IN PROCEEDINGS OF WORKSHOP ON ALGORITHMIC ASCPECTS
OF ADVANCED PROGRAMMING LANGUAGES (WAAAPL), PARIS,
SEPTEMBER 1999

An Algebraic Dynamic Programming Approach to the Analysis of Recombinant DNA Sequences

Robert Giegerich[*] *Stefan Kurtz*[†] *Georg F. Weiller*[‡]

1 Introduction

1.1 From Biosequences to Structure to Function

Dynamic programming (DP, for short) is a fundamental programming technique, applicable to great advantage whenever the input to a problem spawns an exponential search space in a structurally recursive fashion, and solutions to subproblems adhere to an optimality principle. No wonder that DP is the predominant paradigm in computational (molecular) biology. Sequence data—DNA, RNA, and proteins—are determined on an industrial scale today. The desire to give a meaning to these molecular data gives rise to an ever increasing number of sequence analysis tasks. Given the mass of these data and the length of these sequences ($3 \cdot 10^6$ bases for a bacterial genome, $3 \cdot 10^9$ for the human genome), program efficiency is crucial. DP is used for assembling DNA sequence data from the fragments that are delivered by the automated sequencing machines [1], and to determine the intron/exon structure of genes [3]. It is used to infer function of proteins by homology to other proteins with known function [10, 11], and to determine the secondary structure of functional RNA genes or regulatory elements [15]. In some areas, DP problems arise in such variety that a specific code generation system for implementing the typical DP recurrences has been developed [2]. This system, however, does not support the development or validation of these recurrences.

1.2 Outline of Algebraic Dynamic Programming

The systematic development of DP solutions for problems in computational biology has been recently addressed by Giegerich [4]. There, an algebraic approach to dynamic programming (ADP) was developed and applied to the problem of folding an RNA sequence into its secondary structure. Here we will adapt ADP to the problem of comparing *two* sequences in the edit distance model. ADP is based on the following principles:

1. The analysis problem at hand is conceptually split into a *structure recognition* and a *structure evaluation* phase. Recognized structures are represented by an algebraic datatype $\mathcal{S}$. Evaluation is specified in terms of a particular $\mathcal{S}$-algebra.

2. A subset of well-formed structures in $\mathcal{S}$ is distinguished by a *tree grammar*. We require that

[*]Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany, E-mail: robert@techfak.uni-bielefeld.de, partially supported by a grant from the Australian National University

[†]Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany, E-mail: kurtz@techfak.uni-bielefeld.de, partially supported by DFG-grant Ku 1257/1-1

[‡]Bioinformatics Laboratory, Research School of Biological Science, Australian National University, Canberra, ACT 0200, Australia, E-mail: weiller@rsbs.anu.edu.au

- structure recognition finds *all and only all* well-formed structures,
- structure recognition constructs each such structure exactly *once*,
- structure evaluation is performed only on well-formed structures.

3. By providing *parsers* for the terminal symbols and *parser combinators* for the alternative, applicative, and sequential operators of the tree grammar, the grammar turns into a recognizer for its language.

4. A recursive recognizer is turned into a *DP algorithm* by *tabulation*: Each (recursive) parser is substituted by a (recursively defined) table of results. This is achieved by an efficiency annotation that does not change the declarative meaning of the grammar.

5. An *abstract evaluator* is a recognizer written in terms of an abstract $\mathcal{S}$-algebra, applying an abstract choice function to each intermediate result. Instantiated with a concrete $\mathcal{S}$-algebra, it interleaves structure recognition and evaluation. The concrete evaluator so obtained runs in polynomial time and space, if the concrete evaluation algebra has a constant time and space bound with respect to each intermediate result.[1]

6. *DP recurrences*, suitable for implementation in any imperative language, can be derived from the specification by straightforward substitution and program simplification.

1.3 Why Functional Programming Matters

ADP is a program development method, and the resulting program can (and normally will) eventually be implemented in an imperative language. A functional language like *Haskell*, however, makes the approach much more practical, and even enjoyable. The ADP approach can be completely embedded in *Haskell*, allowing us to experiment with executable programs at all stages of development. A wide range of lazy functional programming techniques is used, the most essential being parser combinators [9], programming with unknowns, and lazy (though immutable) arrays.

The productivity of the approach results from the modularity (cf. [8]) we achieve by separating structure recognition from structure evaluation. This advantage only exists in the functional paradigm; it is sacrificed in the final step (see Section 1.2, Principle 6).

