Incremental algorithms on lists

Johan Jeuring[*]
CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
(jt@cwi.nl)

Abstract

Incremental computations can improve the performance of interactive programs such as spreadsheet programs, program development environments, text editors, etc. Incremental algorithms describe how to compute a required value depending on the input, after the input has been edited. By considering the possible different edit actions on the data type lists, the basic data type used in spreadsheet programs and text editors, we define incremental algorithms on lists. Some theory for the construction of incremental algorithms is developed, and we give an incremental algorithm for a more involved example: formatting a text.

CR categories and descriptors: D11 [**Software**]: Programming Techniques — Applicative Programming, D43 [**Software**]: Programming Languages — Language constructs, I22 [**Artificial Intelligence**]: Automatic Programming — Program transformation.

General terms: algorithm, design, theory.

Additional keywords and phrases: Bird-Meertens calculus for program construction, incrementality, list, text-processing.

1 Introduction

There are a number of reasons why as yet intractable problems will be solvable on a computer in the future. First, the processor speed of computers is still increasing. Second, more efficient algorithms may be derived for problems for which the existence of an optimal

[*]This research has been supported by the Dutch organisation for scientific research under project-nr. NF 62.518.

algorithm has not yet been shown. Third, for some classes of problems other kinds of algorithms, such as parallel or incremental, may be derived. This paper deals with incremental algorithms.

If a computation is performed repeatedly on slightly changed data, it is often profitable to describe the computation by means of an incremental algorithm. An incremental algorithm describes how to compute the required value depending on the slightly changed data from the old value, the changes in the data, and perhaps some other information. Examples of computations which are performed repeatedly on slightly changed data can be found in interactive programs such as program development environments and spreadsheet programs. For example, in a spreadsheet program often lists of numbers are summed. If a number of the list is changed from a to b, the sum s should be changed to $s+(b-a)$. This change does not require the summation of all numbers in the lists.

The form of an incremental algorithm depends on the data type on which it is defined, and the edit model used. We propose a method for the description and derivation of incremental algorithms in interactive programs on the data type list. Incremental algorithms on other data types will be described elsewhere. Interactive programs such as text editors and spreadsheet programs are based on the data type list, whereas program development environments are usually based on the data type tree. Let f be a function defined on lists. An incremental algorithm for f describes how to find the value of f when its argument is edited. The possible edit actions are among others deletion or insertion of elements in the argument list, splitting the argument, and joining two arguments. After each edit action the value of f is recomputed. As an example, consider the task of breaking a paragraph into lines such that the result looks nice (text-formatting). Algorithms for this problem have been given by Knuth and Plass [12], and Bird [2]. We will derive an incremental algorithm for this problem with which it is possible two combine two formatted paragraphs in constant time. Furthermore, deleting or inserting a piece of text and breaking the resulting paragraph into lines is done in time linear in the length of the deleted or inserted piece of text. A consequence of these results is that while an author edits a text, the formatted text is always available. This facilitates WYSIWYG-editing. Several other incremental algorithms on lists will be given. The algorithms are derived in the Bird-Meertens style of program construction. The Bird-Meertens style of program construction is a data type oriented calculus for algorithm derivation by means of program transformations. Specifications are transformed into efficient functional algorithms using laws for functions defined on some initial data type. Aspects of the Bird-Meertens calculus can be found in [14], [3], [4], [10], [13], and [15].

Besides lots of articles on incremental algorithms for specific problems, like for example the algorithm for incrementally computing the minimum spanning tree of Frederickson [7], the algorithm for incremental Huffman coding of Knuth [11], and the algorithm for pattern matching with a dynamically changing set of patterns of Meyer [16], several proposals for the derivation and description of incremental algorithms have been given in the literature. The language INC, designed by Yellin and Strom [25], automatically transforms algorithms in an FP-like syntax to incremental algorithms. For each construct in FP an incremental version is given, and since every algorithm consists of a series of FP constructs it can be incrementalised by incrementalising its components. This approach has the disadvantage, shared with all automatic methods for formal program derivation, that not always the most efficient incremental algorithm will be derived. Furthermore, the only data types INC can handle are *bag* and *tuple*, which is rather restrictive. Another approach to incrementality, called finite differencing, is described by Paige [18]. So-called invariants, equalities of the form $E = f(x_1, \ldots, x_n)$, are maintained by means of code which describes how to find the value of E if one or more of the arguments are changed. The approach is generic, and does not distinguish incrementality on different types. The approach sketched in this paper can be compared with the work of Reps, Teitelbaum, and Demers [20] on incremental attribute evaluators. They give an incremental algorithm in an interactive program development environment (a tree editor) for the evaluation of the attributes of a tree. A natural extension to trees of the edit model on lists presented here, would result in a slightly more complex edit model than the model Reps, Teitelbaum, and Demers give. The model obtained thus is more suitable for calculating with algorithms. Some of the incremental algorithms given in this paper have been implemented in the Views System [19].

This paper is organised as follows. Section 2 introduces the data types in the Boom-hierarchy, such as *list*, and several functions defined on these data types, such as map, filter, and catamorphism. Section 3 defines simply incremental algorithms on lists, and gives incremental algorithms for most of the functions introduced in Section 2. The definition of incremental algorithms is then generalised in order to be able to give incremental algorithms for a class of problems for which we could not derive incremental algorithms using the old definition. In fact, using this new definition we can give a (usually inefficient) incremental algorithm for every catamorphism on lists. We give an efficient incremental algorithm for text-formatting. Section 4 describes future work: possible ways in which the theory of incremental algorithms on lists reported on in this paper can be extended to other data types.

2 Preliminaries

This section introduces the basic notions and definitions used in the subsequent section. The first subsection briefly describes the notational conventions for functions and operators. Two important concepts in the Bird-Meertens calculus are the notions of catamorphism and promotion. Catamorphisms are functions, defined on an initial data type, whose inductive definitional pattern mimics that of the type. For every data type, catamorphisms can be defined and a promotion theorem can be given. This process is described in detail by Malcolm [13]. The second subsection introduces the Boom-hierarchy, a hierarchy consisting of the data types set, bag, list, and binary tree, and it defines catamorphisms on these data types. We also give some widely used examples of catamorphisms, such as map and reduction. Finally, some auxiliary functions are introduced in the third subsection.

2.1 Functions and operators

Typical names of functions are f, g and h. Function composition, which is associative, is denoted by a small dot $\cdot$. So the composition of f and g is written as $f \cdot g$. Function application is denoted by white space. So the application of f to an argument a is written as $f\ a$. Function application associates to the right, i.e., we have

$$(f \cdot g \cdot h)\, x = f\,(g\,(h\,x)) = f\ g\ h\ x\ .$$

For every two types A and B there exists the product type $A \times B$ consisting of pairs of elements from A and B. The operator $\times$ is defined on types as well as functions, i.e., given two functions $f : A \to B$ and $g : C \to D$, we have a function $f \times g : A \times C \to B \times D$. The projection functions from $A \times B$ to A and B are denoted by $\ll$, the left-projection, and $\gg$, the right-projection.

Binary operators will often be written in infix notation. Typical names of binary operators are $\oplus$, $\odot$, and $\otimes$. They can be partially parametrised, i.e., if $\oplus$ is a binary operator of type $A \times B \to C$, we consider the expression $(a\oplus)$ to be a unary function of type $B \to C$, and similarly for $(\oplus b)$. These parametrised operators are also known as 'sections'. Binary operators take their left arguments as short as possible and their right arguments as long as possible, and functions take their arguments as long as possible; so, for example,

$$f\ a \oplus g\ b \otimes c \quad = \quad f\,(a \oplus (g\,(b \otimes c)))\ .$$

The notation $x \leq_f y$ expresses that $(f\ x) \leq (f\ y)$, and similarly for $=$, $>$, etc.

