Sneaking Around *concatMap*

Efficient Combinators for Dynamic Programming

Christian Höner zu Siederdissen
Institute for Theoretical Chemistry, University of Vienna, 1090 Wien, Austria
choener@tbi.univie.ac.at

Abstract

We present a framework of dynamic programming combinators that provides a high-level environment to describe the recursions typical of dynamic programming over sequence data in a style very similar to algebraic dynamic programming (ADP). Using a combination of type-level programming and stream fusion leads to a substantial increase in performance, without sacrificing much of the convenience and theoretical underpinnings of ADP.

We draw examples from the field of computational biology, more specifically RNA secondary structure prediction, to demonstrate how to use these combinators and what differences exist between this library, ADP, and other approaches.

The final version of the combinator library allows writing algorithms with performance close to hand-optimized `C` code.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.4 [*Programming Languages*]: Optimization

General Terms Algorithms, Dynamic Programming

Keywords algebraic dynamic programming, program fusion, functional programming

1. Introduction

Dynamic programming (DP) is a cornerstone of modern computer science with many different applications (e.g. Cormen et al. [6, Cha. 15] or Sedgewick [34, Cha. 37] for a generic treatment). Durbin et al. [8] solve a number of problems on bio-sequences with DP and it is also used in parsing of formal grammars [15].

Despite the number of problems that have been solved using dynamic programming since its inception by Bellman [1], little on methodology has been available until recently. Algebraic dynamic programming (ADP) [10, 12, 13] was introduced to provide a formal, mathematical background as well as an implementation strategy for dynamic programming on sequence data, making DP algorithms less difficult and error-prone to write.

One reviewer of early ADP claimed [10] that *the development of successful dynamic programming recurrences is a matter of experience, talent, and luck*.

The rationale behind this sentence is that designing a dynamic programming algorithm and successfully taking care of all corner cases is non-trivial and further complicated by the fact that most implementations of such an algorithm tend to combine all development steps into a single monolithic program. ADP on the other hand separates three concerns: the construction of the search space, evaluation of each candidate (or correct parse) available within this search space, and efficiency via *tabulation* of parts of the search space using annotation [13] of the grammar.

In this work we target the same set of dynamic programming problems as ADP: dynamic programming over sequence data. In particular, we are mostly concerned with problems from the realm of computational biology, namely RNA bioinformatics, but the general idea we wish to convey, and the library based on this idea, is independent of any specific branch of dynamic programming over sequence data.

In particular, our introductory example uses the CYK algorithm [15, Cha. 4.2] to determine if the input forms part of its context-free language. In other words: our library can be used to write generic high-performance parsers.

The idea of expressing parsers for a formal language using high-level and higher-order functions has a long standing in the functional programming community. Hutton [20] designed a library for *combinator parsing* around 20 years ago. Combining simple parsers for terminal symbols using symbolic operators is now widespread. The *parsec* (see [30, Cha 16] for a tutorial) library for Haskell might be the most well-known. Combinators can be used to build complex parsers in a modular way and it is possible to design new combinators quite easily.

The crucial difference in our work is that the combinators in the ADPfusion library provide efficient code, comparable to hand-written `C`, directly in the functional programming language Haskell [19]. The work of Giegerich et al. [13] already provided an implementation of combinators in Haskell, albeit with large constants, for both space and time requirements. Translation to `C` [11], and recently a completely new language (GAP-L) [33] and compiler (GAP-C) based on the ideas of ADP were introduced as a remedy. Another recent, even more specific, approach is the Tornado language [32] for stochastic context-free grammars designed solely to parse RNA sequences.

Most of the work on domain-specific languages (DSL) for dynamic programming points out that using a DSL has a number of benefits [11], either in terms of better parser error handling, higher performance, or encapsulation from features not regarded as part of the DSL.

Designing a DSL written as part of the host language, provides a number of benefits as well, and strangely, one such benefit is being able to use features not provided by the DSL. Designing a language with a restricted set of features always poses the danger of having a potential user requiring *exactly* that feature which has not yet been made available. A direct embedding, on the other hand, simply does

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

not have this problem. If something can be expressed in the host language, it is possible to use it in the DSL.

Another point in favor of staying within an established language framework is that optimization work done on the backend is automatically available as well. One of the points made by our work is that it is not required to move to a specialized DSL compiler to achieve good performance. Furthermore, certain features of the Haskell host language yield performance-increasing code generation (almost) for free.

What we can not provide is simplified error handling, but this becomes a non-issue for us as we specifically aim for Haskell-savvy users or those interested in learning the language.

We can also determine the appropriateness of embedding the ADPfusion DSL by looking at the guidelines given by Mernik et al. [27, Sec. 2.5.2] on "When and How to Develop Domain-Specific Languages". Most advantages of the embedded approach (development effort, more powerful language, reuse of infrastructure) are ours while some disadvantages of embedding (sub-optimal syntax, operator overloading) are easily dealt with using the very flexible Haskell syntax and standards set by ADP.

Our main contributions to dynamic programming on sequence data are:

- a generic framework separating grammatical structure, semantics of parses, and automatic generation of high-performance code using stream fusion;
- removal of the need for explicit index calculations: the combinator library takes care of corner cases and allows for linked index spaces;
- performance close to `C` with real-world examples from the area of computational biology;
- the possibility to use the library for more general parsing problems (beyond DP algorithms) involving production rules.

"Sneaking around *concatMap*" is a play on one of the ways how to write the Cartesian product of two sets. (Non-) terminals in production rules of grammars yield sets of parses. Efficient, generic treatment of production rules in an embedded DSL requires some work as we will explain in this work.

The outline of the paper is as follows: in the next section we introduce a simple parsing problem. Using this problem as an example, we rewrite it using ADP in Sec. 3, thereby showing the benefits of an embedded DSL. A short introduction to stream fusion follows (Sec. 4).

Armed with knowledge of both ADP and stream fusion, we write DP combinators that are compiled into efficient code in Sec. 5. We expand on ADPfusion with nested productions for more efficient code (Sec. 6).

Runtime performance of ADPfusion is given for two examples from RNA bioinformatics in Sec. 7 with comparisons to `C` programs.

Sections 8 and 9 are on specialized topics and we conclude with remarks on further work and open questions in Sec. 10.

2. Sum of digits

To introduce the problem we want to solve, consider a string of matched brackets and digits like $((1)(3))$. We are interested in the sum of all digits, which can simply be calculated by

```
sumD = sum ∘ map readD ∘ filter isDigit
readD x = read [x] :: Int.
```

The above algorithm works, because the structure of the nesting and digits plays no role in determining the semantics (sum of digits) of the input. For the sake of a simple introductory example, we

```
S → 1 | 2 | ... | 0  -- single digit
  |     ( S )        -- '(' substring ')'
  |      S S         -- split into substrings
```

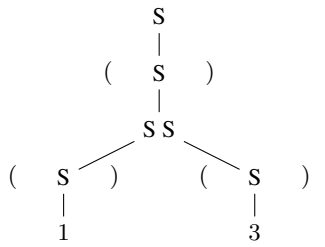

Figure 1. top: A context-free grammar for the nested-digits problem of Sec. 2. CFGs describe the structure of the search space. The semantics of a parse are completely separate.
bottom: Successful parse of the string $((1)(3))$. The semantics of this parse are 4 using the sum of digits semantics.

now assume that we have to solve this problem using a parser that implements the following three rules:

1. a digit may be read only if the string is of size 1, and the single character is a digit;
2. an outermost pair of brackets may be removed, these rules are then applied recursively to the remaining string;
3. the string may be split into two non-empty substrings to which these rules are applied recursively.

These rules can be written as a context-free grammar (CFG) and such a grammar is shown in Fig. 1 together with the successful parse of the string $((1)(3))$. As can be seen, the grammar describes the structure of parses, but make no mention of any kind of semantics. We simply know that a string is a *word* in the grammar but there is no meaning or semantics attached to it. This, of course, conforms to parsing of formal languages in general [15].

One way of parsing an input string is to use the CYK parser, which is a bottom-up parser using dynamic programming [15, Cha. 4.2].

