Incremental Data Compression —extended abstract—

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

1 Introduction

Data compression is used in data transmission and data storage. When data compression is used, data is transmitted faster, and file storage requires less space. Many aspects of data compression are described in Leweler and Hirschberg [6], and Storer [9]. An important technique for data compression is textual substitution. Textual substitution identifies repeated substrings and replaces some or all substrings by pointers to another copy. Here we consider a specific textual substitution method: coding a text with respect to a dictionary. Suppose a dictionary is given, consisting of all 128 ASCII characters together with the 32640 most common substrings of two or more characters of printed English text. The problem of coding a text with respect to this dictionary is to partition the text into as few strings as possible, each of which is contained in the dictionary, and to replace each string in this partition by a 15 bit pointer corresponding to the entry in the dictionary. In the sequel we assume a dictionary D satisfying the following property is given. For each element x in D all segments (subwords) of x are also in D. Various algorithms have been constructed for coding a text with respect to a dictionary; we give an incremental algorithm for it.

Incremental computations can improve the performance of interactive programs such as spreadsheet programs, program development environments, text-editors, etc. Let f be a function. Suppose we want to find the f-value on some input, and that this input is interactively edited. An incremental algorithm prescribes how to recompute the required f-value on the input, after the input has been edited, using the old f-value and possibly some extra information. The input to an algorithm for coding a text is a list of characters. A theory of incremental algorithms on the data type list, based on the Bird-Meertens calculus for program transformation, see Bird [1], [2], and Meertens [7], [8], is presented in

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

Jeuring [3]. The possible edit actions are among others deletion or insertion of elements in the argument list, splitting the argument, and combining two arguments. With an incremental algorithm for the problem of coding a text with respect to a dictionary it is possible to combine two coded texts in constant time. Furthermore, deleting or inserting a piece of text and coding the resulting text is done in time linear in the length of the deleted or inserted piece of text. The advantages of an incremental algorithm for coding a text with respect to a dictionary are immediate: the uncoded text is not needed anymore, it suffices to have the coded text in main memory. And users need not explicitly apply coding algorithms to their texts, each text is automatically coded.

We give an informal description of the incremental algorithm for coding a text with respect to a dictionary. The main observation is that the length of the words of the dictionary is bounded by some constant L. The most important part of the algorithm is the part that combines two coded texts x and y. This part is a dynamic programming algorithm. The coding of the concatenation of x and y (list concatenation is denoted by $+\!\!+$, a singleton list with element a is written $[a]$) is the shortest list of pointers $xi +\!\!+ [a] +\!\!+ yt$ such that a is a pointer to a word in the dictionary corresponding to a tail part of x concatenated with an initial part of y, xi is the coding of the remaining initial part of x, and yt is the coding of the remaining tail part of y. Since the length of the word corresponding to a is bounded by L, it suffices to have the codings of the longest $L{+}1$ initial parts of x and the codings of the longest $L{+}1$ tail parts of y available. If these are available the coding of $x +\!\!+ y$ can be determined in constant time. The other parts of the incremental algorithm are variations on this theme.

To our knowledge, our algorithm is the first incremental algorithm for coding a text with respect to a dictionary. Many algorithms incrementally build a dictionary using so-called dynamic dictionary methods, see Storer [9], but no algorithms have been given for incrementally computing the coding of a text with respect to a static dictionary. Katajainen and Mäkinen [5] describe methods for incrementally coding trees, but they do not use textual substitution. Some of the ideas we use in the construction of the algorithm are based on ideas described in Jeuring [3], where a theory of incremental algorithms on lists is presented, and an incremental algorithm for formatting a text is given.

An interesting topic for further research is the construction of incremental algorithms for data compression using dynamic dictionary methods. Furthermore, the ideas applied in the construction of the incremental algorithm sketched in this paper can be used in the construction of an algorithm for coding a tree with respect to a dictionary of trees.

This paper is organised as follows. Section 2 describes the edit model and the induced form of incremental algorithms. Section 3 specifies the problem of coding a text with respect to a dictionary, and Section 4 sketches a derivation of an incremental algorithm for this problem.