2.2 Data types and catamorphisms

The 'Theory of Lists' described in this section has been introduced by Bird [3] and Meertens [14]. The recursive data type *list* over some base type A, denoted by $A*$, is introduced by means of the following three constructor rules:

$$\frac{}{1_{+\!\!+} \in A*} \qquad \frac{a \in A}{[a] \in A*} \qquad \frac{\begin{array}{c} x \in A* \\ y \in A* \end{array}}{x +\!\!+ y \in A*}$$

where $1_{+\!\!+}$ is the unit of $+\!\!+$, and $+\!\!+$ is associative. This data type is also called *join-list*, because the operator $+\!\!+$ 'joins' two lists together.

The data type list is one of the four data types in the Boom-hierarchy. The Boom-hierarchy, described by Meertens [14], consists of four data types: binary tree, list, bag, and set. These data types are obtained from the above scheme for $A*$ by varying the laws satisfied by $+\!\!+$. If $+\!\!+$ satisfies no laws, then the above scheme leads to the data type *binary tree* with information at the leaves. If $+\!\!+$ is associative we obtain list, and if $+\!\!+$ is associative and commutative we obtain *bag* (then $+\!\!+$, $[\cdot]$, $1_{+\!\!+}$, and $A*$ are written as respectively $\uplus$, $\lfloor\cdot\rfloor$, $1_{\uplus}$, and $A\varpi$). If $+\!\!+$ is also idempotent we obtain the data type *set*.

Let $\oplus : A \times A \to A$ be an associative operator, $f : B \to A$ a function, and $e \in A$ the unit of $\oplus$. Then, by definition of the data type list, there exists a unique function $h : B* \to A$ such that

$$(1) \quad \begin{array}{lcl} h\, 1_{+\!\!+} & = & e \\ h\, [b] & = & f\, b \\ h\, (x +\!\!+\, y) & = & (h\, x) \oplus (h\, y)\ . \end{array}$$

If such a unit element does not exist, we may introduce a fictitious element (see Meertens [14]) with the property that it is the unit of $\oplus$. The function h defined above is called a *catamorphism*.

It is a well-known fact that a catamorphism on list can be written as the composition of a reduction and a map, which are defined as follows. The *map operator* $*$ takes as arguments a function and a list and returns a list consisting of the original elements to which the function is applied. More precisely, if $f : B \to A$, then $f* : B* \to A*$ is defined by

$$(2) \quad \begin{array}{lcl} f{*}\, 1_{+\!\!+} & = & 1_{+\!\!+} \\ f{*}\, [a] & = & [f\, a] \\ f{*}\, (x +\!\!+\, y) & = & (f{*}\, x) +\!\!+\, (f{*}\, y)\ . \end{array}$$

The value of applying the *reduction operator* $/$ to an associative operator $\oplus$ and a list can be obtained by placing $\oplus$ between adjacent elements of the list, so, if $\oplus : A \times A \to A$, then $\oplus/ : A* \to A$ is defined by

$$(3) \quad \begin{array}{lcl} \oplus/\, 1_{+\!\!+} & = & 1_{\oplus} \\ \oplus/\, [a] & = & a \\ \oplus/\, (x +\!\!+\, y) & = & (\oplus/\, x) \oplus (\oplus/\, y)\ . \end{array}$$

where $1_{\oplus}$ is the, possibly fictitious, unit element of $\oplus$. A catamorphism h can be split in a reduction and a map, that is, there exist an operator $\oplus$ and a function f such that

$$(4) \quad h \;=\; \oplus/ \cdot f*\ ,$$

a fact expressed by the Homomorphism Lemma from Meertens [14]. An example of a widely used catamorphism is the *filter operator* $\triangleleft$, which takes a predicate (i.e. a boolean function) and a list and retains the elements satisfying the predicate in a list; so if $p : A \to \mathit{bool}$, then $p\triangleleft : A* \to A*$ is defined by

$$(5) \quad p\triangleleft \;=\; +\!\!+/ \cdot p?*\ ,$$

where $p?\, a = [a]$ if $p\, a$ holds and $p?\, a = 1_{+\!\!+}$ otherwise. For example, $\mathit{odd}\triangleleft[3,4,5] = [3,5]$.

Another important notion of the Bird-Meertens formalism is *promotion*. Every data type has its own promotion theorem. Promotion provides a means for proving equalities of functions avoiding the application of induction in the development of algorithms. Inductive arguments tend to be tedious and are less elegant than proofs using promotion. As early as 1975 this was one of the main motivations of Goguen [8] to use initiality in proofs. Before we give the theorem, we first define *promotability*.

(6) Definition (($\oplus, \otimes$)–promotability) *A function $f : A \rightarrow B$ is $(\oplus, \otimes)$–promotable for associative operators $\oplus : A \times A \rightarrow A$ and $\otimes : B \times B \rightarrow B$ if and only if*

$$\begin{array}{lcl} f\,(x \oplus y) & = & (f\ x) \otimes (f\ y) \\ f\,1_{\oplus} & = & 1_{\otimes}\ . \end{array}$$

For example, function $f*$ is $(+\!\!+, +\!\!+)$–promotable for all functions f, and function $\oplus/$ is $(+\!\!+, \oplus)$–promotable for all operators $\oplus$. The proof of the following theorem (by structural induction or using the uniqueness property of catamorphisms) is given by Meertens [14] and Malcolm [13].

(7) Theorem (Promotion) *A function $f : A \rightarrow B$ is $(\oplus, \otimes)$–promotable if and only if*

$$f \cdot \oplus/ \cdot g* = \otimes/ \cdot (f \cdot g)*\ .$$

From the proof of this theorem it follows that we may weaken the requirement that f is $(\oplus, \otimes)$–promotable, in particular the equality $f\,(x \oplus y) = (f\ x) \otimes (f\ y)$. It suffices to require this equality for x and y in the range of $g*$. An immediate consequence of the Promotion Theorem is the *map distributivity* law:

(8) $\quad f* \cdot g* \ = \ (f \cdot g)*\ .$

Besides catamorphisms we will also use *left-reductions* and *right-reductions* defined on join-list. (Left-reductions can be viewed as catamorphisms on the data type snoc-list, lists which are constructed from left to right.) Given an element $e : B$ and an operator $\oplus : B \times A \rightarrow B$ there exists a unique function $\oplus\!\not\rightarrow\! e : A* \rightarrow B$ satisfying the following two equalities.

(9) $$\begin{array}{lcl} (\oplus\!\not\rightarrow\! e)\,1_{+\!\!+} & = & e \\ (\oplus\!\not\rightarrow\! e)\,(x +\!\!+ [a]) & = & ((\oplus\!\not\rightarrow\! e)\,x) \oplus a\ . \end{array}$$

Such a function is called a left-reduction. Right-reductions are defined similarly. Given an element $e : B$ and an operator $\oplus : A \times B \rightarrow B$ there exists a unique function $\oplus\!\not\leftarrow\! e : A* \rightarrow B$ satisfying the following two equalities.

