Yield grammar analysis in the Bellman's GAP compiler

Robert Giegerich
Universität Bielefeld
Technische Fakultät, 33501 Bielefeld, Germany
robert@techfak.uni-bielefeld.de

Georg Sauthoff
Universität Bielefeld
Technische Fakultät, 33501 Bielefeld, Germany
gsauthof@techfak.uni-bielefeld.de

ABSTRACT

Dynamic programming algorithms are traditionally expressed by a set of table recurrences – a low level of abstraction which renders the design of novel dynamic programming algorithms difficult and makes debugging cumbersome.

Bellman's GAP is a declarative language supporting dynamic programming over sequence data. It implements *algebraic* dynamic programming and allows specifying algorithms by combining so-called yield grammars with evaluation algebras. Products on algebras allow to create novel types of analysis from already given ones, without modifying tested components. Bellman's GAP extends the previous concepts of algebraic dynamic programming in several respects, such as an "interleaved" product operation and the analysis of multi-track input.

Extensive analysis of the yield grammar is required for generating efficient imperative code from the algebraic specification. This article gives an overview of the analyses required and presents three of them in detail. Measurements with "real-world" applications demonstrate the quality of the code produced.

Categories and Subject Descriptors

D.3 [**PROGRAMMING LANGUAGES**]: Language Classifications—*Specialized application languages*

General Terms

Languages,Algorithms

Keywords

Dynamic Programming, Regular Tree Grammars, RNA Structure Prediction

LDTA 2011 Saarbrücken, Germany

1. INTRODUCTION

1.1 Challenges in the design of ADP algorithms

Dynamic programming (DP) is a well-established technique, applicable to a wide class of combinatorial optimization or enumeration problems. Educational examples, found in many introductory texts, are string edit distance, matrix chain multiplication, knapsack problems, El Mamun's caravan, or Fibonacci numbers. In these examples, a single recurrence relation with a two- or three-way case distinction suffices to describe the problem decomposition, the scoring of solution candidates and the selection of an optimal solution. Adding the idea of tabulation for re-use of intermediate results, and replacing top-down recursion by a bottom-up computation governed by for-loops, together with a discussion of Bellman's Principle of Optimality completes a first acquaintance with the essential ingredients of dynamic programming.

Such simplicity is deceiving for several reasons.

1. In realistic application scenarios, we may have tens, sometimes hundreds of recurrences, and an even larger set of cases to be distinguished in the problem decomposition. Tabulating all of these recurrences is prohibitive for space reasons, and the question is which ones need not to be tabulated, while retaining optimal asymptotic efficiency.

2. Often, the same search space is to be analyzed under different scoring schemes or objective functions, and the tight entanglement of these issues encoded in the traditional for-loops is a major obstacle to programmer productivity.

3. Debugging implementations where the case analysis is encoded in hundreds of subscript calculations is a Sisyphean task.

4. Code to backtrace the solution behind an optimal score is tedious to produce and to debug. Even more so, when near-optimal solutions are to be generated. Such code can be automatically generated from the algebraic specification.

All in all, while the basic ideas of DP are easy to grasp, for "real-world" problems, support is desirable which helps to specify problems on a more abstract level and automates away implementation details.

1.2 Previous approaches to support DP

Due to the manifold applications in biosequence analysis, this field has brought about two early approaches to support design and implementation of dynamic programming algorithms over sequence data. The *Dynamite* system [3] models recurrences as state transitions with which scoring functions are associated. The *Telegraph* system [12], provides an object-oriented template library for dynamic programming, striving for greater flexibility. Both approaches firmly rest on the imperative paradigm. Dynamic programming (for a single recurrence) has been cast as a relational calculus by Bird and de Moor [2]. Staging DP [20] provides another combinator library to support dynamic programming. Recently, Morihata has studied the automated derivation of efficient algorithms, using ingredients of dynamic programming, from naive enumerate-and-choose-style specification [13]. In natural language processing (NLP), the development of parsing algorithms is supported by the DYNA language [6], which is based on a general, Prolog-style backtracking scheme.

The discipline of *algebraic dynamic programming* (ADP) [8] is a language-independent formalism, which evolved from a Haskell-basel combinator language [7]. It is a declarative approach, which, in contrast to the aforementioned ideas, achieves a perfect separation of the issues of search space decomposition, candidate scoring, and tabulation for sake of efficiency. This first implementation allowed us to develop several novel bioinformatics tools in a relatively short time, but the approach has not become useful in the hands of others. The main obstacle appeared to be the Haskell-reminiscent syntax of ADP code.

In Bellman's GAP, we create a new syntax, incorporate novel features of the ADP theory, and add compiler optimizations to provide a second generation implementation of ADP. Bellman's GAP is named after its underlying concepts, which are Bellman's Principle of Optimality, Grammars, Algebras, and Products. The overall system consists of the programming language GAP-L, its compiler GAP-C, a library of predefined modules for applications in biosequence analysis (GAP-M), and educational material (GAP-Pages) currently under development. This article focuses on new grammar analysis techniques implemented in GAP-C.

2. ALGEBRAIC DYNAMIC PROGRAMMING IN BELLMAN'S GAP

Syntax.

