Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.) der Technischen Fakultät der Universität Bielefeld

Bellman's GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming

Georg Sauthoff*

March 4, 2011

*gsauthof@techfak.uni-bielefeld.de

Gedruckt auf alterungsbeständigem Papier °° ISO 9706

Abstract

The dissertation describes the new Bellman's GAP which is a programming system for writing dynamic programming algorithms over sequential data. It is the second generation implementation of the algebraic dynamic programming framework (ADP) [20]. The system includes the multi-paradigm language (GAP-L), its compiler (GAP-C), functional modules (GAP-M) and a web site (GAP Pages) to experiment with GAP-L programs. GAP-L includes declarative constructs, e.g. tree grammars to model the search space, and imperative constructs for programming advanced scoring functions. The syntax of GAP-L is similar to C/Java to lower usage barriers. GAP-C translates the high-level and index-free GAP-L programs into efficient C++-Code, which is competitive with handwritten code. It includes a novel table design optimization algorithm, support for dynamic programming (DP) over multiple sequences (multi-track DP), sampling, optional top-down evaluation, various backtracing schemes etc. GAP-M includes modules for use in GAP-L programs. Examples are efficient representations of classification data types and sampling as well as filter helper functions. GAP Pages contain web dialogs for selected text book dynamic programming algorithms implemented in GAP-L. The web dialogs allow interactive ad-hoc experiments with different inputs and combinations of algebras.

Several benchmarks and examples in the dissertation show the practical efficiency of Bellman's GAP in terms of program runtime and development time.

Acknowledgements

I thank my supervisor Prof. Dr. Robert Giegerich for his support during my PhD studies, his open-door policy and very interesting discussions.

I am grateful to the DFG (Deutsche Forschungsgemeinschaft) for funding my work through the GK635.

Thanks are due to Jens Reeder for providing ADP versions of RNA folding algorithms and discussions.

I thank Christian Lang for the excellent teamwork and fruitful discussions during the table design study project in 2005.

I thank Stefan Janssen, Alexander Kaiser and Marco Rüther for fruitful discussions, effective coffee breaks and proofreading efforts. Stefan also did careful alpha and beta testing of Bellman's GAP, which was quite helpful.

Thanks to fellow open-source enthusiasts who are providing an enormous amount of high quality software. I appreciate their efforts and I profited by that a lot. In particular the Linux Kernel, the GNU Compiler Collection, the Boost C++ library, Flex and Bison parser generators, the LaTeX typesetting system and a lot of sophisticated LaTeX packages, including the KOMA classes, tikz and pgfplots all with incredible documentations have been very useful.

I am deeply grateful to my parents for their ongoing support.

Contents

1 Introduction

Dynamic Programming (DP) is an optimization method developed by Richard Bellman in the 1950ies [5]. It is used to solve optimization problems where overall solutions are computed from sub-solutions. Solutions and sub-solutions are computed using the same objective function, e.g. maximization or minimization. Applying the objective functions to compute optimal sub-solutions and the tabulation of reused sub-solutions leads to an algorithm that evaluates a search space of usually exponential size in polynomial runtime and space. Bellman's Principle (Definition 5) specifies the properties an objective function has to satisfy such that dynamic programming can be used to solve the optimization problem.

Traditionally, dynamic programming algorithms are presented as matrix recurrences involving case distinctions. For example, the matrix-entry $M_{i,j}$ contains the sub-solution for the sub-problem (i, j), the complete problem instance is represented by $(0, n)$ and the right hand side of the matrix recurrence specifies for each (i, j) how to recursively compute the solution taking solutions for some (k, l), with $i \leq k \leq l \leq j$ and $(k, l) \neq (i, j)$ into account. Implementing the matrix recurrence as a computer program means using an array as matrix and a loop control structure that computes entries representing smaller sub-solutions before computing larger ones.

Often, one is not only interested in the optimal score as specified by the matrix recurrence, but also in the candidate structure from the search space that represents the score. The structure is derived from the sequence of optimization steps during the computation of the score. Given a score-matrix computing the structure is called backtracing: starting from the global solution recursively the steps yielding that solution are traced back.

In textbooks, dynamic programming is usually introduced with example algorithms that only use one matrix recurrence (e.g. [8]). Such examples are easily understood and implemented in a straight-forward way, but in practice dynamic programming algorithms are common that use several interdepending matrix recurrences, e.g. up to 30 matrix recurrences in manually developed algorithms.

Developing dynamic programming algorithms holds several challenges. One has to decide what the search space of the problem domain is, how many tables are needed, how to score elements of the search space and what structure a scoring scheme implicates. Using matrix recurrences as specification tool for developing DP algorithms, all these concerns are not separated, but need to be solved in an interleaved fashion.

Learning dynamic programming only from small textbook style examples, it is not obvious that these different concerns exist, e.g. one could assume that all matrix

recurrences have to be tabulated in any case.

In addition, index computations in matrix recurrences are error-prone, which leads to tedious debugging sessions when implementing the recurrences as an imperative computer program. For example, when writing $(i+1, j+1)$ instead of $(i+1, j-1)$ in one place may only yield once in a while an obviously wrong solution during optimization.

More difficulties arise, when extending an existing DP algorithm. Changing, for example, the backtracing mode, making minor conceptual changes to the scoring scheme or the need to combine several objectives may require a completely new implementation of the DP algorithm.

Algebraic Dynamic Programming (ADP, Chapter 2), developed by Robert Giegerich around 2004, is a formal framework for developing dynamic programming algorithms over sequences that provides a clear separation of the described concerns and eliminates the use of indices, coining the slogan: "No subscripts, no errors!"

Bellman's GAP is a second generation implementation of ADP, which is the topic of this dissertation and is outlined in the next section. The role of dynamic programming in bioinformatics is discussed in Section 1.2 and several approaches that are similar to ADP in their motivation are reviewed in Section 1.3.

1.1 Problem Statement

The subject of this dissertation is the development of a novel programming system for ADP that provides the advantages and features of ADP as published in 2004, generalizes ADP for dynamic programming over multiple sequences (multi-track DP), provides table design that takes constant runtime factors into account and uses a new domain specific programming language that is easy accessible by ADP novices, efficient to use by ADP professionals and not based on embedding into a host language.

Additional goals of this work are the investigation of new product operations, the development of classification schemes for products and the compilation of products into efficient code. Further, alternative evaluation strategies, besides CYK-style parsing, are to be investigated.

The aim of this dissertation is to establish a new programming system for ADP which is to be called Bellman's GAP in the following. The task is twofold: development of GAP-L (Chapter 4), the novel domain specific programming language for ADP that has a Java/C-like syntax, but has declarative constructs and development of GAP-C from scratch (Chapter 5), an optimizing compiler that translates GAP-L programs into efficient C++ code. Besides the presentation of novel and generalized semantic analysis algorithms, e.g. for table design (Section 5.3.7) and table dimension analysis (Section 5.3.6), the presentation of code-generation technique, e.g. an alternative evaluation scheme that exploits sparseness (Section 5.4.1) and parallelization (Section 5.4.2), Chapter 8 shows several benchmarks of the practical runtime and memory usage of GAP-L programs compiled with GAP-C, comparing

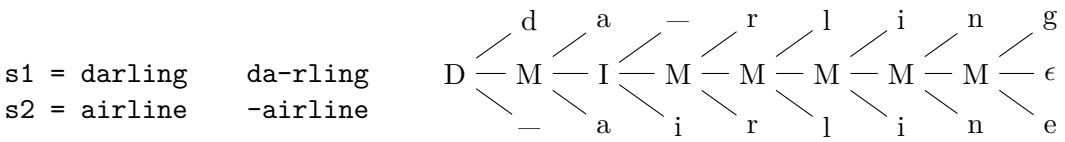


Figure 1.1: Two example sequences and a possible alignment in two different notations.

them with hand-written versions and with versions using previous ADP tools.

I developed GAP-M (Chapter 6) as a runtime-library that is part of GAP-C. It provides efficient implementations of internal data-structures and reusable functions for the application domain of bioinformatics. GAP Pages (Chapter 7) is a website project to present the GAP-L syntax using examples of well known dynamic programming as GAP-L versions and providing an interactive interface for ad-hoc experiments with different inputs and objectives.

The ADP generalizations I developed during the design of GAP-L that are independent of GAP-L are presented in Section 2.2.

Chapter 2 gives an overview over the current ADP framework and Chapter 3 gives an overview over Bellman's GAP including a discussion of the motivation.

Chapter 9 concludes the dissertation and Chapter 10 discusses open problems and further research opportunities.

1.2 Role of Dynamic Programming in Bioinformatics

Dynamic programming plays an important role in bioinformatics. Several optimization problems in bioinformatics can be solved with the help of dynamic programming. For example, in a textbook about sequence analysis [12], every presented algorithm uses the method of dynamic programming. In the following a few example use cases are described.

In the analysis of genomic sequence data, the optimal pairwise sequence alignment is an important building block that can be computed via dynamic programming. Given two sequences, the alignment is a sequence of edit-operations that, applied to the first sequence, yields the second sequence. Figure 1.1 shows an example of an alignment. Examples of edit-operations are deletion, insertion, match, mismatch or transposition of characters. Each edit-operation has a score associated and the score of the alignment is the sum of all edit-operation scores. An alignment is optimal, if it has the best score. Depending on the scoring model, the objective function is to maximize the score, i.e. to maximize the similarities between the sequences, or to minimize the score, i.e. to minimize the dissimilarities or distance between the sequences. The best score is then the maximal or minimal one. The search space of all possible alignments is of exponential size, and with dynamic programming, it can be evaluated in polynomial time.

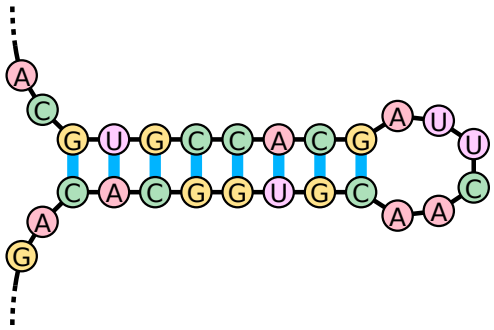


Figure 1.2: Example of an RNA secondary structure. Base pairings are colorized as two bases that are connected with a bold blue edge. This motif is called a hairpin. This image is from Wikimedia Commons. It is licenced under CC-3.0-by-sa. For more information see: `http://commons.wikimedia.org/wiki/File:Stem-loop.svg`

There are several variants of the basic algorithm that involve special scoring schemes for gaps, allow free gaps at the beginning and end or find the two substrings of the input with the highest alignment score (local alignment).

The model of edit-operations is an approximation of the reality in the cell, i.e. a genomic sub sequence in the genome of an organism encodes a molecule or some transcription information. Cell processes that lead e.g. to mutations of bases can be seen as edit operations. When comparing two sequences, an alignment score can be interpreted as measure for relatedness. Or an optimal alignment of a known gene sequence with an unknown sequence with many similarities in a genome can indicate a gene with similar functionality.

Hidden Markov models (HMMs) are used in bioinformatics for probabilistic modeling. A HMM is a set of (hidden) states. Each state emits different characters with certain probabilities. A model defines a set of transitions and state changes along possible transitions with specific probabilities. Given a HMM and a sequence of observed characters, the standard tasks are finding the most probable sequence of hidden states (Viterbi algorithm) and the probability that the model was in state X (Forward/Backward algorithm). Each algorithm uses the method of dynamic programming. Example applications are the search of sequence patterns, e.g. CPG-islands or the start and end of genes in the genome. Another application are profile HMMs, where a HMM is derived from an existing multiple alignment of a sequence family. The resulting HMM is used to find unknown family members in new sequence data.

HMMs have the same modeling power as regular grammars. Stochastic context free grammars (SCFGs) allow probabilistic modeling using the power of CFGs. Use cases are the search of sequence descriptions or patterns and the prediction of the secondary structure of RNA molecules. The secondary structure of an RNA

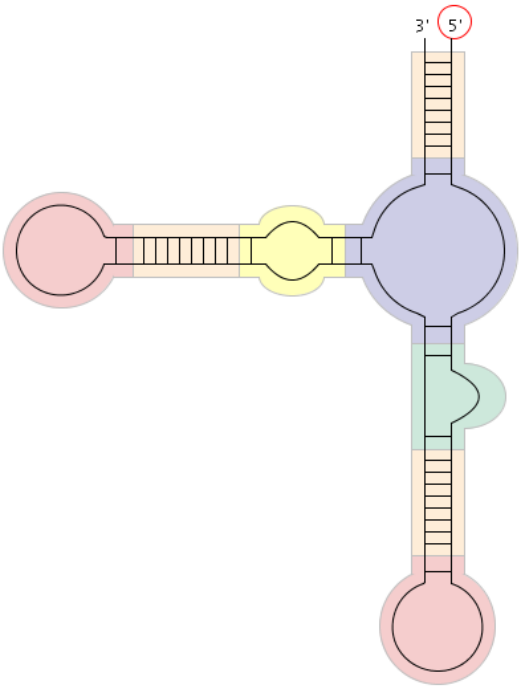


Figure 1.3: Example of an RNA motif, generated with the Locomotif GUI.

molecule is described by the set of base pairings. Figure 1.2 shows an example. In many processes in the cell, the structure of an RNA determines its function. Since different sequences may yield the same structure, only looking at the sequence data is not enough when comparing or searching RNA sequences. Analog to the Viterbi algorithm for HMM, the CYK algorithm [58] can be modified to get the most probable parse of a grammar for a given input sequence. To derive rule probabilities from a training set of inputs, the generalization of the Forward/Backward algorithm for SCFGs is the Inside/Outside algorithm.

Besides probabilistic modeling, CFGs are used to model the search space of possible secondary structure under a minimum free energy (MFE) model. Different structure elements have different free energy contributions to the complete structure of an RNA sequence. The used energy parameters are derived experimentally. The base of this model is that due to physicochemical laws, an RNA molecule in the cell folds itself such that the free energy is minimal. Minimization optimization using dynamic programming (e.g. implemented as a CYK style parser) computes the MFE value and the structure or structures that yield this MFE. Using Boltzmann statistics [54], the probability of structures or structure ensembles under the MFE model can be computed.

In RNA structure analysis, a CFG can be used to model the general search space of an RNA structure. In other use cases, a CFG restricts the search space to specific structural motifs, e.g. to the search space of all clover-leaf like structures (basically

three hairpins enclosed in a hairpin structure), which includes the family of tRNAs (transfer RNAs). Figure 1.3 shows another motif created with the graphical grammar generator Locomotif [38]. Grammars that model restricted structural motifs and use the MFE model are also called thermodynamic matchers (TDMs).

For searching new members of a family of sequences, covariance models are used. Such models use SCFGs, i.e. the use case is similar to profile HMMs, but context free grammars are used instead of regular grammars.

1.3 Related Dynamic Programming Frameworks

The theoretical base of Bellman's GAP is Algebraic Dynamic Programming (ADP, Section 2). ADP is a formal framework for specifying dynamic programming algorithms over sequences on a high level. It eliminates the use of indices, a major source of error in dynamic programming. In ADP, there is a separation of the description of the candidate structure, from the candidate evaluation, from the search space description and from tabulation concerns.

The following sections present and discuss languages, frameworks and libraries for dynamic programming that are related to ADP.

1.3.1 Dynamite

Dynamite [6] is a domain specific language (DSL) for specifying pairwise sequence alignment style DP algorithms. Its compiler generates C code. The main formal concept in the Dynamite language are finite-state machines (FSM). A Dynamite program defines states and transitions. Every transition is associated with a scoring function and two offsets that specify how many characters from each input sequence are consumed during that transition. The explicit specification of the offsets means that one major source of error in DP, the use of indices, is not completely eliminated from the Dynamite language. A state is directly mapped to a two-dimensional table, i.e. for each state all sub-solutions are tabulated.

For example, the Gotoh algorithm [23] for pairwise sequence alignment with affine gap-cost is modelled in Dynamite as FSM with three normal states: *match*, *delete* and *insert*. In addition, the two special states *start* and *end* are defined, which are not mapped to tables. A transition from the *match* to *delete* state corresponds to opening a deletion gap, i.e. the gap opening cost is associated to that transition. The first offset is 1 and the second offset is 0, since a deletion consumes one character from the first sequence and none from the second. Then a gap extension cost is associated to the *delete* state to *delete* state transition, etc.

The Dynamite compiler is coupled with a rich runtime library that includes data-structures and tools needed for bioinformatics sequence analysis, e.g. for integrating database searching, reading FASTA files or using protein scoring matrices. The object system that is used in Dynamite is designed for extensibility, e.g. to integrate a protein HMM or custom sequence types into a sequence alignment algorithm.

As a special code generation feature, the Dynamite compiler supports the generation of linear space alignment computation code which is a generalization of the Hirschberg algorithm [24].

The Dynamite language only supports optimization functions, e.g. minimization or maximization. Synoptic analyses of the search space, e.g. counting via summation, are not possible. The domain of the language comprises pairwise sequence style $O(n^2)$ DP algorithms. There is no support for generic Needleman-Wunsch [35] $O(n^3)$ style pairwise sequence alignment algorithms for using generic gap length functions. Likewise, sequence alignment algorithms for more than two sequences, generic single-track (e.g. $O(n^3)$ RNA secondary structure folding), generic two-track (e.g. the Sankoff algorithm [42], for simultaneous folding and aligning) or generic multi-track DP algorithms over sequences cannot be formulated in Dynamite.

1.3.2 Staging DP

A combinator library for specifying dynamic programming algorithms as multi stage programming (MSP) code is presented in [52]. It is implemented in MetaOCaml that provides elementary MSP constructs. The used MSP constructs are operators to quote expressions and to instruct the compiler to inline quoted expressions.

The combinators of the library are implemented as monads. They do not abstract from the matrix recurrences style of dynamic programming that is usually used in textbooks to present dynamic programming algorithms. This means that one major source of error in dynamic programming is not eliminated with the use of this library. The details of MSP are not encapsulated in the library.

Example implementations and benchmark results are shown for simple one-table dynamic programming algorithms. They are specialized on input sizes from 7 to 34 characters. The task of printing optimal candidates or backtracing is not covered by the library.

1.3.3 Dyna

Dyna [13] is a turing-complete declarative language for specifying weighted deductive programs, which are a generalization of probabilistic parsing. It was developed to simplify the development of algorithms and parsers in the field of natural language processing (NLP). Dynamic programming algorithms can be specified in Dyna, but Dyna is not restricted to dynamic programming.

The syntax of Dyna is Prolog based and some concepts originate from the world of deductive databases. A Dyna program is a set of deductive inference rules. The Dyna compiler generates optimized C++ code, whose main part is an agenda based parser. In NLP large grammars are not unusual and Dyna is able to handle grammars with over 100000 rules.

Since the scope of Dyna is not restricted to dynamic programming, dynamic programming related optimizations are not a main focus. For example, garbage collection is run in fixed intervals, even when an optimizing dynamic programming

algorithm on elementary data types implicates no need for any garbage collection, and in that case garbage collection only introduces overhead. Similar to that, the generic agenda based parsing loop introduces some overhead in practice.

In the examples, the printing of optimal candidates or backtracing is not mentioned. One alternative is to forward compute string representations of candidates with the scores or using an available graph browser to introspect the derivation space.

The Dyna language does not separate the search space description from the evaluation of candidates. For example the use of minimization, maximization or something else as optimizing function or the score scheme are embedded on the left hand side of the inference rules.

In Dyna the optimal sub-solutions are tabulated for sharing of sub-solutions, which yields an asymptotic optimal runtime of dynamic programming algorithms. The compiler does not include optimizations to reduce unnecessary tabulation.

Dyna supports optimizing program transformations like folding and unfolding, but they are not automatically applied by the compiler. The user has to instruct the compiler, where such transformations have to be applied.

1.3.4 Shortcut Fusion

The optimal sequence problem solving framework [33] provides functions to write dynamic programming algorithms specifications. It is implemented as a Haskell [28] library. It uses the functional programming technique of shortcut fusion, which is a program transformation for eliminating intermediate data-structures between function calls. The Haskell library implements several shortcut fusion rules as GHC (Glasgow Haskell Compiler) rewrite rules, which are applied by the GHC to the parse tree of the input program during compile time.

A dynamic programming algorithm is specified in the optimal sequence framework via a Haskell program that enumerates the problem search space and applies one or more objective functions to all candidates of the search space. These specifications are constructed using higher order functions from the library. When compiling the program via GHC, the rewrite rules are applied to the inefficient search space enumeration program and it is automatically transformed into an efficient optimization program.

Since the framework is implemented as a Haskell library, the user is confronted with errors from the Haskell systems exposing implementation details, when an optimal sequence specification contains an error. When a specification is written such that the rewrite rules cannot automatically be applied, an inefficient specification is transformed to an inefficient program without warning.

The framework does not eliminate the use of indices in search space specifications. The framework specifies rewrite rules for selective or scoring objective functions, as e.g. minimum or maximum. Synoptic objective functions, e.g. for computing the size of the search space, are not included.

The rewrite rules generate dynamic programming programs that tabulate every

derived recurrence. The nested framework functions need to keep intermediate candidate lists sorted, which may introduce an asymptotically suboptimal runtime factor. Similar to that the use of a binary map data structure for tabulating solution may asymptotically increase the runtime of the generated program.

[33] presents several example dynamic programming algorithms as optimal sequence specifications, e.g. Knapsack variants, the longest common subsequence algorithm and the optimal binary search tree algorithm. In several benchmarks the runtime of translated specifications, using the optimal sequence framework, are compared to the runtime of directly handwritten Haskell implementations. The generated programs are as fast or faster than the manually implemented dynamic programming versions, but the manually implemented versions are not tuned for efficiency. For example, the manually implemented version for some algorithms consumes a lot of memory in comparison to the generated version such that much runtime is spent during garbage collection.

2 Algebraic Dynamic Programming

Algebraic Dynamic Programming (ADP) is a formal framework for specifying dynamic programming algorithms on sequences. It clearly separates the concerns of search space description, candidate description, candidate evaluation and tabulation.

Tree grammars ($\mathcal{G}$) specify the search space, algebras ($\mathcal{E}$) evaluate candidate terms and signatures (Σ) declare the function reservoir which tree grammars and algebras are using. Tabulation is specified through non-terminal annotation in tree grammars. The use of tree grammars for search space description eliminates subscripts from the algorithm description, i.e. a major source of programming errors in developing DP algorithms.

Algebras are building blocks to wrap different scoring schemes or optimization strategies (h). With product operations they can be combined to more powerful analyses.

2.1 First Generation ADP

The ADP framework as published in 2004 is referred to as the first generation of ADP. This section defines the semantics of the basic GAP-L components. The definitions follow the semantic description of ADP in [20], Section 3.

To simplify the following definitions, we assume the case of one input track, one objective function and one sort. $\mathcal{A}$ denotes the alphabet of the input string. A signature Σ over $\mathcal{A}$ is a set of function symbols and a data type placeholder (sort) $\mathcal{S}$. The return type of each function symbol is $\mathcal{S}$, each argument is of type $\mathcal{S}$ or $\mathcal{A}$. T_Σ denotes the term language described by the signature Σ and $T_\Sigma(V)$ is the term language, where each term may contain variables from the set V. The regular tree grammar $\mathcal{G}$ is defined as tuple $(V, \mathcal{A}, Z, P)$, where V is the set of non-terminals, $Z \in V$ is the axiom, and P is the set of productions. Each production is of the form of Equation 2.1:

$$v \to t \text{ with } v \in V, t \in T_\Sigma(V) \tag{2.1}$$

The language generated by a tree grammar $\mathcal{G}$ is defined by Equation 2.2:

$$\mathcal{L}(\mathcal{G}) = \{t \in T_\Sigma | Z \to^* t\} \tag{2.2}$$

Definition 1 (Yield Function)**.** y denotes the yield function and is of type $T_\Sigma \to \mathcal{A}^*$. It is defined as $y(a) = a$, where $a \in \mathcal{A}$ and $y(f(x_1, \ldots, x_n)) = y(x_1) \ldots y(x_n)$, and $n \geq 0$, for each function symbol f from Σ.

The yield language $\mathcal{L}(\mathcal{G}, y)$ of a tree grammar $\mathcal{G}$ is defined by Equation 2.3.

$$\mathcal{L}(\mathcal{G}, y) = \{y(t) | t \in \mathcal{L}(\mathcal{G})\} \tag{2.3}$$

Definition 2 (Yield Parsing)**.** Computing the inverse of the yield function y is called *yield parsing*. The yield parser $\mathcal{Q}$ of a tree grammar $\mathcal{G}$ computes the search space of all possible yield parses:

$$\mathcal{Q}(x) = \{t | t \in \mathcal{L}(\mathcal{G}), y(t) = x\} \tag{2.4}$$

Note that a context-free parser for a yield language returns parse-trees and a yield parser returns elements from T_Σ. A user of Bellman's GAP needs not to care about how yield parsing works.

Definition 3 (Evaluation Algebra)**.** An evaluation algebra $\mathcal{E}$ for a signature Σ contains a function for every function symbol from Σ with the same arity. The algebra substitutes the sort symbol with a concrete type S', i.e. the i-th argument of the algebra function f is of type S' if the argument of functional symbol f is of type $\mathcal{S}$. In addition an evaluation algebra contains a objective function h of type $[\mathcal{S}'] \to [\mathcal{S}']$.

Definition 4 (ADP Problem Solution)**.** An ADP problem instance is specified by a grammar $\mathcal{G}$, evaluation algebra $\mathcal{E}$ and input sequence $x \in \mathcal{A}^*$. Its solution is defined by Equation 2.5.

$$\mathcal{G}(\mathcal{E}, x) = h_\mathcal{E}[\mathcal{E}(t) | t \in \mathcal{L}(\mathcal{G}), y(t) = x] \tag{2.5}$$

The square brackets in Equation 2.5 denote multi-sets, which are used to allow for co-optimal solutions or a restricted class of sub-optimal solutions.

A GAP-L program encodes $\mathcal{G}$ and $\mathcal{E}$, and running on input x, it produces the solution defined in Equation 2.5. The actual execution of a translated GAP-L program does not enumerate the search space, building all candidate trees and evaluating them. For efficiency reasons the objective function application is interleaved with the evaluation of candidate trees, which are not explicitly constructed. In functional language terminology, this is a case of deforestation. Tabulation of sub-solutions is used to eliminate re-computations that lead to an exponential runtime.

The prerequisite for correct and efficient computation of the solution is Bellman's Principle of Optimality [5], which in the ADP framework is defined by Equations 2.6 and 2.7.

Definition 5 (Bellman's Principle of Optimality)**.**

$$\begin{aligned} &h_\mathcal{E}[\, f_\mathcal{E}(x_1, \ldots, x_k) \mid x_1 \leftarrow Z_1, \ldots, x_k \leftarrow Z_k \,] = \\ &h_\mathcal{E}[\, f_\mathcal{E}(x_1, \ldots, x_k) \mid x_1 \leftarrow h_\mathcal{E}(Z_1), \ldots, x_k \leftarrow h_\mathcal{E}(Z_k) \,] \end{aligned} \tag{2.6}$$

$$h_\mathcal{E}(Z_1 \cup Z_2) = h_\mathcal{E}(\, h_\mathcal{E}(Z_1) \cup h_\mathcal{E}(Z_2) \,) \qquad \text{and} \qquad h_\mathcal{E}[\,] = [\,] \tag{2.7}$$

Bellman's Principle states that applying the objective function to sub-solutions or applying it to sub-multi-sets does not change the computation of the global optimum. An example of an objective function that does not satisfies Bellman's Principle is a function that selects the second best solution, e.g. the second largest score. This function violates Equation 2.7.

2.1.1 Algebra Products

While products of evaluation algebras do not strictly enhance the ADP theory, they are of enormous practical value. Products allow the easy combination of multiple objectives, e.g. to choose the largest pizza among the cheapest ones or to get a list of the best scored pizza place in every district.

Without products these combinations of objectives would need to be manually re-implemented in a new evaluation algebra, which computes on l-tuples for l objectives.

The definition of the lexicographic product operation follows [48, 49].

Definition 6 (Lexicographic Product)**.** Let A and B be evaluation algebras over Σ. The product $A * B$ is an evaluation algebra over Σ and has the functions

$$\begin{array}{l} f_{A\times B}(x_1,\ldots,x_k) = (f_A(a_1,\ldots,a_k), f_B(b_1,\ldots,b_k)) \\ \quad \text{if} \quad x_i = (x_i^A, x_i^B), \quad \text{then} \quad a_i = x_i^A, b_i = x_i^B, \quad x_i^A \in \mathcal{S}_A, x_i^B \in \mathcal{S}_B \\ \quad \text{if} \quad x_i \in \mathcal{A}, \qquad\quad \text{then} \quad a_i = x_i = b_i \end{array} \quad , 1 \le i \le k \tag{2.8}$$

for each $f \in \Sigma$, and the objective function

$$\begin{array}{l} h_{A*B}[(a_1,b_1),\ldots,(a_m,b_m)] = \\ \quad [\,(l,r) \mid \\ \qquad l \leftarrow \mathit{set}(h_A[a_1,\ldots,a_m]), \\ \qquad r \leftarrow h_B[\,r' \mid (l',r') \leftarrow [(a_1,b_1),\ldots,(a_m,b_m)],\ l' = l\,] \quad]. \end{array} \tag{2.9}$$

The expression $\mathit{set}(U)$ reduces the multi-set U to a set. Using the product *minPrice*·*maxSize* implements the first objective combination example, i.e. it selects the largest pizza from the cheapest ones, which is the lexicographic ordering of the two objectives, hence the name of the product. The second example is a classification as described in [48]. Every pizza candidate from the search space is classified into several districts. The product *district* · *best* implements these classification, where the objective function of the *district* algebra is the identity.

2.1.1.1 Example

In this section the Nussinov algorithm [36] is introduced as a running example to apply the ADP definitions and concepts. The Nussinov algorithm is a historic dynamic programming algorithm that takes one character string as input and computes the maximal number of character pairings. Not every character pair can form

a pairing, only certain complementary ones can do this. Each character can only be part of one pairing and two pairings are not allowed to cross each other, i.e. for every two pairings i, j and k, l with $i < j < n$, $k < l < n$ and $i < k$ it holds either $j < k$ or $i < k < l < j$, where n is the length of the input.

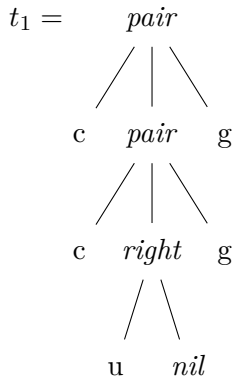
The first component of an ADP algorithm is the specification of the alphabet $\mathcal{A}$. Since the Nussinov algorithm is a historical landmark (see for example [12]) in the field of RNA secondary structure prediction an RNA alphabet is used in this example, i.e. $\mathcal{A} = \{a, u, c, g\}$. In RNA sequence analysis a character from $\mathcal{A}$ is called base and a string of characters is called sequence. An RNA secondary structure is defined by the set of base pairings. Figure 1.2 shows an example structure.

The signature contains the function symbols we need to describe the candidates of the search space:

$$\begin{aligned} \mathit{nil} &: && \to X \\ \mathit{right} &: X \times \mathcal{A} && \to X \\ \mathit{pair} &: \mathcal{A} \times X \times \mathcal{A} && \to X \\ \mathit{split} &: X \times X && \to X \end{aligned}$$

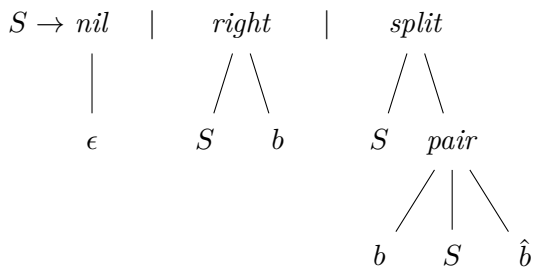
where X denotes the sort symbol. The function symbol *nil* describes the empty structure, *right* describes an unpaired base on the right of a sub-structure, *pair* describes a sub-structure enclosed by a base pairing and *split* describes two sub-structures side by side.

For example for the input sequence $x = ccugg$ the candidate term t_1 describes a structure with two base pairings:



The yield string (Definition 1) $y(t_1)$ equals the input sequence x.

Combining the functional symbols in every possible way yields an unnecessary large search space. Following grammar ($\mathcal{G}_{\text{nuss}}$) restricts the search space to all terms that represent well-formed RNA structures:



The symbol b is shorthand for a base from the alphabet and $\hat{b}$ is shorthand for a complementary base. For RNA the bases (a, u), (u, a), (c, g), (g, c), (g, u) and (u, g) are complementary and may pair with each other.

Algebras define how to evaluate the search space candidates. The following algebra *score* maximizes the base pairings:

$$\begin{aligned} nil &= 0 \\ right(x, e) &= x \\ split(x, y) &= x + y \\ pair(e, x, f) &= x + 1 \\ h(l) &= [\max l] \end{aligned}$$

It substitutes the sort symbol with an integer type. For printing the candidates as Vienna Strings [25], where each unpaired base is marked with a dot and each pairing with parentheses, the algebra *pretty* substitutes the sort for a string data type.

$$\begin{aligned} nil &= \text{“”} \\ right(x, e) &= x + \text{“.”} \\ split(x, y) &= x + y \\ pair(e, x, f) &= \text{“(”} + x + \text{“)”} \\ h(l) &= l \end{aligned}$$

where the + operator denotes string concatenation and the objective function is the identity. Since every candidate from the search space yields a different pretty print under this algebra the above Nussinov grammar is called semantically non-ambiguous.

Following algebra uses an objective function which differs from the ones before. It does not select elements from the input list, it possibly creates new elements. The algebra counts the search space defined by the grammar.

$$
\begin{aligned}
nil &= 0 \\
right(x, e) &= x \\
split(x, y) &= x \cdot y \\
pair(e, x, f) &= x + 1 \\
h(l) &= \left[\sum_{x \in l} x \right]
\end{aligned}
$$

Solving the ADP problem for algebra *score* given an input sequence x, i.e. solving $\mathcal{G}(score, x)$ yields a one element list of type int that contains the maximal number of base pairings. Using algebra *pretty*, $\mathcal{G}(pretty, x)$ returns a list of all candidates in Vienna String notation.

The product $score \cdot pretty$, i.e. solving $\mathcal{G}(score \cdot pretty, x)$, computes a list of tuples, where the left component stores the maximal number of base pairings and the right stores the Vienna String of the candidate that yields this score. Note that more than one candidate may yield the maximal number of base pairings. The product $score \cdot count$ computes how many.

For example for $x =$ ccaggg:

$$
\begin{aligned}
\mathcal{G}_{\text{nuss}}(score \cdot count, x) &= [\,(2, 3)\,] \\
\mathcal{G}_{\text{nuss}}(score \cdot pretty, x) &= [\,(2, \text{“((.)).”}),\ (2, \text{“((..))”})\,]
\end{aligned}
$$

2.1.2 Haskell Embedding of ADP

Haskell-ADP [20] is the implementation of ADP as embedded domain specific language (eDSL) in the purely functional programming language Haskell [28].

The signature of an ADP program is implemented in Haskell-ADP as parametrized type synonym tuple of type signatures of functions, where the alphabet and sort are the type parameters. An algebra is of the signature type, where the type parameters are set to concrete types, i.e. an algebra is a tuple of functions. A tree grammar is specified in Haskell-ADP with the help of parser combinators. Parser combinators are higher order functions to build parsers. Haskell-ADP defines several yield parsing combinators to parse the input. During the yield parsing the evaluation algebra functions are applied such that the resulting program is efficient.

In the following the implementation of the Haskell-ADP combinators is exemplified.

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

A sub-word of the input is represented as a tuple of integers. It is not a Haskell string to save space. The input is available as an array, i.e. $(0, n)$ marks the complete

input string, where n is the length of the input. A parser combinator is of type `Parser`, which is parametrized with the type of the results. A parser takes a subword as input and returns a list of successes, i.e. if it cannot parse the input it returns the empty list.

```
> infixr 6 |||
> (|||)            :: Parser b -> Parser b -> Parser b
> (|||) r q (i,j) = r (i,j) ++ q (i,j)
```

The first line specifies the associativity and the priority of an operator in Haskell. This combinator implements an alternative of two grammar rules. Both argument parsers are called and the results are concatenated.

```
> infix  8 <<<
> (<<<)            :: (b -> c) -> Parser b -> Parser c
> (<<<) f q (i,j) =  map f (q (i,j))
```

The apply parser combinator applies an algebra function to the arguments parsed by argument parsers.

```
> infixl 7 ~~~
> (~~~) :: Parser (b -> c) -> Parser b -> Parser c
> (~~~) r q (i,j) =  [f y | k <- [i..j], f <- r (i,k),
>                           y <- q (k,j)]
```

The next parser combinator nests argument parsers.

```
> infix  5 ...
> (...)            :: Parser b -> ([b] -> [b]) -> Parser b
> (...) r h (i,j) = h (r (i,j))
```

This parser is used to specify an objective function application in grammar rules. The objective function evaluates the parse result list of the parser on the left hand side.