(10) $$\begin{array}{lcl} (\oplus\!\not\leftarrow\! e)\,1_{+\!\!+} & = & e \\ (\oplus\!\not\leftarrow\! e)\,([a] +\!\!+ x) & = & a \oplus ((\oplus\!\not\leftarrow\! e)\,x)\ . \end{array}$$

Every catamorphism can be written as a left-reduction and as a right-reduction, but not vice versa.

2.3 Auxiliary functions and operators

In this section we define some auxiliary functions and operators. The operator cross is used frequently in calculations. Cross takes two lists, and pairs each element of the first list with each element of the second list. The result of cross is a bag of these pairs. For example, $[1,2] \between [3,4,5] = \lfloor (1,3),(1,4),(1,5),(2,3),(2,4),(2,5) \rfloor$. The operator cross has been introduced by Bird [4]. The following definition is taken from Jeuring [9].

$$(11)\quad \begin{array}{lcl} x \between y & = & \uplus/\,(\vdash x)\!*\, y \\ a \vdash x & = & (;a)\!*\, x \\ b\,;a & = & (b,a)\ . \end{array}$$

Operator cross can be subscripted with a binary operator, by which we mean the following.

$$(12)\quad \between_{\oplus} \;=\; \oplus\!* \cdot \between\ .$$

Note that $\oplus$ is a binary prefix operator.

Functions it, tl, hd, and lt have their usual meaning and are defined by

$$\begin{array}{lrcl} (13) & it\,(x +\!\!+ [a]) & = & x \\ (14) & lt\,(x +\!\!+ [a]) & = & a \\ (15) & hd\,([a] +\!\!+ x) & = & a \\ (16) & tl\,([a] +\!\!+ x) & = & x\ . \end{array}$$

Functions tl and it commute with $f*$, that is, for all f

$$\begin{array}{lrcl} (17) & f\!*\cdot it & = & it \cdot f\!* \\ (18) & f\!*\cdot tl & = & tl \cdot f\!*\ . \end{array}$$

Functions hd and lt satisfy for all f

$$\begin{array}{lrcl} (19) & f \cdot hd & = & hd \cdot f\!* \\ (20) & f \cdot lt & = & lt \cdot f\!*\ . \end{array}$$

Using functions hd and tl an indexing operator on lists, denoted by !, can be defined. This operator takes a natural number n and a list of length at least n. It is defined by

$$(21)\quad \begin{array}{lcl} 0!x & = & hd\,x \\ (n+1)!x & = & n!(tl\,x)\ . \end{array}$$

The operator take, $\rightharpoonup$, takes, given a natural number n and a list x, the first n elements of x if $\#\,x \geq n$, and it takes x itself if $\#\,x < n$. For x such that $\#\,x \geq n$ it is defined as follows.

$$(22)\quad \begin{array}{lcl} 0 \rightharpoonup x & = & 1_{+\!\!+} \\ (n+1) \rightharpoonup x & = & [hd\,x] +\!\!+ (n \rightharpoonup tl\,x)\ . \end{array}$$

In the specification of algorithms one often encounters generators, such as *segs* (returning all consecutive substrings of a string), and *parts* (returning all partitions of a string). Using list comprehension, see Turner [24], we define

$$
\begin{array}{lrcl}
(23) & \mathit{segs}\,x & = & [v \mid \exists u, w : x = u +\!\!+\, v +\!\!+\, w] \\
(24) & \mathit{tails}\,x & = & [v \mid \exists w : x = w +\!\!+\, v] \\
(25) & \mathit{inits}\,x & = & [v \mid \exists w : x = v +\!\!+\, w]\ .
\end{array}
$$

Following Bird [2], we characterise *segs* recursively by means of the functions *tails* and *inits* as follows.

$$
\begin{array}{lrcl}
(26) & \mathit{segs} & = & +\!\!+/ \cdot \mathit{tails}{*} \cdot \mathit{inits} \\
(27) & \mathit{tails} & = & \bigtriangledown/ \cdot [[\cdot]]{*} \\
(28) & \mathit{inits} & = & \bigtriangleup/ \cdot [[\cdot]]{*}\ ,
\end{array}
$$

where the function $[[\cdot]]$ is defined by $[[\cdot]]\ a = [[a]]$, and the operators $\bigtriangledown$ and $\bigtriangleup$ are defined by

$$
\begin{array}{lrcl}
(29) & x \bigtriangledown y & = & ((+\!\!+(\mathit{hd}\,y)){*}\,x) +\!\!+\, y \\
(30) & x \bigtriangleup y & = & x +\!\!+\, (((\mathit{lt}\,x)+\!\!+){*}\,y)
\end{array}
$$

Function *parts* enumerates in a bag all ways in which a list can be broken into lists of lists. It is defined by means of bag comprehension as follows.

$$
\begin{array}{lrcl}
(31) & \mathit{parts}\,x & = & \lfloor y \mid x = +\!\!+/\ y \rfloor\ .
\end{array}
$$

There are various recursive characterisations of the function *parts*. A characterisation of *parts* as a left-reduction is given by Bird [2]. Here, a characterisation as a join-list catamorphism of *parts* is given.

$$
(32)\quad
\begin{array}{lcl}
\mathit{parts} & : & A{*} \to A{*}{*}\varpi \\
\mathit{parts}\,1_{+\!\!+} & = & \lfloor 1_{+\!\!+} \rfloor \\
\mathit{parts}\,[a] & = & \lfloor [[a]] \rfloor \\
\mathit{parts}\,(x +\!\!+\, y) & = & (\mathit{parts}\,x)\ \Phi\ (\mathit{parts}\,y)\ ,
\end{array}
$$

where

$$
\begin{array}{lrcl}
(33) & x\ \Phi\ y & = & \uplus/ x \between_{\diamondsuit} y \\
(34) & x \diamondsuit 1_{+\!\!+} & = & \lfloor x \rfloor \\
(35) & 1_{+\!\!+} \diamondsuit x & = & \lfloor x \rfloor \\
(36) & x \diamondsuit y & = & \lfloor \mathit{xi} +\!\!+\, [\mathit{xl}, \mathit{yh}] +\!\!+\, \mathit{yt}, \mathit{xi} +\!\!+\, [\mathit{xl} +\!\!+\, \mathit{yh}] +\!\!+\, \mathit{yt} \rfloor \\
 & & & \text{where} \\
 & & & \mathit{xi} = \mathit{it}\,x,\ \mathit{xl} = \mathit{lt}\,x,\ \mathit{yh} = \mathit{hd}\,y,\ \mathit{yt} = \mathit{tl}\,y\ .
\end{array}
$$

In order to show that function $\mathit{parts} = \Phi/ \cdot (\lfloor\cdot\rfloor \cdot [[\cdot]]){*}$ is well defined, it has to be shown that operator Φ is associative. The proof of this fact is omitted.

Let $f : A \to B$, where B is totally ordered. Then operator $\uparrow_f$ is a binary operator of type $A \times A \to A$. It is defined by

$$(37) \quad x \uparrow_f y \;=\; \begin{array}{ll} x & \text{if } x >_f y \\ y & \text{if } y >_f x \\ x \text{ or } y & \text{otherwise} \end{array} \; .$$

We do not yet define $\uparrow_f$ on arguments which have equal f-values, except that one of the arguments is the outcome. It might be necessary to define $\uparrow_f$ differently for different problems. If the choice made by operator $\uparrow_f$ on equal f-values is immaterial to the problem, we will not give its exact definition. Operator $\downarrow_f$ is defined similarly.

