Science of Computer Programming 35 (1999) 3–27

Science of Computer Programming

www.elsevier.nl/locate/scico

Bridging the algorithm gap: A linear-time functional program for paragraph formatting [1]

Oege de Moor [a,*], Jeremy Gibbons [b]

[a] *Programming Research Group, Oxford University, Oxford, UK*
[b] *School of Computing & Mathematical Sciences, Oxford Brookes University, Headington Campus, Oxford OX3 OBP, UK*

Communicated by M. Sintzoff; received 28 July 1997; received in revised form 16 February 1999

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 proceeds all the way to 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 would allow the whole story to be told in a reasonable amount of space. In this paper we use a functional language to present the development of a sophisticated algorithm all the way to the final code. We hope to bridge the algorithm gap between abstract and concrete implementations, and thereby facilitate communication between the constructive programming and algorithm design communities. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Paragraph formatting; Functional programming; Algorithm design; Sparse dynamic programming; Transformational programming

1. Introduction

The paragraph formatting problem [13] is a favourite example for demonstrating the effectiveness of formal methods. Two particularly convincing derivations can be found in [1, 15]. The algorithms derived in these references are applications of dynamic

[1] This paper is a revision of Technical Report CMS-TR-97-03 from the School of Computing and Mathematical Sciences, Oxford Brookes University.

* Corresponding author.

E-mail addresses: oege@comlab.ox.ac.uk (O. de Moor), jgibbons@brookes.ac.uk (J. Gibbons) 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 [8–11]. Typical presentations of these linear algorithms 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 also an executable *Haskell* 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 imperative language features. One of us (OdM) first attempted to explain the algorithm (which is very similar to that in [9]) in 1992, 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 Haskell as we go. (Of course, some familiarity with a modern lazy functional language such as Haskell, Miranda [1] or Hope would be helpful; but we trust that the notation is fairly self-evident.) In addition to the standard functions of Haskell, 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 'adding in' the

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

next element at each stage using the function `step`. The formal definition of `fold1` is

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

(The first line is a *type declaration*, stating that `fold1` takes three arguments – a binary function of type `a->b->b`, a unary function of type `a->b`, and a list of type `[a]` – and returns a value of type `b`. 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. Because neither equation matches the empty list, `fold1` is undefined there.)

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

```
>scan1 :: (a->b->b) -> (a->b) -> [a] -> [b]
>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 :: [a] -> [[a]]
>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 *scan lemma*:

```
scan1 f g = map (fold1 f g) . tails                                    (1)
```

(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 scan lemma will be useful towards the end of this paper.

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

```
>single :: [a] -> Bool
>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 Haskell 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 :: (a->Int) -> [a] -> a
>minWith f = fold1 choice id
>            where choice a b | f a < f b = a
>                             | otherwise = b
```

(The expression '`f a < f b`' here is a *guard*. The first equation for `choice` applies if the guard holds, and the second equation applies if it does not.)

2. Specifying the problem

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, each of which is a string, that 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 problem can be specified as

```
>par0 :: Txt -> Paragraph
>par0 = minWith cost . filter feasible . formats
```

or informally, to compute the minimum-cost format among all feasible formats. (The function `filter p` takes a list `x` and returns exactly those elements of `x` that satisfy the predicate `p`.) In the remainder of this section we formalise the three components `formats`, `feasible` and `cost` of this specification. The result will be an executable program, but one whose execution takes exponential time.

2.1. Paragraph formats

The function `formats` takes a text and returns all possible formats as a list of paragraphs:

```
>formats   :: Txt -> [Paragraph]
>formats   = 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 the last word alone there is just one possible format, and for each remaining word we have the option either of putting it on a new line at the beginning of an existing paragraph, or of gluing it onto the front of the first line of an existing paragraph.

2.2. *Feasible paragraph formats*

A paragraph format is feasible if every line fits:

```
>feasible :: Paragraph -> Bool
>feasible = all fits
```

(The predicate `all p` holds of a list precisely when all elements of the list satisfy the predicate `p`.)

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

```
>maxw :: Int
>maxw = 70
```

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.

A line `fits` if its width is at most the maximum line width:

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

In formatting 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.