The design of a new, Java-style concrete syntax for algebraic dynamic programming was a major concern in the development of Bellman's GAP. However in this article, for lack of space, we do not describe the concrete syntax of GAP-L, as the semantic concepts behind Bellman's GAP programs are sufficient to explain our grammar analysis techniques. Readers interested in the concrete look-and-feel of GAP-L programs are referred to [17, 16] and forthcoming publications.

Semantics.

$\mathcal{A}$ denotes the alphabet of the string (sequence) to be analyzed. A *signature* Σ over $\mathcal{A}$ is a set of function symbols and a datatype place holder (sort) S. The return type of each function symbol is S, each argument is of type S or $\mathcal{A}$. T_Σ denotes the term language described by the signature Σ and $T_\Sigma(V)$ is the term language with variables from the set V. A Σ-*algebra* or *interpretation* $\mathcal{I}$ is a mathematical structure given by a carrier set $S_\mathcal{I}$ for S and functions $f_\mathcal{I}$ operating on this set, for each $f \in \Sigma$, consistent with its specified type. Interpreting a term $t \in T_\Sigma$ by algebra $\mathcal{I}$ is denoted $\mathcal{I}(t)$ and yields a value in $S_\mathcal{I}$. A *regular tree grammar* G over a signature Σ is defined as tuple $(V, \mathcal{A}, Z, P)$, where V is the set of non-terminals, $\mathcal{A}$ is an alphabet, $Z \in V$ is the axiom, and P is the set of productions. Each production is of form

$$v \to t \text{ with } v \in V, t \in T_\Sigma(V) \tag{1}$$

The *language* generated by a tree grammar $\mathcal{G}$ is the set of trees

$$\mathcal{L}(\mathcal{G}) = \{t \in T_\Sigma | Z \Rightarrow^* t\} \tag{2}$$

where $\Rightarrow^*$ is the reflexive transitive closure of $\to$. By construction, $\mathcal{L}(\mathcal{G}) \subseteq T_\Sigma$ – its elements are seen as trees when it comes to constructing them, and as formulas when it comes to their evaluation.

Symbols from $\mathcal{A}$ reside on the leaves of these trees. 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 $y(f(x_1, \ldots, x_n)) = y(x_1) \ldots y(x_n)$, for f from Σ and $n \geq 0$. The *yield language* $\mathcal{L}_y(\mathcal{G})$ of a tree grammar $\mathcal{G}$ is defined as

$$\mathcal{L}_y(\mathcal{G}) = \{y(t) | t \in \mathcal{L}(\mathcal{G})\} \tag{3}$$

Tree grammars are called *yield grammars* in our context, as we face a special type of parsing problem: Given $x \in \mathcal{A}^*$, we construct the *search space* $\mathcal{S}(x) = \{t | t \in \mathcal{L}(\mathcal{G}), y(t) = x\}$. This process - computing the inverse of y – is called *yield parsing*. A yield parser works essentially like a context-free parser for an ambiguous language, but the trees it returns are not parse trees, but rather elements of T_Σ – the solution candidates in $\mathcal{S}(x)$. A user of Bellman's GAP need not care about how yield parsing works.

An *evaluation algebra* $\mathcal{E}$ is a Σ-algebra augmented with an objective function $h_\mathcal{E} : [S_\mathcal{E}] \to [S_\mathcal{E}]$, where the square brackets denote multisets (in theory, and lists in practice).

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

$$\mathcal{G}(\mathcal{E}, x) = h_\mathcal{E}[\mathcal{E}(t) | t \in \mathcal{L}(\mathcal{G}), y(t) = x] \tag{4}$$

The square brackets in Equation 4 denote a multiset. This is required because in practice, we often ask for all co-optimal solutions, or all solutions within a percentage of optimality. The notation $\mathcal{G}(\mathcal{E}, x)$ suggests the use of the grammar (more precisely, its yield parser) as a function called with evaluation algebra and input as parameters. In the compiled code, however, Equation 4 is not executed literally. Rather, the application of the objective function is amalgamated with the evaluation of the candidate trees, which are not constructed explicitly. (In functional language terminology, this is a case of deforestation.) Also, tabulation of intermediates is used where appropriate, to avoid exponential blow-up.

The prerequisite for correct and efficient computation of the solution in this way is Bellman's Principle of Optimality [1], which in the ADP framework is defined by Equations 5 and 6.

$$h_\mathcal{E}[\, f_\mathcal{E}(x_1,\ldots,x_k) \mid x_1 \leftarrow X_1,\ldots,x_k \leftarrow X_k\,] = \\ h_\mathcal{E}[\, f_\mathcal{E}(x_1,\ldots,x_k) \mid x_1 \leftarrow h_\mathcal{E}(X_1),\ldots,x_k \leftarrow h_\mathcal{E}(X_k)\,] \tag{5}$$

$$h_\mathcal{E}(X_1 \cup X_2) = h_\mathcal{E}(\, h_\mathcal{E}(X_1) \cup h_\mathcal{E}(X_2)\,) \qquad \text{and} \qquad h_\mathcal{E}[\,] = [\,] \tag{6}$$

where the X_i denote multisets.