3 Incremental algorithms on lists

Incremental algorithms on lists can be used in interactive programs such as text editors, spreadsheet programs, etc. Suppose we want to code a text with respect to a dictionary. Usually, the text is coded after it has been edited. By means of an incremental algorithm the text can be coded while it is edited. Consequently, the coded text is available at each moment. In this section we sketch an approach to incrementality on lists. We give a definition of basic incremental algorithms in the first subsection, and we give several examples of problems for which incremental algorithms can be given. Then we give a definition of more general incremental algorithms, and, using this new definition, we show that there exists an incremental algorithm for every catamorphism on lists. The second subsection contains a more involved example: the derivation of an efficient incremental algorithm for formatting a text.

3.1 Two definitions of incremental algorithms

Let f be a function defined on lists: $f : A* \to B$. Suppose that we want to find the f-value of a list, and that we are interactively editing this list. A description of interactive programs in a functional setting has been given by Thompson [23]. When editing a piece of data, a text, a program, or a list of numbers from a spreadsheet program, a cursor is moved through the data. Suppose the data is represented as a list. The cursor is always positioned in between two elements. If the cursor is positioned somewhere in the data, two lists can be distinguished: the part of the data in front of the position of the cursor, and the part after the position of the cursor. Several actions are possible.

- moving the cursor right or left;
- deleting or inserting one or more elements;
- splitting the data in two;
- concatenating two pieces of data.

This list of edit actions is incomplete, but it does comprise the basic components of an editor. Most of the other components of editors consist of compositions of these actions.

After each action, we want the result of f applied to the resulting list(s) to be immediately available. This implies that we have to adapt the interactive program we are working in. After an edit action, the interactive program should also, besides for example showing the result of the edit action on the screen, update the f-value(s). We now describe what should happen after each of the edit actions. This determines the form of an incremental algorithm.

When two pieces of data, say x and y, are concatenated, the value of $f\,(x +\!\!+\, y)$ has to be determined from the values $f\,x$ and $f\,y$. The first, tentative, assumption we make about incremental algorithms is that there exists an operator $\odot$ such that $f\,(x +\!\!+\, y) = (f\,x) \odot (f\,y)$. It follows that f is a join-list catamorphism. This assumption is almost inevitable if we want to deal with insertion and deletion properly, but it is also reasonable. Many functions, possibly tupled with some extra information, are catamorphisms that can be implemented efficiently. If the data is split into two pieces of data, say again x and y, the values of $f\,x$ and $f\,y$ have to be determined from the value $f\,(x +\!\!+\, y) = (f\,x) \odot (f\,y)$. If operator $\odot$ is invertible this is easy; however, most binary associative operators are not invertible. In general, there is no other way to find the values of $f\,x$ and $f\,y$ than to compute them from scratch or to tuple the computation with the computation of the f-value of the list in front of the cursor $(f\,x)$, and the f-value of the list after the cursor $(f\,y)$. We have chosen this last option. Splitting the data into two at the point where the cursor is located is now simple: the f-values of the constituents are immediately available. Concluding, we have assumed that the interactive program is extended with the computation of a triple of values: the f-value of the list in front of the cursor, the f-value of the argument list, and the f-value of the list after the cursor.

The form of incremental algorithms described above provides an elegant way to deal with insertion and deletion of one or more elements. Suppose a list z is inserted in between the two lists x and y, so the triple $(f\,x\,, f\,(x +\!\!+\, y)\,, f\,y)$ should be transformed into $(f\,(x +\!\!+\, z)\,, f\,(x +\!\!+\, z +\!\!+\, y)\,, f\,y)$. To obtain this triple: split $x +\!\!+\, y$ and compute $f\,z$, and then compute $(f\,x) \oplus (f\,z)$ and $(f\,x) \oplus (f\,z) \oplus (f\,y)$. If a segment z is deleted from $x +\!\!+\, z +\!\!+\, y$: first split $x +\!\!+\, z +\!\!+\, y$ into x and $z +\!\!+\, y$, and then split $z +\!\!+\, y$ in z and y. Since the values of $f\,x$ and $f\,y$ are now available, the triple $(f\,x\,, f\,(x +\!\!+\, y)\,, f\,y)$ can be computed.

Finally, we have to deal with cursor movements. Suppose the cursor is positioned in between two nonempty lists, say lists $x +\!\!+\, [a]$ and $[b] +\!\!+\, y$, and the cursor is moved left. Then it is required to find the values $f\,x$ and $f\,([a,b] +\!\!+\, y)$ from the values $f\,(x +\!\!+\, [a])$, $f\,([b] +\!\!+\, y)$, and $f\,(x +\!\!+\, [a,b] +\!\!+\, y)$. Since we assume that f is a catamorphism $\odot/\cdot r*$, we have $f\,([a,b] +\!\!+\, y) = (r\,a) \odot f\,([b] +\!\!+\, y)$. Furthermore, we also have $f\,(x +\!\!+\, [a]) = (f\,x) \odot (r\,a)$. So, if there exists an operator $\otimes$ such that $((f\,x) \odot (r\,a)) \otimes a = f\,x$, then we can express $f\,x$ in terms of $f\,(x +\!\!+\, [a])$ by means of $f\,x = (f\,(x +\!\!+\, [a])) \otimes a$. For incremental algorithms we require the existence of such an operator $\otimes$. When the cursor is moved right it is required to find the values $f\,(x +\!\!+\, [a,b])$ and $f\,y$ from the values $f\,(x +\!\!+\, [a])$, $f\,([b] +\!\!+\, y)$, and $f\,(x +\!\!+\, [a,b] +\!\!+\, y)$. For incremental algorithms we require the existence of an operator $\oplus$ satisfying $a \oplus ((r\,a) \odot (f\,x)) = f\,x$.

(38) Definition (Basic incremental algorithm) *A basic incremental algorithm for f is a 3-tuple*

$$(\odot/ \cdot r* \,,\, \otimes \,,\, \oplus)\ ,$$

such that

$$\begin{array}{lcl} f & = & \odot/ \cdot r* \\ ((f\,x) \odot (r\,a)) \otimes a & = & f\,x \\ a \oplus ((r\,a) \odot (f\,x)) & = & f\,x\ . \end{array}$$

We say a function is *incremental* if there exists a basic incremental algorithm for it. We give some examples of incremental functions.

- For all functions f, function $f*$ is incremental. The basic incremental algorithm for $f*$ is the 3-tuple

  $$(+\!\!+/ \cdot ([\cdot] \cdot f)* \,,\, it \cdot \ll \,,\, tl \cdot \gg)\ .$$

- For all predicates p, function $p\triangleleft$ is incremental. The basic incremental algorithm for $p\triangleleft$ is the 3-tuple

  $$(+\!\!+/ \cdot p?* \,,\, \otimes \,,\, \oplus)\ ,$$

  where operators $\otimes$ and $\oplus$ are defined by

  $$(39)\quad x \otimes a \;=\; \begin{array}{ll} it\;x & \text{if } p\;a \\ x & \text{otherwise} \end{array}$$

  $$(40)\quad a \oplus x \;=\; \begin{array}{ll} tl\;x & \text{if } p\;a \\ x & \text{otherwise}\ . \end{array}$$

- Function $\oplus/$ is incremental, provided sections $(\oplus a)$ and $(a\oplus)$ are invertible. An example of an incremental reduction is $+/$. The basic incremental algorithm for $+/$ is $(+/\,,-\,,-)$.