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 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 different characters may have different widths. In traditional typesetting, proportionally spaced fonts still conform to an underlying ‘unit’ system, whereby the different character widths are all integer multiples of some common unit; we could cope with this generalization by counting units instead of characters. However, in modern digital typesetting, there need be no unit system; in that case, one would need constant-time access arrays, which are not available in all functional languages.

2.3. The cost of a paragraph format

In addition to the maximum line width `maxw`, the problem also depends upon an optimum line width `optw`, another global constant:

```
>optw :: Int
>optw = 63
```

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

A *visually pleasing* paragraph is one in which the width of each line in the paragraph is as close to the optimum line width as possible. More precisely, we wish to minimise the total *cost*, the sum of the squares of the deviations of the line widths from the optimum line width:

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

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

3. The standard algorithm

The specification in Section 2 makes an inefficient program because it maintains too many candidate solutions. It is a *generate and test* algorithm, of the form

```
best . filter ok . generate
```

The standard technique for improving algorithms of this form is to *promote* the test `filter ok` and the selection `best` inside the generator, so as to avoid generating unnecessary candidates. Standard theory [1–3] tells us what properties of `generate`, `ok` and `best` this requires. In our particular case it is sufficient that `new` is monotonic in its second argument:

$$\texttt{cost ls} \leqslant \texttt{cost ls'} \Rightarrow \texttt{cost (new w ls)} \leqslant \texttt{cost (new w ls')} \qquad (2)$$

and furthermore, that `glue` is monotonic in its second argument in a slightly weaker sense – namely, for paragraphs with the same first line:

$$\begin{aligned}&\texttt{cost (l:ls)} \leqslant \texttt{cost (l:ls')}\\ &\qquad \Rightarrow \texttt{cost (glue w (l:ls))} \leqslant \texttt{cost (glue w (l:ls'))}\end{aligned} \qquad (3)$$

Note that neither of these two properties depends on the precise definition of the cost of individual lines, as returned by the function `linc`; all that matters is that the total cost is the sum of the costs of the individual lines.

Using the above two monotonicity properties, the standard theory alluded to above concludes that the following dynamic programming algorithm is a valid solution to the specification `par0`. This is a well-rehearsed development, so we do not repeat it here.

```
>par1
>= minWith cost . fold1 step start
>  where
>   step w ps = filter fitH (new w (minWith cost 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 can benefit from performing a *tupling transformation* [17, 6] to avoid recomputing the width of the first line and the cost of the remaining candidate solutions. We represent the paragraph `(l:ls)` by the triple

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

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

```
>par1'
> = the . minWith cost . fold1 step start
>   where
>    step w ps = filter fitH (new w (minWith cost 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, cost 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
>   cost_tl (ls,n,m)  = m
>   linc_hd p         = (optw - width_hd p)^2
>   cost ([_],_,_)    = 0
>   cost p            = linc_hd p + cost_tl p
>   fitH p            = width_hd p <= maxw
```

4. Improving the standard algorithm

The algorithm at the end of Section 3 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. This algorithm is the final product of the typical textbook derivation. It does not make use of any special properties of `linc`, the function that returns the cost on an individual line. Indeed, if nothing more is known about `linc`, it is not possible to improve upon this algorithm, as noted in [10]. However, as we shall show in Sections 5–8, for our particular choice of `linc`, namely

```
linc l = (optw - width l)^2
```

substantial improvements are possible; in fact, we will end up with a program linear in n and independent of w.

In Section 5 we determine a *dominance criterion*, whereby some candidate solutions can be discarded because they are dominated by other 'better' solutions. After processing each word of the paragraph, this justifies 'trimming' the collection of candidate solutions to remove the dominated ones.

To obtain a linear-time solution we can afford to do at most an amortized constant amount of work for each word in the paragraph. Precisely, one new candidate solution is added for each word, so it suffices that the amount of work performed with each word is proportional to the number of candidate solutions discarded. The obvious implementation of the trimming operation introduced in Section 5 involves reconsidering every candidate solution for each word in the paragraph. In Section 6 we show that this is not necessary: trimming in fact results in removing some candidate solutions from the beginning of the list and some others from the end. Crucially, the solutions in the middle of the list need not be considered; at the beginning and at the end of the list, as soon as one undominated solution is found the trimming can stop.

That still leaves the `filter` and the `map` in the definition of the function `step`, each of which inspects all candidate solutions for each word of the paragraph. In Section 7 we replace the `filter` with a function that performs work proportional to the number of solutions discarded, independently of the number of solutions retained. In Section 8 we eliminate the `map (glue w)`, by making a change of representation under which `glue w` is the identity function. The resulting algorithm is linear in the paragraph length and independent of the line width.

5. Dominance criteria

In this section we determine a *dominance criterion* whereby some paragraph formats can be discarded because they are dominated by others; thus, fewer candidate solutions need be maintained at each step. Dominance criteria are the basis for most improvements over straightforward dynamic programming. In our case, the dominance criterion is a consequence of the fact that the function `linc` is *concave*, in the sense that

