Science of Computer Programming 6 (1986) 159-189
North-Holland

TRANSFORMATIONAL PROGRAMMING AND THE PARAGRAPH PROBLEM

R.S. BIRD

Programming Research Group, Oxford University, Oxford OX1 3QD, United Kingdom

Communicated by M. Sintzoff
Received August 1984
Revised January 1985

Abstract. The problem of breaking paragraphs into lines can be formulated as an optimisation problem: the best arrangement of lines is one which minimises a certain definition of waste. Starting with a specification of this kind, we present a complete derivation, based on transformational programming, of two linear-time algorithms for the problem. The difference between the algorithms reflects different assumptions about the algebraic properties of waste functions: one algorithm employs a greedy strategy, while the other uses dynamic programming. Both algorithms are expressed as purely functional programs, and the advantages of a functional notation are illustrated in the derivations.

1. Introduction

Close inspection of the two paragraphs of Fig. 1 reveals that the same passage—from *Molloy* by Samuel Beckett [3]—has been broken into lines in different ways.

```
Should I set out on my motorcycle? This
was the question with which I began. I
had a methodical mind and never set out
on mission without prolonged
reflection as to the best way of setting
out. It was the first problem to solve,
at the outset of each enquiry, and I
never moved until I had solved it to my
satisfaction. Sometimes I took my
autocycle, sometimes the train,
sometimes the motor-coach, just as
sometimes too I left on foot, or on my
bicycle, silently, in the night. For
when you are beset with enemies, as I
am, you cannot leave on your autocycle,
even in the night, without being
noticed, unless you employ it as an
ordinary bicylce which is absurd.
```

```
Should I set out on my motorcycle?
This was the question with which I
began. I had a methodical mind and
never set out on a mission without
prolonged reflection as to the best
way of setting out. It was the first
problem to solve, at the outset of
each enquiry, and I never moved until
I had solved it to my satisfaction.
Sometimes I took my autocycle, sometimes
the train, sometimes the motor-coach,
just as sometimes too I left on foot,
or on my bicycle, silently, in the
night. For when you are beset with
enemies, as I am, you cannot leave on
your autocycle, even in the night,
without being noticed, unless you employ
it as an ordinary bicycle which is
absurd.
```

Fig. 1.

In both versions, the text is displayed in a fixed-width font, and each line is at most 40 characters long, but the algorithms used for formatting the text are based on different criteria. Suppose we define the *white space* of a line to be the difference between the maximum line width (here, 40 characters) and the actual width of the line. The paragraph on the left of Fig. 1 has been formatted so that the total amount of white space, excluding the white space of the last line, is minimised, while in the paragraph on the right it is the maximum amount of white space on a line, taken over all lines except the last, which is minimised. Thus the two paragraphs are the result of selecting, from among all possible paragraphs, two which minimise different definitions of a notion of waste.

Notice that the paragraph on the left is one line shorter than the one on the right. It turns out that the implementation of the two strategies requires different amounts of computational effort, greater in the case of the mini-max solution. Although finding a mini-max layout takes slightly longer, requires more space to be available for processing, and may lead to a longer paragraph than the mini-sum solution, it can be argued that the result is more aesthetically pleasing. Any subsequent full justification of the lines of the text has fewer spaces to distribute between the words on each line and the result is a more even looking paragraph. The justified versions of the two paragraphs are shown for comparison in Fig. 2.

The present paper has a number of objectives one of which is to present a complete development, from specification to efficient algorithms, for breaking paragraphs into lines by minimising various definitions of waste. This development is based on the method of transformational programming (Burstall and Darlington [8]) and a second, more fundamental objective is to illustrate that method on a nontrivial example. Even so, we only consider a comparatively simple version of the problem

```
Should  I  set  out  on  my  motorcycle?  This
was  the  question  with  which  I  began.  I
had  a  methodolical  mind  and  never  set  out
on    a    mission    without    prolonged
reflection  as  to  the  best  way  of  setting
out.  It  was  the  first  problem  to  solve,
at  the  outset  of  each  enquiry,  and  I
never  moved  until  I  had  solved  it  to  my
satisfaction.    Sometimes    I    took    my
autocycle,    sometimes    the    train,
sometimes    the    motor-coach,    just    as
sometimes  too  I  left  on  foot,  or  on  my
bicycle,  silently,  in  the  night.  For
when  you  are  beset  with  enemies,  as  I
am,  you  cannot  leave  on  your  autocycle,
even    in    the    night,    without    being
noticed,  unless  you  employ  it  as  an
ordinary bicycle which is absurd.
```

```
Should  I  set  out  on  my  motorcycle?
This  was  the  question  with  which  I
began.  I  had  a  methodical  mind  and
never  set  out  on  a  mission  without
prolonged  reflection  as  to  the  best
way  of  setting  out.  It  was  the  first
problem  to  solve,  at  the  outset  of
each  enquiry,  and  I  never  moved  until
I  had  solved  it  to  my  satisfaction.
Sometimes I took my autocycle, sometimes
the  train,  sometimes  the  motor-coach,
just  as  sometimes  too  I  left  on  foot,
or  on  my  bicycle,  silently,  in  the
night.  For  when  you  are  beset  with
enemies,  as  I  am,  you  cannot  leave  on
your  autocycle,  even  in  the  night,
without being noticed, unless you employ
it  as  an  ordinary  bicycle  which  is
absurd.
```

Fig. 2.

and ignore such important aspects such as the possibilities of proportional spacing and word hyphenation: for a more sophisticated line-breaking algorithm see Knuth and Plass [13]. In order to bring the exercise to a successful conclusion a number of methodological issues have to be faced and resolved. Chief among these is the problem of dealing with nondeterminism in specifications. Although there have been a number of attempts to include nondeterministic choice operators in specification languages (see e.g. Bauer et al. [1]), the associated semantics have proved rather complicated. We shall demonstrate that the use of such operators can often be avoided by raising the level of discourse to talk about *sets* of solutions. Only after a substantial part of the transformational development has been carried out at the set level is it necessary to descend to the element level in order to arrive at a final, deterministic solution.

A third objective of the paper concerns functional programming and its notations. The arguments in favour of a purely functional basis for transformational programming are well known [5, 6, 8] and all solutions to the line-breaking problem are expressed as purely functional programs. However, in order to reduce the notational verbosity of expressions without serious loss of readability, we argue the case for augmenting the repertoire of forms of functional composition with a small but powerful set of higher-level operators (such as those introduced in [5]). Most of these are well known and their importance fully recognised, but as yet there is no general agreement on specific syntax.

A final objective of the paper concerns the tactical use of the transformational method. How do we know at each stage what transformations to employ and to what extent can the process be automated? It turns out that the overall derivation falls naturally into a sequence of stages, each stage being devoted to transformations which achieve a certain limited aim. To a large extent these stages are problem independent and include such themes as: developing an inductive definition of a defined but non-recursive function, inferring the definition of a function from a given condition, and tabulating [4] the values of a recursive function. The last is particularly important in the present exercise since dynamic programming is an essential component of the mini-max version of the algorithm. Tabulation, that part of dynamic programming which ensures no function value is computed more than once, assumes increased importance in functional programs since recursively defined functions are often inefficient if their values are computed directly from the definitions. The imposition of an efficient tabulation scheme on the recursive definition of a function is, in general, quite difficult and the problem is one of active research.

The rest of this paper is organised as follows. Section 2 deals with notational issues and describes certain algebraic laws concerning the manipulation of expressions. Section 3 considers the problem of giving a formal specification of the line-breaking problem. Section 4 contains details of a preliminary transformation which is used to derive an executable version of the specification. Section 5 contains the main synthesis of a recursive function which satisfies the specification but is potentially much faster. Section 6 describes how this recursive definition can be implemented efficiently by a tabulation scheme. Section 7 considers a further abstract property of waste functions which guarantees the existence of a greedy solution to the problem. Section 8 deals with the problem of transforming this solution into one which processes sequences of characters. Finally, in Section 9 we present the two results of our endeavours, summarise the main steps of the complete derivation and draw some conclusions about the exercise.

2. Notation and preliminaries

In this section we introduce the notations used extensively in the rest of the paper, and also discuss some basic transformations for manipulating expressions. For an alternative account and further examples of the notation, see Bird [5]. Basically, we employ a functional style of expression very similar to that of a KRC [17]; anyone familiar with that language should have little trouble in assimilating the details. Like KRC, the notation is based on recursion equations and uses curried forms for function application; thus we write $f\,x$ instead of $f(x)$ and regard functions of two or more arguments as one-argument functions of higher type. Application is left associative, so $(mult\,x\,y)$ means $((mult\,x)\,y)$ and not $(mult\,(x\,y))$.

Certain primitive and definable functions can also be written as infix operators, e.g. $x+y$, $x<y$, and so on. In particular there are four operators which will appear frequently, and are described as follows:

$$\begin{array}{lrcl}
\textit{compose} & (f\circ g)\,x & = & f(g\,x),\\
\textit{map} & f*S & = & \{f\,x \mid x\in S\},\\
\textit{filter} & P:S & = & \{x\in S \mid P\,x\},\\
\textit{minimise} & f\downarrow S & = & \{x\in S \mid f\,x = min\,(f*S)\}.
\end{array}$$

Of these, *composition* is a familiar operation from mathematics, and *map* and *filter* are often employed in functional programs. *Minimise* is less common and is introduced only because it is relevant to the problem treated in the present paper. Put simply, the expression $f\downarrow S$ denotes the set of elements of S which minimise the integer-valued function f. As we do not wish to have complicated precedence rules governing the order of application of operators, we make it a ground rule that *all* operators have equal precedence and all are *right* associative. Thus, for example, $f*P:S$ means $f*(P:S)$. The only exception is *application* (the operator denoted by just a space) which has higher priority and is left associative. Parentheses are used in the normal way to change the order of priority and association, as well as just to improve readability.