- Function *parts*, see (32), is incremental. Let function g be defined by

  $$(41)\quad \begin{array}{lcl} g\,(x +\!\!+ [y +\!\!+ [a]]) & = & x +\!\!+ [y]\ ,\ \text{if } y \neq 1_{+\!\!+} \\ g\,(x +\!\!+ [[a]]) & = & x\ . \end{array}$$

  The definition of $g\,1_{+\!\!+}$ is irrelevant. Then, if *remdups* is a function which removes all duplicates,

  $$parts\;x \;=\; remdups\;g*\;parts\,(x +\!\!+ [a])\ .$$

  Similarly, if h is defined by

$$(42)\quad \begin{array}{lcl} h\,([[a] +\!\!+\, y] +\!\!+\, x) & = & [y] +\!\!+\, x\ ,\ \text{if } y \neq 1_{+\!+} \\ h\,([[a]] +\!\!+\, x) & = & x\ , \end{array}$$

then

$$\mathit{parts}\,x \;=\; \mathit{remdups}\,h{*}\,\mathit{parts}\,([a] +\!\!+\, x)\ .$$

It follows that $(\Phi/\cdot(\lfloor\cdot\rfloor\cdot[[\cdot]]){*}, \mathit{remdups}\cdot g{*}\cdot\ll, \mathit{remdups}\cdot h{*}\cdot\gg)$ is a basic incremental algorithm for *parts*.

Let $(\odot/\cdot r{*}\,,\otimes\,,\oplus)$ be a basic incremental algorithm for f, and let g be a function. We want to find conditions on g such that the existence of a basic incremental algorithm for $g\cdot f$ is guaranteed.

(43) Theorem *Let $(\odot/\cdot r{*}\,,\otimes\,,\oplus)$ be a basic incremental algorithm for f, and let g be a $(\odot,\Box)$–promotable function satisfying*

$$\begin{array}{lcl} g\,(x\otimes a) & = & (g\,x)\oslash a \\ g\,(a\oplus x) & = & a\ominus(g\,x)\ , \end{array}$$

for some operators $\oslash$ and $\ominus$. Then $(\Box/\cdot(g\cdot r){*}\,,\oslash\,,\ominus)$ is a basic incremental algorithm for $g\cdot f$.

Proof Since g is $(\odot,\Box)$–promotable it follows using Promotion, Theorem 7, that $g\cdot f = \Box/\cdot(g\cdot r){*}$. Furthermore, we have by calculation

$$\begin{array}{ll} & ((g\,f\,x)\,\Box\,(g\,r\,a))\oslash a \\ = & \quad g \text{ is } (\odot,\Box)\text{–promotable} \\ & (g\,((f\,x)\odot(r\,a)))\oslash a \\ = & \quad (g\,x)\oslash a = g\,(x\otimes a) \\ & g\,(((f\,x)\odot(r\,a))\otimes a) \\ = & \quad (\odot/\cdot r{*}\,,\otimes\,,\oplus) \text{ is a basic incremental algorithm for } f \\ & g\,f\,x\ . \end{array}$$

Similarly, $a\ominus((g\,r\,a)\,\Box\,(g\,f\,x)) = g\,f\,x$. It follows that $(\Box/\cdot(g\cdot r){*}\,,\oslash\,,\ominus)$ is a basic incremental algorithm for $g\cdot f$. □

By the theorem we obtain a basic incremental algorithm for the problem of finding the shortest partition into ascending lists of a string. The problem is specified by

$$(44)\quad \mathit{sap} \;=\; \downarrow_{\#}/\cdot(\mathit{all\ ascending})\triangleleft\cdot\mathit{parts}\ .$$

Given a predicate p, the predicate *all* p is defined by

$$(45)\quad \mathit{all}\,p \;\equiv\; \wedge/\cdot p{*}\ .$$

For example, if we apply function *sap* to the list $[1,3,2,1,4]$ we obtain the partition $[[1,3],[2],[1,4]]$. Since *parts* is a basic incremental algorithm, we try to apply Theorem 43. The derivations of definitions of operators $\Box$, $\oslash$, and $\ominus$ are omitted. Let $\Box$, $\oslash$, and $\ominus$ be defined by

$$\begin{array}{lcll} x \mathbin{\square} y & = & \begin{array}{l} x +\!\!+\, y \\ (\mathit{it}\; x) +\!\!+ [(\mathit{lt}\; x) +\!\!+ (\mathit{hd}\; y)] +\!\!+ (\mathit{tl}\; y) \end{array} & \begin{array}{l} \text{if } \mathit{lt}\;\mathit{lt}\; x > \mathit{hd}\;\mathit{hd}\; y \\ \text{otherwise} \end{array} \\ x \oslash a & = & g\, x & \\ a \ominus x & = & h\, x\ . & \end{array}$$

Then, since $\downarrow_{\#}/ \cdot (\mathit{all\ ascending}) \triangleleft \cdot \lfloor\cdot\rfloor \cdot [[\cdot]] = [[\cdot]]$, we have that $(\square/ \cdot [[\cdot]]*\, , \oslash , \ominus)$ is a basic incremental algorithm for $\downarrow_{\#}/ \cdot (\mathit{all\ ascending}) \triangleleft \cdot \mathit{parts}$.

An example of an algorithm which is not incremental is the reduction $\uparrow/$. For $\uparrow/$ there do not exist operators $\oplus$ and $\otimes$ satisfying respectively $a \oplus (a \uparrow (\uparrow/\, x)) = \uparrow/\, x$ and $((\uparrow/\, x) \uparrow a) \otimes a = \uparrow/\, x$. Another problem for which no basic incremental algorithm exists is the maximum segment sum problem. Since we do want to have incremental algorithms for these problems we generalise the definition of incremental algorithms.

Given a function f which is required to be incremental, the interactive program in which f is computed is extended with the computation of a 3-tuple $(f\, x\, , f\,(x +\!\!+\, y)\, , f\, y)$, where x (y) is the list in front of (after) the cursor. Instead of the 3-tuple $(f\, x , f\,(x +\!\!+\, y), f\, y)$ we extend the interactive program with the computation of the 3-tuple $(g\, x\, , f\,(x +\!\!+\, y)\, , h\, y)$, and we suppose there exist (efficiently computable) functions α and β such that $f = \alpha \cdot g$ and $f = \beta \cdot h$. Furthermore, to deal with cursor movements, g is a left-reduction $\otimes\!\not\to\! e$ such that there exists an operator $\oslash$ satisfying $(x \otimes a) \oslash a = x$, and h is a right-reduction $\oplus\!\not\leftarrow\! u$ such that there exists an operator $\ominus$ satisfying $a \ominus (a \oplus x) = x$. Note that g, or the three-tuple $(\otimes\!\not\to\! e\, , \oslash\, , \alpha)$, and h, or the three-tuple $(\oplus\!\not\leftarrow\! u\, , \ominus\, , \beta)$, play a dual role.