```
linc (l++m) - linc l ⩽ linc (k++l++m) - linc (k++l)
```

Consequently, the monotonicity property of `glue` (Eq. (3)) can be strengthened to

```
cost (l:ls) ⩽ cost ((l++m):ms)
      ⇒ cost (glue w (l:ls)) ⩽ cost (glue w ((l++m):ls))          (4)
```

In words, the concavity property says that appending a line `m` onto a line `l` has no worse an effect than appending `m` onto a longer line `k++l`, and the strengthened monotonicity property 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 – a cheaper paragraph with a shorter first line dominates a costlier paragraph with a longer first line.

5.1. Exploiting concavity

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 width 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 wider than the first line of `p`. Suppose also that `cost p ⩽ cost q`. Then `p` dominates `q`: by the monotonicity property of `new` (Eq. (2)) and the strengthened property of `glue` (Eq. (4)), it follows that `q` may be safely discarded, because any candidate solution generated from `q` will be no better than the candidate solution generated in the same way from `p`. So we may improve the definition of `step` to

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

where the function `trim` discards the dominated candidate solutions (namely, the formats `q` for which there is a format `p` appearing earlier in the collection with `cost p ⩽ cost q`):

```
trim []                              = []
trim [p]                             = [p]
trim (ps++[p,q]) | cost p <= cost q = trim (ps++[p])
                 | otherwise         = trim (ps++[p]) ++ [q]
```

This is not a valid definition in Haskell, because patterns involving ++ are not allowed. However, the pattern-matching is easily re-expressed 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. In Section 6 we will see that trimming from the back is the better choice, because we will develop a criterion for stopping trimming early.

5.2. Trimming introduces order

Now note further that the result of a `trim` is in strictly decreasing order of cost, so the cheapest candidate solution is the last in the list. We can therefore improve the definition of `step` further, by using `last` instead of `minWith cost`:

```
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 step the whole list of intermediate solutions is traversed.

6. Forecasting the future

To avoid having to traverse the whole list of candidate solutions at each step, we will develop a criterion for stopping trimming early. Observe that we maintain a list of paragraph formats with strictly increasing first line widths, and strictly decreasing costs.

Say that a candidate solution element is *bumped* by its predecessor when it is eliminated by `trim`. Can we forecast how much gluing is needed before a particular format `q` is bumped by its predecessor? If so, we may be able to stop trimming early: if a word shorter than the width forecasted for `q` has been glued, then `q` need not be considered for trimming.

6.1. The bump factor

We introduce a function `cg :: Paragraph -> Int -> Int` (for ‘cost-glue’) such that

```
cost (glue w p)  =  cg p (length w + 1)
```

One suitable definition of `cg` is

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

In words, `cg p (n+1)` is the total cost of the paragraph formed after a sequence of words whose width is `n` has been glued to the first line of paragraph `p`. Note that we do not check whether the maximum line width is exceeded, and so the notion of cost may be meaningless in terms of paragraphs. We allow negative values of `n` as arguments of `cg`.

Using the function `cg`, we can forecast when a paragraph `p` will bump a paragraph `q` in the process of gluing more words to both paragraphs. Define the function `bf` (for ‘bump factor’) by

```
  bf p q
=
  setmin {n | cg p n <= cg q n} ‘min‘ (maxw - width (head q) + 1)
```

(This is not a Haskell definition – Haskell does not have sets, and besides, the quantification is over a potentially infinite set – but it does completely determine `bf`.) After increasing by `bf p q` the widths of the first lines of both `p` and `q`, paragraph `p` will be at least as good as `q`; furthermore, if we increase the first line widths by more than `bf 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 `bf` reflects the fact that after increasing the first line widths by more than `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 `bf` returns a negative number, namely when `p` is better than `q` to start with.