2 Incremental algorithms and the edit model

We give a definition of an incremental algorithm defined on the data type *list*. The data type *list* over some base type A is denoted by $A*$. The empty list is denoted by $\Box$. Given an element a of type A, the singleton list with element a is written $[a]$. Given two lists x and y, the concatenation of x and y is written $x +\!\!+ y$. Operator $+\!\!+ : A* \times A* \to A*$ is associative, and the empty list is the unit of $+\!\!+$, that is, for all lists x, $x +\!\!+ \Box = \Box +\!\!+ x = x$. The list with consecutive elements 1, 2, and 8, formally $[1] +\!\!+ [2] +\!\!+ [8]$, is written $[1, 2, 8]$. The *length* of a list is computed by means of the function $\# : A* \to nat$, e.g. $\#\,[1, 2, 8] = 3$.

A *function* is an object with three components written $f : s \to t$, where s is a set called the source of the function, t is a set called the target of the function, and f maps each member x of s to a member of t. This member is denoted $f\,x$, using simple juxtaposition and a little white space to denote application of a function f to an argument x. We use the letters f, g, h, etc., as variables standing for arbitrary functions. Function application is right-associative, i.e., we have $f\,(g\,(h\,x)) = f\,g\,h\,x$. The *composition* of two functions $f : s \to t$ and $g : r \to s$ is written $f \cdot g : r \to t$. Composition is associative. An example of a function is the *identity function*: for each set s, function $id_s : s \to s$ is the identity function on the set s. Given functions $f : A \to B$ and $g : C \to D$, function $f \times g : A \times C \to B \times D$ is defined by $(f \times g)\,(a, c) = (f\,a, g\,c)$, where $A \times B$ is the *cartesian product* of sets (types) A and B. Given functions $f : A \to B$ and $g : A \to C$, function $f \vartriangle g : A \to B \times C$ is defined by $(f \vartriangle g)\,a = (f\,a, g\,a)$.

Suppose we want to find the f-value of a list, and that we are interactively editing this list. An *incremental algorithm* prescribes how to recompute the f-value after an edit action has changed the input to f. It follows that the form of an incremental algorithm is determined by the edit model. In this section we describe the edit model and the form of incremental algorithms.

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.

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

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.

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 associative operator $\odot$ such that $f\,(x +\!\!+\, y) = (f\,x) \odot (f\,y)$. A function defined on lists satisfying this property is called a catamorphism by Meertens [8]. 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. The incremental algorithm for coding with respect to a dictionary sketched later is a catamorphism which can be implemented as a linear-time program. If operator $\odot$ of the catamorphism can be evaluated in constant time, the resulting catamorphism can be implemented as a linear-time program.

Formally, a *catamorphism* is the unique function h of type $A* \to B$ that satisfies for value $e : B$, function $f : A \to B$ and associative operator $\oplus : B \times B \to B$ with unit e:

$$
\begin{array}{lll}
h\,\Box & = & e \\
h\,[a] & = & f\,a \\
h\,(x +\!\!+\, y) & = & (h\,x) \oplus (h\,y)\ .
\end{array}
\qquad (1)
$$

Function h is denoted by $([e, f, \oplus])$. For example, catamorphism $([0, id, +])$, where id is the identity function, sums the elements of a list of integers. A catamorphism of which the second component is the identity function id is called a *reduction* and we write $\oplus/ = ([e, id, \oplus])$. Another special catamorphism-former is the *map-operator* which takes a function and a list and applies the function to each element in the list. Given a function f the function f-map is written $f*$ and it is defined by $f* = ([\Box, \tau \cdot f, +\!\!+])$, where $\tau\,a = [a]$. For example, $(+3)* : nat* \to nat*$ is the function that takes a list and adds 3 to each element of the list. So $(+3)*\,[3, 1, 6] = [6, 4, 9]$. Function $(+3) : nat \to nat$ is an example of a *section*: a binary operator $\oplus$ is turned into a unary operator when supplied with a right or a left argument. So if $\oplus : A \times B \to C$, then $(a\oplus) : B \to C$, and $(\oplus b) : A \to C$. Each catamorphism $([e, f, \oplus])$ can be written as the composition of a reduction with a map: $([e, f, \oplus]) = \oplus/ \cdot f*$.

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 allow some extra freedom. Let functions g and h be such that there exist (efficiently computable) functions α and β such that $f = \alpha \cdot g$ and $f = \beta \cdot h$. We assume that the computation of the f-value is tupled with the computation of the g-value of the list in front of the cursor $(g\,x)$, and the h-value of the list after the cursor $(h\,y)$. 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 via α and β. Concluding, we have assumed that the interactive program is extended with the

computation of a triple of values: the g-value of the list in front of the cursor, the f-value of the argument list, and the h-value of the list after the cursor.