Although ADP is a program development method, and not an equivalence transformation on programs, it bears some resemblance to deforestation [13], particularly in the form of [5]. The essential speed-up from exponential to polynomial time complexity, however, is not achieved by deforestation, but by tabulation and the simultaneous introduction of a choice function that reduces the volume of the intermediate results.

2 Biosequence Comparison in the Edit Distance Model

2.1 Searching for the Signals of Recombination

Comparison of DNA or protein sequences is predominantly done in the edit distance model. Two or more sequences are rearranged by introducing gaps, in a way that best exhibits their (dis)similarities. The concrete way in which distance or similarity is measured is expressed by means of a scoring function for matches, mismatches, and gaps. The scoring function varies from application to application. Sequence similarity is taken as an indication of homology, and multiple alignments or pairwise distances so obtained are frequently fed into programs that try to reconstruct phylogenies, i.e., evolutionary relationships of genes or species.

[1]More precisely, all operations of the algebra may be allowed to have polynomial efficiency, but the choice function is critical and must have a constant bound on the size of its output.

DNA *recombination* is an important mechanism in molecular evolution. Genes that have evolved independently in different strains of a virus, for example, may recombine in a new strain. This adds the power of parallel processing to Darwinian evolution, which is otherwise based on trial and error (i.e., random mutation and selection). In the presence of recombinant DNA, practically all commonly used analysis programs go wrong. There is no longer a tree-like phylogeny, as different parts of a sequence stem from different ancestors. In such a case, the best we can hope for from a tree reconstruction program is to tell us that there is no clear support for either of several possible trees in the distance data.

But there is a difference between data which are just noisy, and data which carry a clear signal about recombination events. There are different ways to explicitly search for recombination signals. The *PhylPro* program [14] does so by monitoring patterns of change in the mutual similarities in a multiple sequence alignment. In this paper, we take a direct approach, applicable to pairwise sequence alignment.

Traditionally, insertions and deletions are seen as random events, independent of their sequence context. But this is not totally adequate: Insertions and deletions in DNA sequences often stem from recombination events. The molecular mechanisms of recombination may leave traces in the form of target site duplications of varying length. Similar repeats may be formed through replication slippage, the other cellular process responsible for indel formation. Current methods of sequence analysis ignore these signals.

2.2 Extending the Edit Distance Model

Let x and y be two DNA sequences of length m and n, respectively. The classical edit distance model considers the following edit operations:

- $R\binom{a}{b}$ denotes the *replacement* of nucleotide a in x by b in y. If $a = b$, this is called a *match*, otherwise a *proper replacement*.
- $I\binom{-}{u}$ denotes the *insertion* of a non-empty sequence u of nucleotides into y, thereby introducing in x a gap of the same length, i.e., a sequence of $|u|$ dashes.
- $D\binom{u}{-}$ denotes the *deletion* of a non-empty sequence u of nucleotides from x, thereby introducing in y a gap of the same length, i.e., a sequence of $|u|$ dashes.

As new edit operations, we introduce recombinant deletion and insertion. Let t be a non-empty (but typically short) sequence of nucleotides that occurs both in x and in y.

- $S\binom{t - -}{t\,u\,t}$ denotes a *recombinant insertion* in y: Following the *target site* t, present in both x and y, a sequence u of nucleotides is inserted into y, followed by a new copy of t in y. In x, a gap of the combined length of u and t is introduced.
- $L\binom{t\,u\,t}{t - -}$ denotes a *recombinant deletion* from x: Following the *target site* t, present in both x and y, a sequence u of nucleotides is deleted from x. This requires a second copy of t to follow u in x. In y, a gap of the combined length of u and t is introduced.

In both cases, we allow the deleted or inserted sequence u to be empty, which makes the target site and its duplication form a tandem repeat in x or y.

Example 1 Here is an alignment of $attcgaa$ and $acgtatacgac$:

$$R\binom{a}{a}\ D\begin{pmatrix} t\ t \\ -\,- \end{pmatrix}\ S\begin{pmatrix} c\ g - - - - - - \\ c\ g\ t\ a\ t\ a\ c\ g \end{pmatrix}\ R\binom{a}{a}\ R\binom{a}{c}$$

It shows three replacements, a short deletion, and a recombinant insertion.