This is a very general formulation of Bellman's Principle of Optimality, which is often only informally stated in the literature. Cf. [5]. When the objective function h is minimization or maximization, this implies (strict) monotonicity of each f in Σ [14]. When Σ holds only a single, binary and commutative function f, (h, f) forms a semi-ring where h distributes over f [15].

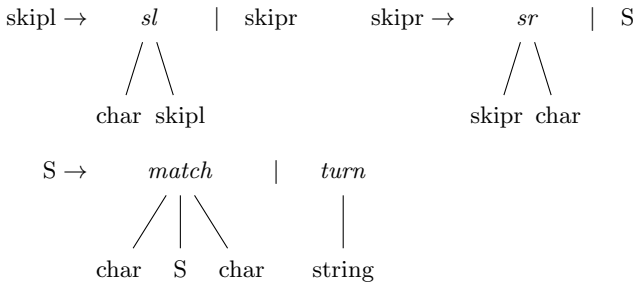
Compiling an ADP algorithm coded in GAP-L, it is always assumed that the evaluation algebra satisfies Bellman's Principle. This is the programmer's responsibility. However, in some cases the compiler can notice that this principle is violated, and will issue a warning.

Example.

We use palindromic strings as an introductory example, since this problem is similar to the RNA structure analysis we deal with in our bioinformatics work, but shorter to explain. The palindromic strings of interest will be described by grammar *Pali* and will be scored by algebra *Local*.

An exact palindrome is a string $u \in \mathcal{A}^*$ such $u = u^{-1}$. The palindromic structure of such a string, if it is a palindrome at all, is unique and not a matter of finding a best structure. We consider, as a more general problem, local gapped palindromes which have the form $aubvc$ with $a, b, c, u \in \mathcal{A}^*$ and $u \approx v^{-1}$. Now many different palindromic structures can be assigned to a string p, and we provide a scoring scheme to designate optimal structure(s). The scoring scheme allows us to impose different penalties on skipped characters in a, b, and c, and different bonuses (for matched characters) and penalties (for mismatched characters) in u and v^{-1}. So, our signature provides function symbols $sl, turn, rl$ and $match$, in the above order of cases.

Grammar *Pali* describes the search space of all candidate terms, where *skipl* is the axiom:



Here *string* is a terminal symbol denoting an arbitrary string over $\mathcal{A}$, and char denotes a character. In our simple scoring scheme, skipping characters in the beginning or end or in the middle turn is for free:

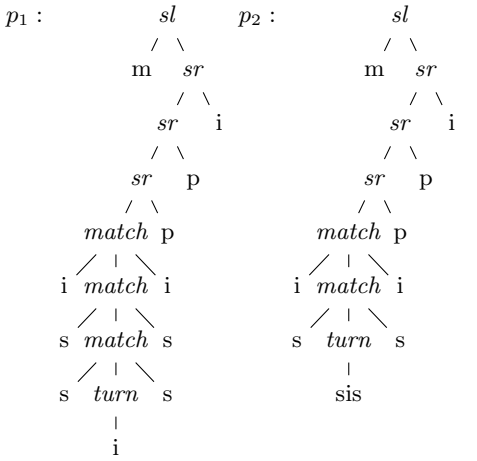
Algebra *Local*, $X = \mathbb{N}$

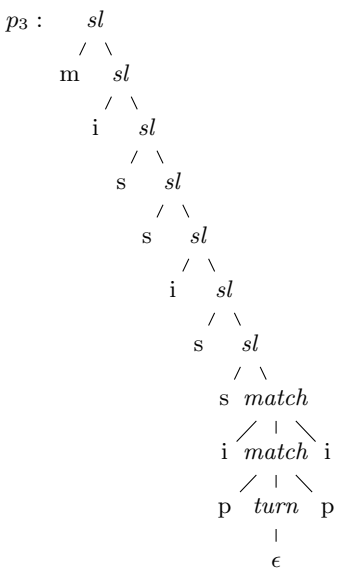
$$\begin{aligned} match(a,x,b) &= \begin{cases} x+3 & \text{if} \quad a = b \\ x-1 & \text{otherwise} \end{cases} \\ turn(x) &= 0 \\ sl(a,x) &= x \\ sr(x,a) &= x \\ h(l) &= [\max l] \end{aligned}$$

If we depict alternative structures using parentheses for matched characters, '+' for the turn, and '-' for leading/-trailing characters, three candidate solutions (out of many) for the string "mississippi" can be depicted as

```
     mississippi
p1: -(((+)))---
p2: -((+++))---
p3: -------(())
```

As elements of $\mathcal{L}(Pali)$, they are represented as trees





where the reader may verify (Eq. 4) that $Pali(Local,$"mississippi"$) = [9]$, where this optimal score is derived from the candidate p_1.

Products on algebras.

A main virtue of *algebraic* dynamic programming is that

algebras can be combined to build more complex analyses from tried-and-tested simpler ones - without re-programming effort. Bellman's GAP provides a cartesian, a lexicographic and an interleaved product operation, where the lexicographic product has been introduced in [18] and the other two are new. Given algebras A and B, the programmer simply writes $A \times B$, $A * B$, or $A \otimes B$, and GAP-C is responsible to create a single algebra from it conforming to the subsequent definitions.

An algebra is called *unitary*, if its evaluation function returns lists of results holding at most one element. A *generic* algebra $A(k)$ may have a parameter k such that it returns the k best solutions under its objective function. Naturally, $A(1)$ is unitary.