The cursor movements are dealt with as follows. 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 $g\,x$ and $h\,([a,b] +\!\!+ y)$ from the values $g\,(x +\!\!+ [a])$, $h\,([b] +\!\!+ y)$, and $f\,(x +\!\!+ [a,b] +\!\!+ y)$. To fulfill these requirements, we make two additional assumptions. First, we suppose that there exists an operator $\oslash$ such that $g\,x = (g\,(x +\!\!+ [a])) \oslash a$. Second, we assume there exists an operator $\oplus$ such that $h\,([a,b] +\!\!+ x) = a \oplus (h\,([b] +\!\!+ x))$.

When the cursor is moved right it is required to find the values $g\,(x +\!\!+ [a,b])$ and $h\,y$ from the values $g\,(x +\!\!+ [a])$, $h\,([b] +\!\!+ y)$, and $f\,(x +\!\!+ [a,b] +\!\!+ y)$. We assume there exists an operator $\otimes$ such that $g\,(x +\!\!+ [a,b]) = (g\,(x +\!\!+ [a])) \otimes b$, and we assume that there exists an operator $\ominus$ such that $a \ominus (h\,([a,b] +\!\!+ x) = h\,([b] +\!\!+ x)$.

The form of incremental algorithms assumed until now provides an elegant way to deal with insertion and deletion of one or more elements. Deletion and insertion are described after the definition of an incremental algorithm.

(2) Definition (Incremental algorithm) *An incremental algorithm is a triple of functions (f,g,h) such that there exist operators $\odot$, $\otimes$, $\oslash$, $\oplus$, $\ominus$, and functions r, α, β such that (the notations $\not\rightarrow$ and $\not\leftarrow$ are defined below)*

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

Given operator $\oplus : A \times B \to B$, and value $e : B$, function $\oplus \not\leftarrow e : A* \to B$ is called a *right-reduction*, and it is defined as the unique function that satisfies the following two equations.

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

Left-reductions are defined similarly. Given operator $\oplus : A \times B \to B$, and value $e : B$, function $\oplus \not\rightarrow e : A* \to B$ is defined by

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

If the operator $\uparrow$: $nat \times nat \to nat$ returns the largest of two natural numbers, the left-reduction $\uparrow \not\rightarrow 0$ returns the maximum of a list.

An incremental algorithm handles insertion or deletion of a list in the following manner. Suppose a list z is inserted in between the two lists x and y, so the triple $(g\,x, f\,(x +\!\!+ y), h\,y)$

should be transformed into $(g\,(x +\!\!+ z), f\,(x +\!\!+ z +\!\!+ y), h\,y)$. To obtain this triple: split $x +\!\!+ y$ and compute $g\,(x +\!\!+ z) = (\otimes\!\!\not\rightarrow(g\,x))\,z$, and $g\,z$, and then compute $(f\,x) \oplus (\alpha\,g\,z) \oplus (f\,y)$. 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 $g\,x$ and $h\,y$ are now available, the triple $(g\,x\,, f\,(x +\!\!+ y)\,, h\,y)$ can be computed.

3 The specification

In this section we give the specification of the problem of coding a text with respect to a dictionary. To code a text with respect to a dictionary D it is required to partition the text into as few elements as possible each of which occurs in D, and to replace each element of this partition by a pointer into the dictionary. For a formal, functional, specification we introduce the following notions. Let *parts* be a function which enumerates in a bag all ways in which a list can be broken into lists of lists. Function *parts* is defined formally after the specification.

The elements in the data type *bag*, also known as multiset, are sets with possibly multiple occurrences of equal elements, or, equivalently, lists with no order imposed on the elements. The operator $\uplus$ denotes bag union. Bag union is associative, commutative and the empty bag $\langle\,\rangle$ is its unit. The bag with elements 1, 2, and 2 is written $\langle 1, 2, 2\rangle$. Function-formers like reduce and map defined on the data type list can be defined in a similar fashion on the data type bag.

A *predicate* is a function with the type *boolean* as its target. The predicate $\in D : A* \rightarrow$ *bool* determines whether a list occurs in dictionary D or not. Let p be a predicate of type $A \rightarrow$ *bool*. The predicate *all* $p : A* \rightarrow$ *bool* is defined by *all* $p = \wedge/ \cdot\, p*$.

