Bridging the Algorithm Gap: A Linear-time Functional Program for Paragraph Formatting

Oege de Moor
Programming Research Group
Oxford University
oege@comlab.ox.ac.uk

Jeremy Gibbons
School of Computing & Math. Sciences
Oxford Brookes University
jgibbons@brookes.ac.uk

August 6, 1997

Abstract

In the constructive programming community it is commonplace to see formal developments of textbook algorithms. In the algorithm design community, on the other hand, it may be well known that the textbook solution to a problem is not the most efficient possible. However, in presenting the more efficient solution, the algorithm designer will usually omit some of the implementation details, thus creating an *algorithm gap* between the abstract algorithm and its concrete implementation. This is in contrast to the formal development, which usually presents the complete concrete implementation of the less efficient solution.

We claim that the algorithm designer is forced to omit some of the details by the relative expressive poverty of the Pascal-like languages typically used to present the solution; the greater expressiveness provided by a functional language allows the whole story to be told in a reasonable amount of space. We therefore hope to bridge the algorithm gap between abstract and concrete implementations, and thereby to facilitate communication between the two communities.

1 Introduction

The paragraph formatting problem [12] is a favourite example for demonstrating the effectiveness of formal methods, and two particularly convincing derivations can be found in [1, 13]. The algorithms derived in these references are applications of dynamic programming, and their time complexity is $O(min(wn, n^2))$, where w is the maximum number of words on a line, and n is the number of words to be formatted.

Among algorithm designers it is well-known that one can solve the paragraph problem in $O(n)$ time, independent of w [7, 8, 9, 10]. The presentations of these linear algorithms

This paper is Technical Report CMS-TR-97-03 from the School of Computing and Mathematical Sciences, Oxford Brookes University. It is currently being submitted for publication.

do however ignore some details (for instance that the white space on the last line does not count), thus creating an *algorithm gap* between the abstract algorithm and its concrete implementation. This contrasts with the formal developments cited above, which present concrete code, albeit for a less efficient solution.

This paper is an experiment in bringing formal methods and algorithm design closer together, through the medium of functional programming. It presents a linear-time algorithm for paragraph formatting in a semi-formal style. The algorithm is given as executable code; in fact, the LaTeX file used to produce this paper is an executable *Gofer* program. In writing this paper we hope to convince the reader that a functional style can be suitable for communicating non-trivial algorithms, without sacrificing rigour or clarity or introducing an algorithm gap.

Of course there exist algorithms that are difficult to express functionally. In fact, it came as a surprise to us that the algorithm presented here can indeed be implemented without resorting to any non-functional language features. One of us (OdM) first attempted to explain the algorithm (which is very similar to that in [8]) two years ago, and then implemented it in Modula-2; when that program was discussed at the Oxford *Problem Solving Club*, it met with a lot of disappointment because of the need for destructive updates. Only recently we realised how the algorithm could be expressed functionally. This paper is therefore also a contribution to the ongoing effort of determining what can be done efficiently in a purely functional style.

1.1 Preliminaries

We do not assume that the reader is an expert in functional programming; we explain the necessary syntax and standard functions of Gofer as we go. (Of course, some familiarity with a modern lazy functional language such as Gofer, Haskell, Miranda[1] or Hope would be helpful; but we hope that the notation is fairly self-evident.) In addition to the standard functions of Gofer, we use a number of other primitives, which are explained in this section.

fold1: The function `fold1` is related to the standard function `foldr`, but it operates on non-empty lists. Informally, we have

```
fold1 step start [a0,a1,...,an]
        =  a0 `step` (a1 `step` (... `step` start an))
```

(Here, '`[a0,a1,...,an]`' denotes a list, and writing the binary function `f` inside backwards quotes, '`` `f` ``', allows it to be used as an infix operator.) In words, `fold1` traverses a list from right to left, applying `start` to the last element, and summing from right to left, at each stage applying the function `step`. The formal definition of `fold1` is

```
>fold1 f g [a]   = g a
>fold1 f g (a:x) = f a (fold1 f g x)
```

[1] Miranda is a trademark of Research Software Ltd.

(The binary operator '`:`' is 'cons', prepending an element onto a list. The definition of `fold1` is by *pattern matching*: the first equation that matches the argument applies.)

scan1. The function `scan1` records all intermediate results of the computation of `fold1` in a list:

```
>scan1 f g = fold1 f' g'
>            where g' a   = [g a]
>                  f' a s = f a (head s) : s
```

(Here, the function `head` returns the first element of a non-empty list.) For example, the function `tails`, which returns all non-empty suffixes of its argument, can be defined

```
>tails = scan1 (:) (:[])
```

(The second argument `(:[])` to `scan1` is the binary operator '`:`' already supplied with its second argument, in this case the empty list; the function `(:[])` that results is the function taking an element into a singleton list containing just that element.)

