}(t)$ with sequence x: terminal symbols (denoting RNA bases) produced in the same step with productions like $S \to aSb$ are considered base paired, while all other ones are considered unpaired. Denoting structures in the familiar dot-bracket notation, where a dot denotes an unpaired base, and matching brackets denote paired bases, we observe $\mathcal{S}_{\mathcal{SCFG}}(t1) = \mathcal{S}_{\mathcal{SCFG}}(t2) = \mathcal{S}_{\mathcal{SCFG}}(t5) =$ `".(.)."`.

When there exist $t \neq t'$ but $\mathcal{S}_{\mathcal{SCFG}}(t) = \mathcal{S}_{\mathcal{SCFG}}(t')$ for grammar G, we say that G is *semantically ambiguous*. This occurs with the trees $t1$ and $t2$ for grammar $G1$ in Fig.2. There are no such trees with $G5$. Hence, $G1$ is semantically ambiguous, while $G5$ is an example of a non-ambiguous grammar.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

With a semantically unambiguous grammar, the most likely parse also means the most likely structure for x – this is exactly what we hope to find. If the grammar is semantically ambiguous, the most likely structure s_{opt} may have several parses such that $s_{opt} = \mathcal{S}_{\mathcal{SCFG}}(t_1) = \mathcal{S}_{\mathcal{SCFG}}(t_2) = \ldots$, with probabilities $p(t_1, x), p(t_2, x), \ldots$, and $P(s_{opt}) = \sum_i p(t_i, x)$. In this situation, it is not guaranteed that one of the parses t_i has maximal probability, and some unrelated parse (indicating a different structure), will be returned by the CYK algorithm. For the grammars $G1$ and $G2$ studied in [5][1] , this happens in 20% resp. 98% of all test cases.

Many simple grammars can be specified for RNA structure that are not semantically ambiguous. Different (non-ambiguous) grammars for the same problem have different characteristics with respect to the probability distributions they define. For example, grammar $G5$, attributed to Ivo Hofacker in [5], is arguably the smallest grammar for the purpose. It has only 21 parameters and showed "abysmal" modeling performance in [5].

B. Embedding SCFGs in a more general framework

In order to deal with ambiguity checking and compensation, both in theory and practice, we embed SCFGs in the more general framework of algebraic dynamic programming (ADP) [11]. This will allow us to replace the probabilistic scoring scheme "hardwired" in the SCFG concept by other evaluation schemes, or use several such schemes in combination. In our application, we will in fact generate equivalent ADP code from Rfam models, to be used for a variety of different purposes aside from stochastic scoring.

Algebraic dynamic programming: ADP is a declarative method to design and implement dynamic programming algorithms over sequence data. ADP and stochastic modeling tools serve complementary purposes (while both rely on the same type of dynamic programming algorithms for their implementation). ADP is designed to give the author of a DP algorithm maximal convenience – high level of abstraction, re-usable components, and compilation into efficient target code. Any type of combinatorial optimization over sequences is possible, provided that Bellman's Principle of Optimality holds. Grammars in ADP are produced by a human designer and are typically small – at least compared to grammars derived from data by stochastic modeling tools. These, in turn, come with a hard-wired scoring scheme for maximizing probability or log-odds scores, and the capability to train the parameters via expectation maximization. Many of the grammars constructed by automatic modeling tools such as *Infernal* have probably never been inspected by a human eye.

The ADP formalism starts from a *signature*, which is a supply of function symbols[2]. One of these, named h by convention, designates the objective function, to be used in subsequent analyses. The other ones are placeholders for scoring functions.

For example, these are the signatures we will use with $G1$ and $G5$:

$G1$		G5	
$openl : \mathcal{A} \times V \to V$	$openr : V \times \mathcal{A} \to V$	$open : \mathcal{A} \times V \to V$	
$pair : \mathcal{A} \times V \times \mathcal{A} \to V$	$split : V \times V \to V$	$pair : \mathcal{A} \times V \times \mathcal{A} \times V \to V$	
$nil : V$	$h : [V] \to [V]$	$nil : V$	$h : [V] \to [V]$

Here, $\mathcal{A}$ denotes the underlying sequence alphabet, V an arbitrary value domain, and $[V]$ a list of values.

Grammars in ADP are tree grammars. A *tree grammar* is analogous to a context free grammar, except that the righthand side in $X \to \alpha$ now is a tree, built from the function symbols of the signature (other than h) at inner nodes, and nonterminal symbols as well as terminal symbols residing at the leaves of the

[1] Dowell and Eddy use the term "structural ambiguity" rather than "semantic ambiguity". This is consistent with our terminology, because for simple SCFGs, a structural semantics is the only one that has been considered so far. When we will turn to family models, there will be different semantics which can be employed. Again, there will be a structural semantics, but it is not the one implemented in today's modeling approaches.

[2] Java programmers may think of it as an interface

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

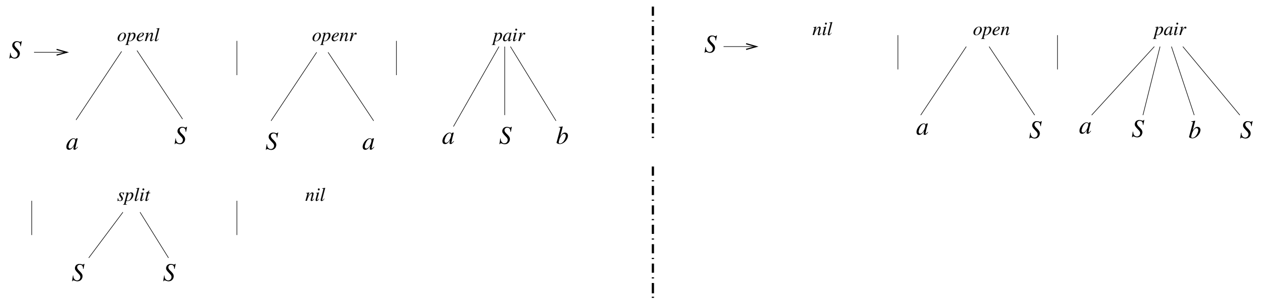

Fig. 3. Tree grammar versions of string grammars G1 (left) and G5 (right).

tree. Occasionally, we have a nullary function symbol, which also marks a leaf. Figure 3 shows the tree grammar versions of G1 and G5.

The derivation with a tree grammar works as with CFGs, except that now it produces a tree. It may derive the same tree in different ways (syntactic ambiguity of tree grammars), but this is easily avoided, and besides, syntactic ambiguity is decidable for this class of *tree* grammars. Therefore, we can assume that each tree has a unique derivation (or tree-parsetree). Each derived tree contains, as the string of its leaf symbols, some sequence $w \in A^*$. These trees represent the candidates in the combinatorial search space associated with sequence w, and in order to avoid the use of "tree" in too many connotations, we will henceforth refer to them as *candidates*.

The introduction of a tree grammar, based on a signature of functions, seems like a minor, artificial change of formalism, but has a profound impact: it decouples the candidates which we analyze from the grammar which generates them. There can be more functions in the signature than there are productions in the grammar, but normally, there are less. Different grammars over the same signature can be used to derive the same set of candidates. Candidates only reflect their signature – they bear no resemblance of the derivation and the grammar which generated them. Our candidates $t1, t2$ and $t5$ as derived by the tree grammars are shown in Figure 4.

The function symbols that constitute the inner nodes of the candidate can be used to associate a variety of meanings with each candidate. This is done by specifying an *evaluation algebra* – i.e. a data domain and a set of functions (which compute on this domain), one for each function symbol in the signature[3], including h. Whatever evaluation we define will be computed by a generic CYK-like algorithm. We do not worry about implementation issues here, and denote the analysis of input sequence x with grammar G and evaluation algebra B as a function call $G(B, x)$.

SCFGs encoded in ADP: To run an ADP grammar as a SCFG, one simply provides an evaluation algebra which implements the function symbols in the signature by functions that compute probabilities.

Evaluation algebra PROB for G1:

$$\begin{array}{lcllcl} h & = & \text{maximum} & & & \\ pair(a,x,b) & = & p_{ab} * x & split(x,y) & = & p_{split} * x * y \\ openl(a,x) & = & p_a * x & nil() & = & p_{nil} \\ openr(x,a) & = & p_a * x & & & \end{array}$$

The probability scores $p_a, p_{ab}, p_{split}, p_{nil}$ are to be estimated from the data.

Evaluation algebra PROB for G5:

$$\begin{array}{lcl} h & = & \text{maximum} \\ pair(a,x,b,y) & = & p_{ab} * x * y \\ open(a,x) & = & p_a * x \\ nil() & = & p_{nil} \end{array}$$

[3]Java programmers may think of implementing the "interface", but – please – with pure mathematical functions without side effects.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

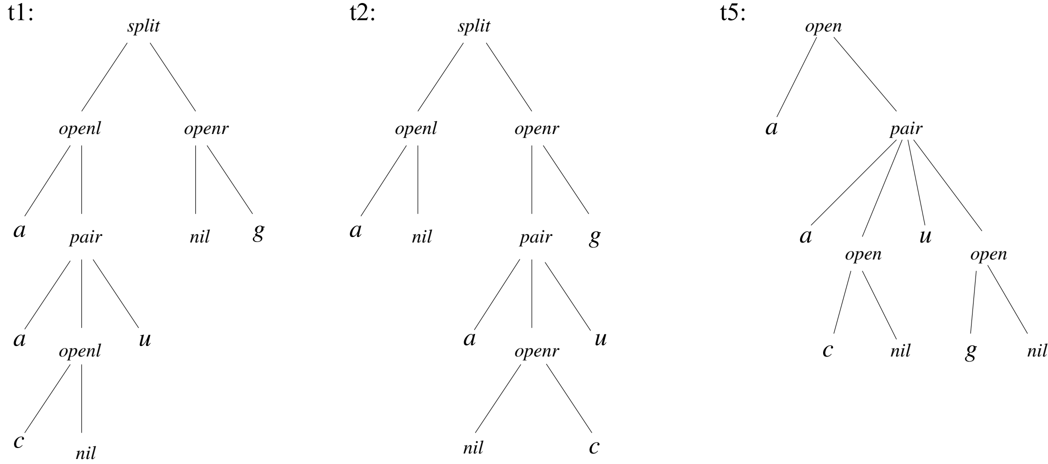

Fig. 4. Candidates $t1, t2$ and $t5$, as derived by their tree grammars $G1$ and $G5$.

Evaluating a candidate in this interpretation yields its probability score, akin to what is achieved by a SCFG, if the candidate was a parse tree. This is how we express the mathematical equivalent of an SCFG in ADP. The advantage is: once we have the grammar in ADP form, we can use it for other purposes besides stochastic scoring.

Encoding the canonical mapping: We use a second evaluation algebra to encode the canonical mapping of candidates to their "meanings". Let us call it CAN.

Evaluation algebra CAN for G1:

$$\begin{array}{lcllcl} h & = & id & & & \\ pair(a,x,b) & = & \texttt{"("} + x + \texttt{")"} & split(x,y) & = & x+y \\ openl(a,x) & = & \texttt{"."} + x & nil() & = & \texttt{""} \\ openr(x,a) & = & x + \texttt{"."} & & & \end{array}$$

In algebra CAN, we define the functions such that they compute, from the candidate, the dot-bracket representation of its associated structure. In other words, CAN implements the semantics $\mathcal{S}_{SC\mathcal{FG}}$ for G1. Operator + here denotes string concatenation, and id denotes the identity function.

Evaluating the G1-candidates $t1$ and $t2$ in the algebras PROB and CAN, we obtain

$$\begin{array}{lll} \text{PROB(t1)} & = split(openl(a, pair(a, openl(c, nil), u), openr(nil, g)) & = p_{split} \cdot p_{nil} \cdot p_{au} \cdot p_a \cdot p_c \cdot p_g \\ \text{PROB(t2)} & = split(openl(a, nil), openr(pair(a, openr(nil, c), u), g)) & = p_{split} \cdot p_{nil} \cdot p_{au} \cdot p_a \cdot p_c \cdot p_g \\ \text{CAN(t1)} & = split(openl(a, pair(a, openl(c, nil), u), openr(nil, g)) & = \texttt{".(.)."} \\ \text{CAN(t2)} & = split(openl(a, nil), openr(pair(a, openr(nil, c), u), g)) & = \texttt{".(.)."} \end{array}$$

In this way, the structure – the meaning of our candidates, which decides about ambiguity – now becomes part of our operational machinery. We can call $G1(CAN, \texttt{"aacug"})$, and multiple occurences of `".(.)."` in the output witness the semantic ambiguity of $G1$. We leave it to the reader to define analogous algebras PROB and CAN for the signature of G5. A powerful feature of the ADP approach is the use of algebra products (see [20] for the precise definition). For example, calling $G5(PROB{*}CAN, x)$ will give us *all* the structures for x that achieve the maximum probability score. Since the grammar G5 is semantically non-ambiguous, there may still be several candidates achieving maximal probability, but they must all produce different structures as indicated by CAN. When the grammar is ambiguous (like G1), neither of the optimal candidates may indicate the most likely structure, as explained in Section II-A. $G1(PROB * CAN, x)$ returns the optimal candidates together with their associated structures, possibly delivering duplicates, but we cannot be sure if any of them denotes the most likely structure.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

$$\widehat{G1}: \qquad S \to (S) \,|\, .S \,|\, S. \,|\, SS \,|\, \varepsilon$$

Fig. 5. Grammar $\widehat{G1}$ derived from G1

C. Automated checking of semantic ambiguity

We now introduce a systematic and automated approach to ambiguity checking. Consider a tree grammar and a canonical mapping algebra which maps candidates to strings over some alphabet $\widehat{A}$. In this setting, one can substitute the string composing functions of the algebra into the righthand sides of the tree productions. By partial evaluation, we eliminate the trees, and righthand sides become strings over $V \cup \widehat{A}$. Starting from the tree grammar G1, we rewrite its rules into those of grammar $\widehat{G1}$, shown in Fig. 5.

Note that the first rule in $\widehat{G1}$ is derived from 16 productions in G1, but since these are mutually exclusive due to their terminal symbols, only one corresponding rule is retained in $\widehat{G1}$.

In this way, from our tree grammar G we obtain a context-free (string) grammar $\widehat{G}$ with the following property:

Theorem 1 The tree grammar G is semantically ambiguous if and only if the string grammar $\widehat{G}$ is syntactically ambiguous.

The proof of this theorem was given in [16]. At that time, the grammar $\widehat{G}$ was handwritten – the new aspect here is that it is now produced automatically from G and the canonical mapping algebra. This is further described in Section V, where we present the pipeline *cm2adp* for upward compilation of *Infernal*-generated models into the ADP framework. Taking these constituents together –

1) the automated re-coding of an SCFG in ADP as a tree grammar G,
2) the specification of a unique string representation as canonical mapping algebra CAN,
3) the automated derivation of a string grammar $\widehat{G}$ from G and CAN,

we are now in a state where we can take a SCFG and submit it to an automatic ambiguity checker.

The only step which is not automated is, of course, the specification of the canonical mapping CAN. Naturally, we must say at one point what the meaning of our candidates really is. However, for grammars coming from the same modeling domain, this must be done only once, as the canonical mapping is the same for all grammars. In this sense, the ambiguity checking pipeline is completely automated now.

D. Ambiguity compensation

The canonical mapping defines (as its reverse image) a semantic equivalence relation on the evaluated candidates. Ambiguity compensation means that all scores within the same equivalence class should be accumulated, rather than maximized over. Let us assume for the moment that we know how to accumulate these scores[4]. We obtain an accumulating algebra $PROB_{acc}$ from $PROB$ by replacing the (maximizing) objective function h by the suitable accumulating function h_{acc}. By calling $G1(CAN * PROB_{acc}, x)$, we correctly compute the probabilities, accumulated over the equivalence classes modulo CAN. So, mathematically, ambiguity compensation is not a problem, and no additional programming effort is required except for the coding of h_{acc}.

However, we will experience an exponential slowdown of our program, consistent with the intractability result of [4]. The asymptotic efficiency of the algorithm is affected by the number of equivalence classes modulo CAN, which must be computed in total – and their number is, in general, exponential in the length of the input sequence. Such an approach is feasible for moderate length RNAs when equivalence classes are defined via shape abstractions [21], but when CAN simply denotes feasible structures of the input sequence, one cannot get very far by this (otherwise quite elegant) method.

[4]For example, log-probabilities must be re-converted into probabilities in order to be added, which may cause numerical problems.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

E. Ambiguity in sequence comparison: alignments versus traces

The phenomenon of semantic ambiguity is not peculiar to SCFGs. It arises with HMMs, and in fact, already with simple, pairwise sequence alignments. As with SCFGs, it depends on the meaning we associate with alignments. Seen as a syntactic object, each alignment of two sequences x and y stands for itself and is distinct from all others. But sequence alignments are often interpreted as a reconstruction of evolutionary history, where both sequences have developed from a common ancestor. Matched residues in the alignment (bases for DNA, amino acids for protein sequences) are considered preserved by evolution. Mismatches mean accepted point mutations. Gaps mean new residues that have been inserted in either x or y. (If we see the same process as evolving from x to y, "insertions" in x appear as deletions, which has no effect on the subsequent discussion.) If new sequence has been inserted in both x and y between two preserved residues, there is no particular ordering of these events. The alignment, however, offers two representations for the same fact: we may write both

```
x:   ACAGGGG---CAC           x:   ACA---GGGGCAC
y:   ACA----TTTCAC           y:   ACATTT----CAC,
```

denoting the same evolutionary history. Classical bioinformatics textbooks do not fail to point to this fact [19], [22]. Naturally, if this situation arises at k locations during the evolution of the sequences, this process has 2^k aligments representing it – significantly disturbing any stochastic model.

"Alignments" where only matches and mismatches are specified, and hence, adjacent deletions/insertions remain implicit, avoid this problem. They are called *traces* in [19], and we will adopt this naming later. An unambiguous notation for traces could be e.g.

```
x:   ACA[GGGG]CAC
y:   ACA[TTT] CAC
```

where the square brackets designate inserted sequences unordered with respect to each other. Another way to avoid ambiguity in alignments is presented later, when we return to this aspect in Section III-C.

III. Semantics of SCFG-based family models

In this section we turn our attention to SCFGs which describe RNA family models, called family model grammars for short. The previously developed SCFG terminology is not sufficient to understand their properties. We will extend it appropriately. In particular, we will find that there are three reasonable, alternative semantics for family model grammars.

A. From RNA folding SCFGs to family model grammars

There are three important differences between the SCFGs as we (and others) have used them as models for structures of individual RNA molecules, and their use in family modeling.

Family model grammars encode a consensus structure: Grammars like $G1$ or $G5$ are unrestricted RNA folding grammars. They will fold a sequence into all feasible secondary structures according to the rules of base pairing. This makes the grammars relatively small, having one rule for every structural feature considered by the scoring scheme, say a base pair or an unpaired base. The scoring scheme evaluates alternative parses and selects the result from the complete folding space of the query sequence.

This is different with grammars that model an RNA family with a particular consensus structure C. The consensus structure C is "hard-coded" in the grammar. To show a concrete consensus, we shall use star and angle brackets in place of dots and parenthesis, e.g. `"*<<*<*>>><*>*"`. This is only for clarity – there is no difference, in principle, between the consensus and ordinary structures. For every position where (say) a base pair is generated, the family model grammar has a special copy of the base pair generating production, with nonterminal symbols renamed. The general rule $S \rightarrow aSu$ becomes $S_i \rightarrow aS_{i+1}u$ for each position i where an $a{-}u$ base pair is in C. The transition parameter associated with this rule can be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

trained to reflect the probability of an $a{-}u$ pair in this particular position. The type of grammar we have seen before, therefore, only serves as a prototype from which such position-specific rules are generated.

A family consensus structure of n residues will lead to a model grammar G_C with kn productions, where k is a small constant. Hence, while folding a query with approximate length n and grammar $G1$ would require $O(n^3)$ computing steps, matching the sequence to the family grammar G_C runs in $O(n^4)$ time, simply because the size of G_C is in $O(n)$.

Family model grammars restrict the folding space of the query: A parse of a sequence x in G_C indicates a structure for x, but this structure is no longer a free folding: it is always a homomorphic image of C, with some base pairings of C possibly missing, and some residues of C possibly deleted. Still, the paired residues may be assigned to the bases of x in different ways; therefore, the structures assigned to x by different parses may vary slightly. This restriction of the folding space to "lookalikes" of C is the second difference between single sequence folding and family modeling.

Family model grammars encode the alignment of a query to the consensus: The third, important difference is that G_C implicitly aligns x to C. For example, a base assigned an unpaired status in x may represent one of three situations: it may (i) be matched to an unpaired residue in C, (ii) be an inserted base relative to C, or (iii) be matched to a paired residue in C, but without having a pairing partner in x.

These three situations are explicitly distinguished in G_C, they are scored separately, and the CYK algorithm returns the parse with maximal score based on these considerations. To achieve this, the prototype grammar needs rules which take care of deletions, insertions, and different types of matches.

Together, these three differences are central to our issue of ambiguity, and we summarize them in the following

Fact *Let M be a covariance model implemented by an SCFG G_C, which implicitly encodes the consensus structure C. Then, parsing x with G_C finds an optimal alignment of x with C which implicitly designates a structure s_x for x. This structure s_x is restricted to one of many possible homomorphic images of C obtained by deleting residues and dropping base pairings from C. There are numerous other alignments which assign the same structure s_x to x, whose (smaller) likelihood contributions are not reflected by the optimal alignment.*

B. Prototype grammar and family model example

At this point the reader rightfully expects an example of a prototype grammar and a family model grammar generated from it. We show a prototype grammar derived from $G5$ and a toy family model grammar generated from it.

The prototype grammar G5M: We extend $G5$ to obtain a prototype grammar $G5M$ capable of describing query alignments to a model. $G5M$ extends $G5$ by rules modeling insertions, deletions and matches. Again, a and b stand for arbitrary bases.

Grammar $G5M$, the axiom is A.

$$\begin{array}{lcl} A & \rightarrow & a\ A \mid M \\ M & \rightarrow & \varepsilon \mid a\ A \mid M \mid \\ & & a\ A\ b\ A \mid a\ A\ M \mid M\ b\ A \mid M\ M \end{array}$$

From a purely syntactic point of view, this grammar appears weird, because the chain rule $M \rightarrow M$ and $M \rightarrow M\ M$ together with $M \rightarrow \varepsilon$ allow for unbounded derivations that produce ε. There is no string in the language of this grammar which has a unique derivation! Ignoring all rules except $\{M \rightarrow \varepsilon,\ M \rightarrow\ a\ A,\ M \rightarrow\ a\ A\ b\ A\}$ and mapping nonterminal symbols A and M to S, we are back at $G5$. The other rules provide for insertions and deletions between the query and the model. Specialization of $G5M$ to the consensus `"*<*>*"` will yield the family model grammar G_{Toy5}. Its context-free core is shown in Fig. 6 for shortness, but G_{Toy5} actually is a tree grammar using the same signature as $G5M$. Details of the generation algorithm are in Section IV.

To make our intentions explicit, we semantically enhance the grammars by adding an evaluation function interface.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

$$
\begin{array}{ll}
A_1 & \rightarrow a\ A_1 \mid M_1 \\
M_1 & \rightarrow a\ A_2 \mid M_2 \\
A_2 & \rightarrow a\ A_2 \mid M_2 \\
M_2 & \rightarrow a\ A_3\ b\ A_5 \mid a\ A_3\ M_5 \mid\ M_3\ b\ A_5 \mid\ M_3\ M_5 \\
A_3 & \rightarrow a\ A_3 \mid M_3 \\
M_3 & \rightarrow a\ A_4 \mid M_4 \\
A_4 & \rightarrow a\ A_4 \mid M_4 \\
M_4 & \rightarrow \varepsilon \\
A_5 & \rightarrow a\ A_5 \mid M_5 \\
M_5 & \rightarrow a\ A_6 \mid M_6 \\
A_6 & \rightarrow a\ A_6 \mid M_6 \\
M_6 & \rightarrow \varepsilon
\end{array}
$$

Fig. 6. Family model grammar G_{Toy5} generated from $G5M$ for consensus $C =$ "*<*>*"

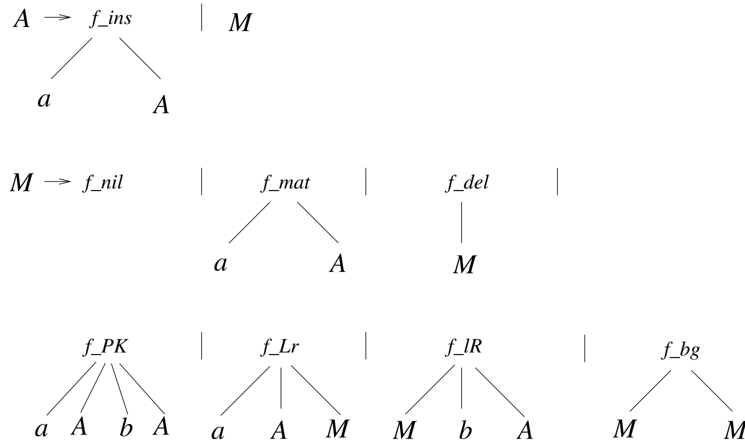

Fig. 7. Prototype grammar $G5M$ as a tree grammar. Functions f_{mat}, f_{ins} and f_{del} mark matches, insertions and deletions of unpaired residues. Functions f_{PK}, f_{Lr}, f_{lR}, and f_{bg} mark matches, partial, or total deletions of paired residues in the model.

Here is the signature:

$$
\begin{array}{llll}
f_{mat}: & \mathcal{A} \times V \rightarrow V & f_{PK}: & \mathcal{A} \times V \times \mathcal{A} \times V \rightarrow V \\
f_{ins}: & \mathcal{A} \times V \rightarrow V & f_{Lr}: & \mathcal{A} \times V \times V \rightarrow V \\
f_{del}: & V \rightarrow V & f_{lR}: & V \times \mathcal{A} \times V \rightarrow V \\
f_{nil}: & V & f_{bg}: & V \times V \rightarrow V \\
h: & [V] \rightarrow [V] & &
\end{array}
$$

Remember that $\mathcal{A}$ denotes the underlying alphabet. The tree grammar version of $G5M$ is shown in Fig. 7.

C. Three semantics for family model grammars

Matching a query x against a family model should return the maximum likelihood score of – what? There are three possibilities, which we will explicate in this section.

For the family models, derived from $G5M$, we can use the same signature as with $G5M$, except that the functions get, as an extra first argument, the position in the consensus with which they are associated. Hence, when specifying a semantics via an evaluation algebra for $G5M$, this implies the analog semantics for all generated models, as they solely consist of position-specialized rules from $G5M$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

The structure semantics: The obvious idea is to ask for the highest scoring structure assigned to x. This is in line with the semantics $\mathcal{S}_{SC\mathcal{FG}}$ introduced for SCFGs previously. Here is the canonical mapping algebra CAN_{struct}:

$$
\begin{array}{llll}
f_{mat}(a,s) & = ''.'' + s & f_{PK}(a,s,b,t) & = ''('' + s + '')'' + t \\
f_{ins}(a,s) & = ''.'' + s & f_{Lr}(a,s,t) & = ''.'' + s + t \\
f_{del}(s) & = s & f_{lR}(s,b,t) & = s + ''.'' + t \\
f_{nil} & = '''' & f_{bg}(s,t) & = s + t \\
h & = id & &
\end{array}
$$

Here again, s and t denote strings derived from substructures, and $+$ denotes string concatenation. a and b are the concrete residues read in the query. CAN_{struct} maps residues of the query x to their assigned paired or unpaired status, while residues from the consensus which are deleted (e.g. where f_{bg} applies) produce no contribution to the output. Hence, the meaning of any candidate evaluated with CAN_{struct} is simply a structure for x in dot-bracket notation.

The alignment semantics: With the alignment semantics, we want to obtain the maximum likelihood score of an alignment of the query to the consensus. This model is more refined than the structure semantics, as a given query structure can be aligned to the consensus in many different ways, and we seek the most likely of those. Let us now formalize this idea.

For capturing the alignment semantics, we must use a canonical representation that expresses not only the structure assigned to x, but also how it is aligned to the consensus structure C. Hence, it is an alignment of two strings, the consensus structure and the structure assigned to the query. Both, naturally, can be padded with gaps. The following are three different alignments of a query sequence to the same consensus:

```
(1)   **<<**--*>*>-      (2)   **<-<*---**>*>     (3)   **<<**-*>*>--
      ._(......__).            ____.(......)_.          __((....)_)..
```

Note that the upper line is always `"**<<***>*>"` when ignoring the gaps. This is because the consensus is hard-coded in the model grammar. In contrast, the query structure is `".(......)."` in alignments (1) and (2), and `"((....))..` `"` in alignment (3).

In defining the canonical mapping algebra CAN_{align} for the alignment semantics, we use functions that generate the alignment column-wise.[5]

Here is the canonical mapping CAN_{align}:

$$
\begin{array}{llll}
f_{mat}(a,s) & = ''\overset{*}{.}'' + s & f_{PK}(a,s,b,t) & = ''\overset{<}{(}'' + s + ''\overset{>}{)}'' + t \\
f_{ins}(a,s) & = ''\overset{-}{.}'' + s & f_{Lr}(a,s,t) & = ''\overset{<}{.}'' + s + ''\overset{>}{_}'' + t \\
f_{del}(s) & = ''\overset{*}{_}'' + s & f_{lR}(s,b,t) & = ''\overset{<}{_}'' + s + ''\overset{>}{.}'' + t \\
f_{nil} & = '''' & f_{bg}(s,t) & ''\overset{<}{_}'' + s + ''\overset{>}{_}'' + t \\
h & = id & &
\end{array}
$$

The trace semantics: Our third semantic idea results from the fact the good old sequence alignments have an ambiguity problem of their own. After all, we are aligning a query sequence to the model. Recall our example from Section II-E of traces of evolutionary processes that are represented ambiguously by sequence alignments:

```
x:  ACAGGGG---CAC          x:  ACA---GGGGCAC
y:  ACA----TTTCAC          y:  ACATTT----CAC
```

This directly pertains to our problem at hand if you consider x as the consensus of (say) a loop region in the model (be it a profile HMM or an SCFG), and y as the loop of a corresponding hairpin in the

[5]In the implementation, unfortunately, we have to replace the nice two-letter columns by ASCII encodings.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

query. If a stochastic model assigns a probability of 0.1 to each of the two alignments, the corresponding trace

```
x:   ACA[GGGG]CAC
y:   ACA[TTT] CAC
```

has probability 0.2 (at least). We have a case of semantic ambiguity, which must be taken care of even in stochastic sequence alignment. The Plan7 architecture of HMMer, for example, does this in a drastic way by requiring at least one intervening match when switching between deletions and insertions [17]. This simply disallows adjacent deletions and insertions altogether (but also rules out some plausible traces).

We will adopt a different route. It is easy to modify the alignment recurrences defining sequence alignment (the grammar in our terminology) such that only one of the possible arrangements of adjacent insertions and deletions is considered as a legal alignment [10]. With such canonization, each trace is uniquely represented by an alignment. The reduction is significant: For the two short sequences shown above, and under the affine gap model, there are 396,869,386 alignments, representing only 92,378 different traces[6]. Traces are considerably more abstract than alignments.

Let us return to our covariance models. Our family model grammars perform both folding and alignment, and hence, they are also affected by this source of ambiguity – at least if we intend that final score designates the most likely evolutionary process that relates the query to the model. The case even becomes more subtle. The following alignment (4) denotes the same trace as alignment (2):

```
(2) C:   **<-<*---**>*>        (4) C:   **-<<*---**>*>
    x:   ___.(......)_.            x:   __._(......)_.
```

What both alignments say is that a paired residue (at position 3) in the consensus C is deleted in x, while another base is inserted in x. As with plain sequence alignments, adjacent deletions and insertions are unrelated; their order is insignificant.

Hence, it makes sense to introduce a *trace semantics* for our family model grammars: we want to obtain the maximum likelihood score of a trace, which uniquely describes an evolutionary process of transforming the consensus into the query.

To capture this idea, we need to design another canonical algebra CAN_{trace}, which maps these two situations (2) and (4) above to the same, unique representation. Let us adopt the canonization rule that insertions must always precede adjacent deletions. By this rule, both alignments (2) and (4) are represented in the form of (4). The canonical mapping algebra CAN_{trace} is almost the same as CAN_{align}, except that deletions that appear to the left of an insertion are pushed to the right.

Algebra CAN_{trace}

$$
\begin{array}{llll}
f_{mat}(a,s) & = \text{"}\overset{*}{.}\text{"} + s & f_{PK}(a,s,b,t) & = \text{"}\overset{<}{(}\text{"} + s + \text{"}\overset{>}{)}\text{"} + t \\
f_{ins}(a,s) & = \text{"}\overset{-}{.}\text{"} + s & f_{Lr}(a,s,t) & = \text{"}\overset{<}{.}\text{"} + s + \text{"}\overset{>}{_}\text{"} \triangleright t \\
f_{del}(s) & = \text{"}\overset{*}{_}\text{"} \triangleright s & f_{lR}(s,b,t) & = \text{"}\overset{<}{_}\text{"} \triangleright s + \text{"}\overset{>}{.}\text{"} + t \\
f_{nil} & = \text{""} & f_{bg}(s,t) & = \text{"}\overset{<}{_}\text{"} \triangleright s + \text{"}\overset{>}{_}\text{"} \triangleright t \\
h & = id & &
\end{array}
$$

$$
\begin{array}{lll}
d \triangleright (a+s) & = \text{ if } a = \text{"}\overset{-}{.}\text{"} \quad \text{then } a + (d \triangleright s) & \text{else } d + a + s \\
d \triangleright \varepsilon & = d &
\end{array}
$$

Wherever a deletion is issued, we have replaced simple string concatenation (+) by the operation $\triangleright$ which moves the deletion to the right over any leading insertions.

[6]Computed with the ADP versions of classical dynamic programming algorithms at http://bibiserv.techfak.uni-bielefeld.de/adp/adpapp.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

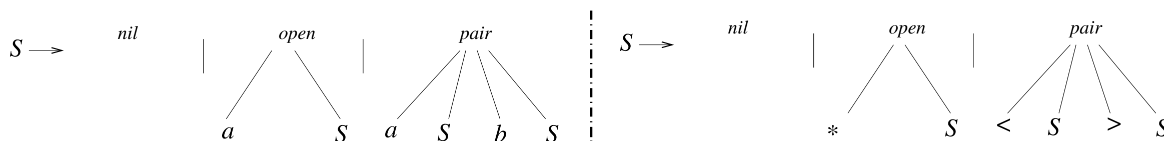

Fig. 8. Grammar $G5$ as a tree grammar for parsing sequences (left) and consensus structures (right).

The semantic hierarchy: Our three semantics form a proper hierarchy – many alignments correspond to the same trace, and many traces assign the same structure to the query. This also implies that a family model which faithfully (unambiguously) implements the alignment semantics is ambiguous with respect to the trace semantics, and one which faithfully implements the trace semantics is ambiguous with respect to the structure semantics, which is the most abstract of the three.

Which semantics to choose? When we are mainly interested in a structure prediction for the query, indicating why x may perform the same catalytic or regulatory function as the family members, then the structure semantics may be most appropriate. When we are interested in estimating the evolutionary closeness of the query to the family members, the trace semantics seems adequate. For the alignment semantics, at the moment we see no case where it should be preferred.

But – can we generate unambiguous family model grammars and efficiently compute either of the three semantics?