We have chosen the example grammar of Fig. 1 for two reasons. First, it covers a lot of different parsing cases. The first production rule describes parsing with a single terminal on the right-hand side. The second rule includes a non-terminal bracketed by two terminal characters. The third rule requires parsing with two non-terminals. In addition, there are up to $n - 1$ different successful parses for the $n - 1$ different ways to split the string into two non-empty substrings. The third rule makes use of Bellman's principle of optimality. Parses of substrings are re-used in subsequent parses and optimal parses of substrings can be combined to form the optimal parse of the current string. This requires memoization.

Second, these seemingly arbitrary rules are actually very close to those used in RNA secondary structure prediction, being described in Sec. 7, conforming to hairpin structures, basepairing, and parallel structures. Furthermore, important aspects of dynamic programming (DP) and context-free grammars (CFGs) are included.

In the next section, we introduce algebraic dynamic programming (ADP), a domain-specific language (DSL) for dynamic programming over sequence data embedded in Haskell. Using the example from above, we will be able to separate structure and semantics of the algorithm.

```
-- signature
readDigit :: Char → S
bracket   :: Char → S → Char → S
split     :: S → S → S
h         :: [S] → [S]

-- structure or grammar
sd = (
readDigit <<< char ‘with‘ isDigit           |||
(bracket   <<< char ‘cThenS‘ sd ‘sThenC‘ char)
               ‘with‘ brackets              |||
split     <<< sd ‘nonEmpty‘ sd              ... h)

-- additional structure encoding
isDigit (i,j)  = j-i≡1 && Data.Char.isDigit (inp!j)
brackets (i,j) = inp!(i+1)≡’(’ && inp!j≡’)’
-- (!) is the array indexing operator: array ! index

-- semantics or algebra
readDigit c = read [c] :: Int
bracket l s r = s
split l r = l+r
h xs = if null xs then [] else [maximum xs]
```

Figure 2. Signature, structure (grammar), and semantics (algebra) of the sum-of-digits example (Sec. 2). The functions `cThenS`, `sThenC`, and `nonEmpty` chain arguments of terminals like `char` and non-terminals like `sd`. A special case is `with` that filters candidate parses based on a predicate. The symbolic functions `(<<<)`, `(|||)`, and `(...)` apply a function, allow different parses, and select a parse as optimal, respectively.

3. Algebraic dynamic programming

In this section we briefly recall the basic premises of algebraic dynamic programming (ADP) as described in Giegerich et al. [13]. We need to consider four aspects. The signature, defining an interface between a grammar and algebra, the grammar which defines the structure of the problem, its algebras each giving specific semantics, and memoization.

ADP makes use of *subwords*. A subword is a pair `(Int,Int)` which indexes a substring of the input. Combinators, terminals, and non-terminals all carry a subword as the last argument, typically as `(i,j)` with the understanding that $0 \leq i \leq j \leq n$ with n being the length of the input.

3.1 Signature

We have a finite alphabet $\mathcal{A}$ of symbols over which finite strings, including the empty string ϵ can be formed. In addition, we have a sort symbol denoted S. A signature Σ in ADP is a set of functions, with each function $f_i \in \Sigma$ having a specific type $f_i :: t_{i1} \rightarrow \cdots \rightarrow t_{in} \rightarrow S$ where $t_{ik} \in \{\mathcal{A}^+, S\}$. In other words, each function within the signature has one or more arguments, and each argument is a non-empty string over the alphabet or of the sort type which in turn is also the return type of each function. It is possible to use arguments which are derivative of these types, for instance providing the length of a string instead of the string itself. Such more specific cases are optimizations which do not concern us here.

The signature Σ includes an objective function $h : \{S\} \rightarrow \{S\}$. The objective function selects from a set of possible answers those which optimize a certain criterion like minimization or maximization.

3.2 Grammar

Grammars in ADP vaguely resemble grammatical descriptions as used in text books [15] or formal methods like the Backus-Naur form. They define the structure of the search space or the set of all parses of an input. In Fig. 2 we have the grammar for the sum of digits example of Fig. 1. The grammar has one non-terminal `sd`, which is equivalent to `S` of the context-free grammar. Furthermore, we have the three rules again. There are, however, major differences in how one encodes such a rule. Consider the third rule (`S → S S`) which now reads (`split <<<sd ‘nonEmpty‘ sd`) minus the left-hand side. From left to right, we recognize one of the function symbols of the signature (`split`), a combinator function (`<<<`) that applies the function to its left to the argument on its right, the non-terminal (`sd`), a second combinator function (`‘nonEmpty‘`) in infix notation, and finally the non-terminal again.

What these combinators do is best explained by showing their source, which also leads us back to why we want to “sneak around *concatMap*”.

```
infix 8 <<<
(<<<) :: (a→b) → (Subword→[a]) → Subword → [b]
(<<<) f xs (i,j) = map f (xs (i,j))

infixl 7 ‘nonEmpty‘
nonEmpty :: (Subword → [y→z])
         → (Subword → [y]) → Subword → [z]
nonEmpty fs ys (i,j) = [ f y
                       | k ← [i+1..j-1]
                       , f ← fs (i,k)
                       , y ← ys (k,j) ]
```

or equivalently

```
nonEmpty fs ys (i,j) = concatMap idx [i+1..j-1]
where
idx k = concatMap (λf → map f (ys (k,j))) (fs (i,k))
```

Each combinator takes a left and a right argument and builds a list from the Cartesian product of the two inputs, with `(<<<)` taking care of the scalar nature of the function to be mapped over all inputs. Importantly, all first arguments are partially applied functions which has a performance impact and hinders optimization.

The third argument of each combinator is the subword index that is threaded through all arguments. (Non-) terminals are functions from a subword to a list of values. For example `char` returns a singleton list with the i'th character of the input when given the subword $(i, i + 1)$ and an empty list otherwise.

```
char :: Subword → [Char]
char (i,j) = [inp!i | i+1≡j]
```

Similarly, the non-terminal `sd` is a function

```
sd :: Subword → [S]
sd = ... -- grammar as above
```

which can be memoized as required.

As a side note, in GHC Haskell, `concatMap` is not used in the implementation of list comprehensions, but the message stays the same: as we will see later in runtime measurements `concatMap` and list comprehensions are hard to optimize.

We complete the argument combinators with `cThenS` and `sThenC` (having the same type as `nonEmpty`):

```
infixl 7 ‘cThenS‘ ‘sThenC‘

cThenS fs ys (i,j) = [ f y | i<j, f ← fs (i,i+1)
                           , y ← ys (i+1,j) ]
```

```
sThenC fs ys (i,j) = [ f y | i<j, f ← fs (i,j-1)
                      , y ← ys (j-1,j) ]
```

We are still missing two combinators, `(|||)` and `(...)`. Both are simple, as ADP deals solely with lists, we just need to take care of the subword index in each case.

```
infixr 6 |||
(|||) :: (Subword→[a])→(Subword→[a])→Subword→[a]
(|||) xs ys (i,j) = xs (i,j) ++ ys (i,j)
infix 5 ...
(...) :: (Subword→[a])→([a]→[a])→Subword→[a]
(...) xs h (i,j) = h (xs (i,j))
```

There is an actual difference in the grammars of Fig. 1 and Fig. 2. In Fig. 1 the terminal symbols are explicit characters like '1' or '(', while in Fig. 2 `char` matches all single characters. We use this to introduce another useful combinator `(with)` that allows us to filter parses based on a predicate:

```
with :: (Subword→[a])→(Subword→Bool)→Subword→[a]
with xs p (i,j) = if p (i,j) then xs (i,j) else []
```

3.3 Algebra

We now have the grammar describing the structure of the algorithm. The function symbols of the signature are included and can be "filled" using one of several algebras describing the semantics we are interested in. Apart from the objective function `h`, all functions describe the semantics of production rules of the grammar. In our example above (Fig. 2) we either read a single digit (`readDigit`), keep just the sum of digits of the bracketed substring (`bracket`), or add sums from a `split` operation.

The objective function `(h)` selects the optimal parse according to the semantics we are interested in. In this case the maximum over the parses.

Another possibility is to calculate some descriptive value over the search space, say its total size. As an example, the *Inside-Outside* algorithm [23],[8, Ch. 10] adds up all probabilities generated by productions in a stochastic CFG instead of selecting one.