As has already been indicated, sets are denoted with braces. Sequences are denoted, as in KRC, with square brackets. The null sequence is denoted by [] and the primitive prefixing operator (i.e. the LISP cons) is denoted by ';' (unlike KRC which uses ':', here reserved for the filter operator). We write $[a]$ in preference to $a;[]$ and, more generally, $[a1, a2, \ldots, an]$ in preference to $a1;a2;..an;[]$. The append operator & (KRC uses ++) can be defined by the equations

$$[]\,\&\,y = y,$$
$$(a;x)\,\&\,y = a;(x\,\&\,y)$$

which illustrates the use of pattern-matching on the left-hand sides of definitions. (By the right association rule of operators, the right-hand expression of the last equation could have been written without parentheses.) The map and filter operators can also be applied to sequences, in which case they possess the following inductive definitions:

$$f*[] = [],$$
$$f*(a;x) = (fa);(f*x);$$

$$P:[] = [],$$
$$P:(a;x) = \begin{cases} a;(P:x) & \textbf{if } Pa, \\ P:x & \textbf{otherwise.} \end{cases}$$

The last definition shows how we write conditional equations.

There are a number of algebraic laws relating the operators introduced so far. The seven which follow are all easily proved from the definitions above:

(L1) $f*g*S = (f\circ g)*S,$

(L2) $P:g*S = g*(P\circ g):S,$

(L3) $f{\downarrow}g*S = g*(f\circ g){\downarrow}S,$

(L4) $f*A\cup B = (f*A)\cup(f*B),$

(L5) $P:A\cup B = (P:A)\cup(P:B),$

(L6) $f{\downarrow}A\cup B = f{\downarrow}(f{\downarrow}A)\cup(f{\downarrow}B),$

(L7) $f{\downarrow}f{\downarrow}A = f{\downarrow}A.$

Law (L5), for instance, says that the filter of the union of two sets is the union of the filters. (L6) is perhaps not so obvious: it says that the f-minimising elements of the union of two sets can be obtained by first taking the f-minimising elements of each set separately, and then taking the f-minimising elements of the result. Law (L7) states that minimise is an idempotent operation. Two further simple properties of *minimise* will be used frequently:

$$f{\downarrow}\{\} = \{\},$$
$$f{\downarrow}\{x\} = \{x\}.$$

Both are immediate consequences of the definition of $\downarrow$. There are three further properties of the operators which will prove useful in Section 5:

(L8) $f * A \subseteq f * B$ whenever $A \subseteq B$,

(L9) $P : A \subseteq P : B$ whenever $A \subseteq B$.

In other words *map* and *filter* are *monotonic* with respect to set inclusion. The minimise operator is *not* monotonic, but we do have

(L10) $f \downarrow A \subseteq f \downarrow B$ whenever $A \subseteq B$ and $f * A = f * B$.

Law (L10) merits a proof: suppose $x \in f \downarrow A$, whence $x \in A$ and $f x = \mathit{min}\ (f * A)$; by assumption we also have $x \in B$ and $f x = \mathit{min}\ (f * B)$; hence $x \in f \downarrow B$.

Apart from the use of such rules, the remaining transformations follow the unfold–fold calculus developed in [8] and familiarity with that paper is assumed.

3. Specification

The problem of breaking text into lines was originally formulated by Naur [15] who gave essentially this specification:

A text, i.e. a non-empty sequence of words separated by blanks (BL) or new line charcters (NL), is to be re-structured according to the following rules:

(R1) Every two words are separated by exactly one BL or NL;

(R2) The first word is preceded by NL; the last character is neither BL nor NL;

(R3) Each line is at most M characters long (not counting NL); within this range it contains as many words as possible.

The input text is required to start with NL; further, no word must contain more than M characters.

Since then the above speciification has been used by a number of authors (e.g. Broy [7], Bauer et al. [2], Gries [12]) as the starting point for a formal derivation of a line-breaking algorithm. However, Meertens [14] has pointed out that, as it stands, the specification is at best ambiguous and at worst unsatisfiable. The trouble lies with the second clause of the rule (R3). The usual interpretation of (R3) adopted by previous authors is the 'Greedy' one

(R3′) No line of the output starts with a word that can go at the end of the previous line.

On the other hand, consider the 'Ydeerg' interpretation

(R3″) No line of the output ends with a word that can go at the start of the following line.

```
Greedy and Ydeerg       Greedy and Ydeerg
cannot always be        cannot always
satisfied               be satisfied
simultaneously.         simultaneously.
```

Fig. 3.

If 'Greedy' is a logical consequence of rules (R1), (R2) and (R3), then, by symmetry considerations, 'Ydeerg' must also be a logical consequence. But it is not always possible to satisfy both 'Greedy' and 'Ydeerg' as Fig. 3 testifies. Hence the specification is unsatisfiable.

One way out of the impasse is to abandon the loosely worded rule (R3) in favour of an alternative such as

(R3‴) The number of lines in the output must be as small as possible (consistent with the other constraints).

This new version permits either the 'Greedy' or the 'Ydeerg' interpretations but, unlike the earlier version, does not demand both. Unfortunately, therein lies its central weakness: any specification which allows a 'Ydeerg' solution is bad because it allows paragraphs to be laid out with short first lines in order to fill the last line. The formatting problem is essentially asymmetric since first lines should be filled but last lines need not be, and this requirement should be built into the specification.

We could, of course, just take the greedy version as the intended one. But then the present paper need not have been written. The alternative is to pursue the idea that a certain measure of waste must be minimised, and look for suitable definitions of waste which exclude the 'Ydeerg' solution. Two such definitions were described in the Introduction. One is to define waste to be the sum of the white space at the end of each line, except the last, (*waste*1, say) and the other is to define waste to be the maximum white space on any line except the last (*waste*2, say). Still other definitions are possible, of course. A second aspect of the problem is that we are concerned with efficient algorithms and should be prepared to sacrifice sophisticated measures of waste in the interests of producing one whose running time is linear in the length of the input. Such an algorithm can obviously be based on the greedy principle, but it is no longer apparent which of the measures of waste, if any, can be minimised by a greedy algorithm. One of the consequences of a formal approach to the problem is to expose the necessary algebraic properties of waste functions which guarantee the existence of efficient solutions.

Before moving on to a formal specification in the suggested manner, one should mention that neither measure of waste so far described guarantees a *unique* solution for every input text. The first two paragraphs of Fig. 4 have the same *waste*1 value, while the second two have the same *waste*2 value.

It follows that no formal specification will be determinate and so we have the freedom to choose particular solutions on the grounds of simplicity or efficiency.

```
minimising       minimising
waste1 does      waste1
not              does not
guarantee        guarantee
determinancy     determinancy

minimising       minimising
waste2 also      waste2
cannot           also cannot
guarantee        guarantee
determinancy     determinancy
```

Fig. 4.

In formulating a precise specification to the line-breaking problem, it is sensible to first move to a higher level of abstraction and rephrase the requirements in terms of data types which permit easier reasoning about their implications (Sufrin [16] calls such a step *representational abstraction*).

Suppose the type *word* is given, together with a function *length* which maps words into positive integers. Intuitively, the length of a word is the number of characters it contains. It may also correspond to a more general measure which takes into account the possibility that different characters have different sizes. Define a *line* to be a sequence of words, and the *width* of a line to be the sum of the lengths of the words it contains, plus a suitable value *sp* for each interword space, i.e.

$$\mathit{width}\,[w1, w2, \ldots, wk] \;=\; (k-1)\mathit{sp} + \sum_{1\leqslant j\leqslant k} \mathit{length}\, wj.$$

For concreteness we shall suppose $sp = 1$.

Finally, define a *paragraph* to be a sequence of *non-empty* lines, and suppose a function *waste* is given which maps paragraphs into nonnegative integers. The two waste functions described in Section 1 can be defined by

$$\mathit{waste}1\,(p\,\&\,[e]) \;=\; \sum_{e'\in p} (M - \mathit{width}\, e'),$$

$$\mathit{waste}2\,(p\,\&\,[e]) \;=\; \operatorname*{Max}_{e'\in p} (M - \mathit{width}\, e')$$

where M is some given positive integer.

Equivalently, we can define these functions by explicit recursions as follows:

$$\mathit{waste}1\,[e] \;=\; 0,$$

$$\mathit{waste}1\,(e;p) \;=\; (M - \mathit{width}\, e) + \mathit{waste}1\, p,$$

$$\mathit{waste}2\,[e] \;=\; 0,$$

$$\mathit{waste}2\,(e;p) \;=\; \mathit{max}\{M - \mathit{width}\, e,\, \mathit{waste}2\, p\}.$$

To be precise, one should now establish abstraction and representation functions between the types *word*, *line* and *paragraph* and the sequences of visible and separator (i.e. BL and NL) characters which actually constitute the text. This step will be postponed until Section 8. The specification can be rephrased in terms of words, lines and paragraphs as follows:

Given is a sequence x of words and a positive integer M such that no word in x has length greater than M. Required is some paragraph p such that

(R1) p 'destructured' is x; that is $x = \mathit{flatten}\, p$ where

$$\mathit{flatten}\ [\,] = [\,],$$

$$\mathit{flatten}\ (e;p) = e \mathbin{\&} \mathit{flatten}\, p,$$

(R2) each line e of p satisfies $\mathit{width}\ e \leq M$,

(R3) p minimises *waste.*

The above specification will not serve as a final version as it is not yet in a form suitable for manipulation. In order to defer the decision as to which paragraph should be selected, we recast the specification in terms of the *set of all* paragraphs which satisfy the constraints. Let us call this set (*formats* x). Its definition can be put in the following way:

$$\mathit{formats}\, x \;=\; \mathit{waste} \downarrow \mathit{ok} : \mathit{layouts}\, x, \tag{3.1}$$

$$\mathit{layouts}\, x \;=\; \{p \mid x = \mathit{flatten}\, p \ \mathbf{and}\ \mathit{nilfree}\, p\}, \tag{3.2}$$

$$\mathit{ok}\, p \;=\; \mathit{and}\ \{\mathit{width}\, e \leq M \mid e \in p\}, \tag{3.3}$$

$$\mathit{nilfree}\, p \;=\; \mathit{and}\ \{e \neq [\,] \mid e \in p\}, \tag{3.4}$$

$$\mathit{width}\, e \;=\; (\#e - 1) + \mathit{sum}(\mathit{length} * e). \tag{3.5}$$

As a commentary on this version we provide the following notes:

(i) (*formats* x) is defined to be the set of *waste*-minimising layouts of x which satisfy the *ok* property. This definition incorporates the condition (R3).