IV. Generating non-ambiguous family models for the trace semantics

In this section we show how family model grammars can be generated which are non-ambiguous with respect to the trace semantics. This will also provide a deeper insight on the meaning of the prototype grammar [7]. We proceed in the following steps: (1) We start from a non-ambiguous prototype grammar. (2) We show how, given a consensus structure C, a model grammar G_C is constructed which generates alignments in the canonical form (insert-before-delete), as required for the trace semantics. (3) We give a proof that for any C, G_C is non-ambiguous under the trace semantics.

Here, we use grammar $G5$, because it is the smallest non-ambiguous grammar. However, the generating technique and proof carries over to any non-ambiguous prototype grammar, which might be more attractive than G5 from the parameter training point of view.

The meaning of prototype grammars: Starting from $G5$, our prototype grammar is $G5M$. We still owe the reader the explanation why this grammar looks the way it does. The key point of $G5M$ is that it enforces the insert-before-delete convention. Only nonterminal symbol A allows for insertions. Whenever a nonterminal symbol stands in the left context of a deletion, an M rather than an A is used.

The real understanding of the prototype grammar comes from the observation that the prototype grammar $G5M$ is a grammar that allows to align a query to *all* possible models:

There is no *specific* model encoded in $G5M$. This is why the grammar can be so small. But each derivation with $G5M$ not only assigns a structure to the query, but also implicitly encodes a model, chosen by that derivation. This meaning of the prototype grammar can be made apparent by plugging the definitions of CAN_{trace} into the tree grammar $G5M$ and symbolically evaluating a bit. Doing so, the tree operators like f_{mat} or f_{PK} are replaced by string concatenations, and we obtain the string grammar $\widehat{G5M}$:

Grammar $\widehat{G5M}$; the axiom is A.

[7]The reader may find it helpful to inspect the actual implementation of the generator and run simple experiments. We therefore have provided the generator among our educational ADP pages at http://bibiserv.techfak.uni-bielefeld.de/adp/nilpairopen.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

$$
\begin{aligned}
A &\rightarrow \overset{-}{\bullet}\ A \ \mid\ M \\
M &\rightarrow \varepsilon \ \mid\ \overset{*}{\bullet}\ A \ \mid\ \overset{*}{_}\ M \ \mid \\
&\quad \overset{<}{(}\ A\ \overset{>}{)}\ A \ \mid\ \overset{<}{\bullet}\ A\ \overset{>}{_}\ M \ \mid\ \overset{<}{_}\ M\ \overset{>}{\bullet}\ A \ \mid\ \overset{<}{_}\ M\ \overset{>}{_}\ M
\end{aligned}
$$

This grammar transformation is not totally trivial because of the use of $\triangleright$ in the definitions of CAN_{trace}. But from the grammar, we observe that canonical strings derived from M cannot start with insertions ($\overset{-}{\bullet}$), while deletions ($\overset{*}{_},\overset{<}{_},\overset{>}{_}$) are only applied before M. Hence, this grammar guarantees that the if-clause in the definition of $\triangleright$ is never positive, and the recursive call to $\triangleright$ disappears. (Since the use of $\triangleright$ is the only difference between CAN_{trace} and CAN_{align}, transforming $G5M$ with CAN_{align} leads to the same string grammar $\widehat{G5M}$).

What does $\widehat{G5M}$ explain about $G5M$? Replaying any derivation of $G5M$ with the analog productions of $\widehat{G5M}$ produces the representation of a model-structure alignment. The top line displays the model "chosen" in this derivation, the bottom line displays the structure assigned to the query. Considering only the model string on the top line, we find that it is produced by productions analog to $G5$, and hence, any consensus structure is possible.

For example, running $\widehat{G5M}$ on input `"au"` produces an infinite number of model/query alignments. This is correct, since models of any length can be aligned to any sequence with a suitable number of deletions. Disabling for a moment the rules which delete unpaired model residues or both residues in a pair (i.e. the uses of f_{del} and f_{bg}), which are the sources of such infinity, the prototype grammar $\widehat{G5M}$ generates the following 23 alignments via the call $G5M(CAN_{trace},$ `"au"`$)$:

"--" ".."	"**" ".."	"<>*" "._."	"<<>>" "..__"	"<><>" "_._."	"-*" ".."	"*<>" ".._"	"<><>" "._._"	"<<>>" "._._"	"<*>" "_.."
"-<>" ".._"	"*<>" "._."	"<><>" ".__."	"<>-" "_.."	"<<>>" "_._."	"-<>" X "._."	"<>" "()"	"<->" X ".._"	"<>*" "_.."	"<<>>" "__.."
"*-" ".."	"<*>" ".._"	"<><>" "_.._"							

Note that we see two alignments (labeled X) that satisfy the insert-before-delete convention, but not their counterparts with delete-before-insert, which is forbidden with the trace semantics. Let us summarize our observations about the role of the prototype grammar.

Fact *The prototype grammar describes, by virtue of its derivations, the alignment of a query to all possible consensi. Generating a specific family model grammar amounts to restricting the prototype grammar, such that all its derivations align the query to the same model consensus.*

In other words, in a family model grammar for consensus structure C, the "upper line" in a derivation always spells out C.

Generating model grammars from consensus structures: We now construct a generator which reads a consensus structure C such as "$\star\star\star<<<\star\star\star\star\star>>\star>\star\star$" and generates a grammar $G5M_C$ which implicitly encodes alignments of a query sequence x to C. With the ADP method at our disposal, we can use a variant of tree grammar $G5$ to parse C, obtained by substituting $\star$ for unpaired residues and < and > for paired ones (cf. Fig. 8 (right)). Since $G5$ is non-ambiguous, there will be only one tree t_C for C. We design an evaluation algebra $genCM$ which generates $G5M_C$ by evaluating t_C. For the sake of explanation, we will proceed in two steps: first we design an algebra $genCFG$ which generates $G5M_C$ as a context free grammar, to explain the logic of the algorithm. Then, we modify $genCFG$ to $genCM$ which generates a tree grammar, i.e. executable ADP code for the model.

$genCFG$ has to take care of two issues. (1) It must generate copies of the rules of $G5M$, specialized to the specific positions in C. Applying (say) rule $M \rightarrow aAbA$ when a and b are at paired positions i and j in C, respectively, will produce the specialized production $M_i \rightarrow aA_{i+1}bA_{j+1}$. (2) $genCFG$ must allow for insertions and deletions without introducing ambiguity. But this has already been taken care of in the design of $G5M$. As long as $genCFG$ only uses position-specialized copies of the rules from $G5M$, this property is inherited.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

Evaluation algebra $genCFG$; the value domain is sets of context-free productions:

$$
\begin{aligned}
nil(\varepsilon_i) &= \{A_i \to a\ A_i \ \mid\ M_i\} \cup \{M_i \to \varepsilon\} && (1)\\
open(a_i, x) &= x \cup \{A_i \to a\ A_i \ \mid\ M_i\} && (2)\\
&\quad \cup \{M_i \to a\ A_{i+1} \ \mid\ M_{i+1}\} && (3)\\
pair(a_i, x, b_j, y) &= x \cup y \cup \{A_i \to a\ A_i \ \mid\ M_i\} && (4)\\
&\quad \cup M_i \to\ a\ A_{i+1}\ b\ A_{j+1}\} && (5)\\
&\quad \cup M_i \to\ a\ A_{i+1}\ M_{j+1}\} && (6)\\
&\quad \cup M_i \to\ M_{i+1}\ b\ A_{j+1}\} && (7)\\
&\quad \cup M_i \to\ M_{i+1}\ M_{j+1}\} && (8)
\end{aligned}
$$

Here, subscripts denote the position where a particular production is applied in the parse of C. In the output, these numbers create nonterminal symbols distinguished by subscripts. By default, the axiom of the generated grammars is A_1. Our reader may verify: computing $G5(genCFG, $ `"*<*>*"`$)$ yields the grammar $Toy5$ shown in Fig. 6.

Finally, to produce executable code, $genCM$ must generate a tree grammar rather than a string grammar, in order to integrate the scoring functions. The rules of the context-free grammar derived with $genCFG$ are now associated with scoring functions from the signature. As we cannot produce graphical output, a tree build from function symbol f and subtrees a, A, b, A is coded in the form `f <<< a~~~A~~~b~~~A`.

Evaluation algebra $genCM$; the value domain is sets of tree grammar productions written in ASCII:

$$
\begin{aligned}
nil(\varepsilon_i) &= \{\texttt{A_i = f_ins <<< a \textasciitilde\textasciitilde\textasciitilde\ A_i ||| M_i}\} && (9)\\
&\quad \cup \{\texttt{M_i = f_nil<<<empty}\} && (10)\\
open(a_i, x) &= x \cup \{\texttt{A_i = f_ins <<< a \textasciitilde\textasciitilde\textasciitilde\ A_i ||| M_i}\} && (11)\\
&\quad \cup \{\texttt{M_i = f_mat <<< a \textasciitilde\textasciitilde\textasciitilde\ A_i+1 ||| f_del <<< M_i+1}\} && (12)\\
pair(a_i, x, b_j, y) &= x \cup y \cup \{\texttt{A_i = f_ins <<< a \textasciitilde\textasciitilde\textasciitilde\ A_i ||| M_i}\} && (13)\\
&\quad \cup \{\texttt{M_i = f_PK <<< a \textasciitilde\textasciitilde\textasciitilde\ A_i+1 \textasciitilde\textasciitilde\textasciitilde\ b \textasciitilde\textasciitilde\textasciitilde\ M_j+1}\} && (14)\\
&\quad \cup \{\texttt{M_i = f_Lr <<< a \textasciitilde\textasciitilde\textasciitilde\ A_i+1 \textasciitilde\textasciitilde\textasciitilde\ M_j+1}\} && (15)\\
&\quad \cup \{\texttt{M_i = f_lR <<< M_i+1 \textasciitilde\textasciitilde\textasciitilde\ b \textasciitilde\textasciitilde\textasciitilde\ A_j+1}\} && (16)\\
&\quad \cup \{\texttt{M_i = f_bg <<< M_i+1 \textasciitilde\textasciitilde\textasciitilde\ M_j+1}\} && (17)
\end{aligned}
$$

Compared to our use of the same signature with (the non-specialized) $G5$, all scoring functions take i as an implicit parameter, so calls to (say) f_{del} from different positions may be trained to assign different probabilities.

Non-ambiguity of generated models: We want to prove next that our model generator $G5(genCM, C)$ generates, for every consensus structure C, a family model grammar which is unambiguous with respect to the trace semantics. The proof consists of two theorems:

Theorem 2 *Grammar $G5M$ is unambiguous with respect to the trace semantics.*

We might strive for an inductive proof of this theorem, but since we already have all the necessary machinery in place, we use an automated proof technique.

From $G5M$ we construct $\widehat{G5M}$ as explained in Section II-C. We have already observed that its derived alignments comply with the insert-before-delete-convention. Therefore, the generated alignments in fact denote traces. Remember that $G5M$ generates the same model-query alignment several times if and only if $\widehat{G5M}$ is syntactically ambiguous. We replace the fancy, two-character columns by single character encodings according to the following table:

$\overset{*}{\bullet}$	$\overset{-}{\bullet}$	$\overset{*}{_}$	$\overset{<}{(}$	$\overset{>}{)}$	$\overset{<}{\bullet}$	$\overset{>}{_}$	$\overset{<}{_}$	$\overset{>}{\bullet}$	$\overset{<}{_}$	$\overset{>}{_}$
M	I	D	P	K	L	r	l	R	b	g

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

This turns $\widehat{G5M}$ into the grammar

$$A \rightarrow \text{"}I\text{"}\ A \mid M \tag{18}$$
$$M \rightarrow \varepsilon \tag{19}$$
$$M \rightarrow \text{"}M\text{"}\ A \mid \text{"}D\text{"}\ M \tag{20}$$
$$M \rightarrow \text{"}P\text{"}\ A\ \text{"}K\text{"}\ A \tag{21}$$
$$M \rightarrow \text{"}L\text{"}\ A\ \text{"}r\text{"}\ M \tag{22}$$
$$M \rightarrow \text{"}l\text{"}\ M\ \text{"}R\text{"}\ A \tag{23}$$
$$M \rightarrow \text{"}b\text{"}\ M\ \text{"}g\text{"}\ M \tag{24}$$

which is proved unambiguous by the *acla* ambiguity checker [3].
Q.E.D.

We can now show that by the generation algorithm, semantic non-ambiguity is inherited from $G5$ to the family model grammars.

Theorem 3 *Covariance models generated from a consensus structure C by $G5(genCM, C)$ are semantically non-ambiguous under the trace semantics.*

We note the following facts:

1) $\widehat{G5M}$ is syntactically non-ambiguous (Theorem 2).
2) Each derivation in $G5M$ describes an alignment of a query against *some* model.
3) By construction, all these alignments observe the insert-before-delete convention.
4) Any derivation in a generated model grammar $G5M_C$ can be mapped to a derivation in $G5M$. This is achieved by applying, for each production from $G5M_C$, the corresponding production without the subscripts form $G5M$. This means that all derivations $G5M_C$ also observe the insert-before-delete convention.
5) This mapping is injective. This holds because we can uniquely reconstruct the positional indices to turn a $\widehat{G5M}$ derivation back into a $\widehat{G5M_C}$ derivation, by keeping track of the number of symbols from $\{M, D, P, K, L, l, R, r, b, g\}$ generated so far (but not counting I).
6) Hence, if $G5M_C$ was ambiguous, $G5M$ would also be ambiguous, in contradiction to point (1).

Altogether, if there was a trace that had two different derivations in $G5M_C$, it would also have two different derivations in $G5M$. This is impossible according to point (1). Hence, a model grammar $G5M_C$ generated by $genCM$ is always non-ambiguous with respect to the trace semantics.
Q.E.D.

The correctness proof for the model generator here crucially depends on the non-ambiguity of the prototype grammar. When a prototype grammar GM is ambiguous, a sophisticated generator can still avoid ambiguity in the generated models! However, in this case a proof might be difficult to achieve. If it fails, we can still convert each generated model GM_C into the corresponding $\widehat{GM_C}$, which can be submitted to ambiguity checking. This is the situation we will encounter when turning towards the "real-world" models in Rfam. There, we have an ambiguous prototype grammar and a sophisticated generation process, which makes it hard to prove properties about. Therefore, we next equip ourselves with an automated pipeline for ambiguity checking of Rfam models.

V. THE AMBIGUITY CHECKING PIPELINE

Our ambiguity checking pipeline consists of three successive stages, named *cm2adp*, *adp2cfg*, and *acla*.

cm2adp: Upward compilation of Infernal generated covariance models: The upward compiler *cm2adp* accepts as input the table encoding a covariance model generated by *Infernal*. It translates it into the constituents of a mathematically equivalent ADP algorithm – a tree grammar, a signature, and an implementation of the stochastic scoring algebra using the parameters generated by *Infernal*. Once available in this form, additional evaluation algebras can be used in place of or jointly in products with the stochastic scoring algebra. Such semantic enrichment was the main purpose of developing *cm2adp*, and its scope

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

will be described in a forthcoming paper. One of these applications is the evaluation of the search space under a canonical mapping algebra, as we do here.

adp2cfg: Partial evaluation of grammar and canonical mapping algebra: The *adp2cfg* program is a simple utility implemented by Peter Steffen subsequent to [16]. It accepts a tree grammar G and a canonical mapping algebra A, such that a call to $G(A,x)$ for some query x, would enumerate all the members of the search space (i.e. all parses) under the canonical string mapping. Provided that the algebra A is very simple and uses only string constants and concatenation, *adp2cfg* succeeds with partial evaluation to produce the context free (string) grammar $\widehat{G}$ suitable for ambiguity checking according to Theorem 1.

acla: Ambiguity checking by language approximations: The *acla* phase simply calls the ACLA ambiguity checker for context free grammars, which is based on the recent idea of ambiguity checking via language approximations [3]. It has been used before, for example, on the grammar designed by Voss for probabilistic shape analysis of RNA [21]. Accumulating probabilities from the Boltzmann distribution of structures depends, just like stochastic scoring, critically on semantic non-ambiguity.

Due to the undecidability of the ambiguity problem, there is no guarantee that the *acla* phase will always return a definite answer. It may be unable to decide ambiguity for some covariance models. However, since the covariance models are larger, but less sophisticated than the grammar by Voss, we are confident that the formal undecidability of ambiguity will not be a practical obstacle in our context.

The overall pipeline: As all family model grammars derived from the same prototype grammar use the same signature, the evaluation algebra implementing the canonical mappings for the structural and the alignment semantics, CAN_{struct} and CAN_{align}, is the same for all, as described above. Let M denote a covariance model generated by *Infernal* from consensus structure C, given in *Infernal*'s tabular output format.

Let $(G_C, PROB) = \mathit{cm2adp}(M)$ be the ADP program equivalent to M, generated by upward compilation.

Let $\widehat{G_{C,S}} = \mathit{adp2cfg}(G_C, CAN_S)$ be the context free grammar generated by partial evaluation, where CAN_S is either CAN_{struct} or CAN_{align}.

Then, $\mathit{acla}(\widehat{G_{C,S}}) \in \{YES, NO, MAYBE\}$ demonstrates semantic ambiguity or non-ambiguity of M with respect to the semantics S.

The trace semantics cannot be handled by *adp2cfg* because the recursive auxiliary function $\triangleright$ in CAN_{trace} can only be eliminated with an inductive argument. To demonstrate (non-)ambiguity with respect to the trace semantics, one shows (non-)ambiguity with respect to the alignment semantics plus (non-)observance of a uniqueness constraint such as the insert-before-delete convention. We now proceed to apply this pipeline.

VI. SEMANTICS OF RFAM FAMILY MODELS

A. Model construction with Infernal

In this section, we look at covariance models as generated by *Infernal*. The difficulty here is that the prototype grammar is ambiguous and we do not have a fully formal specification of the generation algorithm. In order to create some suspense, we start with two quotations. The original publication [8] of 1994 states:

> "...we make the Viterbi assumption that the probability of the model emitting the sequence is approximately equal to the probability of the single best alignment of model to sequence, rather than the sum of all probabilities of all possible alignments. The Viterbi assumption conveniently produces a single optimal solution rather than a probability distribution over all possible alignments."

This points at an alignment or a trace semantics. In a more recent update, the *Infernal* Manual [14] touches on the issue of semantic ambiguity in the description of the model generation process, stating:

> "This arrangement of transitions guarantees that (given the guide tree) there is unambiguously one and only one parse tree for any given individual structure. This is important. The algorithm will find a maximum likelihood

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

parse tree for a given sequence, and we wish to interpret this result as a maximum likelihood structure, so there must be a one-to-one relationship between parse trees and structures."

This seems to aim at a structure semantics, but since the same structure can always be aligned to the consensus (alias the "guide tree") in many ways, there *must* always be several parses for it, the scores of which should accumulate to obtain the likelihood of the structure.

Infernal starts from an initial multiple sequence alignment and generates models in an iteration of consensus estimation, model generation, and parameter training. Here we are concerned with the middle step, model generation from a given (current) consensus. The family consensus structure C is determined with an ambiguous grammar, parsing the multiple alignment and maximizing a mutual information score, and then one optimal parse (out of many) is fixed as the "guide tree". (In our construction, when C is given, this is simply the unique parse of C with tree grammar $G5$.) This guide tree is then used to generate productions by specializing the following prototype grammar:

Grammar $G_{infernal}$ taken from the *Infernal* manual [14]:

State type	Description	Production	Emission	Transition
P	(pair emitting)	$P \to aYb$	$e_v(a,b)$	$t_v(Y)$
L	(left emitting)	$L \to aY$	$e_v(a)$	$t_v(Y)$
R	(right emitting)	$R \to Ya$	$e_v(a)$	$t_v(Y)$
B	(bifurcation)	$B \to SS$	1	1
D	(delete)	$D \to Y$	1	$t_v(Y)$
S	(start)	$S \to Y$	1	$t_v(Y)$
E	(end)	$E \to e$	1	1

Here, Y is any state[8] chosen from the nonterminal symbols (state types) in the leftmost column. One recognizes the rules of the ambiguous $G1$ in the guise of $\{P \to aYb, L \to aY, R \to Ya, B \to SS, E \to \varepsilon\}$. The ambiguity inherent in a rule like $S \to SS$, parsing SSS both as $(SS)S$ and $S(SS)$ is *not* a problem in model generation, because the specialized rules $S_i \to S_j S_k$ are always unambiguous. However, insertions can be generated both from L and R, possibly competing for the generation of the same unpaired residues in the query.

$G_{infernal}$ is not really the complete prototype grammar in our sense, as rules for partial matches of base pairs in the consensus need to be added in the generation process. Overall, the generation method appears too complicated to strive for a formal proof of non-ambiguity of the generated models.

B. Checking Rfam models

We have checked 30 models from Rfam, the 15 smallest models with and without a bifurcation in their consensus structure, respectively. Model names and their consensus structures are listed in the appendix. Here, we give a resume of our findings:

Theorem 4 *In general, Rfam models are ambiguous with respect to the structure semantics. They do not assign a most likely structure to the query.*

This can be seen from testing with our pipeline, but is also easily seen by inspecting the generated models. Actually, alignments (1) and (2) in Section III-C are already an example of ambiguity with respect to the structure semantics, though only in principle, as they are not Rfam models. The explanation is that although *Infernal* takes care that the structural ambiguity of the prototype grammar does not enter the model grammar, it does not compensate for the fact that the same structure (assigned to the query) is aligned to the model in many ways. Hence, the score accounts for the structure associated with the optimal alignment, which need not be the highest scoring structure.
Q.E.D.

[8] The description in [14] uses a mixture of SCFG and HMM terminology.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

Theorem 5 All tested Rfam models are non-ambiguous with respect to the alignment semantics. There is no evidence that this result should not carry over to Rfam models in general.

This observation was proved for some of the smallest models via producing their grammar $\widehat{G}$ and submitting it to the ambiguity checker. For larger models, the $ACLA$ checker ran out of resources. We applied some surgery by reducing successive stretches of either unpaired or paired residues (in the model) to stretches of at most three such residues. This is correct as it has already been tested that the rules within such stretches do not lead to ambiguity. After such surgery, the $ACLA$ checker succeeded for all models except Rf00161 and Rf00384.

For these two models, we resorted to a checking technique (rather than a proof) by use of a non-ambiguous reference grammar, as suggested in [16]: if we have a reference grammar R which generates non-ambiguously the alignments of a query to the given model, then we can compare the *number* of alignments produced by both grammars for a given input length[9]. The enormous size of the search space provides strong evidence that, if the number of alignments considered by either grammar co-incides, the tested model grammar is also unambiguous. To apply this technique, we implemented a second $G5$-based model generator to generate family model grammars that are unambiguous for the alignment semantics. Let us call them G5.Rf00161 and G5.Rf00384. We then checked, using an evaluation algebra $COUNT$ which simply counts the number of solutions generated, for sequences x of various lengths that $Rf00161(COUNT, x) = G5.Rf00161(COUNT, x)$ and $Rf00384(COUNT, x) = G5.Rf00384(COUNT, x)$. For example, the value for $|x| = 10$ is 357,718,985,217,153 (Rf00161) and 261,351,290,279,573 (Rf00384). For $|x| = 20$, it is 774,380,024,914,343,603,750,401 (Rf00161) and 416,290,325,523,207,008,752,681 (Rf00384), computed independently by both models.
Q.E.D.[10]

The positive result that Rfam models correctly implement the alignment semantics is quite remarkable, given the notorious ambiguity introduced by the rules of $G1$, such as $S \to SS$ or $S \to aS|Sa$. This is achieved by details of the *Infernal* implementation. Applications of $S \to SS$ are made unambiguous by the use of the "guide tree", effectively choosing one of the many possible derivations of the consensus structure. Ambiguity effects of $S \to aS|Sa$ are avoided by disabling one of the alternatives in certain situations. Last not least, for searching with a model, *Infernal* has recently switched to using the Inside rather than the CYK algorithm [15], which changes the scoring but bypasses eventual ambiguity problems. However, for optimally aligning a sequence to the model, and hence also for model building, the CYK algorithm is still required. We will return to the use of the Inside algorithm in the conclusion.

Theorem 6 In general, Rfam models are ambiguous with respect to the trace semantics.

This is implied by our previous observations, as a trace corresponds to many alignments.
Q.E.D.

We also wondered whether the Rfam models could be tweaked to compute the trace semantics rather than the alignment semantics, simply by disabling some of the generated transitions (and re-training the parameters). Our upward compilation allows us to eliminate certain transitions. We have been able to reduce the number of alignments considerably, but we have not found a way to reduce it to the number of traces.

C. A synopsis on RF00163 and RF01380

To give an impression of the degree of ambiguity observed with respect to structure and trace semantics, we compute some data for RF00163 and for RF01380, which are currently the smallest Rfam models

[9]Note that the number of alignments only depends on the length of model and query, but not on the concrete query sequence, and not on the grammar which implements the model.

[10]Strictly, this is not proved but only tested for Rf00161 and Rf00384 , but note that by throwing more computational ressources at the problem, we can prove the remaining candidates nonambiguous. For practical concerns, and with an eye on the other models not explicitly studied here, a quick check by the counting method is more appropriate.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

with and without a bifurcation[11]. When counting structures for $n = 31$, computational resources were exhausted, as in contrast to traces and alignments, we have no polynomial time algorithm for counting structures – for the same reasons as discussed with ambiguity compensation.

Model consensus length (size)	RF00163 <<<<<<*******<<<<******>>>>***<<<<>>>>*>>>>>> 45 (31)	RF01380 <<<<<<<****>>>*>>>> 19 (12)
$\lvert x \rvert = 12$		
structures	8,958	2,048
traces	35,330,137,025	141,120,525
alignments	715,114,268,248,121	35,330,137,025
$\lvert x \rvert = 31$		
structures	n.a.	n.a.
traces	1,916,865,700,889,408,588,480	30,405,943,383,200
alignments	1,862,835,338,733,448,037,194,650,687	208,217,738,981,165,823

The numbers of structures, traces and alignments are properties of the search space of a model/query alignment, independent of which algorithm traverses this search space. They have been computed twice, and they perfectly agree between the upward compiled *Infernal* models resulting from *cm2adp* and the models generated by our method from $G5M$. This can be taken as a strong indication that the two approaches are equivalent in terms of the search spaces they create. However, different grammars lead to different numbers of parameters and will, in general, not be equivalent as stochastic models.

D. *Ambiguity compensation, revisited*

Can we, given the trained and upward-compiled Rfam models, which compute the log-likelihood score according to the alignment semantics, use the same models with an accumulating scoring function to obtain the trace or the structure semantics? Mathematically, yes. What was explained in Section II-D for the simple stochastic context free grammars generally holds for all dynamic programming algorithms which can be expressed in the ADP framework, and hence also for our model grammars: Given grammar G, a scoring algebra S and an algebra CAN (i.e. CAN_{struct} or CAN_{trace}), just replace the maximizing objective function of S by an accumulating function, yielding scoring algebra S_{acc}. Then call $G(CAN * S_{acc}, x)$. The product algebra $CAN * S_{acc}$ maps all candidates to their equivalence classes under the canonical mapping, and performs accumulating scoring per class. Given the ADP machinery, which provides a generic implementation of the algebra product, ambiguity compensation comes without reprogramming efforts – but only in principle.

There are two obstacles to this elegant solution:

- In case of stochastic modeling, rather than multiplying probabilities which tend towards zero as the structure gets larger, one prefers to add their logarithms. Due to the monotonicity of the log function, the maximal log score still indicates the answer with maximum probability. However, substituting maximization by accumulation, one needs to compute and add the probabilities, potentially creating numerical problems.
- Efficiency of computing with $G(CAN * S_{acc}, x)$ depends on the number of canonical objects constructed, and as this number (in the case of covariance models) is exponential in the length of the query, this is practical only for very short sequences and small models. The implementation by a product algebra will have efficiency of $O(\alpha^n n^4)$, where α may be close to 1, but probably not close enough. Our counting results on the small models RF00163 and RF01380 indicate this.

[11]The number of structures without a bifurcation for $|x| = n$ is bounded from above by 2^{n-1}. The bound is sharp when $n < 2(p + 1)$, where p is the number of base pairs in the model. For larger n, the restriction that the query cannot be assigned more base pairs than the consensus becomes effective. This is why we see a 2^{11} for $n = 12$ and still cannot compute the exact number for $n = 31$.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

Semantics	Direct computation in $\mathcal{O}(n^4)$ with	Computation by ambiguity compensation in $\mathcal{O}(\alpha^n \cdot n^4)$ with
alignment	$G_{\text{ali}}(\text{PROB}, q)$	—
trace	$G_{\text{trace}}(\text{PROB}, q)$	$G_{\text{ali}}(\text{CAN}_{\text{trace}} * \text{PROB}_{\text{acc}}, q)$
structure	—	$G_{\text{ali}}(\text{CAN}_{\text{struct}} * \text{PROB}_{\text{acc}}, q)$ $G_{\text{trace}}(\text{CAN}_{\text{struct}} * \text{PROB}_{\text{acc}}, q)$
sequence	$G_{\text{ali}}(\text{PROB}_{\text{acc}}, q)$ $G_{\text{trace}}(\text{PROB}_{\text{acc}}, q)$	$G_{\text{ali}}(\text{CAN}_{\text{seq}} * \text{PROB}_{\text{acc}}, q)$ $G_{\text{trace}}(\text{CAN}_{\text{seq}} * \text{PROB}_{\text{acc}}, q)$

TABLE I
THE SEMANTIC HIERARCHY. WE INDICATE GRAMMARS AND EVALUATION ALGEBRAS USED FOR EACH TASK. n IS THE LENGTH OF THE QUERY q. α DENOTES THE BASE OF THE EXPONENTIAL FACTOR, WHICH IS INCURRED WITH AMBIGUITY COMPENSATION. α DECREASES FROM TOP TO BOTTOM; FOR THE SEQUENCE SEMANTICS, $\alpha = 1$, AND BOTH COLUMNS DESCRIBE THE COMPUTATION VIA THE INSIDE ALGORITHM, USED WITH EITHER GRAMMAR.

VII. CONCLUSION

A. *Summary of results*

We have studied the problem of generating non-ambiguous family models from consensus structures. We clarified the notion of a semantics for family model grammars, and found that there are three well motivated, alternative definitions: the structure, the trace and the alignment semantics.

We developed the generation algorithm for the trace semantics, which, to our knowledge, has not been studied before. Along the way, we found a nice explanation of the prototype grammar as a grammar that allows for an infinite set of derivations, describing the alignment of the query to *all* possible models. The generation process can then be described lucidly by an evaluation algebra ($genCM$), which allows, for example, for a proof of non-ambiguity of the generated models.

For a summary of the semantic hierarchy, let us introduce yet another semantics. The *sequence semantics* assigns to each model/query alignment the same object of interest – the aligned query sequence itself. The canonical mapping algebra CAN_{seq} is trivial and left to the reader. Ambiguity compensation with respect to this mapping means summing up probabilities of *all* model/query alignments – this is commonly known as the Inside algorithm! According to this view, our intermediate semantic levels of trace and structure semantics can, alternatively, be viewed as as intermediates between CYK and Inside scoring, governed the equivalence relation induced by the canonical mapping. This view is summarized in Table 1. Note the lack of a grammar G_{struct}, which would allow for the polynomial-time computation of the structure semantics.

On the practical side, we have implemented the upward compilation of *Infernal* generated models to ADP. Here this compilation was used for connecting the Rfam models to our ambiguity checking pipeline. The upward compiled models, however, have other applications of interest, which will be described in a forthcoming study. But still, upward compilation from automatically generated tables is an ad-hoc measure, and in the long run, one might consider producing ADP code for the models directly when generated.

Also on the practical side, we have observed that the models generated from $G5M$ are relatively small. To extend the comparison, we have also implemented a $G5$-based generator for (provably) unambiguous family model grammars and the alignment semantics. Applying both our generators to Rf00163 and Rf01380, we can give concrete examples of the blow-up factor k (cf. Section III-A). We evaluate the size of the generated grammars.

Model	Model length/size	Rfam rules/nonterminals	$G5$ (alignment) rules/nonterminals	$G5$ (trace) rules/nonterminals
Rf00163	45 / 31	617 / 139	151 / 46	182 / 77
Rf01380	19 / 12	282 / 59	66 / 20	78 / 32

The factor (number of rules/model length) affects the runtime as a constant factor. It is about 14 for the Rfam models, 3.4 for the models derived from $G5$ with alignment semantics, and 4.1 for $G5M$-derived

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

models with the trace semantics. These factors cannot be directly compared, as *Infernal* implements affine gap scoring and some other features not included in the $G5$-based models. The factor (number of nonterminals/model size) measures the space requirements, as each nonterminal leads to a dynamic programming table. Here, the respective factors are 4.6, 1.5, and 2.5, approximately.

B. Directions of future research

We shortly sketch some research questions which are raised by our findings.

Investigation of the trace semantics: The trace semantics is new; it can be efficiently computed, and possibly, the performance of covariance models can be improved. Such an improvement is likely especially with respect to remote family members. This is because, when model and query have about the same length, one is likely to find a high-scoring alignment without adjacent deletions and insertions, whose score is not falsely reduced by ambiguity. Remote family members may require more insertions and deletions, some of them adjacent, and ambiguity strikes on a scale which is exponential in the number of such situations. With an eye on the use of the alignment semantics with Rfam, this implies that good scores can be taken as a strong indication of family membership, while low scores must be interpreted with care, especially when model and query significantly differ in length.

Investigation of the structure semantics: The structure semantics has been used so far with simple SCFGs, but not with family model grammars. The structure semantics seems appropriate when the goal is to use the information in the family model to assign a concrete, most likely structure to the query. This structure would have to be experimentally probed in order to verify that the query performs the same function as other family members.

However, in contrast to simple SCFGs, we do not know an efficient method to compute this semantics for family model grammars. Ambiguity compensation, as shown above, suffers from a runtime complexity dependent on the number of structures, which in turn grows exponentially with the sequence length. Efficient computation of the structure semantics is an interesting open challenge, where one must be aware that a polynomial time, exact algorithm may not exist. An ideal modeling tool would allow the user to specify the intended semantics, either at model generation time or when starting a search.

Smaller and faster models: The smaller size and better speed of models derived from a small grammar such as $G5$ deserves further study. Its use may have been discouraged by the diagnosis of the "abysmal" performance of $G5$ reported in [5]. Dowell and Eddy explain this performance by the overloading of rules:

> "The compact grammar $G5$, for instance, must invoke the same bifurcation rule $S \to aS\hat{a}S$ for every base pair and for every structural bifurcation, which are quite different structural features that occur with very different frequencies. The productions of $G5$ are thus "semantically overloaded": they collapse too many different types of information into the same parameters."

This explanation, appropriate as it is for simple SCFGs, also points to a remedy for the case of family model grammars. These grammars have position-specialized productions, and unless we tie parameters together irrespective of their structural position in the model, we can still train different and adequate parameters for different features. This requires careful engineering and empirical testing, but small grammars are still in the race. Note also that filtering techniques, which have been developed to speed up the present *Infernal*-generated models, can also be adapted to models generated from a different prototype grammar.

Comparing the performance of different prototype grammars: Dowell and Eddy diagnosed superior performance of another unambiguous SCFG ($G6$ which stems from Pfold [13]). However, this grammar was not tested as the prototype for model grammar generation. Given our compact algorithm of model generation – the generator from $G5$ is but 164 lines of ADP code – it maybe a justifiable effort to extend the Dowell and Eddy study to different model generators, training family models rather than simple SCFGs. We conjecture that our proof of a correct implementation of the trace (or the alignment) semantics could be adapted for a new family model generator, as long as an unambiguous prototype grammar is used. If not, there is still our ambiguity checking pipeline, which can be used to show correctness of the individual models after their generation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS

ACKNOWLEDGMENTS

We acknowledge inspring discussions we had with many participants of the 2009 Benasque Workshop on Computational Methods for RNA Analysis, organized by E. Rivas and E. Westhof, where these ideas were first presented in public. In particular, Sean Eddy and Eric Nawrocki provided insight into model construction in *Infernal* and the underlying rationale. The authors thank Peter Steffen for implementing *adp2cfg* and Stefan Janssen for a careful reading and valuable comments. We also thank Ivo Hofacker for comments, help and support. This work has been funded, in part, by the Austrian GEN-AU project bioinformatics integration network III.