Haskell-ADP contains further parser combinators for syntactic filtered parsing and to specify tabulation of parse results. It also includes specialization of the basic parser combinators and terminal parsers.

2.1.2.1 Example

In this section the Nussinov algorithm from Section 2.1.1.1 is implemented in Haskell-ADP.

```
> type Signature  alphabet sort = (
>   ()       -> sort,                              -- nil
>   sort     -> alphabet -> sort,                  -- right
>   alphabet -> sort     -> alphabet -> sort,      -- pair
>   sort     -> sort     -> sort,                  -- split
>   [sort]   -> [sort]                             -- h
>   )
```

The signature is a parametrized type synonym with the parameters alphabet and sort. The names of the signature function symbol are written as Haskell comments for documentation purposes. The programmer has to be careful to use the symbol names in algebra and grammar definitions in the same order.

```
> nussinov alg inp = axiom s where
>   (nil, right, pair, split, h) = alg

>   s = tabulated (
>         nil <<< empty |||
>         right <<< s ~~- base |||
>         split <<< s ~~+
>           ((pair <<< base -~~ s ~~- base)
>             `with` basepairing)              ... h
>       )
```

In the grammar definition, variants of the next parser combinator are used that reduce the number of considered sub-word splits for an efficient evaluation. All parse results for all sub-words are tabulated and the second argument of the function symbol split is only parsed, if the `basepairing` filter identifies two complementary bases at that location. The filter is a direct translation of the $b, \hat{b}$ shorthand notation in Section 2.1.1.1.

The *score* and *pretty* algebras are implemented as tuple of Haskell functions:

```
> score :: Signature  Char Int
> score = (nil, right, pair, split, h) where
>    nil _        = 0
>    right   x _  = x
>    pair  _ x _  = x + 1
>    split x y    = x + y
>    h []         = []
>    h xs         = [maximum xs]

> pretty :: Signature Char String
> pretty = (nil, right, pair, split, h) where
>   nil _        = ""
>   right  l b   = l ++ "."
>   pair a l b   = '(':l ++ ")"
>   split l1 l2 = l1 ++ l2
>   h            = id
```

The implementation of the lexicographic product directly follows Definition 6:

```
> infix ***
> alg1 *** alg2 = (nil, right, pair, split, h) where
>   (nil1, right1, pair1, split1, h1) = alg1
>   (nil2, right2, pair2, split2, h2) = alg2
```

Table 2.1: Examples of Haskell-ADP code and the equivalent ADPC-ADP code, where l denotes a list of integers, u a string and c a character. In the first three lines the left column is not valid ADPC-ADP code and vice versa the right column is not valid Haskell code. In the last line the left column is not valid ADPC-ADP code, but the right column is valid Haskell code (it yields a runtime error if l is empty).

Haskell-ADP	ADPC-ADP
`h l = l`	`h x = [id x]`
`u ++ [c]`	`u ++ c`
`c:u`	`c ++ u`
`h [] = [] ; h l = [minimum l]`	`h l = [minimum l]`

```
>   nil     a                 = (nil1 a,        nil2 a)
>   right     (x1, x2) a      = (right1 x1 a,   right2 x2 a)
>   pair    a (x1, x2) b      = (pair1 a x1 b, pair2 a x2 b)
>   split (x1, x2) (y1, y2) = (split1 x1 y1, split2 x2 y2)
>   h xs = [ (x1, x2)| x1 <- nub $ h1 [ y1 |
>                            (y1,y2) <- xs],
>                         x2 <-       h2 [ y2 |
>                            (y1,y2) <- xs, y1 == x1]]
```

Given the described elements of the Nussinov algorithm implemented in Haskell-ADP the expression $\mathcal{G}_{\text{nuss}}(score \cdot pretty, x)$ is equivalent to the Haskell expression `nussinov (score *** pretty) x`.

2.1.3 The ADP Compiler

The ADP compiler (ADPC) [47] was the first effort to compile ADP programs to imperative code. It is written in Haskell and generates C code by default. It contains an alternative backend that generates Java code [43]. The input language of the ADPC is a Haskell-ADP dialect (in the following called ADPC-ADP). The compiler implements semantic analyses, e.g. yield size analysis, table dimension analysis and table design. ADPC includes a type-checker that checks the ADP programs after a successful parse (see Section 5.3.8 for a discussion).

With the help of the ADPC, several RNA related bioinformatics tools were created, e.g. RNAshapes [51], pknotsRG [40] and RNAhybrid [41]. The tools were prototyped in Haskell-ADP which does not scale well, as benchmark results show (Section 8). For a discussion of the reasons see Section 3.1. Using the ADPC to compile these tools made it possible to create versions that are usable or perform better on sequence input of real world sizes. The table-design heuristic derives good results for some grammars.

ADPC-ADP uses the layout rules of literate Haskell: each code line has to begin

with a `>` character, all other lines are comments and a more indented line starts a new block (*off-side rule*). Other similarities to Haskell-ADP are the three character ADP combinators and the Haskell tuple style notation of algebras. Haskell-ADP contains several compiler directives (pragmas) that mark code blocks as signature, algebras and grammars. The ADPC-ADP programs with pragmas and other constructs are invalid Haskell-programs. Basic data structure (e.g. list) operations in ADPC-ADP are inspired by the Haskell or Haskell-ADP syntax, but show some differences. Table 2.1 gives some examples. The differences complicate the type-checking of ADPC-ADP programs. The language ADPC-ADP is not specified in a document.

In case of a parse error the ADPC prints a generic error message with only the line number and no further diagnostic. The ADPC is separated into two programs. The actual compiler `adpcompile` and a frontend program `adpc` that inspects the ADP source file and calls `adpcompile` with a set of low-level options. The frontend generates a command-line interface, a makefile and does some post-processing of the generated C-Code. The end-user of the compiler is expected to use the `adpc` frontend. The frontend does not allow a selection of products, it calls `adpcompile` for the algebras it finds and some simple products like `score *** pretty`. A side effect of the frontend is that it does not check the status or error messages of `adpcompile`. For example, a parse error is not printed to the console. Instead it is written to a C module. The end-user is then confronted with cryptic error messages of the C-compiler if the generated makefile is executed.

The included heuristic table design algorithm does not derive good results for some practical grammars, e.g. auto generated thermodynamic matcher grammars. A problem is that table configurations are derived which yield an asymptotic optimal runtime, but with prohibitively large constant factors.

For the above tools, the ADPC created a basic code structure which was then heavily post-processed in a semi-automatic fashion adding hand-written code to implement features like stochastic-backtracing or more complicated products. The compiler support of certain products is not available for all grammars. For example, ADPC cannot generate shape classifying products for the pknotsRG grammar [39].

ADPC is not advertised to support dynamic programming on multiple input sequences (multi track DP). Probably it includes limited two-track support to compile RNAhybrid [41].

The benchmark chapter (Chapter 8) contains runtime and memory usage comparisons of selected ADP grammars compiled with the ADPC and GAP-C.

2.2 Second Generation ADP

The notion *Second Generation ADP* subsumes extensions of the ADP framework that were mainly developed after 2004 and are unpublished. The interleaved product was developed by Robert Giegerich. The role classification scheme of algebras and the generalization of ADP for multiple input tracks have been developed as

parts of this thesis.

2.2.1 Products

Definition 7 (Unitary Algebra)**.** An objective function that returns lists of size one at most is called a unitary objective function. An algebra containing only unitary objective functions is called a *unitary algebra*.

Definition 8 (Cartesian Product)**.** Let A and B be unitary evaluation algebras over Σ. The cartesian product $A \times B$ is an evaluation algebra over Σ and has the functions

$$\begin{array}{lll} f_{A\times B}(x_1,\ldots,x_k) = (f_A(a_1,\ldots,a_k), f_B(b_1,\ldots,b_k)) & & \\ \quad \text{if} \quad x_i = (x_i^A, x_i^B), & \text{then} \quad a_i = x_i^A, b_i = x_i^B, & x_i^A \in \mathcal{S}_A, x_i^B \in \mathcal{S}_B \\ \quad \text{if} \quad x_i \in \mathcal{A}, & \text{then} \quad a_i = x_i = b_i & \end{array} \quad , 1 \leq i \leq k \tag{2.10}$$

for each $f \in \Sigma$, and the objective function

$$\begin{array}{l} h_{A\times B}[(a_1,b_1),\ldots,(a_m,b_m)] = \\ \quad [\,(l,r) \mid \\ \qquad l \leftarrow h_A[a_1,\ldots,a_m], \\ \qquad r \leftarrow h_B[b_1,\ldots,b_m] \quad]. \end{array} \tag{2.11}$$

Using the cartesian product and just computing $\mathcal{G}(A \times B, x)$ provides no strong advantages compared to computing $\mathcal{G}(A, x)$ and $\mathcal{G}(B, x)$ separately, except the first is a bit faster in practice. However, the cartesian product is used as part of larger products where the combination is advantageous to several redundant re-computations of other parts of the products.

Definition 9 (Generic Algebra)**.** A generic evaluation algebra $A(k)$ has a parameter k and an objective function that returns the k best solutions.

Definition 10 (Interleaved Product)**.** Let A be a Σ-algebra and $B(k)$ a generic Σ-algebra such that $B(1)$ is unitary. The interleaved product $(A \otimes B)(k)$ is a generic Σ-algebra and has the functions

$$f_{A\otimes B} = f_{A\times B} \tag{2.12}$$

for each $f \in \Sigma$, and the objective function

$$\begin{array}{l} h_{(A\otimes B)(k)}[(a_1,b_1),\ldots,(a_m,b_m)] = \\ \quad [\,(l,r) \mid (l,r) \leftarrow U, p \leftarrow V, p = r] \\ \quad \text{where} \\ \qquad U = h_{A*B(1)}[(a_1,b_1),\ldots,(a_m,b_m)] \\ \qquad V = \text{set}(h_{B(k)}[\,v \mid (\_,v) \leftarrow U]) \end{array} \tag{2.13}$$

The interleaved product is an extension of the lexicographic product for more restricted classification purposes. Consider the product $A \cdot B$, where algebra A classifies every solution into a class and algebra B scores and selects the best solution. The result of the product is the best solution of every class. Using a k-scoring algebra, an algebra that selects the k-best solutions for B results in the k-best solutions of every class. Using the lexicographic product, it is not possible to compute the k-best classes with the best solution of every class. However, the interleaved product $(A \otimes B)(k)$ does exactly this.

2.2.2 Generalizations

In the case of algebras containing more than one objective function, the objective function $h_{\mathcal{E}}$ splits into several functions $h^1_{\mathcal{E}}, h^2_{\mathcal{E}}, \ldots$ which all have to satisfy Bellman's Principle of Optimality. In the case of more than one sort, the basic evaluation semantics does not change. For multi-track programs, the return type of the yield function y is the l-tuple of strings from $\mathcal{A}^*$, where l is the number of used tracks. If the GAP-L program computes on l tracks, then Equation 2.14 specifies the semantics.

$$\mathcal{G}(\mathcal{E}, x_1, \ldots, x_l) = h_{\mathcal{E}}[\mathcal{E}(t) | t \in \mathcal{L}(\mathcal{G}), y(t) = (x_1, \ldots, x_l)] \tag{2.14}$$

2.2.3 Algebra characteristics

In Bellman's GAP the compiler inspects the algebras and products and computes two attributes of the objective functions. The first attribute is the role of an objective function and the second one is the maximal length of the list the objective function may return. The reason for this characterization is threefold. First, this information imply in code generation that optimizations like list elimination or the use of hash table data structures in classifying products are safely applicable. Second, when analysing the roles, the compiler is able to print error messages in cases, where it is sure that the product does not satisfy Bellman's Principle. Third, depending on the computed roles, the compiler is able to issue warnings about products that will produce exponentially sized answer lists.

Consider e.g. a scoring algebra with one objective function that computes the minimum of the input list. If the input list is empty, then the empty list is returned, else the minimum value is returned. The ADP framework assumes that every objective function returns a list of successes. In this example, the list is always of size zero or one. Thus, in the generated code there is not really a list data structure needed for dealing with the result of the objective function. Instead of a list of type T, a variable of type T is sufficient to store the result of the objective function. By convention a value from the domain, like e.g. 'infinity', encodes the empty list. Doing such an optimization, the runtime of the resulting code is improved, because list operations are more expensive than elementary data type operations, memory is saved and the CPU's cache is more efficiently utilized.

An example where a whole class of products does not satisfy Bellman's Principle is count$\cdot B$, where the objective function of the algebra count sums over all elements of the input list and B is an algebra that satisfies Bellman's Principle. Following the definition of the lexicographic product (Equation 2.9), the objective function h_1 of the count algebra produces a new value which is not included in the first components of the input list elements, unless the input list is empty or contains just one element and the first component is set to 1. Thus, the answer list of the product's objective function h is always the empty value, unless the input list contains one tuple with a value of 1 in the first component. Consider an input list l, where $l = [(1, \_), (1, \_)]$ and a split of the input list $l = l_1 + + l_2$, where $l_1 = [(1, \_)]$ and $l_2 = [(1, \_)]$. Then, $h(l_1 + + l_2) \neq h(h(l_1) + + h(l_2))$, which violates Bellman's Principle (Equation 2.7).

Examples of algebras that usually yield exponentially sized answer lists are the algebra *pretty* or the product *pretty* $\cdot$ *score* where the objective function of *pretty* is the identity and the objective function of *score* returns the maximal or minimal value of the input list. Both products enumerate the (usually) exponentially sized search space.

If the algebra contains just one objective function, then the role of the algebra is defined as the role of that objective function. Since GAP-L supports algebras with multiple objective functions, the roles of each objective function can differ. In such cases, the role of such algebras is only defined, if all objective functions have the same role. The concept of a role of an algebra is just a short hand notion, internally for the optimizations the compiler works with the roles of the objective functions. A product of two algebras is again an algebra.

GAP-C classifies each algebra into one of four roles: selective, enumerative, set-valued and synoptic. A selective objective function selects one or more elements from the input list according to an optimization criterion. An example for a selective algebra is an algebra that computes scores and minimizes them. An enumerative objective function just returns the input list. For example, a pretty printing algebra is enumerative. A set-valued objective function removes all duplicates from the input list. A synoptic objective function is a function that does not select elements from the list, but computes new values from the input. An example of a synoptic algebra is a counting algebra, where the search space of a GAP-L grammar is counted. The objective function then is the sum of the input list. The following definition summarizes the properties of the various roles:

Definition 11 (Algebra roles)**.** An evaluation algebra A with objective function h_A, with $h_A([\,]) = [\,]$, is

- enumerative, if $h_A(X) = X$,
- set-valued, if $h_A(X) = \text{set}(X)$,
- selective, if $h_A(X) \subseteq X$,
- synoptic, if $|h_A(X)| = 1$ and $\exists X : h_A(X) \not\subset X$

Table 2.2: The roles and the maximum return list sizes of objective functions detected by the compiler.

objective function	role	maximum length
`return list(minimum(l))`	selective	1
`return list(maximum(l))`	selective	1
`return list(sum(l))`	synoptic	1
`return l`	enumerative	n
`return unique(l)`	set-valued	n

for all multisets $X \neq [\,]$.

The compiler tries to detect the role of each evaluation objective. Table 2.2 shows the detected role of certain objective function expressions. If a user-defined objective function is used, then the compiler assumes a scoring role with possibly more than one optimal element returned. When defining an objective function in GAP-L, it is possible to explicitly specify the role of the objective function (see Section 4.5.5.1).

Combining two algebras with the cartesian, lexicographic or interleaved product operation creates a new algebra. Table 2.3 shows the roles and worst-case list sizes of the resulting objective functions depending on the left and right multiplicand. Most cells in the cartesian product matrix are marked as not satisfying Bellman's Principle, because the definition of the cartesian product operation requires that each operand is a unitary algebra (Definition 8) . Likewise, the first operand of the interleaved product has to be a enumerative or set-valued algebra and the second operand has to be a selective algebra with an answer list greater one (Definition 10). For the lexicographic product, combining two enumerative objective functions yields a permutation of the input list, which follows directly from the product definition (Definition 6). First, all the left components of the input list are extracted and all duplicates are removed. Then for each extracted component the corresponding tuples are selected that match the left component. Analogous to that, the product of a set-valued objective function and an enumerative one is a permutation of the input list, too. In the ADP framework a permutation of the answer list of an objective function does not matter, since the lists are viewed as multi-sets, where the order of elements is not important. Thus, during code generation the compiler is able to eliminate the objective function application in enumerative products or in enumerative sub-products, which leads to more efficient code.

Table 2.3: Resulting roles of the different product operations (sel → selective, enum → enumerative, set → set-valued, syn → synoptic). The worst-case length of the returned lists is specified in parentheses, where n is the size of the input list. A - entry in the table means non-preservation of Bellman's Principle.

(a) cartesian

×	sel (1)	sel (n)	enum (n)	set (n)	syn (1)
sel (1)	sel (1)	-	-	-	syn (1)
sel (n)	-	-	-	-	-
enum (n)	-	-	-	-	-
set (n)	-	-	-	-	-
syn (1)	syn (1)	-	-	-	syn (1)

(b) lexicographic

·	sel (1)	sel (n)	enum (n)	set (n)	syn (1)
sel (1)	sel (1)	sel (n)	sel (n)	sel (n)	syn (1)
sel (n)	sel (n)	sel (n)	sel (n)	sel (n)	syn (n)
enum (n)	sel (n)	sel (n)	enum (n)	set (n)	syn (n)
set (n)	sel (n)	sel (n)	enum (n)	set (n)	syn (n)
syn (1)	-	-	-	-	-

(c) interleaved

×	sel (1)	sel (n)	enum (n)	set (n)	syn (1)
sel (1)	-	-	-	-	-
sel (n)	-	-	-	-	-
enum (n)	-	sel (n)	-	-	-
set (n)	-	sel (n)	-	-	-
syn (1)	-	-	-	-	-

3 Bellman's GAP Overview

Bellman's GAP is a programming system for developing dynamic programming algorithms over sequences in the ADP framework. It is named after the most important ADP concepts:

- Bellman's Principle of Optimality (Definition 5)
- Grammars
- Algebras
- Products (Section 2)

Bellman's GAP consists of the following parts:

- *GAP-L* — a declarative language with C/Java like Syntax for specifying ADP programs. Grammars are specified in a declarative style, where the right hand side of non-terminals contains tree patterns resembling function calls. The algebra code is written as imperative code blocks. Instance declarations allow the combination of algebras with product operations to new algebras.
- *GAP-C* — a novel optimizing compiler that translates GAP-L programs to efficient C++ code, which is competitive with handwritten code.
- *GAP-M* — a runtime library for compiled GAP-L programs including efficient implementations of internal data-structures and a module providing convenience functions for accessing energy parameters and computing energy contributions as used in RNA secondary structure prediction.
- *GAP Pages* — an educational website providing an interactive interface to a set of GAP-L examples.

The following section describes the motivation behind designing a new language for ADP and to choose the route of compiling ADP programs instead of embedding it into another language. Section 2.1.3 has discussed the previous ADPC compiler, the first compiler translating ADP to C, and has mentioned its shortcomings.

GAP-L is presented in Chapter 4, where the design goals are discussed, the syntax is specified and new ADP features are shown. Chapter 5 describes GAP-C, presenting the overall architecture of the compiler, specifying several optimization algorithms used in semantic analyses of the compiler, e.g. a novel table design algorithm, and the used code generation techniques, e.g. for parsing or parallelization.

In Chapter 6 GAP-M modules are presented, the optimized implementation of several internal data-structures, like e.g. memory pools and strings, are discussed and the RNA domain specific helper library is specified. GAP Pages is presented in Chapter 7 and Chapter 8 includes several benchmarks of GAP-L versions compiled by GAP-C against ADPC-ADP versions, Haskell-ADP versions and manual coded implementations.

3.1 Limitations of Haskell-embedded ADP

The first implementation of ADP was done as embedded domain specific language (eDSL) in Haskell [21]. Haskell [28] is a general purpose purely functional programming language that uses lazy-evaluation by default. In the following the ADP embedding is called Haskell-ADP.

The motivation for designing a new language and a stand-alone compiler for ADP are two-fold. First, compiled Haskell-ADP does not perform and scale well, see Chapter 8 for benchmarks of Haskell-ADP versions. The reason for this is that the Haskell system does not have knowledge of ADP to apply ADP-specific optimizations. For example, the Haskell system cannot eliminate the use of lists if only a scoring algebra is used. Other factors are the lazy-evaluation and garbage collection of Haskell. Lazy-evaluation means that expressions in functional programs are only evaluated if needed. Lazy-evaluation pays off in cases, where the overhead of bookkeeping, which expressions are or are not computed, is less than the strict computation of all expressions. For example, if the structure of the search space implies that only a few entries of a DP table are used, only these entries are computed with lazy-evaluation. However, if an expression triggers the computation of all subexpression, then lazy-evaluation is just overhead. In some cases a Haskell compiler can optimize unnecessary lazy-evaluation away and the programmer can place strictness annotations in the code. But in Haskell-ADP, the heavy use of higher-order functions (parser combinators) makes it difficult to debug locations where lazy-evaluation or strictness would improve the runtime.

Garbage collection is a runtime system that automatically deletes objects which are not needed any more, and frees their memory. Doing an efficient garbage collection is a hard problem. If e.g. an object is reused several times in the subsequent program execution, then it makes sense to keep it around to avoid re-computations. On the other hand, keeping objects too long wastes memory. Then there is the design decision, how often the garbage collection should run and interrupt the execution of the program. A garbage collection algorithm has to implement a policy if there is memory pressure: which objects to delete first and which cannot be deleted at all. The general trade-off in garbage collection is memory usage efficiency vs. the runtime complexity of the garbage collection algorithm. In practice, the garbage collection often leads to extensive memory consumption for Haskell-ADP programs (see Chapter 8).

In addition, Haskell-ADP uses heavy-weight data-structures. For example, a

```
ERROR "Optbin.lhs":89 - Inferred type is not general
  enough
*** Expression    : tree
*** Expected type : Tree_Algebra Alphabet a
                     -> [Alphabet] -> [a]
*** Inferred type : Tree_Algebra Alphabet ()
                     -> [Alphabet] -> [()]
```

Figure 3.1: Error message example of an ADP error in Haskell-ADP issued by the Haskell interpreter Hugs: in the grammar definition a function symbol is used with the wrong number of arguments.

Haskell string is a list of character objects and even in the optimized case a character uses 12 bytes [9] (using GHC on a 32 bit architecture).

A dedicated ADP compiler can avoid the problems of the ADP Haskell embedding in its code generation. The compiler can inspect the input program and implement several ADP specific optimizations, e.g. list eliminations or table dimension reduction. It is able to restrict lazy-evaluation to cases, where it is likely to pay off. The need for garbage-collection can be eliminated for making dynamic programming programs more efficient, by using an explicit memory management code.

Second, the embedding of ADP in Haskell has some usability implications. Since the Haskell system does not know the semantics of ADP, a (perhaps trivial) error in an ADP program may yield generic and/or long type-inference error reports from the Haskell system. Figure 3.1 shows an error message example. Such messages suppose some knowledge of the Haskell programming language and the implementation details of Haskell-ADP. The design of Haskell-ADP is restricted by its host language. For example, Haskell-ADP has to use the off-side rule. The off-side rule means that the indentation of lines defines where code blocks start and end. To minimize overloading problems with existing operators, the used parser combinators have a length of 3 characters.

A main target audience are bioinformaticians, who are usually not trained Haskell programmers. For them the off-side rule is a new, complicated concept and type-inference error messages are “opaque”. In any case, an ADP programmer should not need to know the implementation details of Haskell-ADP to understand the error messages.

An ADP language implementation outside of Haskell does not need to consider Haskell constraints. For example, the off-side rule is not used and the syntax of the grammar declaration can be designed according to a function-like notation with the well known single character for an alternative operator. Algebra functions do not need to be implemented as Haskell functions. An ADP compiler is the right place to implement an ADP specific type checker (Section 5.3.8) that generates more useful error and warning messages.

4 Bellman's GAP Language

Bellman's GAP Language (GAP-L) is the 2nd generation domain specific language for programming ADP. Its syntax is Java/C like but GAP-L includes declarative constructs for several elements of an ADP algorithm, e.g. for specifying the grammar. It is not embedded into a host language to avoid unnecessary constraints in the design of the language. GAP-L implements the features of the first generation ADP (Section 2.1) and later ADP extensions (Section 2.2). Several concepts, like e.g. syntactic filtering, are generalized in GAP-L and advanced DP techniques, like e.g. sampling, are available via the language in a general way.

The next section discusses the design goals of GAP-L. In Section 4.2 the new ADP features in GAP-L are listed with forward references. The discussion of a GAP-L version of the running Nussinov example in Section 4.3 exemplifies the main syntactic elements of a GAP-L program. Section 4.4 describes the lexical structure and Section 4.5 the syntactic structure using a CFG of an GAP-L program. Finally, Section 4.6 presents advanced language features, like e.g. filtering or multi-track DP that cover several language constructs.

4.1 Design Goals

The main design goal of GAP-L is: It should be easy to learn and to use for new users and it should be usable effectively by ADP experts.

A main target audience of GAP-L are bioinformaticians. Most likely an undergraduate bioinformatics student has some knowledge of an imperative programming language with C-like Syntax like Java or C/C++. Therefore the syntax of large parts of GAP-L is C-like. Using known syntax elements and concepts lowers the barrier of learning and using GAP-L for new users. The signature declaration in GAP-L resembles an interface declaration in Java. An algebra *implements* a signature like a Java class may implement a Java interface. To highlight the reuse of algebra functions, GAP-L has the concept of algebra inheritance. An algebra may *extend* another algebra and only overwrite single functions, as classes can extend other classes in Java. The syntax of algebra function code in GAP-L is C-like, too. The complexity of dealing with data structures is encapsulated in the runtime library that provides high-level data structure operations and tools via a function based API. Thus, GAP-L does not need to provide object oriented language features or pointer-arithmetic. The language for defining algebra functions is basically a mini-Java.

As a consequence of using a C-like syntax and avoiding the off-side rule, GAP-L is easier to parse. Easier parsing results in easier generation of helpful warning

and error messages which helps to satisfy the above main goal. For example, the tracking of exact locations is simplified if no pre-processing of the off-side layout is done.

For the ADP expert, GAP-L provides advanced features like extended filtering constructs in grammar rules and products, instance declarations, generic support for products, parametrized non-terminals and the explicit manipulation of indices where needed. See Section 4.2 for an overview.

Both, the ADP beginner and the ADP expert profit from the GAP-L grammar syntax. The tree grammar patterns on the right hand side of the non-terminal definition are function like. There are no three character parser combinators. The arguments of a function symbol are separated by commas. Special versions of next-to combinators, like in Haskell-ADP, which separate the arguments, are not needed, because the compiler automatically optimizes the moving index boundaries between the arguments (Section 5.3.12). Thus, a GAP-L grammar looks like pseudo-code grammar notation. Again, this syntax is not only easier to read, but also easier to parse and easier to typecheck.

4.2 New ADP features

GAP-L is an implementation of the ADP framework. The ADP concepts of alphabet, signatures, evaluation algebras, tree grammars and products are available in GAP-L. In addition to that, it introduces the following new concepts. The detailed description in later sections is referenced in the listing of features.

- An algebra can *extend* other algebras, i.e. in the extended algebra, algebra functions can be overwritten or added (Section 4.5.5 and 4.6.1).
- Algebras that count the search space or print the candidate term as ASCII serialization are built mechanically after the structure of the signature. In GAP-L the user does not need to program them for each new signature. It is possible to declare them as *automatic* and the compiler automatically generates them (Section 4.5.5).
- A GAP-L program may contain several *instance* declarations that specify different products (Section 4.5.9). For example, instance names allow to reference complicated products when compiling a GAP-L program.
- GAP-L includes new product operations (Section 4.5.9). The take-one product is a specialization of the lexicographic product that ignores co-optimal candidates. The overlay product allows the specification of different algebra during the forward computation and backtracing. The cartesian and interleaved products are not new, but are recent innovations from the ADP community. All product operations are directly supported by GAP-C, i.e. the compiler automatically generates optimized code that has the semantics of the product operation. In comparison to that, in Haskell-ADP, the user is

required to mechanically program the product operation for each signature from scratch as Haskell-Code.

- The concept of syntactic *with*-filters in ADP is generalized in GAP-L (Section 4.5.8.5 and 4.6.2). The patterns of tree grammars can now contain *syntactic filters* that take more than one function symbol argument at once into account. In addition, grammar patterns can be restricted by *semantic filters* that filter on the values of the grammar parsers that result from the referenced grammar pattern.

- Semantic filtering is possible in products, as well (Section 4.5.9 and 4.6.3). This allows to reuse product filters for different products and separate filtering concerns from optimization concerns in the evaluation function of the algebra.

- *Parametrized non-terminals* (Section 4.5.8.1) allow to declare non-terminals in the tree grammar that have arguments. This allows, for example, to restrict the recursion depth of recursive rules or pass context depending information around and feed them into algebra functions.

- The ADP framework is specified for dynamic programming over one input sequence (track). GAP-L generalizes ADP to *multiple input tracks* (Section 4.5.8.2 and 4.6.4). For example, parsers in the tree grammar can read from one or multiple input tracks. It is possible to integrate single track grammars into multi track ones.

- ADP eliminates indices from the search space description. However, in some desperate cases the access and manipulation of moving index boundaries is necessary for efficiency reasons. In GAP-L some language constructs are provided for specifying *explicit indices* and moving index boundaries (Section 4.5.8.4 and 5.4.5). This makes it possible to specify large parts of a dynamic programming algorithm high-level in ADP and have at the same time the possibility to manipulate low-level indices only in single locations.

4.3 Example

Before introducing the syntax of GAP-L in all detail in the next two sections, the basic elements of an ADP algorithm are shown implementing the Nussinov example from Sections 2.1.1.1 and 2.1.2.1 in GAP-L.

The signature is implemented in GAP-L via the following declaration:

```
signature Nuss(alphabet, answer) {

  answer nil(void);
  answer right(answer, alphabet);
  answer pair(alphabet, answer, alphabet);
  answer split(answer, answer);
```

```
  choice [answer] h([answer]);

}
```

The **alphabet** is a placeholder for the type of the input characters and **answer** is the sort, i.e. the placeholder for the value of a candidate under an evaluation algebra. Function symbol signatures are explicitly named. The name is not placed inside a comment as in Haskell-ADP. The order of the function symbol signature declarations does not matter. The syntax of the function symbol declarations is Java/C-like, the type of the return value is written before the function symbol name and not at the end, as in Haskell. The **choice** modifier marks the objective function symbol.

The following declaration shows the Nussinov grammar in GAP-L syntax:

```
grammar nussinov uses Nuss (axiom=struct) {
  struct = nil(EMPTY)            |
           right(struct, CHAR)   |
           split(struct,
                 pair(CHAR, struct, CHAR)
                    with char_basepairing) # h ;
}
```

Tree patterns on the right hand side are written in a function like notation and there are no three-character wide parser-combinator-like operators. In particular, there are no next-combinator variants. Arguments of a function symbol application are separated by commas because the GAP-L compiler automatically optimizes moving index boundaries in the generated code using results from the yield size analysis (Section 5.3.12).

After the description of the search space and the signature, we need to define algebras to assign a meaning to each candidate and specify how to optimize over the different candidates. The easiest algebra specifications are automatic ones:

```
algebra co auto count ;
algebra en auto enum ;
```

The first one generates an algebra that counts the search space under the given tree grammar and the second enumerates the search space, where each candidate is printed in a term representation. The latter is useful to check whether a grammar matches its intention. The GAP-C analyzes the grammar and automatically generates the code for the automatic algebra declaration. In Haskell-ADP these algebras need to be manually implemented for every grammar.

Next the score algebra which computes the maximal number of base pairings is implemented in GAP-L syntax:

```
algebra score  implements Nuss(alphabet = char,
                                answer = int)
{
```

```
  int nil(void) { return 0; }
  int right(int a, char c) { return a; }
  int pair(char c, int m, char d) { return m + 1; }
  int split(int l, int r) { return l + r; }
  choice [int] h([int] l) { return list(maximum(l)); }
}
```

The alphabet and sort placeholders in the signature are mapped to concrete types in the header of the algebra declaration. The code of the algebra function is C/Java like. The functions `list` and `maximum` are pre-defined and specialized versions for different types and products are part of the GAP-L runtime library.

The implementation of the pretty algebra looks like this:

```
algebra pretty implements Nuss(alphabet = char,
                               answer = string)
{
  string nil(void)
  {
    string r;
    return r;
  }

  string right(string a, char c)
  {
    string r;
    append(r, a);
    append(r, '.');
    return r;
  }

  string pair(char c, string m, char d)
  {
    string r;
    append(r, '(');
    append(r, m);
    append(r, ')');
    return r;
  }

  string split(string l, string r)
  {
    string r;
    append(r, l);
    append(r, r);
    return r;
```

```
  }

  choice [string] h([string] l)
  {
    return l;
  }

}
```

In this case, the evaluation function is the identity. The result of the pretty print algebra evaluation is a list of Vienna-Strings [25]. The strings are constructed via the built-in function `append`, which is overloaded for different data-types and arguments. This example shows the imperative nature of the GAP-L algebra code.

The example products from Section 2.1.1.1 are defined in GAP-L via instances:

```
instance scorepp = nussinov ( score * pretty ) ;
instance scoreco = nussinov ( score * count ) ;
```

The GAP-L programmer does not need to manually write the definition of the lexicographic product for the above signature, as in Haskell-ADP, it is automatically derived by GAP-C.

4.4 Lexical Structure

The character set of Bellman's GAPprograms is ASCII. The lexing is case sensitive.

4.4.1 Keywords

The following tokens are keywords:
algebra alphabet auto axiom choice classify else extends extern for grammar if implements import input instance kscoring overlay parameters pretty return scoring signature suchthat synoptic tabulated type uses void with

4.4.2 Comments

Comments are specified as in C++/Java. Everything between `/*` and `*/` and from `//` to the end of line is ignored.

4.4.3 Operators

The following operators are supported:

```
+ - = * / % .  < > == != <= >= && || !  ++ += -- -=
```

4.4.4 Constants

Character constants are enclosed by single ticks (') and string constants are enclosed by double ticks (").

Numbers are encoded as integers or in the standard IEEE 754 floating point notation.

4.4.5 Whitespace

Blanks, tabs and newlines outside of constants are considered as whitespace and are ignored.

4.4.6 Identifiers

Identifiers are described by this regular expression: `[A-Z_a-z][A-Z_a-z0-9]*`

4.4.7 Layout

There is no special treatment of the source code layout, i.e. there is no off-side rule (like in Haskell or Python). Statements are separated by semicolons and code blocks are enclosed by curly braces.

4.5 Program Structure

The syntax of GAP-L is specified as a context-free grammar. Meta-symbols of the grammar are ':' (separating left and right hand side of productions), '|' (separating alternative right hand sides), and ';' (separating productions). An empty alternative on the right hand side is always listed last. Nonterminal symbols are written in plain `typewriter` style, terminal symbols in *`italic-face`* style. The non-terminal symbol `ident` designates arbitrary identifiers. For clarity, it will be written in the form `algebra_ident, var_ident` etc. to indicate the semantic role of the identifier. However, these identifiers for different roles are not distinguished syntactically.

A Bellman's GAP program is structured into several sections. Some sections are optional, but the order of the sections is fixed. The non-terminal `program` is the start symbol of the syntax description. Some optional parts are mandatory to generate target code, but are not needed for semantic analyses by GAP-C.

```
program:
        imports_opt input_opt types_opt
        signature algebras_opt grammar instances_opt
        ;
```

4.5.1 Imports

```
imports_opt:
        imports |
        ;

imports:
        import |
        imports import
        ;

import:
        import module_ident
        ;
```

The `module_ident` can be a module from GAP-M; otherwise module names are treated as names of a user defined module. The module `rna` is an example for a module from GAP-M (see 6.4). It defines several functions for computing free energy contributions of bases in different RNA secondary structure elements.

4.5.2 Input

```
input_opt:
        input input_specifier |
        input '<' inputs '>' |
        ;

inputs:
        input |
        inputs ',' input
        ;

input:
        input_specifier
        ;
```

The input declaration specifies a special input string conversion. By default the input is read as is (`raw`). The input specifier `rna` signals an input conversion from ASCII encoded nucleotide strings to strings, which are encoded in the interval [0..4]:

value	nucleotide
0	an unspecified base
1	A
2	C
3	G
4	U

The alphabet of the input string is specified in the algebra definition.

The input declaration also specifies the number of input tracks in a multi track Bellman's GAP program. For example, `input <raw, raw>` means two-track input and both tracks are read as is. In GAP-L the default is single-track processing.