6.2. *Using the bump factor*

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

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

Consequently, `bf` satisfies

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

and therefore

```
  bf p q < bf r s
⇔
  bf (glue w p) (glue w q) < bf (glue w r) (glue w s)                (5)
```

Finally, define the predicate `better` by

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

so that ‘`better w p q`’ is equivalent to ‘`1 + length w` $\geqslant$ `bf p 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 cost or because the first line of `q` has become too wid. Suppose `p = l0:ls0`, `q = (l0++l1):ls1`, `r = (l0++l1++l2):ls2`, and `bf p q`$\leqslant$`bf q r`. We have the following property:

```
better w q r ⇒ better w p q ∧ better w p r
```

In words, this property says that whenever `q` gets better than `r` by gluing a word `w`, then `p` is better than both `q` and `r` after gluing the same word. 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.

6.3. Tight lists of solutions

We shall exploit this observation by maintaining the list of candidate solutions so that, as well as the properties

(i) the widths of the first lines are in strictly increasing order, and
(ii) the costs of the candidate solutions are in strictly decreasing order,

holding as before, also the property

(iii) the `bf`-values of consecutive adjacent pairs of candidate solutions are in strictly decreasing order.

We call such a list of candidate solutions *tight*.

To see how we can keep the list of candidate solutions tight, recall the definition of `step` from Section 5.2:

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

We argued in Section 5.1 that `step w` maintains properties (i) and (ii). We now show how property (iii) is 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 (i) and (ii) this is obvious. For property (iii) it follows from the fact that `bf p q` $\geqslant$ `bf p r` $\geqslant$ `bf q r`, provided that `bf p q` $>$ `bf q r` and `p`, `q` and `r` are in strictly increasing order of width of first line; this fact is not too hard to establish.

Suppose that `ps` satisfies property (iii). Because `glue` respects the `bf` order (Eq. (5)), the list `map (glue w) ps` also satisfies property (iii). It is however not necessarily the case that `new w (last ps) : map (glue w) ps` satisfies property (iii): 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 definition of `step` is replaced by a *smart constructor* `add`, which does the required pruning – by the argument at the end of Section 6.2, if we have three consecutive candidate solutions `p`, `q` and `r` with `bf p q` $\leqslant$ `bf q r`, solution `q` can be discarded.

```
add p []                                 = [p]
add p [q]                                = [p,q]
add p ([q,r]++rs) | bf p q <= bf q r = add p ([r]++rs)
                  | otherwise            = [p,q,r]++rs
```

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

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

```
trim []  = []
trim [p] = [p]
trim (ps++[p,q]) | cost p <=  cost q = trim (ps++[p])
                 | otherwise         = ps++[p,q]
```

without a recursive call in the last clause. 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 `bf r s` > `bf p q`, by property (iii). Suppose also that `cost p` > `cost 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 `bf` ordering); that is, `cost r` > `cost s` too, and the initial segment ending with solution `q` of the list of candidate solutions already satisfies property (ii). Thus, we have `trim (ps++[p,q]) = ps++[p,q]` – the second recursive call to `trim` can be omitted.

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` are equally efficient. 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 prime – `head'`, `init'`, and so on – and the type of symmetric lists over `a` is written `SymList a`.

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 cost, replacing the `minWith cost` in `par2` by `last`.) This new program is in fact quite efficient: computational experiments show that at each step of the computation, only a very small number of candidate solutions is 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`; we do this in Sections 7 and 8.

6.4. Computing the bump factor

One point that we have not touched upon is how `bf` can be efficiently implemented. This is an exercise in high-school algebra. 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 widest first line. Therefore, there are just two cases to consider in computing `bf p q`: when `q` is a singleton, and when `q` is not. Recall the definition of `bf`:

```
bf p q
   = setmin {n | cg p n <= cg q n} ‘min‘
     (maxw - width (head q) + 1)
```

Note that the second term `rqh = maxw - width (head q) + 1` is always greater than zero.