The specialty of ADP grammars is that they form tree grammars [10], [9, Sec. 2.2.1]. While they are analogous to context-free grammars, the right-hand sides of productions form proper trees with function symbols from the signature as inner nodes and terminal and non-terminal symbols at the leaves. For the example parse in Fig. 1 (bottom) this has the effect of replacing all non-terminal symbols `(S)` with function symbols from the signature. This also means that we can, at least in principle, completely decouple the generation of each parse tree from its evaluation. While the size of the search space might be prohibitive in practice, for small inputs, an exhaustive enumeration of all parses is possible. In an implementation, data constructors can be used as functions, while the objective function is simply the identity function. This allows us to print all possible parse trees for an input.

Each such tree has at its leaf nodes a sequence of characters, a word, from the alphabet: $w \in \mathcal{A}^*$. And for each given word w there are zero or more trees representing this word. If no tree for a given word exists, then the grammar can not parse that word and if more than one tree exists then the grammar is syntactically ambiguous in this regard. This kind of ambiguity is not problematic, typically even wanted, as the objective function can be used to evaluate different tree representations of a word w and return the optimal one.

3.4 Memoization

As noted in Sec. 3.2, non-terminals in grammars can be memoized. ADP introduces a function

`tabulated :: Int →(Subword →[a]) →Subword →[a]`, used as `sd = tabulated (length input) ( productions )`, that stores the answers for each subword in an array. Depending on the algorithm, other memoization schemes, or none at all, are possible. In general, memoization is required to make use of Bellman's principle of optimality and reduce the runtime of an algorithm from exponential to polynomial.

3.5 ADP in short

To summarize, algebraic dynamic programming achieves a separation of concerns. Parsing of input strings for a given grammar is delegated to the construction of candidates which are correct parses. Evaluation of candidates is done by specifying an evaluation algebra, of which there can be more than one. Selection from all candidates based on their evaluation is done by an objective function which is part of each evaluation algebra. Memoization makes the parsing process asymptotically efficient. Giegerich et al. [13] provide a much more detailed description than given here.

As our objective is to perform parsing, evaluation and selection more efficiently, we will, in the next sections, change our view of dynamic programming over sequence data to describe our approach starting with streams as a more efficient alternative to lists.

4. Stream fusion

We introduce the basics of stream fusion here. Considering that the ADPfusion library is based around applications of `map`, `flatten` (a variant of `concatMap`, more amenable to fusion), and `fold`, these are the functions described. For advanced applications, the whole range of fusible functions may be used, but those fall outside the scope of both this introduction to stream fusion and the paper in general. In addition, here and in the description of ADPfusion, we omit that stream fusion and ADPfusion are parametrized over Monads.

Stream fusion is a short-cut fusion system for lists [7], and more recently arrays [25], that removes intermediate data structures from complicated compositions of list-manipulating functions. Ideally, the final, fused, algorithm is completely free of any temporary allocations and performs all computations efficiently in registers, apart from having to access static data structures containing data. Stream fusion is notable for fusing a larger set of functions than was previously possible, including zipping and concatenating of lists.

Stream fusion is built upon two data types, `Stream` captures a function to calculate a single step in the stream and the seed required to calculate said step. A `Step` can indicate if a stream is terminated (`Done`). If not, the current step either `Yields` a value or `Skips`, which can be used for filtering. One other use of `Skip` is in concatenation of streams which becomes non-recursive due to `Skip` as well. Unless a stream is `Done`, each new `Step` creates a new seed, too.

```
data Stream a = ∃ s. Stream (s → Step a s) s
data Step a s = Done
              | Yield a s
              | Skip s
```

The point of representing lists as sequence co-structures is that no function on streams is recursive (except final *fold*s), permitting easy fusion to generate efficient code.

We construct a new stream of a single element using the `singleton` function. A singleton stream emits a single element `x` and is `Done` thereafter. Notably, the `step` function defined here is non-recursive.

```
singleton_S x = Stream step True where
  step True  = Yield x False
  step False = Done
```

Mapping a function over a stream is non-recursive as well, in marked contrast to how one maps a function over a list.

```
mapS f (Stream step s) = Stream nstp s where
  nstp s = case (step s) of
    Yield x s' → Yield (f x) s'
    Skip    s' → Skip s'
    Done       → Done
```

As a warm-up to stream-flattening, and because we need to concatenate two streams with `(|||)` anyway, we look at the stream version of `(++)`.

```
(Stream stp1 ss) ++S (Stream stp2 tt) =
Stream step (Left ss) where
  step (Left s) = case s of
    Yield x s' → Yield x (Left s')
    Skip    s' → Skip    (Left s')
    Done       → Skip    (Right tt)
  step (Right t) = case t of
    Yield x t' → Yield x (Right t')
    Skip    t' → Skip    (Right t')
    Done       → Done
```

The `Left` and `Right` constructors encode which of the two streams is being worked on, while the jump from the first to the second stream is done via a (again non-recursive) `Skip`.

The `flatten` function takes three arguments: a function `mk` which takes a value from the input stream and produces an initial seed for the user-supplied `step` function. The user-supplied `step` then produces zero or more elements of the resulting stream for each such supplied value. Note the similarity to stream concatenation. `Left` and `Right` are state switches to either initialize a new substream or to create stream `Steps` based on this initial seed.

Again, it is important to notice that no function is recursive, the hand-off between extracting a new value from the outer stream and generating part of the new stream is done via `Skip (Right (mk a, t'))`.

```
flattenS mk step (Stream ostp s) = Stream nstp (Left s)
where
  nstp (Left t) = case (ostp s) of
    Yield a t' → Skip (Right (mk a, t'))
    Skip    t' → Skip (Left t')
    Done       → Done
  nstp (Right (b,t)) = case (step b) of
    Yield x s' → Yield x (Right (s',t))
    Skip    s' → Skip (Right (s',t))
    Done       → Left t
```

Finally, we present the only recursive part of the stream fusion, folding a stream to produce a final value.

```
foldS f z (Stream step s) = loop f z where
  loop f z = case (step s) of
    Yield x s' → loop (f z x) s'
    Skip    s' → loop z s'
    Done       → z
```

If such code is used to build larger functions like

```
foldS (+) 0 (flattenS id f (singletonS 10)) where
f x = if (x > 0)
      then Yield x (x-1)
      else Done
```

call-pattern specialization [31] of the constructors (`Yield, Skip, Done`) creates specialized functions for the different cases, and inlining merges the newly created functions, producing an efficient, tight loop. A detailed explanation can be found in Coutts et al. [7, Sec. 7] together with a worked example. The GHC compiler [36] performs all necessary optimizations.

5. Designing efficient combinators for dynamic programming

Algebraic dynamic programming is already able to provide asymptotically optimal dynamic programming recursions. A dynamic program written in ADP unfortunately comes with a rather high overhead compared to more direct implementations. Two solutions have been proposed to this problem. The first was translation of ADP code into `C` using the ADP Compiler [35] and the second a complete redesign providing a new language and compiler (GAP-L and GAP-C) [33]. Both approaches have their merit but partially different goals than ours. Here we want to show how to keep most of the benefits of ADP while staying *within* Haskell instead of having to resort to a different language.

We introduce combinators in a top-down manner, staying close to our introductory example of Fig. 2. An important difference is that functions now operate over stream fusion [7] streams instead of lists. This change in internal representation lets the compiler optimize grammar and algebra code much better than otherwise possible.

We indicate the use of stream fusion functions like `mapS` with a subscript $_S$ to differentiate between normal list-based functions and stream fusion versions.