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

$$\begin{array}{l} f_{A\times B}(x_1,\ldots,x_k) = (f_A(a_1,\ldots,a_k), f_B(b_1,\ldots,b_k)) \\ \quad \text{if} \quad x_i = (x_i^A, x_i^B), \quad \text{then} \quad a_i = x_i^A, b_i = x_i^B, \\ \quad \text{if} \quad x_i \in \mathcal{A}, \qquad\quad \text{then} \quad a_i = x_i = b_i \end{array} , \tag{7}$$

for each $f \in \Sigma, 1 \le i \le k, \quad x_i^A \in \mathcal{S}_A, x_j^B \in \mathcal{S}_B$.
$A \times B$ has the objective function

$$\begin{array}{l} h_{A\times B}[(a_1,b_1),\ldots,(a_m,b_m)] = \quad [\,(l,r) \mid \\ \qquad\qquad\qquad\qquad\qquad\quad l \leftarrow h_A[a_1,\ldots,a_m], \\ \qquad\qquad\qquad\qquad\qquad\quad r \leftarrow h_B[b_1,\ldots,b_m] \quad]. \end{array} \tag{8}$$

The cartesian product has little use by itself – instead of $\mathcal{G}(A \times B, x)$ we might just as well call $\mathcal{G}(A, x)$ and $\mathcal{G}(B, x)$ separately, which gives the same result albeit more slowly. However, in combinations with other products, we have found the cartesian product useful. For example, in RNA structure prediction, a product $SHAPE*(MFE \times COUNT)$ splits the search space of all structures into shape classes (algebra $SHAPE$), and computes for each class its minimum free energy structure (MFE) along with the size of the class ($COUNT$).

For arbitrary A, B the *lexicographic product* $A * B$ is an evaluation algebra over Σ and has the functions

$$f_{A*B} = f_{A\times B} \tag{9}$$

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

$$\begin{array}{l} h_{A*B}[(a_1,b_1),\ldots,(a_m,b_m)] = \\ \quad [\,(l,r) \mid \\ \qquad l \leftarrow \mathrm{set}(h_A[a_1,\ldots,a_m]), \\ \qquad r \leftarrow h_B[\,r' \mid (l',r') \leftarrow [(a_1,b_1),\ldots,(a_m,b_m)],\ l' = l\,] \quad]. \end{array} \tag{10}$$

Here, $\mathrm{set}(X)$ reduces the multiset X to a set. This product gets its name from the fact that if both algebras optimize with respect to orderings $<_A$ and $<_B$, then $A * B$ optimizes with respect to the lexicographic ordering $(<_A, <_B)$. However, this product is not restricted to the case of two optimizing algebras, and in fact it exhibits a surprising versatility of use – see [18].

Our third product combines the objective functions in a more sophisticated way. 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

$$f_{A\otimes B} = f_{A\times B} \tag{11}$$

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

$$\begin{array}{l} h_{(A\otimes B)(k)}[(a_1,b_1),\ldots,(a_m,b_m)] = \\ \qquad [\,(l,r) \mid (l,r) \leftarrow U, p \leftarrow V, p = r] \\ \quad \text{where} \\ \qquad U = h_{A*B(1)}[(a_1,b_1),\ldots,(a_m,b_m)] \\ \qquad V = \mathrm{set}(h_{B(k)}[\,v \mid (\_,v) \leftarrow U]) \end{array} \tag{12}$$

This product is useful when $B(k)$ alone produces the k-best-scoring candidates, and A maps candidates to different classes (not making any choices). Then, $A \otimes B$ yields the 1-best-scoring candidates from k *different* classes. This situation required sophisticated hand-coding in RNA shape analysis [10] – it is now simply written as $(SHAPE \otimes MFE)(k)$.

In many cases, a product of two algebras preserves Bellman's Principle – but not always. Proof of preservation is an obligation of the programmer.

Pragmatics.

More features are required to turn a concise formalism into a practical programming tool. Signatures and algebras may be many-sorted, if subproblems yield results of different types. The objective function h then splits into a separate function for each result type. Filter predicates may be associated with grammar rules, allowing one to further restrict the search space, requiring (say) a minimal size of parsed sub-words (*syntactic filter*) or a score above a certain threshold (*semantic filter*). *Generic* algebras may contain parameters (other than the above k for the number of results to be returned), to be instantiated when the algebra is applied. A default terminal symbol *loc* is supplied which derives ε, returning the input position where this ε was recognized. Thus, one can access input locations in the algebra functions, although otherwise, all subscript fiddling is hidden from the programmer. Sometimes, a grammar requires many rules with renamed nonterminal symbols, but isomorphic in structure. In this case, GAP-L allows the use of nonterminal symbols that pass parameters from the left-hand to the right-hand side. Finally, for programming in the large, GAP-L provides a module concept and allows to mix several algebras, signatures and grammars in one program.

There is one further innovation to report from Bellman's GAP, which gives rise to important optimization challenges.

Multi-track grammars.

GAP-L supports dynamic programming on multiple sequences. A well-known example of a multi-track DP algorithm is the pairwise sequence alignment algorithm, which finds the optimal sequence of edit operations to transform one sequence into the other.