The *filter operator* $\triangleleft$ is a catamorphism on the data type *bag*. Filter $\triangleleft$ takes a predicate and a bag and retains the elements satisfying the predicate in a bag; so $odd \triangleleft \langle 3, 2, 4, 1\rangle = \langle 3, 1\rangle$.

Operator $\downarrow_{\#}$ is a binary operator of type $A* \times A* \rightarrow A*$. It is defined by

$$(5)\quad x \downarrow_{\#} y \;=\; \begin{array}{ll} x & \text{if } (\#\,x) < (\#\,y) \\ y & \text{if } (\#\,x) > (\#\,y) \\ x \text{ or } y & \text{otherwise .} \end{array}$$

We do not yet define $\downarrow_{\#}$ on arguments which have equal #-values, except that one of the arguments is the outcome.

The specification of the problem of coding a text with respect to a dictionary D reads as follows.

$$(6)\quad coding \;=\; repl* \cdot cod\ ,$$

where function *cod* is defined by

$$(7)\quad cod \;=\; \downarrow_{\#}/ \cdot\, (all \in D) \triangleleft \cdot\, parts\ ,$$

and *repl* is a function that replaces a word by a pointer. The precise definition of *repl* is not given. This specification can easily be implemented in a functional language, but this implementation is very inefficient. Given an efficient way to determine the value of *cod*, the value of *coding* is obtained easily. In the sequel we will deal with function *cod* only.

We want to construct an incremental algorithm the components of which can be implemented as efficient programs. The first task is to give a catamorphism $\odot/ \cdot r*$ equal to *cod*. Operator $\odot$ should satisfy $cod\,(x +\!\!+ y) = (cod\,x) \odot (cod\,y)$. For the construction of such an operator $\odot$, function *parts* has to be recursively characterised. The form of the definition of *parts* and *cod* is related, since *cod* is defined in terms of *parts*. In view of constructing a catamorphism for *cod*, function *parts* is defined as follows.

Function *parts* enumerates when applied to a list x all lists of lists y such that $x = +\!\!+/\, y$. There are various recursive characterisations of the function *parts*; it can be characterised as a left-reduction, a catamorphism, etc. Here we give a recursive characterisation of *parts* based on the following observation. For nonempty z

$$parts\, z \;=\; \langle u +\!\!+ [v] +\!\!+ w \mid z = a +\!\!+ v +\!\!+ b, u \in parts\, a, w \in parts\, b, v \neq \Box \rangle\ .$$

We have $parts\,\Box = \langle \Box \rangle$ and $parts\,[a] = \langle [[a]] \rangle$. For the case $parts\,(x +\!\!+ y)$ we use the above observation, the operator cross, and the function *splits* (both introduced below).

$$(8)\quad parts\,(x +\!\!+ y) \;=\; \uplus/\,((parts \times id)*\, splits\, x) \,\backslash\!\!\!\backslash_{\odot}\, ((id \times parts)*\, splits\, y)$$

$$(9)\quad (x, y) \odot (u, v) \;=\; x \,\backslash\!\!\!\backslash_{\ominus_{y +\!\!+ u}}\, v$$

$$(10)\quad a \ominus_c b \;=\; a +\!\!+ [c] +\!\!+ b\ .$$

Operator *cross*, denoted by $\backslash\!\!\!\backslash$, 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, $[1,2] \,\backslash\!\!\!\backslash\, [3,4,5] = \langle (1,3), (1,4), (1,5), (2,3), (2,4), (2,5) \rangle$. The definition of operator cross can be found in Bird [2] and is not repeated here. Operator cross can be subscripted with a binary operator, by which we mean the following.

$$(11)\quad \backslash\!\!\!\backslash_{\oplus} \;=\; \oplus* \cdot \backslash\!\!\!\backslash\ .$$

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

The function *splits* enumerates in a list all possible ways to split a list in two parts. For example, $splits\,[2,3,1] = [(\Box, [2,3,1]), ([2], [3,1]), ([2,3], [1]), ([2,3,1], \Box)]$. We omit the formal definition of *splits*.

4 The derivation of an algorithm

In this section we give a sketch of the derivation of an efficient incremental algorithm for the problem of coding a text with respect to a dictionary specified in (7). Space does not permit us to present a full derivation, instead we show the main steps and explain them giving some operational understanding of the expressions. The full derivation and the complete algorithm will be presented elsewhere (Jeuring [4]).