(46) Definition (Incremental algorithm) *An incremental algorithm for f is a 7-tuple*

$$(\odot/ \cdot r*\, , \otimes\!\not\to\! e\, , \oslash\, , \alpha\, , \oplus\!\not\leftarrow\! u\, , \ominus\, , \beta)\ ,$$

such that

$$\begin{array}{lcl} f & = & \odot/ \cdot r* \\ f & = & \alpha \cdot \otimes\!\not\to\! e \\ f & = & \beta \cdot \oplus\!\not\leftarrow\! u \\ (((\otimes\!\not\to\! e)\, x) \otimes a) \oslash a & = & (\otimes\!\not\to\! e)\, x \\ a \ominus (a \oplus ((\oplus\!\not\leftarrow\! u)\, x)) & = & (\oplus\!\not\leftarrow\! u)\, x\ . \end{array}$$

A basic incremental algorithm $(\odot/ \cdot r*\, , \oslash\, , \ominus)$ for a function f can be extended to an incremental algorithm for f by taking

$$(\odot/ \cdot r*\, , \otimes\!\not\to\! e\, , \oslash\, , \mathit{id}\, , \oplus\!\not\leftarrow\! u\, , \ominus\, , \mathit{id})\ ,$$

where operators $\otimes$ and $\oplus$ are defined by

$$\begin{array}{lcl} x \otimes a & = & x \odot (r\, a) \\ a \oplus x & = & (r\, a) \odot x\ . \end{array}$$

In fact, for every catamorphism there exists an incremental algorithm. This is expressed by the following theorem.

(47) Theorem *Let f be a catamorphism. Then*

$$(f \, , f{*} \cdot \mathit{inits} \, , \mathit{it} \cdot \ll \, , \mathit{lt} \, , f{*} \cdot \mathit{tails} \, , \mathit{tl} \cdot \gg \, , \mathit{hd}) \, ,$$

is an incremental algorithm for f.

Proof The proof consists of showing that $f{*} \cdot \mathit{inits}$ is a left-reduction $\otimes\!\not\rightarrow\!e$ such that $\mathit{it}\,(x \otimes a) = x$ and showing that $f{*} \cdot \mathit{tails}$ is a right-reduction $\oplus\!\not\leftarrow\!u$ such that $\mathit{tl}\,(a \oplus x) = x$. This is easy and omitted. Furthermore, we have to show that $\mathit{lt} \cdot f{*} \cdot \mathit{inits} = f$, and that $\mathit{hd} \cdot f{*} \cdot \mathit{tails} = f$. Since $\mathit{lt} \cdot f{*} = f \cdot \mathit{lt}$, and $\mathit{lt} \cdot \mathit{inits} = \mathit{id}$, and similarly for lt and inits replaced by respectively hd and tails, these equalities follow immediately. □

The maximum segment sum problem is specified by

(48) $\uparrow/ \cdot +/{*} \cdot \mathit{segs}$.

A slight generalisation of this problem (tuple with the maximum sum among the *tails*, the maximum sum among the *inits*, and the sum of the argument list) is a catamorphism, see Smith [22]. Hence Theorem 47 gives an efficient incremental algorithm for finding the maximum segment sum of a list.

3.2 Formatting a text incrementally

The problem considered here is the derivation in the Bird-Meertens style of program construction of an efficient incremental algorithms for breaking a paragraph into lines (text-formatting). The derivation will not be given in full detail, we merely give a brief overview.

When formatting a document, one of the tasks is to break each paragraph into individual lines such that the result looks nice. There are many aspects to this task. In the detailed study on the breaking of paragraphs into lines by Knuth and Plass [12], many of these aspects are treated.

One of the aspects of text-formatting is the formalisation of 'nicely looking'. Knuth and Plass [12] describe several functions, which, given a formatted text, determine the 'badness' of that solution. These functions are called waste functions. Given a waste function, it is then required to find a solution which minimises the amount of waste. Algorithms for this problem have been given by Knuth and Plass [12], Achugbue [1], and Bird [2]. These algorithms are on line: suppose f is the problem to be solved, x is the text processed until now (so $f\ x$ is available), and a new word a is added to the right end of x, then there exists an operator $\oplus$, which requires constant time on the average for its evaluation, such that $f\,(x \mathbin{+\!\!+} [a]) = (f\ x) \oplus a$.

In this paper we derive in the Bird-Meertens calculus an incremental algorithm that solves the paragraph problem. This algorithm combines two paragraphs in constant time, that is, there exists an operator $\otimes$ such that $f\,(x \mathbin{+\!\!+} y) = (f\ x) \otimes (f\ y)$, and taking constant time for its evaluation. Of course, the time to combine two paragraphs is bounded below by the time to write a screen. The solution of Knuth and Plass to the paragraph problem will give different results in some cases. Combining two paragraphs using their solution requires time linear in the length of the second paragraph. Since $\otimes$ can be evaluated in constant time, the algorithm we obtain for formatting a paragraph is linear time, so the algorithm we obtain has the same time complexity as the on-line algorithms. Furthermore, deleting or inserting a piece of text and breaking the resulting paragraph into lines can be done in time linear in the length of the deleted or inserted piece of text. A consequence of these results is that a text can be formatted while it is edited. This facilitates WYSIWYG-editing.

We give a formal specification of the problem of breaking a paragraph into lines. Suppose a list of natural numbers is given, representing the lengths of the words in the text. It is assumed that punctuation marks are glued to the words on which they follow. This list of numbers has to be broken into lists that all fit on a given line length such that a given waste function is minimised. We suppose the line length is given by a constant C, and that all natural numbers in the input are at most this value C. For example, the following sentence from 'Pride and Prejudice' by Jane Austen

> It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

is represented by the list

$$[2,2,1,5,11,13,4,1,6,3,2,10,2,1,4,8,4,2,2,4,2,1,5] \ .$$

If $C = 25$, one of the many formattings is the following.

(49)
It is a truth universally
acknowledged, that a single
man in possession of a good
fortune, must be in want of a
wife.

We specify the problem to be solved, f, by

$$\begin{array}{lcl} f & : & A* \rightarrow A** \\ f & = & \downarrow_w / \cdot (\mathit{all\,fit}) \triangleleft \cdot \mathit{parts} \ , \end{array} \qquad (50)$$

where $\downarrow_w$ and *fit* are defined as follows.

The predicate *fit* is defined by

$$\mathit{fit}\ x \quad = \quad (+/\ x) \leq C \ , \qquad (51)$$

for some given constant C.

For the reduction $\downarrow_w/$, function w has to be defined, and the definition of $\downarrow_w$ on objects with equal w-value has to be given. For function w various choices can be made. We choose one of the simplest, such that, for the on-line case, f can be implemented by a greedy algorithm. Other, more sophisticated, choices of w are given by Bird [2] and Knuth and Plass [12]. I don't know whether incremental algorithms can be derived if one of these other definitions of w is chosen

$$(52)\quad \begin{array}{lcl} w & = & +/ \cdot u{*} \cdot \mathit{it} \\ u\,x & = & \begin{array}{ll} C - (+/\, x) & \text{if } +/\, x \le C \\ \infty & \text{otherwise} \end{array} . \end{array}$$

If $C = 25$, the w-value of the solution given in (49) is 10. For an argument x, 'function' f thus defined may have several different solutions, all with equal w-value, and one of the important themes in previous papers (see Bird [2] and Fokkinga [6]) is to resolve this nondeterminism. In fact, we could specify a relation instead of a function, thus avoiding having to define $\downarrow_w$ on objects with equal w-value. However, we have chosen to stay within the functional framework. A derivation of one of the on-line algorithms for breaking a paragraph into lines in a relational framework is carried out by De Moor [17]. Here, the nondeterminism is resolved in an ad-hoc fashion by defining $\downarrow_w$ on objects with equal w-value as follows. Again, it is not clear to me whether incremental algorithms can be derived for other definitions of $\downarrow_w$. If $x =_w y$, then

$$x \downarrow_w y \quad = \quad (\mathit{it}\, x) \downarrow_w (\mathit{it}\, y) \; .$$

This concludes the specification of the problem. If we apply the specification to Jane Austen's sentence we obtain the solution given in (49).