We introduce multi-track tree grammars. For $l = 2$ they have non-terminal symbols that read from two input tracks. Function symbols can take 2-tuples as arguments. A terminal or non-terminal symbol of the first component reads from the first track and one of the second component reads from the second track. A multi-track grammar may contain single-track rules, which can be called as part of a function symbol argument.

Similarly for $l > 2$ tracks. Their yield function y returns an l-tuple of strings, and Equation 13 specifies the semantics.

$$\mathcal{G}(\mathcal{E}, x_1, \ldots, x_l) = h_{\mathcal{E}}[\mathcal{E}(t) | t \in \mathcal{L}(\mathcal{G}), y(t) =< x_1, \ldots, x_l >] \tag{13}$$

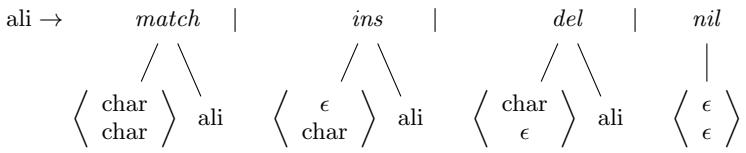


Figure 1: Pairwise sequence alignment algorithm as multi-track grammar.

Note that l tracks in general lead to tables of dimension $2l$, such that an l larger than 2 or 3 is not very practical.

As an example (Figure 1), we write a sequence alignment grammar, using a signature with scoring functions *match*, *ins*, *del*, and *nil* for matching characters, insertions, deletions, and for the empty alignment.

3. YIELD GRAMMAR ANALYSIS

Generating recurrences.

Given yield grammar and algebra(s), the Bellman's GAP compiler eventually generates the typical DP recurrences. For grammar *Pali*, algebra *Local*, and input variable u, we obtain:

$$skipl_i = \max \begin{cases} skipl_{i+1} & \text{if} \quad n-i>1 \\ skipr_{i,n} & \text{if} \quad n-i>0 \\ 0 & \text{otherwise} \end{cases}$$

$$skipr_{i,j} = \max \begin{cases} skipr_{i,j-1} & \text{if} \quad j-i>1 \\ S_{i,j} & \text{if} \quad j-i>0 \\ 0 & \text{otherwise} \end{cases}$$

$$S_{i,j} = \max \begin{cases} S_{i+1,j-1}+3 & \text{if} \quad u_i=u_j, j-i>2 \\ S_{i+1,j-1}-1 & \text{if} \quad u_i\neq u_j, j-i>2 \\ 0 & \text{otherwise} \end{cases}$$

We find that the compiler has decided that nonterminal symbol *skipl* only requires a linear table, while S and *skipr* require an $n \times n$ table. Intensive inlining has been done – the fact that scoring in *Local* does not depend on the concrete characters skipped leads to the fact that in the recurrences for *skipl* and *skipr*, the input string is not even accessed. This analysis can go a long way. In fact, had we modeled exact palindromes by our yield grammar, the compiler would optimize away all the dynamic programming machinery and produce a simple recursive test function.

To achieve code quality competitive to handwritten code requires sophisticated semantic analyses and optimizations. Not only constant factors, but also the asymptotic runtime depends on the compiler. The major tasks are the following:

- *grammar soundness* checks, if the grammar is productive, i.e. it can derive a terminal tree, and does not contain any non-productive derivation loops. Meaningful error messages are generated if this should happen.
- *yield size analysis* (YSA) determines lower and upper bounds on the yields of each nonterminal symbol. This is required to generate tight subscript bounds and inner loops in the recurrences.
- *table dimension analysis* (DA) determines whether intermediate results for a nonterminal symbol, if tabulated, require a table of dimension 1, 2, 3, 4 , ..., $2k$, where $2k$ is the theoretical maximum for a k-track problem.
- *table design* (TD) determines which nonterminals need to be tabulated to achieve minimal space requirements without raising asymptotic runtime above the optimum.
- *backtracing*: certain computations for algebras in products can be implemented more efficiently by an equivalent backtracing scheme, which is generated in such a case.
- *inlining* of algebra functions often allows for substantial improvements.

The phases apply in the order listed here. For space reasons, here we only describe YSA, DA, and TD.

Yield Size Analysis.

The yield size *ys* of a terminal or non-terminal symbol A is defined as the pair of the minimal and maximal sub-word length it can generate:

$$ys_l(A) = \min\{|y(t)| \mid t \in \mathcal{L}(A)\} \tag{14}$$
$$ys_u(A) = \max\{|y(t)| \mid t \in \mathcal{L}(A)\} \tag{15}$$
$$ys(A) = (ys_l(A), ys_u(A)) \tag{16}$$

For example, $ys(\text{char}) = (1,1)$ and $ys(\text{S}) = (0,\infty)$.

Given an input sequence of length n, the maximal relevant yield size is n. This n is not known at compile-time, hence 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 lower and upper yield sizes has saturating semantics, i.e. $0-c=0$ and $n+c=n$, where $c \in \mathcal{B}$.

