`ADPfusion`
Efficient Dynamic Programming over Sequence Data

Christian Höner zu Siederdissen [*]

Institute for Theoretical Chemistry
University of Vienna
A-1090 Vienna, Austria

June 10, 2013

Abstract

`ADPfusion` combines the usual high-level, terse notation of Haskell with an underlying fusion framework. The result is a parsing library that allows the user to write algorithms in a style very close to the notation used in formal languages and reap the performance benefits of automatic program fusion.

Recent developments in natural language processing and computational biology have lead to a number of works that implement algorithms that process more than one input at the same time. We provide an extension of `ADPfusion` that works on extended index spaces and multiple input sequences, thereby increasing the number of algorithms that are amenable to implementation in our framework.

This allows us to implement even complex algorithms with a minimum of overhead, while enjoying all the guarantees that algebraic dynamic programming provides to the user.

Keywords: dynamic programming, regular grammars, context-free grammars, multi-tape grammars, Haskell

1 Introduction

Dynamic programming over sequence is pervasive in many fields related to computer science. A major part of a compiler is a parser which takes an input source file and parses it according to the rule of the programming language. The core of a typical programming language can be described with a context-free grammar (Aho et al., 2007). Computational linguists analyze natural languages with, among other things, stochastic extensions to context-free grammars (Jurafsky et al., 1995). In computational biology there has been a trend towards writing algorithms that analyze protein, desoxy- (DNA), and ribonucleic acid (RNA) sequences with (stochastic) context-free grammars as well. One reason for this trend in computational biology was the insight that many different algorithms can be recast in one formal framework in this way. The formal framework of course being the hierarchy of formal languages devised by Chomsky (1956, 1959).

With the recognition of formal grammars as a way to model algorithmic problems more precisely and less ad-hoc came the need for software frameworks and libraries that lift formal

[*] `choener@tbi.univie.ac.at`

grammars from being a *mathematical device* to an actual tool at the disposal of the programmer. A number of different approaches to this problem have been proposed.

There exist highly specialized frameworks (framework and library is used more or less synonymously here) that target "just" the design of algorithms for the prediction of RNA secondary structure without pseudoknots. `TORNADO` (Rivas et al., 2012) is such a framework and has been used to explore the behaviour of different grammars for RNA secondary structure folding.

Other libraries are designed around more general parsing goals. The `GAP` language (Sauthoff et al., 2011, 2013) comes with its own compiler (`GAP-C`) and allows for a terse high-level description of algorithms.

Parsing in Haskell has a long and rich history. Functional languages are well-suited for parsing and embedding new domain-specific languages. These properties of the language were already used with the original `ADP` work (Giegerich and Meyer, 2002). In `ADP` atomic parsers for, say, single characters, variable-sized regions of the sequence, and non-terminal symbols are combined using combinator symbols that indicate size constraints for the atomic parsers they combine.

If we didn't care about performance our work would actually be done at this point. Unfortunately, however, parsing in this fashion carries a high overhead. The internal administrative expenses for this combinator-style of parsing lead to slow-downs of around $\times 60$–$\times 100$ compared to direct implementations in `C`. In the following sections we present, with `ADPfusion` (Höner zu Siederdissen, 2012), a framework that provides a language similar to `ADP`, yet has much better performance properties and comes very close to `C`-like performance for real-life uses.

2 Algebraic Dynamic Programming and Fusion

In `ADPfusion` we separate the program logic into several distinct parts. Following the original `ADP` work, a dynamic program consists of a signature, a grammar, and several algebras. In addition, we include functions for memoization of sub-results, and extended index spaces.

Below, we will make use of a simple `ADPfusion` program to detect maximal palindromes in sequences of text. This program is actually somewhat similar, though extremely simplified, to the biological problem of finding the minimum-free energy structure (Lorenz et al., 2011; Höner zu Siederdissen, 2012), but in this way we hope to more explicitly convey the point that `ADPfusion` is not restricted to biological problems.

Maximal palindromes

Given a sequence $x = x_1 \dots x_n$, a maximal palindrome on x is a sub-string $p = x_k \dots x_{k+l}$ such that $x_k = x_{k+l} \ \wedge \ x_{k+1} = x_{k+l-1} \ \wedge \cdots \wedge \ x_{k+\lfloor l/2 \rfloor} = x_{k+\lceil l/2 \rceil}$, i.e. the left and the right part of the sub-string are reverse images of each other. In `GATTACA`, the maximal palindrome is `.ATTA...`

Signatures

Signatures in `ADP` and `ADPfusion` contain the set of function symbols that are required by the grammar and need to be implemented by each algebra. Each production rule in a grammar requires one function symbol that is used to evaluate the result of applying a production rule to some input sequence, though in practice several production rules may share a function symbol if they parse equivalent objects.