The relationship between `fold1` and `scan1` is succinctly expressed in the so-called *accumulation lemma*:

$$\texttt{scan1 f g} \;=\; \texttt{map (fold1 f g) . tails}$$

(Here, the higher-order function `map` applies the function which is its first argument to every element of the list which is its second argument; the binary operator '`.`' is function composition.) The accumulation lemma will be useful towards the end of this paper.

single. The operator `single` tests whether its argument is a singleton list:

```
>single [a] = True
>single _   = False
```

(The pattern '`_`' matches any argument, thereby acting as a kind of 'else' clause.) It is thus similar to the standard Gofer function `null`, which tests for emptiness.

minWith. The function `minWith f` takes a list `x` and returns an element of `x` whose `f`-value is minimal. Formally, `minWith` can be defined in terms of `fold1`:

```
>minWith f = fold1 choice id
>            where choice a b = a, if f a <= f b
>                             = b, otherwise
```

2 Paragraph Formatting

In the paragraph problem, the aim is to lay out a given text as a paragraph in a visually pleasing way. A text is given as a list of words, and each word is a sequence of characters:

```
>type Txt = [Word]
>type Word = String
```

A paragraph is a sequence of lines, each of which is a sequence of words:

```
>type Paragraph = [Line]
>type Line = [Word]
```

The `width` of a line is defined to be its total length when the words are printed next to each other, with one space character between each consecutive pair of words:

```
>width :: Line -> Int
>width = fold1 plus length
>        where plus w n = length w + 1 + n
```

(The first line of this definition is a type declaration, specifying that the function `width` takes a `Line` and returns an `Int`.) The function `length` returns the number of elements in a list; this notion of width is appropriate for displaying paragraphs on a device where every character has the same width. The programs presented below can easily be modified to deal with proportional spacing, where each character has its own individual width. In that case, however, one will need constant-time access arrays, which are not available in all functional languages.

We define a global constant `maxw` for the maximum line width.

```
>maxw :: Int
>maxw = 80
```

Of course, for flexibility many of the functions we develop should be parameterized by maxw; however, a global constant makes the presentation of the algorithm less cluttered. It is straightforward to make maxw a parameter throughout.

The function fits checks whether a line fits the maximum line width.

```
>fits :: Line -> Bool
>fits xs = width xs <= maxw
```

In laying out a text as a paragraph, the maximum line width should never be exceeded. Our programs will halt with a run-time error if an individual word exceeds the maximum line width; in practical applications one would need to deal with such pathological inputs more graciously.

In addition to the maximum line width maxw, we also assume the existence of an optimum line width optw, another global constant.

```
>optw :: Int
>optw = 70
```

The optimum line width should of course be at most the maximum line width.

We define the notion of *visually pleasing* as the requirement that the width of each line in the paragraph is as close to the optimum line width as possible. Formally, we wish to minimise the total *waste* in a paragraph, which is a measure of the deviation from the optimum line width:

```
>waste :: Paragraph -> Int
>waste = fold1 plus (const 0)
>         where plus l n = linw l + n
>               linw l = (optw - width l)^2
```

Note that the last line gets treated as a special case: it does not count towards the total waste of the paragraph. This is achieved by the function const, which satisfies the equation const a b = a for any b.

The function pars takes a text and returns all possible layouts as a list of paragraphs:

```
>pars :: Txt -> [Paragraph]
>pars = fold1 next_word last_word
>        where last_word w = [ [[w]] ]
>              next_word w ps = map (new w) ps ++ map (glue w) ps

>new w ls      = [w]:ls
>glue w (l:ls) = (w:l):ls
```

(Here, the binary operator `++` is list concatenation.) That is, for each word we have the option of putting it on a new line at the beginning of an existing paragraph, or to glue it onto the front of the first line of an existing paragraph.

The problem that we seek to solve can now be specified as

```
>par0 :: Txt -> Paragraph
>par0 = minWith waste . filter (all fits) . pars
```

(The function `filter p` takes a list `x` and returns exactly those elements of `x` that satisfy the predicate `p`; the predicate `all p` holds of a list precisely when all elements of the list satisfy the predicate `p`.) This specification is an executable program that solves the problem, but its execution takes exponential time.

3 The Standard Algorithm

To obtain a more efficient solution, we observe that `new` is monotonic:

$$\texttt{waste ls} \leq \texttt{waste ls'} \;\Rightarrow\; \texttt{waste (new w ls)} \leq \texttt{waste (new w ls')}$$

Furthermore, we have that `glue` is monotonic in a slighter weaker sense:

$$\begin{aligned}&\texttt{waste (l:ls)} \leq \texttt{waste (l:ls')}\\ &\qquad\Rightarrow\; \texttt{waste (glue w (l:ls))} \leq \texttt{waste (glue w (l:ls'))}\end{aligned}$$

Note that neither of these two properties depends on the precise definition of the waste on individual lines (as returned by the function `linw`): all that matters is that the total waste is the sum of the waste on the individual lines.

Using the above two monotonicity properties, one may conclude that the following dynamic programming algorithm is a valid solution to the specification `par0` [2].

```
>par1
> = minWith waste . fold1 step start
>   where
>     step w ps = filter fitH (new w(minWith waste ps):map(glue w) ps)
>     start w   = filter fitH [ [[w]] ]
>fitH = fits . head
```

Note that `par1` is not equal to `par0`; in particular, if there is more than one optimal paragraph, `par0` and `par1` may return different (but still optimal) paragraphs. To express this refinement property formally we would have to generalize from functional programming to relational programming [4], a step that is beyond the scope of this paper.