YSA in GAP-C is divided into two phases. The first phase computes the yield sizes for all non-terminal symbols, reading the rules of the grammar right-to-left. The following three equations from [9] define the yield size computation for single-track grammar constructs:

$$ys(g(a,b)) = (a_l + b_l, a_u + b_u),\ g \in \Sigma \tag{17}$$
$$ys(a \mid b) = (\min(a_l, b_l), \max(a_u, b_u)) \tag{18}$$
$$ys(a \textit{ with } f) = (\max(a_l, f_l), \min(a_u, f_u)) \tag{19}$$

where $a, b \in T_\Sigma(V)$ and $f : \mathbb{N} \times \mathbb{N} \to \{true, false\}$ is a filter function that takes the bounding indices of the subword parsed by a as input and filters according to a locally specified minimal or maximal size. g is taken as a binary

function symbol from Σ here; for functions of different arity, analogous definitions apply.

For a multi-track grammar, the yield size of an l-track non-terminal is defined as an l-vector that contains for each track the minimal and maximal sub-word length. The following equations are used to derive the yield size of multi-track rules:

Multi-track requires one new rule, and the others generalize component wise the single-track ones.

$$ys(< a_1, \ldots, a_n >) =< ys(a_1), \ldots, ys(a_n) > \quad (20)$$

$$\begin{aligned} &ys(g(< a_1, \ldots, a_n >, < b_1, \ldots, b_n >)) \\ &= < (a_{1l} + b_{1l}, a_{1u} + b_{1u}), \ldots, (a_{nl} + b_{nl}, a_{nu} + b_{nu}) > \end{aligned} \quad (21)$$

$$\begin{aligned} &ys(< a_1, \ldots, a_n > | < b_1, \ldots, b_n >) \\ &\quad = < (\min(a_{1l}, b_{1l}), \max(a_{1u}, b_{1u})), \ldots, \\ &\qquad (\min(a_{nl}, b_{nl}), \max(a_{nu}, b_{nu})) > \end{aligned} \quad (22)$$

$$\begin{aligned} &ys(< a_1, \ldots, a_n > with < b_1, \ldots, b_n >) \\ &\quad = < (\max(a_{1l}, b_{1l}), \min(a_{1u}, b_{1u})), \ldots, \\ &\qquad (\max(a_{nl}, b_{nl}), \min(a_{nu}, b_{nu})) > \end{aligned} \quad (23)$$

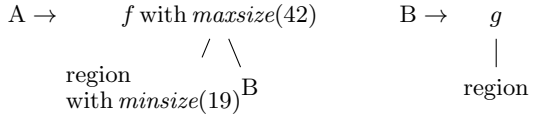
This maps the grammar to a set of recursive equations, and fixpoint iteration is used to compute the yield sizes. Initial values are $(1, n)$ for one-track symbols and $< (1, n), \ldots, (1, n) >$ for multi-track symbols. The termination of the fixpoint iteration is guaranteed because the yield size interval length is monotonically decreasing.

The yield sizes are needed for subsequent optimizations. For example, a quadratic table for a non-terminal with a constant upper yield size c needs only $O(cn)$ space. Equally important are runtime concerns: Consider the following rule:

$$\text{S} \rightarrow \quad g \quad | \; e \quad (24)$$

In general, this type of rule leads to a recurrence for non-terminal S which includes two moving index boundaries k, l on the right hand side of the matrix-recurrence: $S_{i,j} = \max_{i<k\leq l<j} \ldots S_{k,l} \ldots$, and consequenctly to a program with two embedded for-loops and a runtime of $O(n^4)$. If YSA determines that non-terminals A and B have a constant lower and upper yield-size, the indices k, l are eliminated by this optimization phase and the runtime goes down to $O(n^2)$ for this rule.

In its second phase, YSA does a depth-first traversal of the yield-grammar to propagate max-size yield size filter restrictions left-to-right. The following artificial example shows the effect of the second phase:



The effective maximal yield size of B is 23, if B is not called from other rules.

Table Dimension Analysis.

In DA, the compiler seeks to reduce the dimensions of tables storing sub-solutions of non-terminal parsers. For single-track DP algorithms, a table with $O(n^2)$ entries is generally needed. A linear or constant sized table suffices if one or if both indices are constant across all calls of the recurrences. For the *Pali* example, DA returns a linear table for *skipl* and quadratic tables for *skipr* and S. As a consequence, an index elimination phase deletes indices that are not needed due to reduced dimensions, e.g. the j-index of *skipl*, cf. recurrence for *skipl* above.

In the case of reduced dimensions, DA saves space, and it may also influence the runtime of the resulting program. In the multi-track case, DA is even more important, because the table size of the generic case can be prohibitively large in practice. For two-track DP, the default table size is in $O(n^4)$. But e.g. the pairwise sequence alignment algorithm only needs an $O(n^2)$ table and has a runtime of $O(n^2)$. DA succeeds to reduce table dimensions accordingly.

DA analyzes each input track separately and does a depth first traversal (without repetition) of the grammar. During that traversal, moving boundaries and yield-size information are picked up for later case distinctions. For example for a two-track grammar, DA may return for some non-terminal that a linear table of linear tables is needed – the dimension is 2 rather than 4. DA also detects cases where a table access index only changes a constant number times, such that an asymptotically linear or constant size table is allocated.