Proceeding from this operational view of recombination events to the analytic view, we must define the sequence pattern that can be interpreted in retrospect as a signal left from a recombination.

For any sequence z and any $i \in [0, |z|]$, $i{\downarrow}z$ denotes the suffix of z after dropping i symbols from the beginning of z. A *target site duplication in* y is a pair (i, j) such that $i{\downarrow}x = tz$ and $j{\downarrow}y = tutw$ for some t, u, w, z such that t is not empty. It is *maximal* if the first character of u, w, z is not the same, whenever these strings are not empty. A *target site duplication in* x is a pair (i, j) such that $i{\downarrow}x = tutw$ and $j{\downarrow}y = tz$ for some t, u, w, z such that t is not empty. It is *maximal* if the first character of u, w, z is not the same, whenever these strings are not empty.

Note that several target site duplications may be identified at the same position, differing in the length of the target sequence t. However, in the following we restrict to maximal duplications, since a longer target site duplication is to be taken as the stronger signal of a recombination event. We say that a recombinant deletion is *signalled* by a maximal target site duplication in x, and a recombinant insertion is *signalled* by a maximal target site duplication in y.

3 Computing Optimal Alignments in the Extended Edit Distance Model

3.1 An Algebraic Data Type for Extended Alignments

An alignment of x and y is traditionally represented by placing the aligned sequences on different lines, with inserted dashes to denote gaps. Successive dashes inside x are interpreted as an insertion into y, and successive dashes inside y as a deletion from x. The eye of the reader implicitly groups successive dashes into gaps of maximal length. A slightly more explicit view defines the alignment as a sequence of edit operations, with the additional restriction that a deletion (resp. insertion) must not immediately follow another deletion (resp. insertion).

With the new edit operations introduced here, we must resort to an even more explicit notation, marking target sites and their duplications. We also have to distinguish between gaps resulting from recombinations and gaps for which such an event is not indicated. We give up the view of a sequence of edit operations in favor of a recursive datatype `Alignment` with a constructor for each edit operation.

```
type Sequence a = Array Int a   -- indexed from 1
type Region = (Int,Int)         -- region $(i,j)$ of $x$ denotes $x_{i+1} \dots x_j$
data Alignment a = R a (Alignment a) a |
                   D Region (Alignment a) |
                   I (Alignment a) Region |
                   S Region (Alignment a) Region Region Region |
                   L Region Region Region (Alignment a) Region |
                   Empty
```

Within the datatype `Alignment`, a target site duplication $i{\downarrow}x = tz$, $j{\downarrow}y = tutw$, is represented by an expression of the form `S t azw t u t`, wherein `azw` denotes an alignment of the suffixes z and w, and the three occurrences of `t` denote the target site in x and (duplicated) in y. Subwords of x and y are represented by their boundaries. Hence each edit operation requires constant space. If $k = |t|$ and $r = |u|$, then the above expression is actually written as

```
S (i,i+k) azw (j+k+r,j+k+r+k) (j+k,j+k+r) (j,j+k).
```

More space efficient representations are possible, since we only need to store i, j, k, and r.

Example 2 Given a datatype `Base` with constants `A`, `C`, `G`, and `T`, the expression

```
R A (D (1,3) (S (3,5) (R A (R A Empty C) A) (7,9) (3,7) (1,3))) A
```

denotes the alignment of $attcgaa$ and $acgtatacgac$ shown in Example 1. It may be printed in ASCII as

```
x = a t t c g - - - - - - a a
y = a - - c g t a t a c g a c
    R D D S S U U U U T T R R
```

The third line indicates the edit operation involved. The recombinant insertion is labeled in the form `S U T` to indicate the starting target site `S`, the insert `U`, and the duplicated target site `T`.

At this point the reader is encouraged to take a look ahead at Section 4. It shows the improvement of standard alignment algorithms which we go for in the subsequent sections.

3.2 A Grammar for Well-Formed Alignments

The datatype `Alignment` is not specific enough to describe exactly all meaningful alignments. For example, it allows to represent two subsequent insertions, which should rather be merged into a single, longer insertion:

```
x = a t t c g - - - - - - - - a a                -- malformed
y = a - - c g t a t a c g g g a c
    R D D S S U U U U T T I I R R
```

We do not accept a non-recombinant insertion immediately following a recombinant insertion. (We do, however, accept the opposite order.) It seems accidental to locate a duplicated target site in the middle of a gap. In such a situation, the alignment should rather show a single (non-recombinant) insertion (left alignment). Alternatively we might call for a recombinant insertion with a shorter target site (right alignment).

```
x = a t t c g - - - - - - - - a a     x = a t t c g - - - - - - - - a a
y = a - - c g t a t a c g g g a c     y = a - - c g t a t a c g g g a c
    R D D R R I I I I I I I I R R         R D D R S U U U U U U U T R R
```

It will be the task of the scoring function to choose between the latter two alternatives, while the malformed alignment above will not even be scored.