APPENDIX

Rfam models tested for non-ambiguity

15 smallest models from Rfam without a bifurcation:

```
RF00032.cm   *****<<<<<<****>>>>>>*****
RF00037.cm   <<<<<***<<<<<******>>>>>*>>>>>
RF00180.cm   <<*<<<<******>>>>>>******************
RF00390.cm   ********<<<<*******>>>>
RF00453.cm   <<<<<<<***<<<**********>>>>>>>>>>
RF00502.cm   ****<<<<<<<<********>*>>>>>>>*****
RF01072.cm   <<<<**********>>>>*************
RF01080.cm   *******<<<<<<<***********>>>>>>>
RF01081.cm   <<<<<********>>>>>*********
RF01082.cm   *****<<<<<<<*******>>>>>>>
RF01112.cm   *****<<<<<******>>>>>
RF01115.cm   <<<*******>>>*********
RF01380.cm   <<<<<<<****>>>*>>>>
RF01381.cm   ***<<<<<****>>>>>******
RF01382.cm   **<<<<<****>>>>>****
```

15 smallest models from Rfam with a bifurcation:

```
RF00008.cm   *<<<<<<*<<<<<***>>>>>*******<<<<********>>>>***>>>>>>*
RF00057.cm   ******<<<****<<********>*>***>>>********<<<<<<<******>>>>>>>*******
RF00161.cm   <<<<**<<<<<<<<*****>>>>>>>>***<<<<*<<<<<<<*********>>>>>>*>>>>>**>>>>
RF00163.cm   <<<<<<********<<<<*******>>>>***<<<<>>>>*>>>>>>
RF00184.cm   *********<<<************>>>*******<<<<***************>>>>
RF00192.cm   <<<<<<**********>>>>>>**********<<<<<<*************>>>>>>
RF00384.cm   *<<<<<<<******>>>>>>>***<<<<<*<<*<<<******>>>***>>>>>>>*********
RF00454.cm   ********<<<<*<<*****>**>>>>**<<<<*<<<***>>>*>*>>>>********
RF00517.cm   **********<<<<<<*******>>>>>>***<<<<<<****>>>>>>*****
RF00630.cm   **************<<<<***<*****>***>>>>**<<<<<<<<<<******>>>>>>>>>>*
RF00681.cm   <<<<<**<<<<<<<<<<<*<<<*****>>>*<<*****>*>>>>>>>>>>>**>****>>>>>
RF01068.cm   <<<<<<<<*****>>>>>>>>***************<<<<<*****>>>>>
RF01116.cm   <<<<<<<<****>>>>>>>>*******************<<<<<<*****>>>>>>**
RF01388.cm   *<<******<<<<*****>>>>>*****><<<<<<<<*******>>>>>>>>*******
RF01403.cm   <<**<<<<<<<<<<**********>>>>>>>>>****>>>*****<<<<<<<<*******>>>>>>>>*****
```

REFERENCES

[1] P Baldi and S Brunak. *Bioinformatics – the machine learning approach*. MIT Press, 1998.
[2] A Bateman, E Birney, L Cerruti, R Durbin, L Etwiller, S R Eddy, S Griffiths-Jones, K L Howe, M Marshall, and E L Sonnhammer. The Pfam protein families database. *Nucleic Acids Res*, 30(1):276–280, Jan 2002.
[3] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing Ambiguity of Context-Free Grammars. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, volume 4783 of *LNCS*. Springer-Verlag, July 2007. Extended version submitted for journal publication.
[4] Broňa Brejová, Daniel G. Brown, and Tomáš Vinař. The most probable annotation problem in HMMs and its application to bioinformatics. *Journal of Computer and System Sciences*, 73(7):1060–1077, March 2007.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

[5] R D Dowell and S R Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 5:71–71, Jun 2004.
[6] R Durbin, S Eddy, A Krogh, and G Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 2006 edition, 1998.
[7] S R Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.
[8] Sean R. Eddy and Richard Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Res*, 22(11):2079–2088, June 1994.
[9] P P Gardner, J Daub, J G Tate, E P Nawrocki, D L Kolbe, S Lindgreen, A C Wilkinson, R D Finn, S Griffiths-Jones, S R Eddy, and A Bateman. Rfam: updates to the RNA families database. *Nucleic Acids Res*, 37(Database issue):136–140, Jan 2009.
[10] R. Giegerich. Explaining and Controlling Ambiguity in Dynamic Programming. In *Proc. Combinatorial Pattern Matching*, volume 1848 of *Springer Lecture Notes in Computer Science*, pages 46–59. Springer, 2000.
[11] R. Giegerich, C. Meyer, and P. Steffen. A Discipline of Dynamic Programming over Sequence Data. *Science of Computer Programming*, 51(3):215–263, 2004.
[12] J E Hopcroft and J D Ullman. *Formal languages and their relation to automata*. Addison-Wesely, 1969.
[13] B Knudsen and J Hein. Pfold: RNA secondary structure prediction using stochastic context-free grammars. *Nucleic Acids Res*, 31(13):3423–3428, Jul 2003.
[14] Sean Eddy Lab. *INFERNAL User's Guide. Sequence analysis using profiles of RNA secondary structure*, version 1.0 edition, January 2009. http.//infernal.janelia.org.
[15] E P Nawrocki, D L Kolbe, and S R Eddy. Infernal 1.0: inference of RNA alignments. *Bioinformatics*, 25(10):1335–1337, Mar 2009.
[16] Janina Reeder, Peter Steffen, and Robert Giegerich. Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, 6:153, 2005.
[17] Eddy S. Hmmer user's guide. Technical report, Howard Hughes Medical Institute, 2003.
[18] Y Sakakibara, M Brown, R Hughey, I S Mian, K Sjölander, R C Underwood, and D Haussler. Stochastic context-free grammars for tRNA modeling. *Nucleic Acids Res*, 22(23):5112–5120, Nov 1994.
[19] D Sankoff and Kruskal J. *Time warps, string edits, and macromolecules*. Addison-Wesley, 1983.
[20] Peter Steffen and Robert Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(1):224, September 2005.
[21] B Voss, R Giegerich, and M Rehmsmeier. Complete probabilistic analysis of RNA shapes. *BMC Biol*, 4:5–5, 2006.
[22] M Waterman. *Introduction to computational biology*. Chapman & Hall, 1994.

Chapter 6

Discriminatory Power of RNA Family Models

Christian Höner zu Siederdissen and Ivo L. Hofacker.
Discriminatory Power of RNA family models.
Bioinformatics. 2010. 26 (18). i453–i459.

CHzS designed the study and implemented the algorithm available as `CMCompare`. Both authors participated in writing the paper.

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/2.5), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

BIOINFORMATICS

Vol. 26 ECCB 2010, pages i453–i459
doi:10.1093/bioinformatics/btq370

Discriminatory power of RNA family models

Christian Höner zu Siederdissen* and Ivo L. Hofacker

Institute for Theoretical Chemistry, University of Vienna, Währinger Strasse 17, A-1090 Wien, Austria

ABSTRACT

Motivation: RNA family models group nucleotide sequences that share a common biological function. These models can be used to find new sequences belonging to the same family. To succeed in this task, a model needs to exhibit high sensitivity as well as high specificity. As model construction is guided by a manual process, a number of problems can occur, such as the introduction of more than one model for the same family or poorly constructed models. We explore the Rfam database to discover such problems.

Results: Our main contribution is in the definition of the discriminatory power of RNA family models, together with a first algorithm for its computation. In addition, we present calculations across the whole Rfam database that show several families lacking high specificity when compared to other families. We give a list of these clusters of families and provide a tentative explanation. Our program can be used to: (i) make sure that new models are not equivalent to any model already present in the database; and (ii) new models are not simply submodels of existing families.

Availability: www.tbi.univie.ac.at/software/cmcompare/. The code is licensed under the GPLv3. Results for the whole Rfam database and supporting scripts are available together with the software.

Contact: choener@tbi.univie.ac.at

1 INTRODUCTION

Structured non-coding RNAs are nucleotide sequences that are not translated into protein but have, in the folded state, their own specific functions (Mattick and Makunin, 2006). This function is very much dependent on the secondary and tertiary structure (the folded state), while on the other hand, the primary structure or sequence sees more change (Mattick and Makunin, 2006) in the form of mutation.

One can define relationships between non-coding RNAs in different species. A set of related sequences is called an RNA family. Each set is defined by its members performing the same function in different species. When genomes are sequenced, one is interested in finding members of known families in the new data, as well as finding new families if previously unknown non-coding RNAs are discovered.

The problem—finding homologues—exists for proteins, too. Software to perform the same kind of searches exists in the form of HMMer (Eddy, 1998) and the Pfam (Bateman *et al.*, 2002) database. Using profile hidden Markov models (profile HMMs), a mathematically convenient solution was found, around which the algorithms could be built. Unfortunately, the same solution proved inadequate (Durbin *et al.*, 1998, Chapter 10.3) for non-coding RNAs.

The task of building a model that describes a new family is still a mostly manual process. Finding new members of existing families, on the other hand, can be performed using software. The problem we are discussing in this article applies equally well to other algorithms to search for homologue sequences, but as our algorithm is specific toward an existing software package, namely Infernal (Nawrocki *et al.*, 2009a), we will perform our analysis with respect to this software and the corresponding Rfam (Griffiths-Jones *et al.*, 2003) database.

```
human     acgucg aacuaga
cow       accugg aacuaga
dog       acuugg aag uca
cat       acgucgaaacuaga
structure *<<*>>.**<**>*
```

Fig. 1. Multiple alignment of sequences from several species and the consensus structure. Brackets denote nucleotide pairings, a star denotes a consensus unpaired nucleotide and a dot a nucleotide not in the consensus.

In order to model families of non-coding RNAs in a way that provides both sensitivity and specificity, the consensus secondary structure of the set of sequences has to be included in the mathematical model from which the algorithm is created. Stochastic context-free grammars provide access to such models, just as stochastic regular grammars (in the form of profile HMMs) can be used to model protein families.

We begin with a succinct introduction to the process of first designing an Infernal RNA family model and then searching for new family members. Building on those algorithms, we can define the specificity of a given model compared to other known models in a natural way using the already established Infernal language of bit scores.

1.1 Infernal model design

Infernal is based on *covariance models* (Eddy and Durbin, 1994). We assume that a structure-annotated multiple alignment of the family sequences like the one in Figure 1 is at hand. Intuitively, new family members should (i) align well and (ii) show the same secondary structure. The more a sequence deviates from these two requirements the worse it should score.

During the covariance model construction process, a so-called guide tree is derived from the structure annotation. The nodes of the tree fall into six classes: (1) a pair-matching (P) node for a base pair; (2–3) two kinds of single nucleotide nodes, one left- (L) and one right-matching (R); (4) a bifurcation (B) node to allow for multiple external and internal loops; and two house-keeping nodes: (5) a start-node (S) and (6) an end-node (E). Whenever possible, left-matching nodes are used, e.g. in hairpin loops, delegating right-matching nodes to be used only where necessary, such as the right side of the last external loop. This removes ambiguity from the construction process. The alignment from Figure 1 leads to the model depicted in Figure 2.

Mutations in base pairs or single conserved nucleotides are handled in the conventional stochastic RNA modeling approach by

*To whom correspondence should be addressed.

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

© The Author(s) 2010. Published by Oxford University Press.
This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/2.5), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

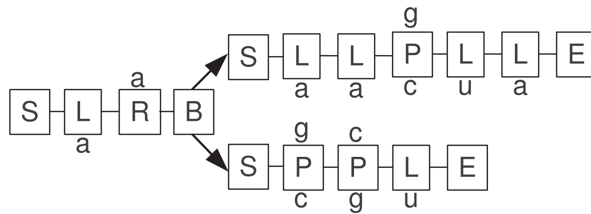

Fig. 2. A covariance model, displaying the six types of nodes needed for construction. The nucleotide annotation follows the human sequence of the multiple alignment of Figure 1.

keeping emission probabilities (or log-odd scores) for each possible base or pair for the emitting nodes (P, L, R).

The way Infernal works, insertion of additional nucleotides or deletion of parts of the consensus sequence cannot be handled by the matching nodes alone. For the final model, each node is replaced by a number of states. One state acts as the main state, that is, for example, each pair (P) node has a pair state matching both a left and a right nucleotide. The deletion of one of the two nucleotides is handled by adding two states, one only left- (L), one only right-matching (R). A fourth state (D) handles the deletion of both nucleotides while two inserting states (IL, IR) are used for insertions relative to the consensus. Transitions from one state to the next happen with some probability which is close to 1.0 for the consensus state and far less likely for the other possible states. The exact numbers are calculated by fitting probability distributions using the multiple alignment data.

Nodes matching only a single nucleotide are extended with a deletion state and either a left- or a right-inserting state, depending on the main state. A bifurcation (B) leads directly to two new start (S) nodes, effectively to two complete submodels. By arbitrary selection, the right start node is extended with a left-inserting state to allow for insertions between a bifurcation.

Mostly, however, it is enough to keep the picture of the model (Fig. 2), using only matching nodes, in mind.

With the additional states the model is completed. The fitting of the probability distributions given the nucleotide consensus data is outside of the scope of this text and we refer the reader to the book on the subject matter by Durbin *et al.* (1998).

1.2 Searching with covariance models

The complete model is a graphical representation of the stochastic context-free grammar that does the real work. A pair state (P), for example leads to a total of 16 productions of the form $P_k \rightarrow aQb$, where P_k is the k'th node to be processed, Q abstracts over the possible targets states, which depend on the node $k+1$ and (a,b) are the 16 possible nucleotide pairs. The whole process leads to CFGs with a huge number of productions (in the order of the number of nodes times a small constant), especially when compared with single RNA folding grammars (Dowell and Eddy, 2004), that have in the order of 10–100 productions.

The actual search process uses the CYK algorithm (newer versions of Infernal use the Inside algorithm to calculate the final score) to find the best parse of an input string given the model. Input strings are all substrings of a genome up to a given length. Using dynamic programming, this approach is fast enough that whole genomes can be processed in a matter of hours or days.

Our interest in this article is not the search process of Infernal, but how a parse is scored and the best alignment of string against model is selected.

NOTATION. *Given an alphabet $\mathcal{A}$, $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$. Let $s \in \mathcal{A}^*$ be a, possibly empty, input string.*

NOTATION. *Let m, m_1, m_2 be covariance models in the form of stochastic context-free grammars conforming to the Infernal definition.*

Given a model m and an input string s, the CYK score can be calculated over all parses P of the string s by the model m:

$$\begin{aligned}&\text{CYK}(m,s) = \\ &\max\{\text{Score}(P(m,s)) | P(m,s) \text{ is successful}\}.\end{aligned} \tag{1}$$

A successful parse is a parse that consumes the complete input s and finishes in terminal end states. During such a parse a score is built up from the transition and emission scores that were calculated for each model during its construction.

Several methods exist to perform the calculations. Arguably, closer to Equation (1) is the use of tree grammars and algebras in Giegerich and Höner zu Siederdissen (2010), but Infernal uses traditional dynamic programming to implement the CYK algorithm. Whichever method is used, they are more efficient than the enumeration of all possible parses. Finally, the alignment of the input against the model can be retrieved using backtracking or other methods.

2 METHOD

A covariance model with high specificity assigns low bit scores to *all* sequences that do not belong to the model family. Finding sequences that lead to false positives, that is having a high score while not belonging to the family, is a problem. We take a view that does not look at a single model, but rather at two models at the same time. Then, we can say that:

A covariance model has low specificity with respect to another model if there exists a sequence $s \in \mathcal{A}^*$ that achieves a high CYK *score in both models.*

We acknowledge that 'high score' is not well-defined, but consider what constitutes a high score in Infernal. Hits in Infernal come as a tuple, the score itself and an *e*-value. One is typically interested in scores of 20 bit or higher and *e*-values of 1.0 or less, depending on the model. The *e*-value is dependent on the genome size, but given such guidelines one finds good candidates. In light of this, the meaning of 'high score' becomes more clear. As we use the same measure as Infernal, a string that achieves, say, 40 bit in two different models points to low specificity, as the string would be considered a good hit when searching for new family members with both models separately.

Using the previous definition, we find an analog to Equation (1) to calculate (i) the highest score achievable by (ii) a single input string:

$$\begin{aligned}&\text{Link}(m_1,m_2) = \text{MaxiMin}(m_1,m_2) = \\ &\text{argmax}_s\{\min\{\text{CYK}(m_1,s), \quad \text{CYK}(m_2,s)\} | s \in \mathcal{A}^*\}.\end{aligned} \tag{2}$$

Here, m_1 and m_2 are two different covariance models. MaxiMin returns the highest scoring string. The highest score is defined as the

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

Table 1. Recursive calculation of the maximal score achieved by an input string common to both model m_1 and m_2

$$\text{minP} \quad (a,b)=\begin{cases} a & a<b \\ b & \text{otherwise} \end{cases} \tag{3}$$

$$\text{maxmin} \quad x=\text{argmax}\left\{\text{minP}(s_1,s_2)|(s_1,s_2)\in x\right\} \tag{4}$$

$$\text{MaxiMin} \quad (k_1,k_2)=\begin{cases}
(0,0) & k_1=\text{E}\wedge k_2=\text{E} \\
\text{maxmin}\{\text{MaxiMin}(k_1',k_2')+(e_{k_1,a,b},e_{k_2,a,b})+(t_{k_1\to k_1'},t_{k_2\to k_2'}) & \\
\quad | \quad k_1'\in c_{k_1},k_2'\in c_{k_2},a\in\mathcal{A},b\in\mathcal{A}\} & k_1=\text{P}\wedge k_2=\text{P} \\
\text{maxmin}\{\text{MaxiMin}(k_1',k_2')+(e_{k_1,a},e_{k_2,a})+(t_{k_1\to k_1'},t_{k_2\to k_2'}) & \\
\quad | \quad k_1'\in c_{k_1},k_2'\in c_{k_2},a\in\mathcal{A}\} & k_1\in\{\text{L,IL}\}\wedge k_2\in\{\text{L,IL}\} \\
\text{maxmin}\{\text{MaxiMin}(k_1',k_2')+(e_{k_1,b},e_{k_2,b})+(t_{k_1\to k_1'},t_{k_2\to k_2'}) & \\
\quad | \quad k_1'\in c_{k_1},k_2'\in c_{k_2},b\in\mathcal{A}\} & k_1\in\{\text{R,IR}\}\wedge k_2\in\{\text{R,IR}\} \\
\text{maxmin}\{\text{MaxiMin}(k_1,k_2')+(0,t_{k_2\to k_2'}) & \\
\quad | \quad k_2'\in c_{k_2}\} & k_1=\text{E}\wedge k_2\in\{\text{D,S}\} \\
\text{maxmin}\{\text{MaxiMin}(k_1',k_2)+(t_{k_1\to k_1'},0) & \\
\quad | \quad k_1'\in c_{k_1}\} & k_1\in\{\text{D,S}\}\wedge k_2=\text{E} \\
\text{maxmin}\{\{\text{MaxiMin}(k_{1,1}',k_{2,1}')+\text{MaxiMin}(k_{1,2}',k_{2,2}') & \\
\quad | \quad \{k_{1,1}',k_{1,2}'\}=c_{k_1},\{k_{2,1}',k_{2,2}'\}=c_{k_2}\} \quad \cup & \\
\quad\quad \{\text{MaxiMin}(k_{1,2}',k_{2,1}')+\text{MaxiMin}(k_{1,1}',\text{E})+\text{MaxiMin}(\text{E},k_{2,2}') & \\
\quad | \quad \{k_{1,1}',k_{1,2}'\}=c_{k_1},\{k_{2,1}',k_{2,2}'\}=c_{k_2}\} \quad \cup & \\
\quad\quad \{\text{MaxiMin}(k_{1,1}',k_{2,2}')+\text{MaxiMin}(k_{1,2}',\text{E})+\text{MaxiMin}(\text{E},k_{2,1}') & \\
\quad | \quad \{k_{1,1}',k_{1,2}'\}=c_{k_1},\{k_{2,1}',k_{2,2}'\}=c_{k_2}\}\} & k_1=\text{B}\wedge k_2=\text{B} \\
\text{maxmin}\{\text{MaxiMin}(k_{1,1}',k_2)+\text{MaxiMin}(k_{1,2}',\text{E}) & \\
\quad | \quad \{k_{1,1}',k_{1,2}'\}=c_{k_1}\} & k_1=\text{B}\wedge k_2\neq\text{B} \\
\text{maxmin}\{\text{MaxiMin}(k_1',k_2')+(t_{k_1\to k_1'},t_{k_2\to k_2'}) & \\
\quad | \quad k_1'\in c_{k_1},k_2'\in c_{k_2}\} & (k_1,k_2)\in\{(\text{S,S}),(\text{D,D})\} \\
(-\infty,-\infty) & \text{otherwise}
\end{cases} \tag{5}$$

We abuse notation quite a bit to reduce notational clutter. The state type of model 1 at index k would be $\text{type}^1_{k_1}$ but we write $k_1=\text{E}$ to determine if the state is an end state. Additional data structures are simplified as well. The states into which a transition is possible (the children of state k) are written c_{k_1} instead of $c^1_{k_1}$. Emission scores for each model are in the matrix e which is indexed by the state k and the nucleotide(s) of the emitting state. Transition scores for transition from state k to k' are found in the matrix t. The case where $k_1=\text{E}\wedge k_2=\text{E}$ terminates the recursion, as each correctly built covariance model terminates (each submodel) with an end-state (E) (cf. Fig. 2). Addition of pairs happens element-wise: $(a,b)+(c,d)=(a+b,c+d)$.

minimum of the two CYK scores. This guarantees that both models score high. Variants of the algorithm are possible, for example MaxPlus which sums both scores before maximizing. However, MaxiMin provides better results in case one of the two models contains many more nodes than the other. More importantly, it provides a score which would actually be achieved during a search using one of the two models, while the other would score even higher. As the sequence s 'links' both models via their discriminative power, we shall use the term Link from now on.

The trivial implementation suggested by Equation (2) is not well-suited for implementation as it requires exponential runtime due to the enumeration of all possible strings in $\mathcal{A}^*$.

In order to find the highest scoring string, we perform a kind of tree alignment with additional sequence information. The tree alignment part optimizes the structure of each model, while sequence alignment is performed for nucleotide emitting states as well. Both alignments are tightly coupled as is the case for covariance models themselves. A pair state (P), for example, leads to another structure than a left-emitting (L) state. This also explains why we have to deal with a small restriction in our algorithm. The tree alignment requires us to align each state with at most one other state, but not two or more. After an explanation of the implementation, we discuss this further.

Implementation: We present a simplified version of our recursive algorithm in Table 1. To set the field, we need two additional functions. Equation (3) defines the minimum of a pair of values in a natural way. The function maxmin [Equation (4)] is a small helper function selecting the maximal pair, where the maximum of two pairs is defined by the maximum of individual minima, hence the name max-min.

The recursion has to be performed simultaneously over both models. For model m_1 we have index k_1 and for model m_2, k_2 will be used. Note though, that by following just one of the elements of

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

the tuples, the CYK algorithm can be recovered. We are, in essence, performing two coupled CYK calculations at the same time.

Internally, all states are kept in an array. The first index is guaranteed to be a start state (S) and the last index to be an end state (E). The first state is the root state of the whole model, too. Three additional arrays are required.

The states that can be reached from a state are stored in an array named c for children. Because indices from one model are never used in the other model, we can always write c_{k_1} instead of $c^1_{k_1}$.

We use the same simplification for emission scores. The array e holds such scores. It is indexed with the nucleotides that are to be emitted. This is to be written as $e_{k_1,a,b}$ for pairs and either a or b are missing for single nucleotide emitting states.

The third required array, t, stores transition scores. Whenever the recursion descends from a state k_1 into a possible child state k_1', a lookup $t_{k_1 \to k_1'}$ is performed. Not all transitions incur a cost. A branch into the two child states always happens with probability 1.0.

We have abused notation to simplify the recursion a bit. The determination of the type of the current state requires an additional data structure to perform the lookup for the indices k. Instead of writing X_{k_1}, where X is such a data structure, we just write $k_1 = \mathrm{E}$ to assess if state k_1 happens to be an end (E) state.

Some of the cases found in the source code have been removed for clarity. Most cases deal with symmetric states. The last state to visit is, for example (E, E). This initializes the CYK score to (0.0, 0.0). The case (MP, MP) handles the emission of a pair of nucleotides. There are some cases like (S, x), where x is any state except (S), that require special handling. These special cases ((E, D) and (E, S) are given as an example) do not contribute any information on how one goes about calculating the common score, but simply make a large recursion more unwieldy.

The algorithm is asymptotically fast. Given the number of states n_1 and n_2 of the two models, each pair of states will be visited once at most. In addition, the number of children c_{k_1} and c_{k_2} per state is fixed by a constant. If h denotes the maximal number of children per state, the total runtime is bounded by $O(n_1 n_2 h^2)$.

A restriction in the implementation: Consider the structure annotation of two different covariance models: `ma`: < < > > and `mb`: *< >*. Model m_a has two nodes $\mathrm{P}^a_1 - \mathrm{P}^a_2$ and model m_b three nodes: $\mathrm{L}^b_1 - \mathrm{R}^b_2 - \mathrm{P}^b_3$. An input string like `ccgg` is likely to result in a good score for both models, especially if we assume that the family sequences are similar to `ccgg`. Equation (2) would return that result after some time. For a fast implementation, those two models are rather inconvenient as P^a_2 has to be matched against both L^b_1 and R^b_2 at the same time. By allowing to match only one state against one other state, our algorithm produces suboptimal scores in such cases. Fortunately, this is a minor problem for real models. This can be explained by the relative scarcity of such cases and the regularity of the covariance model building process. If left-matching and right-matching nodes could be used at will, e.g. in hairpin loops, our simplification would have more than minor consequences.

Local and global scoring: Infernal does not require that a sequence matches the whole model. Instead, a *local* search is performed. Each string is aligned against the part of the model where it scores best. Should this require the deletion of parts of the model, this does not invoke many delete (D) states. One can simply do a transition into a local start or end state. These transitions are possible only with small probability (typically around 0.05 divided by the number of nodes in the model) but this still gives higher scores than potentially having to descend into dozens of delete states.

Since Infernal scores locally with respect to the model, we do the same by default. Details of the implementation are omitted. Using the `-global` switch, this behaviour can be changed. In that case, both models have to be aligned and the resulting string will be optimal with respect to the whole model, not just some submodel. Several other switches known from Infernal are available, too.

Just one string?: Of course if only a single string has a good score in both models, the problem would be moot as the probability to encounter that exact string is close to zero. But consider that from the pairwise score and the corresponding string, suboptimal strings can be generated easily. Given the length k of the string s, then k points for substitutions give $3k$ strings that score almost as high. A further $3\binom{k}{2}$ strings score less, and so forth with 3 and more substitutions. Furthermore, insertions and deletions are possible.

This means that whenever there is one high-scoring string, there will be many more, we just present the worst case.

3 RESULTS

The Rfam 9.1 database contains 1372 different models. All pairwise calculations lead to a total of 940 506 results. The time to calculate the score and string for each pair is typically less than one second, but of course depending on the size of the models in question. Of all pairs, about 70 000 are noteworthy with scores of 20 bit or more. Figure 3 shows the distribution of scores among all pairs of family models. Negative scores have been truncated towards zero as any score lower than this certainly means that the two models in question are separated very well.

Among the high-scoring pairs are several interesting examples, some of which we will take a closer look at. Similar results for other models can be extracted from the data available for download. It is possible to generate, among others, model-centric views that show the high-scoring neighborhood of a particular model and global views that show high-scoring pairs. As Figure 4 aptly demonstrates, clusters of families form early (in this case, only the 20 highest scoring edges are drawn).

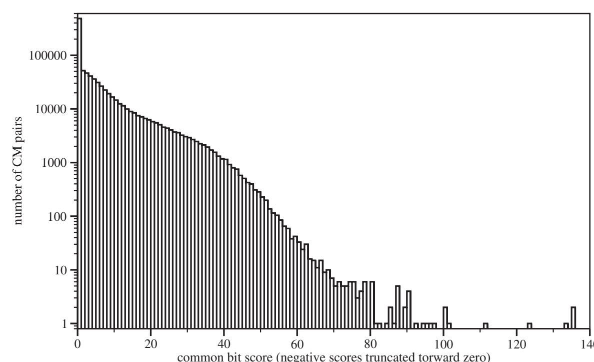

Fig. 3. Distribution of bit scores for all 940 506 pairs of covariance models. About 70 000 pairs have scores of 20 bit or more, pointing towards weak separation between the two models.

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

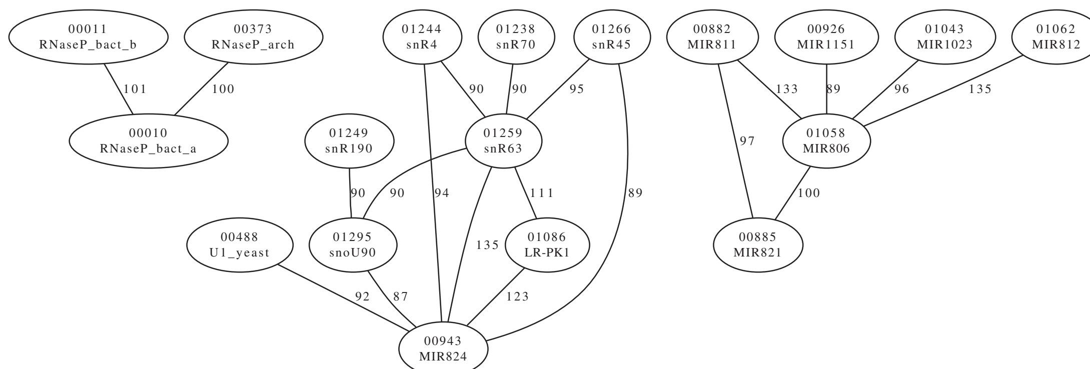

Fig. 4. The 20 highest scoring edges between RNA families. Each edge represents a string that, between the connected nodes, results in a bit score at least as high as the given value. The two connected family models have low discrimination in such a case. For each family model the Rfam index and name are shown.

Table 2. Occurrence of shapes in results with at least 20 bit each

m_1	_	[]	[][]	[][][]	[[][]]	Complex
m_2	_	[]	[][]	[][][]	[[][]]	Complex
Found	19 644	49 289	1576	40	12	28

Unstructured regions (dashes) and hairpins (square brackets) as the common region occur most often. The other shapes show that complex substructures can form. The high number of lone hairpin structures is a direct consequence of the huge meta-family of snoRNAs which have a simple secondary structure. Under 'complex', all structures that did not fit into the given shapes were collected.

In Table 2, we have gathered some results. The 70 000 pair scores over 20 bit have been split according to the abstract shape of the secondary structures of the hit. A shape (Reeder and Giegerich, 2005) is a representation of the secondary structure that abstracts over stem and interior loop sizes. In this case, each pair of brackets defines one stem. Intervening unpaired nucleotides do not lead to the creation of a new stem. Hits such as _/_ are unstructured, but similar, sequences. The shape []/[] is just one hairpin, while the two shapes [[][]]/[[][]] on the same string point to an interesting pair score as the string apparently folds into complex high-scoring structures that align well, too.

In principle, it is possible that the common sequence folds into two different secondary structures. At abstract shape level 5 (the most abstract) this did not happen for the current Rfam database. Our algorithm, however, is capable to deal with such cases.

Let us now take a closer look at two examples that are particularly interesting. The first was selected because RNaseP is a ubiquitous endoribonuclease and the second to highlight how problematic models can be discovered.

1st example: The RNaseP families for bacteria (type a and b) and archaea show weak separation as can be seen in Figure 5. The three involved models (Rfam id 10, 11 and 373) have different noise cutoff scores. The noise cutoff is the highest score for a false hit in the Rfam NR database, scores above this threshold are likely homologues (cf. Nawrocki *et al.*, 2009b). For the three different RNaseP families, these scores are 43, 93 and 59 bit, respectively. A look at Figure 5 shows, that no random sequence could score high in both model 373 and 11, one can, at most, find a hit that is remote at best. The picture is entirely different for the high-scoring sequence between RNaseP,

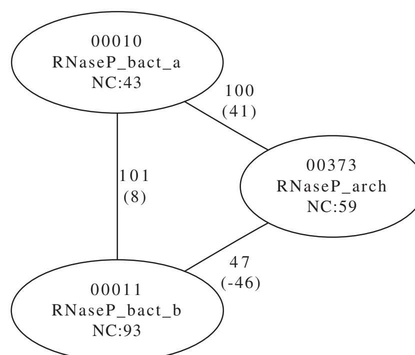

Fig. 5. Link scores for different RNaseP models (with noise cutoff (NC)) with weak separation. Values in brackets are the difference to the noise cutoff thresholds. The difference is as at least as high as given. A negative value means that in one or two of the models, the score was lower than the noise cutoff. For example, the Link score of 101 bit between `bact_a` and `bact_b` is 8 bit higher than the NC of `bact_b`.

type a and RNaseP in archaea. Here, we find a sequence that is at least 41 bit higher than the noise cutoff. A similar picture presents itself for the sequence found for the two bacterial RNaseP models, though the score difference between the noise cutoff and the highest score is only 8 bit.

The sequences and their scores show something else, too. In Section 2, we described how to generate many similar strings from the one returned string. In this case, where the gap between cutoff and score is as wide as 41 bit, we could indeed create a very large number of strings. Each of which with a score that makes it a likely hit.

Additionally, the high scores between the three RNaseP models are somehow expected, given that all three models describe variants of RNaseP. Nuclear RNaseP (not shown), on the other hand, is well separated from these three models with a maximal score of 24 bit.

2nd example: For our second example (Fig. 6), we have chosen a set of four family models. Each presents with not only a high Link score with regard to the others but also the scores are over the noise cutoff threshold by a large margin, too.

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

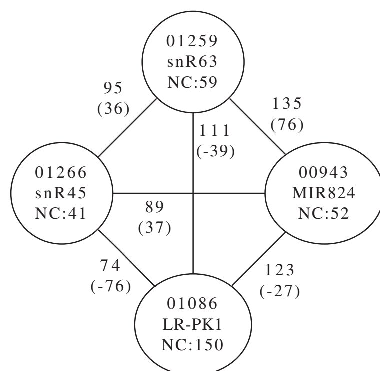

Fig. 6. A high-scoring set of families, explicitly selected for the large difference to the noise cutoff value. Models 1259 and 943 score 135 bit on some input, which is at least 76 bit higher than the respective noise cutoff value. Notice, too, that not all pairs show such a behavior. Models 1086 and 943 have a high Link score with 123 bit, but at least the noise cutoff value is higher than this value (by 26 bit), making a hit less likely in one model. Some of the models were built using very few seed sequences and this seems to increase the chance of finding weak models.

These models show that high noise cutoff values are not necessarily enough. On the one hand there are indeed some 28 500 edges between families where the Link score is higher than both threshold values. In these cases one would reasonably argue to have found a homologue, even though the chance for a false positive does exist. One cannot, on the other hand, simply set the noise threshold to safe levels. This is because interesting sequences in the form of distant family members are likely to be found above the current noise threshold values.