The case distinction of the DA for one track conservatively detects three different table access patterns of non-terminal parser:

quadratic tables

$$A = \{(i, j) | 0 \leq i \leq j \leq n\} \quad (25)$$

linear tables

$$B_l = \{(i, j) | 0 \leq i \leq j \leq n, 0 \leq i \leq k\} \quad (26)$$

$$B_r = \{(i, j) | 0 \leq i \leq j \leq n, k \leq j \leq n\} \quad (27)$$

constant tables

$$C = \{(i, j) | 0 \leq i \leq j \leq n, 0 \leq i \leq k, l \leq j \leq n\} \quad (28)$$

where k and l are constants.

Table Design.

In the TD phase, the compiler derives a *table configuration* for the grammar. In text-book presentations of DP algorithms, it is assumed that all recurrences need to be tabulated, but in fact, this is only mandatory for some of them. A *table configuration* is the set of non-terminals that are tabulated. Non-tabulated non-terminals have their recurrence computed on demand. The *table design problem* is to find a minimal table configuration under which the runtime of the resulting program is asymptotically optimal.

Tabulating everything yields a program with asymptotically optimal runtime, given that table dimensions are minimal. Tabulating nothing usually yields a program with exponential runtime, because of an exponential amount of recomputation of intermediate results.

In general, TD is NP-complete [19], and GAP-C uses a heuristic which appears quite successful in practice. It takes certain constant factors into account, so that extra tables

Table 1: Runtime and space measurements of various DP programs (see text). The Haskell program was run on shortened input in two cases marked (%), as with full length, it ran out of resources.

part	program	asympt. rt	NTs	tables	n	sec	MB
A	RNAfold	$O(n^3)$	-	2	400	0.23	1.4
	ADPfold (Haskell)	$O(n^3)$	11	4	400	30.22	97.0
	ADPfold (ADPC)	$O(n^3)$	11	4	400	0.39	7.0
	ADPfold (GAP-C)	$O(n^3)$	11	5	400	0.38	2.0
B	pknotsRG (Haskell)	$O(n^4)$	25	7	%300	388.00	190.7
	pknotsRG (ADPC)	$O(n^4)$	25	7	400	11.98	18.4
	pknotsRG (GAP-C)	$O(n^4)$	25	19	400	8.12	5.5
C	RNAshapes (ADPC)	$O(\alpha^n n^3)$	26	11	127	390.00	3504.0
	RNAshapes (GAP-C)	$O(\alpha^n n^3)$	26	15	127	103.00	1917.0
D	TDM (Haskell)	$O(n^3)$	195	59	%103	10.96	442.9
	TDM (GAP-C)	$O(n^3)$	195	147	400	16.16	664.0

are used if the constant factor of a runtime saved is above a threshold.

In the palindrome example *Pali*, tabulating only non-terminal S yields still an asymptotically optimal runtime of $O(n^2)$, which is also the result of the heuristic.

The table design algorithm works on the dependency graph of the yield grammar. This is a directed acyclic graph $D = (V, E)$, where V is the set of terminal and non-terminal symbols and $(A, B) \in E$, if symbol A calls symbol B, i.e. A uses B on one of its righthand sides. The number of calls of A from B is an attribute of the edge (A, B). TD works in four phases:

1. Determine the asymptotically optimal runtime $O(n^r)$. This analysis assumes a unitary evaluation algebra with score functions that compute in $O(1)$. Thus, the contribution of search space construction to overall runtime it determined. Assume a full table configuration. A set of runtime recurrences is derived from the grammar. Yield-size information enters at this point: A split of an input subword in two parts derived from non-terminals A and B contributes a factor of n if and only if *A and B* have unbounded yield-size. Solving the runtime recurrences yields the asymptotically optimal runtime. Afterwards, clear the table configuration.

2. Score each non-terminal vertex in the dependency graph according to scoring function s:

$$s(x) = in(x) \cdot out(x) \cdot selfrec(x) \qquad (29)$$

   where *in* computes the number of calls of x, *out* computes the number of calls from x and *selfrec* computes the recursion factor, which depends on the circular dependencies the non-terminal is part of. Thus, the score of a non-terminal is higher, if it is more connected and takes part in more recursions.

3. Sort the non-terminal vertices in descending order with their score as key into a list.

4. Iteratively remove the highest scored non-terminal vertex, set it as tabulated and re-compute the asymptotic runtime under the present table configuration, until it reaches $O(n^r)$.

Note that this heuristic is safe in the sense that it always implements the grammar with the best asymptotic runtime efficiency.

Measurements.

Compilation speed in Bellman's GAP is good. In our largest application (case D in Table 1 below) compilation with GAP-C 1 second runtime and subsequent C++ compilation takes 20 seconds.

To evaluate the quality of GAP-C-generated code, we compare runtimes of bioinformatics applications. Please accept the URL `http://bibiserv.cebitec.uni-bielefeld.de` as a joint reference for these tools. There you find pointers to the literature, statistics of tool usage (via the web and software downloads), and examples. We compare a handcrafted tool, versions of the same algorithms coded in Haskell, their compilations with the first ADP compiler (ADPC), and re-implementations in Bellman's GAP (Table 1). The tools are considered in order of increasing grammar size. Input sizes are typical ones in practice. Space complexity is $O(n^2)$ in all cases. Measurements are averaged over several runs.