For efficiency, we could represent the paragraph `(l:ls)` by the triple

```
  (l:ls, width l, waste ls)
```

(Because `ls` may be empty, we stipulate also that `waste [] = 0`.) The program resulting from this data refinement is as follows.

```
>par1'
> = the . minWith waste . fold1 step start
>   where
>     step w ps = filter fitH (new w(minWith waste ps):map(glue w) ps)
>     start w   = filter fitH [([[w]], length w,0)]
>     new w ([l],n,0)   = ([w]:[l], length w, 0)
>     new w p           = ([w]:ls , length w, waste p) where (ls,n,m) = p
>     glue w (l:ls,n,m) = ((w:l):ls, length w + 1 + n, m)
>     the (ls,n,m)      = ls
>     width_hd (ls,n,m) = n
>     waste_tl (ls,n,m) = m
>     linw_hd p = (optw - width_hd p)^2
>     waste ([_],_,_)   = 0
>     waste p           = linw_hd p + waste_tl p
>     fitH p            =  width_hd p <= maxw
```

This is the standard dynamic programming solution to the paragraph problem, and its time complexity is $O(min(wn, n^2))$ where w is the maximum number of words on a line and n is the number of words in the paragraph. Computational experiments confirm that this is an accurate estimate of the program's behaviour.

4 A Linear Algorithm

The standard algorithm does not make use of any special properties of `linw`, the function that returns the waste on an individual line. In fact, if nothing more is known about `linw`, it is not possible to improve upon the standard algorithm presented in the preceding section, as noted in [9]. Here we shall show that for the particular choice of `linw`, substantial improvements are possible.

4.1 Exploiting Concavity

First of all, observe that the function `linw` is *concave*, in the sense that

$$\texttt{linw (l++m) - linw l} \;\leq\; \texttt{linw (k++l++m) - linw (k++l)}$$

Consequently, the above monotonicity property of `glue` can be strengthened to:

$$\begin{array}{l}\texttt{waste (l:ls)} \leq \texttt{waste ((l++m):ms)} \\ \qquad \Rightarrow \;\; \texttt{waste (glue w (l:ls))} \leq \texttt{waste (glue w ((l++m):ls))}\end{array}$$

In words, this implication says that if a paragraph with a shorter first line is better than another paragraph, it will remain better when more words are glued to the first lines of both paragraphs. This kind of implication is known as a *dominance criterion* — dominance criteria are the basis for most improvements over straightforward dynamic programming.

We can exploit the dominance criterion to arrive at an improved definition of `step`. Note that `step w` maintains the property 'is in strictly increasing order of length of first line' of the list of candidate solutions. Now suppose that we have two formats `p` and `q`, in that order, in the list of candidate solutions; the first line of `q` is longer than the first line of `p`. Suppose also that `waste p` $\leq$ `waste q`. By the monotonicity property of `new` and the stronger property of `glue`, it follows that `q` may be safely discarded, because any candidate solution generated from `q` will always be beaten by the candidate solution generated in the same way from `p`. So we may define

```
step w ps = trim (filter fitH (new w (minWith waste ps):map (glue w) ps))
```

where the function `trim` discards the superfluous candidate solutions:

```
trim []  = []
trim [p] = [p]
trim (ps++[p,q]) = trim (ps++[p]),        if waste p <= waste q
                 = trim (ps++[p]) ++ [q], otherwise
```

This is not a valid definition in Gofer (because patterns involving `++` are not allowed), but `trim` is easily reexpressed in terms of the standard functions `last` and `init` (which return the last element and the remainder of a list, respectively); we omit the details. (Note that we could trim from the back, as here, or from the front. Later on we will see that trimming from the back is the better choice, because we will have a criterion for stopping early.)

Now note further that the result of a `trim` is in strictly decreasing order of waste, so the least wasteful candidate solution is the last in the list, and we can improve the definition of `step` to

```
step w ps = trim (filter fitH (new w (last ps):map (glue w) ps))
```

The resulting algorithm is an improvement over the standard solution, but it is still not linear, because at each stage the whole list of intermediate solutions is traversed.

4.2 Forecasting the future

To remedy this source of inefficiency, observe that at each stage of the algorithm, we keep a list of paragraphs of the following shape:

```
l0:p0
(l0++l1):p1
(l0++l1++l2):p2
...
```

and this list is in strictly decreasing order of cost.

Say that an element is *bumped* by its predecessor when it is eliminated in the computation of `trim`. Can we tell how much gluing is needed before a particular paragraph `p` is bumped by its predecessor?

We introduce a function `f :: Paragraph -> Int -> Int` such that

```
f p (length w + 1) = waste (glue w p)
```

One suitable definition of `f` is

```
f [l] n    = 0
f (l:ls) n = (optw - (n + width l))^2 + waste ls
```

In words, `f p n` is the total waste of paragraph `p` after a sequence of words whose width is `n` has been glued to the first line of `p`. Note that we do not check whether the maximum line width is exceeded, and so the notion of `waste` may be meaningless in terms of paragraphs. We allow negative values of `n` as arguments of `f`.