We derive an incremental algorithm for the paragraph problem f specified in (50). For that purpose, we have to express f as a catamorphism, that is, we have to find an operator $\odot$ and a function r such that

$$f \quad = \quad \odot/ \cdot r{*} \; .$$

Since f is defined by $\downarrow_w/ \cdot (\mathit{all\, fit})\triangleleft \cdot \mathit{parts}$, we could try to apply the theory developed for function *parts* by for example Bird [5]. However, none of this theory is applicable. Intuitively, this can be seen as follows. If $f\,(x +\!\!+\, y)$ has to be expressed in terms of $f\,x$ and $f\,y$, the information obtained from $f\,y$ is useless. Instead of the information how to split y into lines where the first line starts at the left, we would like to have the information how to split y into lines after gluing an initial part of y to the last line of x. It follows that we want to have the f-values of all tails of y available. The specification of the paragraph problem is generalised to k.

$$(53)\quad k \quad = \quad f{*} \cdot \mathit{tails} \; .$$

Since *tails* is a catamorphism $\bigtriangledown/ \cdot [[\cdot]]{*}$, see (27), we can apply Promotion, Theorem 7. Suppose $f{*}\,(x \bigtriangledown y) = (f{*}\,x) \odot (f{*}\,y)$ for some operator $\odot$, then we have

$$(54)\quad f{*} \cdot \mathit{tails} \quad = \quad \odot/ \cdot (f{*} \cdot [[\cdot]]){*} \; .$$

Function $f{*} \cdot [[\cdot]]$ can be simplified as follows.

$$\begin{array}{ll} & f{*}\,[[\cdot]]\,a \\ = & \qquad \text{definition of } [[\cdot]] \\ & f{*}\,[[a]] \\ = & \qquad \text{definition of map, and } f \\ & [[[a]]] \; . \end{array}$$

It follows that $f* \cdot [[\cdot]] = [[[\cdot]]]$, where $[[[\cdot]]]$ is defined by $[[[\cdot]]]\, a = [[[a]]]$. Operator $\odot$ is synthesised as follows.

$$
\begin{array}{ll}
 & f*(x \bigtriangledown y) \\
= & \quad \text{definition of } \bigtriangledown \\
 & f*((+\!\!+(hd\ y))*x) +\!\!+ y \\
= & \quad \text{definition of map, map distributivity} \\
 & ((f \cdot (+\!\!+(hd\ y)))*x) +\!\!+ (f*y) \\
= & \quad \text{assumption} \\
 & ((\diamond(f*y))*f*x) +\!\!+ (f*y)\ ,
\end{array}
$$

where we have assumed that $f \cdot (+\!\!+(hd\ y)) = (\diamond(f*y)) \cdot f$, for some operator $\diamond$. So, provided there exists an operator $\diamond$ satisfying this requirement, we have found that

(55) $k \;=\; \odot/ \cdot [[[\cdot]]]*\ ,$

where operator $\odot$ is defined by

(56) $x \odot y \;=\; ((\diamond y)*x) +\!\!+ y\ .$

The derivation of an operator $\diamond$ is quite difficult and is omitted for reasons of space. It can be defined as follows.

$$
(57)\quad
\begin{array}{lcl}
\diamond & : & A** \times A*** \to A*** \\
x \diamond v & = & (((x,v)\oplus) \cdot ((lt\ x)\oslash)/ \cdot inits \cdot +\!\!+/ \cdot hd)\ v \\
\oslash & : & A* \times (A* \times A*) \to A* \\
c \oslash (a,b) & = & \begin{array}{ll} a \downarrow_{u\cdot(c+\!\!+)} b & \text{if } (fit\ c +\!\!+ a) \vee (fit\ c +\!\!+ b) \\ a \downarrow_u b & \text{if } (fit\ a) \vee (fit\ b) \\ 1_{\downarrow_u} & \text{otherwise} \end{array} \\
\oplus & : & (A** \times A***) \times A* \to A** \\
(x,v) \oplus c & = & ((x\otimes) \cdot ([c]+\!\!+) \cdot (\#c)!)\ v \\
\otimes & : & A** \times A** \to A** \\
x \otimes y & = & \downarrow_w / \ (all\ fit) \triangleleft (x \diamond y)\ .
\end{array}
$$

We give some operational understanding of the algorithm we have obtained. Given the values of $k\ a$ and $k\ b$, we have to find the value of $k\,(a +\!\!+ b)$. The value of $k\,(a +\!\!+ b)$ is the concatenation of a section $(\diamond(k\ b))$ mapped to $k\ a$, and $k\ b$ (this is expressed by operator $\odot$). Given an element x from $k\ a$, the section $(\diamond(k\ b))$ starts with finding the longest element c in $inits\ b$ which fits on one line when appended to the last line of x, or, when no element of $inits\ b$ appended to the last line of x fits on one line, the longest element of $inits\ b$ which fits on one line (this is expressed by operator $\oslash$). Operator $\oplus$ prepends the resulting partition to the $((\#\ c)+1)th$ element from $k\ b$.

The catamorphism given for k in (55) can be implemented such that the resulting algorithm requires quadratic time for its evaluation. We now show how a linear-time algorithm is obtained. Let c be $((lt\ x)\oslash)/\ \mathit{inits}\ b$. Since $((lt\ x)\oslash)/\ \mathit{inits}\ b$ returns the longest element in $\mathit{inits}\ b$ which fits on one line when appended to the last element of an element of $k\ a$, we have $\#\ c \leq C$. In fact, we even have $\#\ c \leq D$, where $D = \#\uparrow_{\#}/\ \mathit{fit}\triangleleft\ \mathit{segs}\,(a \mathbin{+\!\!\!+} b)$. This implies that computing the first $D+1$ elements of k suffices for our purposes. The final specification of our problem reads

$$(58)\quad l \;=\; (D+1 \rightharpoonup)\cdot k\ .$$

Again, we have $l = \odot'/\cdot[[[\cdot]]]*$, where the operator $\odot'$ is the following amendment of operator $\odot$.

$$(59)\quad x \odot' y \;=\; \begin{cases} (\Diamond y)*x & \text{if } \#\,x = D+1 \\ ((\Diamond y)*x) \mathbin{+\!\!\!+} ((D+1-(\#\,x)) \rightharpoonup y) & \text{otherwise}\ . \end{cases}$$

This algorithm can be implemented such that it requires linear time for its evaluation.

Since l is a catamorphism, Theorem 47 can now be applied to obtain an incremental algorithm for it. However, the resulting algorithm is hopelessly space inefficient. Therefore, we derive an alternative incremental algorithm for it. We first try to find a basic incremental algorithm for the paragraph problem l. Using function g defined in equality (41) it can be proved that $(\neq 1_{+\!\!\!+})\triangleleft\, g*\ x \odot' [[[a]]] = x$, so operator $\otimes$ of the basic incremental algorithm can be defined by $(\neq 1_{+\!\!\!+})\triangleleft\cdot g*\cdot \ll$. For the other component of the basic incremental algorithm, I can not find an efficiently computable operator $\oplus$ such that $a\oplus([[[a]]]\odot' x) = x$. Therefore, the search for a basic incremental algorithm for l is abandoned, and we try to find an incremental algorithm for it.