The signature for the maximal palindromes example is:

Listing 1: The Palindrome Signature

```
type Signature m a r =
( ()                  → a    -- empty sub-string (also a palindrome)
, Char → a → Char → a    -- palindromic characters
, Char → a          → a    -- remove character on the left
,        a → Char → a    -- remove character on the right
, Stream m a          → m r  -- a stream (list) of candidate palindromes is reduced
)
```

Strictly speaking, the signature could be inferred automatically by the Compiler, but it is good style to annotate both grammar and algebra(s) with an explicit type annotation – and the automatically inferred type signatures are hilariously[1] unwieldy.

Grammars

Grammars describe the structure of the search space. An input sequence is decomposed according to the rules of the grammar. Each of the decompositions can then be scored with an algebra. This is also where the usual understanding of parsing in compilers, computational biology, linguistics, and other areas differs.

A compiler that parses an input file should create a single correct parse. It could return no parse in case of malformed input, but more than one correct parse is most likely a compiler error.

The assignment of an alignment between two protein sequences, the (secondary) structure of an RNA, or the most likely translation of a sentence in English to a sentence in Mandarin, however, has no single correct answer. Well, it probably has, but it can only be inferred with some probability or a number of more or less correct answers can be sorted by some score.

This means that parsing an input in any of the latter cases yields a, typically very large, set of “candidate answers” which need to be scored. This is why individual answers in the grammar below are immediately evaluated using one of the functions provided by the signature.

The palindrome grammar therefore requires us to bind an algebra (yet to be explained) for production rule evaluation, as well as a non-terminal symbol (`p` for palindrome), and two terminal symbols, one for single characters (`c`), bound to the input sequence, and one for empty sub-strings (`e`). Each non-terminal is paired with the set of production rules that form it’s left-hand side. Individual production rules are combined using an operator (`(|||)`) that concatenates different parses. The optimal parse (or parses) is determined with the help of an objective function (`h`) supplied by the algebra which reduces the set of parses (and is applied with the `(...)` operator). As Haskell functions can not be multi-variadic (but consider the neat trick used by `Text.Printf`[2]) terminal and non-terminal symbols are first packed into an inductive tuple (a Haskell data type of the form `(a:!:b)`). The inductive tuple form allows the `ADPfusion` framework to use the internal `mkStream` function which provides all parses of the right-hand side of a production rule in a readily fusible form (Coutts et al., 2007; Leshchinskiy, 2009; Höner zu Siederdissen, 2012). Once this fusible stream of parses is available it is relatively straightforward to map the algebra functions over the parses and finally reduce the stream using the objective function.

[1] `gNussinov` has an automatically inferred signature here:
`http://hackage.haskell.org/packages/archive/Nussinov78/0.1.0.0/doc/html/BioInf-GAPlike.html`

[2] `http://hackage.haskell.org/packages/archive/base/latest/doc/html/src/Text-Printf.html`

Listing 2: The Palindrome Grammar

```
grammar (empty, palindromic, left, right, h) p c e =
  ( p, empty       <<< e           |||
       palindromic <<< c % p % c   |||
       left        <<< c % p       |||
       right       <<<     p % c   ... h
  )
```

Algebras

In the traditional creation of a dynamic program, the exploration of the search space and evaluation and scoring of elements of the search space are intertwined. Small dynamic programs are sufficiently simple that they can be readily understood – our palindrome example would probably be easy enough. The complexity of dynamic programs does however increase rapidly. RNA secondary structure prediction alone (Zuker and Stiegler, 1981; Lorenz et al., 2011) makes use of around 20 scoring tables in addition to the dynamic programming tables.

This increase in complexity is countered in `ADP` and `ADPfusion` (and other frameworks) by factoring out the scoring scheme. Taken separately, each scoring function is simple enough that it can be readily understood, and maybe more importantly, easily tested for correctness. Once an algebra of functions has been defined (like the one below for palindromes), a complete algorithm is created by simple function application: `grammar maxPalindrome` is the dynamic programming algorithm for finding the palindrome of maximal size in the input. The algorithm still needs to be bound to the input and requires memoization but these two tasks can be automated and are not complicated.

Thanks to fusion, the combination of grammar and algebra will be turned into efficient, effectively intertwined, code – without having to do this part manually, and without the potential to introduce bugs.