The examples chosen for Figure 6 point out another problem with some of the models in the Rfam database. Models like RF00943 were created using only two seed sequences and five sequences in total. This is, of course, not a problem of Infernal but one of biological origin. As long as more members of the class have not been identified, the resulting models are a bit sketchy.

4 DISCUSSION

We have presented a polynomial-time algorithm that, for any two covariance models, returns a string that scores high in both models. Using this algorithm, several questions regarding RNA family models can be answered.

First, it is possible to determine if a model has high discriminative power against other models. This is important to avoid false positive results when searching for previously unknown new family members. The discriminative power can be quantified using the same measure as used in Infernal itself, thereby giving answers in a language, namely bit scores, that makes comparisons possible and easy.

Second, if a model shows overlap with another, it can be determined which regions of the model do actually show this behaviour. This is possible, as we not only return a score value, but other information, too. This includes the offending string, the respective secondary structures and a detailed score account.

Third, the algorithm is extendable. Borrowing ideas from Algebraic Dynamic Programming (Giegerich and Meyer, 2002), an optimization algebra can be anything that follows the dynamic programming constraints. Included are the CYK scoring algebra and the different information functions as well as an algebra product operation. Additional algebras require roughly a dozen lines of code.

Fourth, the MaxiMin, or Link score lends itself as a natural similarity score for RNA families. Closely related families, in terms of primary and secondary structure—not necessarily biological closeness, show a higher Link score than others. This requires further investigation to determine how much biological information can be extracted. Pure mathematics cannot answer which biological relation does actually exist.

In the case of prospective meta-families, we have two open research problems. One is to take a closer look at high-scoring families to determine their biological relationship. Are high scores an artifact of poorly designed families, or a case of an actual meta-family? The other problem became evident in the 1st example, where not all members of the RNaseP family scored high against each other. This suggests that meta-families cannot be modeled in Infernal directly, but how to adapt RNA family models in such a case remains open.

Researchers designing new families will also find value in the tool, as one can scan a new family model against existing ones to be more confident that one has indeed identified a new family and not an already existing one in disguise.

The Infernal Users Guide (Nawrocki *et al.*, 2009b) mentions homology between family models as a reason for the existence of the different cutoff scores for noise, gathering and trusted. We think it is important to be able to determine, computationally, the importance of the cutoff scores when assigning new hits to families.

Another fact is that cutoff scores, like the models themselves, are set by the curators of the family. Our scoring scheme relies on the Infernal scoring algorithm itself. As numbers of models were created from very few seed sequences it is possible that the relevant cutoff scores are set too high to capture remote members. A cutoff score above the highest pair scores involving such a model could be of help while scanning new genomes for remote family members.

Finally, we have to acknowledge that Infernal uses the Inside-, not the CYK-algorithm to determine final scores. This can pose a problem in certain exceptional circumstances but these should be rare. Mathematically (cf. Nawrocki *et al.*, 2009b), $\text{CYK} = \text{Prob}(s,\pi|m)$, while $\text{Inside} = \text{Prob}(s|m)$. The CYK algorithm gives the score for the single best alignment π of sequence s and model m while the Inside algorithm sums up over all possible alignments. This just means that we underestimate the final score, or said otherwise, the Inside scores for the Link sequence given the corresponding models will be even higher than the CYK scores.

Curated thresholds and Infernal 1.0: The version change to Infernal 1.0 requires re-examination of all threshold values (cf. `infernal.janelia.org`). The next release of the Rfam database is expected to have done this, meaning that a comparison between (new) cutoff values and the scores calculated here is of current interest.

ACKNOWLEDGEMENTS

The authors thank Ronny Lorenz for proofreading the article.

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

Funding: The Austrian GEN-AU project bioinformatics integration network III.

Conflict of Interest: none declared.

REFERENCES

Bateman,A. *et al*. (2002) The Pfam protein families database. *Nucleic Acids Res.*, **30**, 276–280.

Dowell,R. and Eddy,S. (2004) Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, **5**, 71.

Durbin,R. *et al*. (1998) *Biological Sequence Analysis*. Cambridge University Press, Cambridge, New York.

Eddy,S. (1998) HMMER: profile HMMs for protein sequence analysis. *Bioinformatics*, **14**, 755–763.

Eddy,S. and Durbin,R. (1994) RNA sequence analysis using covariance models. *Nucleic Acids Res.*, **22**, 2079–2088.

Giegerich,R. and Höner zu Siederdissen,C. (2010) Semantics and ambiguity of stochastic RNA family models. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, **99**. Available at http://www.computer.org/portal/web/csdl/doi/10.1109/TCBB.2010.12.

Giegerich,R. and Meyer,C. (2002) Algebraic Dynamic Programming. In *Algebraic Methodology And Software Technology*, Vol. 2422, Springer, Berlin/Heidelberg, pp. 243–257.

Griffiths-Jones,S. *et al*. (2003) Rfam: An RNA family database. *Nucleic Acids Res.*, **31**, 439–441.

Mattick,J. and Makunin,I. (2006) Non-coding RNA. *Hum. Mol. Genet.*, **15**, R17–R29.

Nawrocki,E. *et al*. (2009a) Infernal 1.0: inference of RNA alignments. *Bioinformatics*, **25**, 1335–1337.

Nawrocki,E. *et al*. (2009b) *INFERNAL User Guide*.

Reeder,J. and Giegerich,R. (2005) Consensus shapes: an alternative to the Sankoff algorithm for RNA consensus structure prediction. *Bioinformatics*, **21**, 3516–3523.

Downloaded from bioinformatics.oxfordjournals.org by guest on October 16, 2010

Chapter 7

A Folding Algorithm for Extended RNA Secondary Structures

Christian Höner zu Siederdissen, Stephan H. Bernhart, Peter F. Stadler, and Ivo L. Hofacker.
A folding algorithm for extended RNA secondary structures.
Bioinformatics. 2011. 27 (13). i129–i136.

CHzS and ILH reduced the exponential runtime of `MC-Fold` to the polynomial runtime $O(n^3)$ of `MC-Fold-DP`. All authors participated in the design of the new extended secondary structure folding algorithm, `RNAwolf`, based on 2-diagrams. CHzS implemented all algorithms in Haskell. The authors have shared writing the paper.

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/2.5), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

BIOINFORMATICS

Vol. 27 ISMB 2011, pages i129–i136
doi:10.1093/bioinformatics/btr220

A folding algorithm for extended RNA secondary structures

Christian Höner zu Siederdissen[1,*], Stephan H. Bernhart[1], Peter F. Stadler[1,2,3,4,5,6] and Ivo L. Hofacker[1,5]

[1]Institute for Theoretical Chemistry, University of Vienna, A-1090 Vienna, Austria, [2]Bioinformatics Group, Department of Computer Science, and Interdisciplinary Center for Bioinformatics, University of Leipzig, D-04107 Leipzig, [3]Max Planck Institute for Mathematics in the Sciences, [4]RNomics Group, Fraunhofer IZI, D-04103 Leipzig, Germany, [5]Center for Non-Coding RNA in Technology and Health, University of Copenhagen, Grønnegårdsvej 3, DK-1870 Frederiksberg, Denmark and [6]The Santa Fe Institute, Santa Fe, 87501 NM, USA

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

ABSTRACT

Motivation: RNA secondary structure contains many non-canonical base pairs of different pair families. Successful prediction of these structural features leads to improved secondary structures with applications in tertiary structure prediction and simultaneous folding and alignment.

Results: We present a theoretical model capturing both RNA pair families and extended secondary structure motifs with shared nucleotides using 2-diagrams. We accompany this model with a number of programs for parameter optimization and structure prediction.

Availability: All sources (optimization routines, RNA folding, RNA evaluation, extended secondary structure visualization) are published under the GPLv3 and available at www.tbi.univie.ac.at/software/rnawolf/.

Contact: choener@tbi.univie.ac.at

1 INTRODUCTION

The classical RNA secondary structure model considers only the Watson–Crick AU and GC base pairs as well as the GU wobble pair. A detailed analysis of RNA 3D structures, however, reveals that there are 12 basic families of interactions between the bases, all of which appear in nature (Leontis and Westhof, 2001; Leontis *et al.*, 2002). Moreover, virtually all known RNA tertiary structures contain the so-called non-Watson–Crick base pairs. This has led to the development of an extended presentation of RNA contact structures with edges labeled by their pairing type (an example can be seen in Fig. 1). This extended description of base pairing is commonly termed after its inventors the Leontis–Westhof (LW) representation.

The LW representation has proved to be a particularly useful means of analyzing 3D structures of RNA as determined by X-ray crystallography and NMR spectroscopy (Leontis and Lescoute, 2006). In particular, it has led to the discovery of recurrent structural motifs, such as kink-turns and C-loops, that act as distinctive building blocks of 3D structures. The sequence variation in these structural motifs follows combinatorial rules that can be understood by the necessity to maintain the overall geometry when base pairs are exchanged. These isostericity rules are discussed in detail by Lescoute *et al.* (2005); Stombaugh *et al.* (2009). As a new level of RNA structure description, the ability to predict non-standard base pairs can be expected to improve the performance of RNA structure prediction. Furthermore, information about evolutionary

*To whom correspondence should be addressed.

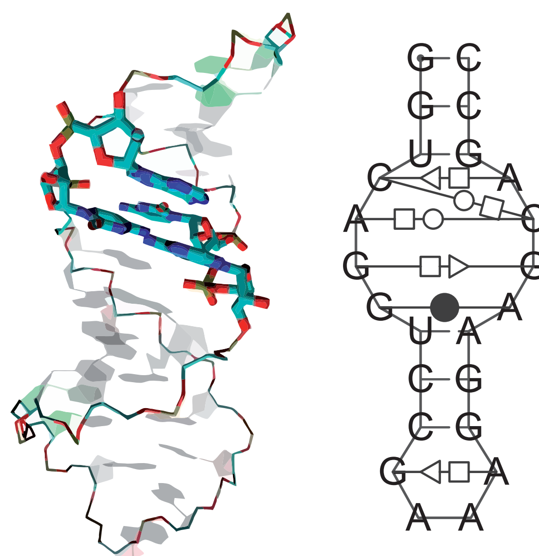

Fig. 1. Example of a structure containing base triplets. The inner part (bases 14–37) of the PDB structure `1dul` is shown in a 3D representation and as a 2D structure plot displaying the non-standard base pairs in LW representation. The four bases highlighted in the 3D structure form the two base triplets that can be seen in the upper part of the interior loop in the 2D structure.

conservation of the isostericity classes of these non-standard base pairs will improve consensus structure-prediction and structure-dependent RNA gene finding.

Since many additional interactions beyond the standard base pairs are represented in the LW formalism, what was considered to be a loop in classical secondary structures can now appear as complex structures of non-standard base pairs. These non-standard base pairs effectively divide the long ‘classical’ loops into much shorter ones. Parisien and Major (2008) proposed a model that contains loops with no more than four unpaired bases. For unbranched structures, the model is scored using a statistical potential estimated from the available 3D structures by counting the relative frequencies of base pairs, short unbranched loops of particular shapes in dependence of their sequences and combinations of loops with a common base pair. An accompanying folding procedure, `MC-Fold` (Parisien and Major, 2008), which exhaustively enumerates stem-loop components, is available and has been used very successfully as a first step toward the *de novo* prediction of RNA 3D structures

© The Author(s) 2011. Published by Oxford University Press.
This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/2.5), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

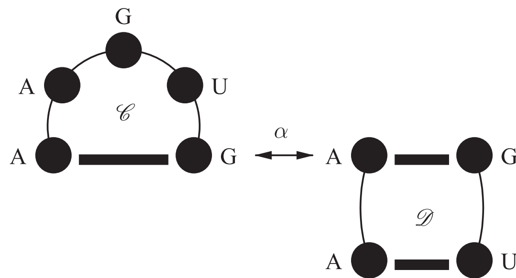

Fig. 2. `MC-Fold` and `MC-Fold-DP` both consider small loops, like the hairpin `AAGUG` ($\mathscr{C}$) and the 2×2 stack `AAGU` ($\mathscr{D}$) (read clockwise, starting bottom left). Each loop is scored by a function $E_c(\mathscr{C}|\text{AAGUG})$. The stack ($\mathscr{D}$) follows analoguously. The interaction term between two loops is calculated as indicated by the arrow (α), where the two loops are overlayed at the common `AG` pair. The contribution of the interaction is $E_{\text{junction+hinge}}(\mathscr{C},\mathscr{D};\theta;\text{A},\text{G})$ with θ the unknown pair family.

using `MC-Sym` (Parisien and Major, 2008), which takes as input the proposed secondary structure from `MC-Fold`.

2 MC-FOLD REVISITED

2.1 Algorithm

Like ordinary secondary structure prediction tools, `MC-Fold` (Parisien and Major, 2008) is based on a decomposition of the RNA structure into 'loops'. In contrast to the standard energy model, however, it considers the full set of base pair types available in the LW representation. Each base pair, therefore, corresponds to a triple $(i,j;\theta)$ where θ is one of the 12 types of pairs. In this model, ordinary secondary structures are the subset of pairs with Watson-Crick—Watson-Crick type ($\theta=$'`WW`') and the two nucleotides form one of the six canonical combinations $\{\mathsf{AU},\mathsf{UA},\mathsf{CG},\mathsf{GC},\mathsf{GU},\mathsf{UG}\}$. This extension of the structure model also calls for a more sophisticated energy model. While the standard model assumes the contributions of the loops to be strictly additive, `MC-Fold` also considers interactions between adjacent unbranched loops (hairpins, stacked pairs, bulges and general interior loops). This means that the total energy of a structure is not only dependent on the loop types present, but also on the arrangement of these loops. Dispensing with details of the parametrization, the scoring function of `MC-Fold` for a structure $\mathfrak{S}$ on sequence x can be written as follows (see Fig. 2):

$$E(\mathfrak{S}|x)=\sum_{\mathscr{C}}E_c(\mathscr{C}|x[\mathscr{C}]) + \sum_{\substack{\mathscr{C}',\mathscr{C}''\\(k,l)=\mathscr{C}'\cap\mathscr{C}''}} E_{j+h}(\mathscr{C}',\mathscr{C}'';\theta;x[k],x[l]) \tag{2.1}$$

where $\mathscr{C},\mathscr{C}',\mathscr{C}''$ are different loops of $\mathfrak{S}$. The additive term E_c tabulates the (sequence-dependent) contributions of the loops. The interaction term E_{j+h} accounts for the 'junction' and 'hinge' terms in stem–loop regions. These interaction terms depend on the type of the adjacent loops as well as on the type θ and sequence $(x[k],x[l])$ of the base pair that connects them. For multiloops, only the additive term is considered.

Let us ignore multiloops for the moment. A basepair $(i,j;\theta)$ then encloses a loop of type $\mathscr{L}$ which is either a hairpin or encloses a loop $\mathscr{K}$. It is connected to $\mathscr{K}$ by a base pair $(k,l;\psi)$ with $i<k<l<j$. Let $B_{ij}(\theta;\mathscr{L})$ be the minimal energy of a structure on $x[i..j]$ enclosed by a base pair $(i,j;\theta)$ with an outermost loop of type $\mathscr{L}$. Note that, in our notation, the loop type $\mathscr{L}$ also specifies its length and hence implicitly determines the coordinates of the inner base pair of an interior loop: $(k,l)=(i+\ell_1(\mathscr{L}),j-\ell_2(\mathscr{L}))$. For simplicity, we write $(k(\mathscr{L}),l(\mathscr{L}))$. If $\mathscr{L}$ is a hairpin, then $B_{ij}(\theta;\mathscr{L})=\mathscr{H}[i,j;\theta;\text{hairpin}]$, a tabulated energy parameter. Otherwise, we have the recursion

$$B_{ij}(\theta;\mathscr{L})=\min_{\psi,\mathscr{K}}\Big(\mathcal{I}[i,j;\theta;\mathscr{L};\psi,\mathscr{K}]+B_{k(\mathscr{L}),l(\mathscr{L})}(\psi;\mathscr{K})\Big) \tag{2.2}$$

This can be expanded to a full 'next-nearest-neighbor' model by enforcing an explicit dependence on the type of the inner base pair:

$$B_{ij}(\theta;\mathscr{L};\psi)=\min_{\psi,\mathscr{K},\phi}\Big(\mathcal{I}[i,j;\theta;\mathscr{L};\psi;\mathscr{K};\phi] + B_{k(\mathscr{L}),l(\mathscr{L})}(\psi;\mathscr{K};\phi)\Big) \tag{2.3}$$

The effort to evaluate this recursion equation for a fixed base pair (i,j) is L^3T^3, where L is the number of loop types and T is the number of base pair types. While this prefactor is inconveniently large, we nevertheless obtain an $\mathcal{O}(n^2)$ [or $\mathcal{O}(n^3)$ with multibranched loops] folding algorithm instead of the exponential runtime of `MC-Fold`.

The problem with this general form of energy parametrization is the unmanageable number of parameters that need to be measured, estimated or learned from a rather limited set of experiments and known RNA structures.

2.2 Parametrization and implementation

Since the folding problem for the `MC-Fold` model can be solved in polynomial time, the associated parameter estimation problem becomes amenable to advanced parameter optimization techniques (Andronescu *et al.*, 2007; Do *et al.*, 2008). At present, however, we have opted to extend the original `MC-Fold` parameters only by simple sparse data corrections that can be applied on top of the original `MC-Fold` database. This has the advantage of allowing a direct comparison between the original version of `MC-Fold` and our dynamic programming version `MC-Fold-DP`. In contrast to the original version, `MC-Fold-DP` can cope with large data sets and long sequences (3 s for 250 nt, about 24 s for 500 nt with `MC-Fold-DP`, compared to 660 s for 100 nt with `MC-Fold`).[1]

In terms of algorithmic design, we have made several changes. The grammar underlying `MC-Fold-DP` follows the ideas of Wuchty *et al.* (1999). This makes the generation of all suboptimal structures in an energy band above the ground state possible. The decomposition of interior loops into small loops implies that `MC-Fold-DP` runs in $\mathcal{O}(n^3)$ time without the need for the usual explicit truncation of long interior loops. The recursion that fills stem loops [*Nucleotide Cyclic Motifs* (NCMs) in the nomenclature of Parisien and Major (2008)] is now reduced to a function $\text{NCM}(i,j,\text{type}_{i,j},k,l,\text{type}_{k,l})$. For the matrix, entry $(i,j,\text{type}_{i,j})$ is minimized over all $(k,l,\text{type}_{k,l})$ with (k,l) determined by the newly inserted motif $\text{type}_{i,j}$. Hairpins are even simpler: they follow $\text{NCM}(i,j,\text{type}_{i,j},\dots)$ but there is no inner part $(k,l,\text{type}_{k,l})$.

[1]Note that the implementation of `MC-Fold-DP` has *not* been aggressively optimized apart from using the polynomial-time algorithm.

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

The total number of motif types is small (15 in the original set, of which not all are actually used). Both the time and space complexities are, therefore, small enough to handle RNAs with a length of several hundred nucleotides, i.e. in the range that is typically of interest. In fact, the time complexity is similar to ordinary secondary structure prediction where interior loop size is bounded by a constant. Since the grammar is unambiguous, it is also straightforward to compute partition functions and base pairing probabilities, although this feature is not available in the current implementation.

3 BEYOND 1-DIAGRAMS

3.1 Base triplets

An important restriction of secondary structures is that each nucleotide interacts with at most one partner. In combinatorial terms, secondary structures are 1-diagrams. A closer analysis of the available 3D structures, however, reveals that many nucleotides form specific base pairs with two other nucleotides, forming *'base triplets'* or, more generally, *'multi-pairings'*. Cross-free b-diagrams with maximal number b of interaction partners for each nucleotide can be treated combinatorially in complete analogy with (pseudoknot-free) secondary structures by conceptually splitting each node into as many vertices as there are incident base pairs (arcs). As in the case of secondary structures, we say that (i,j) and (k,l) cross if $i<k<j<l$ or $k<i<l<j$. A b-diagram is non-crossing if no two arcs cross. Base pairs can then be well-ordered also in this extended setting: two distinct arcs $(i,j)\neq(k,l)$ are either *nested* ($i\leq k<l\leq j$) or *juxtaposed* ($i<j\leq k<l$). This observation is used in `RNAMotifScan` (Zhong *et al.*, 2010) to devise a dynamic programming algorithm for sequence structure alignments along the lines of `RNAscf` (Bafna *et al.*, 2006) or `locarna` (Will *et al.*, 2007), which in turn are restricted variants of the Sankoff algorithm (Sankoff, 1985).

Here, we consider only structures with at most two base pairs involving the same nucleotide, i.e. 2-diagrams. In this case, there is a convenient string representation generalizing the Vienna (dot-parentheses) notation for secondary structures by introducing three additional symbols <, >, `X` for positions in which two arcs meet: `((` = <, `))` = > and `)(` = `X`. For general b, the number of necessary symbols grows quadratically, $s_b=(b+1)(b+2)/2$, since each must encode b_1 opening and b_2 closing pairs with $b_1,b_2\geq 0$ and $b_1+b_2\leq b$. These symbols provide a direct representation of the arc nodes '$\triangleleft, \triangleright, \times$' of Figure 3 and are an optional output of the folding program described below to visualize 2-diagrams in the secondary structure.

3.2 A grammar with base triplets

In order to design a dynamic programming folding algorithm for cross-free 2-structures we need a decomposition, i.e. a grammar for 2-structures. For practical applications, it is desirable to have not only a minimization algorithm, but also a partition function version. To this end, an unambiguous grammar is required (Dowell and Eddy, 2004; Reeder *et al.*, 2005). A simple version, treating base pairs as the elementary entities is shown in Figure 3. It translates into an extension of either a Nussinov-style algorithm for maximizing the number of base pairs or a recursion for counting the number of non-crossing 2-diagrams. Let F_{ij} denote the minimum energy of a structure on the sequence interval $x[i..j]$. We have

$$F_{ij}=\min\begin{cases} F_{i+1,j} \\ C_{ij} \\ \epsilon^a_{i,j}+U_{i,j-1} \\ \epsilon^b_{i,j}+V_{i+1,j} \\ \epsilon^c_{i,j}+W_{i,j} \\ \min\limits_{i<k<j}\begin{cases} C_{i,k}+F_{k+1,j} \\ \epsilon^a_{i,k}+U_{i,k-1}+F_{k+1,j} \\ \epsilon^b_{i,k}+V_{i+1,k}+F_{k+1,j} \\ \epsilon^c_{i,k}+W_{i,k}+F_{k+1,j} \\ \epsilon^a_{i,k}+F_{i+1,k-1}+U_{k,j} \\ \epsilon^c_{i,k}+U_{i,k-1}+U_{k,j} \end{cases} \end{cases} \tag{3.1}$$

and analogous recursion for U_{ij},V_{ij} and W_{ij}, denoting the minimum energies over all structures whose left, right or both ends, are involved in a triplet. The symbol C_{ij} refers to structures enclosed by a non-triplet base pair. In the simplest case, $C_{ij}=\epsilon_{ij}+F_{i+1,j-1}$ (lower right corner of Fig. 3). The terminal symbols are the unpaired base •, the ordinary base pairs and the three types of base pairs involved in triplets, contributing $\epsilon_i=0$ sequence-dependent energy increment ϵ_{ij} and sequence-dependent energy increments ϵ^a_{ij}, ϵ^b_{ij} and ϵ^c_{ij}, respectively. The recursion is initialized with $F_{ii}=0$.

Only certain combinations of types of base pairs can occur in triplets. Thus, in a refined model we need to replace U_{ij}, V_{ij} and W_{ij} by $U_{ij}[\nu]$, $U_{ij}[\mu]$ and $W_{ij}[\xi]$ explicitly referring to the base pair type(s) of the triplet. Furthermore, the energy parameters also become type dependent $\epsilon^a_{ij}\to\epsilon^a_{ij}[\rho]$ or even $\epsilon^a_{ij}[\rho,\nu]$ where ρ is the type of the pair itself and ν is type of the second pair of the triplet. The first variant is chosen for Nussinov-like algorithms, where each individual base pair is evaluated, splitting triplets, and the second variant is more fitting for Turner-like nearest neighbor models. In that case, recursion on W changes to $W_{ij}[\nu,\mu]$ to reflect the pairing choice being made.

3.3 Full loop-based model

The grammar of Figure 3 can be extended to incorporate the standard loop-based Turner energy model (Turner and Mathews, 2010) (which distinguishes hairpin loops, stacks of two base pairs, bulges, interior loops and multibranched loops). The modification of the grammar is tedious but rather straightforward, as seen in Figure 4. Instead of treating the base pairs themselves as terminal symbols (as in Fig. 3), this role is taken over by entire loops. Note that as in the case of ordinary secondary structures, each loop in a given structure is uniquely determined by its closing pair. The energy contributions now depend, in a more complex way, on the characteristics of the loop, hence we also need additional non-terminals to describe e.g. the components of multiloops.

We use a decomposition that is similar to that of `MC-Fold` and in addition encompasses 2-diagrams. A $p\times q$-loop, $p\leq q$, consists of p nucleotides on one strand and q nucleotides on the other one. In particular, 2×2-loops correspond to stacked base pairs, $1\times q$-loops, $q>1$ are triplets and 2×3-loops are stacks with a bulged-out nucleotide. In addition to hairpin loops and these $p\times q$-loops, we consider generic bulges with and without a shared nucleotide, interior loops of larger sizes and multibranched loops,

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

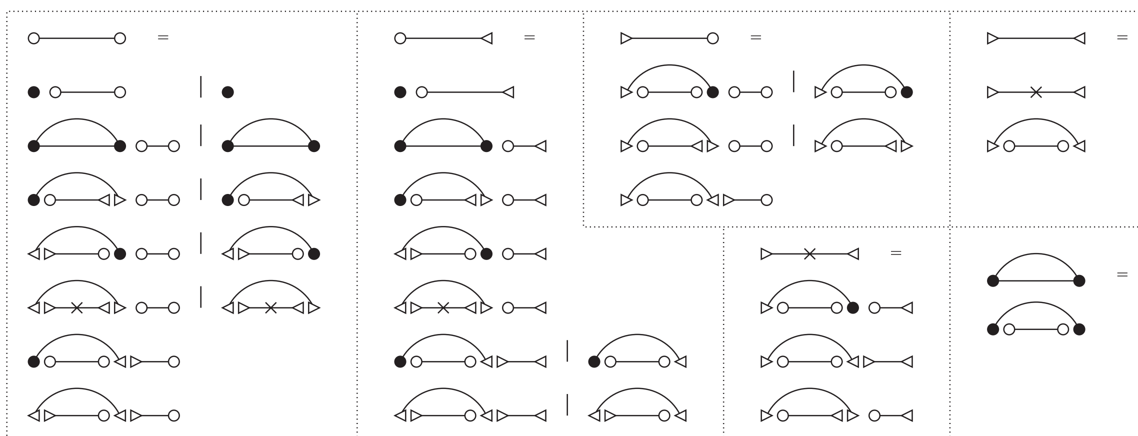

Fig. 3. A simple unambiguous grammar for non-crossing 2-diagrams (The symbols used here denote (non-)terminals in a context-free grammar and are not to be confused with the LW notation used in other figures). Connected parts of diagrams correspond to terminal [individual bullet with no arc (closed circle) = unpaired nucleotide; arc with circular end points (closed circle, open circle) = base pair; arc with triangular endpoints (left-faced triangle, right faced triangle, cross) = part of base triple] or non-terminal (horizontal lines and semicircle) symbols of the grammar. It is important to realize that left-faced and right-faced triangles refer to the same nucleotide when they are adjacent. In terms of a recursion, the index for both left-faced and right-faced triangles is therefore the same. One triangle 'points' to the outer arc and one to the inner arc incident to the same nucleotide.

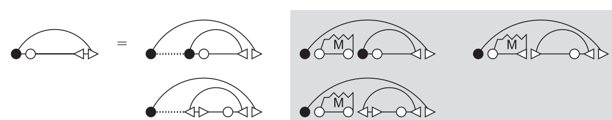

Fig. 4. Decomposition of one non-terminal in the full loop-based model with triples. The l.h.s. of the production rule denotes a structure enclosed by a base pair where the base at the 3′ end is part of a triple. The second base pair of this triple ends within the structure. The structural element is either bulge like (first column) or multiloop like. In the first case, we have to distinguish whether the enclosed structure has a normal pair or a triple at its 5′ side. In the multiloop case, we use the linear decomposition into components familiar from the Turner model with a non-terminal denoting a partial multiloop containing at least one base pair. Here, we need to distinguish whether the 5′ end of the rightmost component and 3′end of the left components are triples or not. As the multiloop part is not implemented in our current version, it is grayed out.

again possibly with shared nucleotides. Figure 4 gives an example for the full loop-based decomposition of one particular non-terminal. In our current implementation, we use several simplifications in particular for multiloops that involve triplets. Some information on the complete grammar used in our implementation can be found in the Appendix A, other information is available on the `RNAwolf` homepage.

4 IMPLEMENTATION

4.1 Folding software

The implementation available on the `RNAwolf` homepage is written in the high-level functional programming language Haskell. While this leads to an increase in running times (by a constant factor), the high-level notation and a library of special functions lead to very concise programs, and enable, e.g. the use of multiple cores.

Currently, the following algorithms are implemented: (i) an optimizer which takes a set of melting experiments and the PDB database as input and produces a parameter file optimized as described below. (ii) A folding program which expects a sequence of nucleotides as input and produces an extended secondary structure prediction which includes nucleotide pairs of non-canonical types. Furthermore, it can contain motifs with base triplets. (iii) An evaluation program which expects both, a sequence and a secondary structure. The input is then evaluated to return the score of said structure and, if requested, tries to fill the given (canonical) structure with additional pairs. This allows to turn a classical secondary structure into an extended secondary structure by filling large loops with non-canonical pairs.

At the moment, base triplets have been restricted slightly in that shared nucleotides are only possible in stem structures, not within a multibranched loop motif. Allowing shared nucleotides between two helices of a multiloop would slow down multiloops by a significant factor. Nevertheless, we will lift this restriction for the full nearest neighbor model we plan to implement. In the full model, we will be able to use data gathered from our current model to reduce the combinatorial complexity of the algorithm within multibranched loops.

4.2 Parameter estimation

In contrast to the Turner model, which considers only canonical base pairs [i.e. Watson–Crick and GU (wobble) pairs], we include all types of base pairs. Thus, we also have to derive parameters for all possible base pair families in our motifs of choice. To this end, we need to find sufficient evidence for each parameter and we need an efficient numerical algorithm for optimizing the parameters.

(i) Even if a large body of sequence/structure pairs is available to train the parameters, it is still highly unlikely that each parameter is witnessed. A simple calculation for canonical stacked pairs already produces $4^4 \times 12^2 = 36\,864$ (ignoring symmetries) parameters to be trained. While symmetries reduce the number of distinct parameters, canonical stacks still require $\sim 10\,000$ independent parameters. In total, the number of parameters easily reaches 10^5, which means that only a very small set of parameters will actually be observed in experimentally verified structures.

(ii) The second problem is of numerical nature in that it gets hard to estimate a solution in $\mathbb{R}^{100000}$ even under ideal circumstances. In addition, the computational effort for the computation of the solution vector is rather high. There are two different types of approaches to this problem, described in some detail by Andronescu *et al.* (2007). In max-margin formulations,

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

parameters are optimized such as to drive them away from wrongly determined structures and toward correctly determined ones. Alternatively, the conditional likelihood of known structures is maximized. Andronescu *et al.* (2010) described an extension of the algorithm that can deal with unobserved configurations by employing a hierarchical statistical model.

We have selected yet another way of dealing with the immense number of features. Instead of optimizing the full set of parameters directly, we first optimize the parameters for a restricted model closely following the simple unambiguous grammar given in Figure 3. In short a loop of type m (e.g. stacked pair, bulge, etc.) enclosed by two pairs p_1, p_2 is assigned an energy $\epsilon(m)+\epsilon^m(p_1)+\epsilon^m(p_2)$, where $\epsilon(m)$ depends on the type and size of the loop but is independent of sequence, and the pair energies depend on the identity of the nucleotides as well as the LW type (e.g. GC,cWW).

We call our model *enhanced Nussinov* as it distinguishes between loops of different types (say bulges of different lengths are assigned different scores) but assumes that pair energies are independent as in the Nussinov model.

This approach has several advantages. First, the resulting algorithm is an accessible ‘toy-model’ that can be employed to test different hypotheses. Second, the estimated parameters provide a useful set of priors for the full model. This is important since, in contrast to the work of Andronescu *et al.* (2007), we cannot derive a complete set of priors from known data. Finally, the computational requirements are significantly lower. Training a full Boltzmann model for conditional likelihood maximization might easily have taken months of CPU time (Andronescu *et al.*, 2010).

Here, we utilize both melting experiments and PDB data for parameter estimation. Melting experiments yield a small set of sequences, structures and corresponding free energies. The structural data, unfortunately, provides almost exclusively canonical Turner features and no information regarding the base pair family, although it can be assumed that all pairs are of the Watson–Crick (cWW) style. The PDB data, on the other hand, contain not only non-canonical base pairs, but also provide information on the base pair family. In addition, PDB entries typically refer to structures that are much larger than those used in melting experiments.

Together, both sets provide data required for the estimation of an extended set of parameters. In order to keep computation times short, we employ the original no-max-margin constraint-generation approach used by Andronescu *et al.* (2007). While not providing the most accurate parameters in the original paper, the relatively short runtimes of ~ 1 CPU day are convenient for experimental purposes. In addition, since we are training an enhanced Nussinov-style model, we can assume that the prediction accuracy is limited by the structure of the model. More advanced, and hence computationally more expensive, training methods are therefore unlikely to lead to substantial improvements of the prediction accuracy.

4.3 Optimization

Our task is to estimate the energy contributions x_j for a given collection of features j. In this context, a feature corresponds to a terminal symbol in our grammar with a fixed underlying sequence, such as as GC/GC stacked pair or a 1×3-loop with sequence (G—AUC) where GA is a Hoogsteen pair and GC is a Watson–Crick pair. We are given the following types of data: (i) a matrix A whose entries $A_{i,j}$ encode how often feature j occurs in sequence/structure pair i, and (ii) a vector y containing measured melting temperatures y_i for experiment i.

Constraints are now generated as follows. For each entry k of the PDB, we extract the (extended) secondary structure features. This means that neither pseudoknots nor intermolecular interactions (which require more complicated grammars) are considered. The entry f_j^T of the row vector f^T counts how often feature j is observed in the structure. Using the current parameter values x (see below), the sequence of PDB entry k is folded and the corresponding feature vector g^T is constructed. If the predicted fold has a lower free energy than the known structure, a new constraint $(f-g)^T x \leq 0$ is introduced. Note that $f^T x$ and $g^T x$ are, by construction, the free energies of the known and the predicted structure evaluated with the current parameters x. Since the true structure is expected to be the thermodynamic ground state, its free energy must be smaller than that of any other structure. The constraint matrix D contains all currently active constraints where $D_{k,.}$ is the k-th active constraint (in this notation $D_{k,.}$ selects the k-th row, while $D_{.,l}$ would select the l-th column).

Following Andronescu *et al.* (2007), we use a slack variable d_k for each constraint so that $D_{k,.}x \leq d_k$. This guarantees that the problem remains feasible as otherwise conflicting constraints could reduce the feasible set for x to the empty set. The slack variables d_k are bounded from below by $0 \leq d_k$ because $(f-g)^T x \geq 0$, with equality for cooptimal structures.