5.1 Combining and reducing streams

Two of the combinators, the choice between different productions `(|||)` and the application of an objective function, stay essentially the same, except that the type of `h` is now `Stream a →b`, instead of `[a] →[b]`. The objective function returns an answer of a scalar type, say `Int`, allowing for algorithms that work solely with unboxed types, or a vector type (like lists, boxed, or unboxed vectors). This gives greater flexibility in terms of what kind of answers can be calculated and choosing the best encoding, in terms of performance, for each algorithm.

```
infixl 7 |||
(|||) xs ys ij = xs ij ++S ys ij

infixl 6 ...
(...) stream h ij = h (stream ij)
```

In addition, the index is not a tuple anymore, but rather a variable `ij` of type `DIM2`. Instead of plain pairs `(Int,Int)` we use the same indexing style as the Repa [22] library. Repa tuples are inductively defined using two data types and constructors:

```
data Z = Z
data a :. b = a :. b
type DIM1 = Z :. Int
type DIM2 = DIM1 :. Int
```

The tuple constructor `(:.)` resembles the plain tuple constructor `(,)`, with `Z` as the base case when constructing a 1-tuple `(Z:.a)`. We can generalize the library to cover higher-dimensional DP algorithms just like the Repa library does for matrix calculations. It allows for uniform handling of multiple running indices which are represented as k-dimensional inductice tuples as well, increasing k by one for each new (non-) terminal. Using plain tuples would require nesting of pairs. Also, subwords are now of type `DIM2` instead of `(Int,Int)`.

5.2 Creating streams from production rules

As of now, we can combine streams and reduce streams to a single value or a set of values of interest. As streams expose many

optimization options to the compiler (cf. Sec. 4 and [7]), we can expect good performance. What is still missing is how to create a stream, given a production rule, in the first place. Rules such as `readDigit <<<char` with a single terminal or non-terminal to the right are the simplest to construct.

The combinator `(<<<)` applies a function to one or more arguments and is defined as:

```
infixl 8 <<<
(<<<) f t ij =
  mapS (λ(_,_,as) → apply f as) (streamGen t ij)
```

The `streamGen` function takes the argument `arg` on the right of `(f <<< arg)`, with `arg` of type `DIM2` $\rightarrow\alpha$, and the current subword index to create a stream of elements. If α is scalar (expressed as `DIM2` $\rightarrow$`Scalar` β), the result is a singleton stream, containing just β, but α can also be of a vector type say $[\beta]$, in which case a stream of β arguments is generated, containing as many elements as are in the vector data structure.

We use a functional dependency to express[1] that the type of the stream `r` is completely determined by the type of the (non-) terminal(s) `t`.

```
class StreamGen t r | t → r where
  streamGen :: t → DIM2 → Stream r
```

The instance for a scalar argument `(DIM2` $\rightarrow$`Scalar` β`)` follows as:

```
instance StreamGen (DIM2 → Scalar β) (DIM2,Z:.Z,Z:.β)
```

delaying the actual implementation for now.

Streams generated by `streamGen` have as element type a triple of inductively defined tuples we call "stacks", whose stack-like nature is only a type-level device, no stacks are present during runtime.

The first element of the triple is the subword index, the second gives an index into vector-like data structures, while the third element of the triple holds the actual values. We ignore the second element for now, just noting that (non-) terminals of scalar type do not need indexing, hence `Z` as type and value of the index. Arguments are encoded using inductive tuples, and as we only have one argument to the right of `(<<<)`, the tuple is `(Z:.`α`)`, as all such tuples or stacks (e.g. subword indices, indices into data structures, argument stacks) always terminate with `Z`.

The final ingredient of `(<<<)`, `apply`, is now comparatively simple to implement and takes an n-argument function `f` and applies it to n arguments `(Z:.`$\mathtt{a}_1$`:.`$\cdots$`:.`$\mathtt{a}_n$`)`. We introduce a type dependency between the arguments of the function to apply and the arguments on the argument stack, using an associated type synonym.

```
class Apply x where
  type Fun x :: ∗
  apply :: Fun x → x

instance Apply (Z:.a1:. ··· :.an → r) where
  type Fun (Z:.a1:. ··· :.an → r)
    = a1 → ··· → an → r
  apply fun (Z:.a1:. ··· :.an) = fun a1 ··· an
```

5.3 Extracting values from (non-) terminals

As a prelude to our first stream generation instance (that we still have to implement) we need to be able to extract values from terminals and non-terminals. There are three classes of arguments that act as (non-) terminals. We have already encountered the type `(DIM2` $\rightarrow$`Scalar` β`)` for functions returning a single (scalar) value. A second class of functions yields multiple values of type β: `(DIM2` $\rightarrow$`Vector` β`)`. In this case we do not have vector-valued arguments to but rather multiple choices from which to select. Finally, we can have data structures. A data structure can again store single (scalar) results or multiple results (vector-like) for each subword. For data structures, it will be necessary to perform an indexing operation (e.g. `(!)` is used for the default Haskell arrays) to access values for a specific subword.

The `ExtractValue` type class presented below is generic enough to allow many possible styles of retrieving values for a subword and new instances can easily be written by the user of the library.

We shall restrict ourselves to the instance `(DIM2` $\rightarrow$`Scalar` β`)`. Instances for other common data structures are available with the library, including lazy and strict arrays of scalar and vector type.

The `ExtractValue` class itself has two associated types, `Asor` denoting the accessor type for indexing individual values within a vector-like argument and `Elem` for the type of the values being retrieved.

For, say, `(DIM2` $\rightarrow$`[`β`])`, a possible `Asor` type is `Int` using the list index operator `(!!)`, while the `Elem` type is β.

For scalar types, the `Asor` will be `Z` as there is no need for an index operation in that case.

The type class for value extraction is:

```
class ExtractValue cnt where
  type Asor cnt :: ∗
  type Elem cnt :: ∗
  extractStream
    :: cnt → Stream (Idx3 z,as,vs)
    → Stream (Idx3 z, as:.Asor cnt,vs:.Elem cnt)
  extractStreamLast
    :: cnt → Stream (Idx2 z,as,vs)
    → Stream (Idx2 z,as:.Asor cnt,vs:.Elem cnt)

type Idx3 z = z:.Int:.Int:.Int
type Idx2 z = z:.Int:.Int
```

`extractStream` and `extractStreamLast` are required to correctly handle subword indices with multiple arguments in productions. Their use is explained below, but note that `extractStream` accesses the 2nd right-most subword (k, l), while `extractStreamLast` accesses the rightmost (l, j) one. Consider the production

```
S → x y z
    i k l j
```

where `y` would be handled by `extractStream` and `z` by `extractStreamLast`, and `x` has already been handled at this point, its value is on the `Elem` stack.

Each function takes a stream and extends the accessor (`Asor`) stack with its accessor and the value (`Elem`) stack is extended with the value of the argument.

Now to the actual instance for `(DIM2` $\rightarrow$`Scalar` β`)`:

```
instance ExtractValue (DIM2 → Scalar β) where
  type Asor (DIM2 → Scalar β) = Z
  type Elem (DIM2 → Scalar β) = β
  extractStream cnt s = mapS f s where
    f (z:.k:.l:.j,as,vs) =
    let Scalar v = cnt (Z:.k:.l)
    in  (z:.k:.l:.j,as:.Z,vs:.v)
  extractStreamLast cnt s = mapS f s where
    f (z:.l:.j,as,vs) =
    let Scalar v = cnt (Z:.l:.j)
    in  (z:.l:.j,as:.Z,vs:.v)
```

[1] Instead of type families for reasons explained in Sec. 9.

5.4 Streams for productions with one (non-) terminal

We can finish the implementation for streams of (DIM2 →Scalar β) arguments. The instance is quite similar to the `singleton` function presented in Sec. 4 but while `singleton` creates a single-element stream unconditionally we have to take care to only create a stream if the subword `(Z:.i:.j)` is legal. An illegal subword $i > j$ should lead to an empty stream.

```
instance
( ExtractValue (DIM2 → Scalar β)
) ⇒ StreamGen (DIM2 → Scalar β) (DIM2,Z:.Z,Z:.β)
  where
  streamGen x ij = extractStreamLast x
                   (unfoldr_S step ij)
  step (Z:.i:.j)
    | i≤j = Just ((Z:.i:.j,Z,Z), (Z:.j+1:.j))
    | otherwise = Nothing
```

In this case, we use the subword `ij` as seed. If the subword is legal, a stream with this subword and empty `(Z) Asor` and `Elem` stacks is created. The new seed is the *illegal* subword $(j+1, j)$ which will terminate the stream after the first element.

We then immediately extend the stream elements using `extractStreamLast` which creates the final stream of type `(DIM2,Z:.Z,Z:.`β`)` by adding the corresponding accessor of type `Z` and element of type β as top-most element to their stack. With one argument, the only argument is necessarily the last one, hence the use of `extractStreamLast` instead of `extractStream`.