(ii) A layout p is *ok* if each line of p has width no greater than M, in accordance with (R2).

(iii) The layouts of x are the sequences p of lines such that p flattened is x (R1) and, moreover, no line in p is blank, in accordance with the requirement that paragraphs are sequences of non-empty lines.

(iv) As equivalent alternatives to (3.3) and (3.4) we give recursive definitions of *ok* and *nilfree*:

$$ok\,[\,] = true,$$

$$ok\,(e;p) = width\, e \leq M \textbf{ and } ok\, p,$$

$$nilfree\,[\,] = true,$$

$$nilfree\,(e;p) = e \neq [\,] \textbf{ and } nilfree\, p.$$

Though less succinct, these versions are more useful in derivations.

(v) The operator # in (3.5) gives the number of elements in a list.

4. First steps in derivation

With one or two changes in syntax the specification at the end of the previous section is very nearly a legal KRC program. Some reservation is necessary because, in the definition of *layouts x*, one would have to provide an explicit enumeration of the set of all possible lines from which to draw the elements *p*. If one did this, the KRC interpreter could at least begin to execute the specification. However, the computation would never terminate. The reason is that the search for even a single waste-minimising element requires examination of every member of *layouts x*; but as the process of generating these members does not terminate (even though the set is finite), no answers will be produced. The situation is the same as if we had asked for the minimum element of the list [1, 1, ...]. The interpreter cannot know the answer is 1 because it never completes the enumeration.

As the first step on the path to an efficient algorithm, we would like to produce an executable version of the specification which is guaranteed to terminate no matter how long it may take. To do this we must derive an alternative definition of layouts which will yield all its elements and then terminate. One way is to synthesise an *inductive* definition of the form

$$layouts\,[\,] = \cdots,$$

$$layouts\,(w;x) = \cdots layouts\, x \cdots .$$

The synthesis makes use of the following equations from the specification, numbered for convenience:

$$flatten\,[\,] = [\,], \tag{4.1}$$

$$flatten\,(e;p) = e \,\&\, flatten\, p, \tag{4.2}$$

$$[\,] \,\&\, p = p, \tag{4.3}$$

$$(w;e) \,\&\, p = w;e \,\&\, p, \tag{4.4}$$

$$nilfree\,[\,] = true, \tag{4.5}$$

$$\textit{nilfree}\,(e;p) \;=\; (e \neq []) \textbf{ and } \textit{nilfree}\,p, \tag{4.6}$$

$$\textit{layouts}\,x \;=\; \{p \mid x = \textit{flatten}\,p \textbf{ and } \textit{nilfree}\,p\}. \tag{4.7}$$

The strategy behind the derivation is basically one of instantiating the two forms for x in (4.7) and simplifying. This requires, in each case, a further instantiation of the two possible forms for p. Thus the activity is very similar to unification [9]. The first case is

$$\begin{aligned}
\textit{layouts}\,[] &= \{p \mid [] = \textit{flatten}\,p \textbf{ and } \textit{nilfree}\,p\} \\
&\qquad \text{(instantiating } x = [] \text{ in (4.7))} \\
&= \{[] \mid [] = \textit{flatten}\,[] \textbf{ and } \textit{nilfree}\,[]\} \\
&\quad \cup \{e;p \mid [] = \textit{flatten}\,(e;p) \textbf{ and } \textit{nilfree}\,(e;p)\} \\
&\qquad \text{(instantiating forms [] and } (e;p) \text{ for } p \text{ in (4.7))} \\
&= \{[]\} \cup \{e;p \mid [] = \textit{flatten}\,(e;p) \textbf{ and } \textit{nilfree}\,(e;p)\} \\
&\quad \text{(unfolding (4.1) and (4.5)).}
\end{aligned}$$

But

$$\begin{aligned}
&\{e;p \mid [] = \textit{flatten}\,(e;p) \textbf{ and } \textit{nilfree}\,(e;p)\} \\
&\quad = \{e;p \mid [] = (e \mathbin{\&} \textit{flatten}\,p) \textbf{ and } e \neq [] \textbf{ and } \textit{nilfree}\,p\}, \\
&\qquad \text{(unfolding (4.2) and (4.6))} \\
&\quad = \{\}
\end{aligned}$$

since $e \neq []$ implies $(e \mathbin{\&} x) \neq []$ for any x. Hence $\textit{layouts}\,[] = \{[]\}$.

In a similar manner we can deal with the inductive case:

$$\begin{aligned}
&\textit{layouts}\,(w;x) \\
&\quad = \{p \mid w;x = \textit{flatten}\,p \textbf{ and } \textit{nilfree}\,p\} \\
&\quad = \{[] \mid w;x = \textit{flatten}\,[] \textbf{ and } \textit{nilfree}\,[]\} \\
&\qquad \cup \{e;p \mid w;x = \textit{flatten}\,(e;p) \textbf{ and } \textit{nilfree}\,(e;p)\} \\
&\qquad \text{(instantiating } p \text{ as before)} \\
&\quad = \{e;p \mid w;x = (e \mathbin{\&} \textit{flatten}\,p) \textbf{ and } e \neq [] \textbf{ and } \textit{nilfree}\,p\} \\
&\qquad \text{(unfolding (4.2) and (4.6) in the second term, and noting the first} \\
&\qquad\ \ \text{term} = \{\}, \text{ since } w;x \neq [] = \textit{flatten}\,[]) \\
&\quad = \{(w';e);p \mid w;x = (w';e) \mathbin{\&} \textit{flatten}\,p \textbf{ and } \textit{nilfree}\,p\} \\
&\qquad \text{(since if } e \neq [], \text{ then } e \text{ must be of the form } (w';e))
\end{aligned}$$

$= \{(w;e);p \mid x = (e \mathbin{\&} \mathit{flatten}\ p) \textbf{ and } \mathit{nilfree}\ p\}$

(unfolding (4.4) and using the fact $a;x = b;y$ implies $a = b$ and $x = y$)

$= \{[w];p \mid x = ([\,] \mathbin{\&} \mathit{flatten}\ p) \textbf{ and } \mathit{nilfree}\ p\}$

$\cup \{(w;e);p \mid x = (e \mathbin{\&} \mathit{flatten}\ p) \textbf{ and } e \neq [\,] \textbf{ and } \mathit{nilfree}\ p\}$

(by case analysis: $e = [\,]$ and $e \neq [\,]$)

$= \{[w];p \mid x = \mathit{flatten}\ p \textbf{ and } \mathit{nilfree}\ p\}$

$\cup \{[w] \mathbin{\&} e);p \mid x = \mathit{flatten}\ (e;p) \textbf{ and } \mathit{nilfree}\ (e;p)\}$

(folding with (4.2) and (4.6) and using $[w] \mathbin{\&} e = w;e$)

$= \{\mathit{cons}\ [w]\ p \mid x = \mathit{flatten}\ p \textbf{ and } \mathit{nilfree}\ p\}$

$\cup \{\mathit{glue}\ [w]\ (e;p) \mid x = \mathit{flatten}\ (e;p) \textbf{ and } \mathit{nilfree}\ (e;p)\},$

where we introduce the two functions

$$\mathit{cons}\ e\ p = e;p,$$
$$\mathit{glue}\ e\ (e';p) = (e \mathbin{\&} e');p.$$

Recalling the definition of the $*$ operator from Section 2, the first term of the derived expression now simplifies to

$\mathit{cons}\ [w] * \{p \mid x = \mathit{flatten}\ p \textbf{ and } \mathit{nilfree}\ p\}$

$= \mathit{cons}\ [w] * \mathit{layouts}\ x,$

folding with (4.7). The second term becomes

$\mathit{glue}\ [w] * \{e;p \mid x = \mathit{flatten}\ (e;p) \textbf{ and } \mathit{nilfree}\ (e;p)\}$

$= \mathit{glue}\ [w] * \{p \mid x = \mathit{flatten}\ p \textbf{ and } p \neq [\,] \textbf{ and } \mathit{nilfree}\ p\}$

$= \mathit{glue}\ [w] * (\neq[\,]) : \mathit{layouts}\ x.$

We can simplify this by defining a minor variant of *layouts*:

$$\mathit{layouts}'\ x = (\neq[\,]) : \mathit{layouts}\ x.$$

An easy calculation shows

$$\mathit{layouts}'[\,] = \{\},$$
$$\mathit{layouts}'\ (w;x) = \mathit{layouts}\ (w;x).$$

Putting the above results together, we have obtained the following inductive definition of *layouts x*:

$$\begin{aligned}
\mathit{layouts}\,[\,] &= \{[\,]\},\\
\mathit{layouts}\,(w;x) &= (\mathit{cons}\,[w] * \mathit{layouts}\,x) \cup (\mathit{glue}\,[w] * \mathit{layouts'}\,x),\\
\mathit{layouts'}\,[\,] &= \{\},\\
\mathit{layouts'}\,(w;x) &= \mathit{layouts}\,(w;x).
\end{aligned}$$

The intuitive explanation of this algorithm is simple. The first word either goes on a new line in front or it goes at the beginning of the existing first line.

We now have an executable specification, albeit one which takes exponential time in the number of words in the input. The exponential nature of the algorithm is a direct consequence of the fact that the size of *layouts* x is 2^{n-1}, where n is the number of words in the input (reason: each word but the last is followed by a break or not). The job of the following section is to see what can be done to bring down this unacceptable cost.

Before ending the section, we note two identities relating the functions *cons* and *glue* which will be used in the next section:

$$\begin{aligned}
\mathit{glue}\,e \circ \mathit{cons}\,e' &= \mathit{cons}\,(e \,\&\, e'),\\
\mathit{glue}\,e \circ \mathit{glue}\,e' &= \mathit{glue}\,(e \,\&\, e').
\end{aligned}$$

The proofs are by simple calculation and are omitted.