The first and most difficult problem that has to be solved is the construction of a catamorphism $\odot/ \cdot r*$, that can be implemented as an efficient program, such that $cod = \odot/ \cdot r*$. Function r is defined by $r\, a = cod\, [a]$, and operator $\odot$ should satisfy $cod\, (x +\!\!+ y) = (cod\, x) \odot (cod\, y)$. We have not been able to construct a catamorphism for cod that can be implemented as a linear-time program. The approach usually taken in such cases is the following. Consider a related problem $gcod$ for which there exists a cheap function j satisfying $j \cdot gcod = cod$, such that there exists an incremental algorithm for $gcod$ that can be implemented as an efficient program. The definition of $gcod$ is suggested by inspecting the expression $cod\, (x +\!\!+ y)$. Instantiating the definitions of cod, $parts$, and cross, and applying several laws for cross and the other components of the expression gives the following equation. The coding of list $x +\!\!+ y$ $(cod\, (x +\!\!+ y))$ is the shortest list among the lists that are a coding of an initial part of x, concatenated with a list that is the concatenation of the remaining tail of x and an initial part of y (if this concatenation is in the dictionary), concatenated with the coding of the remaining tail of y. Formally,

$$(12)\quad cod\, (x +\!\!+ y) \;=\; \downarrow_{\#}/\, ((cod \times id)*\, splits\, x) \,\rangle\!\!\rangle_{\oplus}\, ((id \times cod)*\, splits\, y)$$

$$(13)\quad (x, y) \oplus (u, v) \;=\; \begin{cases} x +\!\!+ [y +\!\!+ u] +\!\!+ v & \text{if } (y +\!\!+ u) \in D \\ \omega & \text{otherwise} \end{cases},$$

where ω is a unit of $\downarrow_{\#}$, that is, for all lists x we have $x \downarrow_{\#} \omega = \omega \downarrow_{\#} x = x$. Note the resemblance of these equations with equations (8) and (10) for the function $parts$. In the calculation of equation (12) we have assumed that $\downarrow_{\#}$ is defined as follows on lists of lists. Let x and y be lists of lists. If $\#\, x \neq \#\, y$, then $x \downarrow_{\#} y$ is defined as in Section 3. If $\#\, x = \#\, y$ we define

$$(14)\quad x \downarrow_{\#} y \;=\; \begin{cases} x & \text{if } \#\, hd\, x > \#\, hd\, y \\ y & \text{if } \#\, hd\, y > \#\, hd\, x \\ [hd\, x] +\!\!+ ((tl\, x) \downarrow_{\#} (tl\, y)) & \text{otherwise} \end{cases},$$

where hd returns the first element of a list. This definition of operator $\downarrow_{\#}$ implies that if two partitions have equal length, then operator $\downarrow_{\#}$ returns the one with the longest lists at the left end of the list. Equation (12) and the wish to construct a catamorphism that can be implemented as a linear-time program suggest to find the codings of all initial and all tail parts of the argument. Specification cod (7) is extended to a specification of $gcod$, where $gcod$ is defined by

$$(15)\quad gcod \;=\; (cod \times id)* \vartriangle (id \times cod)* \cdot splits\ .$$

Function cod is expressed in terms of $gcod$ by means of $cod = \gg \cdot hd \cdot \gg \cdot gcod$, where $\gg$ ($\ll$) returns the right (left) element of a pair. Since $\gg \cdot hd \cdot \gg$ is a cheap function, the remaining task is to fulfill the second condition imposed upon $gcod$: the construction of an incremental algorithm for $gcod$ that can be implemented as a linear-time program. Again, we have not been able to construct a catamorphism for $gcod$ that can be implemented as a linear-time program. We specify yet another function $hcod$ for which there exists a cheap function j satisfying $j \cdot hcod = cod$, and for which we can construct an incremental

algorithm $(hcod, g, h)$ that can be implemented as an efficient program. The specification of the function *hcod* is given after the following observation.