Using the construction scheme of only creating streams if subwords are legal, we effectively take care of all corner cases. Illegal streams (due to illegal subwords) are terminated before we ever try to extract values from arguments. This means that `ExtractValue` instances typically do not have to perform costly runtime checks of subword arguments.

5.5 Handling multiple arguments

We implement a single combinator `(nonEmpty)` as this is already enough to show how productions with any number (≥ 2) of arguments can be handled. In addition, `nonEmpty` has to deal with the corner case of empty subwords ($i = j$) on both sides. That is, its left and right argument receive only subwords of at least size one.

Recall that in ADP the first argument to each combinator turns out to be a partially applied function that is immediately given its next argument with each additional combinator. Partially applied functions, however, can reduce the performance of our code and make it impossible (or at least hard) to change the subword index space dependent on arguments to the left of the current combinator as the function would already have been applied to those arguments.

By letting `nonEmpty` have a higher binding strength than `(<<<)` we can first collect all arguments and then apply the corresponding algebra function. In addition, we need to handle inserting the current running index, `Asor` indices of the arguments, and `Elem` values for a later `apply`. Hence `nonEmpty` is implemented in a completely different way than in ADP:

```
infixl 9 `nonEmpty`
xs `nonEmpty` ys = Box mk step xs ys where
  mk (z:.i:.j,vs,as) = (z:.i:.i+1:.j,vs,as)
  step (z:.i:.k:.j,vs,as)
    | k+1≤j = Yield (z:.i:.k  :.j,vs,as)
                    (z:.i:.k+1:.j,vs,as)
    | otherwise = Done
```

The `nonEmpty` combinator does, in fact, not combine the arguments `xs` and `ys` at all but only prepares two functions `mk` and `step`.

```
streamGen (Box mk step xs ys) =
```

1,2,4		1,3,4		extractStreamLast ys
$xs_{1,2}$	$xs_{1,2}$	$xs_{1,3}$	$xs_{1,3}$	
$ys^1_{2,4}$	$ys^2_{2,4}$	$ys^1_{3,4}$	$ys^2_{3,4}$	

1,2,4	1,3,4	(extractStream xs
$xs_{1,2}$	$xs_{1,3}$	

1,2,4	1,3,4	(flatten$_S$ mk step

1,4	(unfoldr$_S$ step ij)))

Figure 3. A stream from two arguments built step-wise bottom to top. First, a running index is inserted between the original subword $(1, 4)$ indices using `flatten`. Then, elements are extracted from the scalar argument `xs`. The vector-like argument `ys` yields two elements for each subword (indices ^{1} and ^{2}). (`step` as in Sec. 5.4)

These define the set of subwords (i, k) and (k, j) splitting the current subword (i, j) between `xs` and `ys`. Again, we make sure that any corner cases are caught. The first value for k is $i + 1$, after which k only increases. Hence `xs` is nonEmpty. In `step` we also stop creating new elements once $k + 1 > j$ meaning `ys` is never empty. Finally, should the initial subword (i, j) have size $j - i < 2$, the whole stream terminates immediately.

Of course, we are not constructing a stream at all but rather a Box. The implication is that two or more (non-) terminals in a production lead to nested boxes where `xs` is either another `Box` or an argument, while `ys` is always an argument. Furthermore `mk` and `step` are the two functions required by `flatten`. The `streamGen` function will receive such a nested `Box` data structure whenever two or more arguments are involved. The compiler can deconstruct even deeply nested boxes during compile time, enabling full stream fusion optimization for the production rule, completely eliminating *all* intermediate data structures just presented. We expose these optimizations to the compiler with `StreamGen` instances that are recursively applied during compilation.

5.6 Streams from productions with multiple arguments

Efficient stream generation requires deconstucting `Boxes`, correct generation of subwords in streams, and extraction of values from arguments. This can be achieved with a `StreamGen` instance for `Boxes` and an additional type class `PreStreamGen`.

These instances will generate the code shown in Fig. 3 (right).

The `StreamGen` instance fo the outermost `Box`

```
instance
  ( ExtractValue ys, Asor ys ∼ a, Elem ys ∼ v
  , PreStreamGen xs (idx:.Int,as,vs)
  , Idx2 undef ∼ idx
  ) ⇒ StreamGen (Box mk step xs ys)
                (idx:.Int,as:.a,vs:.v) where
  streamGen (Box mk step xs ys) ij
    = extractStreamLast ys
      (preStreamGen (Box mk step xs ys) ij)
```

handles the last argument of a production, extracting values using `extractStreamLast`. `PreStreamGen` instances handle the creation of the stream excluding the last argument recursively employing `preStreamGen`.

And we finally make use of `flatten`. This function allows us to create a stream and use each element as a seed of a substream when

adding an argument further to the right – basically on the way back up from the recursion down of the nested Boxes.

The type class PreStreamGen follows StreamGen exactly:

```
class PreStreamGen s q | s → q where
  preStreamGen :: s → DIM2 → Stream q
```

To handle a total of two arguments, including the last, this PreStreamGen instance is sufficient[2]:

```
instance
( ExtractValue xs, Asor xs ~ a, Elem xs ~ v
, Idx2 undef ~ idx
) ⇒ PreStreamGen (Box mk step xs ys)
                 (idx:.Int,as:.a,vs:.v) where
preStreamGen (Box mk step xs ys) ij
  = extractStream xs
    (flatten_S mk step
    (unfoldr_S step ij))
step (Z:.i:.j)
  | i≤j = Just ((Z:.i:.j,Z,Z), Z:.j+1:.j)
  | otherwise = Nothing
```

For three or more arguments we need a final ingredient. Thanks to overlapping instances (cf. Sec. 9.1 on overlapping instances) this instance

```
instance
( PreStreamGen (Box mkI stepI xs ys) (idx,as,vs)
( ExtractValue ys, Asor ys ~ a, Elem ys ~ v
, Idx2 undef ~ idx
) ⇒ PreStreamGen (Box mk step (Box mkI stepI xs ys) zs)
                 (idx:.Int,as:.a,vs:.v) where
preStreamGen (Box mk step box@(Box _ _ _ ys) zs) ij
  = extractStream ys
    (flatten_S mk step Unknown
    (preStreamGen box ij))
```

which matches two or more nested Boxes, will be used except for the final, innermost Box. Then, the above (more general) instance is chosen and recursion terminates.

As the recursion scheme is based on type class instances, the compiler will instantiate during compilation, exposing each flatten function to fusion. Each of those calculates subword sizes and adds to the subword stack, while Asor and Elem stacks are filled using extractStream and extractStreamLast, thereby completing the ensemble of tools required to turn production rules into efficient code.

5.7 Efficient streams from productions

Compared with ADP combinators (Sec. 3) we have traded a small amount of additional user responsibilities with the potential for enormous increases in performance.

The user needs to write an instance (of ExtractValue) for data structures not covered by the library or wrap such structures with (DIM2 →α) accordingly.

New combinators are slightly more complex as well, requiring the mk and step function to be provided, but again several already exist. Even here, the gains outweigh the cost as each combinator has access to the partially constructed subword, Asor, and Elem stack of its stream step. One such application is found in the RNAfold algorithm (Sec. 7.2) reducing the runtime from $O(n^4)$ to $O(n^3)$ as in the reference implementation.

[2] for type inference purposes, additional type equivalence terms are required for mk and step which are omitted here

6. Applying Bellman's principle locally

All major pieces for efficient dynamic programming are now in place. A first test with a complex real-world dynamic program unfortunately revealed disappointing results. Consider the following production in grammar form:

```
S → char string S string char
    i   i+1     k l     j-1   j
```

Two single characters (char) bracket three arguments of variable size. A stream generated from those five arguments is quadratic in size, due to two indices, k and l, with $i+1 \le k \le l \le j-1$ with k (l) to the left (right) of S. We would like to evaluate the outer arguments (the char terminals) only once, but due to the construction of streams from left to right, the right-most argument between $(j-1, j)$ will be evaluated a total of $O(n^2)$ times. Depending on the argument, this can lead to a noticeable performance drain.

Two solutions present themselves: (i) a more complex evaluation of (non-) terminals or (ii) making use of Bellman's principle. As option (i) requires complex type-level programming, basically determining which argument to evaluate when, and option (ii) has the general benefit of rewriting productions in terms of other productions, let us consider the latter option.