The `maxPalindrome` algebra is rather simple. Empty sub-strings get a score of 0. A potential `palindromic` string is shortened by one character on the left (`l`) and one character on the right (`r`) and the inner part (`p`) was already scored (in principle recursively). If the two outer characters are indeed equal, we have a potential palindrome, otherwise the resulting score is $-\infty$. In order to detect palindromes that do not start at the beginning and end of the sequence, we also allow removing individual characters on the `left` and `right` to access sub-strings. The objective function returns just the maximal score which is equivalent to the longest palindrome yet found.

Listing 3: The Maximal Palindromes Algebra

```
maxPalindrome = (empty, palindromic, left, right, h) where
  empty       ()    = 0
  palindromic l p r = if l == r then p+1 else −∞
  left        l p   = p
  right         p r = p
  h           xs    = Stream.foldl' max 0
```

Memoization

Memoization is the act of taking a function f and its argument x and storing x together with $f(x)$. A memoization function stores maps of the form $x \to f(x)$. This concept is the basis of turning an exponential-time algorithm into a polynomial time algorithm, assuming that Bellman (1952)'s *Principle of Optimality* holds. `ADPfusion` performs memoization using efficient dynamic programming arrays that, at the same time, act as non-terminal symbols in grammars. By this virtue, every non-terminal is memoized (though this is not required and for some algorithms not recommended).

A nonterminal (`nt`) together with its production rules (`f`), which have been fused into efficient loops, is memoized using the `fillTable` functions.

Listing 4: Filling Memoization Tables

```
fillTable (nt,f) = let (_,n) = boundsM nt in
  forM_ [n,n-1..0] $ λi → forM_ [i..n] $ λj → do
    v ← f (i,j)
    writeM nt (i,j) v
```

Putting Everything Together

The author of an `ADPfusion` dynamic program first defines a signature which states how each right-hand side of a production rule looks like. This is equivalent to defining the type of all algebra functions for scoring candidates.

This is followed by the definition of the actual grammar. The grammar defines the search space. Individual productions rules take a sub-string and parse it depending on the terminals and non-terminals that make up each rule. Each parse is immediately scored using one of the function symbols made available by the signature and implemented using an algebra.

Each algebra provides different ways of scoring parses. The scoring scheme does not have to be an actual scoring scheme but can also be a pretty-printing device for backtracing purposes. Quite often, we are not only interested in some optimal score but also the parse or parses that led to this score.

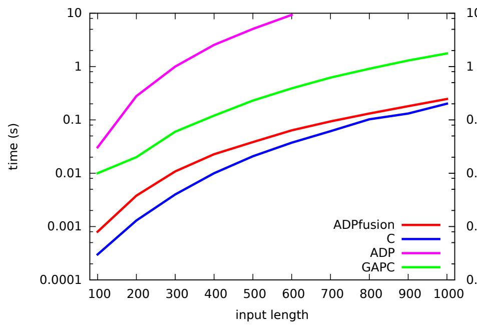
Backtracing works by taking a scoring algebra and combining it with a pretty-printing algebra using an algebra product operation. The algebra product operation provides the necessary machinery for backtracing together with a non-terminal "wrapper". This more involved way of backtracing is necessary as we do not want to store all possible backtraces during the forward phase of the algorithm which would be quite costly.

It is however still possible to combine different algebras in the forward phase. Algebra combinations during the forward phase can be used to implement variants of classified dynamic programming, where parts of the search space may be pruned, or otherwise be handled differently. A number of algorithms exist where classified dynamic programming is required (Reeder and Giegerich, 2005; Huang et al., 2012).

One then finally combines the different parts of the algorithm, by binding the input, and filling the non-terminal memoization tables. Oftentimes, a backtracing phase follows the forward calculation of the optimal score and the backtraces are returned together with the optimal score.

Performance

We now leave the palindrome example behind. It lacks some properties which make parsing a context-free language interesting. We can, of course, for a moment consider what would be required of our grammar to include such a property. Let's say that we are interested in the sum of the lengths of all palindromes that can be found in a string. Given `GATTACAGATTACA`, we'd have `.ATTA...ATTA..` with a score of $2 + 2 = 4$. A production rule to find all local palindromes and correctly sum up their scores is easily written:



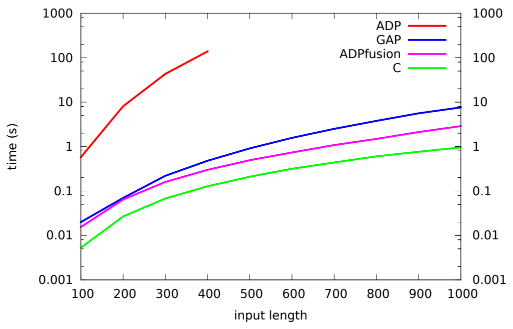


Figure 1: **Left:** The `Nussinov78` algorithm as implemented in different dynamic programming frameworks and directly in `C`. `ADPfusion` comes second only to the direct `C` implementation. **Right:** This trend continues for the more complex `ViennaRNA RNAfold` algorithm.