Multi-track dynamic programming algorithms work on more than one input sequence. For example, the Needleman-Wunsch pairwise sequence alignment algorithm [35] or the Sankoff fold and align algorithm [42] work on two input sequences (two-track). RNA folding, like the Zuker minimum free energy (MFE) algorithm [59], work on one input sequence (single-track).

4.5.3 Types

```
types_opt:
        types |
        ;

types:
        type |
        types type
        ;

type:
        type ident '=' datatype |
        type ident '=' extern
        ;
```

Type declarations at the global level are either type synonyms or declarations of datatypes in imported external modules.

```
datatype:
        type_specifier |
        alphabet |
        void |
        '[' type_specifier ']' |
        '(' named_datatypes ')'
        ;
```

A datatype is either an elementary datatype, the alphabet type, `void`, a list or a (named) tuple.

The `alphabet` type can only be used in the signature declaration. It is a placeholder for an actual datatype of an alphabet. An algebra implementing the signature declares which datatype is the alphabet datatype.

Elementary datatypes are:

`int, integer, float, string, char, bool, rational, bigint, subsequence, shape, void.`

`float` is in double precision, `rational` and `bigint` are of unlimited precision and `subsequence` saves only the begin-/end-index of a substring. `int` is at least 32 bit long and `integer` is at least 64 bit long.

```
named_datatypes:
        named_datatype |
        named_datatypes ',' named_datatype
        ;

named_datatype:
        datatype name_ident
        ;
```

Note that this syntax forces the programmer to name the components used in tuples.

4.5.4 Signature

```
signature:
        signature ident '(' sig_args ')' '{' sig_decls '}'
        ;

sig_args:
        alphabet ',' args signtparas
        ;
```

The parameters of the signature declaration are the alphabet keyword and one or more sorts. A sort is a name for a data type which will be substituted in an algebra that implements the signature. (In Haskell terminology, a Bellman's GAP sort is a type parameter.)

```
args:
        arg |
        args ',' arg
        ;
```

```
arg:
        ident
        ;

signtparas:
        ';' datatypes |
        ;

sig_decls:
        decl ';' |
        sig_decls decl ';'
        ;

decl:
        qual_datatype ident '(' multi_datatypes signtparas ')'
        ;
```

The signature contains one or more signature function declarations. The `qual_datatype` indicates the result type of the function.

```
qual_datatype:
        datatype |
        choice datatype
        ;
```

The qualifier `choice` marks a signature function name as objective function. The declaration of several objective functions is allowed. For the objective function, argument and return types must be list types.

```
datatypes:
        datatype |
        datatypes ',' datatype
        ;

multi_datatype:
        '<' datatypes '>' |
        datatype
        ;
```

A `multi_datatype` is a tuple of datatypes. In an algebra function the i-th component of this type comes from the i-th input track. In a single track context, `datatype` is equal to `< datatype >`.

```
multi_datatypes:
        multi_datatype |
        multi_datatypes ',' multi_datatype
        ;
```

4.5.5 Algebras

An algebra implements a signature. The algebra declaration specifies which data type is used for the alphabet and which data type is used for each sort. The body of the algebra contains a compatible function definition for each signature function declaration, where alphabet and sort types are substituted according to the head of the algebra declaration.

```
algebras_opt:
        algebras |
        ;

algebras:
        algebra |
        algebras algebra
        ;

algebra:
        algebra_head '{' fn_defs '}' |
        algebra ident automatic automatic_specifier ';'
        ;

automatic_specifier:
        enum |
        count
        ;
```

The `automatic` keyword specifies the auto generation of the specified algebra. The Bellman's GAP compiler supports the auto generation of an enumeration (`enum`) algebra and a counting (`count`) algebra under an arbitrary signature. An enumeration algebra prints each candidate term as a human readable string and keeps all candidate strings in the objective function, i.e. running the enumeration algebra alone prints the whole candidate search space. A counting algebra counts how many candidates there are in the search space.

```
algebra_head:
        mode algebra ident parameters implements
              signature_ident '(' eqs ')' |
        algebra ident parameters implements
```

```
            signature_ident '(' eqs ')' |
        algebra ident parameters extends algebra_ident
        ;
```

An algebra is declared as an implementation of a signature or as an extension of a previously defined algebra. If a signature is directly implemented, the mapping between signature parameters (alphabet and sorts) and concrete datatypes is specified. In the case of an extension, every already declared algebra function can be overwritten.

The mode of an algebra is optional and either: `synoptic stringrep classify scoring kscoring`

`kscoring` is the default mode for every objective function of the algebra and can be overwritten by a declaration of an objective function.

In case of no mode specification, the compiler tries to derive the mode automatically. If an objective function uses the generic list minimization function, the objective function mode is autodetected as `scoring`.

```
parameters:
        parameter_block |
        ;

parameter_block:
        '(' var_decl_init_p |
        '(' var_decl_inits var_decl_init_p
        ;

var_decl_inits:
        var_decl_init_k |
        var_decl_inits var_decl_init_k
        ;

var_decl_init_p:
        datatype ident '=' expr ')'
        ;

var_decl_init_k:
        datatype ident '=' expr ','
        ;
```

Parameters of the algebra are optional. If present, they are supplied or overwritten at runtime of the resulting Bellman's GAP program, e.g. via command line switches and are intended to be supplied or overwritten by the user of a generated Bellman's GAP program.

```
eqs:
        eq |
        eqs ',' eq
        ;

eq:
        sig_var '=' datatype
        ;

sig_var:
        sort_ident |
        alphabet
        ;
```

4.5.5.1 Algebra Functions

```
fn_defs:
        fn_def |
        fn_defs fn_def
        ;

fn_def:
        mode_opt qual_datatype ident '(' para_decls fnntparas ')'
              '{' statements '}'
        ;

fnntparas:
        ';' para_decls |
        ;

mode_opt:
        mode |
        ;

para_decls: |
        para_decl |
        para_decls ',' para_decl
        ;

para_decl:
        datatype ident |
        '<' para_decls '>' |
```

```
        void
        ;
```

An algebra contains normal functions and one or more objective functions. A function is marked as objective function by using the keyword `choice` (see definition of `qual_datatype`). In each declaration of the objective function it is possible to overwrite the default algebra mode. It is possible to declare an algebra with two objective functions, where the first one is of `scoring` mode and the second one is of `kscoring` mode.

A `para_decl` is either a single-track, a multi-track or a `VOID` parameter declaration. A multi-track parameter declaration is the implementation of a multi-track tuple type of the corresponding signature function parameter. If a non-terminal parser evaluates a branching element, it feeds each branch result into the corresponding declared parameter of a multi-track parameter declaration.

An example of the multi-track parameter declaration syntax is the following algebra function `match`:

```
int match( <char a, char b>, int rest)
{
  if (a == b)
    return 1 + rest;
  else
    return rest;
}
```

The corresponding signature function is:

```
answer match( <char, char>, answer);
```

The signature function symbol match may be used in a grammar rule, e.g.:

```
ali = match( < CHAR, CHAR>, ali)
```

4.5.6 Statements

```
statements:
        statement |
        statements statement
        ;

statement:
        continue |
        return |
        if |
        for |
        assign |
        var_decl |
```

```
        fn_call |
        '{' statements '}'
        ;

continue:
        continue ';'
        ;

fn_call:
        ident '(' exprs ')' ';'
        ;

return:
        return expr ';'
        ;

if:
        if '(' expr ')' statement %prec LOWER_THAN_ELSE |
        if '(' expr ')' statement else statement
        ;
```

The `%prec LOWER_THAN_ELSE` grammar description annotation specifies that the else part of an if statement belongs to the last started if statement (like in C/Java) while parsing nested conditionals.

```
for:
        for '(' var_decl_init expr ';' inc_stmt ')'
              statement
        ;

assign:
        var_access '=' expr ';'
        ;
```

4.5.7 Variable Access

```
var_access:
        ident |
        var_access '.' name_ident |
        var_access '[' expr ']'
        ;
```

A variable access is either an access to a simple variable, an access to a component of a named tuple or an access to an array.

4.5.8 Grammar

4.5.8.1 Grammar rules

```
grammar:
        grammar ident uses signature_ident
            '(' axiom '=' nt_ident ')' '{' grammar_body '}'
        ;

grammar_body:
        tabulated productions |
        productions
        ;
```

The definition of a grammar specifies the name of the grammar, the used signature and the name of the start symbol. The grammar is a regular tree grammar. The right hand side contains function symbols from the signature as tree nodes. The GAP-C checks whether the grammar is valid under the specified signature.

```
tabulated:
        tabulated '{' args '}'
        ;
```

With the optional `tabulated` declaration it is possible to request the tabulation of a list of non-terminals. In case of an increased optimization level or a non-present `tabulated` declaration the compiler automatically computes a good table configuration (see Section 5.3.7).

```
productions:
        production |
        productions production
        ;

production:
        ident ntargs '=' rhs ';'
        ;

ntargs:
        '(' para_decls ')' |
        ;
```

A non-terminal symbol can be defined with arguments (i.e. a parameterized non-terminal). The arguments, or expressions including the arguments, can be used on the right hand side as extra arguments of a function symbol, a filter function or another parametrized non-terminal call. A parametrized non-terminal cannot be tabulated, because for every combination of parameter values a separate table would be needed.

An example for the use of parametrized non-terminals is the design of RNA pattern matching algorithms in ADP [31], where a non-terminal models e.g. a stack of base pairings and the argument of the non-terminal is the stack length. The argument is then decremented, if greater than zero, and applied to a recursive non-terminal call. Another example is pknotsRG [40], where canonicalization information is supplied via non-terminal parameters (Section 5.4.5).

```
rhs:
        alts |
        alts '#' choice_fn_ident
        ;
```

The right hand side of a production is a set of alternatives with an optional application of an objective function which was declared in the signature.

```
ntparas:
        ';' exprs |
        ;

filters:
        filters ',' filter_fn |
        filter_fn
        ;

tracks:
        track |
        tracks ',' track
        ;

track:
        alt
        ;

alts:
        alt |
        alts '|' alt
        ;
```

```
alt:
        '{' alts '}' |
        sig_fn_or_term_ident '(' rhs_args ntparas ')' |
        symbol_ident |
        alt filter_kw filter_fn |
```

An alternative is a block of enclosed alternatives, a function symbol from the signature plus its arguments, a non-terminal/terminal parser call or a conditional alternative.

4.5.8.2 Multi-Track Rules

```
        '<' tracks '>' |
        alt filter_kw '<' filters '>' |
```

For multi-track dynamic programming an alternative can also be a branching from a multi-track context into several single-track contexts or a conditional alternative guarded by different single-track filters for each track.

A multi-track context of n tracks may contain an n-fold branching `< a_1, ..., a_n >`. Each a_i is then in a single-track context for each track i, where a_i is a terminal- or non-terminal parser call.

For example, `match ( < CHAR, CHAR >, ali )` is a grammar rule that calls two character reading terminal parsers, which read a character from the first or the second input track, respectively.

To apply a filter on different tracks in a multi-track context, a list of filters has to be included in <> parentheses.

In multi-track mode the grammar may contain combinations of single-track and multi-track productions. The following example contains two-track and single-track productions:

```
foo = del ( < CHAR, EMPTY > , foo ) |
      ins ( < EMPTY, CHAR > , foo ) |
      x   ( < fold, REGION >, foo ) # h ;

fold = hl ( BASE, REGION, BASE ) # h' ;
```

4.5.8.3 Non-terminal parameters

```
        alt '.' '(' exprs ')' '.' |
```

This alternative specifies the syntax for calling non-terminals that have parameters. In case `alt` is not a link to another non-terminal, an error should be signaled.

4.5.8.4 Index Hacking

```
        symbol_ident '[' exprs ']' |
        alt '.' '{' alt '}' '.' |
        '.' '[' statements ']' '.' '{' alt '}'
        ;
```

These index hacking related alternatives specify a non-terminal call with explicit indices, an overlay of two alternatives and verbatim index manipulation code before an alternative. The tree grammar search-space specification mechanism from the ADP framework eliminates the need of using explicit indices for most dynamic programming algorithms over sequences. However, some algorithms, like for example pknotsRG [40], need to perform their own index computations at selected non-terminal locations for efficiency reasons. In the example of pknotsRG, canonicalization rules are applied to reduce the number of moving index boundaries. In GAP-L, these rules are implemented as verbatim index manipulation code in the grammar. The overlaying of alternatives is used in the semantic analyses. The left alternative is a fake rule that approximates the resulting index boundaries, such that the runtime analysis computes more realistic results. The right alternative is then used for code generation.

See Section 5.4.5 for more details on index hacking in the use case of pknotsRG.

4.5.8.5 Grammar Filters

```
filter_kw:
        with |
        suchthat |
        with_overlay |
        suchthat_overlay
        ;
```

With `P filter_kw f` in case of the `with` keyword, the filter function f is called before P is parsed, with the sub-word that should be parsed by P, as an (additional) argument. With the `suchthat` keyword the filter function is called after P is evaluated for each parse of P. `with_overlay` and `suchthat_overlay` are variations of `with` and `suchthat` and are only defined if P uses a signature function g. In the case of `with_overlay` the filter function is called with a list of subwords which correspond to the unparsed arguments of g, before P is parsed. With `suchthat_overlay` the filter function is called after the arguments of g are parsed and before the evaluation of g for each combination of argument values.

Filtering through `with` and `with_overlay` clauses is called syntactic filtering, since the filter function depends only on the input word. Filtering with `suchthat`

and `suchthat_overlay` is called semantic filtering, since the filter does not depend on the input word, but on the used algebra.

```
filter_fn:
        ident |
        ident '(' exprs ')'
        ;
```

The filter function can be part of the signature and algebra definition or can be included in a module. The filter function must return a boolean value. In addition to the default arguments, it is possible to supply user defined arguments.

If the return value is false, the left hand side of the filter keyword is not used during parsing. In the case of syntactic filtering this means that the left hand side is neither parsed nor evaluated. With semantic filtering, the left hand side is parsed and evaluated, but the result is discarded.

The filters are used to reduce the search space which is described by the grammar.

```
rhs_args:
        rhs_arg |
        rhs_args ',' rhs_arg
        ;

rhs_arg:
        alt |
        const
        ;

const:
        number |
        '\'' character_constant '\'' |
        '"' string_constant '"'
        ;
```

4.5.8.6 Terminal Symbols

The Bellman's GAP language supports several terminal parsers or symbols. For a terminal parser it is possible to have one or more arguments.

The yield size of a terminal parser is the number of characters it parses from the input word. The `STRING` terminal parser parses some non-empty string, i.e. its minimum yieldsize is 1 and its maximum yieldsize is n, where n is the length of the input word.

The terminal symbols without arguments (including their return type) are listed as follows:

		yield size	
Return type	Parser	min	max
[void]	EMPTY	0	0
[subsequence]	LOC	0	0
[char]	CHAR	1	1
[subsequence]	BASE	1	1
[string]	STRING0	0	n
[string]	STRING	1	n
[subsequence]	REGION0	0	n
[subsequence]	REGION	1	n
[float]	FLOAT	1	n
[int]	INT	1	n
[int]	SEQ	1	n

If a terminal parser cannot parse successfully, an empty list is returned. The parser `LOC` is used to access the position in the input string, where the empty word was parsed. `INT` reads an integer number and returns its value. `SEQ` parses a sub-word from the input string and returns its length.

The list of terminal symbols with arguments is:

```
[alphabet] CHAR(alphabet)
[int] INT(int)
[int] CONST_INT(int)
[subsequence] STRING(string)
[float] CONST_FLOAT(float)
```

The `CONST_*` terminal parsers have a maximum yieldsize of 0, i.e. they don't consume any sub-word of the input. Those terminal parsers can be used in a grammar context to supply a constant argument to an algebra function.

4.5.9 Instances

An instance declaration specifies under which algebra (or product) a grammar is evaluated.

```
instances_opt:
        instances |
        ;

instances:
        instances_
        ;

instances_:
```

```
        instance |
        instances_ instance
        ;

instance:
        instance i_lhs '=' ident '(' product ')' ';'
        ;
```

An instance is named. On the right hand side of the equal sign, the grammar and the product is specified. See Section 2.1.1 for the semantics of the products.

```
product:
        product '*' product |
```

The lexicographic product.

```
        product '/' product |
```

The interleaved product.

```
        product '%' product |
```

The cartesian product.

```
        product '.' product |
```

The take-one product. The difference to the lexicographic product is that only one co-optimal result is chosen in the case of co-optimal results.

```
        product '|' product |
```

The overlay product. With `A | B`, A is used in the forward computation and B is used during backtracing. An use case for this is stochastic backtracing (Section 5.4.3.1), i.e. the sampling of shape strings under a partition function:

```
( (p_func | p_func_id ) * shape5 ) suchthat sample_filter )
```

The objective function of the `p_func` algebra is summation and the objective function of the `p_func_id` algebra is identity. During the forward computation only `p_func` is evaluated. In the backtracing phase the intermediate `p_func` values are evaluated by the `p_func_id` algebra and value lists are filtered by the `sample_filter`. The `sample_filter` interprets the value lists as discrete probability distributions and randomly takes one element from the list under this distribution. During the backtracing the shape representation is randomly built according to the computed probability distribution, i.e. the repeated stochastic backtracing samples shape strings according to their shape probability (see Section 5.4.3.1).

```
'(' product ')' |
algebra_ident |
```

Singleton product.

```
product suchthat filter_fn
;
```

Before evaluating the answers list with the product's objective function, the `filter_fn` is applied to each intermediate (candidate) answer list. The result of the filter_fn is the input for the products objective function.

A usecase for this feature is the probability mode in RNAshapes [51], where in the computation of `shape * pf` every (sub-)candidate is removed during the computation, if the left hand side is < 0.000001. This filter significantly reduces the exponential number of classes, such that the computation of this product for longer sequences is feasible (Section 4.6.2).

4.6 Selected Language Features

The previous section has presented the syntactic constructs of GAP-L, focusing on the structure of GAP-L programs. In this section several language concepts are described in more detail to show how different language features cooperate.

4.6.1 Algebra extension

It is possible to define an algebra as an extension of an existing algebra. This concept is similar to class inheritance in object oriented languages like Java. See Section 4.5.5 for the syntax specification. In the extended algebra, function definitions overwrite the ones of the base algebra. The extended algebra has the same alphabet type and type assignments to the sorts like the base algebra. The extension mechanism is e.g. used in the ElMamun example where the buyer algebra and the seller algebra only differ in its evaluation function (Figure 4.1).

Further examples are the different classification algebras of RNA-folding programs, where the implementation of each shape level abstraction differs only in a few function definitions.

4.6.2 Syntactic filtering

Filtering in the grammar means pruning the search space. Consider the following code example.

```
closed = { stack | hairpin | leftB | rightB | iloop |
           multiloop }
                    with stackpairing # h ;
```

```
algebra buyer implements Bill(alphabet = char,
                               answer = int) {

  ...

  choice [int] h([int] i)
  {
    return list(minimum(i));
  }
}

algebra seller extends buyer {
  choice [int] h([int] l)
  {
    return list(maximum(l));
  }
}
```

Figure 4.1: Example of an algebra extending another. The evaluation function is overwritten.

This means that each possible sub-word from the input string fed into the non-terminal parser `closed` is checked by the filter `stackpairing` before it is subject to be parsed by the non-terminal parsers `stack, hairpin ...`. Only if the filter returns true, the sub-word is parsed. The filter checks, if the first and second characters (or bases) are complementary to the last and second-last characters.

Pruning the search space with filters may reduce computation time in practice, but another advantage is the simplification of algebras and a clear separation between search space description and evaluation. For example, if in a maximization algebra the `stackpairing` would be checked and scored with minus infinity, the grammar filter would not be needed. Then a counting or probabilistic algebra would return wrong results. If extending the algebra for minimization, it is not enough to just exchange the objective function. Further practical problems with this approach would be range-overrun problems when using integers.

For more filter variants and the formal syntax see Section 4.5.8.1. Syntactic filtering is a source of sparseness in dynamic programming that is exploited in top-down evaluation of the search space (Section 5.4.1).

4.6.3 Semantic instance filtering

In the instance declaration, there is optional support for semantic filtering (see Section 4.5.9 for formal syntax). Semantic filtering means that the answer list of a non-terminal parser is filtered with the specified filter function after the objective

function was applied. This language feature makes it possible to cleanly separate filtering concerns from generic algebras.

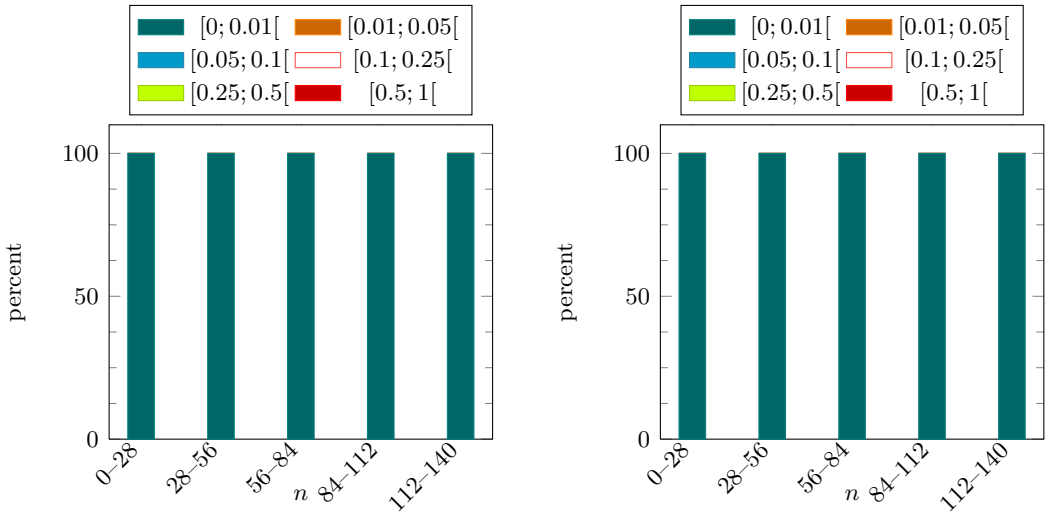
A use case is found in classified DP [48, 49], where the classification algebra computes an exponential number of classes with growing input length, i.e. during the computation a filter heuristically removes those classes that only contribute very small values to the solution. An example is this instance declaration:

```
instance shape5pfx = fold ((shape5 * p_func)
                                  suchthat p_func_filter);
```

The product computes the probability of each shape in the search space (each candidate has a shape, each shape embraces multiple candidates). The filter function `p_func_filter` removes the shape classes with a very low contribution (for example < 0.000001 percent) for all non-terminal parser results.

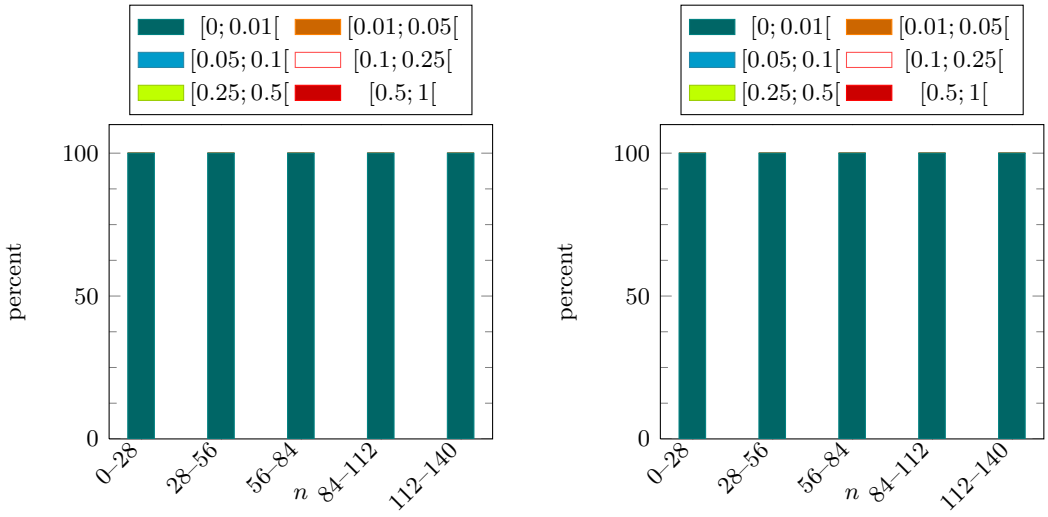
Figure 4.2 shows the distribution of shape probability deviations δ (Equation 5.29, page 122) using two different grammars and various cut-off filter values for 2000 randomly generated sequences. RNAshapes and GAPC nonamb use the same grammar that takes energy contributions of dangling bases unambiguously into account. The δ between two sets of shape probabilities is computed using the results of a computation without filtering and of one with a cut-off filter applied during the computation. The results show that using a cut-off filter of 0.0001 or less only introduces small errors in comparison to the exact computation. For most comparisons the δ are less than 0.01. Only for RNAshapes and a cut-off value of 0.0001 it shows that 5 percent of the sequences larger than 110 bases yield δ between 0.01 and 0.05. This means for the tested sequences that a filtered shape probability deviates 5 percent points from the exactly computed shape-probability in the worst case. Figure 4.3 shows the number of shape classes as a function of the sequence length for different cut-off filter values for the same set of randomly generated sequences. Using filtering the number of shapes is greatly reduced. The numbers of shapes resulting from an unfiltered computation are up to 10^3 or 10^4 times of the numbers resulting from filtering, depending on the used cut-off values and grammars.

An alternative to this use case of semantic filtering is to sample shapes from the grammar (which is also called stochastic backtracing, see Section 5.4.3.1 for details). Figure 5.41 shows the distribution of shape probability deviations due to sampling for the set of sequences used in Figure 4.2. When computing only the partition function (Equation 5.24, page 122) and then sampling shape strings several times (e.g. 1000 iterations) under a shape algebra, the number of samples that return the same shape is divided by the number of samples which approximates the probability of that shape class. Comparing semantic filtering with stochastic backtracing shows that semantic filtering yields smaller shape probability deviations, but stochastic backtracing is more efficient, i.e. filtering is in $O(k^2n^3)$ and stochastic backtracing is in $O(n^3)$, where k is the number of classes.



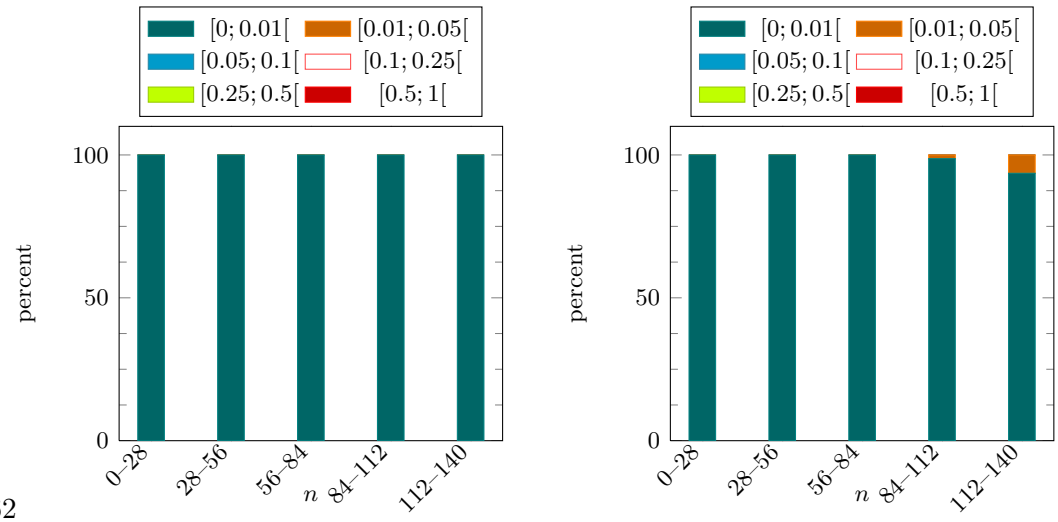
(a) GAPC adpf (cut-off: 0.000001)

(b) GAPC adpf (cut-off: 0.0001)



(c) GAPC nonamb (cut-off: 0.000001)

(d) GAPC nonamb (cut-off: 0.0001)



(e) RNAshapes (cut-off: 0.000001)

(f) RNAshapes (cut-off: 0.0001)

Figure 4.2: Distribution of shape probability deviations δ for two different grammars and various cut-filter values. The δ is computed between the results of an unfiltered computation and a filtered one. Each program was run on the same set of 2000 randomly generated sequences. RNAshapes and GAPC nonamb use the same grammar (see text).

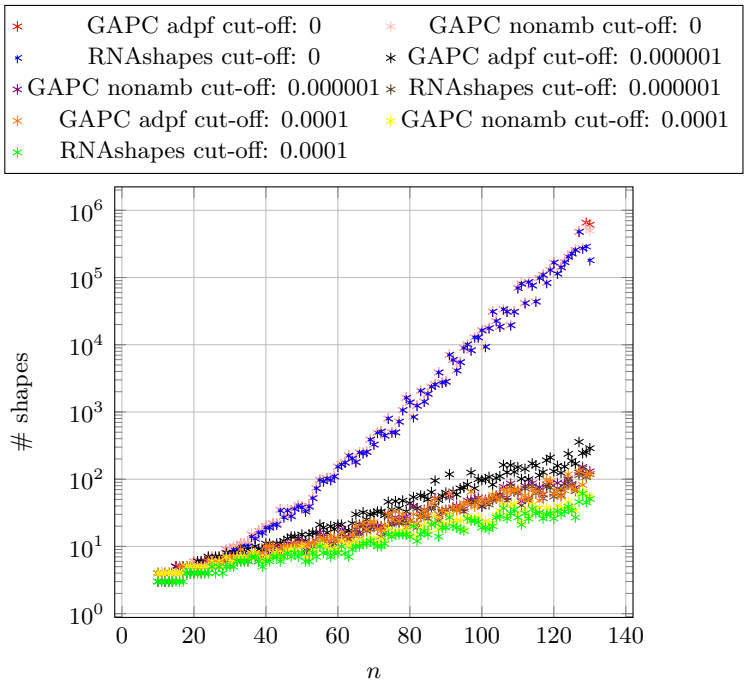


Figure 4.3: The maximal numbers of shapes as a function of the sequence length n for various grammars and cut-off filter values and RNAshapes. The input are 2000 randomly generated sequences. A cut-off value of 0 means that no filtering of shape classes was done during the computation. A cut-off value of x means that all shape classes are filtered during the computation that have a shape-probability of $< x$ percent.

4.6.4 Multi-Track programs

The `input` declaration (Section 4.5.2) specifies on how many input tracks a GAP-L program computes, e.g. for two input tracks:

```
input < raw, raw >
```

In most single track GAP-L programs there is no `input` declaration, since the default is the `input < raw >` declaration (raw means no pre-processing or conversion of the input string). The alphabet of the input tracks is not specified in the input declaration, but in the algebra declaration.

If the input declaration specifies l input tracks, the axiom non-terminal is in a l-track context. An l-track context non-terminal can directly call other l-track non-terminals or use the $< x_1, \ldots, x_l >$ construct (see Section 4.5.8.1) to call for each track a single track context non-terminal or terminal x_i, where $0 \leq i < l$. A single track context non-terminal can be called for different tracks. In this case the non-terminal parser parses different tracks in dependence of the callers track position. Using this track context change feature, single track sub-grammars are usable from a multi-track grammar. A use case for this feature is e.g. the combination of alignment and duplication history computation in one algorithm [1].

The `<>` parentheses are used in the definition of signature and algebra functions according to their use in the grammar. See Section 4.5.5.1 for the formal syntax .

In the general case, a two-track DP-program needs $O(n^4)$ space, since there are two indices for each track. However, the GAP-C analyzes the grammar and eliminates the indices that are constant (Section 5.3.12). For example, in the Pairwise Sequence algorithm, only one index changes for each track, thus the space consumption is in $O(n^2)$.

4.6.5 Alphabets

In GAP-L programs, the alphabet specifies the basic unit of parsing. The terminal parsers like `CHAR`, `REGION` etc. are alphabet polymorphic. That means that `CHAR` returns a `char`, if the alphabet is `char` or a `float` if the alphabet is `float`.

The alphabet can be changed for each algebra, but note that the algebras used in a product have to use the same alphabet. The formal syntax is specified in Section 4.5.5. An example:

```
algebra pretty implements Align(alphabet = single,
  answer = spair) {
...
}
```

The algebra `pretty` works on input strings of single precision floating point numbers.

In the default command line frontend generated by the GAP-C, the user input is converted to the right alphabet. In alphabets other than `char`, whitespace is used to delimit the characters of the input.

5 Bellman's GAP Compiler

The Bellman's GAP Compiler (GAP-C) is the novel ADP compiler which translates GAP-L programs into efficient C++ code. Section 5.1 specifies the overall architecture of GAP-C. An example compile session using the running Nussinov example is presented in Section 5.2. The compiler implements several semantic analyses for optimization purposes, error reporting, type checking and automatic table design, which are specified in Section 5.3. Algorithms and techniques regarding the code-generation phase of GAP-C are reported in Section 5.4.

5.1 Compiler Architecture

The Bellman's GAP compiler is written from scratch in C++. Object-orientation is extensively used for the different parts of the compiler. The architecture of the compiler consists of three modules: The frontend, the middle-end and the backend. See Figure 5.1 for an overview.

The frontend consists of a lexer and a parser. The lexer divides the input GAP-L program into a stream of tokens which are consumed by the parser. The parser creates an abstract syntax tree (AST). The elements of the AST are C++ objects. For example, there is an expression base class and the substraction operation is a sub-class of the expression base class. Figure 5.2 shows a class diagram of the main classes an AST is made of. Both, the lexer and the parser are autogenerated from abstract specifications. The lexer is specified as Flex [37] description and the syntax is specified as Bison [18] grammar. The lexer and parser implement a sophisticated location tracking scheme to be able to report the exact line and column in error messages. A generic error message printing component takes the location object into account in printing the specified line of the input program and highlighting the specified columns, such that a Bellman's GAP programmer gets a more informative error message. Figure 5.3 shows an example message. Syntax parse errors, i.e. violations of the GAP-L grammar, and simple consistency checks are done interleaved with the AST construction. More sophisticated type-checking and reporting of semantic errors/warnings are done in the middle-end, since the whole AST and the output of semantic analyses are needed.

The middle-end takes the AST as input and applies several semantic analyses to it. Examples for semantic analyses are type-checking (Section 5.3.8), yield size analysis (Section 5.3.3) and table-design (Section 5.3.7). The results of the semantic analyses and the AST are the input for the code-generation. The result of the code generation is a data structure that represents the abstract target code. The abstract

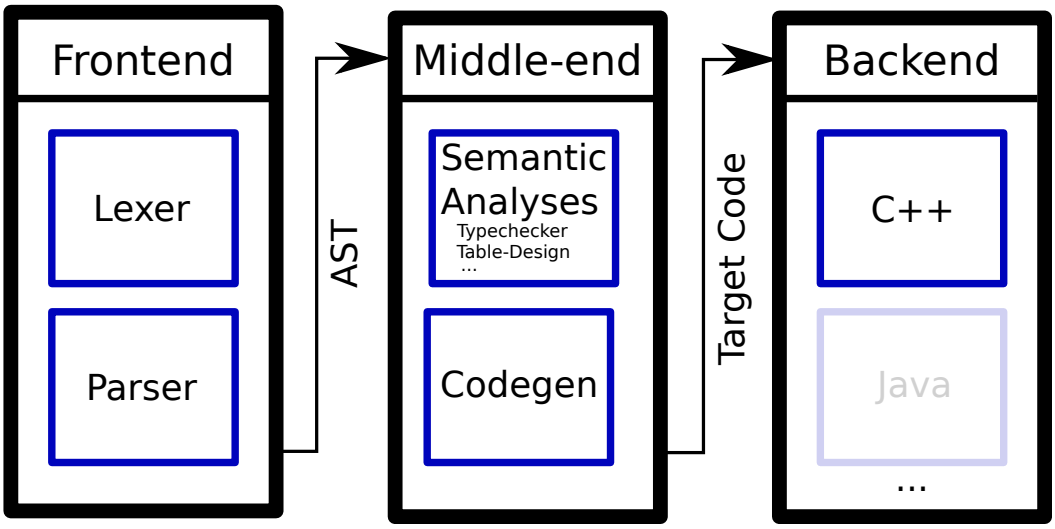


Figure 5.1: Architecture of the Bellman's GAP compiler. AST is the abstract syntax tree. The Java backend is not yet implemented, it symbolizes the extensibility of the backend architecture for more output languages.