If Bellman's principle holds, a problem can be subdivided into smaller problems that, when combined, yield the same result as solving the original problem, and each subproblem is reused multiple times.

If the above production has the same semantics under an objective function, as the one below, we can rewrite it, and benefit from not having to evaluate the right-most argument more than once.

```
S → char T char          T → string S string
    i  i+1 j-1  j            i+1     k l     j-1
```

We want to introduce another non-terminal (T) only conceptually, but translation into ADPfusion is actually quite easy. Given the original code

```
f <<< char ‘then‘ string ‘then‘ s ‘then‘ string
       ‘then‘ char ... h
```

the new nested version is

```
f <<< char ‘then‘
       (g <<< string ‘then‘ s ‘then‘ string ... h)
       ‘then‘ char ... h
```

This version still yields efficient code and the final char argument is evaluated just once. In terms of ADPfusion, bracketing and evaluation of subproductions (g <<< string ‘then‘ ···) is completely acceptable, the inner production has type (DIM2 →α), variants of which are available by default.

The availability of such an optimization will depend on the specific problem at hand and will not always be obvious. As the only changes are a pair of brackets and an inner objective function, changes are easily applied and a test harness of different input sequences can be used to determine equality of the productions with high certainty – even without having to *prove* that Bellman's principle holds. One particularly good option is to automate testing using QuickCheck [5] properties.

7. Two examples from RNA bioinformatics

In this section, we test the ADPfusion library using two algorithms from the field of computational biology. The Nussinov78 [29] grammar is one of the smallest RNA secondary structure prediction grammars and structurally very similar to our introductory example of Figs. 1 and 2. The second algorithm, RNAfold 2.0 [26] tries to find an optimal RNA secondary structure as well.

Both algorithms can be seen as variants of the CYK algorithm [15, Sec. 4.2]. The difference is that every word is part of the language and parsing is inherently syntactically ambiguous: every input allows many parses. By attaching semantics (say: a score or an energy), similar to the sum of digits semantics, the optimal parse is chosen.

We pit ADPfusion code against equivalent versions written in `C`. The `Nussinov78` grammar and algebra (Fig. 4) are very simple and we will basically measure loop optimization. `RNAfold 2.0` is part of an extensive set of tools in the ViennaRNA package [26]. The complicated structure and multiple energy tables lead to a good "real-world" test.

All benchmarks are geared toward the comparison of `C` and ADPfusion in Haskell. Legacy ADP runtimes are included to point out how much of an improvement we get by using strict, unboxed arrays and a modern fusion framework.

The legacy ADP version of RNAfold is not directly compatible with `RNAfold 2.0` (`C` and ADPfusion). It is based on an older version of RNAfold (1.x) which is roughly 5% – 10% faster than 2.0.

We do not provide memory benchmarks. For `C` vs. ADPfusion the requirements for the DP tables are essentially the same, while legacy ADP uses boxed tables and always stores lists of results with much overhead.

The Haskell versions of `Nussinov78` and `RNAfold 2.0` have been compiled with `GHC 7.2.2` and `LLVM 2.8`; compilation options: `-fllvm -Odph -optlo-O3`. The `C` version of `Nussinov78` was compiled using `GCC 4.6` with `-O3`. The ViennaRNA package was compiled with default configuration, including `-O2` using `GCC 4.6`. All tests were done on an Intel Core i7 860 (2.8 GHz) with 8 GByte of RAM.

7.1 Nussinov's RNA folding algorithm

The algorithm by Nussinov et al. [29] is a very convenient example algorithm that is both: simple, yet complex enough to make an interesting test. A variant of the algorithm in ADP notation is shown in Fig. 4 together with its CFG. The algorithm expects as input a sequence of characters from the alphabet $\mathcal{A} = \{ACGU\}$. A *canonical basepair* is one of the six (out of 16 possible) in the set $\{AU, UA, CG, GC, GU, UG\}$. The algorithm maximizes the number of paired nucleotides with two additional rules.

Two nucleotides at the left and right end of a subword (i, j) can pair only if they form one of the six canonical pairs. For all pairs (k, l) it holds that neither $i < k < j < l$ nor $k < i < l < j$ and if $i == k$ then $j == l$. Any two pairs are juxtaposed or one is embedded in the other.

The mathematical formulation of the recursion implied by the grammar and pairmax semantics in Fig. 4 is

$$S[i,j] = \max \begin{cases} 0 & i == j \\ S[i+1,j] & i < j \\ S[i,j-1] & i < j \\ S[i+1,j-1]+1 & \text{if (i,j) pairing} \\ \max_{i<k<j} S[i,k] + S[k+1,j] & . \end{cases}$$

As there is only one non-terminal `S` (respectively DP matrix `s`) and no scoring or energy tables are involved, the algorithm measures mainly the performance for three nested loops and accessing one array.

As Fig. 5 clearly shows, we reach a performance within $\times 2$ of `C` for moderate-sized input. The `C` version used here is part of the `Nussinov78` package available online[3].

[3] `Nussinov78` hackage library: `http://hackage.haskell.org/package/Nussinov78`

```
-- signature
nil   :: S
left  :: Char → S → S
right :: S → Char → S
pair  :: Char → S → Char → S
split :: S → S → S
h     :: Stream S → S

-- structure or grammar
              s = (
S →  ϵ        nil   <<< empty               |||
  |  bS       left  <<< base −∼∼ s          |||
  |   Sb      right <<<          s ∼∼− base |||
  |  bSb      pair  <<< base −∼∼ s ∼∼− base
  |  S S                'with' pairing     |||
              split <<<       s +∼+ s       ... h)

-- semantics or algebra
nil = 0                    pair a s b = s+1
left b s   = s             split l r  = l+r
right  s b = s             h xs = maximum_S xs
```

Figure 4. Top: The signature Σ for the `Nussinov78` grammar. The functions `nil, left, right, pair,` and `split` build larger answers `S` out of smaller ones. The objective function `h` transforms a stream of candidate answers, e.g. by selecting only the optimal candidate.
Center left: The context-free grammar `Nussinov78`. Character `b` $\in \mathcal{A} = \{$A,C,G,U$\}$.
Center right:The `Nussinov78` algorithm in ADPfusion notation with `base :: DIM2 →Char`. This example was taken from [14]. Compared to the CFG notation, the evaluation functions are now explicit as is the non-empty condition for the subwords of `split`. The `(−∼∼)` combinator allows a size-one subword to its left (cf. `cThenS` in Fig. 2). Its companion `(∼∼−)` to the right (`sThenC`). The `(+∼+)` combinator enforces non-empty subwords (`nonEmpty`).
Bottom: Pairmax algebra (semantics); maximizing the number of basepairs. In `pair`, it is known that `a` and `b` form a valid pair due to the `pairing` predicate of the grammar.

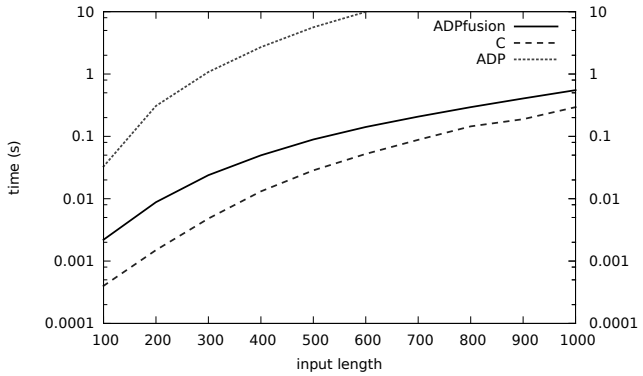

Figure 5. Runtime in seconds for different versions of the `Nussinov78` algorithm. The `Nussinov78` algorithm accesses only one DP matrix and no "energy tables". The comparatively high runtime for the ADPfusion code for small input is an artifact partially due to enabled backtracking.