Norm minimization problems can drive individual variables x_i to extreme values. We, therefore, constrain the energy contribution of individual features to $|x_i| < 5$ kcal/mol. A subset S of features that act as penalties are constrained to positive values, $x_j > 0$ for $j \in S$. The set S is defined along the following principles: unpaired loop regions destabilize the structure relative to a random coil and hence should be penalized. Hairpins, bulges and interior loops fall into this category. In addition, 1×2 and 2×3 stems, which are otherwise modeled as 2×2 stems, are penalized. Hence, for e.g. the 2×3 loop `CAUGG` with `A` unpaired, we have $\epsilon(CG)+\epsilon(UG)+\epsilon(2\times 3)$ where $\epsilon(2\times 3)$ is the penalty term.

Parameter estimation is thus reduced to the constrained norm optimization problem

$$\left\| \begin{pmatrix} A & 0 \\ D & -I \end{pmatrix} \begin{pmatrix} x \\ d \end{pmatrix} - \begin{pmatrix} y \\ 0 \end{pmatrix} \right\|_2 \tag{4.1}$$

with the linear constraints

$$-5 < x_j < 5, \qquad 0 < x_l, \quad l \in S, \qquad 0 < d_k\,. \tag{4.2}$$

Since this optimization problem is convex it can be solved efficiently.

The parameter vector x is optimized iteratively. Initially, D is empty and no slack variables d are used. After the first step, all PDB sequences have been folded and those for which the predicted structure is different from the known structure are included as a row in D as described above. The slack variables are initialized as $d_k = D_{k,.}x + \gamma$ for each constraint k, where $\gamma \in \mathbb{R}_+$ is a small constant. Iterations of the optimization procedure continue until no more constraints have to be added.

The computational effort required, both to estimate the parameters and to fold a single sequence, is higher than what is required for the Turner model. The additional computational effort required by the folding algorithm is mainly a result of the inclusion of the pair family information. In the case of 2-loops (stacks, bulges, interior loops), we incur an additional factor of 12 since each possible pair family has to be considered. More problematic are multibranched loops in the case of shared nucleotides as now there are up to 12×11 possibilities to connect a shared nucleotide with its pairing partners.

4.4 Comparison with turner parameters

A comparison with the parameter sets by Turner (Turner and Mathews, 2010) shows that individual contributions are similar enough to make the the ‘enhanced Nussinov’ model a useful prior in the parameter optimization for the full model. Consider, for example, canonical 2×2 stacks, where one pair is of type **GC, cWW** type and the other pair is of type **XY, cWW**, with **XY** $\in$ {**GC,CG,AU,UA,GU,UG**} and **cWW** stands for *cis/Watson–Crick/Watson–Crick*, the canonical pair type. In the Turner-2004 model, energy contributions range from -1.5 to -3.4 kcal/mol, while the base-pair contribution for the **GC, cWW** pair is -1.36 kcal/mol in the optimized ‘enhanced Nussinov’ model. Depending on the second pair, we observe discrepancies of ≈ 0.5 when comparing the sum of individual pair energies to the total stacking energy. This level of agreement is expected and suggests that it makes sense in later iterations of parameter estimation to constrain features to tighter intervals than the current setting of using the open interval of $]-5,5[$.

5 RESULTS AND DISCUSSION

MC-Fold-DP: `MC-Fold-DP` and the original `MC-Fold` by (Parisien and Major, 2008) show comparable performance on a

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

Table 1. Prediction accuracy of `MC-Fold`, `MC-Fold-DP` and `RNAfold 1.8.4` on a set of 347 sequences from the RNAstrand database

Algorithm	Count	MCC	*F*	S	PPV
`MC-Fold`, $\leq$50 nt	298	0.74	0.74	0.80	0.70
`MC-Fold`, $\leq$100 nt	37	0.54	0.54	0.66	0.46
`MC-Fold`, $>$100 nt	12	0.49	0.49	0.53	0.46
`MC-Fold-DP`, $\leq$50 nt	298	0.71	0.71	0.77	0.68
`MC-Fold-DP`, $\leq$100 nt	37	0.53	0.53	0.64	0.45
`MC-Fold-DP`, $>$100 nt	12	0.38	0.37	0.51	0.29
`RNAfold`, $\leq$50 nt	298	0.76	0.76	0.73	0.81
`RNAfold`, $\leq$100 nt	37	0.73	0.73	0.73	0.73
`RNAfold`, $>$100 nt	12	0.63	0.63	0.66	0.60

All sequences are <200 nt long. The longest sequence took just under an hour of computation time using `MC-Fold`. `MC-Fold-DP` can compute the predicted structure in ~1 s (loading the `MC-Fold` motif database requires an additional 1–2 s). Prediction quality has been measured on canonical base pairs only for comparison purposes. Note the small number of sequences >100 nt. (MCC, Matthews correlation coefficient; F, F-Measure; S, Sensitivity; PPV, Positive Predictive Value).

set of 347 sequences selected from the RNAstrand (Andronescu *et al.*, 2008) database. There are several differences between the two algorithms. First, the runtime, where `MC-Fold-DP` is about $\times 200 - \times 1000$ faster for biologically relevant sequences (i.e. <1000 nt). Table 1 shows a small comparison of the prediction accuracy given different measures. Second, we allow for sparse data correction, which can be disabled by the user. And third, the algorithm accepts non-canonical input (e.g. 'N' characters) and can be configured to calculate approximate scores for motifs containing such characters.

Differences in predictions are the result of internals of the orignial algorithm that have remained unknown to us since they are not described in full detail in Parisien and Major (2008).

It should be noted that our reformulation makes `MC-Fold-DP` amenable for the parameter optimization approaches pioneered by Andronescu *et al.* (2010) for which a polynomial-time prediction algorithm is crucial. The non-ambiguous grammar allows even the advanced, Boltzmann Likelihood-based, approaches to be employed. This presents an opportunity for future research.

RNAwolf: we compared our *enhanced Nussinov* algorithm to three state-of-the-art thermodynamic folding algorithms [RNAfold (Hofacker *et al.*, 1994), UNAfold (Markham and Zuker, 2008) and RNAstructure (Reuter and Mathews, 2010)] to assess the prediction quality of our model. We folded a subset of 550 randomly chosen structures from RNAstrand (Andronescu *et al.*, 2008) and compared the F-measure of our results with those of the other programs. The results in Figure 5A show that, not unexpectedly, the 'enhanced Nussinov' algorithm cannot compete with state-of-the-art tools due to its simplified energy model.

Interestingly, once we focused on data gathered from the PDB database (Fig. 5B), the results showed a remarkable improvement. This could suggest that the PDB structures used for training do not sufficiently cover the RNA structure space and that additional RNAs (for which only secondary structure information is available) should be included in the training.

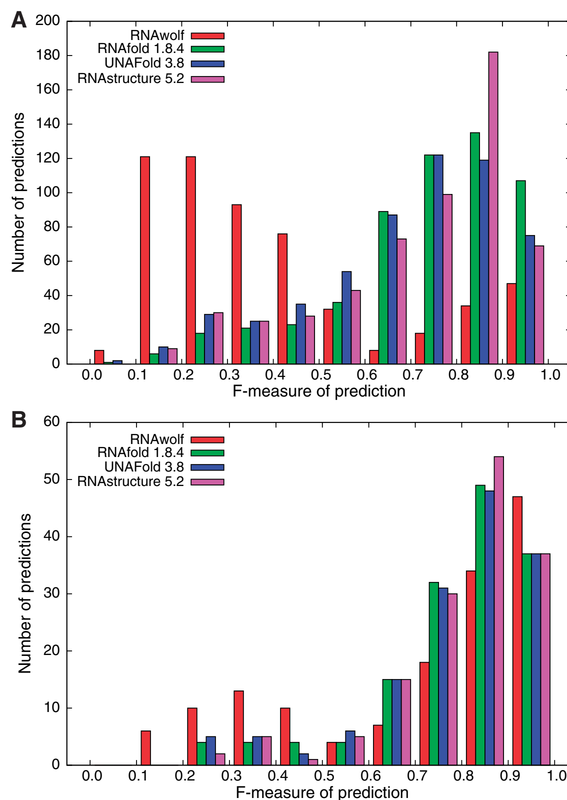

Fig. 5. (**A**) Histogram of F-measures for different folding algorithms, given 550 random RNAstrand entries. (**B**) F-measures, given 155 PDB entries from the RNAstrand, which are a subset of the 550 random RNAstrand entries.

Because of the large number of base pair types, the 'enhanced Nussinov-algorithm', has to perform more work than classical secondary structure prediction programs when filling the dynamic programming matrices. This is reflected by rather high runtimes (25 s for 100 nt, 110 s for 200 nt). However, the asymptotic time complexity is still in $\mathcal{O}(n^3)$.

A constrained folding variant of the 'enhanced-Nussinov' algorithm can be used, for example, to predict non-canonical base pairs in large interior loops of structures. As an example, Figure 6, shows that `RNAwolf` is able to correctly predict the non-canonical base pairs in a situation where the canonical base pairs are already given, i.e. where the input consists of both the sequence and a dot-bracket string representing canonical Watson–Crick base pairs. Only the zig-zag motif (upper part of the interior loop) was not predicted, presumably due to the large penalty of +3.89 for each of the two 1×2 stacks.

Further results and a semi-automatic system for secondary structure prediction comparison (`SSPcompare`) are available on the `RNAwolf` homepage. Table 1 has been created using said program.

6 CONCLUSION

Large experimentally verified RNA structures contain a sizable number of non-canonical base pairs (Stombaugh *et al.*, 2009).

Downloaded from http://bioinformatics.oxfordjournals.org/ at Österreichische Zentralbibliothek fuer Physik on November 2, 2011

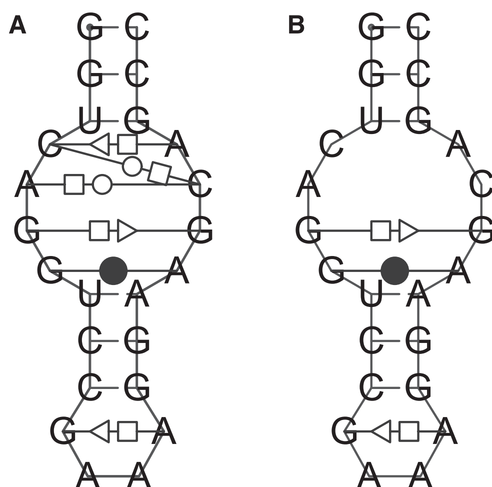

Fig. 6. Prediction of non-canonical base pairs with RNAwolf. (**A**) Known structure of PDB entry 1dul. (**B**) Constrained prediction (canonical base pairs were given) of 1dul. Only the central part of the structure is shown. The outer part of the stem contains only canonical base pairs and is not shown.

However, only a few RNA folding programs predict non-canonical pairs (Do *et al.*, 2006; Parisien and Major, 2008). With the exception of MC-Fold, the pair families are not explicitly taken into account. Here, we have shown that the prediction of non-canonical pairs together with the corresponding pair families and their possible interactions in base triples is feasible by efficient dynamic programming approaches. Although direct thermodynamic measurements are not available to cover all aspects of such an extended and refined model of RNA structures, meaningful parameter sets can nevertheless be constructed. To this end, the information of the thermodynamic measurements is combined with a feature analysis of 3D structures using one of several approaches to large-scale parameter optimization. The extended combinatorial model, which in essence covers the LW representations of RNA structures, allows a much more detailed modeling of the intrinsic structures in particular of hairpins, interior loops and bulges.

We emphasize that our contribution does not yet provide a full-fledged loop-based LW-style energy model. In essence, we still lack an implementation for the full model of multiloops. As the example of Figure 6 suggests, interactions of adjacent loops as in the MC-Fold model may also be required to obtain satisfactory prediction accuracies for practical applications. Due to the computational cost, it will also make sense to investigate the trade-off between further refinements of the model and speed-ups resulting from additive approximations. Another facet that naturally should be taken into account is coaxial stacking, in particular in the context of multiloops (Tyagi and Mathews, 2007). We have demonstrated here that the goal of an accurate, practically applicable folding algorithm for LW structures is meaningful and reachable: the work of Parisien and Major (2008) shows that major improvements of prediction accuracy can be obtained by employing LW-based folding algorithms. Although RNAwolf does not yet reach the desired levels of accuracy, it allows us to explore the missing components of the energy model in a systematic manner, and it demonstrates that this can be achieved without leaving the realm of fast, efficient and exact dynamic programming approaches. The next step, therefore, is a toolkit for optimizing parameters in the full loop-based model.

An interesting possibility for further extensions of the model is the explicit incorporation of recurring RNA structural motifs with non-canonical pairs, such as Kink-Turns (Klein *et al.*, 2001), into the grammar and the energy model. This may be particularly useful in those cases where motifs are not crossing-free and hence would require a pseudoknot version of the folding algorithm. While the inclusion of various types of pseudoknots is conceptually not more difficult than for ordinary secondary structures, the parametrization of such models will be even more plagued by the lack of training data in the LW framework.

The folding algorithm introduced here, furthermore, sets the stage for a complete suite of bioinformatics tools for LW structures. Simple extension can cover the cofolding of two or more RNAs along the lines of (Bernhart *et al.*, 2006; Dimitrov and Zuker, 2004; Dirks *et al.*, 2007). Consensus structures can be predicted from given sequence alignments using the same recursions. As in RNAalifold (Bernhart *et al.*, 2008), it suffices to redefine the energy parameters for alignment columns instead of individual nucleotides. Instead of RIBOSUM-like scores as measures of conservation (Klein and Eddy, 2003), one naturally would employ the isostericity rules for the individual base pair types (Leontis *et al.*, 2002; Lescoute *et al.*, 2005). Inverse folding algorithms (Andronescu *et al.*, 2004; Busch and Backofen, 2006; Hofacker *et al.*, 1994) design RNA sequences that fold into prescribed structures by iteratively modifying and folding sequences to optimize their fit to substructures of the target. This strategy can immediately be generalized to LW structures; in fact, in essence it suffices to replace secondary structure folding by LW style folding. Combining the algorithmic ideas of this contribution with the Sankoff-style alignment approach of Zhong *et al.* (2010) and the progressive multiple alignment scheme of mlocarna (Will *et al.*, 2007) directly leads to an LW variant of structural alignment algorithms.

ACKNOWLEDGEMENTS

The authors thank the participants of the *Refined presentation of RNA structures* workshop for lively discussions, Marc Parisien for kindly answering questions about MC-Fold, the curators of the FR3D database, and Ronny Lorenz for providing comparative data for other folding programs.

Funding: Austrian GEN-AU projects ‘bioinformatics integration network III’ and ‘regulatory ncRNAs’ in part.

Conflict of Interest: none declared.

REFERENCES

Andronescu,M. *et al.* (2004) A new algorithm for RNA secondary structure design. *J. Mol. Biol.*, **336**, 607–624.

Andronescu,M. *et al.* (2007) Efficient parameter estimation for RNA secondary structure prediction. *Bioinformatics*, **23**, i19–i28.

Andronescu,M. *et al.* (2008) RNA STRAND: The RNA secondary structure and statistical analysis database. *BMC Bioinformatics*, **9**, 340.

Andronescu,M. *et al.* (2010) Computational approaches for RNA energy parameter estimation. *RNA*, **16**, 2304–2318.

Bafna,V. *et al.* (2006) Consensus folding of unaligned RNA sequences revisited. *J. Comput. Biol.*, **13**, 283–295.

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

Bernhart,S.H. *et al.* (2006) Partition function and base pairing probabilities of RNA heterodimers. *Algorithms Mol. Biol.*, **1**, 3.

Bernhart,S.H. *et al.* (2008) RNAalifold: improved consensus structure prediction for RNA alignments. *BMC Bioinformatics*, **9**, 474.

Busch,A. and Backofen,R. (2006) INFO-RNA — a fast approach to inverse RNA folding. *Bioinformatics*, **22**, 1823–1831.

Dimitrov,R.A. and Zuker,M. (2004) Prediction of hybridization and melting for double-stranded nucleic acids. *Biophys. J.*, **87**, 215–226.

Dirks,R.M. *et al.* (2007) Thermodynamic analysis of interacting nucleic acid strands. *SIAM Rev.*, **49**, 65–88.

Dowell,R.D. and Eddy,S.R. (2004) Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, **5**, 7.

Do,C.B. *et al.* (2006) CONTRAfold: RNA secondary structure prediction without physics-based models. *Bioinformatics*, **22**, e90–e98.

Do,C.B. *et al.* (2008) Efficient multiple hyperparameter learning for log-linear models. In Platt,J.C. *et al.* (eds), *Advances in Neural Information Processing Systems 20. Proceedings of the Twenty-First Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, 2007*, MIT Press, pp. 3–6.

Hofacker,I.L. *et al.* (1994) Fast folding and comparison of RNA secondary structures. *Mh. Chemie/Chem. Mon.*, **125**, 167–188.

Klein,D.J. *et al.* (2001) The kink-turn: a new RNA secondary structure motif. *EMBO J.*, **20**, 4214–4221.

Klein,R.J. and Eddy,S.R. (2003) RSEARCH: finding homologs of single structured RNA sequences. *BMC Bioinformatics*, **4**, 44.

Leontis,N.B. and Westhof,E. (2001) Geometric nomenclature and classification of RNA base pairs. *RNA*, **7**, 499–512.

Leontis,N.B. *et al.* (2002) The non-Watson-Crick base pairs and their associated isostericity matrices. *Nucleic Acids Res.*, **30**, 3497–3531.

Lescoute,A. *et al.* (2005) Recurrent structural RNA motifs, isostericity matrices and sequence alignments. *Nucleic Acids Res.*, **33**, 2395–2409.

Leontis,N.B. and Lescoute,A.W.E. (2006) The building blocks and motifs of RNA architecture. *Curr. Opin. Struct. Biol.*, **16**, 279–287.

Markham,N.R. and Zuker,M. (2008) UNAFold: software for nucleic acid folding and hybridization. *Methods Mol. Biol.*, **453**, 3–31.

Parisien,M. and Major,F. (2008) The MC-Fold and MC-Sym pipeline infers RNA structure from sequence data. *Nature*, **452**, 51–55.

Reeder,J. *et al.* (2005) Effective ambiguity checking in biosequence analysis. *BMC Bioinformatics*, **6**, 153.

Reuter,J.S. and Mathews,D.H. (2010) RNAstructure: software for RNA secondary structure prediction and analysis. *BMC Bioinformatics*, **11**, 129–129.

Sankoff,D. (1985) Simultaneous solution of the RNA folding, alignment, and proto-sequence problems. *SIAM J. Appl. Math.*, **45**, 810–825.

Stombaugh,J. *et al.* (2009) Frequency and isostericity of RNA base pairs. *Nucleic Acids Res.*, **37**, 2294–2312.

Turner,D.H. and Mathews,D.H. (2010) NNDB: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic Acids Res.*, **38**, D280–D282.

Tyagi,R. and Mathews,D.H. (2007) Predicting helical coaxial stacking in RNA multibranch loops. *RNA*, **13**, 939–951.

Will,S. *et al.* (2007) Inferring non-coding RNA families and classes by means of genome-scale structure-based clustering. *PLoS Comput. Biol.*, **3**, e65.

Wuchty,S. *et al.* (1999) Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, **49**, 145–165.

Zhong,C. *et al.* (2010) `RNAMotifScan`: automatic identification of RNA structural motifs using secondary structural alignment. *Nucleic Acids Res.*, **38**, e176.

APPENDIX A

A PAIRFAMILY-AWARE GRAMMAR

Here, we discuss in some more detail how the base pair types affect the grammar and, hence, the folding algorithm. We start from Figure 3 and the corresponding recursion in Equation (3.1). Each base pair is now colored by its LW family. In particular, therefore, base pairs have type-dependent energy contributions $\epsilon_{ij}[\vartheta]$ for pairs not involved in base triples and energy contributions depending on the type of the pair and on the type of the incident pairs: $\epsilon^a_{ij}[\vartheta,\psi]$ if the $5'$ nucleotide i is a triplet, $\epsilon^a_{ij}[\vartheta,\phi]$ if the $3'$ nucleotide i is a triplet and $\epsilon^c_{ij}[\vartheta,\psi,\phi]$ if both delimiting nucleotides are triplets. Similarly, therefore, non-terminals delimited by triples must be colored by the base pair type(s) to allow the evaluation of the energy of the enclosing base pair. In the simplest case, as implemented in `RNAwolf`, we may assume that $\epsilon^b_{ij}[\theta,\psi]$ only depends on the pair type θ for permitted combinations of pair types and is $+\infty$ otherwise. With explicit representation of the pair family types, Equation (3.1) becomes

$$F_{ij}=\min\begin{cases}F_{i+1,j},C_{ij},U'_{ij},V'_{ij},W'_{ij}\\ \min\limits_{i<k<j}\begin{cases}C_{ik}+F_{k+1,j}\\ V'_{i+1,k}+F_{k+1,j}\\ U'_{i,k-1}+F_{k+1,j}\\ W'_{ij}+F_{i+1,k}\\ F_{i+1,k-1}+\min\limits_{\theta,\psi}\left\{U_{kj}[\psi]+\epsilon^a_{ik}[\theta,\psi]\right\}\\ \min\limits_{\theta,\psi,\phi}\left\{U_{i,k-1}[\psi]+U_{k,j,\phi}+\epsilon^c_{ik}[\theta,\psi,\phi]\right\}\end{cases}\end{cases}$$

Here, we use the abbreviations

$$U'_{ij}=\min_{\theta,\psi}\left(U_{i,j-1}[\theta]+\epsilon^a_{ij}[\theta,\psi]\right)$$

$$V'_{ij}=\min_{\theta,\psi}\left(V_{i+1,j}[\theta]+\epsilon^b_{ij}[\theta,\psi]\right)$$

$$W'_{ij}=\min_{\theta}\left(W_{i,j}[\psi,\phi]+\epsilon^c_{ij}[\theta,\psi,\phi]\right)$$

which are obtained by carrying out the optimization over the combinations of base pairing types at all triples.

The non-terminal C, designating a structure enclosed by an ordinary base pair remains unchanged since the minimization $C_{ij}=F_{i+1,j-1}+\min_\theta\epsilon_{ij}[\vartheta]$ can be carried out in the simplified energy model. The triplet terms, however, are now conditioned on the pair family at all nodes represented as triangles in Figure 3. For instance, for a structure delimited by triplet vertices at both ends which are not connected by a pair, we obtain a recursion of the form

$$W^*_{ij}[\theta,\psi]=\min_{i<k<j}\begin{cases}F_{i+1,k-1}+\epsilon^a_{ik}[\theta]+U_{k+1,j}[\psi]\\ \min_\phi F_{i+1,k-1}+\epsilon^b_{ik}[\theta,\phi]+W_{k+1,j}[\phi,\psi]\\ \min_\phi V_{i+1,k}[\phi]+\epsilon^b_{ik}[\theta,\phi]+V_{k+1,j}[\psi]\end{cases}$$

and $W_{ij}[\theta,\psi]=W^*_{ij}[\theta,\psi]$ if $\theta\neq\psi$ and
$W_{ij}[\theta,\theta]=\min\{W^*_{ij}[\theta,\theta],F_{i+1,j-1}+\epsilon_{ij}[\theta]$.

Similar recursions are obtained for the full loop-based model. For instance, for the two interloop terms in Figure 4 we have to compute

$$V^*_{ij}[\theta,\psi]=\min_{k,l,\psi}\begin{cases}\mathfrak{I}[i,j,\theta|k,l\psi]+V_{kl}[\psi]\\ \mathfrak{I}'[i,j,\theta|k,l\psi]+\min_\phi W_{kl}[\psi,\phi,\theta]\end{cases}$$

where the matrices V^*, V and W now refer to the non-terminal symbols in Figure 4 and $\mathfrak{I}[\ldots]$ and $\mathfrak{I}'[\ldots]$ denote the tabulated energy contributions for the two different types of interior loops with $3'$-triplet. For more detail, we refer to the Supplementary Material which we will make available together with the full loop-based model on the RNAwolf homepage.

Downloaded from http://bioinformatics.oxfordjournals.org/ at Oesterreichische Zentralbibliothek fuer Physik on November 2, 2011

Chapter 8

Sneaking Around *concatMap*: Efficient Combinators for Dynamic Programming

Christian Höner zu Siederdissen.
Sneaking Around *concatMap*: Efficient Combinators for Dynamic Programming.
in *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming* ICFP'12. pages 215–226, ACM, New York, NY, USA, 2012

CHzS designed the study, designed and implemented the `ADPfusion` library, and wrote the paper.

"Sneaking Around *concatMap*: Efficient Combinators for Dynamic Programming"
© 2012 Association for Computing Machinery, Inc. Reprinted by permission.
`http://doi.acm.org/10.1145/2364527.2364559`

Sneaking Around *concatMap*

Efficient Combinators for Dynamic Programming

Christian Höner zu Siederdissen
Institute for Theoretical Chemistry, University of Vienna, 1090 Wien, Austria
choener@tbi.univie.ac.at

Abstract

We present a framework of dynamic programming combinators that provides a high-level environment to describe the recursions typical of dynamic programming over sequence data in a style very similar to algebraic dynamic programming (ADP). Using a combination of type-level programming and stream fusion leads to a substantial increase in performance, without sacrificing much of the convenience and theoretical underpinnings of ADP.

We draw examples from the field of computational biology, more specifically RNA secondary structure prediction, to demonstrate how to use these combinators and what differences exist between this library, ADP, and other approaches.

The final version of the combinator library allows writing algorithms with performance close to hand-optimized `C` code.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.4 [*Programming Languages*]: Optimization

General Terms Algorithms, Dynamic Programming

Keywords algebraic dynamic programming, program fusion, functional programming

1. Introduction

Dynamic programming (DP) is a cornerstone of modern computer science with many different applications (e.g. Cormen et al. [6, Cha. 15] or Sedgewick [34, Cha. 37] for a generic treatment). Durbin et al. [8] solve a number of problems on bio-sequences with DP and it is also used in parsing of formal grammars [15].

Despite the number of problems that have been solved using dynamic programming since its inception by Bellman [1], little on methodology has been available until recently. Algebraic dynamic programming (ADP) [10, 12, 13] was introduced to provide a formal, mathematical background as well as an implementation strategy for dynamic programming on sequence data, making DP algorithms less difficult and error-prone to write.

One reviewer of early ADP claimed [10] that *the development of successful dynamic programming recurrences is a matter of experience, talent, and luck*.

The rationale behind this sentence is that designing a dynamic programming algorithm and successfully taking care of all corner cases is non-trivial and further complicated by the fact that most implementations of such an algorithm tend to combine all development steps into a single monolithic program. ADP on the other hand separates three concerns: the construction of the search space, evaluation of each candidate (or correct parse) available within this search space, and efficiency via *tabulation* of parts of the search space using annotation [13] of the grammar.

In this work we target the same set of dynamic programming problems as ADP: dynamic programming over sequence data. In particular, we are mostly concerned with problems from the realm of computational biology, namely RNA bioinformatics, but the general idea we wish to convey, and the library based on this idea, is independent of any specific branch of dynamic programming over sequence data.

In particular, our introductory example uses the CYK algorithm [15, Cha. 4.2] to determine if the input forms part of its context-free language. In other words: our library can be used to write generic high-performance parsers.

The idea of expressing parsers for a formal language using high-level and higher-order functions has a long standing in the functional programming community. Hutton [20] designed a library for *combinator parsing* around 20 years ago. Combining simple parsers for terminal symbols using symbolic operators is now widespread. The *parsec* (see [30, Cha 16] for a tutorial) library for Haskell might be the most well-known. Combinators can be used to build complex parsers in a modular way and it is possible to design new combinators quite easily.

The crucial difference in our work is that the combinators in the ADPfusion library provide efficient code, comparable to hand-written `C`, directly in the functional programming language Haskell [19]. The work of Giegerich et al. [13] already provided an implementation of combinators in Haskell, albeit with large constants, for both space and time requirements. Translation to `C` [11], and recently a completely new language (GAP-L) [33] and compiler (GAP-C) based on the ideas of ADP were introduced as a remedy. Another recent, even more specific, approach is the Tornado language [32] for stochastic context-free grammars designed solely to parse RNA sequences.

Most of the work on domain-specific languages (DSL) for dynamic programming points out that using a DSL has a number of benefits [11], either in terms of better parser error handling, higher performance, or encapsulation from features not regarded as part of the DSL.

Designing a DSL written as part of the host language, provides a number of benefits as well, and strangely, one such benefit is being able to use features not provided by the DSL. Designing a language with a restricted set of features always poses the danger of having a potential user requiring *exactly* that feature which has not yet been made available. A direct embedding, on the other hand, simply does

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'12, September 9–15, 2012, Copenhagen, Denmark.
Copyright © 2012 ACM 978-1-4503-1054-3/12/09…$10.00

not have this problem. If something can be expressed in the host language, it is possible to use it in the DSL.

Another point in favor of staying within an established language framework is that optimization work done on the backend is automatically available as well. One of the points made by our work is that it is not required to move to a specialized DSL compiler to achieve good performance. Furthermore, certain features of the Haskell host language yield performance-increasing code generation (almost) for free.

What we can not provide is simplified error handling, but this becomes a non-issue for us as we specifically aim for Haskell-savvy users or those interested in learning the language.

We can also determine the appropriateness of embedding the ADPfusion DSL by looking at the guidelines given by Mernik et al. [27, Sec. 2.5.2] on "When and How to Develop Domain-Specific Languages". Most advantages of the embedded approach (development effort, more powerful language, reuse of infrastructure) are ours while some disadvantages of embedding (sub-optimal syntax, operator overloading) are easily dealt with using the very flexible Haskell syntax and standards set by ADP.

Our main contributions to dynamic programming on sequence data are:

- a generic framework separating grammatical structure, semantics of parses, and automatic generation of high-performance code using stream fusion;
- removal of the need for explicit index calculations: the combinator library takes care of corner cases and allows for linked index spaces;
- performance close to C with real-world examples from the area of computational biology;
- the possibility to use the library for more general parsing problems (beyond DP algorithms) involving production rules.

"Sneaking around *concatMap*" is a play on one of the ways how to write the Cartesian product of two sets. (Non-) terminals in production rules of grammars yield sets of parses. Efficient, generic treatment of production rules in an embedded DSL requires some work as we will explain in this work.

The outline of the paper is as follows: in the next section we introduce a simple parsing problem. Using this problem as an example, we rewrite it using ADP in Sec. 3, thereby showing the benefits of an embedded DSL. A short introduction to stream fusion follows (Sec. 4).

Armed with knowledge of both ADP and stream fusion, we write DP combinators that are compiled into efficient code in Sec. 5. We expand on ADPfusion with nested productions for more efficient code (Sec. 6).

Runtime performance of ADPfusion is given for two examples from RNA bioinformatics in Sec. 7 with comparisons to C programs.

Sections 8 and 9 are on specialized topics and we conclude with remarks on further work and open questions in Sec. 10.

2. Sum of digits

To introduce the problem we want to solve, consider a string of matched brackets and digits like $((1)(3))$. We are interested in the sum of all digits, which can simply be calculated by

```
sumD = sum ∘ map readD ∘ filter isDigit
readD x = read [x] :: Int.
```

The above algorithm works, because the structure of the nesting and digits plays no role in determining the semantics (sum of digits) of the input. For the sake of a simple introductory example, we

```
S → 1 | 2 | ... | 0  -- single digit
  |    ( S )         -- '(' substring ')'
  |     S S          -- split into substrings
```

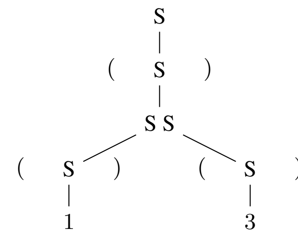

Figure 1. top: A context-free grammar for the nested-digits problem of Sec. 2. CFGs describe the structure of the search space. The semantics of a parse are completely separate.
bottom: Successful parse of the string $((1)(3))$. The semantics of this parse are 4 using the sum of digits semantics.

now assume that we have to solve this problem using a parser that implements the following three rules:

1. a digit may be read only if the string is of size 1, and the single character is a digit;
2. an outermost pair of brackets may be removed, these rules are then applied recursively to the remaining string;
3. the string may be split into two non-empty substrings to which these rules are applied recursively.

These rules can be written as a context-free grammar (CFG) and such a grammar is shown in Fig. 1 together with the successful parse of the string $((1)(3))$. As can be seen, the grammar describes the structure of parses, but make no mention of any kind of semantics. We simply know that a string is a *word* in the grammar but there is no meaning or semantics attached to it. This, of course, conforms to parsing of formal languages in general [15].

One way of parsing an input string is to use the CYK parser, which is a bottom-up parser using dynamic programming [15, Cha. 4.2].

We have chosen the example grammar of Fig. 1 for two reasons. First, it covers a lot of different parsing cases. The first production rule describes parsing with a single terminal on the right-hand side. The second rule includes a non-terminal bracketed by two terminal characters. The third rule requires parsing with two non-terminals. In addition, there are up to $n - 1$ different successful parses for the $n - 1$ different ways to split the string into two non-empty substrings. The third rule makes use of Bellman's principle of optimality. Parses of substrings are re-used in subsequent parses and optimal parses of substrings can be combined to form the optimal parse of the current string. This requires memoization.

Second, these seemingly arbitrary rules are actually very close to those used in RNA secondary structure prediction, being described in Sec. 7, conforming to hairpin structures, basepairing, and parallel structures. Furthermore, important aspects of dynamic programming (DP) and context-free grammars (CFGs) are included.

In the next section, we introduce algebraic dynamic programming (ADP), a domain-specific language (DSL) for dynamic programming over sequence data embedded in Haskell. Using the example from above, we will be able to separate structure and semantics of the algorithm.

```
-- signature
readDigit :: Char → S
bracket   :: Char → S → Char → S
split     :: S → S → S
h         :: [S] → [S]

-- structure or grammar
sd = (
readDigit <<< char ‘with‘ isDigit          |||
(bracket   <<< char ‘cThenS‘ sd ‘sThenC‘ char)
                ‘with‘ brackets            |||
split      <<< sd ‘nonEmpty‘ sd            ... h)

-- additional structure encoding
isDigit (i,j)  = j-i≡1 && Data.Char.isDigit (inp!j)
brackets (i,j) = inp!(i+1)≡'(' && inp!j≡')'
-- (!) is the array indexing operator: array ! index

-- semantics or algebra
readDigit c = read [c] :: Int
bracket l s r = s
split l r = l+r
h xs = if null xs then [] else [maximum xs]
```

Figure 2. Signature, structure (grammar), and semantics (algebra) of the sum-of-digits example (Sec. 2). The functions `cThenS`, `sThenC`, and `nonEmpty` chain arguments of terminals like `char` and non-terminals like `sd`. A special case is `with` that filters candidate parses based on a predicate. The symbolic functions `(<<<)`, `(|||)`, and `(...)` apply a function, allow different parses, and select a parse as optimal, respectively.

3. Algebraic dynamic programming

In this section we briefly recall the basic premises of algebraic dynamic programming (ADP) as described in Giegerich et al. [13]. We need to consider four aspects. The signature, defining an interface between a grammar and algebra, the grammar which defines the structure of the problem, its algebras each giving specific semantics, and memoization.

ADP makes use of *subwords*. A subword is a pair `(Int,Int)` which indexes a substring of the input. Combinators, terminals, and non-terminals all carry a subword as the last argument, typically as `(i,j)` with the understanding that $0 \leq i \leq j \leq n$ with n being the length of the input.

3.1 Signature

We have a finite alphabet $\mathcal{A}$ of symbols over which finite strings, including the empty string ϵ can be formed. In addition, we have a sort symbol denoted S. A signature Σ in ADP is a set of functions, with each function $f_i \in \Sigma$ having a specific type $f_i :: t_{i1} \rightarrow \cdots \rightarrow t_{in} \rightarrow S$ where $t_{ik} \in \{\mathcal{A}^+, S\}$. In other words, each function within the signature has one or more arguments, and each argument is a non-empty string over the alphabet or of the sort type which in turn is also the return type of each function. It is possible to use arguments which are derivative of these types, for instance providing the length of a string instead of the string itself. Such more specific cases are optimizations which do not concern us here.