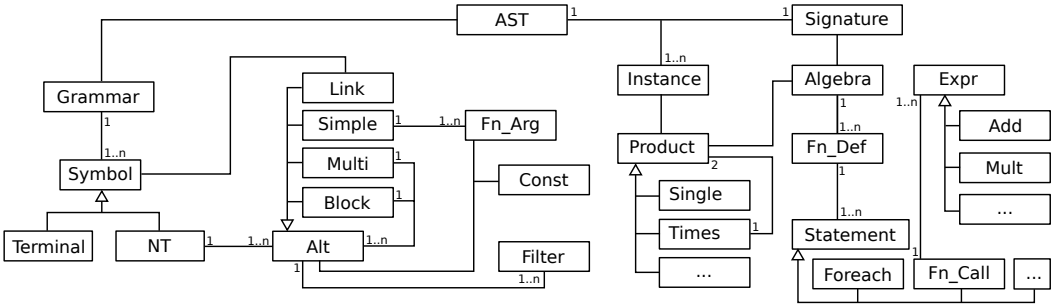


Figure 5.2: Class diagram of the main abstract syntax tree (AST) classes.

```
Error:
            mult(times, formula) # h ;
            ^--^
elm.gap:186.13-16: Function mult has 2 arguments, but
Error:
  answer mult(answer, alphabet, answer);
         ^--^
elm.gap:9.10-13: it is defined with 3 arguments here.
```

Figure 5.3: Example of an error message of GAP-C. A context of the error location is printed and the columns of the location are marked.

Table 5.1: Examples of ADTs used in code generated by GAP-C. They are provided by the runtime library GAP-M. For each ADT a few examples of access function are listed.

ADT	functions
List	push_back
	append
	empty
	is_empty
	...
String	append
	empty
	...
Hashtable	push_back
	append
	update_filter
	finalize
	...

target code is an internal language. This language is imperative. It contains high-level statements, e.g. for-loops and functions, and it abstracts from implementation details of central data structures. They are better defined in a language dependent backend. For example, the way tables are stored and accessed efficiently, and the efficient representation of intermediate lists and backtrace structures depend highly on the output language. Thus the target language contains a set of built-in abstract datatypes (ADTs) and operations on them. This includes ADTs for lists, backtracing structures, arbitrary precision rationals/integers, classification data structures and tabulation. Table 5.1 shows an overview. As a consequence, the target language does not need to implement pointer arithmetic. This simplifies the implementation of backends for output languages that do not have pointers (as e.g. Java).

The backend takes the target code as input and generates output language code. The backend architecture is constructed for extensibility, i.e. to enable the addition of new output languages. There is an abstract base class that defines the interface, which each output language backend has to implement. Figure 5.4 sketches the interface of the backend classes. Currently, a language backend for C++ output is included. The C++ backend maps the ADTs operations to generated C++ classes and to a Bellman's GAP C++ runtime library. On the one hand, the backtracing target code is translated into generated classes, since the data structure sensitively depends on the signature and algebra of the GAP-L input program. On the other hand, the memory management and an efficient list data type is implemented as C++ template classes in the runtime library because they are independent of the

```
      void print(const Statement::For &stmt);
      void print(const Statement::If &stmt);
      void print(const Statement::Backtrace_Decl &stmt);
      void print(const Statement::Hash_Decl &stmt);
      void print(const Statement::Table_Decl &stmt);
...
      void print(const Expr::Base &);
...
      void print(const Type::List &expr);
      void print(const Type::BigInt &expr);
...
```

Figure 5.4: Excerpts of the interface which a language backend of GAP-C has to implement.

translated program and C++ as output language is powerful enough to allow for an efficient runtime library implementation. The advantage of moving functionality, where it is appropriate, into a runtime library is the possibility to optimize or exchange parts of the runtime library without having to change the compiler, thus reducing the complexity of the C++ language backend.

Besides the translation of target code to output language code, the C++ backend generates a makefile and a generic command line interface. The makefile contains the build dependencies of the generated code and the generic interface code. By default, everything needed is built and linked into an executable. The command line interface is optional. The generated C++ code is enclosed in a C++ class which implements a common API. This makes it easy to integrate it into other C++ code. Examples are more sophisticated, specialized interface programs or a C++ program where a GAP-L program is just one step in a bigger pipeline.

5.2 Example

Before the internals of the compiler are described in the next sections, this section shows how the example from Section 4.3 is compiled with GAP-C.

A specific product of a GAP-L program is compiled with GAP-C with following commands:

```
$ gapc -t nussinov.gap -p 'score*count'
$ make -f out.mf
```

The option `-t` instructs the compiler to automatically determine the non-terminal parsers, whose solutions needs to be tabulated[1] (Section 5.3.7) and the `-p` option

[1]The option `-t` is enabled by default if the GAP-L program does not contain a table configuration.

specifies the product to compile. The compiler then produces optimized C++ code and a makefile. The program can be executed like this:

```
$ ./out ccaggg
( 2, 3 )
```

In some use cases of dynamic programming algorithms, not only the optimal score for an input string is of interest, but also the structure of the optimal candidate. In this example, this is accomplished with using the instance `scorepp`, which is declared in the source program:

```
$ gapc -t nussinov.gap -i scorepp
$ make -f out.mf
$ ./out ccaggg
( 2, ((.)). )
( 2, ((..)) )
```

The standard product operation computes all co-optimal solutions.

In addition to the default compiler call for the `scorepp` example, GAP-C can be instructed with the `--backtrace` option to generate backtrace code for the product (Section 5.4.3). This is more efficient, since in the forward computation the `score` algebra is computed and only in a backtracing step the algebra `pretty` is used. In addition to that, we can choose to generate a CYK-style bottom-up evaluator with the `--cyk` option. The default is the generation of top-down evaluators (Section 5.4.1). Top-down evaluators do extra bookkeeping but have advantages if the algebra is expensive to compute and the grammar has a lot of search space restrictions such that the top-down evaluation only does a very sparse computation of the tables. In this example this is not the case. Thus the bottom-up evaluator leads to more efficient code.

5.3 Semantic Analyses

In the following sections several semantic analyses are described which are part of GAP-C. A semantic analysis is an algorithm that computes properties of the AST or does optimizations. The results are needed for the generation of efficient code, for error reporting and to warn the user of problematic constructs.

In most cases the pseudo-code of a semantic analysis is given as a completive specification of the algorithm. The algorithms are of recursive nature and traverse the grammar data-structure, i.e. the AST. The relationship of the classes of the AST is shown in a class diagram (Figure 5.38). Figure 5.5 shows an object diagram of the AST of the Nussinov example from Section 4.3. It exemplifies the mapping of the different syntax elements in the source program to concrete objects.

Not included in the diagrams are the attributes of the objects that are used in the pseudo-code since they follow an easy scheme, e.g. `Alt::Block` and `Symbol::NT`

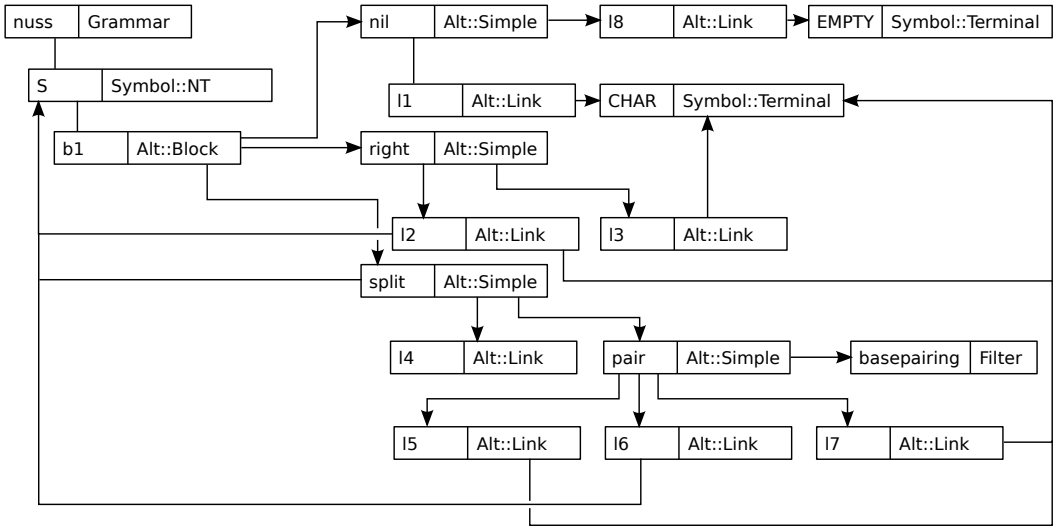


Figure 5.5: Object diagram of the grammar data-structure (AST) instance of the Nussinov example from Section 4.3. The directed edges indicate that the source contains the pointer of destination object. Multiple pointers are saved in an attribute of list type. The left part of the node contains a human readable object identifier and the right part the class name.

objects both contain a list of `Alt::Base` pointers, which is called `alts`, a `Alt::Link` object references a `Symbol::Base` object of which the pointer is saved in the attribute `nt` and the arguments of a function symbol, represented by an `Alt::Simple` object, are saved as a list of pointers to `Alt::Base` objects in the `args` attribute. Class hierarchies have their own namespace (the prefix up to `::`) and the base class of a hierarchy is called `Base`.

The present example is a single track algorithm such that the object diagram does not include `Alt::Multi` objects. An `Alt::Multi` object references for each track an `Alt::Base` object for which the pointers are saved in the attribute `tracks`.

The pseudo-code is similar to a scripting language and mixes traditional pseudo-code elements with Python and C++ ones. For space saving the code uses the off-side rule as in Python, i.e. a wider indentation starts a new block. Namespaces are used as in C++. As in scripting languages, variables need not to be declared: the first assignment automatically creates them as new object attributes. Similarly, the return type of methods is not declared, it is derived from the context. Object oriented features are used, i.e. a method call may target a method defined in the base class or calling a method on an object of base-class type dispatches to the version overwritten in a sub-class. Every method is by default declared as virtual like e.g. in Java.

```
grammar nonproductive uses Signature(axiom=start) {

  start = foo(CHAR('+'), start) ;

}
```

Figure 5.6: A non-productive GAP-L grammar.

```
grammar nonproductive2 uses Signature(axiom=S) {

  S = m   ( < CHAR, CHAR   >, S ) |
      ins ( < fold, EMPTY  >, S ) |
      nil ( < EMPTY, EMPTY >    ) # h ;

  fold = f ( CHAR, fold, CHAR ) ;
}
```

Figure 5.7: A non-productive GAP-L multi-track grammar.

5.3.1 Unreachable Non-Terminals

An unreachable non-terminal is a non-terminal which cannot be part of any derivation tree: a non-terminal B is unreachable if $axiom \not\rightarrow^* B$ is true.

The GAP-C automatically detects unreachable non-terminals. It issues warnings about them and removes them from the internal grammar data structure to simplify later grammar analyses and transformations.

To detect the unreachable non-terminals the transitive-closure of the reachable-from-axiom relation is computed using a standard algorithm.

5.3.2 Productive Checking

A non-terminal is productive, if a parser of this non-terminal terminates. A derivation including a non-productive non-terminal runs indefinitely. Figure 5.6 shows a sample grammar with a non-productive non-terminal. Translating a GAP-L grammar with non-productive non-terminals results in GAP-L programs that do not terminate. Thus, GAP-C checks for non-productive non-terminals and issues error messages for each one found. The check is implemented using a simple fixed point iteration algorithm. At the beginning every non-terminal is initialized as non-productive and each terminal is initialized as productive. In each fixed point iteration each non-terminal data structure is traversed recursively until a call to another non-terminal or terminal symbol. During the traversal the objects of the non-terminal data structure are marked as productive if the linked objects are marked as productive, otherwise they are marked as non-productive. For example,

```
Grammar::init_productive:
  changed = true
  while changed:
    changed = false
    foreach(nt in nts): changed = changed || nt->init_productive

Productive::start:
  changed = false; p = true

Productive::step(x):
  changed = changed || x->init_productive(); p = p && x->productive

Productive::set:
  if p != productive: productive = p; return true
  return changed

Symbol::Terminal::init_productive:
  productive = true; return false

Symbol::NT::init_productive:
  start()
  foreach (alt in alts): step(alt)
  return set()

Alt::Simple::init_productive:
  start()
  foreach (arg in args): step(arg)
  return set()

Alt::Block::init_productive:
  start()
  foreach (alt in alts): step(alt)
  return set()

Alt::Multi::init_productive:
  start()
  foreach (track in tracks): step(track)
  return set()

Alt::Link::init_productive:
  if nt->productive != productive:
    productive = nt->productive; return true
  return false
```

Figure 5.8: Pseudo code of the productive check algorithm. Productive is a superclass of the Alt and Symbol classes.

a right hand side alternative data structure is only productive if all its alternative elements are productive. A multi-track non-terminal is only productive, if all its tracks are productive. See Figure 5.7 for an example of a non-productive multi-track grammar. The algorithm is finished if no object changes its productive state variable anymore. Figure 5.8 shows the pseudo-code of the algorithm.

5.3.3 Yield Size Analysis

The yield or yield string of a derivation tree is the string of its concatenated leafs in a pre-order traversal. The yield function y is of type $T_\Sigma \to \mathcal{A}^*$ and it is defined as $y(f(t_1, \ldots, t_n)) = y(t_1) \ldots y(t_n)$ and $y(a) = a$, where $a \in \mathcal{A}^*$ and f is a function symbol from the signature Σ (see Definition 1, page 18).

The yield size of a non-terminal or terminal symbol A is defined as the tuple of the minimal and maximal size of a yield string of all possible derivation trees that start at that symbol (Equation 5.3). The minimal or maximal yield size of a symbol A is defined by Equation 5.1 or 5.2.

$$ys_{min}(A) = \min\{size(y(t)) \mid t \in \mathcal{L}(A)\} \tag{5.1}$$

$$ys_{max}(A) = \max\{size(y(t)) \mid t \in \mathcal{L}(A)\} \tag{5.2}$$

$$ys(A) = (ys_{min}(A), ys_{max}(A)) \tag{5.3}$$

In the context of a non-terminal or terminal parser that parses the language of the non-terminal or terminal symbol A, the yield size of the parser is the minimal and maximal size of the input string the parser is able to consume.

The yield size analysis is important, as its result is used by several other subsequent semantic analyses, e.g. the table dimension analysis or the runtime analysis. For example, if a non-terminal has a constant maximal yield size of 30 then only a $30 \times n$ table is needed for tabulating results for all parsed sub-words instead of a n^2 table in the general case, where n is the length of the input string. If in the grammar rule $A = f(B, C)$ and it holds that $ys_{min}(B) = ys_{max}(B)$ and $ys_{min}(C) = ys_{max}(C)$, then the resulting non-terminal parser of A only needs to consider one split of the input string between the parsers of B and C. In the general case it needs to consider $O(n)$ splits of the input string.

The description of the yield size computation algorithm for a single-track dynamic programming follows [21]. The basic algorithm is here extended for multi-track GAP-L programs.

The yield sizes of terminal symbols are known a priori. See Table 5.2 for an overview of the yield sizes of the pre-defined terminal parsers. The yield size of the non-terminal symbols is initialized with $(1, n)$. From there the fixed point iteration is started and in every iteration the yield size of the elements of the grammar data structure are computed until nothing changes any more. The termination of the analysis is guaranteed because the yield size interval length is monotonically decreasing. Each track has a yield size associated. In Equations 5.4 to 5.6 the basic

Table 5.2: Overview of the yield sizes of the terminal parsers.

	Yield Size	
Parser	min	max
`STRING`	1	n
`REGION`	1	n
`FLOAT`	1	n
`INT`	1	n
`SEQ`	1	n
`STRINGO`	0	n
`REGIONO`	0	n
`CHAR`, `CHAR(arg)`	1	1
`BASE`	1	1
`CONST_INT`	0	0
`EMPTY`	0	0
`LOC`	0	0
...		

operations are defined over yield sizes of single-track contexts and in Equation 5.7 over tuples of yield sizes of multi-track contexts, where $\circ$ is one of $+=$, $/=$ or $|=$ operations, as used in the fixed point iteration algorithm.

$$+=(a,b) = (a.\min + b.\min, a.\max + b.\max) \tag{5.4}$$
$$/=(a,b) = (\min(a.\min, b.\min), \max(a.\max, b.\max)) \tag{5.5}$$
$$|=(a,b) = (\max(a.\min, b.\min), \min(a.\max, b.\max)) \tag{5.6}$$

$$\begin{aligned} &\circ(< a_1, \ldots, a_n >, < b_1, \ldots, b_n >) \\ &\quad = (< a_1 \circ b_1, \ldots, a_n \circ b_n >) \end{aligned} \tag{5.7}$$

Note that yield size tuples are elements from the set $\mathcal{B} \times \mathcal{B}$, where $\mathcal{B} = \mathbb{N} \cup \{n\}$ and n is a symbol. The arithmetic of minimal and maximal yield sizes has saturating semantics, i.e. $0 - c = 0$ and $n + c = n$, where $c \in \mathcal{B}$.

Figure 5.9 shows the pseudo-code for the computation of the yield size of a non-terminal symbol and Figure 5.10 shows the pseudo-code of the yield size computation of the data structure elements on the right hand side of the non-terminal symbol. The yield size analysis algorithm also considers the use of minimal and maximal yield size filters in the grammar. For example, the yield size of the grammar rule `b = f(REGION) with minsize(2) with maxsize(2) ;` is $(2, 2)$.

```
Symbol::NT:
  foreach (alt in alts):
    alt.init_ys()
    ys /= alt.ys()
  ys |= (minsize, maxsize)
```

Figure 5.9: Yield-size computation pseudo-code for a non-terminal symbol.

```
Alt::Simple::init_ys:
  foreach (arg in args):
    arg.init_ys()
    ys += arg.ys()
  ys |= (minsize, maxsize)

Alt::Block::init_ys:
  foreach (alt in alts):
    alt.init_ys()
    ys /= alt.ys()
  ys |= (minsize, maxsize)

Alt::Link::init_ys:
  ys = nt.ys()
  ys |= (minsize, maxsize)

Alt::Multi::init_ys:
  i = 0
  foreach (track in tracks):
    track.init_ys
    ys[i++] = track.ys
  ys |= (minsize, maxsize)
```

Figure 5.10: Yield size analysis algorithm pseudo-code for the data structure elements on the right hand side of the non-terminal. The variables minsize and maxsize are minimal or maximal yield size restrictions given in the grammar filter. If no filter is present, minsize and maxisze are set to 0 and n.

```
grammar Loop uses Signature(axiom = A)
{

  A = f(B, A, C) | g(CHAR) ;

  B = STRING0 ;

  C = STRING0 ;

}
```

Figure 5.11: A tree grammar with a loop. Non-terminal A is part of a loop, because the minimal yield size of the STRING terminal parser is 0.

5.3.4 Loop Checking

A non-terminal symbol A is part of a loop if there is a derivation $A \rightarrow^* A$ that does not consume characters of the input string. If a top-down parser is generated from A and enters this derivation, then it does not terminate. A bottom-up parser cannot be constructed, since the parse of (i, j) is needed to compute the parse for a sub-word (i, j). Figure 5.11 shows a loop grammar example.

GAP-C implements a loop detection algorithm that is run for every non-terminal of the grammar. It takes the grammar data structure and the yield size information from the yield size analysis computation as input. In case of a detected loop a detailed error message is given and the compilation ends after this phase. Figure 5.3.4 shows the pseudo-code of the recursive algorithm. For each non-terminal the algorithm does a depth-first traversal of the grammar data structure as long as the left hand context and right hand context of the current structure element have both a minimal yield size of 0. A traversal stops if an element is reached which is still part of a running computation. If the non-terminal from the start of the traversal is reached, it is part of a loop.

5.3.5 Max size filter propagation

The yield size analysis algorithm from Section 5.3.3 considers the minimal and maximal yield size filters in the grammar. The filter restrictions in the yield size analyses only propagate bottom-up. See the following grammar as an example for a situation, where it makes sense that the filter information should propagate top-down:

```
grammar Max uses Signature(axiom = A)
{
  A = f(REGION with minsize(12), B, REGION with minsize(7) )
        with maxsize(42) ;
```

```
Symbol::NT::detect_loop:
  Yield::Size p
  foreach (track in tracks): p[track] = (0, 0)
  foreach (alt in alts):
    if alt->detect_loop(p, p, this):
      error(...)

Symbol::NT::detect_loop(left, right, nt):
  if active: return false
  active = true
  r = false
  foreach (alt in alts):
    r = r || alt->detect_loop(left, right, nt)
  active = false
  return r

Alt::Simple::detect_loop(left, right, nt):
  r = false
  foreach (arg in args):
    t = arg.next.ys + ... + args.last.ys
    if (forall i in tracks,
          left[i].ys.min == 0 && t[i].ys.min == 0):
      r = r || arg->detect_loop(left, t, nt)
    left += arg->ys
  return r

Alt::Block::detect_loop(left, right, nt):
  r = false
  foreach (alt in alts):
    r = r || alt->detect_loop(left, right, nt)
  return r

Alt::Link::detect_loop(left, right, nt):
  if this->nt == nt: return true
  return nt->detect_loop(left, right, nt)

Alt::Multi::detect_loop(left, right, nt):
  r = false
  foreach (track in tracks):
    r = r || track->detect_loop(left[track], right[track], nt)
  return r
```

Figure 5.12: Pseudo-code of the loop detection algorithm for the data structure elements of the non-terminal symbol. The parameters left and right contain yield sizes.

```
  B = g(STRING0) ;
}
```

Looking only at non-terminal B, we can derive its yield size from the known yield size of terminal symbol $STRING0$, which is $(0, n)$. This is also computed by the yield size analysis algorithm if it is run on the complete grammar Max. However, in the grammar Max, the non-terminal B is only called from non-terminal A. This call is restricted by a maxsize filter of 42. Thus, one can derive that the effective maximal yield size of B is 42. Taking also the enclosed minsize filters into account we can derive that the effective maximal yield size of B is 23.

In the following we only look at the propagation of maximal yield sizes, because it is asymptotically more interesting: getting a constant maximal yield size instead of n yields a table with a constant number of entries instead of $O(n^2)$ in the general single-track case. Increasing only the minimal yield size yields a table with a constant number of rows or columns less than in the general case. However, the computation of minimal yield size propagation is analogous to the computation of maximal yield size propagation.

The algorithm does a depth-first traversal of the grammar data structure and picks up maximal yield size filter restrictions during this traversal. Figure 5.13 shows the pseudo-code of the algorithm. After the traversal is finished, each non-terminal and right hand side data structure element contains an initialized member `max_ys`. In multi-track contexts with l tracks, `max_ys` is an l-tuple.

5.3.6 Table Dimension Analysis

The tabulation of a non-terminal means that the generated non-terminal parser saves its results for each sub-word in a table. In the general single track case such a table has $O(n^2)$ entries. Depending on the contexts, from which a non-terminal is called, a reduction of the quadratic table size is possible. Consider for example a start symbol which is not called from elsewhere in the grammar. It is called just one time for the sub-word $(0, n)$, i.e. the complete input string, where n is the length of the input. Thus, to tabulate this start symbol, only a constant-size table with one entry is needed.

Another table dimension reduction example is the case where one index of a non-terminal is constant and only the other changes. In this case only a linear table with one row or one column is needed. Consider grammar `tabdim1`:

```
grammar tabdim1 uses Signature(axiom = skipR) {
  skipR = skip_right(skipR, CHAR) |
          skipL # h ;

  skipL = ... ;
}
```

```
Symbol::NT::propagate_max_filter(max_ys):
  if active: return
  active = true
  m = min(ys.max, max_ys)
  if (m <= this->max_ys): active = false; return
  this->max_ys = max(this->max_ys, m)
  foreach (alt in alts):
    alt->propagate_max_filter(m)
  active = false

Alt::Base::propagate_max_filter(max_ys):
  this->max_ys = max(this->max_ys, min(ys.max, max_ys))

Alt::Simple::propagate_max_filter(max_ys):
  Base::propagate_max_filter(max_ys)
  Yield::Size left
  foreach (arg in args):
    right = arg.next.ys + ... + args.last.ys
    max_ys = min(max_ys, ys.max)
    max_ys -= left.low
    max_ys -= right.low
    arg->propagate_max_filter(max_ys)
    left += arg->ys

Alt::Block::propagate_max_filter(max_ys):
  Base::propagate_max_filter(max_ys)
  foreach (alt in alts)
    alt->propagate_max_filter(min(ys.max, max_ys))

Alt::Link::propagate_max_filter(max_ys):
  Base::propagate_max_filter(max_ys)
  nt->propagate_max_filter(min(ys.max, max_ys))

Alt::Multi::propagate_max_filter(max_ys):
  Base::propagate_max_filter(max_ys)
  foreach (track in tracks):
    m = min(ys.max, max_ys)
    track->propagate_max_filter(m[track])
```

Figure 5.13: Pseudo-code of the maximal yield size propagation algorithm.

The non-terminal `skipR` is the axiom and it is only called from its own right hand side. For each call for the sub word (i, j) the non-terminal parser `skipR` calls itself recursively for the sub-word $(i, j-1)$ (because the terminal parser `CHAR` has a yield size of $(1, 1)$). Since the axiom is called for the complete input string $(0, n)$, `skipR` is always called for $i = 0$ and only a linear table with one row is needed.

The benefits of reducing the table dimension of non-terminal parsers are threefold. First, reducing table dimensions saves memory. In the single-track case a reduction from $O(n^2)$ to $O(n)$ is important, in the general two-track case a table needs $O(n^4)$ and there it is mandatory to try to reduce the dimension of the table.

Second, in the case of generating bottom-up CYK-style parsers (see Section 5.4.1) it improves the runtime of the program. For the single-track case, the general CYK-loop is composed of two nested for-loops that explicitly fill the table entries for each sub-word, going from smaller to bigger sub-words. If it is derived that a non-terminal has only a constant or linear table, it can be moved outside of the innermost CYK-loop. This optimization eliminates unnecessary memory accesses, which are expensive. Consider the case, where all non-terminals of a grammar only need tables of linear size and there is no moving boundary on the right hand side. Then an unoptimized general CYK-loop would lead to a program with an asymptotically suboptimal runtime of $O(n^2)$ instead of $O(n)$. Again, in the multi-track case using the general CYK-loop unconditionally is prohibitive.

Third, exact table dimension data improve the runtime computation (Section 5.3.7.1). The runtime computation algorithm depends on the results of the table dimension analysis. For example, the compiler checks if the given table configuration yields a GAP-L program with asymptotically optimal runtime. For this, the runtime under the given table configuration is compared with the runtime under the full table configuration (every non-terminal is tabulated). If the runtime under the full table configuration is asymptotically better, then the compiler issues a warning. Thus, without the table dimension data, the runtime computation always has to assume the runtime of tabulating $O(n^2)$ table entries in the single-track case. In the worst case for a program of asymptotically linear runtime and linear tables, the runtime computation algorithm would falsely derive an asymptotically suboptimal runtime of $O(n^2)$ instead of $O(n)$. In the two-track case, the general tabulation runtime is in $O(n^4)$. In the runtime computation of the pairwise sequence alignment grammar, $O(n^4)$ would overlay the asymptotically optimal runtime of $O(n^2)$ without exact table dimension data.

The previous grammar examples of table dimension reduction possibilities were easy, because an automatic table dimension analysis only has to check the case when one or both indices do not change. However, consider grammar `tabdim2`:

```
grammar tabdim2 uses Signature(axiom = skipR) {
  start = x(CHAR, start) |
          b ;

  b     = x(CHAR, b)     |
```

```
            x(CHAR, c)      |
            y(c, CHAR) ;

  c       = z(REGION, d) ;

  d       = w(REGION, REGION) ;
}
```

Non-terminal `c` is called from two locations on the right hand side of non-terminal `b`. If non-terminal parser `b` is called for the sub-word (i, j) then `c` is called for $(i+1, j)$ and for $(i, j-1)$. Thus, a naive table dimension analysis would derive a quadratic table for non-terminal `c` because both indices are changing. However, from the general point of view, for (i, j) the non-terminal parser `start` and `b` call themselves for $(i+1, j)$. The right index of non-terminal parser `c` is always n or $n-1$, resulting from the call of non-terminal `start` for $(0, n)$. Thus an asymptotically linear table for c is enough (a table with two columns).

For the case of multiple input tracks it is sufficient to call the single-track table dimension reduction algorithm for each track independently. For example, the analysis returns for a two-track program the need of a linear table for the first track and a quadratic table for the second track. This means that a linear table of quadratic tables is needed (or a table with three dimensions). Consider the example of the basic edit distance grammar:

```
grammar Ali uses Signature(axiom = alignment) {

  alignment = nil( < EMPTY, EMPTY > )            |
              del( < CHAR, EMPTY  >, alignment)  |
              ins( < EMPTY, CHAR  >, alignment)  |
              match( < CHAR, CHAR >, alignment)  # h ;

}
```

Only looking at the first track, a vertical split of the two-track grammar results in the following pseudo grammar for the first track:

```
  alignment_t1 = nil( EMPTY )                |
                 del( CHAR, alignment_t1)    |
                 ins( EMPTY, alignment_t1)   |
                 match( CHAR, alignment_t1) # h ;
```

For the second track it results in:

```
  alignment_t2 = nil( EMPTY )                 |
                 del( EMPTY, alignment_t2)    |
                 ins( CHAR, alignment_t2)     |
                 match(  CHAR, alignment_t2) # h ;
```

Note that these pseudo-grammars contain a loop, but this does not matter for table dimension reduction analysis. In both grammars the right index of non-terminals `alignment_t1` and `alignment_t2` is constant. Thus, the table space is reduced from $O(n^4)$ for the generic two-track case to $O(n^2)$.

In GAP-C, a table dimension reduction analysis algorithm is implemented that recognizes if a non-terminal only needs a constant table of one entry or a linear table of one row/column and it checks whether a non-terminal needs an asymptotically constant or linear table as in the example of grammar `tabdim2`. In the case of asymptotically constant or linear tables the exact index range is computed.

The algorithm does a depth-first traversal on the grammar data structure and during the traversal the sums of the minimal and maximal yield sizes of the left and right context of the current object are picked up. If e.g. at a non-terminal the maximal yield size sum of the left context is n and that of the right context is a constant, then only an asymptotically linear table is needed. The case distinction at the non-terminal data structure object has to take simple recursive (a non-terminal directly calls itself) and general recursive non-terminals into account. Thus, the algorithm keeps track, which computations of non-terminals are currently active and it does a re-computation if a later recursion changes the table dimension state of an object, whose computation is not finished.

The table dimension reduction analysis of the first generation ADP compiler [47] is not able to derive a reduction to asymptotically linear tables in examples like `tabdim2`.

5.3.7 Table Design

In the ADP framework, there is a clear separation between search space design, i.e. creating the grammar, and choosing the set of non-terminals whose parsing results are stored in tables. Deriving the set of non-terminals for tabulation, i.e. the table configuration, is called table design. Tabulating every non-terminal in the GAP-L grammar is a safe choice, because the runtime of the resulting GAP-L program is asymptotically optimal. Tabulating no non-terminal of a grammar that describes an exponentially sized search space leads to a program with exponential runtime, because sub-solutions are re-computed recursively. However, according to the runtime computation equations (Section 5.3.7.1) the runtime of a grammar may be asymptotically optimal even if a subset of the non-terminals is tabulated. Figure 5.14 shows an example of such a table configuration.

Thus, finding the minimal table configuration under which the GAP-L program still runs in asymptotically optimal time is a feasible objective, since saved tables may reduce the memory footprint of the generated program significantly. This optimization problem is called the table design problem. Unfortunately, the table design problem is NP-complete [50]. Another motivation for table design is the possibility of reducing the constant runtime factors, if the overhead of storing the table with its entries does not pay off in comparison to just computing the needed parsing results in constant time for each sub-word.

5.3.7.1 Runtime computation

Given the GAP-L grammar and its table configuration ϑ, i.e. the set of tabulated non-terminals, as input, it is possible to compute the asymptotic runtime of its generated parser. The following runtime equations follow [50] and for simplicity it is assumed that the generated parser computes under a unitary algebra (Definition 7, page 28). For convenience, "the runtime of grammar A" or "the runtime of a non-terminal B" is just a shorthand writing for "the runtime of the parser generated from grammar A" or "the runtime of the non-terminal parser that parses the language of non-terminal B".

In the following the right hand sides of the runtime equations contain big-O notation symbols and an asymptotic arithmetic is assumed, e.g. $23n^3 + 42n^2 = O(n^3)$ and $42 = O(1) = 23$.

Equation 5.8 describes the runtime of a non-terminal parser A,

$$rt(A) = \sum_{x \in rhs(A)} calls(A, x) \cdot \begin{cases} rt(x) & \text{if } x \notin \vartheta \\ O(1) & \text{if } x \in \vartheta \end{cases} \tag{5.8}$$

where the function *rhs* returns the set of terminal and non-terminal symbols on the right hand side of the non-terminal A, the function *calls* returns the number of calls from non-terminal A to symbol x and ϑ is the set of tabulated non-terminals. The runtime of a terminal parser is constant. Thus, it is necessary to compute the runtime of the axiom and the runtime of tabulating all tabulated non-terminals for computing the complete runtime of a GAP-L program. Equation 5.9 specifies the complete runtime of a GAP-L program P:

$$rt(P) = rt(S) + \sum_{x \in \vartheta} n^{dim(x)} \cdot rt(x) \tag{5.9}$$

where S is the axiom, n denotes the length of the input string, and the function *dim* returns the table dimension needed to store the parse results for all sub-words. In the single track-case, the image of *dim* is $\{0, 1, 2\}$.

The computation of the runtime of a GAP-L program is an important semantic analysis, since other semantic analyses in the compiler depend on it. First, the compiler checks, whether the runtime under the full table configuration is asymptotically better than under the user-supplied table configuration. If this is the case, then the user-supplied table configuration is asymptotically sub-optimal and a warning message is printed. Second, in table design the runtime of a grammar is computed several times for changing table configurations. Table design is the optimization problem to find the minimal table configuration, under which the runtime of the grammar is still asymptotically optimal (Section 5.3.7).

If the generated parser uses a non-unitary algebra, i.e. an algebra whose objective function returns more than one result, then the asymptotic runtime of the parser may deteriorate. For example, consider the case, where the asymptotic runtime of a GAP-L program under a given table configuration and a unitary algebra is in

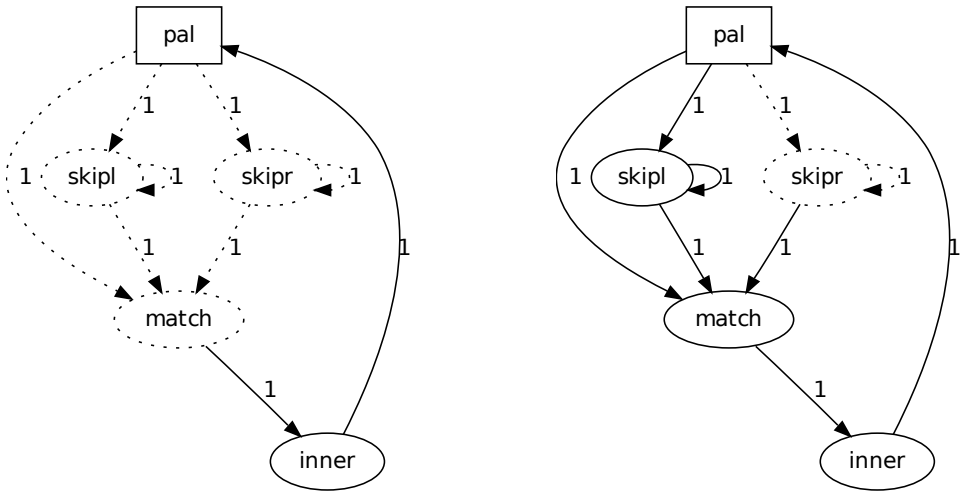
$O(n^3)$. If the parser just uses a pretty printing algebra, whose objective function is the identity, the runtime is in $O(2^n)$, since the search space of exponential size is enumerated.

Constant Factors The Equations 5.8 and 5.9 describe how to compute the asymptotic runtime of a GAP-L program. In practice, constant factors are also of interest. It matters if the runtime under two different table configuration is asymptotic optimal, but the constant factors of the first runtime are prohibitively large (e.g. $999999n^3$ vs. $20n^3$). The runtime functions just need to use a data type that represents polynomials with factors to support constant factors. $O(1)$ is replaced by 1 and the term $n^{dim(x)}$ is replaced by $cells(x)$. The function *cells* returns the asymptotic number of cells to tabulate all results of a non-terminal parser x. If the table dimension reduction analysis (Section 5.3.6) computes a lower bound (including constant factors), then this bound is returned. Note that the constant factors of the runtime computation are an approximation of the real constant factors of a generated parser. For example, not all terminal parsers have the exact runtime of one time unit and syntactic grammar filters, which depend on the input, introduce some conditional code blocks in the generated parser code.

Dependency Graph A dependency graph of a GAP-L grammar is a directed acyclic graph, where each non-terminal and terminal symbol is a vertex and an edge (A, B) specifies that symbol A calls symbol B. The number of calls of A from B is an attribute of the edge (A, B). If a non-terminal is tabulated, then its vertex outline is dotted. Figure 5.14 shows an example of a dependency graph.

For computing the runtime of a GAP-L program, the compiler has to establish the runtime equations for each non-terminal and compute all runtime equations with a recurrent polynomial solver. Such a recurrent solver needs to be efficient, since the runtime computation is the major time consuming operation for the table design analysis. The author is not aware of an open source recurrent polynomial solver which is efficient, stable and maintained. Thus, the GAP-C implements the runtime computation as a recursive algorithm that directly traverses the grammar data structure. The grammar data structure is a dependency graph.