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

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

```
  cg p n = cost p + n^2 - 2*n*(optw - width (head p))
```

and so `cg p n` $\leqslant$ `cg q n` precisely when

```
  n ⩾ (cost p-cost q)/(2*(width (head q)-width (head p)))
```

(the divisor is non-zero, because of the ordering on widths of first lines). Therefore, the first term in `bf p q` is the ceiling of this quotient, and `bf p q` itself is the smaller of this first term and `rqh`.

Thus, we can implement `bf` as follows:

```
  bf p q
    | single q && cost pt == 0 = (optw - wph) ‘min‘ rqh
    | single q                 = rqh
    | otherwise                = ceildiv (cost p - cost q)
                                         (2*(wqh - wph))
                                   ‘min‘ rqh

  where
    ph:pt = p
    qh:qt = q
    wph   = width ph
    wqh   = width qh
    rqh   = maxw - wqh + 1
```

where `ceildiv x y` rounds the fraction `x/y` up to the next integer.

7. Filtering

Because the list of candidate paragraphs is kept in increasing order of width of the first line, the `filter` is easily dealt with. The net effect of filtering is that the last few formats (namely, those with the widest first lines) are discarded, and the remainder are retained. Therefore, we can use `drop_nofit = droptail (not . fitH)` instead of `filter fitH`, where

```
droptail :: (a->Bool) -> [a] -> [a]
droptail p []                    = []
droptail p (xs++[x]) | p x       = droptail p xs
                     | otherwise = xs ++ [x]
```

Informally, `droptail p x` discards elements from the end of the list `x`, stopping when the list is empty or the last element does not satisfy predicate `p`.

8. Differencing

In this 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, trim` and `drop_nofit`) on the list of candidate solutions is proportional to the number of candidate solutions discarded.

8.1. A data refinement

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)
  | single p    = width (concat p)
  | otherwise   = width (concat p) - width (concat (tail p)) - 1
cost p
  | single p    = 0
  | otherwise   = (optw - width (head p))^2 + cost (tail p)
```

(Recall that we stipulated that `cost [] = 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)),
         cost (tail p),
         length (concat (tail p)))
```

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

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

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

Here, the functions `fst3`, `snd3` and `thd3` return the first, second and third components of a triple, respectively:

```
>fst3 (a,b,c) = a
>snd3 (a,b,c) = b
>thd3 (a,b,c) = c
```

On this representation, the function `glue w` is the identity function, as required:

```
rep (glue w p) = rep p
```

8.2. The overall structure

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)) (n-l)
>                    where l = n - m
>                          (ws1,ws2) = splitAt l ws
```

(Here, `splitAt l x` is a pair of lists, the first element of the pair being the first `l` elements of `x` and the second element being the remainder; `drop l x` is the second component of `splitAt l x`.) The proof that this works is an induction over all tails of the argument, and a detailed exposition can be found in [3]. The crucial observation is that in each occurrence of `tile ws ms n`, we have

```
ms = map (length.concat.tail.par0) (tails ws)
n  = length ws
   = length ms
```

It is perhaps interesting to note that a program involving `tile` is the starting point for the paper by Hirschberg and Larmore [10]; 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 scan lemma (Eq. (1)), which showed how the computation of `fold1 f g` on all tails can be written in terms of `scan1 f g`,

```
  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' in the obvious way a triple of lists, all of the same length, into a list of triples.)
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)
=   { scan lemma }
  map (len_tl . last . fst3) zs
```

The resulting program is below. (Recall that the primed list operations are the operations on symmetric lists, defined in Appendix A.)

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

8.3. Implementing the data refinement

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

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

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, Width, Length) ->
>       (SymList Par, Width, Length)
>stepr w (ps,tw,tl)
> = (trim (drop_nofit (new (last' ps) `add` ps)), tot_width, tot_len)
>    where
>      single p        = len_tl p = = 0
>      cost p
>        | single p    = 0
>        | otherwise   = cost_tl p + (optw - width_hd p)^2
>      width_hd p
>        | single p    = tot_width
>        | otherwise   = 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 cost 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 | single p  = (tw,0,tl)
>            | otherwise = (tw,cost_tl p + (optw-old_width_hd p)^2,tl)
>      old_width_hd p | single p = tw
>                     | otherwise= tw - width_tl p - 1
```