The signature Σ includes an objective function $h : \{S\} \rightarrow \{S\}$. The objective function selects from a set of possible answers those which optimize a certain criterion like minimization or maximization.

3.2 Grammar

Grammars in ADP vaguely resemble grammatical descriptions as used in text books [15] or formal methods like the Backus-Naur form. They define the structure of the search space or the set of all parses of an input. In Fig. 2 we have the grammar for the sum of digits example of Fig. 1. The grammar has one non-terminal `sd`, which is equivalent to `S` of the context-free grammar. Furthermore, we have the three rules again. There are, however, major differences in how one encodes such a rule. Consider the third rule (`S → S S`) which now reads (`split <<<sd ‘nonEmpty‘ sd`) minus the left-hand side. From left to right, we recognize one of the function symbols of the signature `(split)`, a combinator function `(<<<)` that applies the function to its left to the argument on its right, the non-terminal `(sd)`, a second combinator function `(‘nonEmpty‘)` in infix notation, and finally the non-terminal again.

What these combinators do is best explained by showing their source, which also leads us back to why we want to “sneak around *concatMap*”.

```
infix 8 <<<
(<<<) :: (a→b) → (Subword→[a]) → Subword → [b]
(<<<) f xs (i,j) = map f (xs (i,j))

infixl 7 ‘nonEmpty‘
nonEmpty :: (Subword → [y→z])
         → (Subword → [y]) → Subword → [z]
nonEmpty fs ys (i,j) = [ f y
                       | k ← [i+1..j-1]
                       , f ← fs (i,k)
                       , y ← ys (k,j) ]
```

or equivalently

```
nonEmpty fs ys (i,j) = concatMap idx [i+1..j-1]
where
idx k = concatMap (λf → map f (ys (k,j))) (fs (i,k))
```

Each combinator takes a left and a right argument and builds a list from the Cartesian product of the two inputs, with `(<<<)` taking care of the scalar nature of the function to be mapped over all inputs. Importantly, all first arguments are partially applied functions which has a performance impact and hinders optimization.

The third argument of each combinator is the subword index that is threaded through all arguments. (Non-) terminals are functions from a subword to a list of values. For example `char` returns a singleton list with the i'th character of the input when given the subword $(i, i + 1)$ and an empty list otherwise.

```
char :: Subword → [Char]
char (i,j) = [inp!i | i+1≡j]
```

Similarly, the non-terminal `sd` is a function

```
sd :: Subword → [S]
sd = ... -- grammar as above
```

which can be memoized as required.

As a side note, in GHC Haskell, `concatMap` is not used in the implementation of list comprehensions, but the message stays the same: as we will see later in runtime measurements `concatMap` and list comprehensions are hard to optimize.

We complete the argument combinators with `cThenS` and `sThenC` (having the same type as `nonEmpty`):

```
infixl 7 ‘cThenS‘ ‘sThenC‘

cThenS fs ys (i,j) = [ f y | i<j, f ← fs (i,i+1)
                     , y ← ys (i+1,j) ]
```

```
sThenC fs ys (i,j) = [ f y | i<j, f ← fs (i,j-1)
                         , y ← ys (j-1,j) ]
```

We are still missing two combinators, `(|||)` and `(...)`. Both are simple, as ADP deals solely with lists, we just need to take care of the subword index in each case.

```
infixr 6 |||
(|||) :: (Subword→[a])→(Subword→[a])→Subword→[a]
(|||) xs ys (i,j) = xs (i,j) ++ ys (i,j)
infix 5 ...
(...) :: (Subword→[a])→([a]→[a])→Subword→[a]
(...) xs h (i,j) = h (xs (i,j))
```

There is an actual difference in the grammars of Fig. 1 and Fig. 2. In Fig. 1 the terminal symbols are explicit characters like '1' or '(', while in Fig. 2 `char` matches all single characters. We use this to introduce another useful combinator `(with)` that allows us to filter parses based on a predicate:

```
with :: (Subword→[a])→(Subword→Bool)→Subword→[a]
with xs p (i,j) = if p (i,j) then xs (i,j) else []
```

3.3 Algebra

We now have the grammar describing the structure of the algorithm. The function symbols of the signature are included and can be "filled" using one of several algebras describing the semantics we are interested in. Apart from the objective function `h`, all functions describe the semantics of production rules of the grammar. In our example above (Fig. 2) we either read a single digit (`readDigit`), keep just the sum of digits of the bracketed substring (`bracket`), or add sums from a `split` operation.

The objective function `(h)` selects the optimal parse according to the semantics we are interested in. In this case the maximum over the parses.

Another possibility is to calculate some descriptive value over the search space, say its total size. As an example, the *Inside-Outside* algorithm [23],[8, Ch. 10] adds up all probabilities generated by productions in a stochastic CFG instead of selecting one.

The specialty of ADP grammars is that they form tree grammars [10], [9, Sec. 2.2.1]. While they are analogous to context-free grammars, the right-hand sides of productions form proper trees with function symbols from the signature as inner nodes and terminal and non-terminal symbols at the leaves. For the example parse in Fig. 1 (bottom) this has the effect of replacing all non-terminal symbols `(S)` with function symbols from the signature. This also means that we can, at least in principle, completely decouple the generation of each parse tree from its evaluation. While the size of the search space might be prohibitive in practice, for small inputs, an exhaustive enumeration of all parses is possible. In an implementation, data constructors can be used as functions, while the objective function is simply the identity function. This allows us to print all possible parse trees for an input.

Each such tree has at its leaf nodes a sequence of characters, a word, from the alphabet: $w \in \mathcal{A}^*$. And for each given word w there are zero or more trees representing this word. If no tree for a given word exists, then the grammar can not parse that word and if more than one tree exists then the grammar is syntactically ambiguous in this regard. This kind of ambiguity is not problematic, typically even wanted, as the objective function can be used to evaluate different tree representations of a word w and return the optimal one.

3.4 Memoization

As noted in Sec. 3.2, non-terminals in grammars can be memoized. ADP introduces a function

`tabulated :: Int →(Subword →[a]) →Subword →[a]`, used as `sd = tabulated (length input) ( productions )`, that stores the answers for each subword in an array. Depending on the algorithm, other memoization schemes, or none at all, are possible. In general, memoization is required to make use of Bellman's principle of optimality and reduce the runtime of an algorithm from exponential to polynomial.

3.5 ADP in short

To summarize, algebraic dynamic programming achieves a separation of concerns. Parsing of input strings for a given grammar is delegated to the construction of candidates which are correct parses. Evaluation of candidates is done by specifying an evaluation algebra, of which there can be more than one. Selection from all candidates based on their evaluation is done by an objective function which is part of each evaluation algebra. Memoization makes the parsing process asymptotically efficient. Giegerich et al. [13] provide a much more detailed description than given here.

As our objective is to perform parsing, evaluation and selection more efficiently, we will, in the next sections, change our view of dynamic programming over sequence data to describe our approach starting with streams as a more efficient alternative to lists.

4. Stream fusion

We introduce the basics of stream fusion here. Considering that the ADPfusion library is based around applications of `map`, `flatten` (a variant of `concatMap`, more amenable to fusion), and `fold`, these are the functions described. For advanced applications, the whole range of fusible functions may be used, but those fall outside the scope of both this introduction to stream fusion and the paper in general. In addition, here and in the description of ADPfusion, we omit that stream fusion and ADPfusion are parametrized over Monads.

Stream fusion is a short-cut fusion system for lists [7], and more recently arrays [25], that removes intermediate data structures from complicated compositions of list-manipulating functions. Ideally, the final, fused, algorithm is completely free of any temporary allocations and performs all computations efficiently in registers, apart from having to access static data structures containing data. Stream fusion is notable for fusing a larger set of functions than was previously possible, including zipping and concatenating of lists.

Stream fusion is built upon two data types, `Stream` captures a function to calculate a single step in the stream and the seed required to calculate said step. A `Step` can indicate if a stream is terminated (`Done`). If not, the current step either `Yield`s a value or `Skip`s, which can be used for filtering. One other use of `Skip` is in concatenation of streams which becomes non-recursive due to `Skip` as well. Unless a stream is `Done`, each new `Step` creates a new seed, too.

```
data Stream a = ∃ s. Stream (s → Step a s) s
data Step a s = Done
              | Yield a s
              | Skip s
```

The point of representing lists as sequence co-structures is that no function on streams is recursive (except final *fold*s), permitting easy fusion to generate efficient code.

We construct a new stream of a single element using the `singleton` function. A singleton stream emits a single element `x` and is `Done` thereafter. Notably, the `step` function defined here is non-recursive.

```
singleton_S x = Stream step True where
  step True  = Yield x False
  step False = Done
```

Mapping a function over a stream is non-recursive as well, in marked contrast to how one maps a function over a list.

```
mapS f (Stream step s) = Stream nstp s where
  nstp s = case (step s) of
    Yield x s' → Yield (f x) s'
    Skip    s' → Skip s'
    Done       → Done
```

As a warm-up to stream-flattening, and because we need to concatenate two streams with (|||) anyway, we look at the stream version of (++).

```
(Stream stp1 ss) ++S (Stream stp2 tt) =
Stream step (Left ss) where
  step (Left s) = case s of
    Yield x s' → Yield x (Left s')
    Skip    s' → Skip    (Left s')
    Done       → Skip    (Right tt)
  step (Right t) = case t of
    Yield x t' → Yield x (Right t')
    Skip    t' → Skip    (Right t')
    Done       → Done
```

The `Left` and `Right` constructors encode which of the two streams is being worked on, while the jump from the first to the second stream is done via a (again non-recursive) `Skip`.

The `flatten` function takes three arguments: a function `mk` which takes a value from the input stream and produces an initial seed for the user-supplied `step` function. The user-supplied `step` then produces zero or more elements of the resulting stream for each such supplied value. Note the similarity to stream concatenation. `Left` and `Right` are state switches to either initialize a new substream or to create stream `Step`s based on this initial seed.

Again, it is important to notice that no function is recursive, the hand-off between extracting a new value from the outer stream and generating part of the new stream is done via `Skip (Right (mk a, t'))`.

```
flattenS mk step (Stream ostp s) = Stream nstp (Left s)
where
  nstp (Left t) = case (ostp s) of
    Yield a t' → Skip (Right (mk a, t'))
    Skip    t' → Skip (Left t')
    Done       → Done
  nstp (Right (b,t)) = case (step b) of
    Yield x s' → Yield x (Right (s',t))
    Skip    s' → Skip (Right (s',t))
    Done       → Left t
```

Finally, we present the only recursive part of the stream fusion, folding a stream to produce a final value.

```
foldS f z (Stream step s) = loop f z where
  loop f z = case (step s) of
    Yield x s' → loop (f z x) s'
    Skip    s' → loop z s'
    Done       → z
```

If such code is used to build larger functions like

```
foldS (+) 0 (flattenS id f (singletonS 10)) where
f x = if (x > 0)
      then Yield x (x-1)
      else Done
```

call-pattern specialization [31] of the constructors (`Yield`, `Skip`, `Done`) creates specialized functions for the different cases, and inlining merges the newly created functions, producing an efficient, tight loop. A detailed explanation can be found in Coutts et al. [7, Sec. 7] together with a worked example. The GHC compiler [36] performs all necessary optimizations.

5. Designing efficient combinators for dynamic programming

Algebraic dynamic programming is already able to provide asymptotically optimal dynamic programming recursions. A dynamic program written in ADP unfortunately comes with a rather high overhead compared to more direct implementations. Two solutions have been proposed to this problem. The first was translation of ADP code into C using the ADP Compiler [35] and the second a complete redesign providing a new language and compiler (GAP-L and GAP-C) [33]. Both approaches have their merit but partially different goals than ours. Here we want to show how to keep most of the benefits of ADP while staying *within* Haskell instead of having to resort to a different language.

We introduce combinators in a top-down manner, staying close to our introductory example of Fig. 2. An important difference is that functions now operate over stream fusion [7] streams instead of lists. This change in internal representation lets the compiler optimize grammar and algebra code much better than otherwise possible.

We indicate the use of stream fusion functions like $\texttt{map}_S$ with a subscript $_S$ to differentiate between normal list-based functions and stream fusion versions.

5.1 Combining and reducing streams

Two of the combinators, the choice between different productions (|||) and the application of an objective function, stay essentially the same, except that the type of `h` is now `Stream a →b`, instead of `[a] → [b]`. The objective function returns an answer of a scalar type, say `Int`, allowing for algorithms that work solely with unboxed types, or a vector type (like lists, boxed, or unboxed vectors). This gives greater flexibility in terms of what kind of answers can be calculated and choosing the best encoding, in terms of performance, for each algorithm.

```
infixl 7 |||
(|||) xs ys ij = xs ij ++S ys ij

infixl 6 ...
(...) stream h ij = h (stream ij)
```

In addition, the index is not a tuple anymore, but rather a variable `ij` of type `DIM2`. Instead of plain pairs `(Int,Int)` we use the same indexing style as the Repa [22] library. Repa tuples are inductively defined using two data types and constructors:

```
data Z = Z
data a :. b = a :. b
type DIM1 = Z :. Int
type DIM2 = DIM1 :. Int
```

The tuple constructor `(:.)` resembles the plain tuple constructor `(,)`, with `Z` as the base case when constructing a 1-tuple `(Z:.a)`. We can generalize the library to cover higher-dimensional DP algorithms just like the Repa library does for matrix calculations. It allows for uniform handling of multiple running indices which are represented as k-dimensional inductice tuples as well, increasing k by one for each new (non-) terminal. Using plain tuples would require nesting of pairs. Also, subwords are now of type `DIM2` instead of `(Int,Int)`.

5.2 Creating streams from production rules

As of now, we can combine streams and reduce streams to a single value or a set of values of interest. As streams expose many

optimization options to the compiler (cf. Sec. 4 and [7]), we can expect good performance. What is still missing is how to create a stream, given a production rule, in the first place. Rules such as `readDigit <<<char` with a single terminal or non-terminal to the right are the simplest to construct.

The combinator `(<<<)` applies a function to one or more arguments and is defined as:

```
infixl 8 <<<
(<<<) f t ij =
  mapS (λ(_,_,as) → apply f as) (streamGen t ij)
```

The `streamGen` function takes the argument `arg` on the right of `(f <<< arg)`, with `arg` of type `DIM2` $\rightarrow \alpha$, and the current subword index to create a stream of elements. If α is scalar (expressed as `DIM2` $\rightarrow$ `Scalar` β), the result is a singleton stream, containing just β, but α can also be of a vector type say $[\beta]$, in which case a stream of β arguments is generated, containing as many elements as are in the vector data structure.

We use a functional dependency to express[1] that the type of the stream `r` is completely determined by the type of the (non-) terminal(s) `t`.

```
class StreamGen t r | t → r where
  streamGen :: t → DIM2 → Stream r
```

The instance for a scalar argument (`DIM2` $\rightarrow$`Scalar` β) follows as:

```
instance StreamGen (DIM2 → Scalar β) (DIM2,Z:.Z,Z:.β)
```

delaying the actual implementation for now.

Streams generated by `streamGen` have as element type a triple of inductively defined tuples we call "stacks", whose stack-like nature is only a type-level device, no stacks are present during runtime.

The first element of the triple is the subword index, the second gives an index into vector-like data structures, while the third element of the triple holds the actual values. We ignore the second element for now, just noting that (non-) terminals of scalar type do not need indexing, hence `Z` as type and value of the index. Arguments are encoded using inductive tuples, and as we only have one argument to the right of `(<<<)`, the tuple is (`Z:.`α), as all such tuples or stacks (e.g. subword indices, indices into data structures, argument stacks) always terminate with `Z`.

The final ingredient of `(<<<)`, `apply`, is now comparatively simple to implement and takes an n-argument function `f` and applies it to n arguments (`Z:.`a_1`:.`$\cdots$`:.`a_n). We introduce a type dependency between the arguments of the function to apply and the arguments on the argument stack, using an associated type synonym.

```
class Apply x where
  type Fun x :: *
  apply :: Fun x → x

instance Apply (Z:.a1:. ··· :.an → r) where
  type Fun (Z:.a1:. ··· :.an → r)
    = a1 → ··· → an → r
  apply fun (Z:.a1:. ··· :.an) = fun a1 ··· an
```

5.3 Extracting values from (non-) terminals

As a prelude to our first stream generation instance (that we still have to implement) we need to be able to extract values from terminals and non-terminals. There are three classes of arguments that act as (non-) terminals. We have already encountered the type (`DIM2` $\rightarrow$`Scalar` β) for functions returning a single (scalar) value. A second class of functions yields multiple values of type β: (`DIM2` $\rightarrow$`Vector` β). In this case we do not have vector-valued arguments to but rather multiple choices from which to select. Finally, we can have data structures. A data structure can again store single (scalar) results or multiple results (vector-like) for each subword. For data structures, it will be necessary to perform an indexing operation (e.g. `(!)` is used for the default Haskell arrays) to access values for a specific subword.

The `ExtractValue` type class presented below is generic enough to allow many possible styles of retrieving values for a subword and new instances can easily be written by the user of the library.

We shall restrict ourselves to the instance (`DIM2` $\rightarrow$`Scalar` β). Instances for other common data structures are available with the library, including lazy and strict arrays of scalar and vector type.

The `ExtractValue` class itself has two associated types, `Asor` denoting the accessor type for indexing individual values within a vector-like argument and `Elem` for the type of the values being retrieved.

For, say, (`DIM2` $\rightarrow$`[`β`]`), a possible `Asor` type is `Int` using the list index operator `(!!)`, while the `Elem` type is β.

For scalar types, the `Asor` will be `Z` as there is no need for an index operation in that case.

The type class for value extraction is:

```
class ExtractValue cnt where
  type Asor cnt :: *
  type Elem cnt :: *
  extractStream
    :: cnt → Stream (Idx3 z,as,vs)
    → Stream (Idx3 z, as:.Asor cnt,vs:.Elem cnt)
  extractStreamLast
    :: cnt → Stream (Idx2 z,as,vs)
    → Stream (Idx2 z,as:.Asor cnt,vs:.Elem cnt)

type Idx3 z = z:.Int:.Int:.Int
type Idx2 z = z:.Int:.Int
```

`extractStream` and `extractStreamLast` are required to correctly handle subword indices with multiple arguments in productions. Their use is explained below, but note that `extractStream` accesses the 2nd right-most subword (k, l), while `extractStreamLast` accesses the rightmost (l, j) one. Consider the production

```
S → x y z
   i k l j
```

where `y` would be handled by `extractStream` and `z` by `extractStreamLast`, and `x` has already been handled at this point, its value is on the `Elem` stack.

Each function takes a stream and extends the accessor (`Asor`) stack with its accessor and the value (`Elem`) stack is extended with the value of the argument.

Now to the actual instance for (`DIM2` $\rightarrow$`Scalar` β):

```
instance ExtractValue (DIM2 → Scalar β) where
  type Asor (DIM2 → Scalar β) = Z
  type Elem (DIM2 → Scalar β) = β
  extractStream cnt s = mapS f s where
    f (z:.k:.l:.j,as,vs) =
    let Scalar v = cnt (Z:.k:.l)
    in  (z:.k:.l:.j,as:.Z,vs:.v)
  extractStreamLast cnt s = mapS f s where
    f (z:.l:.j,as,vs) =
    let Scalar v = cnt (Z:.l:.j)
    in  (z:.l:.j,as:.Z,vs:.v)
```

[1] Instead of type families for reasons explained in Sec. 9.

5.4 Streams for productions with one (non-) terminal

We can finish the implementation for streams of (DIM2 →Scalar β) arguments. The instance is quite similar to the `singleton` function presented in Sec. 4 but while `singleton` creates a single-element stream unconditionally we have to take care to only create a stream if the subword (`Z:.i:.j`) is legal. An illegal subword $i > j$ should lead to an empty stream.

```
instance
( ExtractValue (DIM2 → Scalar β)
) ⇒ StreamGen (DIM2 → Scalar β) (DIM2,Z:.Z,Z:.β)
  where
  streamGen x ij = extractStreamLast x
                   (unfoldr_S step ij)
  step (Z:.i:.j)
    | i≤j = Just ((Z:.i:.j,Z,Z), (Z:.j+1:.j))
    | otherwise = Nothing
```

In this case, we use the subword `ij` as seed. If the subword is legal, a stream with this subword and empty `(Z) Asor` and `Elem` stacks is created. The new seed is the *illegal* subword $(j + 1, j)$ which will terminate the stream after the first element.

We then immediately extend the stream elements using `extractStreamLast` which creates the final stream of type (`DIM2,Z:.Z,Z:.`β) by adding the corresponding accessor of type `Z` and element of type β as top-most element to their stack. With one argument, the only argument is necessarily the last one, hence the use of `extractStreamLast` instead of `extractStream`.

Using the construction scheme of only creating streams if subwords are legal, we effectively take care of all corner cases. Illegal streams (due to illegal subwords) are terminated before we ever try to extract values from arguments. This means that `ExtractValue` instances typically do not have to perform costly runtime checks of subword arguments.

5.5 Handling multiple arguments

We implement a single combinator (`nonEmpty`) as this is already enough to show how productions with any number (≥ 2) of arguments can be handled. In addition, `nonEmpty` has to deal with the corner case of empty subwords ($i = j$) on both sides. That is, its left and right argument receive only subwords of at least size one.

Recall that in ADP the first argument to each combinator turns out to be a partially applied function that is immediately given its next argument with each additional combinator. Partially applied functions, however, can reduce the performance of our code and make it impossible (or at least hard) to change the subword index space dependent on arguments to the left of the current combinator as the function would already have been applied to those arguments.

By letting `nonEmpty` have a higher binding strength than (`<<<`) we can first collect all arguments and then apply the corresponding algebra function. In addition, we need to handle inserting the current running index, `Asor` indices of the arguments, and `Elem` values for a later `apply`. Hence `nonEmpty` is implemented in a completely different way than in ADP:

```
infixl 9 ‘nonEmpty‘
xs ‘nonEmpty‘ ys = Box mk step xs ys where
  mk (z:.i:.j,vs,as) = (z:.i:.i+1:.j,vs,as)
  step (z:.i:.k:.j,vs,as)
    | k+1≤j = Yield (z:.i:.k  :.j,vs,as)
                    (z:.i:.k+1:.j,vs,as)
    | otherwise = Done
```

The `nonEmpty` combinator does, in fact, not combine the arguments `xs` and `ys` at all but only prepares two functions `mk` and `step`.

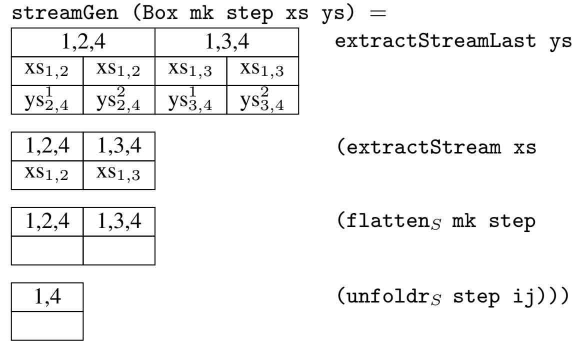

Figure 3. A stream from two arguments built step-wise bottom to top. First, a running index is inserted between the original subword $(1, 4)$ indices using `flatten`. Then, elements are extracted from the scalar argument `xs`. The vector-like argument `ys` yields two elements for each subword (indices [1] and [2]). (`step` as in Sec. 5.4)

These define the set of subwords (i, k) and (k, j) splitting the current subword (i, j) between `xs` and `ys`. Again, we make sure that any corner cases are caught. The first value for k is $i + 1$, after which k only increases. Hence `xs` is nonEmpty. In `step` we also stop creating new elements once $k + 1 > j$ meaning `ys` is never empty. Finally, should the initial subword (i, j) have size $j - i < 2$, the whole stream terminates immediately.

Of course, we are not constructing a stream at all but rather a `Box`. The implication is that two or more (non-) terminals in a production lead to nested boxes where `xs` is either another `Box` or an argument, while `ys` is always an argument. Furthermore `mk` and `step` are the two functions required by `flatten`. The `streamGen` function will receive such a nested `Box` data structure whenever two or more arguments are involved. The compiler can deconstruct even deeply nested boxes during compile time, enabling full stream fusion optimization for the production rule, completely eliminating *all* intermediate data structures just presented. We expose these optimizations to the compiler with `StreamGen` instances that are recursively applied during compilation.

5.6 Streams from productions with multiple arguments

Efficient stream generation requires deconstucting `Box`es, correct generation of subwords in streams, and extraction of values from arguments. This can be achieved with a `StreamGen` instance for `Box`es and an additional type class `PreStreamGen`.

These instances will generate the code shown in Fig. 3 (right).

The `StreamGen` instance fo the outermost `Box`

```
instance
  ( ExtractValue ys, Asor ys ∼ a, Elem ys ∼ v
  , PreStreamGen xs (idx:.Int,as,vs)
  , Idx2 undef ∼ idx
  ) ⇒ StreamGen (Box mk step xs ys)
                (idx:.Int,as:.a,vs:.v) where
  streamGen (Box mk step xs ys) ij
    = extractStreamLast ys
      (preStreamGen (Box mk step xs ys) ij)
```

handles the last argument of a production, extracting values using `extractStreamLast`. `PreStreamGen` instances handle the creation of the stream excluding the last argument recursively employing `preStreamGen`.

And we finally make use of `flatten`. This function allows us to create a stream and use each element as a seed of a substream when

adding an argument further to the right – basically on the way back up from the recursion down of the nested Boxes.

The type class PreStreamGen follows StreamGen exactly:

```
class PreStreamGen s q | s → q where
  preStreamGen :: s → DIM2 → Stream q
```

To handle a total of two arguments, including the last, this PreStreamGen instance is sufficient[2]:

```
instance
( ExtractValue xs, Asor xs ~ a, Elem xs ~ v
, Idx2 undef ~ idx
) ⇒ PreStreamGen (Box mk step xs ys)
                 (idx:.Int,as:.a,vs:.v) where
preStreamGen (Box mk step xs ys) ij
  = extractStream xs
    (flatten_S mk step
    (unfoldr_S step ij))
step (Z:.i:.j)
  | i≤j = Just ((Z:.i:.j,Z,Z), Z:.j+1:.j)
  | otherwise = Nothing
```

For three or more arguments we need a final ingredient. Thanks to overlapping instances (cf. Sec. 9.1 on overlapping instances) this instance

```
instance
( PreStreamGen (Box mkI stepI xs ys) (idx,as,vs)
( ExtractValue ys, Asor ys ~ a, Elem ys ~ v
, Idx2 undef ~ idx
) ⇒ PreStreamGen (Box mk step (Box mkI stepI xs ys) zs)
                 (idx:.Int,as:.a,vs:.v) where
preStreamGen (Box mk step box@(Box _ _ _ ys) zs) ij
  = extractStream ys
    (flatten_S mk step Unknown
    (preStreamGen box ij))
```

which matches two or more nested Boxes, will be used except for the final, innermost Box. Then, the above (more general) instance is chosen and recursion terminates.

As the recursion scheme is based on type class instances, the compiler will instantiate during compilation, exposing each flatten function to fusion. Each of those calculates subword sizes and adds to the subword stack, while Asor and Elem stacks are filled using extractStream and extractStreamLast, thereby completing the ensemble of tools required to turn production rules into efficient code.

5.7 Efficient streams from productions

Compared with ADP combinators (Sec. 3) we have traded a small amount of additional user responsibilities with the potential for enormous increases in performance.

The user needs to write an instance (of ExtractValue) for data structures not covered by the library or wrap such structures with (DIM2 $\rightarrow\alpha$) accordingly.

New combinators are slightly more complex as well, requiring the mk and step function to be provided, but again several already exist. Even here, the gains outweigh the cost as each combinator has access to the partially constructed subword, Asor, and Elem stack of its stream step. One such application is found in the RNAfold algorithm (Sec. 7.2) reducing the runtime from $O(n^4)$ to $O(n^3)$ as in the reference implementation.

[2] for type inference purposes, additional type equivalence terms are required for mk and step which are omitted here

6. Applying Bellman's principle locally

All major pieces for efficient dynamic programming are now in place. A first test with a complex real-world dynamic program unfortunately revealed disappointing results. Consider the following production in grammar form:

```
S → char string S string char
     i   i+1    k l   j-1   j
```

Two single characters (char) bracket three arguments of variable size. A stream generated from those five arguments is quadratic in size, due to two indices, k and l, with $i+1 \leq k \leq l \leq j-1$ with k (l) to the left (right) of S. We would like to evaluate the outer arguments (the char terminals) only once, but due to the construction of streams from left to right, the right-most argument between $(j-1, j)$ will be evaluated a total of $O(n^2)$ times. Depending on the argument, this can lead to a noticeable performance drain.

Two solutions present themselves: (i) a more complex evaluation of (non-) terminals or (ii) making use of Bellman's principle. As option (i) requires complex type-level programming, basically determining which argument to evaluate when, and option (ii) has the general benefit of rewriting productions in terms of other productions, let us consider the latter option.

If Bellman's principle holds, a problem can be subdivided into smaller problems that, when combined, yield the same result as solving the original problem, and each subproblem is reused multiple times.

If the above production has the same semantics under an objective function, as the one below, we can rewrite it, and benefit from not having to evaluate the right-most argument more than once.

```
S → char T char            T → string S string
     i  i+1 j-1  j              i+1     k l    j-1
```

We want to introduce another non-terminal (T) only conceptually, but translation into ADPfusion is actually quite easy. Given the original code

```
f <<< char ‘then‘ string ‘then‘ s ‘then‘ string
      ‘then‘ char ... h
```

the new nested version is

```
f <<< char ‘then‘
      (g <<< string ‘then‘ s ‘then‘ string ... h)
      ‘then‘ char ... h
```

This version still yields efficient code and the final char argument is evaluated just once. In terms of ADPfusion, bracketing and evaluation of subproductions (g <<< string ‘then‘ $\cdots$) is completely acceptable, the inner production has type (DIM2 $\rightarrow\alpha$), variants of which are available by default.

The availability of such an optimization will depend on the specific problem at hand and will not always be obvious. As the only changes are a pair of brackets and an inner objective function, changes are easily applied and a test harness of different input sequences can be used to determine equality of the productions with high certainty – even without having to *prove* that Bellman's principle holds. One particularly good option is to automate testing using QuickCheck [5] properties.

7. Two examples from RNA bioinformatics

In this section, we test the ADPfusion library using two algorithms from the field of computational biology. The Nussinov78 [29] grammar is one of the smallest RNA secondary structure prediction grammars and structurally very similar to our introductory example of Figs. 1 and 2. The second algorithm, RNAfold 2.0 [26] tries to find an optimal RNA secondary structure as well.

Both algorithms can be seen as variants of the CYK algorithm [15, Sec. 4.2]. The difference is that every word is part of the language and parsing is inherently syntactically ambiguous: every input allows many parses. By attaching semantics (say: a score or an energy), similar to the sum of digits semantics, the optimal parse is chosen.

We pit ADPfusion code against equivalent versions written in C. The `Nussinov78` grammar and algebra (Fig. 4) are very simple and we will basically measure loop optimization. `RNAfold 2.0` is part of an extensive set of tools in the ViennaRNA package [26]. The complicated structure and multiple energy tables lead to a good "real-world" test.

All benchmarks are geared toward the comparison of `C` and ADPfusion in Haskell. Legacy ADP runtimes are included to point out how much of an improvement we get by using strict, unboxed arrays and a modern fusion framework.

The legacy ADP version of RNAfold is not directly compatible with `RNAfold 2.0` (`C` and ADPfusion). It is based on an older version of RNAfold (1.x) which is roughly 5% – 10% faster than 2.0.

We do not provide memory benchmarks. For `C` vs. ADPfusion the requirements for the DP tables are essentially the same, while legacy ADP uses boxed tables and always stores lists of results with much overhead.

The Haskell versions of `Nussinov78` and `RNAfold 2.0` have been compiled with `GHC 7.2.2` and `LLVM 2.8`; compilation options: `-fllvm -Odph -optlo-O3`. The `C` version of `Nussinov78` was compiled using `GCC 4.6` with `-O3`. The ViennaRNA package was compiled with default configuration, including `-O2` using `GCC 4.6`. All tests were done on an Intel Core i7 860 (2.8 GHz) with 8 GByte of RAM.

7.1 Nussinov's RNA folding algorithm

The algorithm by Nussinov et al. [29] is a very convenient example algorithm that is both: simple, yet complex enough to make an interesting test. A variant of the algorithm in ADP notation is shown in Fig. 4 together with its CFG. The algorithm expects as input a sequence of characters from the alphabet $\mathcal{A} = \{ACGU\}$. A *canonical basepair* is one of the six (out of 16 possible) in the set $\{AU, UA, CG, GC, GU, UG\}$. The algorithm maximizes the number of paired nucleotides with two additional rules.

Two nucleotides at the left and right end of a subword (i, j) can pair only if they form one of the six canonical pairs. For all pairs (k, l) it holds that neither $i < k < j < l$ nor $k < i < l < j$ and if $i == k$ then $j == l$. Any two pairs are juxtaposed or one is embedded in the other.

The mathematical formulation of the recursion implied by the grammar and pairmax semantics in Fig. 4 is

$$S[i,j] = \max \begin{cases} 0 & i == j \\ S[i+1,j] & i < j \\ S[i,j-1] & i < j \\ S[i+1,j-1]+1 & \text{if (i,j) pairing} \\ \max_{i<k<j} S[i,k] + S[k+1,j] & . \end{cases}$$

As there is only one non-terminal `S` (respectively DP matrix `s`) and no scoring or energy tables are involved, the algorithm measures mainly the performance for three nested loops and accessing one array.

As Fig. 5 clearly shows, we reach a performance within ×2 of `C` for moderate-sized input. The `C` version used here is part of the `Nussinov78` package available online[3].

[3] `Nussinov78` hackage library: `http://hackage.haskell.org/package/Nussinov78`

```
-- signature
nil   :: S
left  :: Char → S → S
right :: S → Char → S
pair  :: Char → S → Char → S
split :: S → S → S
h     :: Stream S → S

-- structure or grammar
                s = (
S →  ε          nil   <<< empty                |||
  |  bS         left  <<< base −~~ s           |||
  |   Sb        right <<<          s ~~− base  |||
  |  bSb        pair  <<< base −~~ s ~~− base
  |  S S                  ‘with‘ pairing       |||
                split <<<       s +~+ s        ... h)

-- semantics or algebra
nil = 0                     pair a s b = s+1
left b s   = s              split l r  = l+r
right  s b = s              h xs = maximum_S xs
```

Figure 4. Top: The signature Σ for the `Nussinov78` grammar. The functions `nil, left, right, pair,` and `split` build larger answers `S` out of smaller ones. The objective function `h` transforms a stream of candidate answers, e.g. by selecting only the optimal candidate.
Center left: The context-free grammar `Nussinov78`. Character `b` $\in \mathcal{A} = \{$A,C,G,U$\}$.
Center right:The `Nussinov78` algorithm in ADPfusion notation with `base :: DIM2 →Char`. This example was taken from [14]. Compared to the CFG notation, the evaluation functions are now explicit as is the non-empty condition for the subwords of `split`. The `(−~~)` combinator allows a size-one subword to its left (cf. `cThenS` in Fig. 2). Its companion `(~~−)` to the right (`sThenC`). The `(+~+)` combinator enforces non-empty subwords (`nonEmpty`).
Bottom: Pairmax algebra (semantics); maximizing the number of basepairs. In `pair`, it is known that `a` and `b` form a valid pair due to the `pairing` predicate of the grammar.