Part A compares ADP based implementations to the program *RNAfold*, which was written in 1994 [11] and carefully engineered since then. It serves a more specialized purpose than the three ADPfold versions; in particular, it does not provide for enumeration of near-optimal solutions, and hence requires only two tables rather than four[1]. All tools were used for the most basic task, minimum free energy prediction of a single RNA structure. We find that GAP-C improves over earlier ADP implementations, and comes close *RNAfold* in spite of its more refined model.

For the algorithms tested in Part B – D, no "hand-crafted" implementations exist to compare to. Here we test more resource-demanding tools, where *RNAshapes* uses a triple product algebra and computes output that grows exponentially with $\alpha \approx 1.1$. The program TDM is a so-called thermodynamic matcher, an RNA folding program specialized to structures of a common overall shape. Such programs are generated from an abstract shape description (by yet an-

[1]Of course, a yield grammar exactly modeling *RNAfold* and using two tables *can* be written in Bellman's GAP.

other tool). Automated table design is essential in such a case, as there is no human programmer involved.

Summarizing, we observe that GAP-C consistently improves over earlier implementations of ADP. Notably, with GAP-C, TD optimization often leads to more than a minimal number of tables for better constant runtime factors, and thanks to the DA optimization, this still requires less space.

4. CONCLUSION

For the development and evaluation of Bellman's GAP, it has been fortunate that there were "real-world" applications already based on ADP. We only had to transliterate them into Bellman's GAP to check preservation of semantics and improvement of performance. Naturally, these tools do not yet make use of the extensions introduced in Bellman's GAP. We will now enter the stage of developing new tools. The first challenge on our list is the problem of minisatellite comparison. Minisatellites are repetitive DNA sequences used in population studies, and their analysis requires a combination of a single track problem, the reconstruction of duplication histories, and a two-track problem, the alignment of minisatellites.

Semantic ambiguity is a pervasive problem in dynamic programming. Although undecidable in general, successful ambiguity checking methods are available for the one track case [4]. Extending ambiguity checking to the multi-track case is another open challenge.

Bellman's GAP carefully analyses the yield grammar for efficient code, but does not transform it. There are known grammar transformations which improve efficiency. In the palindrome example, the grammar may be rewritten such that *skipr* maps to a one-dimensional table. Such optimizations have not been touched upon, as they are semantically delicate: they also require a faithful transformation of the evaluation algebra.

5. REFERENCES

[1] R. E. Bellman. *Dynamic Programming*. PUP, 1957.

[2] R. S. Bird and O. d. Moor. *Algebra of Programming*. PH, 1997.

[3] E. Birney and R. Durbin. Dynamite: A flexible code generating language for dynamic programming methods used in sequence comparison. In *Proc. of the 5th ISCB*, pages 56–64, 1997.

[4] C. Brabrand, R. Giegerich, and A. Møller. Analyzing Ambiguity of Context-Free Grammars. *SCICO*, 75(3):176–191, March 2010.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, pages 309–314. MIT Press, 1990.

[6] J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proc. of HLT-EMNLP*, pages 281–290, 2005.

[7] R. Giegerich, S. Kurtz, and G. Weiller. An algebraic dynamic programming approach to the analysis of recombinant DNA sequences. In *Proc. WAAAPL*, pages 77–88, 1999.

[8] R. Giegerich, C. Meyer, and P. Steffen. A discipline of dynamic programming over sequence data. *SCICO*, 51(3):215–263, 2004.

[9] R. Giegerich and P. Steffen. Implementing algebraic dynamic programming in the functional and the imperative programming paradigm. In E. Boiten and B. Möller, editors, *Mathematics of Program Construction*, pages 1–20. LNCS 2386, 2002.

[10] R. Giegerich, B. Voß, and M. Rehmsmeier. Abstract shapes of RNA. *Nucleic Acids Research*, 32(16):4843, 2004.

[11] I. L. Hofacker, W. Fontana, P. F. Stadler, L. S. Bonhoeffer, M. Tacker, and P. Schuster. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie*, 125(2):167–188, 1994.

[12] I. Holmes, S. Slater, E. Birney, and G. M. Rubin. Telegraph: A new dynamic programming library. In *ISCB*, 2000.

[13] A. Morihata. A short cut to optimal sequences. *New Generation Computing*, accepted, 2010.

[14] T. L. Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 86:665–674, 1982.

[15] G. Rote. Path problems in graphs, Manuscript, 1989.

[16] G. Sauthoff. *Bellman's GAP: A 2nd Generation Language and System for Algebraic Dynamic Programming*. PhD thesis, Bielefeld University, 2010.

[17] G. Sauthoff and R. Giegerich. Bellman's GAP language report. Technical report, Bielefeld University, 2010.

[18] P. Steffen and R. Giegerich. Versatile and declarative dynamic programming using pair algebras. *BMC Bioinf.*, 6(1):224, 2005.

[19] P. Steffen and R. Giegerich. Table Design in Dynamic Programming. *Information and Computation*, 204(9):1325–1345, 2006.

[20] K. Swadi, W. Taha, and O. Kiselyov. Staging dynamic programming algorithms. In *Proc. of the 10th ACM SIGPLAN ICFP*, 2005.