The definition of trim is not changed at all:

```
>      trim ps_pq  | null' ps_pq        = ps_pq
>                  | single' ps_pq      = ps_pq
>                  | cost p <= cost q   = trim ps_p
>                  | otherwise          = ps_pq
>                    where ps_p = init' ps_pq
>                          q    = last' ps_pq
>                          p    = last' ps_p
```

whereas drop_nofit is an implementation of droptail (not . fitH), using the new implementation width_hd of width . head.

```
>     drop_nofit ps_p  | null' ps_p            = ps_p
>                      | width_hd p > maxw     = drop_nofit ps
>                      | otherwise             = ps_p
>                        where ps = init' ps_p
>                              p  = last' ps_p
```

The definition of add is similarly unaffected.

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

Finally, the data-refined version of bf becomes

```
>     bf p q
>       | single q && cost_tl p == 0 = (optw - wph) `min` rqh
>       | single q                   = rqh
>       | otherwise                  = ceildiv (cost p-cost q)
>                                              (2*(wqh-wph))
>                                         `min` rqh
>         where
>           wph = width_hd p
>           wqh = width_hd q
>           rqh = maxw - wqh + 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.

9. Haskell 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 (standard Unix's *fmt* uses a naive greedy strategy, and the resulting paragraphs are *not* visually pleasing; however, the Gnu version of *fmt* uses the algorithm from [13]). 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 Haskell program and an implementation of the same algorithm in a more conventional language. To measure the performance gap we compared the Haskell program for `fmt` to a hand-coded C++ implementation that is in close correspondence to the program presented here. The conventional program in C++ does make extensive use of destructive updates, however, and the implementation of symmetric lists is replaced by an array implementation. Because the program only adds candidate solutions at one end of the list, we can implement it by declaring an array whose size is an upper-bound on the number of words in a paragraph, with two pointers that indicate the beginning and end of the symmetric list. (If we also added solutions at the other end, we would need to use the folklore circular array code for queues.) All data structures in the conventional program are therefore of fixed size. Appropriate size bounds were determined by experimentation. The conventional program is 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 third. 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.

In writing the C++ code, we attempted to apply all the standard tricks that good C++ programmers employ to speed up their programs. For example, index calculations were avoided through use of pointers, and we provided ample hints to the compiler through *const* declarations and *inline* directives. To check that we did indeed conform to good practice in writing the C++ program, we compared its performance to that of the Gnu *fmt* utility: for longer line lengths (over 90 characters), our code for the sophisticated algorithm is faster than *fmt*. (As explained before, the running time of the standard dynamic programming solution, as used in *fmt*, increases with the maximum line length.) By contrast, the Haskell program in this paper has *not* been fine-tuned for performance at all, and we directly compiled the LaTeX source of this paper. It follows that the performance measurements reported give an edge to C++.

All three programs were compiled on a 400 MHz Pentium II processor, with 128MB of RAM, running RedHat Linux with a 2.0.35 kernel. For Haskell we used version 4.01 of the Glasgow compiler `ghc`, because it produces the best code of all Haskell compilers available. The same code was also compiled with `hbc` (version `0.9999.4`), which also has a good reputation for speed and reliability. For C++ we used the Gnu compiler `g++`, version `2.90.27`. All three executables were reduced in size using the utility `strip`. The Haskell executables are, as expected, vastly larger than the C++ code – they differ by about a factor of 25. In all cases we switched on all optimizers. This has a spectacular effect for the `ghc` program: it ran more than four times faster

Table 1

	Lines	Chars	Size (kb)	Time (s)
Haskell (hbc)	183	4676	196	10.34
Haskell (ghc)	183	4676	453	4.72
C++	416	6310	8	0.43
Haskell (hbc)/(ghc)	1.00	1.00	2.31	2.19
Haskell (ghc)/C++	0.44	0.74	24.50	11.27

than without the optimisation switch. Indeed, this is where we claw back some of the gains obtained by hand-coded optimisations in the C++ code: the `ghc` compiler aggressively applies optimising program transformations [18].

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 [20]. The three programs were run to format this file for a maximum line width of 70 characters and an optimum width of 63. The CPU time was measured using the `time` command provided by the Linux `bash` shell. The `ghc` executable is about twice as fast as the `hbc` program, which shows how much can be achieved by automatic transformation of Haskell programs. The C++ program is eleven times faster again, which reflects the effort put into the respective compilers, and the fact that we did not bother to fine-tune the Haskell code.

Table 1 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 column shows their execution time (in CPU seconds).

In summary, the performance gap is not all that great; it furthermore seems likely that advances in compiler technology (illustrated by the difference between `hbc` and `ghc`) will cancel the remaining advantages of languages like C++ over Haskell in the next few years.

10. 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 the ‘bump factor’, 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 [16] such as those espoused by Smith [19] and Liu [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* [8]. It would be nice to see whether a generic treatment of this class of algorithms is possible, in the style of De Moor [7]. It seems that such a generic approach is within reach, but we have not investigated this in any depth.

Appendix A. Symmetric lists

The implementation of symmetric lists given below is explained in some depth in [12]. 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) | not (null x) = head x
>            | otherwise    = head y
```