We introduce a grammar generating exactly the well-formed alignments. Following the discipline of [4], we use a tree grammar over the datatype `Alignment`, see Figure 1. The terminal symbols of this grammar are *base*, *region*, *uregion*, denoting a single nucleotide, a non-empty and an arbitrary sequence of nucleotides, respectively. The nonterminals are *alignment*, *noDel*, *noIns*, and *match*.

A production in this notation should be read as: “An alignment is either a *match*, or alternatively a *deletion* of some region from x followed by a *noDel*, or alternatively an *insertion* of some *region* in y, followed by a *noIns*.”

As easily seen in the grammar *noDel* generates all alignments that do not start with a deletion. The use of *noDel* in the first production prevents successive deletions. Similarly for *noIns*. Leaving out the clauses for recombinant deletions and insertions, this tree grammar expresses the classical edit distance model [10, 11], used in biosequence analysis as well as in string processing.

The grammar still lacks some syntactic restriction: The three occurrences of *region* in the productions associated with `S` and `L` must all derive the same nucleotide sequence.

We now turn the grammar into a recognizer by defining terminal parsers and parser combinators [9]. For simplicity (and reasons of space) we assume that the input sequences x and y, as well as their length m and n are globally known. We do not show how these values are threaded through the functions.

A parser is given a pair of indices (i, j) and returns a list of all well-formed alignments of the suffix $i{\downarrow}x$ with the suffix $j{\downarrow}y$. Parsers for terminal symbols, however, are applied to one of the input sequences, so in their case, a call for (i, j) recognizes the subword (i, j) in either x or y. There is a parser combinator for

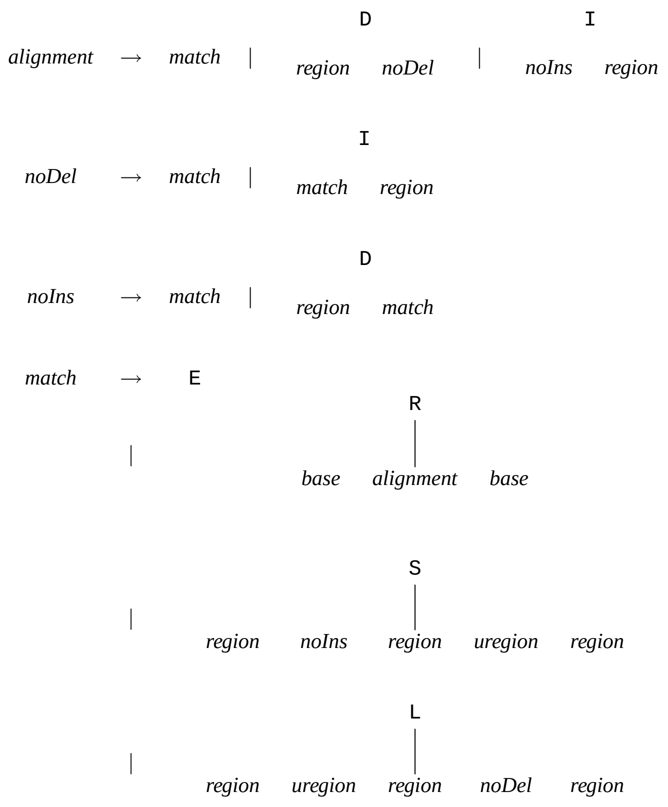
alignment → match | D(region noDel) | I(noIns region)
noDel → match | I(match region)
noIns → match | D(region match)
match → E
| base R(alignment) base
| region noIns S(region) uregion region
| region uregion L(region) noDel region

