Science of Computer Programming 87 (2014) 2–22

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Yield grammar analysis and product optimization in a domain-specific language for dynamic programming

Georg Sauthoff [a], Robert Giegerich [b,*]

[a] Universität Bielefeld, Technische Fakultät, 33501 Bielefeld, Germany
[b] Universität Bielefeld, Technische Fakultät and Center of Biotechnology, 33501 Bielefeld, Germany

ARTICLE INFO

Article history:
Received 10 January 2012
Received in revised form 4 March 2013
Accepted 27 June 2013
Available online 1 October 2013

Keywords:
Algebraic dynamic programming
Domain-specific language
Regular tree grammar
Compiler optimization
RNA structure prediction

ABSTRACT

Dynamic programming algorithms are traditionally expressed by a set of matrix 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, domain-specific language, which supports dynamic programming over sequence data. It implements the *algebraic* style of dynamic programming and allows one to specify algorithms by combining so-called yield grammars with evaluation algebras. Products on algebras allow to create novel types of analyses 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 and the evaluation algebras is required for generating efficient imperative code from the algebraic specification. This article gives an overview of the analyses required and presents several of them in detail. Measurements with "real-world" applications demonstrate the quality of the code produced.

1. Introduction

1.1. Challenges in the design of dynamic programming algorithms

Dynamic programming (DP) is a well-established programming technique, applicable to a wide class of combinatorial optimization or enumeration problems. Its roots go back to the work of Richard Bellman [1], in the days where a mathematical "program" was still executed by a human. Today, it is a subject of many computer science textbooks. Educational examples, found in many introductory texts, are string edit distance, matrix chain multiplication, knapsack problems, binary search trees with optimal expected access time, El Mamun's caravan, or Fibonacci numbers. Due to their simplicity, these example problems can be well explained in a monolithic fashion. 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. In the implementation, it is necessary to tabulate solutions of subproblem for later re-use, and the top-down recursion in the logic of problem decomposition gives way to a bottom-up computation governed by for-loops. Backtracing needs to be coded, if the solution candidate behind the optimal score is also of interest. A discussion of Bellman's Principle of Optimality informally describes the scope of problems that can be handled with dynamic programming.

Such simplicity is deceiving for several reasons:

* Corresponding author.
E-mail addresses: gsauthof@techfak.uni-bielefeld.de (G. Sauthoff), robert@techfak.uni-bielefeld.de (R. Giegerich).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.09.011

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. Tabulation for all of these recurrences is prohibitive for space reasons, and the question arises which ones need (not) to be tabulated, while retaining optimal asymptotic efficiency. Therein lures an NP-complete problem [2] that is not visible in the study of small textbook examples.
2. Often, the same search space is to be analyzed under different scoring schemes or objective functions, or the same scoring is to be applied to a modified search space. When encoded in the traditional for-loops, the tight entanglement of search space construction and candidate evaluation is a major obstacle to programmer productivity.
3. Debugging implementations of dynamic programming recurrences, where the case analysis is encoded in dozens of subscript calculations, is a Sisyphean task. A trivial error of (say) $(i + 1, j)$ instead of $(i, j - 1)$ may lead to a wrong (suboptimal) answer once in a while, and is likely to go unnoticed for a long time.
4. The code to backtrace the solution behind an optimal score is tedious to produce and to debug. Even more so, when all co-optimal or certain near-optimal solutions (up to a threshold) are to be generated.
5. When working with stochastic models such as Hidden Markov Models (HMMs) or Stochastic Context-Free Grammars (SCFGs) [3], which are implemented via dynamic programming, it must be secured that the case analysis covers the search space in a non-ambiguous fashion. This problem is formally undecidable [4], and without supporting "grammar-ware", this property is difficult to establish for the algorithm designer.

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

1.2. Dynamic programming in biosequence analysis

Bioinformatics is a field where dynamic programming is ubiquitous, models are sophisticated and data size is large. A central problem approached with dynamic programming is the comparison of protein, DNA and RNA sequences. This problem comes in many variants (global versus local alignment under different scoring schemes and gap models), aligning a sequence to a profile, predicting intron/exon structure of genes, evaluation of data from mass spectrometry, reconstructing the duplication history of repetitive sequences, so-called minisatellites, and many more. Families of protein sequences or structural (non-coding) RNAs are modelled via HMMs and SCFGs, respectively. Our own interest in dynamic programming comes from the area of RNA secondary structure prediction, where we have created a number of state-of-the-art tools [5], which we will use for efficiency measurements in this study.

1.3. Previous approaches to support dynamic programming

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 [6] models recurrences as state transitions with associated scoring functions. Multiple recurrences lead to several sub-fields of a state. C-code is generated automatically from state transition tables.

A follow-up effort was the *Telegraph* system [7], providing an object-oriented template library for dynamic programming, striving for greater flexibility. It included some features inspired from functional programming, such as model parameters bound to user-specified functions. However, these ideas have not reached a stable state of implementation. A purely functional approach with similar goals led to a combinator language for expressing dynamic programming algorithms [8], which later matured to the algebraic approach described below.

Dynamic programming (for a single recurrence) has been cast as a relational calculus by Bird and de Moor [9]. Staging DP [10] provides another combinator library to support dynamic programming. Using ingredients of dynamic programming, Morihata recently studied the automated derivation of efficient algorithms from naïve enumerate-and-choose-style specifications [11].

In the above settings, the problem decomposition for an input sequence can be seen as a relatively simple, context-free parsing problem. In the natural language processing community, there is strong interest in the development of more sophisticated parsing algorithms for more general classes of grammars. This is supported by the Dyna language [12], which was originally based on a forward-chaining scheme in the style of Datalog, allowing the programmer to concentrate on the logic of the parsing algorithms, rather than their implementation. A recent upgrade of the Dyna language goes far beyond this, and we will return to it below.

In all of the above cases, unfortunately, it must be said that these approaches have worked well in the hands of their creators, but the gain in abstractness does not appear large enough for others to justify the set-up cost of adopting a novel approach, and these approaches have not found more widespread use.

The discipline of *algebraic dynamic programming* (ADP) [13] is a language-independent formalism, which evolved from the *Haskell*-based combinator language in [8]. 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. Its first implementation was a domain-specific language embedded in *Haskell*. For efficiency reasons, a compiler translating *Haskell*-embedded ADP code to C, was developed [14,15], with more and more optimizations gradually added.