Function *hcod* is a slight variant of function *gcod*. Observe that the length of the longest word in the dictionary is $L = \#\uparrow_{\#}/\,D$. According to equation (12), the value of $cod\,(x +\!\!+ y)$ is an element $(a, b) \oplus (c, d)$ where $(a, b) \in \ll gcod\,x$ and $(c, d) \in \gg gcod\,y$. All elements (a, b) of $\ll gcod\,x$ with $\#\,b > L$ can be discarded, since $b +\!\!+ c$ is not an element of D, and $\oplus$ evaluates to ω for these elements. Similarly, all elements (c, d) of $\gg gcod\,y$ with $\#\,c > L$ can be discarded. It follows that for the computation of *cod* it suffices to have the last $L{+}1$ elements of $\ll gcod\,x$ and the first $L{+}1$ elements of $\gg gcod\,y$ available. Function *hcod* is specified in terms of *gcod* as follows.

$$(16)\quad hcod \;=\; (L{+}1\leftharpoonup) \times (L{+}1 \rightharpoonup) \cdot gcod\ ,$$

where functions $(L{+}1\leftharpoonup)$ and $(L{+}1\rightharpoonup)$ are defined as follows. Function $(L{+}1\leftharpoonup)$ returns, given a list x, the tail part of length $L{+}1$ of x. If $\#\,x \leq L{+}1$, then $(L{+}1\leftharpoonup)$ is the identity function. Similarly, function $(L{+}1\rightharpoonup)$ returns the first $L{+}1$ elements of its argument list, and if $\#\,x \leq L{+}1$, then $(L{+}1\rightharpoonup)$ is the identity function.

Function *cod* is expressed in terms of *hcod* by *hcod* by $cod = \gg \cdot hd \cdot \gg \cdot hcod$. Function *hcod* is a catamorphism $\odot/ \cdot r*$ that can be implemented as a linear-time program. Function r is defined by $r\,a = hcod\,[a]$, with

$$hcod\,[a] \;=\; ([(\Box, [a]), ([[a]], \Box)], [(\Box, [[a]]), ([a], \Box)])\ .$$

It remains to find an operator $\odot$ such that $hcod\,(x +\!\!+ y) = (hcod\,x) \odot (hcod\,y)$. We do not give the exact definition of $\odot$, we merely point out some of the more interesting details. Split the computation of $hcod\,(x +\!\!+ y)$ in the computation of $\ll hcod\,(x +\!\!+ y)$ and $\gg hcod\,(x +\!\!+ y)$. We will only discuss $\ll hcod\,(x +\!\!+ y)$. We show how to express $\ll hcod\,(x +\!\!+ y)$ in terms of $hcod\,x$ and $hcod\,y$.

$$\begin{array}{ll} & \ll hcod\,(x +\!\!+ y) \\ = & \quad \text{omitted calculation} \\ & ((L{+}1\leftharpoonup)\,(cod \times (+\!\!+ y))*\,splits\,x) +\!\!+ (((cod \cdot (x +\!\!+)) \times id)*\,splits\,y) \\ = & \quad \text{definition of } (L{+}1\leftharpoonup) \\ & ((L{-}\#\,y\leftharpoonup)\,(cod \times (+\!\!+ y))*\,splits\,x) +\!\!+ ((L{+}1\leftharpoonup)\,((cod \cdot (x +\!\!+)) \times id)*\,splits\,y)\ . \end{array}$$

Consider the left-hand argument and the right-hand argument of $+\!\!+$ seperately. For the left-hand argument of $+\!\!+$ we have

$$\begin{array}{ll} & (L{-}\#\,y\leftharpoonup)\,(cod \times (+\!\!+ y))*\,splits\,x \\ = & \quad \text{split } \times \text{ and map} \\ & (L{-}\#\,y\leftharpoonup)\,(id \times (+\!\!+ y))*\,(cod \times id)*\,splits\,x \\ = & \quad \text{definition of } hcod\text{, swap map and } (L{+}1\leftharpoonup) \\ & (id \times (+\!\!+ y))*\,(L{-}\#\,y\leftharpoonup) \ll hcod\,x\ . \end{array}$$

For the right-hand argument of $+\!\!+$ we have

$$
\begin{array}{ll}
 & (L{+}1 \leftharpoonup)\,((cod \cdot (x +\!\!+)) \times id)* \; splits\; y \\
= & \quad \text{swap } (L{+}1 \leftharpoonup) \text{ and the map-expression} \\
 & ((cod \cdot (x +\!\!+)) \times id)* \,(L{+}1 \leftharpoonup)\; splits\; y\ .
\end{array}
$$