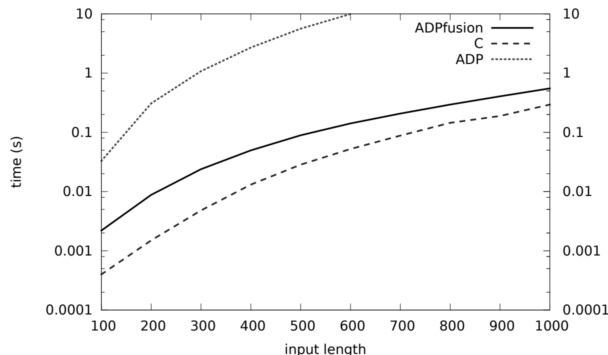

Figure 5. Runtime in seconds for different versions of the `Nussinov78` algorithm. The `Nussinov78` algorithm accesses only one DP matrix and no "energy tables". The comparatively high runtime for the ADPfusion code for small input is an artifact partially due to enabled backtracking.

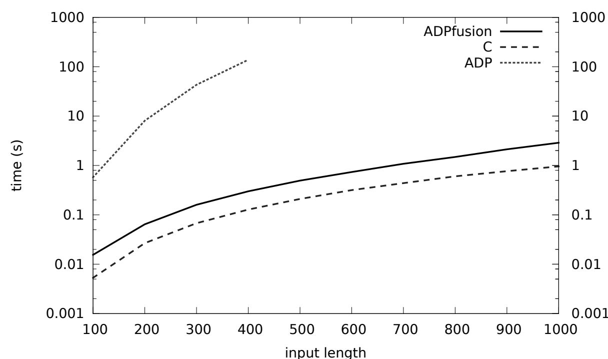

Figure 6. Runtime in seconds for different implementations of the `RNAfold 2.0` algorithm for random input of different length. The highly optimized `C` code is used by the official ViennaRNA package. ADPfusion is the code generated by our library. For illustrative purposes, ADP is the performance of the original Haskell implementation of the older `RNAfold 1.x` code.

An algorithm like `Nussinov78` is, however, not a good representative of recent developments in computational biology. Modern algorithms, while still adhering to the basic principles formulated by Nussinov et al. [29], use multiple DP matrices and typically access a number of additional tables providing scores for of different features. The `RNAfold 2.0` algorithm, described next, is one such algorithm.

7.2 RNAfold

The ViennaRNA package [16, 26] is a widely used state-of-the-art software package for RNA secondary structure prediction. It's newest incarnation is among the top programs in terms of prediction accuracy and one of the fastest. It provides an interesting target as it has been optimized quite heavily toward faster prediction of results. Compared to other programs, speed differences of $\times 10$ to $\times 100$ in favor of `RNAfold 2.0` are not uncommon [26].

The complete ViennaRNA package provides many different algorithms which makes it impractical to re-implement the whole package in Haskell. We concentrate on the minimum-free energy prediction part, which is the most basic of the offered algorithms.

We refrain from showing the ADPfusion version of the grammar. A version of RNAfold using recursion and diagrams for visualization is described in [2] and the ADPfusion grammar itself can be examined online[4].

We do, however, give some statistics. The grammar uses 4 non-terminals, three of which are interdependent while the fourth is being used to calculate "exterior" structures and only $O(n)$ matrix cells are filled instead of $O(n^2)$ as for the other three tables. A total of 17 production rules are involved and 18 energy tables. One production has an asymptotic runtime of $O(n^2)$ for each subword yielding a total runtime of $O(n^4)$. By restricting the maximal size for two linear-size subwords in the grammar to at most 30, the final runtime of RNAfold is bounded by $O(n^3)$. This restriction is present in both the `C` reference version and the ADPfusion grammar where we make use of a combinator that restricts the maximal subword size based on subword sizes calculated by another combinator, thus giving us the required restriction.

[4] `RNAFold` hackage library: `http://hackage.haskell.org/package/RNAFold`

Given inputs of size 100 (nucleotides) or more, ADPfusion code is efficient enough to get within $\times 2$ – $\times 3$ of the `C` implementation. Fig. 6 shows runtimes for legacy ADP, ADPfusion, and `C` code.

8. Backtracking and algebra products

ADP introduced the concept of algebra products. A typical dynamic programming algorithm requires two steps: a forward step to fill the dynamic programming matrices and a backward or backtracking step to determine the optimal path from the largest input to the smallest sub-problems. For a CYK parser, the forward step determines if a word is part of the language while the backward step yields the parse(s) for this word.

This forces the designer of a DP algorithm to write the recurrences twice, and keep the code synchronized as otherwise subtle bugs can occur. Algebra products "pull" the backward step into the forward step. Considering the case of the optimal path and its backtrace, one writes `(opt ***backtrace)`, where `opt` is the algebra computing the score of the optimal answer, while `backtrace` is its backtrace, and `(***)` the algebra product operation. This yields a new algebra that can be used as any other.

It has the effect of storing with each optimal result the description of how it was calculated or some information derived from this description. This is conceptually similar to storing a pointer to the cell(s) used for the calculation of the optimal result.

The algebra product is a very elegant device that allows for simple extension of algorithms with proper separation of ideas. A backtrace does not have to know about scoring schemes as each answer for the first argument of `(***)` is combined with exactly one answer of the second argument. Adding, say, sub-optimal results requires a change only to `opt` to capture more than one result, while co-optimal results are automatically available from the ADP definition of the algebra product.

The algebra product as used in ADP is, unfortunately, a problematic device to use in practice. While it allows for a simple design of algorithms and removes another source of potential bugs, it comes with a high runtime cost.

Consider an algorithm that calculates a large number of co- or sub-optimal results, like the `Nussinov78` algorithm in backtracking.

Standard implementations calculate the DP matrices in the forward step and then enumerate all possible backtraces within a certain range. The forward step does not change compared to just asking for the optimal result. The backward step, while tedious to get right, only has to deal with one backtrace at a time – unless they all have to be stored. ADP, on the other hand, stores *all* backtraces within its DP matrices. The memory cost is much higher as all answers – and all answers to sub-problems – that pass the objective function are retained within the matrices.

In addition, we can not use strict, unboxed arrays of Ints (or Floats or Doubles) if we store backtraces directly in the DP matrices.

For ADPfusion we prefer to have an explicit backtrace step. As a consequence, the programmer is faced with a slightly bigger task of defining the forward algebra and the backward algebra separately instead of just using the algebra product, but this is offset by the gains in runtime and memory usage. One can even use a version of the algebra product operation in the backward step to keep most of its benefits. In this case, the use of the algebra product becomes quite harmless as we no longer store each answer within the matrices. In terms of absolute runtime, this approach works out favorably as well. The costly forward phase (for RNAfold: $O(n^3)$) is as efficient as possible, while the less costly backtracking (for RNAfold: $O(n^2 * k)$, with k the number of backtracked results) uses the elegant algebra product device.

9. Technical details

9.1 Functional dependencies vs. type families

Type families [3] are a replacement for functional dependencies [21]. As both approaches provide nearly the same functionality, it is a good question why this library requires both: type families and functional dependencies. The functions to extract values from function arguments, collected in the type class `ExtractValue`, are making use of associated type synonyms as this provides a (albeit subjectively) clean interface.

The stream generation system, using the `StreamGen` and `PreStreamGen` type classes, is based on functional dependencies. The reasons are two-fold: (i) the replacement using type families does not optimize well, and (ii) functional dependencies allow for overlapping instances.

The type family-based version[5] of the ADPfusion library does not optimize well. Once a third argument, and hence nested `Boxes` come into play, the resulting code is only partially optimized effecting performance by a large factor. This seems to be due to insufficient compile-time elimination of `Box` data constructors. This problem is currently under investigation.

Using a fixed number of instances, say up to 10, would at best be a stop-gap measure since this restricts the user of the library to productions of at most that many arguments and leads to highly repetitive code.

As functional dependencies allow unlimited arguments, require only overlapping instances, and consistently produce good code, they are the better solution for now even though they are, in general, not well received[6].

9.2 Efficient memoization

The ADPfusion library is concerned with optimizing production rules independent of underlying data structures, lazyness, and boxed or unboxed data types. The author of a DP algorithm may choose the data structure most suitable for the problem and by giving an `ExtractValue` instance makes it compatible with ADPfusion. If priority is placed on performance, calculations can be performed in the ST or IO monad. The `PrimitiveArray`[7] library provides a set of unboxed array data structures that have been used for the algorithms in Sec. 7 as boxed data structures cost performance.

When first writing a new DP algorithm, lazy data structures can be used as this frees the programmer from having to specify the order in which DP tables (or other data structures) need to be filled. Once a proof-of-concept has been written, only small changes are required to create an algorithm working on unboxed data structures.

10. Conclusion and further work

High-level, yet high-performance, code is slowly becoming a possibility in Haskell. Projects like DPH [4] and Repa [22] show that one does not have to resort to unsightly low-level (and/or imperative-looking) algorithms anymore to design efficient algorithms. Furthermore, we can reap the benefits of staying within a language and having access to libraries and modern compilers compared to moving to a domain-specific language and its own compiler architecture.

The ability to write ADP code and enjoy the benefits of automatic fusion and compiler optimization are obvious as can be shown by the improvements in runtime as described in Sec. 7. Furthermore, one can design dynamic programming algorithms with the ease provided by ADP [10] and seamlessly enable further optimizations like strict, unboxed data structures, without having to rewrite the whole algorithm, or having to move away from Haskell.

With this new high-performance library at hand, we will redesign several algorithms. Our Haskell prototype of `RNAfold 2.0` allows us to compare performance with its optimized `C` counterpart. `RNAwolf` [18] is an advanced folding algorithm with a particularly complicated grammar including nucleotide triplets for which an implementation is only available in Haskell. `CMCompare` [17] calculates similarity measures for a restricted class of stochastic context-free grammars in the biological setting of RNA families.

Some rather advanced techniques that have become more appreciated in recent years (stochastic sampling of RNA structures [28] being one recent example) can now be expressed easily and with generality.

The ADP homepage [14] contains further examples of dynamic programming algorithms, as well as certain specializations and optimizations which will drive further improvements of this library. Of particular interest will be dynamic programming problems *not* in the realm of computational biology in order to make sure that the library is of sufficient generality to be of general usefulness.

The creation of efficient parsers for formal grammars, including CYK for context-free languages, is one such area of interest. Another are domain-specific languages that have rule sets akin to production rules in CFGs but do not require dynamic programming.

The ability to employ monadic combinators, which are available in the library, will be of help in many novel algorithmic ideas. We ignored the monadic aspect, but the library is indeed completely monadic. The non-monadic interface hides the monadic function application combinator `(#<<)`, nothing more. This design is inspired by the `vector`[8] library.

Coming back to the title of "sneaking around *concatMap*", we can not claim complete success. While we have gained huge improvements in performance, the resulting library is rather heavy-weight (requiring both, functional dependencies and type families, and by extension, overlapping, flexible, and undecidable instances). Unfortunately, we currently see no way around this. As already pointed out in the stream fusion paper [7, section 9], optimizing for `concatMap` is not trivial. Furthermore, we would need optimizations that deal well with partially applied functions to facilitate a faithful translation of ADP into high-performance code.

Right now, results along these lines seem doubtful (considering that the stream fusion paper is from 2007) to become available soon. In addition, our view of partitioning a subword allows us to employ certain specializations directly within our framework. We know of no obvious, efficient way of implementing them within the original ADP framework. The most important one is the ability to observe the index stack to the left of the current combinator making possible the immediate termination of a stream that fails definable criteria like maximal sum of sub-partition sizes.

The code generated by this library does show that we have achieved further separation of concerns. While algebraic dynamic programming already provides separation of grammar (search space) and algebra (evaluation of candidates and selection via objective function) as well as asymptotic optimization by partial tabulation, we can add a further piece that is very important in practice – optimization of constant overhead. While the application of Bellman's principle still has to happen on the level of the grammar and by proof, all *code optimization* is now moved into the ADPfusion library.

The ADPfusion library itself depends on low-level stream optimization using the stream fusion work [7, 25] and further code optimization via GHC [36] and LLVM [24]. Trying to expose cer-

[5] github: branch `tf`

[6] cf. "cons" on overlap: `http://hackage.haskell.org/trac/haskell-prime/wiki/OverlappingInstances`

[7] `http://hackage.haskell.org/package/PrimitiveArray`

[8] `http://hackage.haskell.org/package/vector`

tain compile-time loop optimizations either within ADPfusion or the stream fusion library seems very attractive at this point as does the potential use of modern single-instruction multiple-data mechanisms. Any improvements in this area should allow us to breach the final $\times 2$ gap in runtime but we'd like to close this argument by pointing out that it is now *easy* to come very close to hand-optimized dynamic programming code.

Availability

The library is BSD3-licensed and available from hackage under the package name ADPfusion: `http://hackage.haskell.org/package/ADPfusion`. The git repository, including the type families (`tf`) branch, is available on github: `https://github.com/choener/ADPfusion`.

Acknowledgments

The author thanks Robert Giegerich and the Practical Computer Science group at Bielefeld University (ADP), Ivo Hofacker (dynamic programming), Roman Leshchinskiy (`vector` library, fusion, high-performance Haskell), and his family for letting him design, code and (mostly) finish it during the 2011-12 winter holidays. Several anonymous reviewers have provided detailed and valueable comments for which I am very thankful.

This work has been funded by the Austrian FWF, project "SFB F43 RNA regulation of the transcriptome"

References

[1] R. E. Bellman. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, 38(8):716–719, 1952.

[2] A. F. Bompfünewerer, R. Backofen, S. H. Bernhart, J. Hertel, I. L. Hofacker, P. F. Stadler, and S. Will. Variations on RNA folding and alignment: lessons from Benasque. *Journal of Mathematical Biology*, 56(1):129–144, 2008.

[3] M. M. Chakravarty, G. Keller, and S. Peyton Jones. Associated Type Synonyms. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP'05, pages 241–253. ACM, 2005.

[4] M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP'07, pages 10–18. ACM, 2007.

[5] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP'00, pages 268–279. ACM, 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT press, 2001.

[7] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 315–326. ACM, 2007.

[8] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge Univ. Press, 1998.

[9] R. Giegerich and C. Höner zu Siederdissen. Semantics and Ambiguity of Stochastic RNA Family Models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(2):499–516, 2011.

[10] R. Giegerich and C. Meyer. Algebraic Dynamic Programming. In *Algebraic Methodology And Software Technology*, volume 2422, pages 243–257. Springer, 2002.

[11] R. Giegerich and P. Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1603–1609. ACM, 2006.

[12] R. Giegerich, C. Meyer, and P. Steffen. Towards a Discipline of Dynamic Programming. *Informatik bewegt, GI-Edition-Lecture Notes in Informatics*, pages 3–44, 2002.

[13] R. Giegerich, C. Meyer, and P. Steffen. A Discipline of Dynamic Programming over Sequence Data. *Science of Computer Programming*, 51(3):215–263, 2004.

[14] R. Giegerich et al. Algebraic Dynamic Programming Website. `http://bibiserv.techfak.uni-bielefeld.de/adp/`, 2004.

[15] D. Grune and C. J. Jacobs. *Parsing techniques: a practical guide*. Springer-Verlag New York Inc, 2008.

[16] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte für Chemie/Chemical Monthly*, 125(2):167–188, 1994.

[17] C. Höner zu Siederdissen and I. L. Hofacker. Discriminatory power of RNA family models. *Bioinformatics*, 26(18):453–459, 2010.

[18] C. Höner zu Siederdissen, S. H. Bernhart, P. F. Stadler, and I. L. Hofacker. A folding algorithm for extended RNA secondary structures. *Bioinformatics*, 27(13): 129–136, 2011.

[19] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 1–55. ACM, 2007.

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

[21] M. P. Jones. Type Classes with Functional Dependencies. *Programming Languages and Systems*, pages 230–244, 2000.

[22] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 261–272. ACM, 2010.

[23] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech & Language*, 4(1):35–56, 1990.

[24] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[25] R. Leshchinskiy. Recycle Your Arrays! *Practical Aspects of Declarative Languages*, pages 209–223, 2009.

[26] R. Lorenz, S. H. Bernhart, C. Höner zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(26), 2011.

[27] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[28] M. Nebel and A. Scheid. Evaluation of a sophisticated SCFG design for RNA secondary structure prediction. *Theory in Biosciences*, 130:313–336, 2011. ISSN 1431-7613.

[29] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for Loop Matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[30] B. O'Sullivan, D. B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2009.

[31] S. Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 327–337. ACM, 2007.

[32] E. Rivas, R. Lang, and S. R. Eddy. A range of complex probabilistic models for RNA secondary structure prediction that includes the nearest-neighbor model and more. *RNA*, 18(2):193–212, 2012.

[33] G. Sauthoff, S. Janssen, and R. Giegerich. Bellman's GAP - A Declarative Language for Dynamic Programming. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP'11, pages 29–40. ACM, 2011.

[34] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Co., Inc., 1983.

[35] P. Steffen. *Compiling a domain specific language for dynamic programming*. PhD thesis, Bielefeld University, 2006.

[36] The GHC Team. The Glasgow Haskell Compiler (GHC). `http://www.haskell.org/ghc/`, 2012.

Chapter 9

Outlook

The previous chapters contain a selection of scientific works exploring three topics in bioinformatics: (i) genome-scale search for non-coding RNA with two papers, (ii) prediction of extended RNA secondary structures, and (iii) performance optimization of a domain-specific language for dynamic programming. In this chapter, some further topics are discussed. These range from research based on the presented ideas to future areas of exploration. In particular, avenues of unification and generalization of ideas, as well as how these topics are tied together, are explored.

9.1 Generalizing the Language of Grammars

The key aspect of the domain-specific language for formal grammars presented in Chapter 8 was to achieve performance close to `C` while providing a high-level programming environment. This was, in fact, achieved. Using `ADPfusion` it is possible to write RNA bioinformatics algorithms on single-sequence inputs using a high-level style of programming. The framework and the Haskell compiler turn high-level code into efficient code without the user having to think about the details of this automatic conversion. The `RNAwolf` algorithm can now be re-implemented in `ADPfusion` just like the examples given in Chapter 8, `Nussinov78` and `RNAfold`. As writing a large set of rules (or recursions in other terminology) has now become easier, implementing the full extended model for multibranched loops, as sketched in Chapter 7, Fig. 4 becomes feasible.

In this way, the advancements in Chapter 8 actively encourage exploring new and more complicated RNA structure spaces, by containing the complexity of writing algorithms like `RNAwolf`.

Multi-tape Grammars

The `Nussinov78`, `RNAfold`, and `RNAwolf` algorithms have in common that they all predict the RNA secondary structure of a *single sequence*. Using `ADPfusion`, one can also write algorithms that accept *multiple sequences* as input, using the idea of multi-tape grammars. This includes algorithms for a fixed number of inputs, like the Sankoff algorithm (Sankoff,

1985) for structural alignment of two RNA sequences, as well as algorithms operating on a variable number of sequences as used in `RNAalifold` (Bernhart et al., 2008). Other algorithms require an alphabet consisting of more complex "characters". In natural language processing, the alignment of words in a sentence is important. By allowing more complex terminal symbols, a wide range of algorithms can be implemented quickly and efficiently.

This flexibility to encode multiple inputs and inputs of complex types efficiently is a clear advantage over similar frameworks. A particularly interesting question is how to properly encode for *non-linear* input such as tree-like data structures.

Extending `ADPfusion` to allow for such additional abstraction of algorithms does not yield performance penalties. In fact, `ADPfusion` has gained in raw speed since its initial publication. Some of these performance considerations are being detailed below.

Heterogeneous Index Spaces

Further generalization options are possible due to the use of shape polymorphism. Keller et al. (2010) defined inductive tuples, named `Shape`s to provide an efficient encoding of the index space of arrays. A `Shape` is a recursively defined index, where each dimension can be chosen from another domain. The shape of the lookup table for stacked pairs in `RNAfold` can be encoded as $\mathbb{B}^4$ with $\mathbb{B} = \{\texttt{ACGU}\}$, while the memoization tables for the same algorithm use the space $\mathbb{N} \times \mathbb{N}$. A length-dependent, and closing-pair dependent hairpin could similarly be encoded using the index space $\mathbb{B}^2 \times \mathbb{N}$.

For RNA folding, one typically requires upper-triangular memoization matrices as only cells (i, j) with $i \leq j$ need to be filled. Defining a new type of *subword*s makes it possible to create such an index. Used in `ADPfusion`, thusly defined subwords not only explicitly forbid illegal subwords $j < i$, they also, transparently for the user, create memoization tables that are of upper-triangular form, requiring half the amount of memory, square tables would require.

Mathematically speaking, this is not surprising and just states the domain of the index set. Computationally however, being able to use heterogeneous index spaces, define new basic elements of theses space (like $\mathbb{B}$), and combine them freely, allows `ADPfusion` to work generically over all user-defined index spaces.

9.2 Performance, Automatic Parallelization, and SIMD extensions

Haskell is an unusual choice of programming language, considering that `C/C++` for performance reasons and various scripting languages as "scripting glue" seem to be prevalent in the computational biology community. One of the problems in choosing a high-level language is efficiency. Traditional design of Haskell programs, especially laziness (which entails boxing of all data), leads to performance losses[1] in the range of $\times 60 - \times 100$ for dynamic programs. See, for example, Fig. 5 and Fig. 6 in Chapter 8 for `Nussinov78` and

[1] Performance loss is highly dependent on the algorithm. This is why Sec 4.1.2 mentions performance losses starting at only $\times 1.25$.

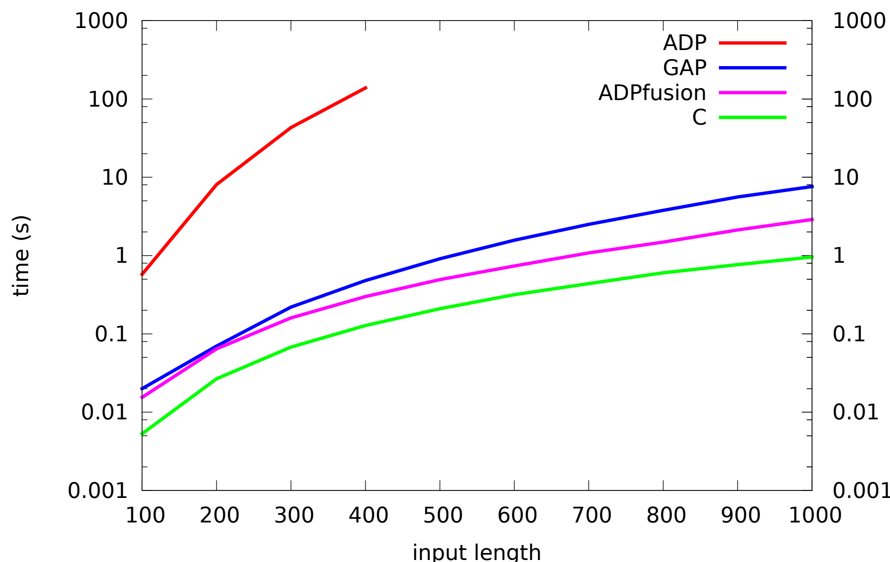

Figure 9.1: Post-ICFP'12 (autumn 2012) runtimes for `RNAfold`. The implementation in `C` provides the performance base line. Haskell `ADP` clearly shows the infeasibility of writing high-performance code in the original framework which used neither fusion nor strict memoization tables. The `GAP` compiler produces code within $\times 8$ of the `C` base line. `ADPfusion` comes within $\times 3$ of `C`.

`RNAfold` implementations in ADP (Giegerich and Meyer, 2002). Fortunately, the past 3–5 years have led to very impressive improvements in terms of efficient code generation by the compiler. Of utmost importance for the `ADPfusion` work was the introduction of stream fusion by Coutts et al. (2007) which paved the way for use of the stream fusion modules in the `vector` library.

In a comparison between the native `C` implementation of `RNAfold` and several reimplementations in "`ADP`-like" languages, including the `GAP language`, `ADPfusion` shows performance second only to the native implementation. The comparison in Fig. 9.1 is done on typical inputs of length 100 – 1000 and clearly shows that high performance and high-level implementations in a functional language are not at odds with each other.

All of the implementations in this comparison use only a single thread, they are not parallelized to exploit the availability of multiple cores as found in modern CPUs. Instruction-level parallelism (SIMD instructions) is ignored as well.

Parallelization of Dynamic Programming Algorithms

Parallelization of algorithms is hard as there are a number of obstacles that have to be overcome. Four important aspects are: (i) One first has to determine which parts of an algorithm can be parallelized. (ii) The parallelized version of an algorithm often looks quite different than the single-core version, and is implemented differently than the single-core version. One has to maintain *two* versions of the algorithm. (iii) Race conditions

need to be prevented. (iv) Parallelized code should actually be faster than serialized code.

All these aspects can be addressed efficiently in the `ADPfusion` framework.

(i): To discover parallelization opportunities it is advantageous to consider multiple classes of algorithms at the same time. Common to RNA structure prediction (minimum free energy, partition function) and RNA family model based scans is the use of upper-triangular memoization tables. Even though the former algorithms use a small, fixed set of tables and the latter a large, model-dependent set, they share some basic rules on how the memoization tables are filled. Filling begins at the main diagonal(s) and continues to the upper-right corner(s).

Typically, all elements on a single diagonal are independent of each other. With independence comes opportunity for parallelization. As a common pattern, it has to be identified only once and can then be shared as a function among many algorithms.

(ii): Algorithms developed in the style of `ADP` separate grammar, algebra, and the table filling scheme. Parallelization can be done on the level of the filling scheme alone and requires no changes to either the grammar or the algebra. As each grammar and algebra are now shared between the serial and parallel version, only one version of each algorithm needs to be maintained. Furthermore, as mentioned, a whole class of algorithms can share one filling scheme further reducing any chance of code duplication.

(iii): Race conditions can easily occur in parallelized dynamic programs. This happens if one parallelized calculation requests a value from a memoization table that still has to be computed by another parallelized calculation. Haskell has a system in place that prevents race conditions but this system requires laziness. Laziness, unfortunately, costs more in performance than can be gained by parallelization.

Thus parallelized code still has to use the efficient filling scheme based on methods mentioned in Sec. 4.1.2. Done wrongly, this scheme can introduce race conditions in parallel code. In general, race conditions are very hard to detect but Haskell provides convenient ways to test for these conditions.

Such a test algebra is much simpler than the scoring algebras that are normally used and easily implemented by the user. Each production rule with only terminals on the right-hand side (RHS) is evaluated to a boolean `True`. Production rules with non-terminals evaluate to a logical `AND` of all non-terminals. The memoization tables are initialized to `False`. If, after parallel calculation, any `False` remains in the table, a race condition has been localized. Using Template Haskell (Sheard and Peyton Jones, 2002), creating such an algebra can be automated. Together with QuickCheck (Claessen and Hughes, 2000), testing a grammar for race conditions can be automated as well.

(iv): The performance aspect requires careful creation of parallelized memoization table filling code that is compatible with the `ADPfusion` framework and respects the other three aspects. The `Repa` library (Keller et al., 2010) is a high-performance, parallelized, multi-core library for numerical calculations. It works on regular shape-polymorphic arrays similar to those used by `ADPfusion`. In addition, `ADPfusion` actually uses the `Shape`s defined by `Repa`. Both libraries are based on the `vector` library. `Repa` also provides a sophisticated framework to guide block wise parallelization which significantly simplifies defining the correct fill order.

This makes it possible to use the parallelization guidance framework developed for `Repa` in `ADPfusion`. By this feat, most of the burden of developing efficient parallel code is already done as all algorithms written in `ADPfusion` can be parallelized in this way – as long as the algorithm itself can be parallelized.

Together, these considerations make it feasible to extend `ADPfusion` to support multi-core architectures. Such an extension is most beneficial for algorithms with an asymptotically higher runtime than RNA secondary structure prediction, say simultaneous folding and alignment. In addition it is those more complex algorithms that benefit most from a reduction in the complexity of actually implementing them with parallelization support.

SIMD Extensions

Modern CPUs are equipped with single-instruction multiple-data (SIMD) units that allow for a level of data parallelism. Where a normal CPU instruction works on scalars, SIMD instructions process 2–4 scalars in parallel with each instruction. For algorithms that execute the same computational kernel repeatedly, this kind of performance improvement can be very beneficial. Linear algebra routines are excellent candidates, as they process a large amount of numerical data using a small set of instructions.

More generic dynamic programming algorithms require a lot more programming effort when one wants to implement SIMD-aware versions. An algorithm like `RNAfold` accesses a large number of different arrays and combines the arrays in different ways. In addition, each algorithm in the `Vienna RNA` suite uses a different set of operations. Re-writing all algorithms to use SIMD instructions would be a significant undertaking.

Recently, Mainland et al. (2013) presented an extension of stream fusion in Haskell that includes automatic SIMD support. This extension is based on the `vector` library used by `ADPfusion`. It is now possible to adapt `ADPfusion` to use SIMD extensions for dynamic programs. Algorithms that have been implemented using our framework do not have to be changed at all. Once the library has been adapted, performance improvements become available for free.

Only on the library implementors side will this require additional work, due to the multitude of ways in which arrays can be accessed and array data processed.

9.3 Extended Secondary Structures

The `RNAwolf` algorithm for the prediction of extended RNA secondary structures as described in Chapter 7 predicts base pair triplets and for each paired nucleotide the pairing edge. It provides the proof that it is possible to design an algorithm that is asymptotically as fast as other secondary structure prediction algorithms. It is slower, by a constant, than others but predicts structures from a larger structural space.

Now that the grammatical underpinnings of extended secondary structures have been provided, we will have to reevaluate parametrization, multibranched loops, an extension to a whole suite of programs, and novel algorithms that make use of the additional information provided by extended structures.

Parametrization: The most important of these questions is how to provide a set of energy parameters that is competitive with established folding algorithms and also how exactly the energy parameters are to be modelled. Recent research on folding algorithms in general (Rivas et al., 2012; Rivas, 2013) suggests that significant further improvements in prediction accuracy will be hard to achieve.

One has to find a sweet spot between grammatical complexity, ability to train the parameters to capture important biological features, and avoid overfitting if possible. The resulting algorithm should provide actual free energy values for folded structures instead of just probabilities. Providing loop energy values is further complicated by the fact that no physical experiments exist[2] where melting energies for structures with non-canonical features have been measured.

The work on isostericity of RNA base pairs (Leontis et al., 2002) gives a good starting point on designing a useful model. Isosteric base pairs may be exchanged with each other in RNA structures. They should have roughly the same probability of occurrence and thus parameters for isosteric pairs can be collapsed into one parameter. This reduces the dimensionality of the parameter space and should be considered strongly where only a sparse set of data is available. Stacking loops in helices, on the other hand, can be modelled in much more detail.

Three kinds of data sets are available for parameter training. Melting experiments provide actual energy parameters, but the set of experiments is small and includes neither non-canonical pairs nor information on the actually paired edges.

`PDB` (or `FR3D`) information is statistical in nature. Using this source, it is possible to model base triplets and pairing edges in detail. The available structures have to be considered carefully as the data is of variable quality and structures can be, for example, in compounds.

The third kind of data comes from RNA family databases like `Rfam`. Each family is annotated with a consensus secondary structure and nucleotide (pair) distribution frequencies can be extracted. As shown by `CONTRAfold` (Do et al., 2006), it is possible to achieve prediction accuracies comparable to energy-based algorithms using structure statistics gathered from structural family databases. The number of available sequences and structures is huge compared to the other two sources, but again of variable quality. One has to be careful, for example, to distinguish between structures with strong experimental support and predicted (from covariance information) structures. While base pairing information does not include triplets or the exact pairing edge, all 4×4 possible pairs are considered instead of just the canonical 6.

Multibranched loops: `RNAwolf` currently implements a base triplet grammar for interior loop structures. This grammar is depicted in Chapter 7, Fig. 3. Including base triplets in interior loops already increases the number of non-terminals by four additional cases, and 27 additional rules. Allowing base triplets in multibranched loops increases this number further. While interior loops join two helices, multibranched loops join three or more. In addition to this, individual helices in multibranched loops may engage in coaxial stacking.

Armed with the `ADPfusion` framework we are confident that the complexity of the

[2] to the authors knowledge

multibranch-enabled algorithm can now be handled, and an implementation of the triplet model is possible even for multibranched loops.

The `RNAwolf` suite of programs: For now, `RNAwolf` consists of a minimum-free energy structure prediction algorithm, including sub-optimal backtracking. In the near future, this will be extended to partition function calculations and an `RNAalifold` variant to predict the consensus secondary structure of a set of aligned RNA sequences. These algorithms together will then form a core set from which further developments can follow.

Novel algorithms: The non-canonical structures that are interspersed with helical regions are often the active centers of structural RNAs or, like kink-turns, form unusual features of the three-dimensional structure. `RNAz` (Gruber et al., 2010) predictions produce a large number of candidates for novel structural non-coding RNAs. Each of the candidates can be evaluated with `RNAwolf` to determine if it contains a region of non-canonical base pairs. This region could be a biologically active site. It may be possible to improve the identification of functional structures in this way.

Another use case for extended structures is the prediction of RNA-Protein binding sites. Assuming that binding sites are formed of non-canonical structures, the successful prediction of these sites becomes possible with this new extended model for secondary structures.

RNA Structural Modules

Small RNA structural modules (see Chapter 2.4) are a structural feature that remains outside of the possibilities of (even extended) secondary structure prediction. The above-mentioned kink-turn can be partially replicated using an extended secondary structure model, but a number of crossing interactions within the module are out of reach of the model itself.

Recent experience with the integration of `G-quadruplex` structures (Lorenz et al., 2013) has shown that structure prediction tools like `RNAfold` can be extended with new, localized, structural features without harming performance. An extension of `RNAfold` and `RNAwolf` with these modules seems possible.

In Theis et al. (2013), a library of RNA structural modules was developed. These were extracted from `PDB` files and combined with `Rfam` multiple alignments. The resulting database extends the number of known RNA structural modules considerably and is the first step towards the improvements just discussed. The same modules can be used in tools like `RMdetect` (Cruz and Westhof, 2011) to scan multiple alignments of sequences for additional instances of the modules in RNA families. One of the open questions is to what extend the novel modules have known (i.e. published) – or unknown – biological functions.

9.4 RNA Family Models

Eddy and Durbin (1994) and Sakakibara et al. (1994) pioneered the use of stochastic context-free grammars as models for structural RNA families. `Infernal` (Nawrocki et al.,

2009; Kolbe and Eddy, 2011) covariance models are currently the de-facto standard with a large database, `Rfam` (Gardner et al., 2009, 2011), of existing families. `Infernal` and `Rfam` together are widely used to scan genomes for homologs of known RNA families. Since these earlier works many improvements have been integrated into the machinery. In this thesis two aspects of stochastic RNA family models are considered.

The first aspect is the impact of choosing a specific grammar to encode a stochastic RNA family model. `Infernal` makes the assumption that the most likely parse corresponds to the maximum likelihood solution using a specific semantics.

In Chapter 5 the exact semantics used by `Infernal` is formalized as are two others. The three semantics, structural, trace, and alignment semantics, correspond to three different ways in which to score the alignment of a sequence against an `Infernal` covariance model.

There are two main results. The first is that `Infernal` models are indeed unambiguous under the alignment semantics. This proof is important to have, as one can now state that under the alignment semantics the most likely parse tree is the maximum likelihood solution.

From a biological standpoint, many different alignments have the same biological meaning. This is a problem of sequence alignment ambiguity. All alignments with the same biological meaning map to one trace. The second main result in Chapter 5 is the development of a new grammar for stochastic RNA family models that is unambiguous with regard to traces. This grammar has the potential to better capture remote homologs of family members.