Using the function `f`, we can forecast when a paragraph `p` will become better than a paragraph `q` in the process of gluing more words to both paragraphs. Define the function `k` by

```
  k p q
=
  setmin {n | f p n <= f q n}  `min` (maxw - width (head q) + 1)
```

(This is not a Gofer definition — Gofer does not have sets, and besides, the quantification is over a potentially infinite set — but it does completely determine `k`.) After gluing a width of `(k p q)` to both `p` and `q`, paragraph `p` will be at least as good as `q`; furthermore, if we glue more than `(k p q)`, paragraph `p` will still be as good as `q` (by concavity), so `q` can be discarded. The second term in the definition of `k` reflects the fact that after gluing `(maxw - width (head q) + 1)`, paragraph `p` is always better than paragraph `q`, because the first line of `q` exceeds the maximum line width. It can happen that `k` returns a negative number, namely when `p` is itself better than `q` to start with.

Let us now formalise these observations as properties of `glue`. Starting with `f`, we have that

```
f (glue w p) n  = f p (n + 1 + length w)
```

Consequently, `k` satisfies

```
k (glue w p) (glue w q) = k p q - (1 + length w)
```

and therefore

$$\begin{array}{l} \texttt{k p q} < \texttt{k r s} \\ \qquad \Leftrightarrow\ \texttt{k (glue w p) (glue w q)} < \texttt{k (glue w r) (glue w s)} \end{array}$$

Finally, define the predicate `better` by

```
better w p q  =  waste (glue w p) <= waste (glue w q) ||
                 not (fitH (glue w q))
```

(The binary operator `||` is boolean disjunction; later on we use `&&`, which is boolean conjunction.) In words, `better w p q` states that, after gluing a word `w`, paragraph `p` will be better than paragraph `q`, either on grounds of waste or because the first line of `q` has become too long. Suppose `p = l0:ls0`, `q = (l0++l1):ls1`, `r = (l0++l1++l2):ls2`, and `k p q` $\leq$ `k q r`. We have the following property:

$$\texttt{better w q r} \ \Rightarrow\ \texttt{better w p q} \wedge \texttt{better w p r}$$

In words, this property says that whenever `q` gets better than `r` by gluing a word `w`, we have that `p` is better than both `q` and `r`. It follows that `q` can never be useful, and therefore, if we had a triple like `p`, `q` and `r` in the list of candidate solutions, `q` could be safely discarded.

We shall exploit this observation by maintaining the list of candidate solutions so that

1. the lengths of the first lines are in strictly increasing order;
2. the costs of the candidate solutions are in strictly decreasing order; and
3. the `k`-values of consecutive adjacent pairs of candidate solutions are in strictly decreasing order.

Such a list of candidate solutions is said to be *tight*.

To see how we can keep the list of candidate solutions tight, recall the definition of `step` from the previous section:

```
step w ps = trim (filter fitH (new w (last ps):map (glue w) ps))
```

We argued in Section 4.1 that `step w` maintains property 1. We now show how the other two properties are maintained.

Notice that all three properties are preserved when taking a subsequence of (that is, deleting some elements from) the list of candidate solutions. For properties 1 and 2 this is obvious. For property 3 it follows from the fact that $\texttt{k p q} \geq \texttt{k p r} \geq \texttt{k q r}$, provided that $\texttt{k p q} > \texttt{k q r}$ and `p`, `q` and `r` are in strictly increasing order of length of first line; this fact is not too hard to establish.

Suppose that `ps` satisfies property 3. Because `glue` respects the `k` order, the list `map (glue w) ps` also satisfies property 3. It is however not necessarily the case that `new w (last ps) : map (glue w) ps` satisfies property 3: the presence of the new element at the beginning may require some of the initial elements of `map (glue w) ps` to be removed. So, the cons operator `(:)` in the above expression is replaced by an operator `add`, which does the required pruning — by the reasoning given earlier, if we have three consecutive candidate solutions `p`, `q` and `r` with $\texttt{k p q} \leq \texttt{k q r}$, solution `q` can be discarded.

```
add p []  = [p]
add p [q] = [p,q]
add p ([q,r]++rs) = add p ([r]++rs), if k p q <= k q r
                  = [p,q,r]++rs,     otherwise
```

Now the list of candidate solutions `` new w (last ps) `add` map (glue w) ps `` satisfies property 3. Note that `` p `add` ps `` is a subsequence of `p:ps`, so property 1 is still maintained; for the same reason, properties 1 and 3 are both maintained by the `filter fitH` too.

The third component of tightness, property 2, is achieved by the function `trim` given earlier. Again, `trim` returns a subsequence of its argument, and so properties 1 and 3 are maintained.

Now, however, property 3 permits an optimization of `trim`, whereby we can stop trimming early. This was the reason for introducing the function `k` in the first place. Suppose that `ps` satisfies property 3; we claim that `trim ps` can be written

```
trim []  = []
trim [p] = [p]
trim (ps++[p,q]) = trim (ps++[p]),   if waste p <= waste q
                 = ps++[p,q],        otherwise
```

Here is the justification. Suppose that `r`, `s` are adjacent candidate solutions in `ps`, and `p`, `q` are adjacent candidate solutions in `ps`, and `r` is before `p` in the list. Then `k r s > k p q`, by property 3. Suppose also that `waste p > waste q`. Then the word `w` that has just been processed was too short for `p` to bump `q`, and so was also too short for `r` to bump `s` (because of the `k` ordering); that is, `waste r > waste s` too, and the initial segment of the list of candidate solutions ending with solution `q` already satisfies property 2. Thus, we have `trim (ps++[p,q]) = ps++[p,q]`, and the second recursive call to `trim` can be omitted, when `ps++[p,q]` satisfies property 3 and `waste p > waste q`.