5. The main synthesis

At this point the following executable version of the specification has been derived:

$$\begin{aligned}
\mathit{formats}\,x &= \mathit{waste}\downarrow \mathit{ok}\colon \mathit{layouts}\,x, && (5.1)\\
\mathit{layouts}\,[\,] &= \{[\,]\}, && (5.2)\\
\mathit{layouts}\,(w;x) &= (\mathit{cons}\,[w] * \mathit{layouts}\,x) \cup (\mathit{glue}\,[w] * \mathit{layouts'}\,x), && (5.3)\\
\mathit{layouts'}\,[\,] &= \{\}, && (5.4)\\
\mathit{layouts'}\,(w;x) &= \mathit{layouts}\,(w;x), && (5.5)\\
\mathit{ok}\,p &= \mathit{and}\,\{\mathit{width}\,e \leqslant M \mid e \in p\}. && (5.6)
\end{aligned}$$

The next step is to use these equations in the derivation of a more direct and efficient inductive definition of *formats.* First we have

$$\begin{aligned} \mathit{formats}\,[\,] &= \mathit{waste} \downarrow \mathit{ok} : \mathit{layouts}\,[\,] && \text{(by (5.1))}\\ &= \mathit{waste} \downarrow \mathit{ok} : \{[\,]\} && \text{(by (5.2))}\\ &= \mathit{waste} \downarrow \{[\,]\} && \text{(by (5.6))}\\ &= \{[\,]\}. \end{aligned}$$

The inductive case is

$$\begin{aligned} &\mathit{formats}\,(w;x)\\ &= \mathit{waste} \downarrow \mathit{ok} : \mathit{layouts}\,(w;x) && \text{(by (5.1))}\\ &= \mathit{waste} \downarrow \mathit{ok} : (\mathit{cons}\,[w] * \mathit{layouts}\,x) \cup (\mathit{glue}\,[w] * \mathit{layouts}'\,x) && \text{(by (5.3))}\\ &= \mathit{waste} \downarrow (\mathit{ok} : \mathit{cons}\,[w] * \mathit{layouts}\,x) \cup (\mathit{ok} : \mathit{glue}\,[w] * \mathit{layouts}'\,x)\\ &\qquad \text{(using law (L5) } P : A \cup B = (P : A) \cup (P : B) \text{ from Section 2)}\\ &= \mathit{waste} \downarrow (\mathit{waste} \downarrow \mathit{ok} : \mathit{cons}\,[w] * \mathit{layouts}\,x) \cup (\mathit{waste} \downarrow \mathit{ok} : \mathit{glue}\,[w] * \mathit{layouts}'\,x)\\ &\qquad \text{(using law (L6) } f \downarrow A \cup B = f \downarrow (f \downarrow A) \cup (f \downarrow B)\text{, from Section 2)}\\ &= \mathit{waste} \downarrow f1\,[w]\,x \cup f2\,[w]\,x, \end{aligned}$$

where we introduce the functions

$$f1\; e\, x = \mathit{waste} \downarrow \mathit{ok} : \mathit{cons}\; e * \mathit{layouts}\; x, \tag{5.7}$$

$$f2\; e\, x = \mathit{waste} \downarrow \mathit{ok} : \mathit{glue}\; e * \mathit{layouts}'\; x. \tag{5.8}$$

Further work on the problem must deal with $f1$ and $f2$ separately. We shall leave $f1$ aside for a while as it requires special treatment. Instead we concentrate on $f2$ and again derive an inductive definition. The base case $x = [\,]$ is

$$\begin{aligned} f2\; e\,[\,] &= \mathit{waste} \downarrow \mathit{ok} : \mathit{glue}\; e * \mathit{layouts}'\,[\,] && \text{(from (5.8))}\\ &= \mathit{waste} \downarrow \mathit{ok} : \mathit{glue}\; e * \{\} && \text{(from (5.4))}\\ &= \mathrm{waste} \downarrow \mathit{ok} : \{\} && \text{(definition of } *)\\ &= \mathit{waste} \downarrow \{\} && \text{(definition of :)}\\ &= \{\} && \text{(definition of } \downarrow). \end{aligned}$$

The inductive step is

$$
\begin{aligned}
& f2\ e\ (w;x) \\
&\quad = \mathit{waste}\downarrow \mathit{ok} : \mathit{glue}\ e * \mathit{layouts}'\ (w;x) && \text{(from (5.8))} \\
&\quad = \mathit{waste}\downarrow \mathit{ok} : \mathit{glue}\ e * \mathit{layouts}\ (w;x) && \text{(from (5.4))} \\
&\quad = \mathit{waste}\downarrow \mathit{ok} : \mathit{glue}\ e * (\mathit{cons}\ [w] * \mathit{layouts}\ x) && \text{(from (5.3))} \\
&\qquad\qquad \cup (\mathit{glue}\ [w] * \mathit{layouts}'\ x) \\
&\quad = \mathit{waste}\downarrow \mathit{ok} : ((\mathit{glue}\ e \circ \mathit{cons}\ [w]) * \mathit{layouts}\ x) \\
&\qquad\qquad \cup ((\mathit{glue}\ e \circ \mathit{glue}\ [w]) * \mathit{layouts}'\ x)
\end{aligned}
$$

(using law (L4) $f * (A \cup B) = (f * A) \cup (f * B)$ and law (L1) $f * g * A = (f \circ g) * A$ described in Section 2)

$$
\begin{aligned}
&\quad = \mathit{waste}\downarrow \mathit{ok} : (\mathit{cons}\ (e\ \&\ [w]) * \mathit{layouts}\ x) \\
&\qquad\qquad \cup (\mathit{glue}\ (e\ \&\ [w]) * \mathit{layouts}'\ x)
\end{aligned}
$$

(using the two identities on *cons* and *glue* described at the end of the last section)

$$
\begin{aligned}
&\quad = \mathit{waste}\downarrow (\mathit{waste}\downarrow \mathit{ok} : (\mathit{cons}\ (e\ \&\ [w]) * \mathit{layouts}\ x) \\
&\qquad\qquad \cup (\mathit{waste}\downarrow \mathit{ok} : (\mathit{glue}\ (e\ \&\ [w]) * \mathit{layouts}'\ x)
\end{aligned}
$$

(using laws (L5) and (L6) once more)

$$= \mathit{waste}\downarrow f1\ (e\ \&\ [w])\ x \cup f2\ (e\ \&\ [w])\ x,$$

finally folding with the definitions of $f1$ and $f2$.

Putting what we have together:

$$\mathit{formats}\ [] = \{[]\}, \tag{5.9}$$

$$\mathit{formats}\ (w;x) = \mathit{waste}\downarrow f1[w]\ x \cup f2\ [w]\ x, \tag{5.10}$$

$$f2\ e\ [] = \{\}, \tag{5.11}$$

$$f2\ e\ (w;x) = \mathit{waste}\downarrow f1\ (e\ \&\ [w])\ x \cup f2\ (e\ \&\ [w])\ x. \tag{5.12}$$

This leaves us with the task of simplifying definition (5.7) of the function $f1$. It is surprising that we have got so far without any knowledge whatsoever of the waste function, or indeed any knowledge of the predicate *ok*, but we can go no further. At this point some information about waste is needed because $f1$ cannot be treated satisfactorily without it. As the reader may check, no obvious inductive definition of $f1$ is possible. The best we can do in the way of manipulation is

$$\begin{aligned} f1\;e\,x &= \mathit{waste}\downarrow \mathit{ok} : \mathit{cons}\;e * \mathit{layouts}\;x \\ &= \mathit{waste}\downarrow \mathit{cons}\;e * (\mathit{ok} \circ \mathit{cons}\;e) : \mathit{layouts}\;x \\ &= \mathit{cons}\;e * (\mathit{waste} \circ \mathit{cons}\;e)\downarrow (\mathit{ok} \circ \mathit{cons}\;e) : \mathit{layouts}\;x \end{aligned}$$

using the laws (L2) $P:(f * A) = f * (P \circ f) : A$ and (L3) $g\downarrow(f * A) = f * (g \circ f)\downarrow A$ described in Section 2. Moreover, we do have

$$(\mathit{ok} \circ \mathit{cons}\;e)p \;=\; \mathit{ok}\,(\mathit{cons}\;e\,p) \;=\; \mathit{ok}\,(e;p) \;=\; \mathit{width}\;e \leqslant M \;\textbf{and}\; \mathit{ok}\,p$$

using the definition of *cons* and (5.6). Hence

$$f1\;e\,x = \begin{cases} \mathit{cons}\;e * (\mathit{waste} \circ \mathit{cons}\;e)\downarrow \mathit{ok} : \mathit{layouts}\;x & \textbf{if } \mathit{width}\;e \leqslant M, \\ \{\} & \textbf{otherwise.} \end{cases}$$

It would be pleasant if we had

$$(\mathit{waste} \circ \mathit{cons}\;e)\downarrow S \;=\; \mathit{waste}\downarrow S$$

for all sets S, for then we could continue and derive

$$\begin{aligned} f1\;e\,x &= \mathit{cons}\;e * \mathit{waste}\downarrow \mathit{ok} : \mathit{layouts}\;x && \textbf{if } \mathit{width}\;e \leqslant M, \\ &= \mathit{cons}\;e * \mathit{formats}\;x && \textbf{if } \mathit{width}\;e \leqslant M, \end{aligned}$$

folding with definition (5.1) of *formats*. However, this imposes too strong a restriction on waste functions. Fortunately, a weaker condition suffices to make progress: suppose

$$(\mathit{waste}\;p \leqslant \mathit{waste}\;q)\;\mathit{implies}\;(\mathit{waste}\;(e;p) \leqslant \mathit{waste}\;(e;q))$$

for all lines e satisfying width e ≤ M.

Let us call this *condition A*. Two easy proofs by induction show that the functions *waste*1 and *waste*2 described in Section 3 satisfy condition A. Indeed, a little reflection shows that the condition is quite reasonable and fairly mild. The important point is that if *waste* satisfies condition A, then any p which minimises *waste* also minimises

$$(\mathit{waste} \circ \mathit{cons}\;e),$$

since $(\mathit{waste} \circ \mathit{cons}\;e)\;p = \mathit{waste}\;(e;p)$. We know therefore that

$$(\mathit{waste} \circ \mathit{cons}\;e)\downarrow S \;\supseteq\; \mathit{waste}\downarrow S$$

for any set S, and we can use this to derive the following fact about $(f1\;e\,x)$ in the case $\mathit{width}\;e \leqslant M$:

$$\begin{aligned} f1\;e\,x &= \mathit{cons}\;e * (\mathit{waste} \circ \mathit{cons}\;e)\downarrow \mathit{ok} : \mathit{layouts}\;x \\ &\supseteq \mathit{cons}\;e * \mathit{waste}\downarrow \mathit{ok} : \mathit{layouts}\;x \\ &= \mathit{cons}\;e * \mathit{formats}\;x, \end{aligned}$$

folding with (5.1).