According to the definition of incremental algorithms, finding an incremental algorithm for l amounts to finding a left-reduction $\mathbin{\overset{\rightarrow}{\circ}}\!\nrightarrow e$ and a right-reduction $\mathbin{\overset{\uparrow}{\circ}}\!\nleftarrow u$ such that there exists functions α, and β, and operators $\mathbin{\overset{\leftarrow}{\circ}}$, and $\mathbin{\overset{\downarrow}{\circ}}$ satisfying the following equations.

$$\begin{array}{lll}
(60)\ \ l & = & \alpha\cdot \mathbin{\overset{\rightarrow}{\circ}}\!\nrightarrow e \\
(61)\ \ l & = & \beta\cdot \mathbin{\overset{\uparrow}{\circ}}\!\nleftarrow u \\
(62)\ \ (((\mathbin{\overset{\rightarrow}{\circ}}\!\nrightarrow e)\,x) \mathbin{\overset{\rightarrow}{\circ}} a) \mathbin{\overset{\leftarrow}{\circ}} a & = & (\mathbin{\overset{\rightarrow}{\circ}}\!\nrightarrow e)\,x \\
(63)\ \ a \mathbin{\overset{\downarrow}{\circ}} (a \mathbin{\overset{\uparrow}{\circ}} ((\mathbin{\overset{\uparrow}{\circ}}\!\nleftarrow u)\,x)) & = & (\mathbin{\overset{\uparrow}{\circ}}\!\nleftarrow u)\,x\ .
\end{array}$$

Function α and operators $\mathbin{\overset{\rightarrow}{\circ}}$, and $\mathbin{\overset{\leftarrow}{\circ}}$ are defined as follows. Since l is a catamorphism $\odot'/\cdot[[[\cdot]]]*$, we have $l = \mathbin{\overset{\rightarrow}{\circ}}\!\nrightarrow 1_{\odot'}$, where operator $\mathbin{\overset{\rightarrow}{\circ}}$ is defined by $x \mathbin{\overset{\rightarrow}{\circ}} a = x \odot' [[[a]]]$. So we may take $\alpha = id$. Furthermore, as has been remarked above, if we define operator $\mathbin{\overset{\leftarrow}{\circ}}$ by $(\neq 1_{+\!\!\!+})\triangleleft\cdot g*\cdot\ll$, then we have $(x \mathbin{\overset{\rightarrow}{\circ}} a) \mathbin{\overset{\leftarrow}{\circ}} a = x$ for all x and a.

Function β and operators $\mathbin{\overset{\uparrow}{\circ}}$ and $\mathbin{\overset{\downarrow}{\circ}}$ are defined as follows. For right-reduction $\mathbin{\overset{\uparrow}{\circ}}\!\nleftarrow u$ we take the right-reduction for function k. Since k is a catamorphism $\odot/\cdot[[[\cdot]]]*$, we have $k = \mathbin{\overset{\uparrow}{\circ}}\!\nrightarrow 1_{\odot}$, where operator $\mathbin{\overset{\uparrow}{\circ}}$ is defined by $x \mathbin{\overset{\uparrow}{\circ}} a = [[[a]]] \odot x$. By definition of l and k we have $l = (D+1 \rightharpoonup)\cdot k$, so we let β be the section $(D+1 \rightharpoonup)$. If we define operator $\mathbin{\overset{\downarrow}{\circ}}$ by $tl\cdot\gg$, then $a \mathbin{\overset{\downarrow}{\circ}} (a \mathbin{\overset{\uparrow}{\circ}} x) = x$ for all x and a.

We have found the following incremental algorithm for l.

$$(\odot'/ \cdot [[[\cdot]]]* \,, \ominus\not\rightarrow\!1_{\odot'} \,, \ominus \,, id \,, \oplus\not\leftarrow\!1_{\odot} \,, \oplus \,, (D+1 \rightharpoonup))\,.$$

Another problem for which I have derived an incremental algorithm is the problem of coding a text with respect to a dictionary. This algorithm can be used in algorithms for data compression. On-line algorithms for this problem are well known, see e.g. Rodeh, Pratt, and Even [21]. The specification of this problem is similar to the specification of the paragraph problem; it is specified by

(64) $\downarrow_{\#}/ \cdot (all\,(\in D))\triangleleft \cdot parts$,

where D is a dictionary. The derivation of an efficient algorithm for this problem uses another recursive characterisation of function *parts*.

4 Future work

The work reported on here is part of ongoing research on incremental algorithms. The theory developed thus far is specific to lists. A first natural extension to the theory is to model the notion of cursor in the Boom-hierarchy (see Section 2), and to develop theory for the four data types in this hierarchy in one go. A second extension is to model incremental data types by the admissible edit actions; to let an incremental data type be modelled by the initial data type induced by a set of constructors corresponding to the edit actions. Together with the development of theory of incremental data types, examples of incremental algorithms should be derived to support the development of theory.

Acknowledgements. The discussions on incremental algorithms with Richard Bird, Maarten Fokkinga, and Jaap van der Woude are gratefully acknowledged. Furthermore, Maarten Fokkinga and Jaap van der Woude commented on previous versions of this paper and pointed out several ways to improve the presentation and contents.

References

[1] J.O. Achugbue. On the line breaking problem in text formatting. *ACM SIGOA Newsletter*, 2(1 & 2):117–121, 1981.

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

[3] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer–Verlag, 1987.

[4] R.S. Bird. Lectures on constructive functional programming. In M. Broy, editor, *Constructive Methods in Computing Science*, volume F55 of *NATO ASI Series*, pages 151–216. Springer–Verlag, 1989.

[5] R.S. Bird. Small specification exercises. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty Is Our Business, A Birthday Salute to Edsger W. Dijkstra*, pages 390–398. Springer-Verlag, 1990.

[6] M.M. Fokkinga. Using underspecification in the derivation of some optimal partition algorithms. CWI, Amsterdam, 1990.

[7] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[8] J.A. Goguen. Memories of ADJ. *Bulletin of the EATCS*, 39:97–102, 1989.

[9] J. Jeuring. Deriving algorithms on binary labelled trees. In P.M.G. Apers, D. Bosman, and J. van Leeuwen, editors, *Proceedings SION Computing Science in the Netherlands*, pages 229–249, 1989.

[10] J. Jeuring. Algorithms from theorems. In M. Broy and C.B. Jones, editors, *Programming Concepts and Methods*, pages 247–266. North-Holland, 1990.

[11] D.E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

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

[13] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[14] L. Meertens. Algorithmics—towards programming as a mathematical activity. In J.W. de Bakker, M. Hazewinkel, and J.K. Lenstra, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[15] L. Meertens. Paramorphisms. Technical Report CS-R9005, CWI, 1990. To appear in Formal Aspects of Computing.

[16] B. Meyer. Incremental string matching. *Information Processing Letters*, 21:219–227, 1985.

[17] O. de Moor. Categories, relations and dynamic programming. submitted for publication, 1991.

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

[19] S. Pemberton. Views: An open-architecture user-interface system. to appear in: Proceedings of Interacting with computers, preparing for the nineties, Noordwijkerhout, The Netherlands, 1990.

[20] T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, 1983.

[21] M. Rodeh, V.R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

[22] D.R. Smith. Applications of a strategy for designing divide-and-conquer algorithms. *Science of Computer Programming*, 8:213–229, 1987.

[23] S. Thompson. Interactive functional programs, a method and a formal semantics. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 249–285. Addison-Wesley Publishing Company, 1990.

[24] D.A. Turner. An overview of Miranda. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 1–16. Addison-Wesley Publishing Company, 1990.

[25] D. Yellin and R. Strom. INC: A language for incremental computations. *ACM Transactions on Programming Languages and Systems*, 13(2):211–236, 1991.