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

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

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

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

Appendix B. 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 `format`, and the backward conversion using `unformat`. The definitions of `unlines`, `lines`, `unwords` and `words` have been commented out because they are already defined in the standard Haskell prelude. The function `id` is the identity function.

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

>format :: Eq a => a -> [a] -> [[a]]
>format a [] = [[]]
>format a x  = fold1 (break a) (start a) x
>       where break a b xs | a == b    = []:xs
>                          | otherwise = (b:head xs):tail xs
>       start a b = break a b [[]]

*unlines :: [String] -> String
*unlines = unformat '\n'

*lines :: String -> [String]
*lines = format '\n'

*unwords :: [String] -> String
*unwords = unformat ' '
```

```
*words :: String -> [String]
*words = filter (/=[]) . format ' '

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

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

>parse :: String -> [[[String]]]
>parse = paras . map words . lines

>unparse :: [[[String]]] -> String
>unparse = unlines . map unwords . unparas
```

References

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

[2] R.S. Bird, A calculus of functions for program derivation, in: D.A. Turner (Ed.), Research Topics in Functional Programming, University of Texas at Austin Year of Programming Series, Addison-Wesley, Reading, MA, 1990, pp. 287–308.

[3] R.S. Bird, O. De Moor, List partitions, Formal Aspects Comput. 5 (1) (1993) 61–78.

[4] R.S. Bird, O. De Moor, Algebra of Programming, International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1996.

[5] R.S. Bird, P.Wadler, Introduction to Functional Programming, International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[6] W.N. Chin, Automatic methods for program transformation, Ph. D. Thesis, Imperial College, London, 1990.

[7] O. De Moor, A generic program for sequential decision processes, in: M. Hermenegildo, D.S. Swierstra (Eds.), Programming Languages: Implementations, Logics, and Programs, Lecture Notes in Computer Science, Vol. 982, Springer, Berlin, 1995.

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

[9] Z. Galil, R. Giancarlo, Speeding up dynamic programming with applications to molecular biology, Theoret. Comput. Sci. 64 (1989) 107–118.

[10] D.S. Hirschberg, L.L. Larmore, The least weight subsequence problem, SIAM J. Comput. 16 (4) (1987) 628–638.

[11] D.S. Hirschberg, L.L. Larmore, New applications of failure functions, J. Assoc. Comput. Mach. 34 (3) (1987) 616–625.

[12] R.R. Hoogerwoord, A symmetric set of efficient list operations, J. Funct. Programm. 2 (4) (1992) 505–513.

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

[14] Y.A. Liu, T. Teitelbaum, Systematic derivation of incremental programs, Science of Computer Programming 24 (1) (1995) 1–39.

[15] C.C. Morgan, Programming from Specifications, International Series in Computer Science, 2nd edition, Prentice-Hall, Englewood cliffs, NJ, 1994.

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

[17] A. Pettorossi, Methodologies for transformations and memoing in applicative languages, Ph. D. Thesis, Department of Computer Science, Edinburgh, 1984.
[18] S.L. Peyton Jones, A.L.M. Santos, A transformation-based optimiser for Haskell, Science of Computer Programming 32 (1–3) (1998) 3–47.
[19] D.R. Smith, KIDS: A semi-automatic program development system, IEEE Trans. Software Eng. 16 (9) (1990) 1024–1043.
[20] T. Hardy, Far from the Madding Crowd, Gutenberg Project, 1994. Available at http://www.promo.net/pgl.