The second aspect is a measure for quality control of existing as well as novel RNA families discussed in Chapter 6. The `Rfam` data base contains over 2 200 models as of August 2012. With the `CMCompare` algorithm a measure, the `Link Score`, was introduced, that calculates how closely related two RNA families are, when viewed through the `Infernal` covariance model lens.

Instead of trying to compare a pair of structural RNA families directly, `CMCompare` aligns the two stochastic context-free grammars. The maximally scoring parse of this alignment produces a sequence, the `Link Sequence`. In principle, the `Link Sequence` can then be scored with `Infernal` against the two covariance models to be compared. This produces two bit scores. The smaller of these two scores is the `Link Score`. `CMCompare` calculates the `Link Score` in addition to the `Link Sequence` without the detour of using `Infernal`. If this score is high (20 bit were arbitrarily chosen as a cutoff) then the two models are likely to consider the same high-scoring position in a genome as a homolog, which creates a conflict.

Based upon these findings we can design a set of tools that uses the trace semantics instead of the alignment semantics. In addition, we can simplify the user experience of checking RNA family models with `CMCompare` and provide a model control mechanism for a trace-based algorithm as well.

A Novel Grammar for Non-coding RNA Search

With the rise of `Infernal` as the, probably, most often used stochastic context-free grammar for non-coding RNA search, many advances in improving the performance of the

algorithm were made. The actual SCFG however has largely been untouched since its inception.

We now have a new grammar based on traces to better capture remote homologs. This grammar is described in Chapter 5. We also have the computational and algorithmic framework, based on the results in Chapter 8, to implement this new grammar efficiently.

Armed with these two tools, it is possible to test the effect of a trace-based grammar for genome-wide scans. Just as with the planned improvements for extended secondary structures (Section 9.3) it will be required to implement the new grammar in `ADPfusion` to guarantee good performance and to provide a parameter-training mechanism that translates from RNA family models to the trace-based equivalent of covariance models ("trace-CMs").

The former task is actually quite straight-forward due to the possibility of encoding the pair of RNA family model and query sequence using a heterogeneous index shape (Sec. 9.1) which considerably simplifies the grammatical implementation.

The complexity of the latter task depends on a number of factors. `Infernal` uses a Bayesian prior that ties a number of different model features together to improve prediction accuracy (Nawrocki and Eddy, 2007). It should be possible to transfer this prior for parameter estimation of trace-CMs in the new grammar. Maximum-likelihood training without prior information would be even simpler and can be used at least to test the new search tool. The design of a specialized parameter estimator for trace-CMs could be more involved.

Quality Control for RNA Family Models

The `Link Score`, as a measure of similarity between RNA families, has been put to use to show that certain families of RNA replication elements (Chen and Brown, 2012) are actually well-separated from the rest of the Rfam database. The high `Link Score`s between the newly proposed families are indicative of a `Clan` (in `Rfam` terminology) of related families, as they should be. Similarity between RNA families was also considered in a study on structure and accessibility (Lange et al., 2012) to assert that there was no undue sequence or structural bias in the test data. In general, the `Rfam` database has improved in quality as measured by lower `Link Score`s when compared to earlier updates (considering versions 9 – 11 of `Rfam`).

To further simplify checking non-coding RNA families for uniqueness, or similarity to other, already known, families, a web server has been made available (Eggenhofer et al., 2013). It removes the problem of having to install the `CMCompare` command line tools. The web server provides a suite of additional features, from working with a whole set of families to providing graphical output, that simplifies examining the "neighborhood" of one or multiple RNA families. It is envisioned to extend the server to other stochastic family models. This includes Hidden-Markov models as used by the `Pfam` database (Bateman et al., 2002).

Care has to be taken in interpreting the calculated `Link Score` between two families. `Rfam` does not follow the concept of "one family – one model" strictly. For a number of families, it would be very complicated to keep all sequences in one big family. There are,

for example, six `RNaseP` families currently in `Rfam`. Each of these families is diverged on the structure and sequence level from the others. Trying to combine the six families into one would dilute the statistical power of the family compared to the six source families individually. To make the user aware that thusly related families can assign the same locus as a homolog, around 100 clans of related families have been defined.

High `Link Score`s therefore can mean either: two families that should be re-evaluated to increase their discriminatory power, or that a known or new `Rfam` clan has been discovered. The more one moves toward automated generation of complete family models and away from manual curation, the more important measures of quality will become. In this regard, it is appropriate to explore how the meaning of discriminatory power or familial relationships between models can be refined.

Finally, once the novel grammar for RNA family models (Chapter 5) is available in a more comprehensive suite of tools it, too, will be provided with a `CMCompare`-based quality control measure for generated families.

9.5 From Functions to Grammars and Back

The four scientific works presented in this thesis make use of the language of formal grammars to solve different problems in RNA bioinformatics. Using a formal grammar abstracts away more low-level problems that otherwise tend to obstruct the view. In this way one moves up from individual (recursive) functions to whole grammars.

Beginning with `ADPfusion` (Chapter 8) it has been possible to take a grammatical description and embed it efficiently and directly in Haskell. The grammar is parametrized over the algebra, which provides the scoring functions, and the input(s).

In Haskell and `ADPfusion` a grammar is just a normal function. This statement in itself is not noteworthy as it simply mirrors the effect of embedding a domain-specific language in Haskell. In contrast to this, the consequences are remarkable. By making the grammatical symbols, the terminals and non-terminals, of the grammar variable as well, we gain a further level of abstraction.

A grammar is now a function over its symbols, an algebra, and an input set which itself is tied to the terminal symbols. This is important because now a grammar becomes a function providing the structure for a *set of algorithms*.

It was mentioned in this thesis that the natural-language community uses formal grammars to describe language features, perform alignments, and other tasks. This is quite similar to many tasks done in RNA bioinformatics. One can probably find many fields where the same algorithms are used once topic-specific details, like the alphabet of the terminal characters, have been stripped away.

Viewed in this way, it should be possible to provide a library of grammars that provide an abstract description of a certain task. This description can be made concrete for a given problem by providing the correct terminal and non-terminal symbols. From this point on, the grammar behaves as usual, expecting a scoring scheme in the form of an algebra and the input for which to calculate a result. The shared grammar might be used in RNA secondary structure prediction and alignment of human languages, dependent on

which symbols have been selected.

This concept is quite fascinating because it apparently comes without further costs. Grammatical design is not more complicated and actually doesn't change at all. Even the, for `ADPfusion`, most important aspect of performance can be handled – the additional level of abstraction comes for free as grammars are already functions. In this more abstract setting they just require some additional arguments.

Using this approach we expect to be able to tackle novel problems more rapidly because more and more parts of our machinery become re-usable. Ultimately, it is envisioned to abstract away more and more of the details of each specific algorithm until they are reduced to their core functionality that is unique to each problem and solution.

It should, of course, not be concealed that frameworks like `ADPfusion` have to be extended to allow this to happen. It is not possible to remove details of implementations completely from our view but it is at least possible write a generic design once and then re-use it.

Bibliography

Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2007.

Tatsuya Akutsu. Dynamic programming algorithms for RNA secondary structure prediction with pseudoknots. *Discrete Applied Mathematics*, 104(1):pages 45–62, 2000.

Can Alkan, Emre Karakoc, Joseph H Nadeau, S Cenk Sahinalp, and Kaizhong Zhang. RNA-RNA Interaction Prediction and Antisense RNA Target Search. *Journal of Computational Biology*, 13(2):pages 267–282, 2006.

Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):pages 403–410, 1990.

Mirela Andronescu, Anne Condon, Holger H. Hoos, David H. Mathews, and Kevin P. Murphy. Efficient parameter estimation for RNA secondary structure prediction. *Bioinformatics*, 23(13):pages i19–i28, 2007.

Mirela Andronescu, Anne Condon, Holger H. Hoos, David H. Mathews, and Kevin P. Murphy. Computational approaches for RNA energy parameter estimation. *RNA*, 16(12):pages 2304–2318, 2010a.

Mirela S. Andronescu, Christina Pop, and Anne E. Condon. Improved free energy parameters for RNA pseudoknotted secondary structure prediction. *RNA*, 16(1):pages 26–42, 2010b.

Mirela Ştefania Andronescu. *Computational approaches for RNA energy parameter estimation*. Ph.D. thesis, The University Of British Columbia (Vancouver), 2008.

Alex Bateman, Ewan Birney, Lorenzo Cerruti, Richard Durbin, Laurence Etwiller, Sean R. Eddy, Sam Griffiths-Jones, Kevin L. Howe, Mhairi Marshall, and Erik L.L. Sonnhammer. The Pfam Protein Families Database. *Nucleic Acids Research*, 30(1):pages 276–280, 2002.

Richard E. Bellman. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, 38(8):pages 716–719, 1952.

Helen M Berman, John Westbrook, Zukang Feng, Gary Gilliland, TN Bhat, Helge Weissig, Ilya N Shindyalov, and Philip E Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28(1):pages 235–242, 2000.

Stephan H. Bernhart, Ivo L. Hofacker, Sebastian Will, Andreas R. Gruber, and Peter F. Stadler. RNAalifold: improved consensus structure prediction for RNA alignments. *BMC Bioinformatics*, 9(1):page 474, 2008.

Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

Richard Bird, Geraint Jones, and Oege de Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):pages 541–547, 1997.

Athanasius F. Bompfünewerer, Rolf Backofen, Stephan H. Bernhart, Jana Hertel, Ivo L. Hofacker, Peter F. Stadler, and Sebastian Will. Variations on RNA folding and alignment: lessons from Benasque. *Journal of Mathematical Biology*, 56(1):pages 129–144, 2008.

David Burkett, John Blitzer, and Dan Klein. Joint Parsing and Alignment with Weakly Synchronized Grammars. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 127–135. Association for Computational Linguistics, 2010.

Liming Cai, Russell L. Malmberg, and Yunzhou Wu. Stochastic modeling of RNA pseudoknotted structures: a grammatical approach. *Bioinformatics*, 19(suppl 1):pages i66–i73, 2003.

Neil A. Campbell and Jane B. Reece. *Biologie*. Spektrum Verlag, Heidelberg, Berlin, 6. edition, 2003.

Manuel M.T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant Generics: Fast and Easy. 2009.

Augustine Chen and Chris Brown. Distinct families of cis-acting RNA replication elements epsilon from hepatitis B viruses. *RNA Biology*, 9(2):pages 1–7, 2012.

Hamidreza Chitsaz, Raheleh Salari, S Cenk Sahinalp, and Rolf Backofen. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, 25(12):pages i365–i373, 2009.

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):pages 113–124, 1956.

Noam Chomsky. On Certain Formal Properties of Grammars. *Information and control*, 2(2):pages 137–167, 1959.

Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP'00, pages 268–279. ACM, 2000.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT press, 2001.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 315–326. ACM, 2007.

Francis Crick. Central Dogma of Molecular Biology. *Nature*, 227:pages 561–563, 1970.

José Almeida Cruz, Marc-Frédérick Blanchet, Michal Boniecki, Janusz M. Bujnicki, Shie-Jie Chen, Song Cao, Rhiju Das, Feng Ding, Nikolay V. Dokholyan, Samuel Coulbourn Flores, Lili Huang, Christopher A. Lavender, Véronique Lisi, François Major, Katarzyna Mikolajczak, Dinshaw J. Patel, Anna Philips, Tomasz Puton, John Santalucia, Fredrick Sijenyi, Thomas Hermann, Kristian Rother, Magdalena Rother, Alexander Serganov, Marcin Skorupski, Tomasz Soltysinski, Parin Sripakdeevong, Irina Tuszynska, Kevin M. Weeks, Christina Waldsich, Michael Wildauer, Neocles B. Leontis, and Eric Westhof. RNA-Puzzles: A CASP-like evaluation of RNA three-dimensional structure prediction. *RNA*, 18(4):pages 610–625, 2012.

José Almeida Cruz and Eric Westhof. Sequence-based identification of 3D structural modules in RNA with RMDetect. *Nature Methods*, 8(6):pages 513–519, 2011.

Noah M Daniels, Andrew Gallant, and Norman Ramsey. Experience report: Haskell in computational biology. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 227–234. ACM, 2012.

Rhiju Das and David Baker. Automated de novo prediction of native-like RNA tertiary structures. *Proceedings of the National Academy of Sciences*, 104(37):pages 14 664–14 669, 2007.

Rhiju Das, John Karanicolas, and David Baker. Atomic accuracy in predicting and designing noncanonical RNA structure. *Nature methods*, 7(4):pages 291–294, 2010.

Richard C Deonier, Simon Tavaré, and Michael S Waterman. *Computational Genome Analysis: An Introduction*. Springer, 2005.

Chuong B. Do, Daniel A. Woods, and Serafim Batzoglou. CONTRAfold: RNA secondary structure prediction without physics-based models. *Bioinformatics*, 22(14):page e90, 2006.

Robin D. Dowell and Sean R. Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 5(1):page 71, 2004.

Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis*. Cambridge Univ. Press, 1998.

R Kent Dybvig. *The SCHEME programming language*. Mit Press, 2003.

Sean R. Eddy. HMMER: profile HMMs for protein sequence analysis. *Bioinformatics*, 14:pages 755–763, 1998.

Sean R. Eddy and Richard Durbin. RNA sequence analysis using covariance models. *Nucleic Acids Research*, 22(11):pages 2079–2088, 1994.

Florian Eggenhofer, Ivo L. Hofacker, and Christian Höner zu Siederdissen. CMCompare webserver: Comparing RNA families via Covariance Models. *submitted*, 2013.

Mathieu Fourment and Michael R Gillings. A comparison of common programming languages used in bioinformatics. *BMC bioinformatics*, 9(1):page 82, 2008.

Paul P Gardner, Jennifer Daub, John Tate, Benjamin L Moore, Isabelle H Osuch, Sam Griffiths-Jones, Robert D Finn, Eric P Nawrocki, Diana L Kolbe, Sean R Eddy, et al. Rfam: Wikipedia, clans and the "decimal" release. *Nucleic Acids Research*, 39(suppl 1):pages D141–D145, 2011.

Paul P. Gardner, Jennifer Daub, John G. Tate, Eric P. Nawrocki, Diana L. Kolbe, Stinus Lindgreen, Adam C. Wilkinson, Robert D. Finn, Sam Griffiths-Jones, Sean R. Eddy, and Alex Bateman. Rfam: updates to the RNA families database. *Nucleic Acids Research*, 37(suppl 1):pages D136–D140, 2009.

Robert Giegerich and Christian Höner zu Siederdissen. Semantics and Ambiguity of Stochastic RNA Family Models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(2):pages 499–516, 2011.

Robert Giegerich and Carsten Meyer. Algebraic Dynamic Programming. In *Algebraic Methodology And Software Technology*, volume 2422, pages 243–257. Springer, 2002.

Robert Giegerich, Carsten Meyer, and Peter Steffen. Towards a Discipline of Dynamic Programming. *Informatik bewegt, GI-Edition-Lecture Notes in Informatics*, pages 3–44, 2002.

Robert Giegerich, Carsten Meyer, and Peter Steffen. A Discipline of Dynamic Programming over Sequence Data. *Science of Computer Programming*, 51(3):pages 215–263, 2004.

Andrew Gill, John Launchbury, and Simon Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232. ACM, 1993.

Sam Griffiths-Jones. RALEE – RNA ALignment editor in Emacs. *Bioinformatics*, 21(2):pages 257–259, 2005.

Sam Griffiths-Jones, Alex Bateman, Mhairi Marshall, Ajay Khanna, and Sean R. Eddy. Rfam: an RNA family database. *Nucleic Acids Research*, 31(1):pages 439–441, 2003.

Andreas R. Gruber, Sven Findeiß, Stefan Washietl, Ivo L. Hofacker, and Peter F. Stadler. RNAz 2.0: Improved noncoding RNA detection. In *Pacific Symposium on Biocomputing*, volume 15, pages 69–79. 2010.

Andreas R Gruber, Ronny Lorenz, Stephan H Bernhart, Richard Neuböck, and Ivo L Hofacker. The Vienna RNA Websuite. *Nucleic acids research*, 36(suppl 2):pages W70–W74, 2008.

Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-Verlag New York Inc, 2008.

Ralf Hinze, Thomas Harper, and Daniel W.H. James. Theory and Practice of Fusion. *Implementation and Application of Functional Languages*, pages 19–37, 2011.

Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing Approaches to Generic Programming in Haskell. *Datatype-Generic Programming*, pages 72–149, 2007.

Ivo L. Hofacker, Martin Fekete, and Peter F. Stadler. Secondary Structure Prediction for Aligned RNA Sequences. *Journal of Molecular Biology*, 319(5):pages 1059–1066, 2002.

Ivo L. Hofacker, Walter Fontana, Peter F. Stadler, L. Sebastian Bonhoeffer, Manfred Tacker, and Peter Schuster. Fast Folding and Comparison of RNA Secondary Structures. *Monatshefte für Chemie/Chemical Monthly*, 125(2):pages 167–188, 1994.

Christian Höner zu Siederdissen. Sneaking Around concatMap: Efficient Combinators for Dynamic Programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 215–226. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1054-3.
URL `http://doi.acm.org/10.1145/2364527.2364559`

Christian Höner zu Siederdissen, Stephan H. Bernhart, Peter F. Stadler, and Ivo L. Hofacker. A folding algorithm for extended RNA secondary structures. *Bioinformatics*, 27(13):pages 129–136, 2011.

Christian Höner zu Siederdissen and Ivo L. Hofacker. Discriminatory power of RNA family models. *Bioinformatics*, 26(18):pages 453–459, 2010.

Fenix W.D. Huang, Jin Qin, Christian M. Reidys, and Peter F. Stadler. Partition function and base pairing probabilities for RNA–RNA interaction prediction. *Bioinformatics*, 25(20):pages 2646–2654, 2009.

Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 1–55. ACM, 2007.

John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):pages 98–107, 1989.

Daniel Jurafsky, Chuck Wooters, Jonathan Segal, Andreas Stolcke, Eric Fosler, G Tajchaman, and Nelson Morgan. Using a stochastic context-free grammar as a language model for speech recognition. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 189–192. IEEE, 1995.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 261–272. ACM, 2010.

Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. *Reflections on the Work of CAR Hoare*, pages 301–331, 2010.

Diana L. Kolbe and Sean R. Eddy. Fast Filtering for RNA Homology Search. *Bioinformatics*, 27(22):pages 3102–3109, 2011.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Notices*, volume 38, pages 26–37. ACM, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *ACM SIGPLAN Notices*, volume 40, pages 204–215. ACM, 2005.

Sita J. Lange, Daniel Maticzka, Mathias Möhl, Joshua N. Gagnon, Chris M. Brown, and Rolf Backofen. Global or local? Predicting secondary structure and accessibility in mRNAs. *Nucleic Acids Research*, 2012.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

Neocles B. Leontis, Jesse Stombaugh, and Eric Westhof. The non-Watson-Crick base pairs and their associated isostericity matrices. *Nucleic Acids Research*, 30(16):pages 3497–3531, 2002.

Neocles B. Leontis and Eric Westhof. Geometric nomenclature and classification of RNA base pairs. *RNA*, 7(4):pages 499–512, 2001.

Neocles B. Leontis and Eric Westhof. Analysis of RNA motifs. *Current Opinion in Structural Biology*, 13(3):pages 300–308, 2003.

Roman Leshchinskiy. Recycle Your Arrays! *Practical Aspects of Declarative Languages*, pages 209–223, 2009.

Felipe Lessa, Daniele Neto, Kátia Guimarães, Marcelo Brigido, and Maria Walter. Regene: Automatic Construction of a Multiple Component Dirichlet Mixture Priors Covariance Model to Identify Non-coding RNA. *Bioinformatics Research and Applications*, pages 380–391, 2011.

Art Lew and Holger Mauch. *Dynamic Programming: A Computational Tool*, volume 38. Springer, 2006.

Ronny Lorenz, Stephan H. Bernhart, Fabian Externbrink, Jing Qin, Christian Höner zu Siederdissen, Fabian Amman, Ivo L. Hofacker, and Peter F. Stadler. RNA Folding Algorithms with G-Quadruplexes. In M.C.P. De Souto and M.G. Kann, editors, *Brazilian Symposium on Bioinformatics (BSB 2012), Lecture Notes in Bioinformatics*, volume 7409, pages 49–60. Springer, Heidelberg, 2012.

Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(26), 2011.

Ronny Lorenz, Stephan H. Bernhart, Jing Qin, Christian Höner zu Siederdissen, Andrea Tanzer, Fabian Amman, Ivo L. Hofacker, and Peter F. Stadler. 2D meets 4G: G-Quadruplexes in RNA Secondary Structure Prediction. *IEEE/ACM Transactions on Computation Biology and Bioinformatics*, 2013.

Zhi John Lu, Douglas H. Turner, and David H. Mathews. A set of nearest neighbor parameters for predicting the enthalpy change of RNA secondary structure formation. *Nucleic Acids Research*, 34(17):pages 4912–4924, 2006.

Rune B Lyngsø and Christian NS Pedersen. Pseudoknots in rna secondary structures. In *Proceedings of the fourth annual international conference on Computational molecular biology*, pages 201–209. ACM, 2000.

Geoffrey Mainland, Roman Leshchinskiy, Simon Peyton Jones, and Simon Marlow. Haskell Beats C Using Generalized Stream Fusion. 2013.

David H. Mathews, Jeffrey Sabina, Michael Zuker, and Douglas H. Turner. Expanded Sequence Dependence of Thermodynamic Parameters Improves Prediction of RNA Secondary Structure. *Journal of Molecular Biology*, 288(5):pages 911–940, 1999.

John S. Mattick and Igor V. Makunin. Non-coding RNA. *Human Molecular Genetics*, 15(Review Issue 1):pages R17–R29, 2006.

John S. McCaskill. The equilibrium partition function and base pair binding probabilities for RNA secondary structure. *Biopolymers*, 29(6-7):pages 1105–1119, 1990.

Irmtraud M. Meyer and István Miklós. SimulFold: Simultaneously Inferring RNA Structures Including Pseudoknots, Alignments, and Trees Using a Bayesian MCMC Framework. *PLoS computational biology*, 3(8):page e149, 2007.

Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS'89)*, pages 14–23. IEEE, 1989.

Ulrike Mückstein, Hakim Tafer, Stephan H. Bernhart, Maribel Hernandez-Rosales, Jörg Vogel, Peter F. Stadler, and Ivo L. Hofacker. Translational control by RNA–RNA interaction: Improved computation of RNA–RNA binding thermodynamics. *Bioinformatics Research and Development*, pages 114–127, 2008.

Ulrike Mückstein, Hakim Tafer, Jörg Hackermüller, Stephan H. Bernhart, Peter F. Stadler, and Ivo L. Hofacker. Thermodynamics of RNA–RNA binding. *Bioinformatics*, 22(10):pages 1177–1182, 2006.

Eric P. Nawrocki and Sean R. Eddy. Query-Dependent Banding (QDB) for Faster RNA Similarity Searches. *PLoS Computational Biology*, 3(3):page e56, 2007.

Eric P. Nawrocki, Diana L. Kolbe, and Sean R. Eddy. Infernal 1.0: inference of RNA alignments. *Bioinformatics*, 25(10):pages 1335–1337, 2009.

Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms for Loop Matchings. *SIAM Journal on Applied Mathematics*, 35(1):pages 68–82, 1978.

Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

Marc Parisien and Francois Major. The MC-Fold and MC-Sym pipeline infers RNA structure from sequence data. *Nature*, 452:pages 51–55, 2008.

Dimitri D. Pervouchine. IRIS: Intermolecular RNA Interaction Search. *Genome informatics series*, 15(2):page 92, 2004.

Simon Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 327–337. ACM, 2007.

Simon L. Peyton Jones and André L.M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1):pages 3–47, 1998.

Jens Reeder and Robert Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC bioinformatics*, 5(1):page 104, 2004.

Christian M. Reidys, Fenix W.D. Huang, Jørgen E. Andersen, Robert C. Penner, Peter F. Stadler, and Markus E. Nebel. Topology and prediction of RNA pseudoknots. *Bioinformatics*, 27(8):pages 1076–1085, 2011.

Vladimir Reinharz, François Major, and Jérôme Waldispühl. Towards 3D structure prediction of large RNA molecules: an integer programming framework to insert local 3D motifs in RNA secondary structure. *Bioinformatics*, 28(12):pages i207–i214, 2012.

Elena Rivas. The four ingredients of single-sequence RNA secondary structure prediction: A unifying perspective. *submitted*, 2013.

Elena Rivas and Sean R. Eddy. The language of RNA: a formal grammar that includes pseudoknots. *Bioinformatics*, 16(4):pages 334–340, 2000.

Elena Rivas, Raymond Lang, and Sean R. Eddy. A range of complex probabilistic models for RNA secondary structure prediction that includes the nearest-neighbor model and more. *RNA*, 18(2):pages 193–212, 2012.

Yasubumi Sakakibara, Michael Brown, Richard Hughey, I Saira Mian, Kimmen Sjölander, Rebecca C Underwood, and David Haussler. Stochastic context-free grammers for tRNA modeling. *Nucleic Acids Research*, 22(23):pages 5112–5120, 1994.

David Sankoff. Simultaneous solution of the RNA folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics*, pages 810–825, 1985.

Michael Sarver, Craig L. Zirbel, Jesse Stombaugh, Ali Mokdad, and Neocles B. Leontis. FR3D: Finding Local and Composite Recurrent Structural Motifs in RNA 3D Structures. *Journal of Mathematical Biology*, (56):pages 215–252, 2008.

K. Sato, Y. Kato, M. Hamada, T. Akutsu, and K. Asai. IPknot: fast and accurate prediction of RNA secondary structures with pseudoknots using integer programming. *Bioinformatics*, 27(13):pages i85–i93, 2011.

Georg Sauthoff. *Bellman's GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming*. Ph.D. thesis, Bielefeld University, 2011.

Georg Sauthoff, Stefan Janssen, and Robert Giegerich. Bellman's GAP - A Declarative Language for Dynamic Programming. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP'11, pages 29–40. ACM, 2011.

Georg Sauthoff, Mathias Möhl, Stefan Janssen, and Robert Giegerich. Bellman's GAP – a Language and Compiler for Dynamic Programming in Sequence Analysis. *Bioinformatics*, 2013.

Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

Jennifer A Smith. RNA Search with Decision Trees and Partial Covariance Models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 6(3):pages 517–527, 2009.

TF Smith and MS Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147:pages 195–197, 1981.

Guy L Steele. *Common LISP: the language*. Digital Press, 1990.

Peter Steffen. *Compiling a Domain Specific Language for Dynamic Programming*. Ph.D. thesis, Bielefeld University, 2006.

David A. Terei and Manuel M.T. Chakravarty. An LLVM Backend for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell (Haskell '10)*, pages 109–120. ACM, 2010.

Corinna Theis, Christian Höner zu Siederdissen, Ivo L. Hofacker, and Jan Gorodkin. Automated identification of 3D modules with discriminative power in RNA structural alignments. *submitted*, 2013.

Ignacio Tinoco, Philip N. Borer, Barbara Dengler, Mark D. Levine, Olke C. Uhlenbeck, Donald M. Crothers, and Jay Gralla. Improved Estimation of Secondary Structure in Ribonucleic Acids. *Nature*, 246(150):pages 40–41, 1973.

Ignacio Tinoco, Olke C. Uhlenbeck, and Mark D. Levine. Estimation of Secondary Structure in Ribonucleic Acids. *Nature*, 230(5293):pages 362–367, 1971.

Douglas H. Turner and David H. Mathews. NNDB: the nearest neighbor parameter database for predicting stability of nucleic acid secondary structure. *Nucleic Acids Research*, 38:pages D280–D282, 2010.

Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):pages 231–248, 1990.

Amy E. Walter, Douglas H. Turner, James Kim, Matthew H. Lyttle, Peter Müller, David H. Mathews, and Michael Zuker. Coaxial stacking of helixes enhances binding of oligoribonucleotides and improves predictions of RNA folding. *Proceedings of the National Academy of Sciences*, 91(20):pages 9218–9222, 1994.

Stefan Washietl, Ivo L. Hofacker, and Peter F. Stadler. Fast and reliable prediction of noncoding RNAs. *Proceedings of the National Academy of Sciences of the United States of America*, 102(7):pages 2454–2459, 2005.

Sebastian Will, Tejal Joshi, Ivo L. Hofacker, Peter F. Stadler, and Rolf Backofen. LocARNA-P: Accurate boundary prediction and improved detection of structural RNAs. *RNA*, 18(5):pages 900–914, 2012.

Sebastian Will, Kristin Reiche, Ivo L. Hofacker, Peter F. Stadler, and Rolf Backofen. Inferring Non-Coding RNA Families and Classes by Means of Genome-Scale Structure-Based Clustering. *PLoS Computational Biology*, 3(4):page e65, 2007. ISSN 1553-7358.

Han Min Wong, Linda Payet, Julian Leon Huppert, et al. Function and targeting of G-quadruplexes. *Current Opinion in Molecular Therapeutics*, 11(2):page 146, 2009.

Stefan Wuchty, Walter Fontana, Ivo L. Hofacker, and Peter Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49(2):pages 145–165, 1999. ISSN 0006-3525.

Tianbing Xia, John SantaLucia Jr, Mark E. Burkard, Ryszard Kierzek, Susan J. Schroeder, Xiaoqi Jiao, Christopher Cox, and Douglas H. Turner. Thermodynamic Parameters for an Expanded Nearest-Neighbor Model for Formation of RNA Duplexes with Watson-Crick Base Pairs. *Biochemistry*, 37(42):pages 14 719–14 735, 1998.

Zizhen Yao, Zasha Weinberg, and Walter L. Ruzzo. CMfinder a covariance model based RNA motif finding algorithm. *Bioinformatics*, 22(4):pages 445–452, 2006.

Michael Zuker. On Finding all Suboptimal Foldings of an RNA Molecule. *Science*, 244(4900):pages 48–52, 1989.

Michael Zuker and Patrick Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):pages 133–148, 1981.

Christian Höner zu Siederdissen
Institut für Theoretische Chemie
Währinger Straße 17
1090 Wien
Austria
Tel: +43-1-4277-52737
Fax: +43-1-4277-52793
choener@tbi.univie.ac.at
http://www.tbi.univie.ac.at/∼choener/
Date of Birth: September, 6th 1981
Nationality: Germany

Research Interests

computational biology	• RNA secondary structure prediction • genome-wide search for structural non-coding RNAs • optimization methods (*dynamic programming, convex optimization*)
functional programming	• functional programming as a tool for computational biology • high-performance functional programming • Haskell (a lazy functional programming language)
Bayesian statistics	• statistical methods as a basis for accurate prediction algorithms

Education

July 2008 –	• PhD student with Prof. Dr. Ivo L. Hofacker at the *Institut für Theoretische Chemie, Universität Wien*; thesis title: *Grammatical Approaches to Problems in RNA Bioinformatics*
Feb 2008 – Jun 2008	• research assistant with Prof. Dr. Robert Giegerich, *AG Praktische Informatik, Universität Bielefeld*
2002 – Dec 2007	• Diploma (Masters) in *Computer Science in the Natural Sciences, Bielefeld University*; supervisors: Prof. Dr. Sven Rahmann & Prof. Dr. Robert Giegerich; thesis title: *Determination of the Sum Formula for Small Molecules from High Resolution FT-ICR Spectra*

List of Publications

[1] Corinna Theis, Christian Höner zu Siederdissen, Ivo L. Hofacker, and Jan Gorodkin. Automated identification of 3D modules with discriminative power in RNA structural alignments. *submitted*, 2013.

[2] Florian Eggenhofer, Ivo L. Hofacker, and Christian Höner zu Siederdissen. CMCompare webserver: Comparing RNA families via Covariance Models. *submitted*, 2013.

[3] Ronny Lorenz, Stephan H. Bernhart, Jing Qin, Christian Höner zu Siederdissen, Andrea Tanzer, Fabian Amman, Ivo L. Hofacker, and Peter F. Stadler. 2D meets 4G: G-Quadruplexes in RNA Secondary Structure Prediction. *IEEE/ACM Transactions on Computation Biology and Bioinformatics*, 2013.

[4] Joaquin Vierna, Stefanie Wehner, Christian Höner zu Siederdissen, Andrés Martínez-Lage, and Manja Marz. Evolutionary analysis of 5S ribosomal DNA in metazoans. *submitted*, 2013.

[5] Christian Höner zu Siederdissen. Sneaking Around concatMap: Efficient Combinators for Dynamic Programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 215–226, New York, NY, USA, 2012. ACM.

[6] Ronny Lorenz, Stephan H. Bernhart, Fabian Externbrink, Jing Qin, Christian Höner zu Siederdissen, Fabian Amman, Ivo L. Hofacker, and Peter F. Stadler. RNA Folding Algorithms with G-Quadruplexes. In M.C.P. De Souto and M.G. Kann, editors, *Brazilian Symposium on Bioinformatics (BSB 2012), Lecture Notes in Bioinformatics*, volume 7409, pages 49–60. Springer, Heidelberg, 2012.

[7] Conrad Helm, Stephan H. Bernhart, Christian Höner zu Siederdissen, Birgit Nickel, and Christoph Bleidorn. Deep sequencing of small RNAs confirms an annelid affinity of Myzostomida. *Molecular Phylogenetics and Evolution*, 64:198–203, 2012.

[8] Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(26), 2011.

[9] Christian Höner zu Siederdissen, Stephan H. Bernhart, Peter F. Stadler, and Ivo L. Hofacker. A folding algorithm for extended RNA secondary structures. *Bioinformatics*, 27(13):129–136, 2011.

[10] Manja Marz, Andreas R. Gruber, Christian Höner zu Siederdissen, Fabian Amman, Stefan Badelt, Sebastian Bartschat, Stephan H. Bernhart, Wolfgang Beyer, Stefanie Kehr, Ronny Lorenz, Andrea Tanzer, Dilmurat Yusuf, Hakim Tafer, Ivo L. Hofacker, and Peter F. Stadler. Animal snoRNAs and scaRNAs with exceptional structures. *RNA Biology*, 8(6):1–9, 2011.

[11] Robert Giegerich and Christian Höner zu Siederdissen. Semantics and Ambiguity of Stochastic RNA Family Models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(2):499–516, 2011.

List of Publications (continued)

[12] Matthias Hackl, Tobias Jakobi, Jochen Blom, Daniel Doppmeier, Karina Brinkrolf, Rafael Szczepanowski, Stephan H. Bernhart, Christian Höner zu Siederdissen, Juan A. Hernandez Bort, Matthias Wieser, Renate Kunert, Simon Jeffs, Ivo L. Hofacker, Alexander Goesmann, Alfred Pühler, Nicole Borth, and Johannes Grillari. Next-generation sequencing of the Chinese hamster ovary microRNA transcriptome: identification, annotation and profiling of microRNAs as targets for cellular engineering. *Journal of Biotechnology*, 153:62–75, 2011.

[13] Christian Höner zu Siederdissen and Ivo L. Hofacker. Discriminatory power of RNA family models. *Bioinformatics*, 26(18):453–459, 2010.

[14] Christian Höner zu Siederdissen. Determination of the Sum Formula for Small Molecules from High Resolution FT-ICR Spectra. Diploma Thesis, AG Genominformatik, Technische Fakultät, Universität Bielefeld, 2007.

[15] Christian Höner zu Siederdissen, Susanne Ragg, and Sven Rahmann. Discovering Biomarkers for Myocardial Infarction from SELDI-TOF Spectra. In Reinhold Decker and Hans J. Lenz, editors, *Advances in Data Analysis*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 569–576. Springer Berlin Heidelberg, 2007.