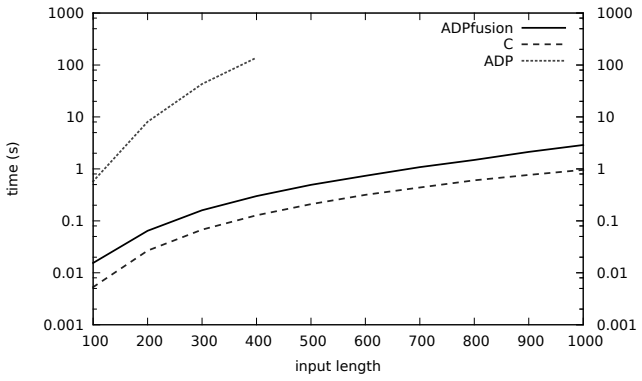

Figure 6. Runtime in seconds for different implementations of the `RNAfold 2.0` algorithm for random input of different length. The highly optimized `C` code is used by the official ViennaRNA package. ADPfusion is the code generated by our library. For illustrative purposes, ADP is the performance of the original Haskell implementation of the older `RNAfold 1.x` code.

An algorithm like `Nussinov78` is, however, not a good representative of recent developments in computational biology. Modern algorithms, while still adhering to the basic principles formulated by Nussinov et al. [29], use multiple DP matrices and typically access a number of additional tables providing scores for of different features. The `RNAfold 2.0` algorithm, described next, is one such algorithm.

7.2 RNAfold

The ViennaRNA package [16, 26] is a widely used state-of-the-art software package for RNA secondary structure prediction. It's newest incarnation is among the top programs in terms of prediction accuracy and one of the fastest. It provides an interesting target as it has been optimized quite heavily toward faster prediction of results. Compared to other programs, speed differences of $\times 10$ to $\times 100$ in favor of `RNAfold 2.0` are not uncommon [26].

The complete ViennaRNA package provides many different algorithms which makes it impractical to re-implement the whole package in Haskell. We concentrate on the minimum-free energy prediction part, which is the most basic of the offered algorithms.

We refrain from showing the ADPfusion version of the grammar. A version of RNAfold using recursion and diagrams for visualization is described in [2] and the ADPfusion grammar itself can be examined online[4].

We do, however, give some statistics. The grammar uses 4 nonterminals, three of which are interdependent while the fourth is being used to calculate "exterior" structures and only $O(n)$ matrix cells are filled instead of $O(n^2)$ as for the other three tables. A total of 17 production rules are involved and 18 energy tables. One production has an asymptotic runtime of $O(n^2)$ for each subword yielding a total runtime of $O(n^4)$. By restricting the maximal size for two linear-size subwords in the grammar to at most 30, the final runtime of RNAfold is bounded by $O(n^3)$. This restriction is present in both the `C` reference version and the ADPfusion grammar where we make use of a combinator that restricts the maximal subword size based on subword sizes calculated by another combinator, thus giving us the required restriction.

[4] `RNAFold` hackage library: `http://hackage.haskell.org/package/RNAFold`

Given inputs of size 100 (nucleotides) or more, ADPfusion code is efficient enough to get within $\times 2$ – $\times 3$ of the `C` implementation. Fig. 6 shows runtimes for legacy ADP, ADPfusion, and `C` code.

8. Backtracking and algebra products

ADP introduced the concept of algebra products. A typical dynamic programming algorithm requires two steps: a forward step to fill the dynamic programming matrices and a backward or backtracking step to determine the optimal path from the largest input to the smallest sub-problems. For a CYK parser, the forward step determines if a word is part of the language while the backward step yields the parse(s) for this word.

This forces the designer of a DP algorithm to write the recurrences twice, and keep the code synchronized as otherwise subtle bugs can occur. Algebra products "pull" the backward step into the forward step. Considering the case of the optimal path and its backtrace, one writes `(opt ***backtrace)`, where `opt` is the algebra computing the score of the optimal answer, while `backtrace` is its backtrace, and `(***)` the algebra product operation. This yields a new algebra that can be used as any other.

It has the effect of storing with each optimal result the description of how it was calculated or some information derived from this description. This is conceptually similar to storing a pointer to the cell(s) used for the calculation of the optimal result.

The algebra product is a very elegant device that allows for simple extension of algorithms with proper separation of ideas. A backtrace does not have to know about scoring schemes as each answer for the first argument of `(***)` is combined with exactly one answer of the second argument. Adding, say, suboptimal results requires a change only to `opt` to capture more than one result, while co-optimal results are automatically available from the ADP definition of the algebra product.

The algebra product as used in ADP is, unfortunately, a problematic device to use in practice. While it allows for a simple design of algorithms and removes another source of potential bugs, it comes with a high runtime cost.

Consider an algorithm that calculates a large number of co- or sub-optimal results, like the `Nussinov78` algorithm in backtracking.

Standard implementations calculate the DP matrices in the forward step and then enumerate all possible backtraces within a certain range. The forward step does not change compared to just asking for the optimal result. The backward step, while tedious to get right, only has to deal with one backtrace at a time – unless they all have to be stored. ADP, on the other hand, stores *all* backtraces within its DP matrices. The memory cost is much higher as all answers – and all answers to sub-problems – that pass the objective function are retained within the matrices.

In addition, we can not use strict, unboxed arrays of Ints (or Floats or Doubles) if we store backtraces directly in the DP matrices.

For ADPfusion we prefer to have an explicit backtrace step. As a consequence, the programmer is faced with a slightly bigger task of defining the forward algebra and the backward algebra separately instead of just using the algebra product, but this is offset by the gains in runtime and memory usage. One can even use a version of the algebra product operation in the backward step to keep most of its benefits. In this case, the use of the algebra product becomes quite harmless as we no longer store each answer within the matrices. In terms of absolute runtime, this approach works out favorably as well. The costly forward phase (for RNAfold: $O(n^3)$) is as efficient as possible, while the less costly backtracking (for RNAfold: $O(n^2 * k)$, with k the number of backtracked results) uses the elegant algebra product device.

9. Technical details

9.1 Functional dependencies vs. type families

Type families [3] are a replacement for functional dependencies [21]. As both approaches provide nearly the same functionality, it is a good question why this library requires both: type families and functional dependencies. The functions to extract values from function arguments, collected in the type class `ExtractValue`, are making use of associated type synonyms as this provides a (albeit subjectively) clean interface.

The stream generation system, using the `StreamGen` and `PreStreamGen` type classes, is based on functional dependencies. The reasons are two-fold: (i) the replacement using type families does not optimize well, and (ii) functional dependencies allow for overlapping instances.

The type family-based version[5] of the ADPfusion library does not optimize well. Once a third argument, and hence nested `Boxes` come into play, the resulting code is only partially optimized effecting performance by a large factor. This seems to be due to insufficient compile-time elimination of `Box` data constructors. This problem is currently under investigation.

Using a fixed number of instances, say up to 10, would at best be a stop-gap measure since this restricts the user of the library to productions of at most that many arguments and leads to highly repetitive code.

As functional dependencies allow unlimited arguments, require only overlapping instances, and consistently produce good code, they are the better solution for now even though they are, in general, not well received[6].

9.2 Efficient memoization

The ADPfusion library is concerned with optimizing production rules independent of underlying data structures, lazyness, and boxed or unboxed data types. The author of a DP algorithm may choose the data structure most suitable for the problem and by giving an `ExtractValue` instance makes it compatible with ADPfusion. If priority is placed on performance, calculations can be performed in the ST or IO monad. The `PrimitiveArray`[7] library provides a set of unboxed array data structures that have been used for the algorithms in Sec. 7 as boxed data structures cost performance.

When first writing a new DP algorithm, lazy data structures can be used as this frees the programmer from having to specify the order in which DP tables (or other data structures) need to be filled. Once a proof-of-concept has been written, only small changes are required to create an algorithm working on unboxed data structures.

10. Conclusion and further work

High-level, yet high-performance, code is slowly becoming a possibility in Haskell. Projects like DPH [4] and Repa [22] show that one does not have to resort to unsightly low-level (and/or imperative-looking) algorithms anymore to design efficient algorithms. Furthermore, we can reap the benefits of staying within a language and having access to libraries and modern compilers compared to moving to a domain-specific language and its own compiler architecture.

The ability to write ADP code and enjoy the benefits of automatic fusion and compiler optimization are obvious as can be shown by the improvements in runtime as described in Sec. 7. Furthermore, one can design dynamic programming algorithms with the ease provided by ADP [10] and seamlessly enable further optimizations like strict, unboxed data structures, without having to rewrite the whole algorithm, or having to move away from Haskell.