Because we are now manipulating the list of candidate solutions at both ends, it will be profitable to use a symmetric set of list operations, where `head` and `last` get treated on the same footing. Such an implementation of lists is summarized in an appendix to this paper. Below, whenever we use symmetric lists, the familiar list operations are written using a dash: `head'`, `init'`, and so on, and the type of symmetric lists over `a` is written `SymList a`.

One point that we have not touched upon is how `k` can be efficiently implemented. This is an exercise in high-school mathematics. Note that, when `p` and `q` appear in that order in the list of candidate solutions, `p` cannot be a singleton: there is just one way of formatting a paragraph into a single line, and if that line fits it will be the last candidate solution because it has the longest first line. Therefore are just two cases to consider in computing `k p q`: when `q` is a singleton, and when `q` is not. Note that the second term `rq0 = maxw - width (head q) + 1` is always greater than zero.

Case `q` is a singleton: so `f q n` is zero for any `n`. Thus, the only value of `n` for which `f p n` $\leq$ `f q n` would be one for which `f p n` is zero; this can only happen when `n` $=$ `optw - width (head p)` and `waste (tail p)` $= 0$. This case therefore splits into two subcases: if `waste (tail p)` is zero, then `k p q` is the smaller of `rq0` and `optw - width (head p)`; otherwise, there are no suitable values of `n`, and `k p q` is simply `rq0`.

Case `q` is not a singleton: Note that, for non-singleton `p`,

```
f p n  =  waste p + n^2 - 2*n*(optw - width (head p))
```

and so `f p n` $\leq$ `f q n` precisely when

```
n  ≥  (waste p-waste q)/(2*(width (head q)-width (head p)))
```

(note that the divisor is non-zero, because of the ordering on lengths of first lines). Therefore, the first term in `k p q` is the ceiling of the fraction on the right-hand side of this inequation, and `k p q` itself is the smaller of this and `rq0`.

Thus, we can implement `k` as follows:

```
k p q
  =  (optw - wp0) `min` rq0, if single q && waste p1 == 0
  =  rq0,                    if single q
  =  ceildiv (waste p - waste q) (2*(wq0 - wp0)) `min` rq0,
                             otherwise
     where
       p0:p1 = p
       q0:q1 = q
```

```
        wp0 = width p0
        wq0 = width q0
        rq0 = maxw - wq0 + 1
```

where `ceildiv x y` returns the ceiling of the fraction `x/y`.

In outline, the program now is

```
par2 = last . fold1 step start
step w ps = trim (filter fitH (new w (last ps) `add` map (glue w) ps))
```

(Note that we have made use again of the fact that a tight list of paragraphs is in strictly decreasing order of waste, replacing the `minWith waste` in `par2` by `last`.) This new program is in fact quite efficient: computational experiments show that at each stage of the computation, only a very small number of candidate solutions are kept. Still, all candidate solutions get inspected each time `step` is evaluated, and this remains a source of inefficiency. To make progress, we shall have to remove the subexpressions `filter fitH` and `map (glue w)` from the definition of `step`.

4.3 Filtering

Because the list of candidate paragraphs is kept in increasing order of length of the first line, the `filter` is easily disposed of; it can be subsumed in the definition of `trim`. We introduce a function `trimf` such that

```
trimf = trim . filter fitH
```

Using the above observation, we transform this definition to

```
trimf []          = []
trimf [p]         = [],  if not (fitH p)
                  = [p], otherwise
trimf (ps++[p,q]) = trimf (ps++[p]),   if   waste p <= waste q
                                          || not (fitH q)
                  = ps++[p,q],         otherwise
```

In the next section we will get rid of the `map` from the definition of `step`, by making a change of representation under which `glue w` is the identity function. If we assume that the list operations (`head`, `tail`, `init` and `last`) take amortized constant time, then this gives an amortized linear-time algorithm for paragraph formatting. Every word of the paragraph contributes at most one new candidate solution, and the amount of work performed (by `add` and `trimf`) on the list of candidate solutions is proportional to the number of candidate solutions discarded.

4.4 Differencing

As suggested above, a change of representation which avoids the notion of gluing altogether will eliminate the occurrence of `map` from the definition of `step`.