Summarising, we have

$$f1\ e\,x \begin{cases} \supseteq cons\ e * formats\ x & \textbf{if } width\ e \leqslant M, \\ = \{\} & \textbf{otherwise} \end{cases} \qquad (5.13)$$

provided *waste* satisfies condition A.

The last phase of this section is to use (5.7)-(5.13) in the construction of a *single* member, (*format x*) say, of (*formats x*). Thus a single solution to the line-breaking problem will be chosen, and the indeterminancy of the original specification finally exorcised. As a necessary generalisation we shall first derive an inductive definition of a function f such that

$$f\,e\,x \ \in\ waste\!\downarrow\! f1\ e\,x \cup f2\ e\,x$$

for all x and e satisfying $width\ e \leqslant N$. Having done this we can define $format\ (w;x) = f\,[w]\,x$, as this paragraph will be in $formats\ (w;x)$ by (5.10). The base case is

$$\begin{aligned} f\,e\,[\,] &\in\ waste\!\downarrow\! f1\ e\,[\,] \cup f2\ e\,[\,] \\ &=\ waste\!\downarrow\! f1\ e\,[\,] \cup \{\} \qquad \text{(by (5.11))} \\ &=\ f1\ e\,[\,], \\ &\qquad \text{(using (5.7) and the idempotence of } (waste\downarrow)) \\ &\supseteq\ cons\ e * formats\,[\,] \qquad \text{(by (5.13))} \\ &=\ cons\ e * \{[\,]\} \qquad \text{(by (5.9))} \\ &=\ \{[e]\} \qquad \text{(definition of } cons). \end{aligned}$$

Hence we can suppose $f\,e\,[\,] = [e]$.

For the inductive case we require

$$f\,e\,(w;x)\ \in\ waste\!\downarrow\! f1\ e\,(w;x) \cup f2\ e\,(w;x).$$

Moreover, by (5.12) we know

$$f2\ e\,(w;x)\ =\ waste\!\downarrow\! f1\,(e\,\&\,[w])\,x \cup f2\,(e\,\&\,[w])\,x.$$

In order to make use of this in the construction of $f\,e\,(w;x)$ we have to consider two cases:

Case 1: $width\ (e\,\&\,[w]) \leqslant M$. Here, suppose by induction that

$$f\,(e\,\&\,[w])\,x \in \text{waste}\!\downarrow\! f1\,(e\,\&\,[w])\,x \cup f2(e\,\&\,[w])\,x,$$
and so $f\,(e\,\&\,[w])\,x \in f2\ e\,(w;x)$.

Furthermore, $width\ (e\,\&\,[w]) \leqslant M$ implies $width\ e \leqslant M$, and so

$$\begin{aligned} f1\ e\,(w;x) &\supseteq cons\ e * formats\,(w;x) \qquad \text{(by (5.13))} \\ &= cons\ e * waste\!\downarrow\! f1[w]\,x \cup f2\,[w]\,x. \end{aligned}$$

Thus, as $f\,[w]\,x \in (\mathit{waste}\!\downarrow\! f1\,[w]\,x \cup f2\,[w]\,x)$ (by a second appeal to induction, noting $\mathit{width}\,[w] \leqslant M$), we have

$$e;f\,[w]\,x \;\in\; f1\;e\,(w;x).$$

Therefore, putting these facts together,

$$\{e;f\,[w]\,x, f\,(e\,\&\,[w])\,x\} \;\subseteq\; f1\;e\,(w;x) \cup f2\;e\,(w;x).$$

As preparation for the final step, we need to observe

$$\mathit{waste} * \{e;f\,[w]\,x\} \;=\; \mathit{waste} * f1\;e\,(w;x),$$

$$\mathit{waste} * \{f\,(e\,\&\,[w])\,x\} \;=\; \mathit{waste} * f2\;e\,(w;x),$$

since, by (5.7) and (5.8) *every* element of $(f1\;e\,x)$ has the same waste; similarly for $(f2\;e\,x)$. Law (L10) of Section 2 can thus be used to obtain

$$\begin{aligned} f\,e\,(w;x) \;&\in\; \mathit{waste}\!\downarrow\! f1\;e\,(w;x) \cup f2\;e\,(w;x) \\ &\supseteq\; \mathit{waste}\!\downarrow\!\{e;f\,[w]\,x, f\,(e\,\&\,[w])\,x\}. \end{aligned}$$

Hence we can define

$$f\,e\,(w;x) \;=\; \mathit{pick}\,(e;f\,[w]\,x)(f\,(e\,\&\,[w])\,x)$$

where we can suppose

$$\mathit{pick}\,p\,q \;=\; \begin{cases} p & \textbf{if } \mathit{waste}\,p < \mathit{waste}\,q, \\ q & \textbf{otherwise}. \end{cases}$$

Notice with this definition we have chosen *not* to split lines if the two *waste* values turn out to be the same. Replacing $<$ by $\leqslant$ in the definition of *pick* reverses this decision.

Case 2. $\mathit{width}\,(e\,\&\,[w]) > M$. In this case we have $f1\,(e\,\&\,[w])\,x = \{\}$ by (5.13); furthermore, an easy induction shows $f2\,(e\,\&\,[w])\,x = \{\}$ for all x. Hence $f2\;e\,(w;x) = \{\}$, and so

$$\begin{aligned} f\,e\,(w;x) \;&\in\; \mathit{waste}\!\downarrow\! f1\;e\,(w;x) \cup f2\;e\,(w;x) \\ &=\; \mathit{waste}\!\downarrow\! f1\;e\,(w;x) \\ &=\; f1\;e\,(w;x) \\ &\supseteq\; \{e;f\,[w]\,x\} \end{aligned}$$

by similar reasoning to Case 1. We can therefore define

$$f\,e\,(w;x) \;=\; e;f\,[w]\,x.$$

The result of the above manipulations is the following program;

$$
\begin{aligned}
&\mathit{format}\,[\,] = [\,],\\
&\mathit{format}\,(w;x) = f\,[w]\,x,\\
&f e\,[\,] = [e],\\
&f e\,(w;x) = \begin{cases} \mathit{pick}\,(e;f\,[w]\,x)(f\,(e\,\&\,[w])\,x) & \textbf{if } \mathit{width}\,(e\,\&\,[w]) \le M,\\ e;f\,[w]\,x & \textbf{otherwise,}\end{cases}\\
&\mathit{pick}\,p\,q = \begin{cases} p & \textbf{if } \mathit{waste}\,p < \mathit{waste}\,q,\\ q & \textbf{otherwise.}\end{cases}
\end{aligned}
$$

It is easy to see intuitively what is going on in this algorithm. The first argument e of f holds the current line; if the next word w will fit at the end of e, then we try the two alternatives of putting w on the line or not, and pick the better result. If w will not fit, all we can do is output e and optimise the rest of the paragraph.

The important methodological point about the sequence of transformations described in the present section is the treatment of nondeterminism. Use of an explicit choice operator, with its attendant semantic problems, is avoided by splitting the derivation into two phases: in the first phase manipulations are carried out on the set of *all* solutions; in the second, the derived definition of the set of all solutions, combined with the condition of membership, is used to derive a single deterministic solution.

6. Tabulation

Although a considerable amount of work went into deriving the final function f of the last section, the sad fact is the result is not significantly more efficient than the executable version of the specification derived at the end of Section 4. The reason is that in any demand for a value of f there will be many recalculations of subsidiary values and the whole process will still take exponential time. In order to see this more clearly, Fig. 5 shows the shape of the *dependency graph* of f. This is a graph showing which values of the function are required in recursive calls (see Bird [4] for further details).

Observe that nodes in the graph of the form $f[e][w\ldots]$ for $\#e > 1$ have constant indegree 1, whereas nodes of the form $f[w][w'\ldots]$ do not. This reflects the fact that values associated with the latter type are calculated more than once. To avoid these redundant computations we have to impose a *tabulation* scheme on the definition, whereby $f[w1][w2\ldots]$, $f[w2][w3\ldots]$, $f[w3][w4\ldots]$, ... are calculated just once and then stored in a suitable table for future use.

In general, the task of imposing an efficient tabulation scheme on a recursively defined function can be quite tricky, especially if one wants the result to be expressed

$$\begin{tikzcd}
 & & & f[w1][w2\ldots] \arrow[dl] \arrow[dr] & & \\
 & & f[w2][w3\ldots] \arrow[dl] \arrow[dr, no head] & & f[w1, w2][w3\ldots] \arrow[dlll] \arrow[dr] & \\
 & f[w3][w4\ldots] \arrow[dl] \arrow[dr, dashed, no head] & & f[w2, w3][w4\ldots] \arrow[dlll] \arrow[dr, dashed, no head] & & f[w1, w2, w3][w4\ldots] \arrow[dlllll] \\
f[w4][w5\ldots] \arrow[dr, dashed, no head] & & {} & & {} & \\
 & {} & & & &
\end{tikzcd}$$

Fig. 5.

as a purely functional program. (For examples of general techniques, see [4].) However, in the present case there is a simple solution. First, let us define the sequence of values to be tabulated:

$$\mathit{flist}\,[\,] = [\,],$$

$$\mathit{flist}\,(w;x) = f\,[w]\,x;\mathit{flist}\,x.$$

For non-empty x we have $\mathit{format}\,x = \mathit{hd}\ (\mathit{flist}\,x)$, where $\mathit{hd}\ (a;s) = a$. We can use this 'table' in the definition of f:

$$f e\,[\,] = [e],$$

$$f e\,(w;x)$$

$$= \begin{cases} \mathit{pick}(e;\mathit{hd}\ (\mathit{flist}\,(w;x)), f\,(e \,\&\, [w])\,x) & \textbf{if } \mathit{width}\,(e \,\&\, [w]) \leq M, \\ e;\mathit{hd}\ (\mathit{flist}\,(w;x)) & \textbf{otherwise.} \end{cases}$$