Algorithm Figure 5.15 shows the pseudo-code for computing the runtime of the grammar and the non-terminal symbols. Figure 5.16 shows the pseudo-code of the runtime-computation of the objects on the right hand side of a non-terminal symbol. The runtime computation algorithm does a depth-first traversal of the dependency graph and during that traversal the runtimes of the data structure elements are set. The algorithm updates during traversal a list of non-terminals whose computation is not yet finished (`active_list`) as well as the accumulated runtime (`accum_rt`, or accumulated number of calls) over the traversed path. In the case of a cycle at non-terminal X being detected in the traversal, the accumulated runtime specifies how many recursive calls follow from one call of non-terminal X.



(a) Runtime: n^2

(b) Runtime: 2^n

Figure 5.14: Dependency graphs of a simple palindrome grammar for two different table configurations. Tabulated non-terminals are drawn with a dotted outline and axioms are drawn with a rectangle outline.

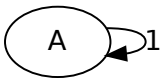
```
Grammar::runtime:
  active_list = []
  rt = axiom->runtime(active_list, 1)
  foreach(nt in tabulated):
    rt += nt->runtime(active_list, 1) * nt->cells()
  return rt

Symbol::Terminal::runtime(active_list, accum_rt):
  return 1

Symbol::NT::set_rec(accum_rt):
  if (accum_rt == 1 && rec <= 1):
    rec = n
  else:
    rec = 2^n

Symbol::NT::set_recs(active list, accum_rt):
  foreach (nt in this, ..., active_list->last):
    nt->set_rec(accum_rt)

Symbol::NT::runtime(active_list, accum_rt):
  if (rt_computed)
    return rt;
  if (active):
    set_recs(active_list, accum_rt)
    return 1
  active = true
  active_list->push(this)
  rec = 1
  foreach (alt in alts):
    rt += alt->runtime(active_list, accum_rt)
    if rt == 2^n:
      rt_computed = true
      active = false
      return rt
  rt *= rec
  active = false
  active_list.pop
  rt_computed = true
  return rt
```

Figure 5.15: Runtime computation pseudo-code of Bellman's GAP grammar, non-terminal and terminal data structure objects.

```
Alt::Simple::runtime(active_list, accum_rt):
  rt = 1
  foreach(arg in args):
    rt += arg->runtime(active_list, accum_rt)
  return rt

Alt::Block::runtime(active_list, accum_rt):
  rt = 0
  foreach (alt in alts):
    rt += alt->runtime(active_list, accum_rt)
  return rt

Alt::Link::runtime(active_list, accum_rt):
  rt = calls
  if (!nt->is_tabulated):
    rt *= nt->runtime(active_list, accum_rt * calls)
  return rt

Alt::Multi::runtime(active_list, accum_rt):
  rt = 0
  foreach(track in tracks)
    rt += track->runtime(active_list, accum_rt)
  return rt
```

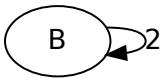
Figure 5.16: Runtime computation pseudo-code of data structure objects of the right hand side of a non-terminal data structure.

```
A = f(A, CHAR) |
    g(CHAR) # h ;
```



(a) One call of non-terminal A yields n calls of A without tabulation.

```
B = u(B, CHAR) |
    v(CHAR, B) |
    w(CHAR) # h ;
```



(b) One call of non-terminal B yields 2^n calls of B without tabulation.

Figure 5.17: Two grammar rules and the corresponding dependency graphs.

If it is one call, then a factor n is multiplied to the current runtime, because in each recursion at least one character of the input string (of length n) is consumed. Otherwise the grammar would be unproductive or would contain a loop. If one call of a non-terminal X yields more than one recursive call, then the overall runtime is exponential. Figure 5.17 shows both cases with dependency graphs. The list of active non-terminals is necessary, since in a cycle over more than one non-terminal, the runtime of every non-terminal of the cycle needs to be updated according to the accumulated runtime. See Figure 5.18 as an example. Figure 5.19 shows an example, where the runtime computation algorithm would compute wrong results without an active list.

The extension of the runtime computation algorithm for multi-track grammars is straight forward. If the grammar uses multiple tracks, then the runtime of each track sums up in an `Alt::Multi` object.

The runtime of the runtime computation algorithm is composed of the runtime of the dependency graph traversal and the runtime of the `set_recs` function. The depth-first traversal of the graph is in $O(|V| + |E|)$. One call of `set_recs` is in $O(|V|)$. If a non-terminal is more than one time in the `active_list` when `set_recs` is executed, then the computed overall runtime is exponential, the traversal of the dependency graph is aborted and `set_recs` is not called again. Thus, the worst-case runtime complexity of the runtime computation algorithm is in $O(|V| + |E|)$. The space used by the algorithm is $O(|V| + |E|)$, which is the space complexity of the dependecy graph data structure representation.

A prerequisite of the runtime computation algorithm is the correct initialisation of the grammar data structure objects, i.e. the number of calls of another non-terminal in a `Alt::Link` object and the number of cells needed to store the parse

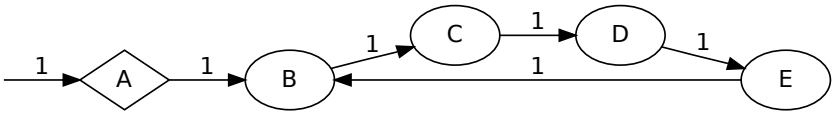


Figure 5.18: A circle over multiple non-terminals in a dependency graph. In the runtime computation traversal beginning at non-terminal A, the active list contains all non-terminals when B is reached the second time. Then the runtimes of all non-terminals of the circle, excluding the first non-terminal from the active list, are multiplied with an additional factor n.

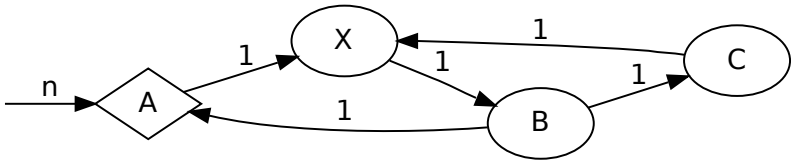


Figure 5.19: A dependency graph example, where one call of non-terminal X yields 2 recursive calls of itself. Without a list of active non-terminals, the runtime computation could not keep track of such cases and would miss the additional 2^n factor in this example.

results for all sub-words in a `Symbol::NT` object. The number of cells is computed in the table dimension reduction analysis (Section 5.3.6). The number of calls of a non-terminal from another one is computed in a depth-first traversal. While traversing the data structure, moving boundaries from the left and right context of a symbol are picked up and translated to a polynomial of the number of calls. A moving boundary is introduced in a grammar data structure if in the derivation there are two or more symbols horizontal neighbors whose minimal yield size is unequal to its maximal yield size. Consider for example the following grammar rule:

```
A = f(REGION, CHAR, REGION, CHAR, g(CHAR, B, CHAR), CHAR) ;
B = b(REGION) ;
```

The terminal parser `REGION` has a yield size of $(1, n)$ and the terminal parser `CHAR` has a yield size of $(1, 1)$. On the right hand side of non-terminal A, the left context of symbol B is `REGION` two times and `CHAR` two times and the right context of B is `CHAR` two times. Since the minimal yield size of B is unequal to its maximal yield size and the minimal yield sizes of two symbols from the left context of B are unequal to their maximal yield sizes, two moving boundaries are introduced. Each one is translated to a factor of n. Thus, the non-terminal B is called n^2 times from the non-terminal A.

5.3.7.2 Approximative Algorithm

Looking at the dependency graph of small GAP-L programs, an experienced ADP programmer has no difficulties to identify those non-terminals, whose tabulation breaks the cycles in the grammar that introduce additional linear or exponential factors in the runtime computation. However, manually deriving an optimal table configuration for larger grammars is tedious and error prone. Thus, the compiler should implement an algorithm that automatically solves the table design problem. Since the table design problem is NP-complete, an exact computation of the optimal table configuration is in $O(2^n)$. An exact algorithm has to do a brute-force search in the exponentially sized search space of all possible table configurations. The exact computation of the optimal configuration being in $O(2^n)$ does not necessarily mean that in practice the runtime of an exact algorithm explodes for moderately sized input grammars. A careful implementation of the exact algorithm is able to exclude certain non-terminals whose tabulation always deteriorates the asymptotic runtime for reducing the search space. Such an implementation computes the optimal table configuration for grammars up to 14 non-terminals in reasonable time in practice. However, hand-written grammars exist with up to 30 – 40 non-terminals and there are Bellman's GAP grammar generating programs like RapidShapes [27] that generate grammars with 200 – 300 non-terminals. Another example of a grammar generating tool is Locomotif [38] that translates graphically constructed RNA structure motifs into ADP RNA-motif-matcher programs. A moderately sized motif easily translates into a grammar with 50 or more non-terminals.

Thus, an approximative optimization algorithm is needed to solve the table design problem heuristically. GAP-C implements a novel table design algorithm that computes a good table configuration. A table configuration is good if the corresponding grammar runs in asymptotically optimal time, i.e. a good table configuration may include more non-terminals than an optimal table configuration.

The algorithm is approximative in the sense that it may return a table configuration which is larger than the optimal table configuration. It never returns a table configuration under which the grammar has an asymptotically suboptimal runtime.

The main idea of the algorithm is to identify a non-terminal whose tabulation breaks one or more runtime-increasing cycles. This step is repeated until the resulting table configuration yields an asymptotically optimal runtime. In breaking a cycle that contains more than one non-terminal, there are all the non-terminals of the cycle to be chosen from and ideally a non-terminal is selected that breaks the most cycles at the same time. The algorithm tries to identify such important non-terminals with the method of scoring all non-terminals of the grammar and then picking that with the highest score. Equation 5.10 shows the used scoring function s:

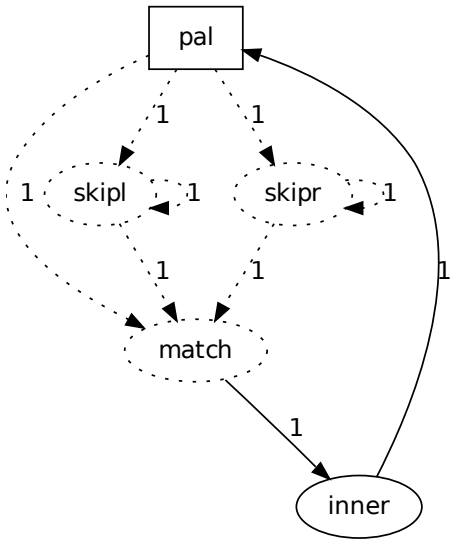
$$s(x) = in(x) \cdot out(x) \cdot selfrec(x) \tag{5.10}$$

where the function *in* returns the number of incoming edges, function *out* returns the number of outgoing edges and function *selfrec* returns the recursion factor of the non-terminal x. Thus, the score of a non-terminal is higher than another, if it is more connected and takes part in more recursions. Figure 5.20 shows the score computation of a non-terminal in the palindrome example grammar. The recursion factor of a non-terminal is a counter that is incremented during several depth-first traversals, if a cycle is detected. Figure 5.21 shows the pseudo-code for computing the recursion factors for each non-terminal.

The approximate table design algorithm works in three phases. First, all non-terminals are scored according to the scoring function s. Second, the non-terminals are stored in a vector. The vector is sorted with the score as key. And in the third phase, iteratively the highest scored non-terminal is set tabulated until the resulting table configuration yields an asymptotically optimal runtime of the grammar. Figure 5.22 shows the pseudo-code of this algorithm.

The worst-case runtime of the first phase is in $O(|V|(|V|+|E|))$, where V is the set of vertices and E is the set of edges of the dependency graph. Sorting the vertices is in $O(|V|log|V|)$. Since the worst-case complexity of one runtime computation is in $O(|V| + |E|)$ the complexity of the third phase is in $O(|V|^2 + |V||E|)$. Thus, the overall worst-case runtime of the algorithm is in $O(|V|^2 + |V||E|)$. The space used by the algorithm is in $O(|V| + |E|)$, because the dependency graph and an additional vector for sorting is stored. In the worst case the algorithm returns the full table configuration and in every case the returned table configuration yields an asymptotically optimal runtime.

Solving or approximating the table design problem means computing a table-



attribute	value
in(match)	3
out(match)	1
selfrec(match)	7

Figure 5.20: Example of the score computation of the non-terminal match under the given table configuration on the left. The score of match is 21.

```
Grammar::init_self_rec:
  foreach (nt in nts):
    nt->init_self_rec
    foreach (nt in nts): nt->active = false

Symbol::init_self_rec:
  if (active):
    if (started):
      self_rec++
    return
  active = true; started = true
  foreach (alt in alts): alt->init_self_rec
  started = false

Alt::Simple::init_self_rec:
  foreach (arg in args): arg->init_self_rec

Alt::Link::init_self_rec: nt->init_self_rec

Alt::Block::init_self_rec:
  foreach (alt in alts): alt->init_self_rec

Alt::Multi::init_self_rec:
  foreach (track in tracks): track->init_self_rec
```

Figure 5.21: Pseudo-code of recursive factor attribute computation for each non-terminal.

```
Grammar::approx_table_design:
  foreach (nt in nts): nt->tabulated = true
  opt = runtime
  v = []
  foreach (nt in nts):
    nt->tabulated = false;
    nt->init_score
    v.push(nt)
  sort(v, \nt -> nt->score)
  reverse(v)
  r = runtime
  foreach (x in v):
    if (opt == r):
      break
    x->tabulated = true
    r = runtime
```

Figure 5.22: Pseudo-code of the approximative table design algorithm.

configuration, under which the runtime of the generated parser is asymptotically optimal. However, this asymptotic condition is not sufficient in practice. Consider e.g. two table configurations of a grammar whose asymptotically optimal runtime is in $O(n^3)$. The first table configuration may yield a runtime of $26n^3$ and the second table configuration may yield a runtime of $666666n^3$. A user would probably accept the first table configuration in any case, even if it contains more non-terminals than the second one, because such a large constant factor is prohibitive in practice. Thus, the table design algorithm in GAP-C also takes constant factors into account during the third optimization phase.

Constant Factors The table design objective is extended, to take constant factors into account and to compute a table configuration that yields a runtime with good constant factors. The extended table design objective is: Find the minimal table configuration, under which the runtime of the generated parser is still asymptotically optimal and the constant factor of the largest polynomial is at most c percent higher than the constant factor of the largest polynomial of the runtime under the full table configuration. In GAP-C c is set to 20, which leads to good results. The extension of the approximative table design algorithm to check for the extended table design objective is straight forward. The runtime calculations need to use a polynomial data type that supports constant factors and another abort condition in the main loop of the third optimization phase. Figure 5.23 contains the pseudo-code.

```
Grammar::approx_table_design(c):
  foreach (nt in nts): nt->tabulated = true
  opt = runtime
  const_factor = opt.last.factor
  v = []
  foreach (nt in nts):
    nt->tabulated = false;
    nt->init_score
    v.push(nt)
  sort(v, \nt -> nt->score)
  reverse(v)
  r = runtime
  foreach (x in v):
    if (opt == r):
      a = r.last.factor
      if (a <= const_factor + const_factor * c / 100):
        break
    x->tabulated = true
    r = runtime
```

Figure 5.23: Pseudo-code of the approximative table design algorithm variant that takes constant factors into account.

5.3.7.3 Benchmarks

Table 5.3 shows runtime and memory usage benchmark results of various GAP-L programs compiled with GAP-C under different table configurations.

The ADPfold (adpf) program is a GAP-L version of RNAfold [25], adpf_nonamb uses the RNAshapes grammar [51], Loco3stem is a simple RNA motif matcher generated with Locomotif [38], pknotsRG [40] is a GAP-L version of the pknotsRG program and the shape programs are differently sized shape matchers generated by RapidShapes. The *mfe* algebra does free energy minimization, the algebra *count* counts the search space and the algebra *pf* computes the partition function. In the benchmark each program was run for all table configurations for 10 randomly generated sequences and the runtime and memory usage values in the table are averages over these runs. All tests were run on the Athlon 64 Linux system described in Section 8.

Every algorithm was run under three different table configurations: the full table configuration, a table configuration derived by a human ADP expert or an expert system and the table configuration computed by the table design algorithm of GAP-C which is described in the previous section. The ADP expert is in most cases the creator of the algorithm. For the automatically generated grammars the expert table configuration is derived by the generators Locomotif and RapidShapes. Note that RapidShapes was developed after the table design algorithm of GAP-C was available and tested to yield good results in practice such that RapidShapes by default uses the table design feature of GAP-C, and the expert table configuration feature of RapidShapes was developed with a low priority of deriving good results.

The shown theoretical runtime expressions are computed by the runtime computation algorithm of GAP-C and are only approximations because different kinds of statements are assumed to yield the same constant runtime unit-cost and other performance relevant factors, e.g. caching effects, are not considered. However, in most cases the theoretical runtime expression rt_{approx} is consistent with the measured practical runtime rt_{prac} data, i.e. it holds that $rt_{\text{approx}}(a) \leq rt_{\text{approx}}(b) \Rightarrow rt_{\text{prac}}(a) \leq rt_{\text{prac}}(b)$, where a and b are two GAP-L grammars. However, for example in the comparison of the ADPfold expert version with the full table configuration version the implication does not hold because of the approximation.

The runtime of the pknotsRG algorithm is in $O(n^4)$ which does not match the theoretical runtime expressions because the table design is not able to consider the index hacking constructs (Section 5.4.5) that eliminate two moving index boundaries.

The results show that designing a table configuration is in most cases a trade-off between memory saving and runtime speedup. For example, the expert table configuration for ADPfold uses an additional table in comparison to the expert table configuration such that the memory usage is only half of the full table configuration version and not a third as the expert version, but it is 24 percent faster than the expert version.

In most cases the table design algorithm computes a table configuration that

algorithm	#NTs	n	strategie	$\lvert\vartheta\rvert$	theo. rt	rt	mem	Ratio rt	Ratio mem
adpf_mfe	11	4000	all	11	$6n^3 + 6280n^2 + 8n + 6$	707	306	1.00	1.00
			expert	4	$15n^3 + 6276n^2 + 11n + 6$	672	92	1.05	3.30
			design	5	$6n^3 + 6274n^2 + 8n + 6$	535	123	1.32	2.48
adpf_nonamb_mfe	26	1500	all	26	$48n^3 + 6042n^2 + 18n + 8$	337	1136	1.00	1.00
			expert	9	$137n^3 + 6052n^2 + 18n + 3$	799	363	0.42	3.13
			design	15	$57n^3 + 6044n^2 + 18n + 4$	451	620	0.75	1.83
loco3stem_count	23	4000	all	23	$8n^3 + 311n^2 + 6n + 4$	802	1344	1.00	1.00
			expert	8	$19n^3 + 465n^2 + 6n + 4$	505	428	1.59	3.14
			design	9	$9n^3 + 476n^2 + 6n + 4$	525	489	1.53	2.75
pknotsRG_mfe	25	1000	all	25	$30n^8 + 14n^3 + 6348n^2 + 8n + 6$	282	62	1.00	1.00
			expert	6	$54n^8 + 172n^3 + 6321n^2 + 35n + 6$	425	20	0.66	3.06
			design	19	$34n^8 + 14n^3 + 6343n^2 + 8n + 6$	292	56	0.97	1.10
shape1_pf	43	600	all	43	$88n^3 + 11954n^2 + 30n + 18$	106	437	1.00	1.00
			expert	12	2^n	-	-	-	-
			design	17	$99n^3 + 11934n^2 + 33n + 20$	113	200	0.94	2.18
shape2_pf	95	600	all	95	$200n^3 + 29978n^2 + 30n + 18$	135	1084	1.00	1.00
			expert	29	2^n	2852	337	0.05	3.21
			design	68	$200n^3 + 29956n^2 + 28n + 12$	136	810	0.99	1.34
shape3_pf	195	600	all	195	$468n^3 + 54098n^2 + 30n + 18$	169	2329	1.00	1.00
			expert	59	2^n	-	-	-	-
			design	147	$468n^3 + 54055n^2 + 28n + 12$	169	1794	1.00	1.30

Table 5.3: Runtime and memory usage benchmarks of various GAP-L programs compiled with GAP-C using various table configurations ϑ. The first column encodes the name of the algorithm and the used algebra (see text), the second column shows the number of non-terminals (NT) of the grammar and n denotes the length of the input sequence. The strategy is one of all, expert or design, i.e. a table configuration where everything is tabulated, a table configuration derived by a human ADP expert or an expert system respectively a table configuration computed by the table design algorithm of GAP-C. The theoretical runtime (theo. rt) expressions are approximations as derived by the runtime computation algorithm of GAP-C. The runtime (rt) is measured in seconds and the memory usage (mem) in megabytes as an average over 10 sequences (see text). The ratios are given in reference to the values under the full table configuration.

yields a better runtime than the expert table configuration. Only for the Loco3stem program the expert version is 4 percent faster. In comparison to the runtime of the programs under the full table configuration the design table configuration yields speedups greater 1 or a similar runtime in most cases. Only the runtime of the table design version of adpf_nonamb is 33 percent slower than the full table configuration version.

The table design algorithm removes several tables from tabulation in all cases, i.e. it maximally tabulates 75 percent and minimally tabulates 35 percent of the non-terminals. The practical memory usage is maximally reduced by a factor of more than one half.

In conclusion, the results show that the table design algorithm of GAP-C works well in practice. For several non-trivial GAP-L programs the computed table configurations yield a better runtime and less memory usage than versions using a full table configuration or even an expert table configuration.

5.3.8 Type Checking

Type checking means checking for type errors in GAP-L programs. A type error occurs, when e.g. a function symbol in the signature declaration has two arguments, but is used with three arguments in the grammar, or when the third argument of a function symbol is declared as of type **int** in the signature and in the algebra the third argument is of type **char**. Another example is a non-terminal with two alternatives on the right hand side of different types. The compiler has to detect such errors, because it does not know how to generate a correct parser from it.

When programming ADP in Haskell-ADP, the Haskell compiler or interpreter does type-inference on the input program. Since the Haskell type inference system does not know anything about ADP, the type inference error message may expose implementation details of ADP Domain Specific Language constructs, in the case of a type error in ADP code. Such error message may obfuscate the real location and the perhaps simple cause of the error message. Consider this correct grammar snippet:

```
formula = mult(formula, times, formula)
```

If we delete the second argument of the function symbol `mult`, a simple error is introduced and yields this type inference message in the interactive Haskell interpreter hugs:

```
ERROR "El2.lhs":116 - Type error in application
*** Expression     : add <<< formula ~~- plus ~~~ formula |||
                     mult <<< formula ~~- formula
*** Term           : add <<< formula ~~- plus ~~~ formula
*** Type           : (Int,Int) -> [Char]
*** Does not match : (Int,Int) -> [Char -> Char]
```

```
Error:
            mult(formula, formula) # h ;
            ^--^
e2.gap:186.13-16: Function mult has 2 arguments, but
Error:
  answer mult(answer, alphabet, answer);
         ^--^
e2.gap:9.10-13: it is defined with 3 arguments here.
```

Figure 5.24: Example of a type error message of GAP-C. The function symbol application in the grammar misses the second argument.

When integrating a special purpose type checking algorithm into the compiler, it is possible to derive more useful error messages in type error cases. The GAP-C implementation succeeds in displaying the exact location and context of type errors where the GAP-L programmer has to fix the error. See Figure 5.24 for an example error message.

The type checking analysis of GAP-C is divided into three phases. First, the grammar is checked against the signature. Second, each algebra is checked against the signature. And last, types in the body of algebra functions are checked.

If type errors are detected in one phase, then the following type checking phases are skipped. Otherwise, the compiler would display a lot of redundant type error messages which reference the same error, but report it in different type contexts. Such messages would decrease the signal to noise ration of the error message output. Consider e.g. a GAP-L program where x algebras correctly implement the signature. If the grammar uses symbols from the signature in an erroneous way, then the type checking of the grammar against each algebra necessarily finds the same errors. The difference is that sort symbols are replaced by concrete types.

The type checking of GAP-C is related to the ADP Typechecker [44]. It started as a standalone program and was later integrated into the old ADPC. As part of the ADPC it can check ADPC-ADP language programs, but no Haskell-ADP programs. In comparison to the GAP-C type checking design, it does not stop checking algebras against the grammar if the grammar uses signature functions incorrectly.

Checking the grammar against the signature means that the signature declarations are inserted into the grammar data structure, i.e. the type checking algorithm traverses the grammar data structure and at each object that uses a function symbol, it is looked up in the signature. The return type of the signature function is propagated upwards in the data structure and the types of the arguments are propagated downwards. A type error is found, if one propagated type information does not match another. Other errors are present, if a function symbol is not declared in the signature or the number of arguments differs.

Checking an algebra against the signature means checking each algebra function

definition against the corresponding signature function symbol declaration. The algebra definition contains a mapping between signature sorts and alphabet type to concrete types. Thus, during the checking this information is looked up in a symbol table.

The body of an algebra function is not directly checked by GAP-C. Instead, the compiler generates the algebra function code and includes line and file position pragmas in the output. These pragmas are understood by a C++ compiler: if the C++ compiler detects type errors in the generated algebra code they are reported in the context of the GAP-L source program. This strategy eliminates the need to implement a classic type checker for imperative algebra code inside GAP-C. A C++ compiler already does a good job at type checking that code.

5.3.9 List analysis

The list analysis algorithm takes the results of the algebra characteristics analysis (Section 2.2.3) and the grammar data structure as input and computes the worst-case list sizes resulting from symbol parser calls at different elements of the grammar data structure. Using fixed point iteration, the algorithm does several depth-first traversals of the grammar data structure to propagate list size influencing factors until the computed list sizes do not change anymore. In the beginning the worst-case list sizes of the non-terminals are initialized with n, which denotes the length of the input.

During a traversal the following rules are applied:

$$\text{lsize}(\text{X} = \text{Y} \;;) = \text{lsize}(\text{Y}) \tag{5.11}$$

$$\text{lsize}(\text{X} \;\#\; \text{h}) = \text{lsize}(\text{h}) \tag{5.12}$$

$$\text{lsize}(\text{X} \mid \text{Y}) = \text{lsize}(\text{X}) + \text{lsize}(\text{Y}) \tag{5.13}$$

$$\text{lsize}(f(a_1, \ldots, a_k)) = \text{lsize}(a_1) \cdot \ldots \cdot \text{lsize}(a_k) \cdot n^b \tag{5.14}$$

$$\text{lsize}(< X_1, \ldots, X_k >) = \text{lsize}(X_1) \cdot \ldots \text{lsize}(X_k) \tag{5.15}$$

The exponent b denotes the number of unrestricted moving index boundaries in the right hand side symbol context inside and outside of f. Looking at the context of a single-track, an unrestricted moving index boundary is introduced if two symbols with maximal yield size of n are placed side by side, possibly interleaved with constant yield sized symbols and nested function symbol applications. Each additional maximal yield sized symbol adds another index boundary. The number of moving boundaries of multiple-tracks multiply with each other.

The role of an objective function influences the worst-case list-size of a non-terminal, as computed by the algebra characteristics computation (Equation 5.11). In the case of a scoring algebra, the non-terminal with an objective function on the right hand side then has a worst-case list size of 1.

The results of list-size analysis are used in two ways. First, the use of lists is unnecessary at locations where a worst-case list-size of 1 is detected, and is thus eliminated. This leads to more efficient code, since the memory and cache is used more efficiently and the list-accesses include more overhead. Second, the compiler checks for missing objective function applications on the right hand side of non-terminal rules. If an objective function is missing then it is checked, whether the application of an objective function at that location would reduce the worst-case list-size and improve the asymptotic runtime of the resulting program. If this is the case, a warning message with the exact location of possible objective function application is printed.

5.3.10 Dependency analysis

The compiler supports the generation of two different styles of parsers: top-down Unger-style parsers and bottom-up CYK-style parsers. In top-down parsing, the control flow of the parser functions implicitly fills a table entry of a tabulating non-terminal parser before it is accessed by a computation of another table entry. In bottom-up parsing, the tables are filled explicitly in a main loop. This main loop has to satisfy two conditions. First, smaller entries for smaller sub-words have to be computed before larger sub-words are addressed. Second, for each sub-word a table entry of one table has to be computed before it is accessed by another non-terminal parser computation for the same sub-word. For more details on the different parsing schemes see Section 5.4.1. The first condition is satisfied by generating a correct loop structure. The results of the dependency analysis are needed to generate an ordering of non-terminal tabulate calls from the inner CYK-loop that satisfies the second condition.

In the following it is assumed for simplicity that the input is a one-track grammar.

If a GAP-L program contains multiple tabulated non-terminals, then it is possible that computation of a table entry (i, j) of one non-terminal depends on the computed entry (i, j) of another table. In bottom-up parsing the compiler has to derive these dependencies and generate the parser results table filling calls from the main CYK-loop in a non-conflicting order.

A tabulating non-terminal parser A depends on another tabulating non-terminal parser B, if there is a derivation from A to B and for a call of A for the sub-word (i, j), the parser B is called for the same sub-word (i, j), i.e. during the derivation the left and right context of the call location of B is empty. Figure 5.25 shows two grammar snippet examples.

The dependency derivation algorithm works on the grammar data structure and depends on the results of the yield size analysis (Section 5.3.3), where the yield size of each object is computed. The algorithm works in two phases. First, for each non-terminal the algorithm does a depth-first traversal as long as the left and right context of a link to another non-terminal is empty. If both contexts are empty, then the dependency is recorded in a list. A context is empty, if all neighboring leaves in the derivation tree have a minimal yield size of 0. In the second phase, the

```
A = f(REGION0, B) ;

B = g(CHAR) ;
```
(a) Parser P_A depends on parser P_B.

```
A = f(REGION, B) ;

B = g(CHAR) ;
```
(b) Parser P_A does not depend on parser P_B

Figure 5.25: Two grammar snippets, where a parser execution dependency for one sub-word is analyzed. The minimal yield sizes of the terminal parsers REGION0 and REGION are 0 and 1, respectively.

```
Grammar::parser_deps:
  l = []
  foreach (nt in nts):
    nt->collect_deps(l)
  tsort(l)
  return l

Symbol::NT::collect_deps(l):
  Yield::Size left, right;
  foreach (alt in alts):
    alt->collect_deps(l, this, left, right)

Alt::Simple::collect_deps(l, n, left, right):
  foreach (arg in args):
    t = arg.next.ys + ... + args.last.ys
    arg->collect_deps(l, n, left, right+t)
    left += arg->ys

Alt::Block::collect_deps(l, n, left, right):
  foreach (alt in alts):
    alt->collect_deps(l, n, left, right)

Alt::Link::collect_deps(l, n, left, right):
  if left == right == ( (0, _), ..., (0, _) ):
    l.push_back( (n, nt) )

Alt::Multi::collect_deps(l, n, left, right):
  foreach (track in tracks):
    track->collect_deps(l, n, left[track], right[track])
```

Figure 5.26: Pseudo-code of the parser dependency collection algorithm.

resulting list is sorted topologically. Figure 5.26 contains the pseudo-code of the algorithm. For the topological sort, the collected dependencies must not contain cycles. This is the case, because the dependency analysis is only executed, if the loop analysis (Section 5.3.4) does not find a loop. The worst-case runtime of both phases is in $O(|V| + |E|)$, where V is the set of non-terminals and E is the set of links in the grammar data structure.

5.3.11 Non-terminal inlining

The compiler supports inlining of non-terminal symbols. Inlining a non-terminal A means that it is removed from the grammar and the non-terminal call is replaced by a copy of the right hand side of A at each calling location. This grammar transformation is only possible, if the inlined non-terminal is not part of a cycle. Inlining a non-terminal may increase or decrease the practical runtime of the generated program. The code generation phase implements a non-terminal parser as a code function for each non-terminal. If the overhead of a function call is more expensive than the direct computing of the function body, then inlining improves the constant factors of the runtime. In the inlining phase, the compiler inlines only those non-terminals which are not part of a cycle and contain a simply structured right hand side. A right hand side is considered simply structured, if it just contains one alternative, no objective function application and a worst-case answer list size of 1. Because of this simple structure, the algorithm approximates that inlining improves the runtime of the resulting code, i.e. inlining depends on the computed data from the list analysis. Depending on the selected product, a present objective function application could be present in the source grammar, but would be removed by a previous objective function elimination phase.

The code generation subsystem of the compiler does not contain more low-level inlining phases for inlining very small generated code functions or runtime library functions. Since current C++ compilers already include good general purpose inline optimization phases, GAP-C does not need to duplicate this effort for more low-level code. It is sufficient that the runtime library code in question and the generated low-level code is structured in such a way that a generic C++ inline optimizer is not limited in its operation. When an inlining should be considered by the compiler of the generated code, it needs access to the function definitions in all translation units and the function must not be too large.

5.3.12 Index analysis

The index analysis uses the results from the yield size analysis (Section 5.3.3) and the table dimension analysis (Section 5.3.6) as input and creates index expressions for the elements of the grammar data structure. In multi-track programs, each track is processed independently, because the index boundaries of one symbol access of one track do not influence the boundaries of the other tracks. For each non-terminal the index analysis does a depth-first traversal of the data structure

```
iloop = il( BASE, BASE, REGION with maxsize(30), closed,
            REGION with maxsize(30), BASE, BASE) # h ;
```
(a) GAP-L non-terminal

```
for (k_0 = i + 3; k_0 <= j - 10 && k_0 <= i + 32; ++k_0)
  for (k_1 = j - (k_0 + 7) >= 32 ?
              j - 32 : k_0 + 7; k_1 <= j - 3; ++k_1)

    BASE(i, (i + 1))
    BASE(i + 1, i + 2)
    REGION(i + 2, k_0)
    closed(k_0, k_1);
    REGION(k_1, j - 2)
    BASE(j - 2, j - 1)
    BASE(j - 1, j)
```
(b) index pseudo-code

Figure 5.27: Grammar snippet and the computed indices for symbol accesses by the index analysis algorithm.

that represents the right hand side. When on the right hand two symbols with the minimal yield size unequal to the maximal yield size are horizontally side by side, then a new moving index boundary is detected and a new index-variable is created. The detection of moving boundaries keeps track of nested function symbol applications and interleaved symbols with constant minimal and maximal yield sizes. Each new index variable introduces a new loop construct that contains the upper and lower bounds of this index depending on the outer indices of the outer non-terminal, which are needed for the code generation phase. When a symbol with minimal yield size equal to maximal yield size is traversed, then the value is collected to compute the index boundaries of the following symbols as tight as possible.

Figure 5.27 shows an example of a GAP-L grammar snippet and resulting indices of parser calls.

During index analysis indices are eliminated if the table dimension analysis has found one or more indices of a non-terminal parser being constant.

Another part of the index analysis is the generation of conditional expressions that consider implicit and explicit yield size limits. In code generation they are then used to generate if-statements that guard against unnecessary executions of code blocks depending on the size of the parser sub-word argument.

5.4 Code Generation

The code-generation phase of GAP-C takes the AST and the results from the semantic analyses as input and generates optimized target code for the backend. The tasks of the code-generation phase are the generation of an efficient implementation of yield parsing (Section 5.4.1), the generation of code that exploits parallelism on shared memory architectures (Section 5.4.2) and the application of several backtracing schemes where possible in the generated code (Section 5.4.3). In Section 5.4.4 the Window-Mode feature is described and Section 5.4.5 presents the code-generation for index-hacking constructs, for application domain specific optimizations, and discusses their motivation.

5.4.1 Parsing Schemes

The compiler implements two different schemes for generating the non-terminal parser code: top-down Unger-style parsing [53] and bottom-up CYK-style parsing [58].

5.4.1.1 Top-Down

In top-down parsing, each non-terminal parser is generated as a function. At the beginning of a parse, the axiom parser is called for the complete input as argument. The parser code then recursively calls the referenced non-terminal parsers on the right hand side for all possible splits of the current sub-word. When a non-terminal parser is tabulating, then it tests at every call, if it was already called with the same sub-word argument. If yes, then it returns the already computed result from the table, else it computes the parse and saves it into the table. Figure 5.28(b) shows the pseudo-code of a top-down non-terminal parser for a non-terminal from an example grammar.

Using the results from the yield size and index analysis (Sections 5.3.3 and 5.3.12) the top-down parser code generation eliminates recursions into unnecessary splits of the sub-word argument that cannot return a valid parse, if e.g. a split would yield a parser call with a sub-word argument of size greater than the maximal yield size of the non-terminal.

The runtime of a top-down parser is in $O(n^{l+m})$ where n is the maximal track length, l is the maximal number of dimensions of a non-terminals table and m is the number of non-restricted moving index boundaries on the right hand side of a non-terminal. A non-restricted moving boundary is introduced if one input track contains two links to symbols with a maximal yield size of n. Each additional link to symbol with maximal yield size of n introduces another moving boundary. Thus, the top-down parser runtime corresponds to the equations in the runtime computation analysis (Section 5.3.7.1).