With this new high-performance library at hand, we will redesign several algorithms. Our Haskell prototype of `RNAfold 2.0` allows us to compare performance with its optimized `C` counterpart. `RNAwolf` [18] is an advanced folding algorithm with a particularly complicated grammar including nucleotide triplets for which an implementation is only available in Haskell. `CMCompare` [17] calculates similarity measures for a restricted class of stochastic context-free grammars in the biological setting of RNA families.

Some rather advanced techniques that have become more appreciated in recent years (stochastic sampling of RNA structures [28] being one recent example) can now be expressed easily and with generality.

The ADP homepage [14] contains further examples of dynamic programming algorithms, as well as certain specializations and optimizations which will drive further improvements of this library. Of particular interest will be dynamic programming problems *not* in the realm of computational biology in order to make sure that the library is of sufficient generality to be of general usefulness.

The creation of efficient parsers for formal grammars, including CYK for context-free languages, is one such area of interest. Another are domain-specific languages that have rule sets akin to production rules in CFGs but do not require dynamic programming.

The ability to employ monadic combinators, which are available in the library, will be of help in many novel algorithmic ideas. We ignored the monadic aspect, but the library is indeed completely monadic. The non-monadic interface hides the monadic function application combinator `(#<<)`, nothing more. This design is inspired by the `vector`[8] library.

Coming back to the title of "sneaking around *concatMap*", we can not claim complete success. While we have gained huge improvements in performance, the resulting library is rather heavyweight (requiring both, functional dependencies and type families, and by extension, overlapping, flexible, and undecidable instances). Unfortunately, we currently see no way around this. As already pointed out in the stream fusion paper [7, section 9], optimizing for `concatMap` is not trivial. Furthermore, we would need optimizations that deal well with partially applied functions to facilitate a faithful translation of ADP into high-performance code.

Right now, results along these lines seem doubtful (considering that the stream fusion paper is from 2007) to become available soon. In addition, our view of partitioning a subword allows us to employ certain specializations directly within our framework. We know of no obvious, efficient way of implementing them within the original ADP framework. The most important one is the ability to observe the index stack to the left of the current combinator making possible the immediate termination of a stream that fails definable criteria like maximal sum of sub-partition sizes.

The code generated by this library does show that we have achieved further separation of concerns. While algebraic dynamic programming already provides separation of grammar (search space) and algebra (evaluation of candidates and selection via objective function) as well as asymptotic optimization by partial tabulation, we can add a further piece that is very important in practice – optimization of constant overhead. While the application of Bellman's principle still has to happen on the level of the grammar and by proof, all *code optimization* is now moved into the ADPfusion library.

The ADPfusion library itself depends on low-level stream optimization using the stream fusion work [7, 25] and further code optimization via GHC [36] and LLVM [24]. Trying to expose cer-

[5] github: branch `tf`

[6] cf. "cons" on overlap: `http://hackage.haskell.org/trac/haskell-prime/wiki/OverlappingInstances`

[7] `http://hackage.haskell.org/package/PrimitiveArray`

[8] `http://hackage.haskell.org/package/vector`

tain compile-time loop optimizations either within ADPfusion or the stream fusion library seems very attractive at this point as does the potential use of modern single-instruction multiple-data mechanisms. Any improvements in this area should allow us to breach the final $\times 2$ gap in runtime but we'd like to close this argument by pointing out that it is now *easy* to come very close to hand-optimized dynamic programming code.

Availability

The library is BSD3-licensed and available from hackage under the package name ADPfusion: `http://hackage.haskell.org/package/ADPfusion`. The git repository, including the type families (`tf`) branch, is available on github: `https://github.com/choener/ADPfusion`.

Acknowledgments

The author thanks Robert Giegerich and the Practical Computer Science group at Bielefeld University (ADP), Ivo Hofacker (dynamic programming), Roman Leshchinskiy (`vector` library, fusion, high-performance Haskell), and his family for letting him design, code and (mostly) finish it during the 2011-12 winter holidays. Several anonymous reviewers have provided detailed and valueable comments for which I am very thankful.

This work has been funded by the Austrian FWF, project "SFB F43 RNA regulation of the transcriptome"

References

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

[2] A. F. Bompfünewerer, R. Backofen, S. H. Bernhart, J. Hertel, I. L. Hofacker, P. F. Stadler, and S. Will. Variations on RNA folding and alignment: lessons from Benasque. *Journal of Mathematical Biology*, 56(1):129–144, 2008.

[3] M. M. Chakravarty, G. Keller, and S. Peyton Jones. Associated Type Synonyms. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP'05, pages 241–253. ACM, 2005.

[4] M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP'07, pages 10–18. ACM, 2007.

[5] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP'00, pages 268–279. ACM, 2000.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT press, 2001.

[7] D. Coutts, R. Leshchinskiy, and D. 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.

[8] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis*. Cambridge Univ. Press, 1998.

[9] R. Giegerich and C. Höner zu Siederdissen. Semantics and Ambiguity of Stochastic RNA Family Models. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(2):499–516, 2011.

[10] R. Giegerich and C. Meyer. Algebraic Dynamic Programming. In *Algebraic Methodology And Software Technology*, volume 2422, pages 243–257. Springer, 2002.

[11] R. Giegerich and P. Steffen. Challenges in the compilation of a domain specific language for dynamic programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1603–1609. ACM, 2006.

[12] R. Giegerich, C. Meyer, and P. Steffen. Towards a Discipline of Dynamic Programming. *Informatik bewegt, GI-Edition-Lecture Notes in Informatics*, pages 3–44, 2002.

[13] R. Giegerich, C. Meyer, and P. Steffen. A Discipline of Dynamic Programming over Sequence Data. *Science of Computer Programming*, 51(3):215–263, 2004.

[14] R. Giegerich et al. Algebraic Dynamic Programming Website. `http://bibiserv.techfak.uni-bielefeld.de/adp/`, 2004.

[15] D. Grune and C. J. Jacobs. *Parsing techniques: a practical guide*. Springer-Verlag New York Inc, 2008.

[16] 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/Chemical Monthly*, 125(2):167–188, 1994.

[17] C. Höner zu Siederdissen and I. L. Hofacker. Discriminatory power of RNA family models. *Bioinformatics*, 26(18):453–459, 2010.

[18] C. Höner zu Siederdissen, S. H. Bernhart, P. F. Stadler, and I. L. Hofacker. A folding algorithm for extended RNA secondary structures. *Bioinformatics*, 27(13): 129–136, 2011.

[19] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 1–55. ACM, 2007.

[20] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[21] M. P. Jones. Type Classes with Functional Dependencies. *Programming Languages and Systems*, pages 230–244, 2000.

[22] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. 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.

[23] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech & Language*, 4(1):35–56, 1990.

[24] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[25] R. Leshchinskiy. Recycle Your Arrays! *Practical Aspects of Declarative Languages*, pages 209–223, 2009.

[26] R. Lorenz, S. H. Bernhart, C. Höner zu Siederdissen, H. Tafer, C. Flamm, P. F. Stadler, and I. L. Hofacker. ViennaRNA Package 2.0. *Algorithms for Molecular Biology*, 6(26), 2011.

[27] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[28] M. Nebel and A. Scheid. Evaluation of a sophisticated SCFG design for RNA secondary structure prediction. *Theory in Biosciences*, 130:313–336, 2011. ISSN 1431-7613.

[29] R. Nussinov, G. Pieczenik, J. R. Griggs, and D. J. Kleitman. Algorithms for Loop Matchings. *SIAM Journal on Applied Mathematics*, 35(1):68–82, 1978.

[30] B. O'Sullivan, D. B. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2009.

[31] S. Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP'07, pages 327–337. ACM, 2007.

[32] E. Rivas, R. Lang, and S. 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.

[33] G. Sauthoff, S. Janssen, and R. 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.

[34] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Co., Inc., 1983.

[35] P. Steffen. *Compiling a domain specific language for dynamic programming*. PhD thesis, Bielefeld University, 2006.

[36] The GHC Team. The Glasgow Haskell Compiler (GHC). `http://www.haskell.org/ghc/`, 2012.