As it stands, the revamped definition is not much help: *flist* is not passed to the definition of f as a precomputed list. We can remedy this by adding an extra parameter t to f which contains the relevant portion of the table. In any call of $(f\,e\,x\,t)$ the value t will be equal to $(\mathit{flist}\,x)$; this means we can define f by

$$f e\,[\,]\,[\,] = [e],$$

$$f e\,(w;x)\,(p;t) = \begin{cases} \mathit{pick}\,(e;p, f\,(e \,\&\, [w])\,x\,t) & \textbf{if } \mathit{width}\,(e \,\&\, [w]) \leq M, \\ e;p & \textbf{otherwise.} \end{cases}$$

To ensure that the third argument of f does satisfy the condition, we simply have to redefine *flist* as follows:

$$\mathit{flist}\,[\,] = [\,],$$

$$\mathit{flist}\,(w;x) = (f\,[w]\,x\,t);t \quad \textbf{where } t = \mathit{flist}\,x.$$

In case the reader is not convinced that the above version is substantially more efficient than the former, we shall now show that, ignoring for a moment the time spent on evaluating *widths* and *wastes*, the algorithm is in fact *linear* in the number of words in the input. The recurrence relations for the running times of *flist* and *f* can be deduced directly from the above definitions and read

$$\begin{aligned}
time_{flist}[\,] &= \mathrm{O}(1),\\
time_{flist}(w;x) &= time_f([w], x, flist\,x) + time_{flist}(x) + \mathrm{O}(1),\\
time_f(e, [\,], [\,]) &= \mathrm{O}(1),\\
time_f(e, w;x, p;t)&\\
&= \begin{cases}\mathrm{O}(1) + time_f(e\,\&\,[w], x, t) & \textbf{if } width\,(e\,\&\,[w]) \leqslant M,\\ \mathrm{O}(1) & \textbf{otherwise}.\end{cases}
\end{aligned}$$

As no line can contain more than M words, it follows that $time_f([w], x, t) = \mathrm{O}(M)$. Hence

$$time_{flist}(x) = \mathrm{O}(Mn),$$

where n is the number of words in the input.

The last stage in the derivation of an efficient algorithm is to eliminate the recalculations of *waste* and *width*. Since we have not yet chosen a *waste* function, let us first concentrate on *width* calculations. Recall from Section 3 the definition of *width*:

$$width\,e = (\#e - 1) + sum(length * e).$$

We first recast this as an explicit recursion:

$$\begin{aligned}
width\,[w] &= length\,w,\\
width\,(w;e) &= 1 + length\,w + width\,e.
\end{aligned}$$

The way to optimise the calculation of widths is to rewrite the definition of f to include an extra argument d which equals the *width* of the current line. We omit the straightforward details which give

$$\begin{aligned}
f\,d\,e\,[\,]\,[\,] &= [e]\\
f\,d\,e\,(w;x)\,(p;t)&\\
&= \begin{cases} pick\,(e;p)(f\,(d+k+1)(e\,\&\,[w])\,x\,t) & \textbf{if } (d+k+1) \leqslant M\\ e;p & \textbf{otherwise}\end{cases}\\
&\qquad \textbf{where } k = length\,w.
\end{aligned}$$

In order to deal with *waste* we must choose a *waste* function, the only requirement being that it must satisfy condition A. Both *waste*1 and *waste*2 are acceptable, but for the aesthetic reasons outlined in Section 1 we shall use *waste*2 as our criterion.

Values of *waste*2 can be computed quickly by redefining f to return a *pair* of values, the *waste* as well as the paragraph. In the program which follows, the paragraph is extracted by using the selector function *fst*, where $fst\ (a, b) = a$:

$$\begin{aligned}
&format\,[\,] = [\,],\\
&format\,(w;x) = fst\,(hd\,(flist\,(w;x))),\\
&flist\,[\,] = [\,],\\
&flist\,(w;x) = (f\,(length\ w)\,[w]\,x\,t);t \quad \textbf{where } t = flist\ x,\\
&f\,d\,e\,[\,]\,[\,] = ([e], 0),\\
&f\,d\,e\,(w;x)(pc;t)\\
&\quad = \begin{cases} pick\,(add\,d\,e\,pc)(f\,(d+k+1)(e\,\&\,[w])\,x\,t) & \textbf{if } (d+k+1) \leq M,\\ (add\,d\,e\,pc) & \textbf{otherwise,}\end{cases}\\
&\quad \textbf{where } k = length\ w,\\
&add\,d\,e\,(p, c) = (e;p, max\,\{M-d, c\}),\\
&pick\,(p1, c1)(p2, c2)) = \begin{cases} (p1, c1) & \textbf{if } c1 < c2,\\ (p2, c2) & \textbf{otherwise.}\end{cases}
\end{aligned}$$

7. The Greedy algorithm

Although the program derived at the end of Section 5 was transformed in the last section to run in linear time, it is interesting to see what further conditions can be imposed on waste to simplify the program even more. In particular, it would be nice to avoid using the function *pick*. If one could prove, for instance, that

$$waste\ (f\,(e\,\&\,[w])\,x) \leq waste\ (e;f\,[w]\,x),$$

then function f can be defined simply by

$$\begin{aligned}
&f\,e\,[\,] = [e],\\
&f\,e\,(w;x) = \begin{cases} f\,(e\,\&\,[w])\,x & \textbf{if } width\,(e\,\&\,[w]) \leq M,\\ e;f\,[w]\,x & \textbf{otherwise.}\end{cases}
\end{aligned}$$

It is easy to see that the above version of the program embodies the greedy approach to a solution: words are buffered until the width of the buffered line gets too large, when it is output and the process starts again with the next word on a new line. Notice the program no longer depends on any notion of *waste*, so we are asking for properties of *waste* which allow us to ignore it altogether.

The condition on *waste* we need turns out to be

$$\begin{aligned} &\mathit{waste}\,((e1 \,\&\, e2 \,\&\, e3);p) \\ &\quad\leqslant \mathit{waste}\,((e1 \,\&\, e2);e3;p) \\ &\quad\leqslant \mathit{waste}\,(e1;(e2 \,\&\, e3);p) \end{aligned}$$

for all p and $e1$, e, 2, $e3$ satisfying *width* $(e1 \,\&\, e2 \,\&\, e3) \leqslant M$. Let us call this *condition B*. Intuitively, condition B says it is wasteful to split lines that do not have to be split (first inequality), and white space at the beginning of a paragraph is more wasteful than anywhere else (second inequality).

Lemma. *Supoose waste satisfies conditions* A *and* B. *Then*

$$\begin{aligned} &\mathit{waste}\,(f\,(e1 \,\&\, e2 \,\&\, e3)\; x) \\ &\quad\leqslant \mathit{waste}\,((e1 \,\&\, e2);f e3\; x) \\ &\quad\leqslant \mathit{waste}\,((e1;f\,(e2 \,\&\, e3)\; x) \end{aligned}$$

for all x *and* $e1$, $e2$, $e3$ *with width* $(e1 \,\&\, e2 \,\&\, e3) \leqslant M$.

Corollary. $\mathit{waste}\,(f\,(e \,\&\, [w])\; x) \leqslant \mathit{waste}\,(e;f\,[w]\; x)$.

Proof. Take $e1 = e$, $e2 = []$ and $e3 = [w]$ in the statememt of the lemma. □

Proof of the Lemma. The proof is by induction. Abbreviate the three terms in the statement of the lemma by lhs, mid and rhs respectively.

Case 1. $x = []$. Since $f e\,[] = e;[]$ we have

$$\begin{aligned} \text{lhs} &= \mathit{waste}\,(e1 \,\&\, e2 \,\&\, e3;[]), \\ \text{mid} &= \mathit{waste}\,((e1 \,\&\, e2);e3;[]), \\ \text{rhs} &= \mathit{waste}\,(e1;(e2 \,\&\, e3);[]), \end{aligned}$$

and lhs $\leqslant$ mid $\leqslant$ rhs by condition B.

Case 2: $x = w;x$. There are four subcases to consider; in each case we can suppose by induction that $f e\,(w;x) = f\,(e \,\&\, [w])\; x$ if *width* $(e \,\&\, [w]) \leqslant M$.

(i) $M <$ *width* $(e3 \,\&\, [w])$. Here

$$\begin{aligned} \text{lhs} &= \mathit{waste}\,((e1 \,\&\, e2 \,\&\, e3);f\,[w]\; x), \\ \text{mid} &= \mathit{waste}\,((e1 \,\&\, e2);e3;f\,[w]\; x), \\ \text{rhs} &= \mathit{waste}\,(e1;(e2 \,\&\, e3);f\,[w]\; x), \end{aligned}$$

and lhs $\leqslant$ mid $\leqslant$ rhs follows immediately from condition B.

(ii) $width\ (e3\ \&\ [w]) \leq M < width\ (e2\ \&\ e3\ \&\ [w])$. Here

$$\begin{aligned} \text{lhs} &= waste\ ((e1\ \&\ e2\ \&\ e3);f\ [w]\ x) \\ \text{mid} &= waste\ ((e1\ \&\ e2);f\ (e3\ \&\ [w])\ x), \\ \text{rhs} &= waste\ (e1;(e2\ \&\ e3);f\ [w]\ x). \end{aligned}$$

First, lhs ≤ mid by the induction hypothesis (second inequality); second

$$\text{mid} \leq waste\ ((e1\ \&\ e2);e3;f\ [w]\ x)$$

combining condition A with the fact that

$$waste\ (f\ (e3\ \&\ [w])\ x) \leq waste\ (e3;f\ [w]\ x)$$

(induction hypothese, first inequality). Finally, mid ≤ rhs by condition B (second inequality).

(iii) $width\ (e2\ \&\ e3\ \&\ [w]) \leq M < width\ (e1\ \&\ e2\ \&\ e3\ \&\ [w])$. Here,

$$\begin{aligned} \text{lhs} &= waste\ ((e1\ \&\ e2\ \&\ e3);f\ [w]\ x), \\ \text{mid} &= waste\ ((e1\ \&\ e2);f\ (e3\ \&\ [w])\ x), \\ \text{rhs} &= waste\ (e1\ \&\ f\ (e2\ \&\ e3\ \&\ [w])\ x), \end{aligned}$$

and lhs ≤ mid ≤ rhs by the induction hypothesis (second inequality).