The top-down parsing scheme is comparable to the parsing of the Haskell-ADP. The Haskell-ADP parser combinators also work top-down. The on-demand table entry checking is implicit, since Haskell uses lazy-evaluation.

```
grammar nussinov uses Fold(axiom=struct) {

struct = nil(EMPTY)           |
         right(start, CHAR)   |
         split(start, pair(CHAR, start, CHAR)
           with basepairing) # h ;

}
```

(a) Grammar

```
comp_N(0, n)

comp_N(i, j):
  if (computed(N, i, j))
    return N[i,j]
  ...
  foreach k, i<k<j:
    ...
    tmp = comp_N(i, k-1)
          + comp_N(k+1, j-1)
    N[i,j] = max(N[i,j], tmp)
    ....
  return N[i,j]
```

(b) Top-Down

```
for j=0; j<n; ++j
  for i=j+1; i>1; i--
    compute_N(i-1,j)

compute_N(i, j):
  ...
  foreach k, i<k<j:
    ...
    tmp = N[i,k-1]
          + N[k+1,j-1]
    N[i,j] = max(N[i,j], tmp)
    ....
```

(c) Bottom-Up

Figure 5.28: Nussinov algorithm grammar and two pseudo-code skeletons for top-down and bottom-up evaluation of a basepair maximization algebra.

5.4.1.2 Bottom-Up

In bottom-up parsing the tabulating non-terminal parsers are called explicitly from a main loop to fill the tables with parser results. The tables must be filled in the order of increasing sub-word size, because a smaller sized entry could be referenced to compute the parse result of a table entry. If there are more than one tabulating non-terminal parser then the order of computing table entries for same-sized sub-words must take the parser dependencies into account (Section 5.3.10). Non-tabulating non-terminal parsers are called during the computation of table entries top-down as recursive functions. If the axiom is tabulated, then the parsing result is stored in the axiom table in the entry that represents the whole input parse (e.g. $(0, n)$ in the single-track case). Else the axiom parser is called top-down with the whole input as argument. Figure 5.28(c) shows the pseudo-code of a bottom-up parser for a small single-track grammar.

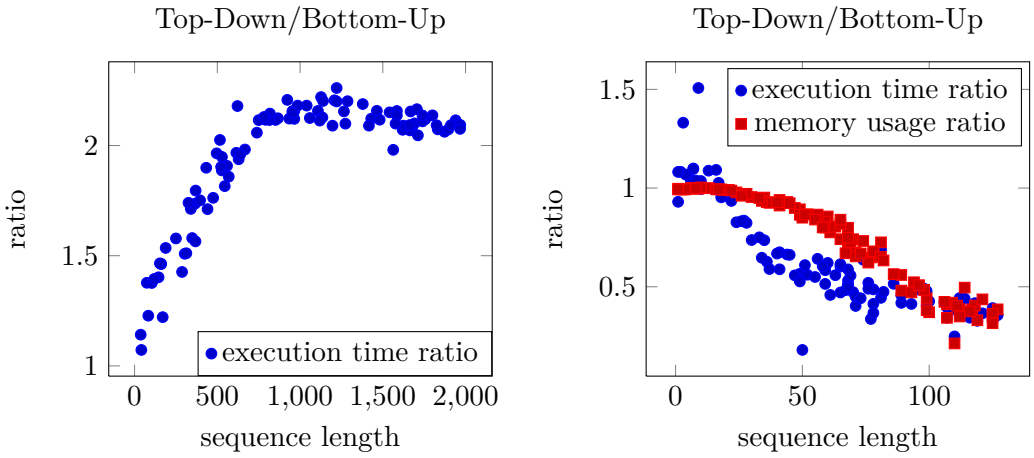
The worst-case runtime of a bottom-up parser is the same as the runtime of a top-down parser. The main loop iterates over $O(n^l)$ sub-words and for each sub-word, while computing the entries, $O(m)$ non-restricted index-boundaries have to be considered.

The code generated by ADPC uses bottom-up parsing.

5.4.1.3 Benchmarks

The practical runtime, i.e. the constant factors, of the generated parsers may differ significantly in the two parsing schemes. On the one hand, top-down parsing introduces some overhead, because a recursion stack has to be administered and for each sub-word each tabulating non-terminal parser has to execute table checking. In bottom-up parsing the conditional code is eliminated and a stack is only needed for non-tabulated parsers. On the other hand, if the grammar of the programming introduces sparseness, then the entries of the tables are accessed sparsely as well. During bottom-up parsing all table entries are computed. But in top-down parsing, sparseness may prune some derivations which yield sparsely filled tables. Sparseness is introduced due to grammar filters (e.g. `stackpairing`) or non-parsable symbols on the right hand side of a non-terminal. The effect of sparseness depends on the used algebra as well. A well structured CYK-style loop is able to exploit caching and pre-fetching effects of the CPU, if for example a scoring algebra works on an elementary data type. These effects may outweigh reduced computation due to sparseness. However, for an algebra with more expensive operations, the top-down overhead may pay off in reducing the number of algebra computations. Conversely, in a dense grammar the top-down overhead would yield no benefit.

Figure 5.29 shows two plots of the top-down vs. bottom-up runtime ratio of the compiled ADPfold algorithm for two different algebra products. The ADPfold grammar is basically a GAP-L version of RNAfold[25]. It introduces sparseness, because large parts of the grammar are protected by `stackpairing` filters (Figure 5.30). The mfe algebra computes the minimum free energy, i.e. the energy contribu-



(a) mfe

(b) shape · mfe

Figure 5.29: Runtime and memory usage ratios of top-down and bottom-up parsers for two different products.

```
closed = { stack | hairpin | leftB | rightB |
           iloop | multiloop }
             with stackpairing # h ;
```

Figure 5.30: Non-terminal rule that is protected by the `stackpairing` syntactic filter, i.e. the right hand side is only parsed if the first and second character of the sub-word form basepairings with the last and second-last one.

tions of substructures are added and the objective function minimizes over all values. In this case, the bottom-up computation is two times faster than the top-down computation, even if the grammar introduces some sparseness. The `shape*mfe` product computes the MFE for each shape in the search space. The number of shapes grows exponentially with the input size, i.e. the product objective function returns a list of shapes and the runtime of a parser for each sub-word increases from $O(n)$ in the mfe case to (nm^2), where m is the maximal list size of an argument parser. In this case, the overhead of top-down parsing does pay off and the top-down runtime and space usage is just half the bottom-up ones. During bottom-up parsing a lot of sub-word entries are computed which represent successful sub-parsers but which are not used by any computation of bigger sub-word parses, because at that level a stackpairing filter returns false for all those bigger sub-words.

5.4.1.4 Argument Reordering

In both top-down and bottom-up parser code generation, the compiler does a reordering of the argument call order of function symbols on the right hand side of a non-terminal. The heuristic used there is to score each argument and then sort the arguments with decreasing score. An argument is scored higher than another if it is approximately more likely to be computed less expensively and thus may return an unsuccessful parse with less computation, i.e. terminal parsers are scored higher than non-terminal parsers and local grammar filters increase the score.

5.4.1.5 Sparseness

Pure top-down parsing exploits sparseness in a grammar, but pre-processing of the input and on-the-fly bookkeeping to eliminate moving index boundaries can exploit the sparseness on a higher level, using algebra and grammar properties. See Section 10.1 for a discussion.

5.4.1.6 CYK Loops

In the single-track case, the main CYK-style loop is constructed with two nested for-loops as displayed in Figure 5.28(c). Using results from the table dimension analysis (Section 5.3.6) the basic CYK-style loop is optimized during code generation. For example, the table entry filling code for a parser P does not need to be called for every sub-word (i, j), if the table dimension analysis shows that P needs only a constant sized table. In that case the tabulating parser only needs to be called for the complete sub-word $(0, n)$. Analogous to that, a parser that only needs a linear table, because it is always called with a constant left index, does not need to be called from the innermost for-loop, but for every j. The CYK-style loop is specialized and rolled out during code-generation and the parser-calls are moved out of the nested loops as much as possible to save unnecessary calls to tabulating code. Figure 5.31 shows the pseudo-code of this optimization. If the asymptotically optimal runtime of the GAP-L program is in $O(n^x)$, where $x > 1$, then this

```
for (unsigned j = 0; j < n; ++j) {
  for (unsigned i = j + 1; i > 1; i--) {
    nt_tabulate_A(i-1, j);
  }

  unsigned i = 1;
  nt_tabulate_A(i-1, j);
  nt_tabulate_B(i-1, j);
}

unsigned j = n;
for (unsigned i = j + 1; i > 1; i--) {
  nt_tabulate_A(i-1, j);
  nt_tabulate_C(i-1, j);
}

unsigned i = 1;
nt_tabulate_A(i-1, j);
nt_tabulate_B(i-1, j);
nt_tabulate_C(i-1, j);
nt_tabulate_D(i-1, j);
```

Figure 5.31: Specialized 2-track CYK-style loop where non-terminal A needs a quadratic table, B and C a linear table and D a constant sized table.

optimization does not change the asymptotic runtime of the code, but reduces the constant runtime factors of the generate code. Otherwise, if the asymptotically optimal runtime is in $O(n)$, just generating the basic CYK-style loop instead of doing this optimization would yield a program with asymptotically suboptimal runtime. In practice, reducing constant runtime factors may yield significant runtime improvements.

The CYK-loop specialization is generalized in the multi-track case. Figure 5.32 shows the pseudo-code of the loop code generation algorithm. The optimization is applied recursively on the tracks and recursively loops with empty bodies are eliminated. Consider the pairwise sequence alignment algorithm as a two-track GAP-L program example. The general two-track CYK-style loop would consider all sub-words, i.e. iterating over $O(n^4)$ index combinations of the two tracks. Applying this optimization yields a specialized loop that eliminates two of four indices that are constant and thus iterates over $O(n^2)$ index combinations, which is the asymptotically optimal runtime of the algorithm. The resulting two-fold nested loop resembles a handwritten loop if manually implementing the control flow of the pairwise sequence alignment algorithm textbook recurrences in an imperative programming language.

5.4.2 Parallelization

Today, multi-core CPUs and multi-socket computer systems are widely available. One can interpret this trend as a result of Moore's Law that states: "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [...]. Certainly over the short term this rate can be expected to continue, if not to increase." [32] This means that every year more transistors are available for producing a CPU at the same costs as last year. Thus, putting more cores in one CPU-package is one way to utilize the increasing transistor count. Figure 5.33 shows that this law still fits with current developments.

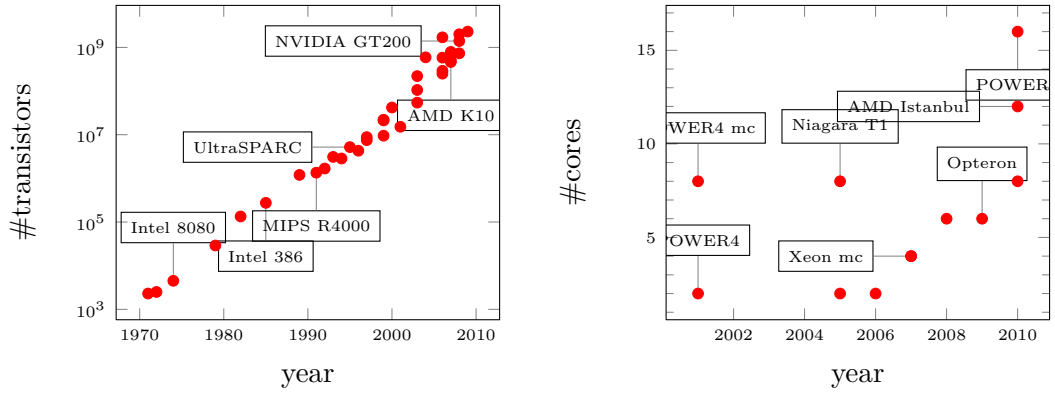
As a consequence, new algorithms should scale well on parallel machines.

There are several parallel versions of dynamic programming algorithms on sequences. For example, [30] describes a parallel version of the pairwise sequence alignment algorithm. Since the computation of a table entry (i, j) depends on the values of the neighboring entries $(i-1, j)$, $(i, j-1)$ and $(i-1, j-1)$, it is not possible to compute the entries in parallel in arbitrary order. The described parallelization scheme respects the table entry dependencies and the edit distance table is filled in a diagonalized fashion (diagonal after diagonal), because the computation of the entries on the diagonal does not depend on each other and thus it is possible to compute them in parallel. Another example is the parallel version of McCaskill's partition function algorithm [17], which uses a similar parallelization scheme. McCaskill's algorithm is a single-track $O(n^3)$ algorithm that takes an RNA sequence as input.

The GAP-L compiler supports the generation of code that is parallelized and optimized for shared memory architectures. It aims at shared memory architec-

```
partition_nts(tord, all, inner, left, right, track):
  foreach (nt in tord):
    if (!nt.is_cyk_left(track) && !nt.is_cyk_right(track) &&
        !nt.is_cyk_const(track))
      inner.push_back(*i);
    if (!nt.is_cyk_right(track) && !nt.is_cyk_const(track))
      left.push_back(*i);
    if (!nt.is_cyk_left(track) && !nt.is_cyk_const(track))
      right.push_back(*i);
    all.push_back(*i);

print_cyk(tord, track)
  partition_nts(tord, all, inner, left, right, track);
  print("...");
  if (!inner.empty())
    print("for (..) { for (..) ... {");
    print_cyk2(inner, track);
    print("}");
    if (!left.empty())
      print_cyk2(left, track);
    print("}");
  if (inner.empty() && !left.empty())
    print("for (..) {");
    print_cyk2(left, track);
    print("}");
  if (!right.empty())
    print("for (..) {");
    print_cyk2(right, track);
    print("}");
  if (!all.empty())
    print_cyk2(all, track);

print_cyk2(tord, track):
  if track == tracks:
    // generate nt tabulating calls
  else:
    print_cyk(tord, track+1)
```

Figure 5.32: Pseudo-code of the generic multi-track optimized CYK-style loop generation algorithm. The `tord` argument contains the table configuration topological sorted according to the parser dependencies.



(a) transistor counts

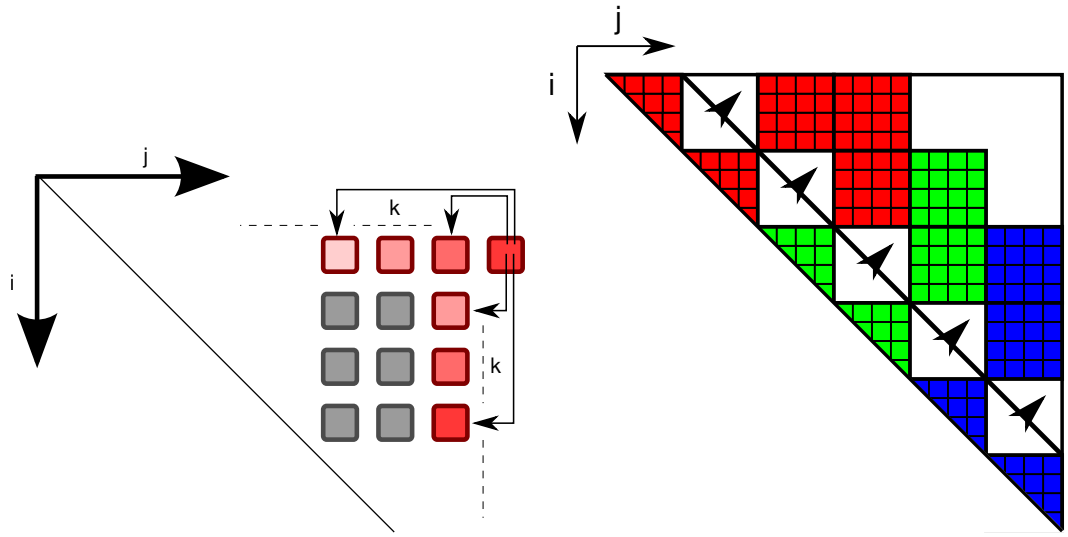
(b) number of cores

Figure 5.33: The increase of transistor counts in CPUs and the trend to integrate more cores into one CPU package.

tures, because the communication overhead of $O(n^3)$ single track DP algorithms is likely to decrease significantly the parallel efficiency of the resulting program in a message passing environment. In such an algorithm a moving boundary at the right hand side of a rule means that for computing the value for the current sub-word, $O(n)$ values have to be considered, which are not locally available in the worst case. Even with a specialized low-latency message-passing network, the communication overhead is then prohibitive in comparison with local memory accesses. Besides that, shared memory architectures are an attractive target, because they are widely available.

To generate portable parallel code, the compiler generates code according to the OpenMP standard [7]. OpenMP is a free-available open standard that specifies language extensions for writing parallel programs for shared memory environments in C, C++ and Fortran. OpenMP constructs are pragmas, which declare how annotated language statements, like e.g. loops, should be parallelized by the compiler. A compiler that does not support OpenMP, ignores these pragmas. The resulting program is then just single-threaded, but still yields correct results. OpenMP support is widely available in Open-Source and closed-source C/C++ compilers.

GAP-C generates code that computes the table entries diagonal after diagonal, as in the mentioned examples, to satisfy the entry dependencies. Figure 5.34(a) shows the dependencies of an entry in a $\geq O(n^3)$ single-track DP algorithm and Figure 5.34(b) shows the diagonal control flow in the table computations. The generated code does not compute on single entries of diagonals in parallel, but computes on diagonals of blocks of entries in parallel. The advantage of using blocks as smallest distribution unit is that during the computation of one or more blocks, a core could profit from caching and prefetching effects. Another advantage of this is that the synchronization overhead is reduced. The block size is configurable, but a local



(a) Table entry dependencies. The computation of (i, j) depends in general on all entries (k, k'), where $i \leq k \leq k' \leq j$.

(b) Diagonal control flow. Each color symbolizes the computation on a different processor in parallel.

Figure 5.34: Dependencies of the computation of one table entry in an $\geq O(n^3)$ single-track DP algorithm and the parallelization scheme used in the code generation by GAP-C.

search on an Opteron system showed that a block size of 32×32 is a good choice for $O(n^3)$ DP algorithm under an integer based scoring algebra. The partitioning of the blocks of a diagonal to the available processors is done for every diagonal from scratch such that the available processors are better utilized in the case of smaller diagonals.

The generated backtracing code is not parallelized since e.g. in the case of a single-track DP $O(n^3)$ algorithm, the backtracing phase is in $O(n^2)$ and profiling shows that the practical backtracing runtime in that case is just an insignificant fraction of the complete runtime.

To measure the efficiency of the described parallelization scheme that is implemented by GAP-C, the parallel speedup and efficiency of running the compiled ADPfold GAP-L version under the mfe algebra on different shared memory architectures is benchmarked. The following definitions are introduced for discussing the benchmark results.

Definition 12 (Speedup)**.** The speedup of a parallelized program is defined as

$$\mathrm{su}(n) = \frac{T_1}{T_n} \tag{5.16}$$

where T_1 is the runtime of the program on one processor and T_n is the runtime of running the program on n processors in parallel.

Disregarding caching effects, it follows that the maximal speedup of a program is:

$$\text{maxsu}(n) = \frac{T_1}{T_1/n} = n \tag{5.17}$$

Definition 13 (Parallel efficiency)**.**

$$\text{eff}(n) = \frac{\text{su}(n)}{\text{maxsu}(n)} \tag{5.18}$$

Definition 14 (Amdahl's Law)**.**

$$\text{es}(n) = \frac{1}{(1-p) + \frac{p}{n}} \tag{5.19}$$

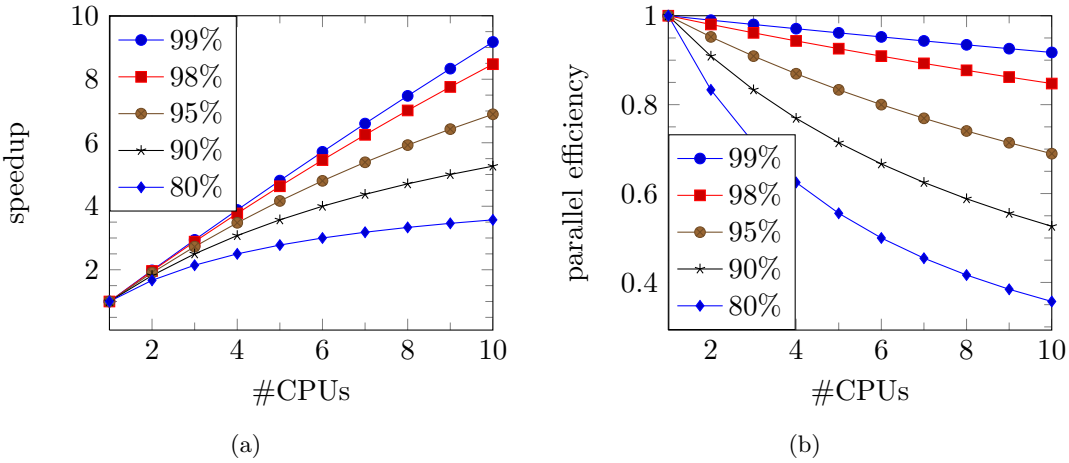
$$\leq \frac{1}{1-p} \tag{5.20}$$

$$\text{es}^*(n) = \frac{1}{(1-p) + \frac{p}{n} + \phi(n)} \tag{5.21}$$

Amdahl's law [2] defines the expected speedup (es) when parallelizing a single-threaded program where p is the fraction of the program which can be perfectly parallelized. $1 - p$ is the inherent single-threaded fraction of the original program that cannot be parallelized. es* describes the practical expected speedup, where ϕ describes cost factors that are a result of parallelization, like e.g. communication or synchronization overhead.

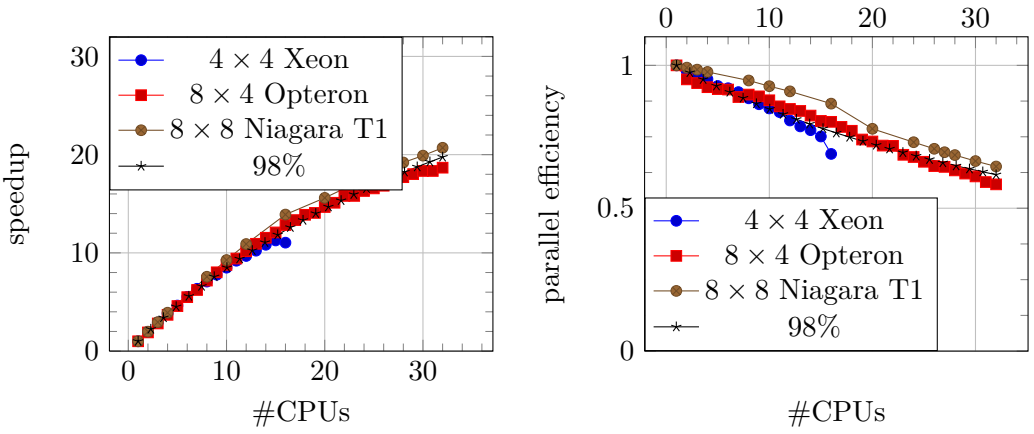
Figure 5.35 shows plots of the expected parallel speedup and efficiency for different values of p. For example, even when disregarding extra parallelization costs (ϕ), an assumed program with 95 percent perfectly parallelized code and only 5 percent inherently non-parallelizeable code, the expected parallel efficiency running it on 10 CPUs is less than 70 percent. Thus, effectively the computation time of 3 CPUs cannot be utilized through parallelization in that case.

Figure 5.36 shows the benchmark results of running the GAP-L version of the ADPfold algorithm under the mfe algebra. The ADPfold is an $O(n^3)$ single-track algorithm that is an ADP version of the RNAfold algorithm [25]. RNAfold predicts the secondary structure of RNA molecules. The input are random uniformly distributed RNA sequences of size 4000. For comparison, plots of the expected speedup (as defined in Equation 5.19) of an assumed program P with 98 percent perfectly parallelizeable code are included in the display of the results. The results show that the generated code scales well on different machines. The plots of the parallel speedup and efficiency fit the plots of the expected speedup and efficiency of the assumed program P.



(a) (b)

Figure 5.35: Plots of the parallel speedup and efficiency of different parallelizable ideal programs according to Amdahl's Law for various x percent. An x percent plot means that it represents a program of which x percent of the code is perfectly parallelizable. Communication overhead is not taken into account.



(a) (b)

Figure 5.36: Parallel speedup and efficiency of running the compiled ADPfold GAP-L program on different machines (see text). For comparison, plots of an ideal program with 98 percent perfectly parallelizable code are included. The input of ADPfold are uniformly distributed random sequences of size 4000. All machines were running Solaris 10 and Sun Workshop Pro 12 was used as C++ compiler.

5.4.3 Backtracing

In dynamic programming, backtracing denotes the process of tracing back the optimization decisions for a computed DP-table, and building a pretty-printed string representation of that path during the backtrace. A forward dynamic programming computation computes the table which is the input of the backtracing phase. Consider for example the pairwise sequence alignment algorithm. In the forward computation the edit distance is minimized for two sequences. The backtracing phase works on the distance-value table and during backtracing, the actual alignment is constructed that has the minimal computed edit distance.

Backtracing is not a concept of the ADP framework. The effect of backtracing, i.e. computing a representation of the optimal path, is implemented via products of algebras in ADP. Conceptually, different algebras for scoring and pretty-printing are specified, under which the candidates of the search space are evaluated. To compute the string representation of the optimal scored candidates, the lexicographic product (Definition 6) is used. For example:

$$\text{score} \cdot \text{pretty} \tag{5.22}$$

where score is a scoring algebra and pretty has an enumerative role.

Backtracing is then an optimization of the GAP-L compiler in the code generation. The compiler inspects the specified product and splits it into two parts that are computed in a forward and backtrace computation, if the product satisfies certain conditions. In the example the generated code for the forward computation uses just algebra score for computation and the generated backtracing code uses the pretty algebra for backtracing. The backtrace code generation phase tries to split the product into two parts

$$A \cdot B \tag{5.23}$$

where A is an algebra or an algebra product of role scoring and B is an algebra or an algebra product of role enumerative. If this is not possible, then no backtracing code is generated. Disabling the backtracing optimization yields code that has the same asymptotic runtime, but the constant factors of the practical runtime are higher. Without backtracing the computation of B is done in the forward computation, i.e. B is computed for sub-candidates, which are not part of the optimal solution and hence are not computed during backtracing. Consider e.g. a single-track $O(n^3)$ RNA folding algorithm, with the optimal backtracing phase in $O(n^2)$. Starting from table entry $(0, n)$, the backtracing traverses $O(n)$ entries above and $O(n)$ entries to the left in the worst case. At each entry a non-restricted index boundary implies a lookup of $O(n)$ values.

GAP-C supports several backtracing schemes for code-generation: optimal, co-optimal, sub-optimal and stochastic backtracing.

Figure 5.37(b) shows the pseudo-code for an optimal-backtrace code for a small grammar example (Figure 5.37(a)). Optimal backtracing means that in the situ-

```
formula = number |
  add(formula, plus,
      formula) |
  mult(formula, times,
       formula)
  # h ;
```

(a) GAP-L grammar

```
string bt_formula(i,j):
  score = formula[i,j]
  if number[i,j] == score:
    return bt_number(i,j)
  foreach k, i<k<j:
    if add(formula[i,k], plus(k, k+1),
           formula[k+1, j]) == score:
      return add_pp(bt_formula(i, k),
                    plus(k, k+1),
                    bt_formula(k+1, j))
    ...
```

(b) Optimal backtrace

```
[string] bt_formula(i,j):
  ret = []
  score = formula[i,j]
  if number[i,j] == score:
    ret.add(bt_number(i,j))
  foreach k, i<k<j:
    if add(formula[i,k], plus(k, k+1),
           formula[k+1, j]) == score:
      ls = bt_formula(i, k)
      rs = bt_formula(k+1, j)
      foreach l in ls, r in rs:
        ret.add(add_pp(l, plus(k, k+1), r))
    ...
  return ret
```

(c) Co-Optimal backtrace

Figure 5.37: An example for a GAP-L grammar and the pseudo-code for optimal and co-optimal backtracing that follows the grammar structure.

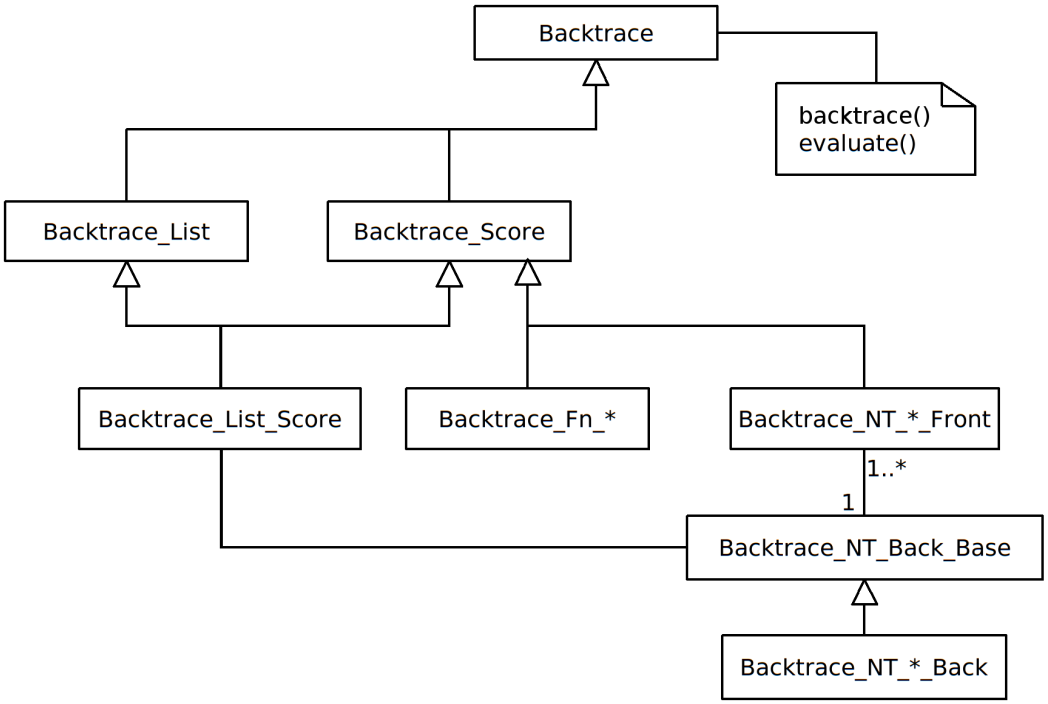


Figure 5.38: Diagram of the classes that are used for constructing the backtrace data-structure. Class names with a star represent a set of classes that are generated by the compiler. The other classes are part of the runtime library.

ations with more than one optimal sub-solution, only one is considered and the others are ignored. In co-optimal backtracing every optimal sub-solution is taken into account, such that the backtracing phase may return several co-optimal results. This is consistent with the definition of the lexicographic product operation in ADP (Definition 6). From the definition of the products objective function it directly follows that all co-optimal candidates are processed. Figure 5.37(c) shows the pseudo-code of co-optimal backtrace code for the example. Instead of choosing the first optimal split and return one optimal solution, all splits are considered and a list of optimal solutions is returned.

The generated code for co-optimal backtracing is similar to the pseudo-code examples, because it also uses a set of recursive functions. Using recursive functions has the advantage of eliminating the explicit administration of a stack during backtracing, since the function call stack is used for that. In addition, data-structures are generated that represent a backtrace.

Figure 5.38 shows the diagram of the classes used to construct a backtrace in the generated code. The class names that contain a star are generated by the compiler and the classes are part of the runtime library, because they are independent of the source program. For each function symbol of the algebra a `Backtrace_Fn_*` class

```
typedef (score, Backtrace) bt-tupel
typedef [bt-tupel] bt-list
...
bt-list ret_2 = bt_proxy_nt_formula(i, k_0);
if (is_not_empty(ret_2))
  foreach (x_0, ret_2)
    foreach (x_2, ret_4)
      bt-tupel ans = add_bt(x_0, a_1, x_2);
      push_back_min_other(answers, ans);
...
bt-list eval = h_bt(answers);
bt bt_list = execute_backtrace_k(eval);
return bt_list
```

Figure 5.39: Simplified code of the generated co-optimal backtracing code by GAP-C for the grammar example.

is generated and objects of that class represent the choice of that function during the backtracing. For each non-terminal symbol a class `Backtrace_NT_*_Front` is generated that represents a call of a non-terminal parser in the backtrace. It may reference a list of `Backtrace_NT_Back_Base` objects that represent co-optimal backtraces for sub-words. Considering the previous grammar example, Figure 5.39 shows a simplified version of the generated code that creates objects of the backtracing data-structure. The function `bt_proxy_nt_formula` generates a list of score and `Backtrace_NT_formula_Front` object tuples and the function `add_bt` returns score and `Backtrace_Fn_add` objects. The function `push_back_min_other` is an optimized list-append version that only appends, if the score is less than the top of the list, and because of this the special objective function `h_bt` is just the identity. At the end of the recursive backtrace function a call `execute_backtrack_k` triggers the `backtrace` member functions of the collected objects that continue the recursion. After the backtracing is finished and a list of backtrace path representing `Backtrace` objects is returned for the input, the `evaluate` member function is called to actually call the algebra functions of B recursively according to the constructed backtrace path.

This backtracing scheme resembles top-down parsing (Section 5.4.1). The difference is that the part B of the product is replaced by a generated backtracing algebra and the parsing is guided by the results of the forward computation. The execution of the backtracing code is delayed after the objective function execution with the help of proxy-objects of the backtracing data structure.

In suboptimal backtracing during backtracing not only the optimal scored candidates are considered, but all candidates with scores x' where $x' \leq x + \delta$ or $x' \geq x - \delta$ for minimization or maximization objective functions. Since usually the

Table 5.4: Examples of Vienna Strings and their shapes (at shape level 5).

Structure	Shape
`....(((.....)))...`	`[]`
`....(((...((....))....)))...`	`[]`
`..(((...((...))...((((...))))...)))..`	`[[][]]`

search space of a GAP-L program is of exponential size, depending on the value of δ the runtime of suboptimal backtracing is exponential. The generated code by GAP-C is similar to the co-optimal case. The difference is that the construction of score and backtrace-object tuple lists takes the δ value into account. This scheme is similar to that of RNAsubopt [57]. RNAsubopt uses an explicit stack during backtracing.

5.4.3.1 Stochastic Backtracing

In stochastic backtracing, the score component of the score and the backtrace-object tuples are interpreted as an discrete probability distribution and the backtracing objective function chooses a candidate from the candidate list at random under this distribution. Stochastic backtracing means sampling candidates from the search space under an algebra B and according to a probability distribution, defined by algebra A (Equation 5.23).

A use-case for stochastic backtracing is the situation, where computing the algebra product $C \cdot D$ is expensive, because computing C is expensive, but it is possible to compute D in polynomial time and D is a synoptic algebra that defines a probability distribution for the candidates. An alternative to directly computing $C \cdot D$ is to compute D and then use D during stochastic backtracing. Thus, an approximation of the result of $C \cdot D$ is obtained via several samplings from the search space.

An example of this is the computation of the product $shape \cdot pfunc$ for a single-track $O(n^3)$ RNA-folding algorithm like ADPfold (it is a GAP-L version of RNAfold [25]). The algebra *shape* has a classifying role and the algebra *pfunc* has a synoptic role. Algebra *pfunc* computes and sums over the partition function values of the candidates. The ADPfold grammar is semantically non-ambiguous under the Vienna String representation [25]. Each candidate from the search space has a unique Vienna String representation. A Vienna String prints a dot for an unpaired base in the input and a matched pair of brackets for a base pairing. A shape neglects small differences in Vienna Strings [22], such that e.g. all candidates from the search space with exactly one hairpin structure have different Vienna Strings, but the same shape. Table 5.4 shows examples of Vienna Strings and shape strings of various candidates. Using Boltzmann statistics [54, 55] the partition function value of the

$$l_{candidates} = [hl(\ldots), ml(\ldots, il(\ldots), \ldots), \ldots] \quad (5.27)$$

$$l_{pf} = [1, 23, 42, \ldots] \quad (5.28)$$

Figure 5.40: Conceptual list of candidate structures and their corresponding partition function values. The values define the discrete probability distribution during sampling.

search space S is defined as

$$Q = \sum_{s \in S} e^{-\beta E_s} \quad (5.24)$$

where E_s is the energy of structure s. Accordingly the partition function value of a shape X is defined as the sum over all candidate values with the same shape.

$$pf(X) = \sum_{s \in X} e^{-\beta E_s} \quad (5.25)$$

Then the shape-probability is defined as:

$$p(X) = \frac{pf(X)}{Q} \quad (5.26)$$

The computation of product $shape \cdot pfunc$ provides the shape probabilities of an input string. The search-space of the ADPfold grammar is of exponential size and the shape space size depends on the search space size. The shape abstraction reduces the search space size in comparison to the Vienna Strings space, but the number of shapes still grows exponentially with the input size [29]. Thus, computing $shape \cdot pfunc$ leads to an exponential runtime in the worst-case. Computing *pfunc* is in $O(n^3)$. Thus, using stochastic backtracing, the complete runtime is then $O(n^3 + in^2)$, where $O(n^2)$ is the worst-case runtime of one stochastic backtracing and i is the number of sampling iterations. Figure 5.40 shows an example of the correspondence of partition function values and search space candidates.