Listing 5: All Local Palindromes

```
grammar (empty, palindromic, left, right, h) p c e =
  ( p, ...
      right     <<<   p % c  |||
      locals    <<<   p % p  ... h
  )
```

The `locals` rule combines all possible parses of the sub-string $x_i \dots x_j$ at a split point k such that we get two sub-strings $x_i \dots x_k$ and $x_{k+1} \dots x_j$. This new grammar has a runtime of $O(n^3)$ and is also *very* similar to the `Nussinov78` (Nussinov et al., 1978) RNA secondary structure prediction algorithm. We use this algorithm as a testbed to experiment with new features for three reasons. First, the algorithm is very simple, yet already has all the grammatical features other secondary structure prediction algorithms have. Second, implementations in different languages are readily available. Third, runtime measurements directly give us feedback on the performance of the generated loops as there is only one non-terminal which needs to be memoized and no external scoring tables or otherwise costly or complicated functions need to be called. So while being a bit simplistic, it is perfect as a testbed.

We shall now give a figure with run times for the `Nussinov78` and the `ViennaRNA RNAfold` (Lorenz et al., 2011) algorithm as implemented in `ADPfusion`, `C`, `ADP`, and the `GAP` language. Fig. 1 shows that while `ADPfusion` can not (yet) beat `C`, we do at least come close and outperform other frameworks.

3 Extensions: Extended Index Spaces, Multiple Inputs

In this section we give a short overview over two new features that have recently become available in `ADPfusion` that increase the number of problems that can be solved using `ADPfusion`.

Extended Index Spaces

One of the long-standing goals of `ADP` and `ADPfusion` is to hide actual index calculations from the programmer. Explicit handling of indices may easily introduce bugs and as such indices should only be touched when writing a new terminal parser. Why then discuss index spaces? The reason is that we do not want to be bound by the $\mathbb{Z}^2$ index space for sub-strings on sequences where each pair (i, j), $i \leq j$ defines a particular sub-string.

Left- or right-linear grammars require only the space $\mathbb{Z}$ and the memoization tables can accordingly be made much smaller. Even the $\mathbb{Z}^2$ index space is "too large". All legal sub-strings can be memoized using an upper-triangular matrix and we would like to be able to make this `Subword` index type explicit.

Yet other algorithms use more complicated index spaces. The `Infernal` suite of programs (Nawrocki et al., 2009) creates and uses *covariance models*, stochastic context-free grammars with *hundreds* of non-terminals. The index space here is best encoded as `Subword` $\times$ `NT-index` and two coupled context-free grammars could well use the $\mathbb{Z}^{2\times 2}$ index space.

Accordingly, the internal machinery for `ADPfusion` is now parametrized over all possible index spaces. While this requires us to provide new terminal parsers for each new index space, this is less problematic in practice as Haskell is expressive enough to only require us to provide implementations for each "one-dimensional" case. The higher-dimensional cases are available via an implementation of an inductive tuple for the multi-dimensional case.

Multiple Inputs

Once index spaces for multi-dimensional parsing are available, multiple inputs are available as well. Just as multi-dimensional indices can be defined inductively, so can terminal parsers be defined. This automatically allows us to provide all the machinery that is needed for alignment-style algorithms.

4 Conclusion

`ADPfusion` allows us to write parsers for context-free grammars that are efficient enough to come close to hand-optimised implementations in `C`. There is a certain amount of additional "complexity" in setting up the signature, grammar, and algebras. This upfront cost is, however, soon repaid by (i) much easier maintenance, (ii) extensibility, (iii) performance, (iv) absence of hard-to-catch bugs, and (v) scalability to problems that require more complex grammars.

The `ADPfusion` work has until now focused much on performance. This was mainly to be able to write algorithms for large-scale applications instead of having to resort to `C` once the prototype stage is left behind. The framework has now reached a stage of maturity where it has become possible to extend the number of algorithms that can be described using our language.

Our main aim is to allow our users to develop complex algorithms on single and multiple sequence inputs and let them focus on the algorithmic problem, *not low-level issues*. This is especially important as algorithms become more complex once more than one input sequence needs to be considered. While the alignment of two sequences (Needleman and Wunsch, 1970) is now taught in introductory courses on computational biology – and computational linguists probably have their version of the algorithm – problems on more than one sequence are hard to solve and hard to implement correctly.