Elimination of `glue` can be achieved by computing only the tail of each paragraph. As long as we have the original text available (which is the concatenation of the paragraph), all necessary quantities can be computed in terms of the tail alone:

```
length (head p) = length (concat p) - length (concat (tail p))
width (head p)  = width (concat p) - width (concat (tail p)) - 1
waste p         = 0, if single p
                = (optw - width (head p))^2 + waste (tail p),
                     otherwise
```

(Recall that we stipulated that `waste [] = 0`.) The exploitation of this type of equation is known as *differencing*. We shall represent a paragraph `p` by the triple `rep p` where

```
rep p = ((width.concat.tail) p, (waste.tail) p, (length.concat.tail) p)
```

It will be useful to have a type synonym for the new representation of paragraphs:

```
>type Par = (Width,Waste,Length)
>type Width = Int
>type Waste = Int
>type Length = Int

>width_tl = fst3
>waste_tl = snd3
>len_tl = thd3
```

(The functions `fst3`, `snd3` and `thd3` return the first, second and third components of a triple, respectively.)

On this representation, the function `glue w` is the identity function, as required. Before we go into the details of the implementation of other operators on paragraphs, we outline the structure of the final program.

The program presented below is based on the fact that a solution to `par0` is returned by `par3`, where

```
par3 :: Txt -> Paragraph
par3 ws
  = tile ws (map (length.concat.tail.par0) (tails ws), length ws)
```

The function `tile xs` produces the required solution by exploiting the differencing equation for `(length . head)`:

```
>tile :: Txt -> ([Length],Length) -> Paragraph
>tile ws ([],n)   = []
>tile ws (m:ms,n) = ws1 : tile ws2 (drop l (m:ms),m)
>                  where l = n - m
>                        (ws1,ws2) = splitAt l ws
```

(Here, `splitAt n x` is a pair of lists, the first element of the pair being the first `n` elements of `x` and the second element being the remainder; `drop n x` is the second component of `splitAt n x`.) The proof that this works is an induction over all tails of the argument, and a detailed exposition can be found in [3]. It is perhaps interesting to note that a program involving `tile` is the starting point for the papers by Hirschberg and Larmore [9]; for us, it is part of a final optimisation.

Adapting the algorithm developed in previous sections to the new representation of paragraphs, one can find functions `stepr` and `startr` — data refinements of `step` and `start` — such that

```
  fold1 stepr startr (map length ws)
 =
  (map rep (fold1 step start ws), width ws, length ws)
```

and so, by the accumulation lemma of the introductory section, which showed how the computation of `fold1 f g` on all tails can be written in terms of `scan1`,

```
  scan1 stepr startr (map length ws)
 =
  zip3 (map (map rep . fold1 step start) (tails ws),
        map width (tails ws),
        map length (tails ws))
```

(The function `zip3` 'zips' a triple of lists — here, all of the same length — into a list of triples in the obvious way.)

Let `zs = scan1 stepr startr (map length ws)`. Then

```
  length ws = thd3 (head zs)
```

and

```
  map (length . concat . tail . par0) (tails ws)
 =   { rep }
  map (len_tl . rep . par0) (tails ws)
 =   { par0 = last . fold1 step start }
  map (len_tl . last . map rep . fold1 step start) (tails ws)
 =   { above }
  map (len_tl . last . fst3) zs
```

The resulting program is

```
>par3 :: Txt -> Paragraph
>par3 ws
> = tile ws (map (len_tl.last'.fst3) zs, thd3 (head zs))
>   where zs = scan1 stepr startr (map length ws)
```

It remains to give appropriate definitions of `stepr` and `startr`. The definition of `startr` is

```
>startr :: Length -> (SymList Par, Width, Length)
>startr a = (cons' (0,0,0) nil',a,1), if a <= maxw
```

The definition of `stepr` mirrors that in the preceding section, except that all operations on paragraphs have been data-refined to the new representation of paragraphs. Those modifications are justified by the differencing equations stated above, and the following definitions are immediate consequences of those identities:

```
>stepr :: Length ->
>         (SymList Par, Waste, Length) ->
>         (SymList Par, Waste, Length)
>stepr w (ps,tw,tl)
> = (trimf (new (last' ps) `add` ps), tot_width, tot_len)
>   where
>     single p   = len_tl p == 0
>     waste p    = 0, if single p
>                = waste_tl p + (optw - width_hd p)^2, otherwise
>     width_hd p = tot_width - width_tl p - 1
>     tot_width  = w + 1 + tw
>     tot_len    = 1 + tl
```

The operator new adds a new line to the front of a paragraph. It is important that, in computing the waste of the tail of the newly created paragraph, we use the old width of the head, that is, without taking the new word w into account:

```
>     new p = (tw,0,tl), if single p
>           = (tw,waste_tl p + (optw-old_width_hd p)^2,tl), otherwise
>     old_width_hd p = tw - width_tl p - 1
```

The definition of trimf is not changed at all; it only needs to use the new implementation of width . head, namely width_hd. (Recall that the dashed list operations are the operations on symmetric lists, defined in the appendix.)

```
>     trimf ps_pq = ps_pq,      if null' ps_pq
>                 = nil',       if single' ps_pq && width_hd q > maxw
>                 = ps_pq,      if single' ps_pq
>                 = trimf ps_p, if waste p <= waste q ||
>                                  width_hd q > maxw
>                 = ps_pq,      otherwise
>                   where ps_p = init' ps_pq
>                         q    = last' ps_pq
>                         p    = last' ps_p
```