Figure 1: A tree grammar for well-formed alignments

the alternative, and for using parser results. Since we have two strings to process, there are two sequential combinators `+~~` and `~~+`. The combinators `-~~` and `~~-` are special forms of these.[2]

```
type Parser b = (Int,Int)->[b]               -- all parses of suffix pair

xbase,ybase::Parser a
xbase (i,j) = [x!j | i+1 == j]               -- recognize a base from x
ybase (i,j) = [y!j | i+1 == j]               -- recognize a base from y

region,uregion::Parser (Int,Int)
region  (i,j) = [(i,j) | i < j]              -- recognize a non-empty region
uregion (i,j) = [(i,j) | i <= j]             -- recognize any region

empty::b->(Parser b)
empty v (i,j) = [v | i == m && j == n]       -- recognize empty alignment

(|||)::(Parser b)->(Parser b)->(Parser b)
(|||) q r inp = q inp ++ r inp               -- alternative

(<<<)::(b->c)->(Parser b)->(Parser c)
(<<<) f q = map f.q                          -- using parser results

(+~~),(~~+),(-~~),(~~-)::(Parser (b->c))->(Parser b)->(Parser c)
(+~~) q r (i,j) = [s t | k<-[i..m], s<-q (i,k),   t<-r (k,j)  ]
(~~+) q r (i,j) = [s t | k<-[j..n], s<-q (i,k),   t<-r (j,k)  ]
(-~~) q r (i,j) = [s t | i < m,     s<-q (i,i+1), t<-r (i+1,j)]
(~~-) q r (i,j) = [s t | j < n,     s<-q (i,j+1), t<-r (j,j+1)]

suchthat::(Parser b)->(b->Bool)->(Parser b)
suchthat q f inp = [s | s <- q inp, f s] -- check property of parser results

axiom::((Int,Int)->b)->b
axiom q = q (0,0)                          -- declare start symbol of grammar
```

In the grammar, written as a recognizer, we add syntactic restrictions for maximal target site duplication to the corresponding productions.

[2] To explain our ideas it would sometimes suffice to present less *Haskell*-code. However, we want to make our paper self-contained and therefore show the code almost completely.

```
enum_alignments::(Eq a)=>(Sequence a)->(Sequence a)->[Alignment a]
enum_alignments x y = axiom alignment where

  alignment = match                               |||
              D <<< region +~~ noDel              |||
              I <<< noIns  ~~+ region

  noDel = match                                   |||
          I <<< match ~~+ region

  noIns = match                                   |||
          D <<< region +~~ match

  match = empty Empty                             |||
          R <<< xbase -~~ alignment ~~- ybase     |||
          recombIns                               |||
          recombDel

  recombIns = ((S <<< region +~~ noIns ~~+ region ~~+ uregion ~~+ region)
              `suchthat` targetsiteduplication)
              `suchthat` maximality

  recombDel = ((L <<< region +~~ uregion +~~ region +~~ noDel ~~+ region)
              `suchthat` targetsiteduplication)
              `suchthat` maximality
```

3.3 Dynamic Programming = Parsing + Tabulation

The above recognizer is easy to develop, but its associated parser is highly inefficient: Not only is there an exponential number of well-formed alignments for each pair of input sequences. The recognizer will also repeatedly parse the same subwords when called from different contexts.

The latter inefficiency is removed by introducing tabulation of intermediate parser results (representing alignments of suffixes of the two inputs). In other words, we employ DP. In contrast to memoization [7], DP uses explicitly and statically allocated tables.

```
type Parsetable b = Array (Int,Int) [b]

tabulated::Parser b->Parsetable b
tabulated q = array ((0,0),(m,n)) [((i,j),q (i,j)) | i<-[0..m],j<-[0..n]]
```

We modify the previous grammar, such that all parsers that do a non-constant amount of work per call shall use tabulation. Calling a parser means a table lookup. For reasons of space we only show the parser `alignment`. Note that our "efficiency annotation" does not affect the declarative meaning of the grammar.

```
dp_alignments::(Eq a)=>(Sequence a)->(Sequence a)->[Alignment a]
dp_alignments x y = axiom (alignment!) where

  alignment = tabulated (
              (match!)                   |||
              D <<< region +~~ (noDel!) |||
              I <<< (noIns!)  ~~+ region)
```

It is folklore knowledge that DP combines recursion and tabulation. After all, DP is normally formulated via matrix recurrences. The remarkable point here is the swiftness of transition, merely by adding the "keyword" `tabulated` and a few "`!`" to the grammar. The declarative and the operational meaning of the grammar remain unaffected, while efficiency improves from exponential to polynomial. If we had not been in love with *Haskell* before, this is where it would have happened.