There has been increased interest in such multi-input problems in both, computational biology and linguistics recently. Huang et al. (2009); Chitsaz et al. (2009) considered the problem of two interacting RNA sequences. Both algorithms have extremely complex grammars. In linguistics a similar problem deals with aligning sentences in two different human languages (say English and Mandarin). It is in general not possible to align words on a one-to-one basis as structural properties of the different languages mean that completely different sentence structures need to be considered. This problem of weak synchronization was considered by Burkett et al. (2010).

The extended `ADPfusion` framework is now powerful enough to handle parsing problems encoded as regular or context-free grammars and involving one or multiple sequences. Upcoming work will include `SIMD` instructions using generalized fusion (Mainland et al., 2013) and an extension to multi-threading based on Repa (Keller et al., 2010). This allows the user of `ADPfusion` to tackle complex problems on sequences with better insurance that the resulting algorithms are correct while enjoying high performance.

Acknowledgments

The author thanks Johannes Waldmann and Peter F. Stadler for their kind invitation to `HaL 8` in Leipzig. Aspects of this work have been discussed with Robert Giegerich, Ivo L. Hofacker, and Peter F. Stadler on many occasions.

References

Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson/Addison Wesley, 2007.

Richard E. Bellman. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences*, 38 (8):716–719, 1952.

David Burkett, John Blitzer, and Dan Klein. Joint Parsing and Alignment with Weakly Synchronized Grammars. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 127–135. Association for Computational Linguistics, 2010.

Hamidreza Chitsaz, Raheleh Salari, S Cenk Sahinalp, and Rolf Backofen. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, 25(12):i365–i373, 2009.

Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

Noam Chomsky. On Certain Formal Properties of Grammars. *Information and control*, 2(2):137–167, 1959.

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 315–326. ACM, 2007.

Robert Giegerich and Carsten Meyer. Algebraic Dynamic Programming. In *Algebraic Methodology And Software Technology*, volume 2422, pages 243–257. Springer, 2002.

Christian Höner zu Siederdissen. Sneaking Around concatMap: Efficient Combinators for Dynamic Programming. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 215–226, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364559. URL `http://doi.acm.org/10.1145/2364527.2364559`.

Fenix W.D. Huang, Jin Qin, Christian M. Reidys, and Peter F. Stadler. Partition function and base pairing probabilities for RNA–RNA interaction prediction. *Bioinformatics*, 25(20):2646–2654, 2009.

Jiabin Huang, Rolf Backofen, and Björn Voß. Abstract folding space analysis based on helices. *RNA*, 18(12):2135–2147, 2012.

Daniel Jurafsky, Chuck Wooters, Jonathan Segal, Andreas Stolcke, Eric Fosler, G Tajchaman, and Nelson Morgan. Using a stochastic context-free grammar as a language model for speech recognition. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 189–192. IEEE, 1995.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP'10, pages 261–272. ACM, 2010.

Roman Leshchinskiy. Recycle Your Arrays! *Practical Aspects of Declarative Languages*, pages 209–223, 2009.

Ronny Lorenz, Stephan H. Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F. Stadler, and Ivo L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(26), 2011.

Geoffrey Mainland, Roman Leshchinskiy, Simon Peyton Jones, and Simon Marlow. Haskell Beats C Using Generalized Stream Fusion. 2013.

Eric P. Nawrocki, Diana L. Kolbe, and Sean R. Eddy. Infernal 1.0: inference of RNA alignments. *Bioinformatics*, 25 (10):1335–1337, 2009.

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(3):443–453, 1970.

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.

Jens Reeder and Robert Giegerich. Consensus shapes: an alternative to the Sankoff algorithm for RNA consensus structure prediction. *Bioinformatics*, 21(17):3516–3523, 2005.

Elena Rivas, Raymond Lang, and Sean R. Eddy. A range of complex probabilistic models for RNA secondary structure prediction that includes the nearest-neighbor model and more. *RNA*, 18(2):193–212, 2012.

Georg Sauthoff, Stefan Janssen, and Robert Giegerich. Bellman's GAP - A Declarative Language for Dynamic Programming. In *Proceedings of the 13th international ACM SIGPLAN symposium on Principles and practices of declarative programming*, PPDP'11, pages 29–40. ACM, 2011.

Georg Sauthoff, Mathias Möhl, Stefan Janssen, and Robert Giegerich. Bellman's GAP – a Language and Compiler for Dynamic Programming in Sequence Analysis. *Bioinformatics*, 2013.

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.