(iv) $width\ (e1\ \&\ e2\ \&\ e3\ \&\ [w]) \leq M$.

$$\begin{aligned} \text{lhs} &= waste\ (f\ (e1\ \&\ e2\ \&\ e3\ \&\ [w])\ x), \\ \text{mid} &= waste\ ((e1\ \&\ e2);f\ (e3\ \&\ [w])\ x), \\ \text{rhs} &= waste\ (e1;f\ (e2\ \&\ e3\ \&\ [w])\ x). \end{aligned}$$

Now lhs ≤ mid by the induction hypothesis (first inequality), and mid ≤ rhs by the second inequality. This establishes the induction and the proof is complete. □

As we have seen, conditions A and B on *waste* together ensure the existence of a greedy solution to the problem, which is independent of the actual measure of *waste* envisaged in the specification. It is time to look once more at the two definitions of *waste* given in Section 3:

$$\begin{aligned} waste1\ [e] &= 0, \\ waste1\ (e;p) &= (M - width\ e) + waste1\ p; \end{aligned}$$

$$\begin{aligned} waste2\ [e] &= 0, \\ waste2\ (e;p) &= max\ \{M - \text{width}\ e, waste2\ p\}. \end{aligned}$$

As we have seen, both measures satisfy condition A; however, only *waste*1 satisfies condition B. The proof of

$$
\begin{aligned}
&\mathit{waste}1\ ((e1\ \&\ e2\ \&\ e3);\ p)\\
&\quad\leqslant\ \mathit{waste}1\ ((e1\ \&\ e2);e3;p)\\
&\quad\leqslant\ \mathit{waste}1\ (e1;(e2\ \&\ e3);p)
\end{aligned}
$$

divides into two cases: if $p=[]$, then

$$
\begin{aligned}
\text{lhs} &= 0,\\
\text{mid} &= M-\mathit{width}\ (e1\ \&\ e2),\\
\text{rhs} &= M-\mathit{width}\ e1,
\end{aligned}
$$

and lhs$\leqslant$mid$\leqslant$rhs as $\mathit{width}\ e1 \leqslant \mathit{width}\ (e1\ \&\ e2)$.

If $p \neq []$, then

$$
\begin{aligned}
\text{lhs} &= M-\mathit{width}\ (e1\ \&\ e2\ \&\ e3)+\mathit{waste}1\ p,\\
\text{mid} &= M-\mathit{width}\ (e1\ \&\ e2)+M-\mathit{width}\ e3+\mathit{waste}1\ p,\\
\text{rhs} &= M-\mathit{width}\ e1+M-\mathit{width}\ (e2\ \&\ e3)+\mathit{waste}1\ p,
\end{aligned}
$$

and lhs$\leqslant$mid$\leqslant$rhs by an easy calculation of the relevant widths.

A counterexample to condition B in the case of *waste*2 is to take $\mathit{width}\ e1=30$, $\mathit{width}\ e2=20$, $\mathit{width}\ e3=20$, $\mathit{waste}2\ p=10$ and $M=80$. We have

$$
\begin{aligned}
\mathit{waste}2\ ((e1\ \&\ e2);e3;p) &= \mathit{max}\ \{29, 60, 10\}=60,\\
\mathit{waste}2\ (e1;(e2\ \&\ e3);p) &= \mathit{max}\ \{50, 39, 10\}=50.
\end{aligned}
$$

So, line-breaking with the *waste*2 measure requires a dynamic programming solution, while with the *waste*1 measure the following program suffices:

$$
\begin{aligned}
&\mathit{format}\ []\ =\ [],\\
&\mathit{format}\ (w;x)\ =\ f\ [w]\ x,\\
&f e\ []\ =\ [e],\\
&f e\ (w;x)\ =\begin{cases} f\ (e\ \&\ [w])\ x & \textbf{if}\ \text{width}\ (e\ \&\ [w])\leqslant M,\\ e;f\ [w]\ x & \textbf{otherwise}.\end{cases}
\end{aligned}
$$

8. Low-level synthesis

The greedy algorithm developed in the last section, and the dynamic programming solution of Section 6 both take sequences of words as input and deliver sequences of lines as output. In reality, the input and output are made up of sequences of characters, divided into two classes: visible and separator. In this section we formulate the mappings, promised in Section 3, between visible and separator characters and the data types *word* and *line*. In addition, we consider certain ways of improving the running time of the resulting programs. We assume throughout this section that the function *width* is given by

$$\mathit{width}\,[w] = \mathit{length}\,w,$$

$$\mathit{width}\,(w;e) = \mathit{length}\,w + 1 + \mathit{width}\,e.$$

The abstraction function which converts character sequences into words is given by the function *inwords*, where

$$\mathit{inwords}\,[\,] = [\,], \tag{8.1}$$

$$\mathit{inwords}\,(a;t) = \begin{cases} \mathit{inwords}\,t & \textbf{if } a = \mathrm{BL} \textbf{ or } a = \mathrm{NL}, \\ \mathit{buffer}\,[a]\,t & \textbf{otherwise}, \end{cases} \tag{8.2}$$

$$\mathit{buffer}\,w\,[\,] = [w], \tag{8.3}$$

$$\mathit{buffer}\,w\,(a;t) = \begin{cases} w;\mathit{inwords}\,t & \textbf{if } a = \mathrm{BL} \textbf{ or } a = \mathrm{NL}, \\ \mathit{buffer}\,(w\,\&\,[a])\,t & \textbf{otherwise}. \end{cases} \tag{8.4}$$

The converse representation functions are given by

$$\mathit{outpara}\,[\,] = [\,], \tag{8.5}$$

$$\mathit{outpara}\,(e;p) = \mathrm{NL};\mathit{outline}\,e\,\&\,\mathit{outpara}\,p, \tag{8.6}$$

$$\mathit{outline}\,[w] = w, \tag{8.7}$$

$$\mathit{outline}\,(w;e) = w\,\&\,(\mathrm{BL};\mathit{outline}\,e). \tag{8.8}$$

The expression which yields the required solution is

$$\mathit{outpara}\,(\mathit{format}\,(\mathit{inwords}\,t)). \tag{8.9}$$

Both the greedy and dynamic programming definitions of *format* buffer lines on output. This is necessary in the case of the latter since the waste of each line must be calculated, but it is not necessary in the case of the greedy algorithm which requires storage only to determine the length of each word. The main aim of the present section is to justify the last remark by deriving a version of the greedy algorithm which economises on space.

As a first step, we rewrite the greedy version of the function *f* to include an extra parameter *d* which equals the width of the current line. The result is the program:

$$\mathit{format}\,[\,] = [\,], \tag{8.10}$$

$$\mathit{format}\,(w;x) = f\,(\mathit{length}\,w)\,[w]\,x \tag{8.11}$$

$$f\,d\,e\,[\,] = [e], \tag{8.12}$$

$$f\,d\,e\,(w;x) \;=\; \begin{cases} f\,(d+k+1)\,(e\,\&\,[w])\,x & \textbf{if } (d+k+1) \leq M, \\ e;f\,k\,[w]\,x & \text{otherwise} \end{cases} \tag{8.13}$$

$$\textbf{where } k = \textit{length } w.$$

The next task is to synthesise a more efficient alternative for *outpara* (*format x*). To do this we first derive a definition for a function *outform* which satisfies the condition.

$$\text{NL};\textit{outline } e\ \&\ \textit{outform } d\,x \;=\; \textit{outpara}\ (f\,d\,e\,x). \tag{8.14}$$

Although such a step appears unmotivated, it turns out to be just the right generalisation which gives us what we want. The base case is

$$\begin{aligned} & \textit{outpara}\ (f\,d\,e\,[\,]) \\ &= \textit{outpara}\ [e] && \text{(by (8.12))} \\ &= \text{NL};\textit{outline } e\ \&\ \textit{outpara}\ [\,] && \text{(by (8.6))} \\ &= \text{NL};\textit{outline } e\ \&\ [\,] && \text{(by (8.5))} \end{aligned}$$

from which we can derive *outform d* [] = [].

For the inductive step, suppose first that $(d+k+1) > M$, where $k = \textit{length } w$. Then

$$\begin{aligned} & \textit{outpara}\ (f\,d\,e\,(w;x)) \\ &= \textit{outpara}\ (e;f\,k\,[w]\,x) && \text{(by (8.13))} \\ &= \text{NL};\textit{outline } e\ \&\ \textit{outpara}\ (f\,k\,[w]\,x) && \text{(by (8.6))} \\ &= \text{NL};\textit{outline } e\ \&\ (\text{NL};\textit{outline}\ [w]\ \&\ \textit{outform } k\,x) && \text{(by (8.14))} \\ &= \text{NL};\textit{outline } e\ \&\ (\text{NL};w\ \&\ \textit{outform } k\,x) && \text{(by (8.7))}. \end{aligned}$$

Hence *outform d* $(w;x) = \text{NL};w\ \&\ \textit{outform } k\,x$.

In the alternative case:

$$\begin{aligned} & \textit{outpara}\ (f\,d\,e\,(w;x)) \\ &= \textit{outpara}\ (f\,(d+k+1)\,(e\,\&\,[w])\,x) && \text{(by (8.13))} \\ &= \text{NL};\textit{outline}\ (e\,\&\,[w])\ \&\ \textit{outform}\ (d+k+1)\,x && \text{(by (8.14))}. \end{aligned}$$

In order to continue, we need the result that, if $e \neq [\,]$, then

$$\textit{outline}\ (e\,\&\,[w]) \;=\; \textit{outline } e\ \&\ \text{BL};w.$$

This result will be proved below.

Since $e \neq [\,]$ in all occurrences of f, we have

$$\begin{aligned} & \textit{outpara}\ (f\,d\,e\,(w;x)) \\ &= \text{NL};(\textit{outline } e\ \&\ \text{BL};w)\ \&\ \textit{outform}\ (d+k+1)\,x \\ &= \text{NL};\textit{outline } e\ \&\ (\text{BL};\ w\ \&\ \textit{outform}\ (d+k+1)\,x) \end{aligned}$$

Hence $outform\ d\ (w;x) = \mathrm{BL};w\ \&\ outform\ (d+k+1)\ x$.