The recognizer specified by this grammar runs in $O(n^2)$ space and in $O(n^6)$ time, due to the four sequential combinators in the productions associated with recombinant insertions and deletions.[3]

3.4 An $\mathbf{O(n^3)}$ Implementation Using a Precomputed Lookahead

The above parser independently chooses three regions for the target site in x, in y, and for the duplication site in either x or y. Thereafter, those are checked for identity. Its efficiency can be greatly improved by the following observation: Consider a maximal target site duplication $i{\downarrow}x = tz$, $j{\downarrow}y = tutw$. Assume we have chosen and fixed the *combined* length $h = |tu|$ of the target site t and the insert u. Now for given x and y, there is really no variation left for the remaining constituents of the pattern:

- The start positions of the identical subwords must be i, j, and $j + h$.
- Their lengths are uniquely determined by the maximality condition.

Thus we will modify the parser to guess the position $j+h$, and then use a precomputed table `lookahead` to determine the length of t. For each $(i, j) \in [0, m] \times [0, n]$ this table stores the length of the longest common prefix of $i{\downarrow}x$ and $j{\downarrow}y$. It is computed and stored in $O(n^2)$ time and space. The overall running time of the recognizer is reduced to $O(n^3)$, while the space requirement remains $O(n^2)$. Note that since the three sites are now chosen as identical subwords of maximal length, this approach obviates the a-posteriori check for these properties. The resulting grammar is very similar to the grammar `ab_alignments` given in Section 3.5.

3.5 The Abstract Evaluator and Evaluation Algebras

According to [4], an abstract evaluator is obtained by abstracting from the constructors of the underlying datatype `Alignment`. Additionally, an abstract choice function is associated with each production, by the combinator `(...)`. Such an ensemble of functions of appropriate types constitutes an alignment-algebra.

```
type Algebra a b
  = (b,                                                  -- Empty
     a->b->a->b,                                         -- R
     (Int,Int)->b->b,                                    -- D
     b->(Int,Int)->b,                                    -- I
     (Int,Int)->(Int,Int)->(Int,Int)->b->(Int,Int)->b,   -- L
     (Int,Int)->b->(Int,Int)->(Int,Int)->(Int,Int)->b,   -- S
     [b]->[b])                                           -- choice function

(...)::Parser b->([b]->c)->(Int,Int)->c
(...) q choice = choice.q                 -- applying a choice function
```

The abstract evaluator takes an alignment algebra as an additional parameter and adds the choice function.

[3]We generally assume that $m \in O(n)$, to simplify asymptotic results.

```
ab_alignments::(Eq a)=>(Algebra a b)->(Sequence a)->(Sequence a)->[b]
ab_alignments alg x y = axiom (alignment!) where

  (fE, fR, fD, fI, fL, fS, choice) = alg

  alignment = tabulated (
              (match!)                                    |||
              fD <<< region +~~ (noDel!)                  |||
              fI <<< (noIns!)  ~~+ region ... choice)

  noDel = tabulated (
          (match!)                                        |||
          fI <<< (match!) ~~+ region ... choice)

  noIns = tabulated (
          (match!)                                        |||
          fD <<< region +~~ (match!) ... choice)

  match = tabulated (
          empty fE                                        |||
          fR <<< xbase -~~ (alignment!) ~~- ybase         |||
          (recombIns!)                                    |||
          (recombDel!) ... choice)

  recombIns = tabulated (r ... choice) where
    r (i,j) = [fS t’ noins d u t | l <-[j+1..n-1],
                                    let k = min h (lookahead!(i,l)),
                                    t’<- region (i,i+k),
                                    noins <- noIns!(i+k,l+k),
                                    d <- region (l,l+k),
                                    u <- uregion (j+k,l),
                                    t <- region (j,j+k)]
               where h = lookahead!(i,j)

  recombDel = tabulated (r ... choice) where
    r (i,j) = [fL t u d nodel t’ | l <-[i+1..m-1],
                                    let k = min h (lookahead!(l,j)),
                                    t <- region (i,i+k),
                                    u <- uregion (i+k,l),
                                    d <- region (l,l+k),
                                    nodel <- noDel!(l+k,j+k),
                                    t’<- region (j,j+k)]
               where h = lookahead!(i,j)
```