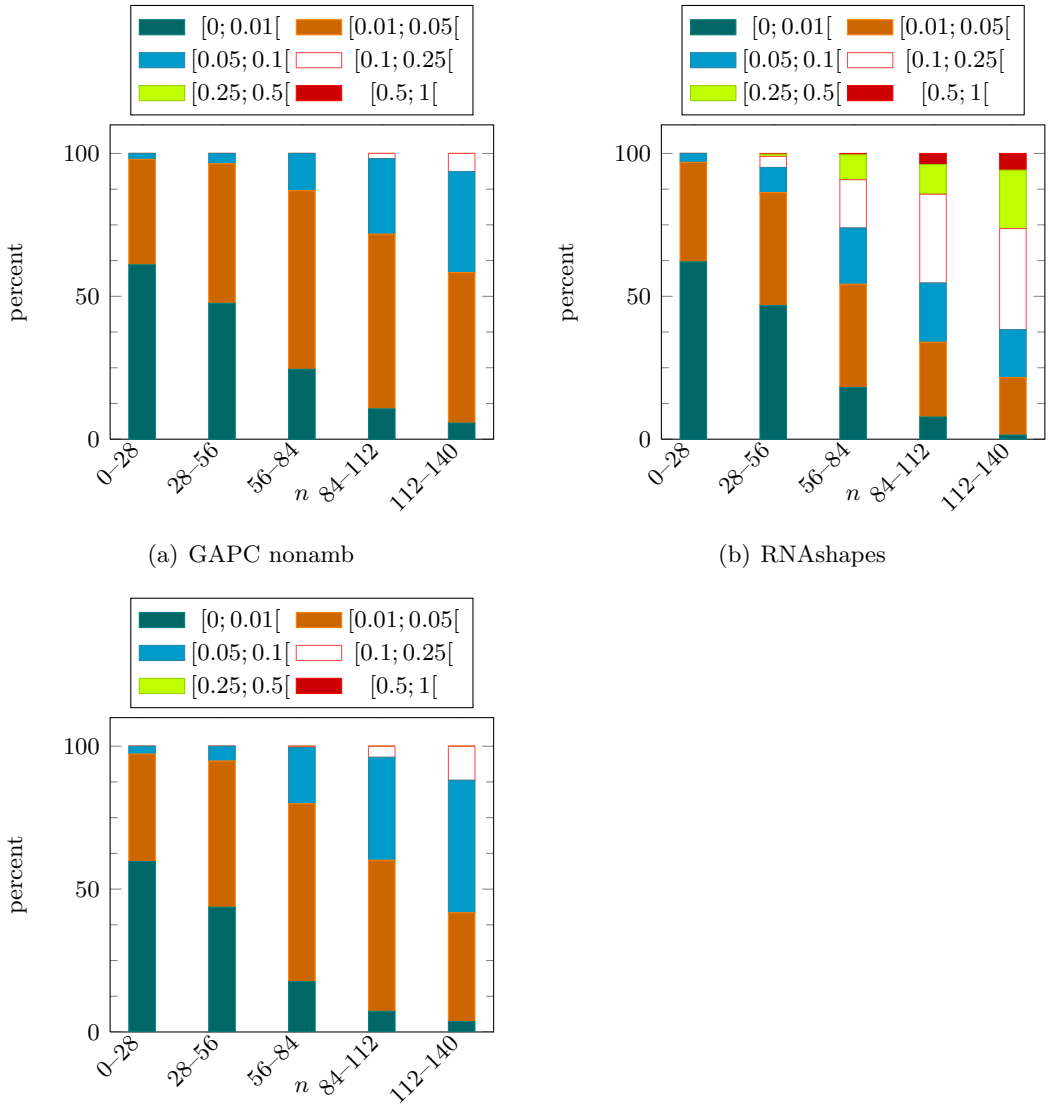
To analyze the errors of shape probabilities obtained via stochastic backtracing, the following expression is used:

$$\delta(u, x, y) = \sum_{S \in \mathcal{S}_x \cup \mathcal{S}_y} |p_x(S) - p_y(S)| \quad (5.29)$$

where u is an RNA sequence, x and y are two shape probability computation methods, $\mathcal{S}_x$ and $\mathcal{S}_y$ are the shape spaces of the two methods for sequence u,

$$0 \leq \delta(u, x, y) \leq 2 \quad (5.30)$$

and $\delta(u, x, x) = 0$. A δ of 2 means that the shape spaces of the two sequences do not share any shape.



(a) GAPC nonamb

(b) RNAshapes

(c) GAPC adpf

Figure 5.41: Distribution of shape probability deviations δ (Equation 5.29) as a function of sequence length n when comparing the exact shape probabilities with shape probabilities approximated by various programs via stochastic backtracing (see text). Each program was run with the same set of 2000 random sequences. Each δ value is collected in one of the 6 intervals shown in the legend boxes. The versions in 5.41(a) and 5.41(b) use the same ADP grammar that unambiguously takes dangling bases into account. The version in 5.41(c) implements the RNAfold grammar.

Figures 5.41(a) and 5.41(c) compare the shape-probabilities for several random sequences obtained via stochastic backtracing and via an exact forward computation. The plot shows that 1000 sampling iterations suffice in that case to produce very good approximations. [11] describes the program sfold, which is an $O(n^3)$ RNA folding algorithm that samples RNA secondary structures using Boltzmann statistics. The paper gives statistical reproducibility guarantees for sampling.

The stochastic backtracing is available in GAP-L via the use of the overlay product and an instance filter. For the previous example the right hand side of the instance declaration in GAP-L is:

```
(pfunc | pfunc_id ) * shape5 suchthat sample_filter_pf
```

The overlay product specifies that the left operand is used during the forward computation and the right operand is used during backtracing. In this case *pfunc_id* is an algebra derived from *pfunc* and the objective function is replaced by the identity function. An instance is called in the generated code after the objective function on the results of the objective function. In this case `sample_filter_pf` interprets the first components of the result tuples as points of a discrete probability distribution and chooses one tuple under this distribution at random. Doing the actual sampling in a filter and not in the objective function has the advantage of better code reuse: the filter can be plugged together with other partition function-like algebras.

In stochastic backtracing the objective function always chooses one sub-solution from the list of tuples. For the RNAshapes [22] grammar, this is not sufficient. The RNAshapes grammar is an RNA folding grammar. It is non-ambiguous under the canonical Vienna String representation of the candidates in the search space and it unambiguously takes energy contributions of dangling bases into account while computing the minimum free energy (MFE). As a consequence, the result type of the mfe algebra is a tuple of the score, possible dangling contributions and indices, because at some locations in the grammar the decision to use a dangling energy contribution is delayed to a later application of another algebra objective function. Such constructions, in combination with a unitary objective function, violate Bellman's Principle of Optimality (Definition 5), because sub-solutions with co-optimal scores may yield differently scored super-solutions, i.e. two sub-solutions with the same MFE, but with different dangling-energy contributions. In stochastic backtracing this results in approximation problems for the RNAshapes grammar and sampling $shape \cdot pfunc$. For example, the comparison of RNAshapes (Figure 5.41(b)) and the GAP-L version of the RNAfold grammar (Figure 5.41(c)) shows that the stochastic backtracing of RNAshapes produces some larger approximation errors. In the GAP-L version of the RNAshapes grammar the used sampling filter also unambiguously takes dangling-energy contributions into account. As a result, the distribution of δ values of this version (Figure 5.41(a)) is comparable to the GAP-L version of the RNAfold grammar (Figure 5.41(c)).

Using stochastic backtracing is not restricted to partition function-like algebras. It is possible to implement Stochastic Context Free Grammars (SCFGs, [4]) as

GAP-L programs. The rule probabilities are then coded in an algebra.

5.4.4 Window Mode

In window mode, the computation is done in a sliding window over the input string, where the sub-solutions of the overlap region between two iterations are reused.

For example, for a cubic runtime algorithm the space requirement is then in $O(w^2)$ and the runtime in $O(w^2 n)$, where w is the window-size and n the input length.

The code-generation of GAP-C supports the optional generation of window mode code for arbitrary single-track GAP-L programs.

5.4.5 Index Hacking

While introducing ADP, usually at some point the slogan "No subscripts, no errors!" is cited. The ADP framework avoids the use of indices with the concept of tree grammars. A tree grammar specifies the search space of a DP problem instance in a declarative fashion and the compiler derives efficient matrix recurrences from the tree grammar.

Standard RNA-folding algorithms like the Nussinov algorithm [36] and standard minimum free energy folding algorithms [25] operate under the (nested) base pairing condition. The base pairing condition says that two base pairings may not intersect: for two base pairings (u_i, u_j) and (u_k, u_l), where $i < k$ then either either $i < j < k < l$ or $i < k < l < j$, where u is the input string and $0 \leq i, j, k, l < |u|$.

For the modeling of RNA secondary structures that may include pseudo-knots, the pseudo-knot motifs violate the base pairing condition [40]. Figure 5.42 describes the basic segments of canonical simple recursive pseudo-knots which are recognized by the RNA folding algorithm pknotsRG [40]. A naive implementation of the pknotsRG algorithm would run in $O(n^8)$ because 6 moving index boundaries in the pseudo-knot need to be considered for each sub-word of the input string [40]. The algorithm reduces the number of boundaries to 2, due to canonicalization rules, while still calling 7 non-terminal parsers for each pseudo-knot segment. This results in an overall runtime of $O(n^4)$. The algorithm is implemented in the ADP framework, but with one exception. In the grammar rule which describes the pseudo-knot structure, indices are explicitly manipulated outside of the grammar.

To make the efficient implementation of pknotsRG and similar pseudo-knot-aware folding algorithms possible in Bellman's GAP, GAP-L contains constructs for index hacking (see Section 4.5.8.4). These constructs allow for a mix of explicit manual index optimizations and clean declarative grammar code, to get the best of both worlds. The usual application is the reduction of moving boundaries in the right hand side of a grammar rule for which some expert knowledge is used. In the case of moving index boundary reduction because of a constant yield size of a non-terminal parser, no index-hacking is necessary, since the compiler automatically removes the moving boundaries (see Section 5.3.12) in such cases.

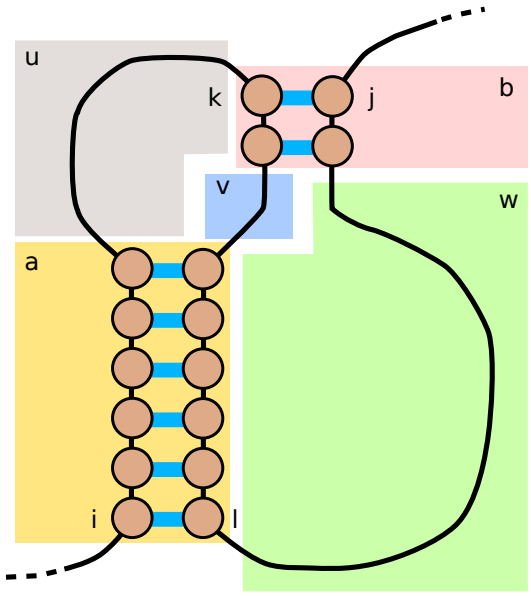


Figure 5.42: Basic segments of a canonical simple recursive pseudo-knot [39]. The boxes a, b, u, v and w mark regions of the pseudo-knot, where a and b denote helices and u, v and w denote loops. The loops may contain complex RNA structures including recursive pseudo-knots. The indices are i, k, l and j, where (i, j) denotes the sub-word which is parsed as pseudo-knot and k and l are moving index boundaries.

```
grammar pknotsRG uses Algebra(axiom = struct) {
  ...
  help_pknot_free_kl =
    .[
      int i = t_0_i; int j = t_0_j;
      if (i+11 < j) {
        for (int l = i+7; l <= j-4; l=l+1) {
          int alphamaxlen = second(stacklen(t_0_seq, i, l));
          if (alphamaxlen < 2) continue;
          for (int k = i+3; k <= l-4; k=k+1) {
            int alphareallen = min(alphamaxlen, k-i-1);
            if (alphareallen < 2) continue;
            int betamaxlen = second(stacklen(t_0_seq, k, j));
            if (betamaxlen < 2) continue;
            ...
            INNER(CODE);
          }
        }
      }
     ].
    {
      pknot(REGION, REGION, REGION) .{
        pknot(REGION[i, i+alphareallen],
        front[i+alphareallen+1, k] .(j).,
        REGION[k, k+betareallen],
        middle[k+betareallen, l-alphareallen]
           .(j-betareallen, i+alphareallen).,
        REGION[l-alphareallen, l],
        back[l, j-betareallen-2] .(i).,
        REGION[j-betareallen, j] ;
        stackenergies)
      }.
    } # hKnot;

  middle(int betaRightInner, int alphaLeftInner) =
    ... |
    middlr(BASE, mid, BASE ; betaRightInner, alphaLeftInner)
    ... # ;

  ...
}
```

Figure 5.43: Grammar rule examples from the pknotsRG GAP-L that uses index hacking and parametrized non-terminals (see text).

Figure 5.43 shows a grammar snippet from the pknotsRG GAP-L grammar that heavily uses index hacking and parametrized non-terminals (see Section 4.5.8.1). Line 5 accesses implementation details of the generated code. The nested for-loops in line 7 and 10 explicitly state how the remaining two index boundaries in a simple recursive pseudo-knot are iterated. In line 16 a pragma is used that tells the compiler to insert the generated rule code (line 23 until line 31) at that location during code output. Line 22 is only used as replacement for semantic analyses. An example of a parser call with explicit indices is line 23 (`REGION)`. In line 24 a parametrized non-terminal is called, where the non-terminal parameter is enclosed in special parentheses. The `stackenergies` parameter of the algebra function `pknot` in line 31 is separated by a semicolon because it does not result from a parser application. `middle` (line 35) is a non-terminal parametrized with two parameters that are used as additional arguments to the algebra function `middlr`.

6 Bellman's GAP Modules

Bellman's GAP Modules (GAP-M) is the runtime library for GAP-L programs that are translated with GAP-C. In this chapter design and implementation choices of important modules of GAP-M are presented. Since currently the default backend of GAP-C is generating C++ code, in the following the C++ version of GAP-M is described. The next section shows the design of memory pools, Section 6.2 describes the design of list data-structures, Section 6.3 shows the design of different string data-structures and Section 6.4 presents a module of reusable functions for RNA folding algorithms.

6.1 Memory Pools

In the generated code of GAP-L programs the use of several data-structures leads to a lot of allocations and de-allocations of fixed-size memory slices, many of which are short lived. Examples are temporary lists containing backtrace objects which are destructed at the end of the function, except the optimal one or the heavy use of small string concatenations in pretty printing algebras.

For these use-cases memory pools are advantageous. A memory pool maintains large blocks of memory and allows only fixed-size memory allocations. An allocation from a memory pool just returns a pointer into a memory block after a minimal amount of internal bookkeeping and a de-allocation induces only cheap internal bookkeeping that marks the location as reusable. The memory of the pool is de-allocated at once when the pool is not needed anymore, e.g. at program end. This concept amortizes the overhead of general purpose allocator allocations over multiple-allocations.

The memory pool implementation in GAP-M uses memory blocks of 100 MB. It obtains them from the kernel's virtual memory system, i.e. via the `mmap` syscall. A page in that virtual memory is automatically allocated by the kernel from real memory at the first write into it, where a common page size is 4 KB.

A memory block is divided into multiple entries. An entry contains space for a next pointer and an element of fixed-size. In the basic case an allocation from a block returns the next free entry and increments an index variable. A de-allocation prepends the entry into a linked list of freed entries which uses the next pointers in the freed entries. If freed entries are available, an allocation returns the last freed entry and removes it from the linked list. When a block is full, a new block is obtained and used as the primary block in the pool.

GAP-M also provides memory pools for allocations of multiples of a fixed size.

The data-structure internally uses a separate pool and does multiplexing between them for each multiplier.

Since the current output language of GAP-C is C++ and GAP-M is implemented in C++ as well, the memory pool API of the common Boost C++ library [10] would be an alternative to implementing a new memory pool solution for GAP-M. Actually, early versions of GAP-M contained a small wrapper around the Boost memory pool API (using version 1.38), but profiling showed that the usage of real memory (RSS, Resident Set Size) was large, i.e. depending on the input and the GAP-L program, up to 40 percent more memory usage is observed compared to a version only using the system default general purpose allocator (malloc). The GAP-M memory pool allocator uses RSS more efficiently, i.e. it uses less memory than a malloc based version. The runtime is the same as with the Boost memory pool.

6.2 Lists

Lists are used in the generated code during backtracing of co-optimal or sub-optimal candidates or when in the forward computation a non-unitary product is used. Dominant list operations are appending objects to a list, the joining of two lists and the copying of list objects.

Thus, the implementation of the list-data type in GAP-M uses fixed-size memory slices which are allocated from a memory pool (Section 6.1). Each slice has space for a few list elements and a pointer to a following slice. The small number of elements a slice can hold that yields good results in practice. The list objects used in the generated code are smart-pointers that reference the real list implementation objects that manage the slices. Smart-pointers are small objects that implement reference counting, i.e. they automatically destruct the referenced object if it is not referenced any more in the program. In addition to that lazy-allocation is used.

Using reference counting makes the copying around of list objects cheap, using fixed-size slices amortizes the allocation cost over multiple element append options and the next pointer inside a slice makes the joining of two large lists cheap. The use of memory improves the performance of memory allocations and de-allocations in general.

6.3 String Data Structures

GAP-M contains several optimized string modules for GAP-L programs. They are optimized for different use cases, but share the same API such that they are easily exchangeable via type-synonyms if performance profiling shows optimizing opportunities.

First, the implementation of the GAP-L `string` data-type is optimized for use in pretty-print algebras, e.g. a Vienna-String pretty printing algebra of an RNA folding algorithm. The most used operations in this use case are the concatenating

of small sub-strings and the copying string objects. It uses internally fixed-size slices of memory which are allocated from a memory pool (see Section 6.1). A slice contains a sequence of characters, references to sub-strings and repeat-codings. Using references means that if a string object is appended to another one then only a reference is appended to the destination slice and not the content is copied around. In the case of an append operation of a character that is repeated several times only the characters and the number of repeats is saved for space efficiency reasons.

The implementation of the string data-type uses a class hierarchy where the string objects, which are used in the generated code, are only smart-pointers that reference the heavy-weight objects which reference string slice objects. Besides reference counting, the implementation of the **`string`** data-type uses lazy-allocation and copy-on-write for efficiency.

Second, for classifying algebras, i.e. algebras that are used in classifying products, as e.g. algebra *shape* in $shape \cdot mfe$ (see Section 2.1.1), there is a `shape_t` data-type in GAP-L and GAP-M contains an efficient implementation of it. During classification, the construction of strings from sub-strings and copying are common operations, too. In addition to that, dominating operations are the computation of hash values and string equality testing. GAP-C generates code that uses hashtables as an optimization of classifying products. In a lot of use cases the alphabet of class strings is very small, e.g. for shapes it is of size 3, and class strings are not very long in practice.

Thus, the implementation of the shape data-type uses slices of multiples of machine-word length and 2 bits for a character. As a result it packs up to 32 characters into a single slice on a 64 bit architecture. Slices are pooled in a memory pool which is optimized for allocations of multiples of a fixed small size. If the string length does not fit into a single slice then a new large enough multi-slice is allocated from the pool and the old content is copied. String append operations are optimized using elementary find-first-set (FFS, the index of the first bit set) of the machine, when available. The implementation uses also lazy allocations to avoid costs for empty strings and reference counting.

The data-structure is parametrized with the concrete alphabet such that it is re-usable for the definition of new optimized classification data-types that need different alphabets.

Third, GAP-M provides a general purpose string data-type, which is called `rope`. It should be used for classification strings that use a larger alphabet and string algebras where the slice size of the **`string`** data-type is not large enough. The implementation of the data-type uses reference counting, lazy allocation and fix sized memory slices from a memory pool. When a rope string grows larger than the slice size then another slice is used as an extension. There is no copy-on-write and the contents of appended rope objects are directly copied. When comparing the rope data-type with the shape data-type using ADPfold and a shape algebra, the shape data-type implementation was 2 times faster.

6.4 librna

The GAP-M module `rna` provides several functions for programming RNA folding algorithms, especially for computing local energy contributions of different possible RNA secondary structure elements, like e.g. a hairpin loop or a bulge loop of different sizes and bases. The energy functions are used e.g. in MFE and partition function algebras.

In GAP-L programs the module can be used via `import rna`. It provides the following filter functions:

```
bool basepairing(Subsequence);
bool stackpairing(Subsequence);
```

And following energy functions are predefined:

```
int dl_energy(Subsequence, Subsequence);
int dr_energy(Subsequence, Subsequence);
int termaupenalty(Subsequence, Subsequence);
int sr_energy(Subsequence, Subsequence);
int hl_energy(Subsequence, Subsequence);
int bl_energy(Subsequence, Subsequence, Subsequence);
int br_energy(Subsequence, Subsequence, Subsequence);
int il_energy(Subsequence, Subsequence);
int dli_energy(Subsequence, Subsequence);
int dri_energy(Subsequence, Subsequence);
int ss_energy(Subsequence);
```

The functions work on the arguments of the built-in type `Subsequence`. A subsequence object is returned e.g. by the terminal parsers `LOC`, `BASE` and `REGION`. A sub-sequence object represents a substring (i, j) of the input string $(0, n)$, i.e. the string $s[i] \ldots s[j-1]$ of $s[0] \ldots s[n-1]$ [1].

A `stackpairing` grammar filter returns `true` if the first two bases are complementary to the last two bases of the sub-sequence. This filter is used in some RNA folding algorithms because a lonely base pairing is considered as very unlikely in nature. The naming of the energy functions follows the naming of structure elements, e.g. `dl` for *left dangle*, `bl` for *left bulge loop* and `ss` for *single stacking*. The arguments of an energy function represent part of the structure that influences the energy contribution, e.g. for `hl_energy` the first sub-sequence marks the beginning and the second sub-sequence marks the end of a hairpin loop. Figure 6.1 shows an example.

The `rna` module is a thin wrapper around the *librna* C library which is also part of GAP-M. The C API of *librna* uses C-strings and raw indices as parameters to the energy functions such that it is reusable for arbitrary RNA folding algorithms and not just for GAP-L ones. Figure 6.2 shows an excerpt from the *librna* API. In addition to the functions of the GAP-M module it provides a few specialized energy

[1] In the definition of Haskell-ADP the character indexing scheme starts from 1.

```
grammar fold uses FS(axiom = struct) {
...
  hairpin = hl(BASE, BASE, REGION with minsize(3), BASE, BASE) ;
...
}
algebra mfe implements FS(alphabet = char, comp = int)
{
...
  int hl(Subsequence lb, Subsequence f1, Subsequence x,
         Subsequence f2, Subsequence rb)
  {
    return hl_energy(f1, f2) + sr_energy(lb, rb);
  }
...
}
```

Figure 6.1: Grammar and algebra example, where a MFE algebra uses energy functions from the GAP-M module rna.

```
enum base_t { N_BASE, A_BASE, C_BASE, G_BASE, U_BASE };

typedef unsigned int rsize;

int hl_energy_stem(const char *s, rsize i, rsize j, rsize n);

int sr_energy(const char *s, rsize i, rsize j);

...
```

Figure 6.2: Excerpt of the *librna* C API.

functions and functions needed for partition function computations. The indexing scheme is the same as for the GAP-M module. Internally, the energy functions use the energy tables which are distributed with the Vienna RNA package [25].

7 Bellman's GAP Pages

Bellman's GAP Pages is an interactive web-site for presenting GAP-L by examples. A list of example GAP-L versions of textbook dynamic programming algorithms, like e.g. optimal matrix chain execution, pairwise sequence alignment variants or palindromic RNA secondary structure like folding variants, are available. For each program a page presents a web-form. It includes a description of the algorithm, a link to the source code and input fields. A user can enter input sequences and construct an algebra product from multiple drop-down boxes. The resulting program is executed on the server and the results are shown to the user. Figure 7.1 shows a screenshot of an example.

The purpose of GAP Pages is to provide a platform, where practical aspects of GAP-L and ADP can be studied without the need to install the full compiler on a local machine. The accessible algebra product selections should inspire users to experiment with different algebra products. In addition, the presented GAP-L versions of well known dynamic programming algorithms may show how to use certain GAP-L constructs in practice and act as a starting point for own developments.

Behind the scenes, a server process generates all possible algebra products and pre-compiles them with GAP-C. This speeds up the web interface and reduces the load on the web server. A part of the GAP Pages concept is to inspect the user selection of algebras and input sequence and to issue a warning if a product with exponential runtime and an input length above a threshold is entered. For certain products, this could be implemented in the server software by executing a modified product which includes the count algebra, before the entered algebra product is executed. The count algebra counts the candidate search space of the algebra and GAP-C supports the automatic generation of a count-algebra for every GAP-L grammar (Section 4.5.5). If a run with the count algebra returns a number above a threshold, an appropriate error message is presented. For example, for a product that starts with an enumerative algebra, the list of answers is of the size of the search space.

Apart from this, the server software has to monitor the resource use of the compiled GAP-L programs. For example, an input sequence could induce a lot of co-optimal candidates, which have to be sent back to the user of the web-page. After a reasonable threshold of output lines and runtime the server software should truncate the output and display a helpful explanatory warning message.

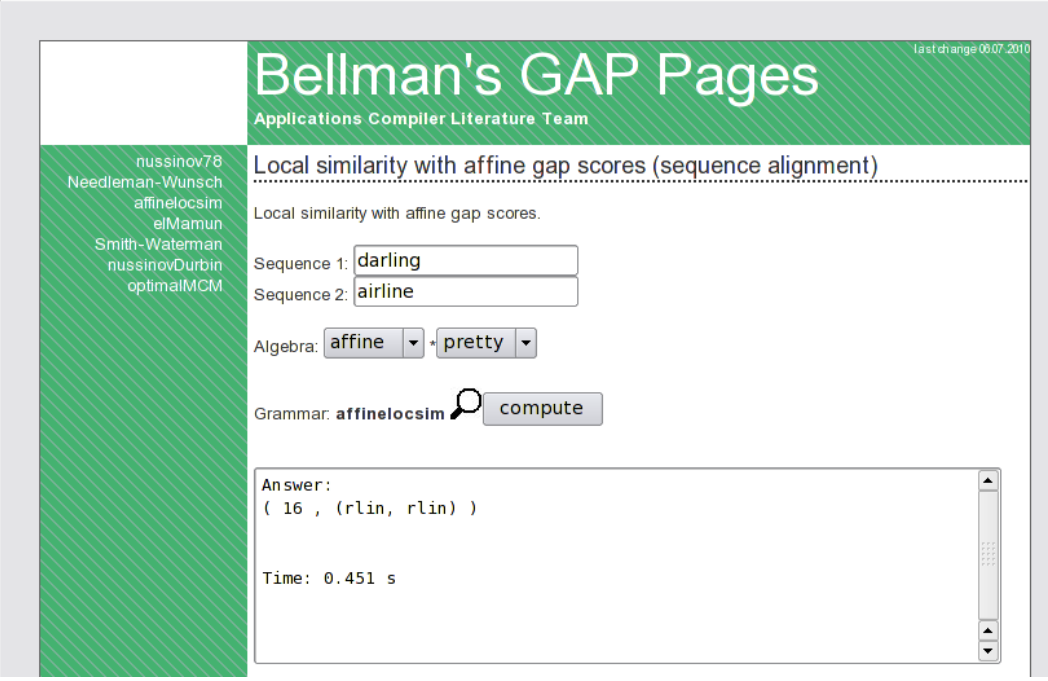


Figure 7.1: Screenshot of GAP Pages. As an example GAP-L program the local sequence alignment algorithm is shown.

7.1 BiBiServ

The GAP Pages should be integrated into the BiBiServ [45]. The BiBiServ is a web-site that provides several bioinformatic tools as web-services and web-forms. An example set of applications on the BiBiServ is RNA Studio [46]. The BiBiServ exists since 1996 and is actively maintained and improved. Integrating a web-version of a tool into the BiBiServ guarantees a stable internet address and stable support of the underlying software and hardware infrastructure. Also, generic improvements of the BiBiServ framework, like e.g. machine readable tool descriptions, come for free. Basic first-level support questions are filtered out and answered by the BiBiServ team. Only higher level issues are forwarded to the tool authors.

Since the BiBiServ is written in Java and makes heavy use of Java application server technologies, GAP Pages is implemented in Java for better integration. The web interface is implemented using the Java Server Faces (JSF 2.0) API. This makes it easy to integrate dynamic changes in the web-forms, like for example displaying the available output, without a complete reload of the page. A disadvantage of using Java is that it does not provide an API for the monitoring of called external programs. Also, executing external programs from Java has a noticeable runtime overhead[1].

[1] 0.2 to 0.5 seconds under Solaris 10 using Oracle Java

8 Benchmarks

To test the practical overall efficiency of the code generated by GAP-C, several GAP-L programs of different sizes are benchmarked in the following. The implemented algorithms are well-known bioinformatics RNA folding algorithms. For comparison, the GAP-L version is benchmarked against the original implementation, a Haskell-ADP version and an ADPC-ADP version, where available.

Benchmark results of the generated parallelized code are presented in Section 5.4.2, benchmarks comparing top-down-style vs. bottom-up-style code generation are shown in Section 5.4.1.3 and efficiency results running the GAP-L program with table configuration computed by the heuristic table design algorithm are presented in Section 5.3.7.3. The errors during stochastic backtracing of shape strings under a partition function algebra in comparison with the exact computation are shown in Section 5.4.3.1.

All benchmarks in this chapter were run under a Debian Linux 5.0 system with default package versions, especially GNU GCC 4.3.2 for compiling C and C++ code and GHC 6.8.2 for compiling Haskell-ADP code. Haskell-ADP code was compiled with optimization flags `-O2` and C/C++ code with `-O3`, unless the default was set to `-O2`. The GNU GCC and the GHC produced 64 bit code. The table configurations of the GAP-L versions were automatically derived by the GAP-C. The hardware consists of a AMD Athlon 64 X2 Dual Core 5200+ (2.6 GHz, cache size of 2 times 1 MB) and 4 GB RAM main memory. The benchmark runs were all single-threaded.

In each benchmark, a sequence of randomly generated RNA sequences were used as inputs. The sequences were randomly generated under a uniform distribution of lengths and nucleotides. In the comparison of different implementations of an algorithm every implementation was called with the same random sequence of sequences.

Runtimes of less then one second and runtimes over a threshold of 10 minutes are excluded from the plots, unless otherwise stated. Likewise data are excluded from programs, where the memory usage hits the threshold of the GHC runtime garbage collection. The measured memory sizes are the high-water marks of the actually used memory (Resident Set Size – RSS).

Note that the maximal sequence length used in the benchmarks is chosen to show the principal scaling behavior of the generated DP code and does not necessarily mean that a concrete program is usually used in practice with that large sequences. For example, in biology practice computing the secondary structure with the minimum free energy (MFE) for an RNA sequence becomes less useful for larger sequences greater than 400 bases because of accumulating errors. In general, depending on the sort of RNA sequences under examination, the sequence lengths

vary, e.g. sequences of up to 2000 bases may be considered while searching for target sites, and other classes of RNA only contain very small sequences, e.g. up to a few 100 bases.

8.1 RNAfold

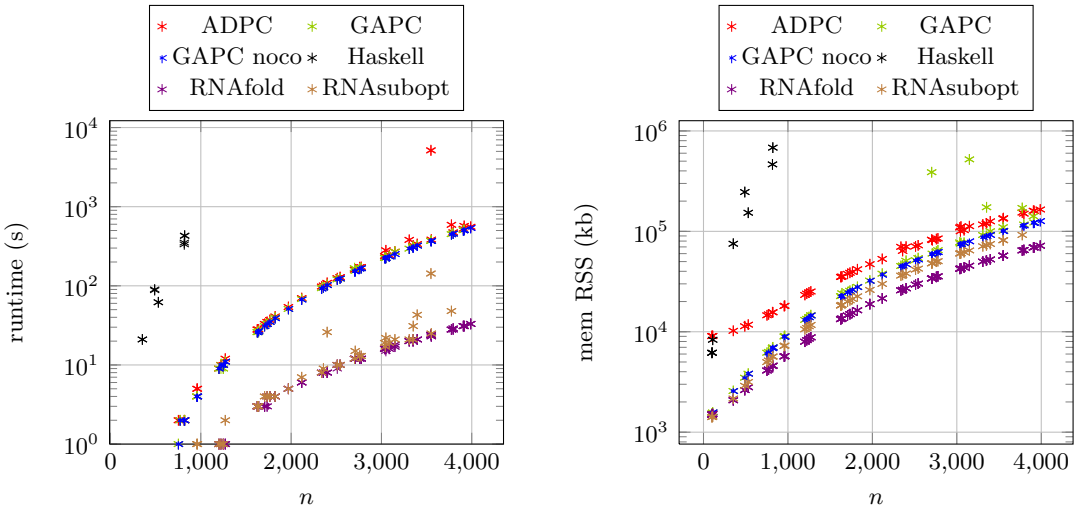
RNAfold[25] computes the optimal secondary structure of an RNA input sequence under a minimum free energy (MFE) model. It is a $O(n^3)$ algorithm which is manually implemented in C. The underlying matrix recurrences describe semantic ambiguity free under the canonical Vienna String candidate notation the search space of candidate structures. Semantic non-ambiguity, as defined in [19], means that each candidate has a unique Vienna String. Jens Reeder translated the RNAfold recurrences (assuming dangling mode `-d2` and forbidding lonely base pairings `-noLP`) into an ADP grammar and an mfe algebra in Haskell-ADP. This version was translated to ADPC-ADP and to GAP-L. It is available as example grammar both with the ADPC and GAP-C.

RNAfold does not print co-optimal candidates; this is possible with RNAsubopt [57]. Besides co-optimal backtracing it supports suboptimal backtracing.

As benchmark, the runtimes and memory usages of RNAfold, RNAsubopt, a Haskell-ADP version, an ADPC-ADPC version of the RNAfold grammar and two GAP-L versions of the RNAfold grammar are compared. RNAfold was run with the options `-d2 -noLP`, RNAsubopt was run with `-e0 -d 2 -noLP` (only co-optimal candidates) and the ADPC-ADP version was compiled with ADPC 0.8 and run with `-e0` (only co-optimal candidates). One GAP-L version was compiled with co-optimal backtracing enabled and the other with single-optimal backtracing. The four ADP based versions used the algebra product $mfe \cdot pretty$, where the *pretty* algebra generates a Vienna String representation of a candidate. For benchmarking 50 randomly generated sequences were used in the length interval of $[100; 4000]$.

Figure 8.1 shows the runtime and memory results for the different RNAfold version. The runtimes of RNAfold and RNAsubopt are mostly close together and for larger input length mostly 16 times faster than the ADPC and GAPC versions. The runtime of RNAsubopt increases up to 7 times in comparison to RNAfold in cases where a lot of co-optimal candidates exist and are backtraced. The runtime of the Haskell-ADP version is 200 times of the runtime of the GAP-L version and more for small sequences under 1000 nucleotides. In these range the runtimes of the non-Haskell versions are mostly under 1 second and not plotted. For larger sequences the Haskell-ADP version is out of memory.

The runtime of the ADPC-ADP version usually is close to the runtime of the GAP-C versions. For some sequences there are some dramatic deviations. The output of the ADPC-ADP version indicates some problems with the backtracing. For some inputs during backtracing there are a lot of candidates which are printed several times. For the complete benchmark run, every candidate is printed 30 times during backtracing on average. Since the grammar is semantically unambiguous,



(a) RNAfold runtime

(b) RNAfold memory

Figure 8.1: Runtime and memory usage of various versions of the basic RNAfold algorithm as a function of input length n.

this looks like a bug in the backtracing code generated by the ADPC.