The definition of add is similarly unaffected. On an intuitive level, there seems to be a duality between the ways trimf and add operate, but we have been unable to bring this out in the code, partly because trimf also performs the filtering operation.

```
>     add p qr_rs = cons' p qr_rs, if single' qr_rs || null' qr_rs
>                 = add p r_rs,  if k p q <= k q r
>                 = cons' p qr_rs, otherwise
>                   where r_rs = tail' qr_rs
>                         q  = head' qr_rs
>                         r  = head' r_rs
>
```

Finally, the data-refined version of k becomes

```
>     k p q
>       =  (optw - wp0) `min` rq0, if single q && waste_tl p == 0
>       =  rq0,                    if single q
>       =  ceildiv (waste p-waste q) (2*(wq0-wp0)) `min` rq0,
```

```
>                                   otherwise
>          where
>            wp0 = width_hd p
>            wq0 = width_hd q
>            rq0 = maxw - wq0 + 1

>ceildiv n m = (n+m-1) `div` m
```

It is not hard to check that program par3 does indeed have (amortised) linear time complexity. This theoretical bound is confirmed in computational experiments, and for all but the smallest inputs, par3 outperforms the standard algorithm par1.

5 Haskell vs Modula-2 vs C++

We now have the ingredients for writing a program that has the same functionality as the Unix utility *fmt*, although its output will be far superior (*fmt* uses a naive greedy strategy, and the resulting paragraphs are *not* visually pleasing). We shall make use of the functions

```
parse :: String -> [Paragraph]
unparse :: [Paragraph] -> String
```

which are well-known text-processing primitives in functional programming [5]. Their definitions are included in an appendix to this paper. Using these primitives, our implementation of fmt takes a single line:

```
>fmt = unparse . map (par3 . concat) . parse
```

Joe Programmer may not be happy about this implementation of a high-quality fmt. Although there is no algorithm gap, one might expect a *performance gap* between the Gofer program and an implementation of the same algorithm in a more conventional language. To measure the performance gap we translated the Gofer program for fmt into Haskell, and compiled it. This Haskell program was compared to hand-coded Modula-2 and C++ implementations that are in close correspondence to the Gofer program presented here. The conventional programs (those in Modula-2 and C++) do make extensive use of destructive updates, however, and the implementation of symmetric lists is replaced by the standard circular array code for queues. All data structures in the conventional programs are therefore of fixed size. Appropriate size bounds were determined by experimentation. The conventional programs, and especially the Modula-2 program, are of course longer than the Haskell program, but this is mostly due to the unwieldy syntax, as the difference is only a factor of one and a half to two. Personally we found the conventional code much harder to write because it uses a lot of indexing in arrays, as opposed to the standard list processing functions in Haskell.

All three programs were compiled on a Sun Sparc-10 workstation. For Haskell we used the Glasgow compiler `ghc`, because it produces the best code of all Haskell compilers available. For Modula-2 we used the Sun compiler, and for C++ the Gnu compiler. The Haskell executable is, as expected, vastly larger than the Modula-2 code — they differ by about a factor of six — but the Haskell and C++ executables are of comparable size. In all cases we switched on all optimizers. This has a spectacular effect for the Haskell program: it ran more than three times faster than without the `-O2` switch.

To compare the performance of the two executables, we formatted the full text of Thomas Hardy's *Far from the madding crowd*, an ASCII file of approximately 780Kb [15]. The three programs were run to format this file for a maximum line width of 70 characters. The CPU time and memory usage were measured using the `time` command provided by the UNIX C-shell. The Modula-2 program is only two and a half times faster than the Haskell program, which is very encouraging; the C++ program is nearly five times faster again, which probably reflects the effort put into the respective compilers. As one might expect the Haskell program gobbles memory, and in fact it was run using the standard heap size provided by `ghc`, which is 4Mb. This memory usage does *not* increase for larger input files.

The table below summarises the above comparison: the first two columns compare the programs with respect to their textual length (lines and characters), the third column is the size of the executable (in Kbytes), and the last two columns show their execution time (in CPU seconds) in the experiment, and how much memory (in Kbytes) was used.

	lines	chars	size (Kb)	time (s)	memory (Kb)
Haskell	163	4640	410	45.2	4504
Modula-2	551	10005	74	18.6	356
C++	389	5832	328	3.9	368
Haskell/Modula-2	0.30	0.46	5.54	2.4	12.65
Haskell/C++	0.42	0.80	1.25	11.6	12.24

In summary, the performance gap is not all that great; it furthermore seems likely that advances in compiler technology will cancel the remaining advantages of languages like C++ and Modula-2 over Haskell in the next few years.

6 Discussion

This paper was an experiment in using a functional language for presenting a non-trivial algorithm in a semi-formal style. We personally believe that for a large class of problems, this style of presentation is adequate, at once closing the algorithm gap and reconciling algorithm design with formal methods. The comparison with the hand-coded conventional implementations indicates that for non-trivial algorithms like the one presented here, the performance gap is rather small too. There are, however, two unsatisfactory aspects of the material presented here:

- First, we are not entirely satisfied with the semi-formal style of this paper. Up to the introduction of `trim`, the program derivation is absolutely standard, and no invention is involved in synthesizing the program. That part of the paper could easily be cast in calculational form, given the right machinery. The invention of `k`, and its role in 'forecasting the future', is however rather *ad hoc*, and escapes, at present, an elegant calculational treatment. This is unsatisfactory, especially since the technique seems more generally applicable.