The virtue of the abstract evaluator is, of course, that it can be called with arbitrary `Alignment` algebras: The *enumeration algebra* is trivially given by the constructors of the `Alignment` datatype.

```
enum_alg::Algebra a (Alignment a)
enum_alg = (Empty, R, D, I, L, S, id)
```

The *counting algebra* may be used to determine the number of well-formed alignments (without calculating the alignments, of course).

```
count_alg::Algebra a Int
count_alg = (fE, fR, fD, fI, fL, fS, choice) where
  fE       = 1
  fR _ x _ = x
  fD _ x   = x
  fI x _   = x
  fL _ _ _ x _ = x
  fS _ x _ _ _ = x
  choice [] = []
  choice xs = [sum xs]
```

The following scoring algebra implements a model with *affine* gap scores [6]. Such a model is used e.g. by *CLUSTALW*, a popular sequence alignment tool [12]. We have extended this algebra by scores for recombinant insertions and deletions. We have given a clear advantage to recombinant indels over regular ones by dividing their penalties by the length of the observed target site duplication.

```
affine_alg::Algebra Base Float
affine_alg = (fE, fR, fD, fI, fL, fS, choice) where
  fE   = 0
  fR a x b = x + matchscore a b
  fD (i,j) x = x + open + fromInt(j-i)*extend
  fI x (i,j) = x + open + fromInt(j-i)*extend
  fL (i,j) (u,u') _ x _ = x + ropen (i,j) + fromInt(u'-u)*rextend
  fS (i,j) x _ (u,u') _ = x + ropen (i,j) + fromInt(u'-u)*rextend
  choice [] = []
  choice xs = [minimum xs]
  open = 5.0
  extend = 0.2
  ropen (i,j) = open/fromInt(j-i)
  rextend = extend

matchscore::Base->Base->Float
matchscore a b | a == b    = 0
               | a > b     = matchscore' b a     -- function is symmetric
               | otherwise = matchscore' a b
                 where matchscore' A G = 1
                       matchscore' A _ = 3
                       matchscore' C G = 3
                       matchscore' C T = 1
                       matchscore' G T = 3
```

The *optimal alignment algebra* combines the scoring algebra with the enumeration algebra. This is straightforward. It returns an optimal alignment together with its score, in $O(n^2)$ space and $O(n^3)$ time.

4 Applications

We have applied our programs to chicken immunoglobin sequences taken from a multiple alignment. The typical improvements achieved by our algorithm are shown in Figure 2:

- In the left part of the recombinant alignment, a gap of length 12 (present in the multiple alignment) is re-discovered in the correct position. Additionally, it is marked as a direct repeat, as it may result from a recombinant insertion with an empty insert. Further experiments reveal that an alignment insensitive to recombination, but with the same scoring otherwise, has an insertion in approximately the same position, but does not exhibit the repeat due to an accidental ambiguity which causes one base to shift from the end to the beginning of the insert.

```
                                         >*    * ** * **     *   <
...tac------------tatggctggtaccag...ctccggttccctatccggctccacaggcacat...
...tactatggctggtactatggctggtaccag...ctccggctccccaggcagaaccacaagcacat...

                                         >                *      <
...tactatggctggtac------------cag...ctccggttccctatccggctccacaggca------------cat...
...tactatggctggtactatggctggtaccag...ctccgg------------ctccccaggcagaaccacaagcacat...
...RRRSSSSSSSSSSSSTTTTTTTTTTTTRRR...RLLLLLUUUUUUUTTTTTRRRRRRRRSSSUUUUUUUUUTTTRRR...
```

Figure 2: Original alignment (top) and recombinant alignment (bottom)

- The right part of the multiple alignment is poor with 8 mismatches (marked by the symbol `*`) within a region of 23 bases (between the delimiters > and <). The recombinant alignment offers an alternative explanation. It exhibits both a recombinant deletion and an insertion, with significant target sites, reducing the mismatch count to 1 over the same region as in the top alignment.

From the *Haskell*-program, DP recurrences were derived (see Section 1.2, Principle 6). Their implementation in C by a student required three days of work, including debugging. The functional program helped to spot errors in the C program that might otherwise have gone unnoticed. First measurements show that the C-program runs faster than the compiled *Haskell*-program by a factor of 68, while using 2% of the space.[4]

5 Conclusion

ADP is a method for algorithm development. It can be applied beneficially merely with pencil and paper. Its embedding in *Haskell* adds the convenience to test ideas very early, i.e., on a very high level of abstraction. The benefits of the functional methods are manyfold:

1. *Haskell*'s infix operators are notational convenience which is essential in this context.