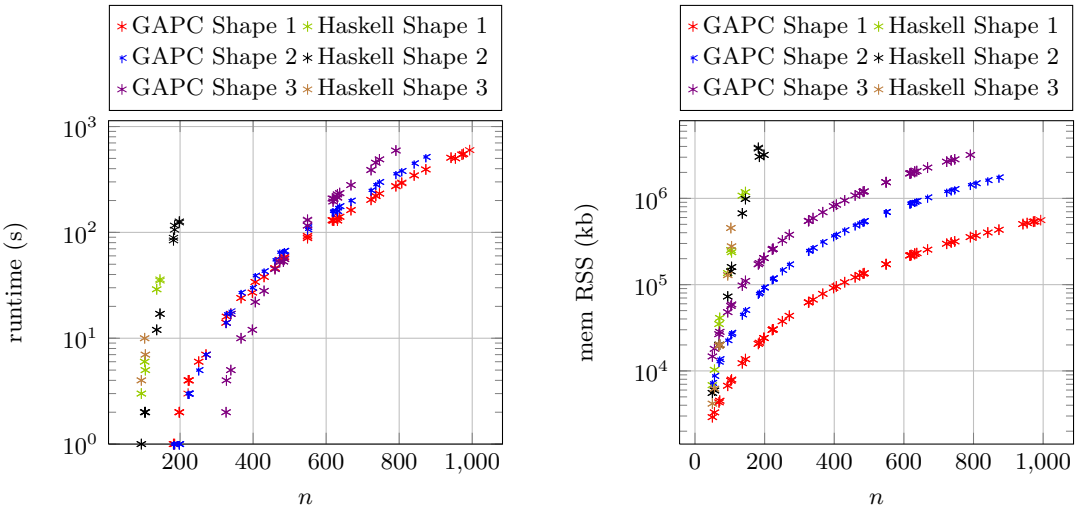
The memory usage of the GAP-L versions is usually smaller than the memory usage of the ADPC-ADP version. Only the GAP-L version with co-optimal backtracing has larger memory usages for some sequences that have a lot of co-optimal candidates, because the co-optimal backtracing code generated by GAP-C collects all co-optimal backtraces in a data-structure, before printing them. This is useful, if the generated code is interfaced by external code that inspects the backtraces. RNAsubopt directly prints finished backtraces and frees their memory.

The runtimes of the GAP-L co-backtracing and non-co-backtracing versions do not differ much. This is quite different from the runtime comparison of RNAfold and RNAsubopt.

The speedup of the manually in C implemented RNAfold versions compared to the compiled ADP versions shows that they are highly optimized. In that case the use of a compiler means paying an abstraction penalty, because the compiler does not recognize all chances of MFE-dependent low-level optimization, as a human programmer does. On the other hand, one can argue that the development time of a bug-free and sufficiently efficient version using GAP-L is greatly reduced in comparison to implementing the recurrences by hand.

8.2 Thermodynamic matchers

RapidShapes [27] is a bioinformatics tools that computes exact shape probabilities of input sequences using thermodynamic matchers (TDMs). In a preprocessing



(a) TDM runtime

(b) TDM memory

Figure 8.2: Benchmark results of TDMs, i.e. runtime and memory RSS as a function of input length n for three different shapes (43 non-terminals, 95 non-terminals and 195 non-terminals). Each TDM was created as Haskell-ADP and GAP-L version.

phase it approximates shape probabilities under an RNAshapes grammar. Heuristically the highest scored shapes are selected and for each shape a TDM is generated and run. In that case a TDM is an ADP grammar that describes the RNA structure folding space of exactly one given shape and a generic partition function algebra. The runtime of each TDM is in $O(n^3)$. To compute the exact shape probability of a given shape, the partition function value using the corresponding TDM and the partition function value, using the generic, all-shapes-allowing RNAshapes grammar are computed under the partition function algebra.

RapidShapes supports the generation of TDMs as Haskell-ADP or GAP-L programs. In practice RapidShapes generates GAP-L programs.

In the benchmark, the runtime and memory usage of the Haskell-ADP and the GAP-L versions were compared for three TDMs, which match various shapes. The first TDM grammar contains 43 non-terminals, the second 95 non-terminals, and the third 195 non-terminals. Each program was called for each input from a set of 50 randomly generated input sequences of length 50 to 1000.

Figure 8.2 shows the results. For all shapes the Haskell-ADP TDM versions are out-of-memory for short sequences of size greater than 200 bases. The runtime of the Haskell-ADP versions are up to 200 times of the runtime of the GAP-L versions. The results show that compiling the TDM-grammars via GAP-C just enables their use in practice. Without the compilation to efficient code the RapidShapes approach would not scale.

The generation of TDMs has a manageable level of complexity at the level of tree grammars and algebras. One can imagine that the complexity level of TDM generation increases at the lower level of matrix recurrences. Writing and debugging a TDM generator outside of ADP is expected to significantly increase the development time.

8.3 RNAshapes

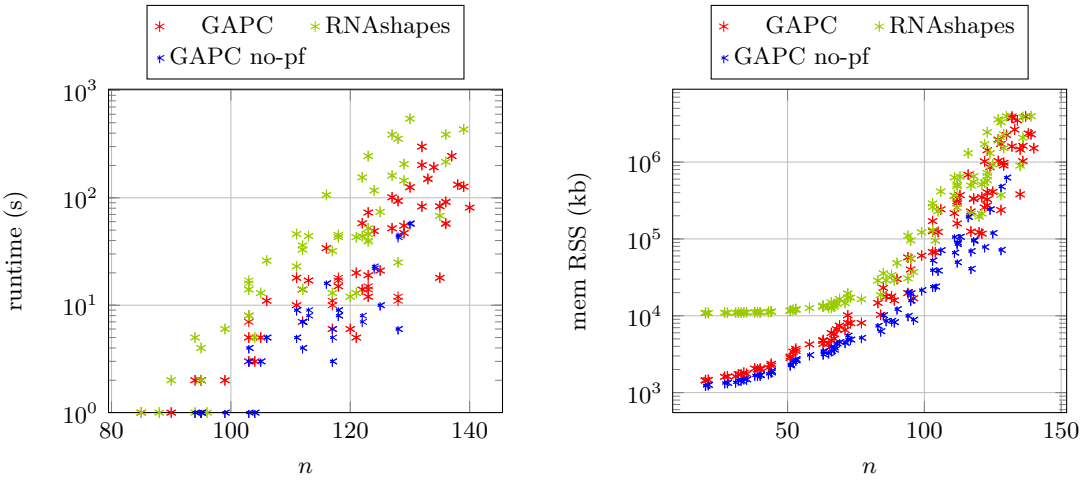
RNAshapes [55] provides several shape related optimization algorithms. For benchmarking the exact computation of shape probabilities is selected. In GAP-L this is the computation of the algebra product *shape* · *pfunc*. For each shape of the shape search space the partition function is computed. This task is challenging, because the number of shapes grows exponentially with the length of the input (see Section 5.4.3.1 for a discussion). The runtime of the algorithm is in $O(\alpha^n n^3)$, where $\alpha > 1$. The programs have to use efficient data structures to store and access a large number of shape classes. See Section 6.3 for the description of the shape data type. For classifying products, GAP-C generates optimized code that uses hashtables. An efficient hashtable implementation tailored for GAP-L programs is part of GAP-M.

In this benchmark the RNAshapes version 2.1.6 was used. It was run with the options `-p -F 0`, which disables the heuristic filtering of shape classes with low probabilities during the computation. By default, RNAshapes uses the shape level 5, and this is what the GAP-L versions used. A GAP-L version of the RNAshapes grammar was ran under the algebra product *shape* · *pfunc*. To show the impact of the partition function computation a second GAP-L version that computes just the algebra *shape* was run. Each version was called for 100 randomly generated sequences in the length interval of $]20; 140[$.

Figure 8.3 shows the benchmark results. The runtime of the GAP-L version is from 2 to 6 times faster than the runtime of RNAshapes, depending on the input sequence. The GAP-L version is more memory efficient than RNAshapes. Some larger sequences could not be computed with RNAshapes on that computer, because not enough memory was available. With the GAP-L version the exact shape probabilities of these sequences could be computed within the 4 GB of RAM. Depending on the input sequence the memory usage of RNAshapes was 2 to 10 times that of the GAP-L version.

8.4 pknotsRG

pknotsRG [40] is an RNA secondary structure prediction program that also allows a restricted class of pseudo-knots in its structure search space. The runtime of the algorithm is in $O(n^4)$. The algorithm was developed with ADP with some index access extensions to allow for pseudo-knot structures (see Section 5.4.5 for discussion and the GAP-L syntax). In the benchmark the product *mfe* · *pretty* is computed.



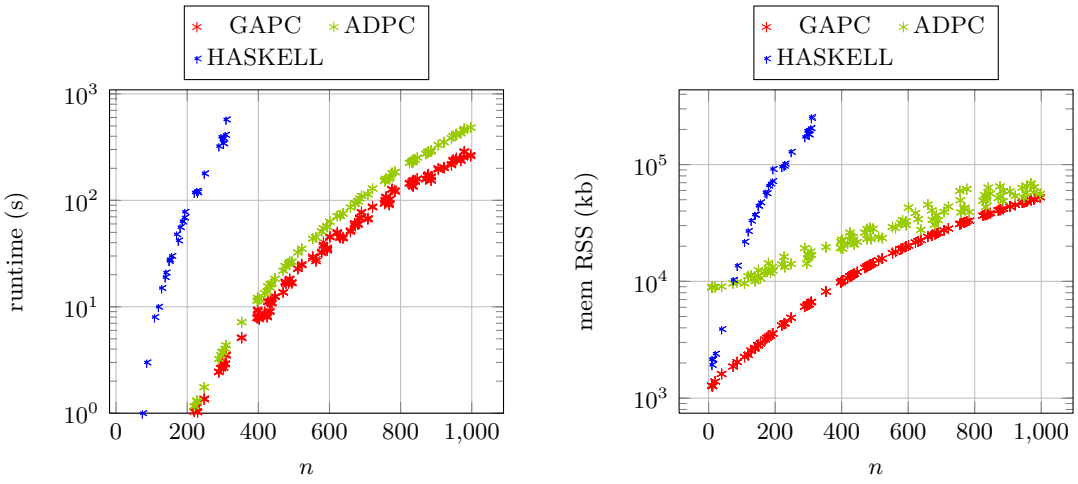
(a) RNAshapes runtime

(b) RNAshapes memory

Figure 8.3: Benchmark of the *shape* · *pfunc* classification algebra product in combination with the RNAshapes grammar, i.e. runtime and memory RSS as a function of input length n for the RNAshapes program and a GAP-L version.

The benchmark compares the runtime and the memory usage of the original Haskell-ADP version of the pknotsRG algorithm, the pknotsRG program version 1.3 and a GAP-L version of the pknotsRG grammar and algebra. The pknotsRG program contains C code that was generated by the ADPC compiler. The GAP-L version and the pknotsRG program both do backtracing while ignoring co-optimal candidates. The pknotsRG program provides suboptimal backtracing, which was turned off during benchmarking. All versions were run with the same set of 100 randomly generated sequences from the length interval of $]10; 1000[$.

Figure 8.4 shows the results of the benchmark runs. The runtime of the Haskell-ADP version is up to 100 times the runtime of the other versions. For sequences larger than 250 bases the Haskell-Version is out of memory. The GAP-L version has a runtime speedup of 2 to 3 times compared to the pknotsRG program. The memory usage of the pknotsRG program is 2 times that of the GAP-L version or more for most sequences. Only for a few larger sequences the memory usage of the pknotsRG program is less than 2 times that of the memory usage of the GAP-L version.



(a) pknotsRG runtime

(b) pknotsRG mem

Figure 8.4: Benchmark results of the pknotsRG program, i.e. runtime and memory RSS as a function of input length n a for a Haskell-ADP version, a ADPC version and a GAP-L version of the pknotsRG grammar. Input were 100 random generated sequences.

9 Conclusion

Bellman's GAP simplifies the development of dynamic programming algorithms over sequences.

Especially for large scale DP algorithms Bellman's GAP reduces the development time. An example is the tool RapidShapes that generates GAP-L grammars with up to a few hundred non-terminals. At the level of tree grammars the complexity of code-generation in RapidShapes is manageable in comparison to low-level matrix recurrences (see Section 8.2).

Benchmarks show that the novel heuristic Table Design algorithm for deriving a good table configuration that takes constant factors into account yields good results in practice. In the tested cases the table configuration heuristically derived leads to a better practical runtime than the domain expert choices or expert systems. The recent RapidShapes grammar generator by default uses the automatic table design of GAP-C because of these results and this design choice simplifies the generating process (Section 5.3.7.3).

The novel domain specific language for ADP GAP-L supports general multi-track DP, lexicographic products from ground up and the new cartesian and interleaved product. It generalizes the concept of simple syntactic grammar filtering and introduces semantic filtering in grammar and instance constructs. Advanced DP features, like e.g. sampling or filtering of candidates, are available via orthogonal language constructs, such that the need of error-prone and tedious manual hacking of these features is eliminated. The use of established language concepts and syntax elements from widely-used languages like Java and C makes GAP-L more easily accessible to new ADP users (Chapter 4).

The novel GAP-C compiler which translates GAP-L programs to optimized C++ code, provides state of the art error and warning support during parsing and a type-checker for better usability. Benchmark results show that the generated code scales well on parallel shared memory architectures (Section 5.4.2). The code-generation for multi-track GAP-L programs profits from an improved table dimension analysis and CYK-loop optimization. Performance optimizations for the single-track case are generalized for the multi-track case such that multi-track optimizations, like e.g. in the pairwise-sequence alignment example, are obtained as a side effect (Section 5.3.6 and Section 5.4.1.6). The generated code is dependable, as the test results of sampling and filtering code show and the errors are less than in compared non-GAP-L program versions (Section 5.4.3.1 and Section 4.6.3). The overall runtime and memory usage performance of the generated code by GAP-C in cases of ADP programs that are also compilable with the ADPC is equal or several times better than the results from ADPC versions. Comparing the performance to Haskell-

ADP versions, the GAP-C versions show huge speedups and in several cases the Haskell-ADP version runs out of space or show a prohibitively long runtime for relatively small inputs. Thus the availability of GAP-C enables the practical usage or improves the practical value of ADP algorithms in several cases (Chapter 8).

During the study of alternative evaluating schemes in generated yield parsing code a top-down scheme shows advantages for a certain kind of products and grammars (Section 5.4.1). In cases where sparseness can be exploited and using substantial products twofold runtime and memory improvements are possible. Top-down parsing is available in GAP-C in addition to a bottom-up evaluation scheme.

In Section 2.2.3 two classification schemes for algebras and products are introduced. Their benefit is twofold. They allow to establish and discuss product properties on a general level, like e.g. the lexicographic product of two unary selective algebras satisfies Bellman's Principle, and GAP-C uses the roles internally for optimization and warning-generation purposes.

The description of selected modules of GAP-M, the runtime library of GAP-C shows that using high-level abstract data-types in the code-generation of the compiler allows for greater flexibility in optimizing the concrete implementation for a certain output language and reduces the complexity of the compiler (Section 5.1 and Chapter 6).

10 Outlook

There are several open topics in the ADP framework and in the compilation of ADP regarding the exploitation of further generalization and optimization possibilities.

10.1 Sparse ADP

A sparse DP algorithm is a functionally equivalent variant of an existing DP algorithm that systematically omits parts of the search space due to constraints implied by the problem domain. The process to modify an existing DP algorithm such that sparse properties of the problem space are exploited is called sparsification. Depending on the problem a sparse algorithm version may achieve an asymptotically improved runtime in the average-case in comparison to the non-sparse one. There are different kinds of sparseness that can be exploited in dynamic programming.

A form of sparseness is introduced by the data-flow of the program. For example consider a GAP-L program, where in a top-down evaluation scheme filtering constructs in the grammar or implicit yield size constraints may prune large parts of the search-space depending on the input (see Section 5.4.1.3).

Another form of sparseness can be exploited doing a pre-processing of the input as described in [14, 15, 16], which includes several examples of sparsified DP algorithms. For example, in ADPfold, the GAP-L version of the standard $O(n^3)$ MFE RNA folding algorithm, there exist several right hand sides of non-terminals that induce a moving index boundary in the generated code. These moving index boundaries are responsible for one $O(n)$ factor of the overall runtime. Depending on the structure of the right hand side, the number of considered boundaries can be reduced in practice. Figure 10.1 shows two example rules. Since non-terminal `closed` is guarded by a filter, only those index boundaries need to be considered where the filter result at the referenced non-terminal is true. In an $O(n^2)$ time and space pre-processing of the input sequence a data-structure can be constructed that contains all successful boundaries for each (i, j). Thus, when evaluating the moving index boundaries the data-structure is used to iterate only over a restricted set of boundaries. Such a modification is not heuristic, and the algorithm is still guaranteed to return the same optimal results as the original algorithm.

A more aggressive pre-processing can lead to more sparseness and an approximative version of the original algorithm. For example, in [39] sparse versions of pknotsRG are presented, where an inner loop is replaced by a loop that iterates only over index boundaries that induce base pairings with a base pair probability above a certain threshold. The runtime of the original pknotsRG algorithm is in $O(n^4)$,

```
struct = sadd(BASE, struct)   |
         cadd(dangle, struct) |
         nil(EMPTY) # h ;

dangle = dlr(LOC, closed, LOC) ;

closed = { stack | hairpin | leftB | rightB | iloop |
           multiloop }
         with stackpairing # h ;
```

Figure 10.1: Example of a source of sparseness in a GAP-L grammar. The rules are part of the ADPfold grammar. The moving index boundary of the second alternative of non-terminal `struct` only needs to consider those boundaries where the `stackpairing` filter is true.

the computation of all base pair probabilities is in $O(n^3)$ and the approximative pknotsRG version is in $O(n^3)$.

Besides pre-processing, additional sparseness can be exploited using additional bookkeeping during runtime of the algorithm. In [56] a sparsified version of the RNAfold algorithm is presented that has an average case runtime of $O(n^2\psi(n))$ where ψ converges against a constant. The basic matrix recurrences are arranged in such a way that the triangle inequality holds for the computed values. This property is used in the sparse version to skip the iteration of index boundaries that cannot improve the result, i.e. it is iterated in the inner loop only over a constant number of boundaries. pknotsRG was sparsified using a similar technique [34] and [3] concentrates on the space usage improvements in RNA folding exploiting sparseness.

Thus, reviewing the state of sparseness in DP, general support for automatic sparsification of GAP-L programs in GAP-C is a promising research topic. A sparsification analysis in GAP-C could analyze the used product and identify certain score algebras, if conditions needed for different kind of sparseness exploitations are satisfied. Alternatively, a user could manually mark a scoring algebra as satisfying such conditions, if complicated scoring functions are used. When analysing the grammar, moving index boundaries that reference guarded non-terminals could be automatically identified and pre-processing sparseness optimization could be generalized for different kind of filters and implicit yield size constraints. The code-generation could use the result of sparseness related analyses to generate specialized sparseness exploiting target code. Ideally, in the development of a new DP algorithm in GAP-L an automatic sparsification feature in GAP-C would liberate the GAP-L programmer from manually implementing low-level sparsification versions. Instead only a recompilation would be needed, similar to the automatic table design feature, which liberates the developer from tedious manual tabulation concerns.

10.2 Knapsack style DP algorithms

ADP is a formal framework to develop dynamic programming algorithms over sequences. It started with single-track support and Bellman's GAP implementation of ADP generalizes it to include full multi-track support. Looking at textbook style dynamic programming algorithms on sequences, the Knapsack algorithm [8] presents a challenge to implement it in the ADP framework.

The Knapsack algorithm runs in pseudo-poly time and solves the optimization problem that: with a set of objects and their weights w_i and values m_i the most valuable subset which fits into a weight restricted backpack ($\leq w_{max}$) is computed. Equation 10.1 specifies the matrix recurrences of this algorithm.

$$K_{i,j} = \max_{snd} \begin{cases} K_{i,j-1} & \text{if} \quad j > 0 \\ (w_j, m_j) & \text{if} \quad i \geq w_j \\ K_{i-w_j,j-1} + (w_j, m_j) & \text{if} \quad i \geq w_j, j > 0 \\ (0,0) \end{cases} \tag{10.1}$$

To transform the matrix recurrences into GAP-L we model it as a two-track GAP-L program. The first track is the string u of increasing weighs, with $u = 0 \ldots w_{max}$, and the second track is the string v of weight and value tuples, with $v = (w_1, m_1) \ldots (w_n, m_n)$. Grammar `sack` is a first draft of the search space description:

```
grammar sack uses Bill (axiom=knap) {

  knap = ins(sack, <WEIGHT, CHAR>) |
         start( <WEIGHT, CHAR>) |
         skip( <EMPTY, CHAR>) # h ;
}
```

An imaginative terminal parser `WEIGHT` takes care to parse as much of u as the current item consumed by the CHAR terminal parser of the second track weights. Such a `WEIGHT` terminal parser is currently not possible in GAP-L since there is no language concept of user-defined terminal parsers and terminal parsers cannot interact between different tracks of a multi-track program. The GAP-C has no concept of a terminal parser that has a variable yield size which is constant for a sub-word of the other track which would lead to an additional unneeded inner loop in the generated target code.

The problem with the implementation of the Knapsack and similar style algorithms in ADP is that a key concept of ADP is the separation of the search space description (tree grammar) and the evaluation of candidates (algebra). In the Knapsack algorithm these aspects are inherently intermixed.

One solution is to extend GAP-L to allow for the definition of new terminal parsers. The new user-defined terminal parser definition syntax has to provide the possibility to define a multi-track terminal parser and to specify the yield size of one track in dependence of the other track. In particular, the syntax has to allow

for the possibility to indicate that the yield size of the first track is variable, but constant for each sub-word of the second track. This is the precondition for a future semantic analysis extension of GAP-C which would automatically eliminate the moving boundary on the first track of the `ins` alternative of the `knap` grammar rule.

Thus, the extension of GAP-L and GAP-C in the drafted way is feasible to allow the implementation of the class of Knapsack-like algorithms from fields like operational research.

10.3 ADP over Trees

The ADP framework was developed for dynamic programming over one input string and in GAP-L it is generalized for multiple input strings (Multi-Track DP). However, the method of dynamic programming is not restricted to string inputs. Another class of dynamic programming algorithms are DP algorithms over trees.

An example is RNAforester [26], which computes the optimal tree alignment between two trees or forests. In the description of the algorithm there are also special purpose combinators defined in Haskell which are used for a Haskell version. For efficiency reasons the RNAforester program is manually implemented in C++.

Generalizing ADP to tree and forest inputs would simplify the development of new dynamic programming algorithms over trees, e.g. an RNAforester variant that supports affine gap-costs or other modifications. However, the design of ADP over trees provides several challenges.

What is the appropriate grammar device for search space description? How to generalize the tree-grammar concept from single sequence ADP? The yield of a candidate tree should be the input tree or forest. An ASCII representation of such a grammar formalism has to be sufficiental practical to be useful for programming.

What are useful pattern matching mechanisms in a grammar that are powerful, easy to use and manageable to optimize in a compiler?

Another challenge is the research of other tree DP algorithms, to design ADP over trees in such a way that it is not only applicable to tree alignment problems. Analogous to the pairwise sequence alignment problem, where the standard $O(n^2)$ space algorithm is already an optimisation of the general $O(n^4)$ two-track scheme (see Section 5.4.1.6), one could study the general structure of two-track tree DP algorithms. Also, it would be instructive to study "single-track" tree DP algorithms, i.e. algorithms that take only one tree or forest as input, for designing ADP over trees.

In general, ADP over trees provides several open language design questions and several opportunities to develop novel compiler optimizations.

Bibliography

[1] Mohamed I. Abouelhoda, Robert Giegerich, Behshad Behzadi, and Jean-Marc Steyaert. Alignment of Minisatellite Maps: A Minimum Spanning Tree based Approach. In *Asia Pacific Bioinformatics Conference Kyoto, Japan, 14-17 January 2008*, 2008. 4.6.4

[2] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483—485, 1967. Available from: `http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf`. 14

[3] Rolf Backofen, Dekel Tsur, Shay Zakov, and Michal Ziv-Ukelson. Sparse RNA folding: Time and space efficient algorithms. In Gregory Kucherov and Esko Ukkonen, editors, *Proceedings of the 20th Symposium on Combinatorial Pattern Matching*, volume 5577 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2009. Available from: `http://www.bioinf.uni-freiburg.de/Publications/backofen09:_spars_rna_foldin.pdf`, `doi:10.1007/978-3-642-02441-2_22`. 10.1

[4] J. K. Baker. Trainable grammars for speech recognition. *The Journal of the Acoustical Society of America*, 65(S1):S132–S132, 1979. `doi:10.1121/1.2017061`. 5.4.3.1

[5] Richard E. Bellman. *Dynamic Programming*. Princeton University Press, 1957. 1, 2.1

[6] Ewan Birney and Richard Durbin. Dynamite: A flexible code generating language for dynamic programming methods used in sequence comaprison. In *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology*, pages 56–64, 1997. Available from: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.49.6539&rep=rep1&type=pdf`. 1.3.1

[7] OpenMP Architecture Review Board. OpenMP application program interface version 3.0. Technical report, OpenMP Architecture Review Board, 2008. Available from: `http://www.openmp.org/mp-documents/spec30.pdf`. 5.4.2

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001. 1, 10.2

[9] Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting haskell strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, pages 50–64. Springer-Verlag, January 2007. Available from: `http://www.cse.unsw.edu.au/~dons/papers/fusion.pdf`. 3.1

[10] Boost developers. Boost — free peer-reviewed portable c++ source libraries, 2010. Available from: `http://www.boost.org/`. 6.1

[11] Ye Ding and Charles E. Lawrence. A statistical sampling algorithm for RNA secondary structure prediction. *Nucleic Acids Research*, 31(24):7280–7301, December 2003. Available from: `http://nar.oxfordjournals.org/cgi/reprint/31/24/7280.pdf`, `doi:10.1093/nar/gkg938`. 5.4.3.1

[12] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis*. Cambridge, 1998. 1.2, 2.1.1.1

[13] Jason Eisner, Eric Goldlust, and Noah A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT-EMNLP)*, pages 281–290, Vancouver, October 2005. Available from: `http://cs.jhu.edu/~jason/papers/eisner+goldlust+smith.emnlp05.pdf`. 1.3.3

[14] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 513–522, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics. 10.1

[15] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming i: linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992. `doi:10.1145/146637.146650`. 10.1

[16] David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming ii: convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992. `doi:10.1145/146637.146656`. 10.1

[17] Martin Fekete, Ivo L. Hofacker, and Peter F. Stadler. Prediction of RNA base pairing probabilities using massively parallel computers. *Journal of Computational Biology*, 1999. Available from: `http://www.santafe.edu/media/workingpapers/98-06-057.pdf`. 5.4.2

[18] Free Software Foundation. Bison — GNU parser generator, 2010. Available from: `http://www.gnu.org/software/bison/`. 5.1

[19] Robert Giegerich. Explaining and controlling ambiguity in dynamic programming. In *Proceedings of Combinatorial Pattern Matching*, volume 1848 of *Springer Lecture Notes in Computer Science*, pages 46–59. Springer,

2000. Available from: http://bibiserv.techfak.uni-bielefeld.de/adp/ps/ambi.pdf. 8.1

[20] Robert Giegerich, Carsten Meyer, and Peter Steffen. A discipline of dynamic programming over sequence data. *Science of Computer Programming*, 51(3):215–263, June 2004. Available from: http://bibiserv.techfak.uni-bielefeld.de/adp/ps/GIE-MEY-STE-2004.pdf, doi:10.1016/j.scico.2003.12.005. (document), 2.1, 2.1.2

[21] Robert Giegerich and Peter Steffen. Implementing algebraic dynamic programming in the functional and the imperative programming paradigm. In E.A. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. LNCS 2386, 2002. Available from: http://bibiserv.techfak.uni-bielefeld.de/adp/ps/adp_implementing.ps.gz. 3.1, 5.3.3

[22] Robert Giegerich, Björn Voß, and Marc Rehmsmeier. Abstract shapes of RNA. *Nucleic Acids Research*, 32(16):4843, September 2004. Available from: http://nar.oxfordjournals.org/cgi/reprint/32/16/4843.pdf, doi:doi:10.1093/nar/gkh779. 5.4.3.1, 5.4.3.1

[23] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982. 1.3.1

[24] Dan S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.7183&rep=rep1&type=pdf. 1.3.1

[25] 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*, 125(2):167–188, 1994. Available from: http://fontana.med.harvard.edu/www/Documents/WF/Papers/vienna.rna.pdf, doi:10.1007/BF00818163. 2.1.1.1, 4.3, 5.3.7.3, 5.4.1.3, 5.4.2, 5.4.3.1, 5.4.5, 6.4, 8.1

[26] Matthias Höchsmann. *The Tree Alignment Model: Algorithms, Implementations and Applications for the Analysis of RNA Secondary Structures*. PhD thesis, Universität Bielefeld, 2005. Available from: http://bieson.ub.uni-bielefeld.de/volltexte/2005/709/pdf/diss.pdf. 10.3

[27] Stefan Janssen and Robert Giegerich. Faster computation of exact RNA shape probabilities. *Bioinformatics*, 26(5):632–639, 2010. Available from: http://bioinformatics.oxfordjournals.org/cgi/reprint/26/5/632.pdf, doi:10.1093/bioinformatics/btq014. 5.3.7.2, 8.2

[28] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003. Available from: http://www.haskell.org/definition/haskell98-report.pdf. 1.3.4, 2.1.2, 3.1

[29] W. A. Lorenz, Yann Ponty, and Peter Clote. Asymptotics of RNA shapes. *Journal of Computational Biology*, 15(1):31–63, 2008. Available from: http://www.lri.fr/~ponty/docs/AsymptoticsRNAShapes-JCompBiol-LorenzPontyClote.pdf, doi:10.1089/cmb.2006.0153. 5.4.3.1

[30] Wellington S. Martins, Juan B. Del Cuvillo, Francisco Jose Useche, Kevin B. Theobald, and Guang R. Gao. A multithreaded parallel implementation of a dynamic programming algorithm for sequence comparison. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 311–322, 2001. Available from: http://psb.stanford.edu/psb-online/proceedings/psb01/martins.pdf. 5.4.2

[31] Carsten Meyer and Robert Giegerich. Matching and Significance Evaluation of Sequence-Structure Patterns in RNA. *Journal of Physical Chemistry*, 216:1–24, 2002. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.2.7910&rep=rep1&type=pdf. 4.5.8.1

[32] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), 1965. Available from: ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf. 5.4.2

[33] Akimasa Morihata. A short cut to optimal sequences. *New Generation Computing*, accepted, 2010. 1.3.4

[34] Mathias Möhl, Raheleh Salari, Sebastian Will, Rolf Backofen, and S. Sahinalp. Sparsification of RNA structure prediction including pseudoknots. In Vincent Moulton and Mona Singh, editors, *Proceedings of the 10th Workshop on Algorithms in Bioinformatics (WABI)*, volume 6293 of *Lecture Notes in Computer Science*, pages 40–51. Springer Berlin / Heidelberg, 2010. Available from: http://www.bioinf.uni-freiburg.de/Publications/moehl_wabi10:Sparsification.pdf, doi:10.1007/978-3-642-15294-8_4. 10.1

[35] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970. 1.3.1, 4.5.2

[36] Ruth Nussinov, George Pieczenik, Jerrold R. Griggs, and Daniel J. Kleitman. Algorithms for loop matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978. Available from: http://link.aip.org/link/?SMM/35/68/1, doi:10.1137/0135006. 2.1.1.1, 5.4.5

[37] The Flex Project. flex: The Fast Lexical Analyzer, 2010. Available from: http://flex.sourceforge.net/. 5.1

[38] Janina Reeder and Robert Giegerich. A graphical programming system for molecular motif search. In *Proceedings of the 5th international Conference on Generative Programming and Component Engineering*, pages 131–140, Portland, Oregon, USA, October 22 - 26 2006. ACM Press, New York, NY. GPCE'06. Available from: `http://aop.cslab.openu.ac.il/~lorenz/www/ontheShelf/p131.pdf`. 1.2, 5.3.7.2, 5.3.7.3

[39] Jens Reeder. *Algorithms for RNA secondary structure analysis : prediction of pseudoknots and the consensus shapes approach*. Thesis, Universität Bielefeld, 2007. Available from: `http://bieson.ub.uni-bielefeld.de/volltexte/2008/1276/pdf/thesis.pdf`. 2.1.3, 5.42, 10.1

[40] Jens Reeder and Robert Giegerich. Design, implementation and evaluation of a practical pseudoknot folding algorithm based on thermodynamics. *BMC Bioinformatics*, 5:104, August 2004. Available from: `http://www.biomedcentral.com/content/pdf/1471-2105-5-104.pdf`, `doi:10.1186/1471-2105-5-104`. 2.1.3, 4.5.8.1, 4.5.8.4, 5.3.7.3, 5.4.5, 8.4

[41] Marc Rehmsmeier, Peter Steffen, Matthias Höchsmann, and Robert Giegerich. Fast and effective prediction of microRNA/target duplexes. *RNA*, 10:1507–1517, 2004. Available from: `http://rnajournal.cshlp.org/content/10/10/1507.full.pdf`. 2.1.3, 2.1.3

[42] David Sankoff. Simultaneous solution of the rna folding, alignment and protosequence problems. *SIAM Journal on Applied Mathematics*, 45(5):68–82, October 1985. 1.3.1, 4.5.2

[43] Georg Sauthoff. *Java-Backend für den ADP-Compiler*. Diplomarbeit, Universität Bielefeld, 2007. 2.1.3

[44] Stefanie Schirmer. *A Frontend for the ADP compiler*. Diplomarbeit, Universität Bielefeld, 2006. 5.3.8

[45] Alexander Sczyrba and Jan Krüger. BiBiServ – Bielefeld University Bioinformatics Server, 2010. Available from: `http://bibiserv.techfak.uni-bielefeld.de`. 7.1

[46] Alexander Sczyrba, Jan Krüger, Henning Mersch, Stefan Kurtz, and Robert Giegerich. RNA-related tools on the Bielefeld Bioinformatics Server. *Nucleic Acids Research*, 31(13):3767–3770, 2003. Available from: `http://nar.oupjournals.org/cgi/reprint/31/13/3767.pdf`. 7.1

[47] Peter Steffen. *Compiling a Domain Specific Language for Dynamic Programming*. PhD thesis, Technische Fakultät Universität Bielefeld, 2006. Available from: `http://bieson.ub.uni-bielefeld.de/volltexte/2007/1035/pdf/diss.pdf`. 2.1.3, 5.3.6

[48] Peter Steffen and Robert Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 6(1):224, September 2005. Available from: http://www.biomedcentral.com/content/pdf/1471-2105-6-224.pdf, doi:10.1186/1471-2105-6-224. 2.1.1, 2.1.1, 4.6.3

[49] Peter Steffen and Robert Giegerich. Correction: versatile and declarative dynamic programming using pair algebras. *BMC Bioinformatics*, 7:214, April 2006. Available from: http://www.biomedcentral.com/content/pdf/1471-2105-7-214.pdf, doi:10.1186/1471-2105-7-214. 2.1.1, 4.6.3

[50] Peter Steffen and Robert Giegerich. Table Design in Dynamic Programming. *Information and Computation*, 204(9):1325–1345, 2006. Available from: http://www.techfak.uni-bielefeld.de/~psteffen/pub/tabulate.pdf. 5.3.7, 5.3.7.1

[51] Peter Steffen, Björn Voß, Marc Rehmsmeier, Jens Reeder, and Robert Giegerich. RNAshapes: an integrated RNA analysis package based on abstract shapes. *Bioinformatics*, 22(4):500–503, February 2006. Available from: http://bioinformatics.oxfordjournals.org/cgi/reprint/22/4/500.pdf. 2.1.3, 4.5.9, 5.3.7.3

[52] Kedar Swadi, Walid Taha, and Oleg Kiselyov. Staging dynamic programming algorithms. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, 2005. Available from: www.cs.rice.edu/~taha/publications/preprints/2005-04-13.pdf. 1.3.2

[53] Stephen H. Unger. A global parser for context-free phrase structure grammars. *Communications of the ACM*, 11(4):240–247, April 1968. 5.4.1

[54] Björn Voß. *Advanced Tools for RNA Secondary Structure Analysis*. Thesis, Universität Bielefeld, 2004. Available from: http://bieson.ub.uni-bielefeld.de/volltexte/2005/664/pdf/Diss.pdf. 1.2, 5.4.3.1

[55] Björn Voß, Robert Giegerich, and Marc Rehmsmeier. Complete probabilistic analysis of RNA shapes. *BMC Biology*, 4(1):5, February 2006. Available from: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1479382/pdf/1741-7007-4-5.pdf, doi:10.1186/1741-7007-4-5. 5.4.3.1, 8.3

[56] Ydo Wexler, Chaya Zilberstein, and Michal Ziv-Ukelson. A study of accessible motifs and rna folding complexity. *Journal of Computational Biology*, 14(6), 2007. doi:10.1089/cmb.2007.R020. 10.1

[57] Stefan Wuchty, Walter Fontana, Ivo L. Hofacker, and Peter Schuster. Complete suboptimal folding of RNA and the stability of secondary structures. *Biopolymers*, 49:145–165, 1998. Available from: http://www.santafe.edu/~walter/Papers/subopt.pdf. 5.4.3, 8.1

[58] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, February 1967. 1.2, 5.4.1

[59] Michael Zuker and Patrick Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research*, 9(1):133–148, 1981. 4.5.2