Finally we have

$$\begin{aligned} outpara\ (format\ (w;x)) &= outpara\ (f\ (length\ w)\ [w]\ x) \quad \text{(by (8.11))} \\ &= \mathrm{NL};\ w\ \&\ outform\ (length\ w)\ x \\ &= outform\ M\ (w;x) \end{aligned}$$

using the derived definition of *outform*.

It follows that the program can be written as

$$\begin{aligned} greedy\ t &= outform\ M\ (inwords\ t), \\ outform\ d\ [] &= [], \\ outform\ d\ (w;x) &= \begin{cases} \mathrm{BL};w\ \&\ outform\ (d+k+1)\ x & \textbf{if } (d+k+1) \le M, \\ \mathrm{NL};w\ \&\ outform\ k\ x & \textbf{otherwise} \end{cases} \\ & \qquad \textbf{where } k = \text{length } w \end{aligned}$$

It finally remains to prove the lemma

$$outline\ (e\ \&\ [w]) = outline\ e\ \&\ \mathrm{BL};w$$

provided $e \neq []$. There are two steps to the induction:

Case 1: $[v]$.

$$\begin{aligned} & outline\ ([v]\ \&\ [w]) \\ &= outline\ (v;[w]) && \text{(definition of \&)} \\ &= v\ \&\ \mathrm{BL};outline\ [w] && \text{(by (8.8))} \\ &= v\ \&\ \mathrm{BL};w && \text{(by (8.7))} \\ &= outline\ [v]\ \&\ \mathrm{BL};w && \text{(by (8.7)).} \end{aligned}$$

Case 2: $(v;e)$.

$$\begin{aligned} & outline\ ((v;e)\ \&\ [w]) \\ &= outline\ (v;(e\ \&\ [w])) && \text{(definition of \&)} \\ &= v\ \&\ \mathrm{BL};outline\ (e\ \&\ [w]) && \text{(by (8.8))} \\ &= v\ \&\ \mathrm{BL};(outline\ e\ \&\ \mathrm{BL};w) && \text{(induction hypothesis)} \\ &= (v\ \&\ \mathrm{BL};outline\ e)\ \&\ \mathrm{BL};w && \text{(associativity of \&)} \\ &= outline\ (v;e)\ \&\ \mathrm{BL};w && \text{(by (8.8)).} \end{aligned}$$

The last step establishes the induction and proves the lemma.

9. Summary and conclusions

The cumulative results of our endeavours are the two programs displayed in Figs. 6 and 7.

Let us now summarise the main steps by which these two programs were obtained:

Step 1. The first step was to express the specification in terms of data types appropriate to the problem.

Step 2. Although written as a functional program, the specification was found not to be executable as it required examination of every element of a set which, though finite, was generated by an infinite process. The second step was to synthesise a finite process for this set in the form of an inductive definition.

Step 3. The executable version of the specification given by Step 2 was then manipulated, using algebraic laws about operators, to derive an inductive definition of the set of all solutions which satisfied it.

Step 4. A single solution to the problem was then constructed by transforming the result of Step 3 into an inductive definition for a single member.

Step 5. This solution was then implemented efficiently by imposing a tabulation scheme on the recursive definition.

Step 6. Under a more restrictive property of *waste*, the program of Step 4 was transformed into a simple greedy algorithm.

Step 7. The greedy algorithm was then converted into a program which processed characters, and further optimised.

$$\mathit{inwords}\ [\,] = [\,],$$

$$\mathit{inwords}\ (a;t) = \begin{cases} \mathit{inwords}\ t & \textbf{if } a = \mathrm{BL} \textbf{ or } a = \mathrm{NL}, \\ \mathit{buffer}\ [a]\ t & \textbf{otherwise}, \end{cases}$$

$$\mathit{buffer}\ w\ [\,] = [w],$$

$$\mathit{buffer}\ w\ (a;t) = \begin{cases} w;\mathit{inwords}\ t & \textbf{if } a = \mathrm{BL} \textbf{ or } a = \mathrm{NL}, \\ \mathit{buffer}\ (w \,\&\, [a])\ t & \textbf{otherwise}, \end{cases}$$

$$\mathit{outform}\ d\ [\,] = [\,],$$

$$\mathit{outform}\ d\ (w;x) = \begin{cases} \mathrm{NL};w \,\&\, \mathit{outform}\ k\ x & \textbf{if } (d+k+1) > M, \\ \mathrm{BL};w \,\&\, \mathit{outform}\ (d+k+1) & \textbf{otherwise}, \end{cases}$$

$$\textbf{where}\ \ k = \mathit{length}\ \ w,$$

$$\mathit{greedy}\ t = \mathit{outform}\ M\ (\mathit{inwords}\ t).$$

Fig. 6.

$$
\begin{aligned}
&\mathit{outpara}\,[\,] = [\,],\\
&\mathit{outpara}\,(e;p) = \mathrm{NL};\ \mathit{outline}\ e\ \&\ \mathit{outpara}\ p,\\
&\mathit{outline}\,[w] = w,\\
&\mathit{outline}\,(w;e) = w\ \&\ \mathrm{BL};\mathit{outline}\ e,\\
&\mathit{format}\,[\,] = [\,]\\
&\mathit{format}\,(w;x) = \mathit{fst}\,(\mathit{hd}\,(\mathit{flist}\,(w;x))),\\
&\mathit{flist}\,[\,] = [\,],\\
&\mathit{flist}\,(w;x) = (f\,(\mathit{length}\ w)\,[w]\ x\ t);t \quad \textbf{where } t = \mathit{flist}\ x,\\
&f\,d\,e\,[\,]\,[\,] = ([e],0),\\
&f\,d\,e\,(w;x)\,(pc;t)\\
&\quad = \begin{cases} \mathit{add}\ d\ e\ pc & \textbf{if } (d+k+1) < M,\\ \mathit{pick}\,(\mathit{add}\ d\ e\ pc)(f\,(d+k+1)(e\ \&\ [w])\ x\ t) & \textbf{otherwise}\end{cases}\\
&\quad \textbf{where } k = \mathit{length}\ w\\
&\mathit{add}\ d\ e\,(p,c) = (e;p,\ \mathit{max}\,\{M-d,c\}),\\
&\mathit{pick}\,(p1,c1)(p2,c2) = \begin{cases}(p1,c1) & \textbf{if } c1 < c2,\\ (p2,c2) & \textbf{otherwise},\end{cases}\\
&\mathit{dynamic}\ t = \mathit{outpara}\,(\mathit{format}\,(\mathit{inwords}\ t))
\end{aligned}
$$

Fig. 7.

The main conclusion of the exercise is that the formal treatment of even a modest problem takes substantial time and effort. On the other hand the specification adopted for the problem was at some distance from the final algorithms. Even so, one could argue that the treatment would have been even longer if all the reasoning was directed to the discovery of imperative rather than functional programs. It is commonly believed by functional programmers that their programs are up to an order of magnitude shorter than the corresponding imperative versions. If one combines this with Dijkstra's evidence [10] that formal reasoning can be expected to take an order of magnitude more pages than the text of the final program, the advantages of a functional approach should be obvious. (There is, of course, another order of magnitude statistic: currently, functional programs are up to an order of magnitude *slower* than their procedural counterparts.)

Finally we return to the question raised in the Introduction concerning the possibilities for automating the transformational process. If there is a real difference between transformational programming and standard forms of proof, it is because the former relies on expression manipulation as the main source of inspiration. Our experience with the present and similar examples is that, for such an activity, notation is more important than machine assistance. Given an adequate notation, the computer can check reasoning, ensure type and syntax consistency, and carry out a number of routine transformations. But it can never absolve the programmer from the task of mastering the complexity of his or her program.

Acknowledgment

The author would like to thank Lambert Meertens who provided the main inspiration for the work, and John Hughes, Bernard Sufrin and Phil Wadler for many enjoyable discussions about functional and transformational programming. Thanks are also due to Tony Hoare who made numerous constructive criticisms and suggestions on a first draft. The work was supported by the Science and Engineering Research Council of Great Britain, under grant GR/C/78704.

References

[1] F.L. Bauer et al., Report on a wide spectrum language for program specification and development, Institut für Informatik, Technische Universität München, TUM-I8104, 1981.

[2] F.L. Bauer et al., Programming in a wide spectrum langauge: A collection of examples, *Sci. Comput. Programming* **1** (1981) 73-114.

[3] S. Beckett, *Molloy* (Calder & Boyars, London, 1959).

[4] R.S. Bird, Tabulation techniques for recursive programs, *ACM Comput. Surveys* **12**(4) (1980) 403-417.

[5] R.S. Bird, The promotion and accumulation strategies in transformational programming, *ACM Trans. Prog. Lang. and Systems* **6**(4) (1984) 487-504.

[6] R.S. Bird, Using circular programs to avoid multiple traversals of data, *Acta Inform.* **21** (1984) 239-250.

[7] M. Broy, Zur Spezifikation von Programmen für die Textverabeitung, in: R. Wossildo, Ed., *Textverarbeitung und Informatik*, Informatik-Fachberiche 30 (Springer, Berlin, 1980) 75-93.

[8] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *J. ACM* **24**(1) (1977) 46-67.

[9] J. Darlington, Unification of functional and logic programming, Imperial College, London, 1983.

[10] E.W.D. Dijkstra, *Selected Writings on Computing: A Personal Perspective* (Springer, Berlin, 1982).

[11] M. Feather, A system for assisting program transformation, *ACM Trans. Prog. Lang. and Systems* 4(1) (1982) 1-20.

[12] D. Gries, *The Science of Programming* (Springer, Berlin, 1981).

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

[14] L.G.L.T. Meertens, Presentation at WG2.1 meeting, Munich, May, 1983.

[15] P. Naur, Programming by action clusters, *BIT* **9** (1976) 250-258.

[16] B. Surfin, Formal system specifications: notations and examples, in: D. Neel, Ed., *Tools and Notations for Program Construction* (Cambridge University Press, London, 1982).

[17] D. Turner, Recursion equations as a programming language, in: J. Darlington, P. Henderson and D. Turner, Eds., *Functional Programming and its Applications* (Cambridge University Press, London, 1982).