2. The combinator parsing technique allows to have a consistent declarative and operational meaning of the grammar.

3. The equivalence of arrays and functions gives us polynomial efficiency without intellectual complication.

4. Laziness frees us from explicitly programming the order of computation of table entries, which is a most error-prone task in strict setting. Our experience is summarized in the motto "No subscripts, no errors".

5. Algebraic data types and higher order functions allow to separate recognition phase and evaluation algebra. m grammars and n evaluation algebras combine to $m \cdot n$ different analyses. In biosequence analysis, which involves much experimental programming, this compositionality takes the logarithm of the programming effort required otherwise.

The implementation effort can be summarized as follows. Having applied ADP in a different context before, it took an afternoon to adapt the combinator definitions and arrive at the $O(n^6)$ algorithm. Different evaluation

[4]For example, when computing the alignment of Figure 2 (for sequences of length 200), the C-program takes 5 seconds using 1.08 megabytes of space, while the Haskell program takes 340 seconds using 50 megabytes of space. These results were obtain on a Pentium PII computer with 300 MHz and 128 MB RAM. We used the C-compiler *gcc* version 2.7.2.3, and the *Haskell*-compiler *ghc* version 4.04-1.

algebras were helpful to test the program. Coming up with the lookahead based implementation required some thinking, but again, its implementation and testing was a matter of hours.

Although the improved parsers are “hard-coded” rather than defined via combinators, they fit in the rest of the program without friction. The flexibility makes us believe that the ADP method has virtually unlimited potential for improving programming productivity in biosequence analysis.

Acknowledgement We thank Matthias Höchsmann for implementing the recurrences in C and performing the experiments. Dirk Evers carefully read previous versions of the paper.

References

[1] E.L. Anson and E.W. Myers. ReAligner: A Program for Refining DNA Sequence Multi-Alignments. *J. Comp. Biol.*, **4**:369–384, 1997.

[2] E. Birney and R. Durbin. Dynamite: A Flexible Code Generation Language for Dynamic Programming Methods Used in Sequence Comparison. In *Proc. of ISMB 97*, pages 56–64, 1997.

[3] M. A. Gelfand, L. I. Podolsky, T.V. Astakhova, and M. A. Roytberg. Recognition of Genes in Human DNA Sequences. *J. Comp. Biol.*, **3**(2):223–234, 1996.

[4] R. Giegerich. A Declarative Approach to the Development of Dynamic Programming Algorithms, Applied to RNA-Folding. Report 98–02, Technische Fakultät, Universität Bielefeld, 1998. `ftp://ftp.uni-bielefeld.de/pub/papers/techfak/pi/Report98-02.ps.gz`.

[5] A. Gill, J. Launchbury, and S. Peyton-Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, June 1993*. ACM Press, New York, NY, 1993.

[6] O. Gotoh. An Improved Algorithm for Matching Biological Sequences. *J. Mol. Biol.*, **162**:705–708, 1982.

[7] J. Hughes. Lazy Memo-Function. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 129–146. Lecture Notes in Computer Science **201**, Springer Verlag, 1985.

[8] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, **32**(2):98–107, 1989.

[9] G. Hutton. Higher Order Functions for Parsing. *J. Functional Programming*, **3**(2):323–343, 1992.

[10] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino-Acid Sequence of Two Proteins. *J. Mol. Biol.*, **48**:443–453, 1970.

[11] T.F. Smith and M.S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Biol.*, **147**:195–197, 1981.

[12] J.D. Thompson, D.G. Higgins, and T.J. Gibson. *CLUSTAL W*: Improving the Sensitivity of Progressive Multiple Sequence Alignment through Sequence Weighting, Position Specific Gap Penalties and Weight Matrix Choice. *Nucleic Acids Res.*, **22**:4673–4680, 1994.

[13] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, **73**:231–248, 1990.

[14] G.F. Weiller. Phylogenetic Profiles: A Graphical Method for Detecting Genetic Recombinations in Homologous Sequences. *Mol. Biol. Evol.*, **15**(3):326–335, 1998.

[15] M. Zuker. The Use of Dynamic Programming Algorithms in RNA Secondary Structure Prediction. In *Mathematical Methods for DNA Sequences, Waterman, M.S. (editor)*, pages 159–184. 1989.