- Second, we are very dissatisfied with the way one has to program differencing in a functional language. In a sense this is the least interesting part of the programming process, and yet it is quite error-prone. Moreover, differencing destroys some of the delightful elegance that characterises the functional expression of the standard algorithm. Meta-programming features in the spirit of Paige's `invariant` construct [14] might be used to circumvent this problem, but unfortunately we do not know of any modern functional language that supports those ideas.

Finally, the algorithm presented here is representative of a large class of ingenious algorithms, collectively known under the name *sparse dynamic programming* [7]. It would be nice to see whether a generic treatment of this class of algorithms is possible, in the style of [6]. It seems that such a generic approach is within reach, but we have not investigated this in any depth.

References

[1] R. S. Bird. Transformational programming and the paragraph problem. *Science of Computer Programming*, 6(2):159–189, 1986.

[2] R. S. Bird. A calculus of functions for program derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison–Wesley, 1990.

[3] R. S. Bird and O. De Moor. List partitions. *Formal Aspects of Computing*, 5(1):61–78, 1993.

[4] R. S. Bird and O. De Moor. *Algebra of Programming*. International Series in Computer Science. Prentice–Hall, 1996.

[5] R. S. Bird and P. Wadler. *Introduction to Functional Programming*. International Series in Computer Science. Prentice–Hall, 1988.

[6] O. De Moor. A generic program for sequential decision processes. In M. Hermenegildo and D. S. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 982 of *Lecture Notes in Computer Science*. Springer–Verlag, 1995.

[7] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming II: Convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

[8] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64:107–118, 1989.

[9] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

[10] D. S. Hirschberg and L. L. Larmore. New applications of failure functions. *Journal of the Association for Computing Machinery*, 34(3):616–625, 1987.

[11] R. R. Hoogerwoord. A symmetric set of efficient list operations. *Journal of Functional Programming*, 2(4):505–513, 1992.

[12] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11:1119–1184, 1981.

[13] C. C. Morgan. *Programming from Specifications*. International Series in Computer Science. 2nd edition, Prentice–Hall, 1994.

[14] R. Paige. Programming with invariants. *IEEE Software*, 3(1):56–69, 1986.

[15] Gutenberg Project. Many original texts as ASCII files. Available by ftp from: `mrcnext.cso.uiuc.edu`.

Appendix: Symmetric Lists

The implementation of symmetric lists given below is explained in some depth in [11]. Briefly, a list `x` is represented as a pair of lists `(y,z)` such that `abs (y,z) = x`, where the *abstraction function* `abs` is defined by

```
abs (y,z) = y ++ reverse z
```

Moreover, the following invariant is maintained: if either of the two lists is empty, the other is empty or a singleton.

The operations below implement their non-symmetric counterparts in the sense that

```
head' = head . abs
abs . tail' = tail . abs
```

and so on. The implementation is such that each operation takes amortised constant time.

```
>type SymList a = ([a],[a])

>single' (x,y) = (null x && single y) || (single x && null y)

>null' ([],[]) = True
>null' _       = False

>nil' = ([],[])
```

```
>head' (x,y) = head x, if not (null x)
>            = head y, otherwise

>last' (y,x) = head x, if not (null x)
>            = head y, otherwise

>cons' a (x,y) = (a:x,y), if not (null y)
>              = ([a],x), otherwise

>snoc' a (y,x) = (y,a:x), if not (null y)
>              = (x,[a]), otherwise

>tail' (x,y) = ([],[])          , if null x
>            = (reverse y1, y0), if single x
>            = (tail x, y)     , otherwise
>              where (y0,y1) = splitAt (length y / 2) y

>init' (y,x) = ([],[]),          if null x
>            = (y0, reverse y1), if single x
>            = (y, tail x)     , otherwise
>              where (y0,y1) = splitAt (length y / 2) y
```

Appendix: Text Processing

The text processing package given below is explained in [5]. It provides primitives for converting between strings and lines, lines and words, and paragraphs and lines. In each case, the forward direction can be programmed using the generic solution `do`, and the backward conversion using `undo`. The definitions of `unlines`, `lines`, `unwords` and `words` have been commented out because they are already defined in the standard Gofer prelude. The function `id` is the identity function.

```
>undo :: a -> [[a]] -> [a]
>undo a = fold1 insert id
>         where insert xs ys = xs ++ [a] ++ ys

>do :: Eq a => a -> [a] -> [[a]]
>do a [] = [[]]
>do a x  = fold1 (break a) (start a) x
>          where break a b xs = []:xs,   if a == b
>                             = (b:head xs):tail xs, otherwise
>                start a b = break a b [[]]
```

```
*unlines = undo '\n'
*lines = do '\n'

*unwords = undo ' '
*words = filter (/=[]) . do ' '

>unparas :: [[[String]]] -> [[String]]
>unparas = undo []

>paras :: [[String]] -> [[[String]]]
>paras   = filter (/=[]) . do []

>parse    = paras . map words . lines
>unparse  = unlines . map unwords . unparas
```