Let (y_1, y_2) be an arbitrary element of $(L{+}1 \leftharpoonup)\; splits\; y$. It is required to express $cod\,(x +\!\!+ y_1)$ in terms of $hcod\; x$ and $hcod\; y$. Applying equation (12) we obtain

$$(17)\quad cod\,(x +\!\!+ y_1) \;=\; \downarrow_{\#}/\,(\ll hcod\; x)\; \}_{\oplus}\; (\gg hcod\; y_1)\ ,$$

so it suffices to express $\gg hcod\; y_1$ in terms of $hcod\; y$. There are various ways to express $\gg hcod\; y_1$ in terms of $hcod\; y$. One way is to peel off the last $\#\, y - \#\, y_1$ elements of each of the elements of $\gg hcod\; y$, giving $\gg hcod\; y_1$. An important property of function cod used here is that if xi is an initial part of x, then $cod\; xi$ is the initial part of $cod\; x$ with xi elements. This is a consequence of the assumption that if x is a word in D, then all segments of x are also words in D.

To obtain an incremental algorithm for $hcod$, we want to find functions g and h satisfying the following. Function g is a left-reduction $\otimes \not\rightarrow e$ such that there exists an operator $\oslash$ satisfying $(((\otimes \not\rightarrow e)\, x) \otimes a) \oslash a = (\otimes \not\rightarrow e)\, x$, and there exists a function α such that $hcod = \alpha \cdot g$. Function h is a right-reduction $\oplus \not\leftarrow u$ such that there exists an operator $\ominus$ satisfying $a \ominus (a \oplus ((\oplus \not\leftarrow u)\, x)) = (\oplus \not\leftarrow u)\, x$, and there exists a function β such that $hcod = \beta \cdot h$. For an efficient incremental algorithm the components α, $\otimes$, $\oslash$, β, $\oplus$, $\ominus$ should be such that their implementations can be evaluated in constant time.

An obvious candidate for both function g and function h is function $hcod$ itself. The characterisation of $hcod$ as a catamorphism immediately provides characterisations of $hcod$ as a left-reduction and as a right-reduction. Functions α and β are the identity function. The only problem is the definition of the operators $\oslash$ and $\ominus$. We have not been able to construct an operator $\ominus$ that, given the codings of the longest $L{+}1$ tails of $[a] +\!\!+ y$, returns the codings of the longest $L{+}1$ tails of y, and that can be implemented as a function that can be evaluated in constant time. Instead of function $hcod$ we take the following variants of function $gcod$ as the other components of the incremental algorithm. Functions g and h that do satisfy the conditions imposed upon incremental algorithms are specified by

$$(18)\quad g \;=\; (\ll *) \times (L{+}1 \rightharpoonup) \cdot gcod$$
$$(19)\quad h \;=\; (L{+}1 \leftharpoonup) \times (\gg *) \cdot gcod\ .$$

We briefly discuss the equations h is supposed to satisfy. The first task is to exhibit a function β such that $hcod = \beta \cdot h$. Since $hcod = (L{+}1 \leftharpoonup) \times (L{+}1 \rightharpoonup) \cdot gcod$ and $h = (L{+}1 \leftharpoonup) \times (\gg *) \cdot gcod$, it suffices to express $(L{+}1 \rightharpoonup) \cdot (id \times cod)* \cdot splits$ in terms of $(cod \cdot \gg)* \cdot splits$. This is fairly easy, and does not require any understanding of the algorithm. Therefore, we omit the definition of function β. The definition of operator $\oplus$ from the right-reduction $\oplus \not\leftarrow u$ can be derived from the definition of operator $\odot$ from the catamorphism $\odot/ \cdot r*$ for $hcod$. Finally, operator $\ominus$ is an operator of the form $a \ominus (x, y) = (k\; y, tl\; y)$, where k is some rather complicated, but not very interesting, function.

Acknowledgements. The fruitful discussions with Jaap van der Woude and Eddy Boeve on the subject of this paper are gratefully acknowledged.

References

[1] 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.

[2] 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.

[3] J. Jeuring. Incremental algorithms on lists. In J. van Leeuwen, editor, *Proceedings SION Computing Science in the Netherlands*, pages 315–335, 1991.

[4] J. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, Utrecht University, 1992. To appear.

[5] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *International Journal of Foundations of Computer Science*, 1(4):425–447, 1990.

[6] D.A. Leweler and D.S. Hirschberg. Data compression. *ACM Computing Surveys*, 19(3):261–296, 1987.

[7] 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.

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

[9] J.A. Storer. *Data Compression; Methods and Theory.* Computer Science Press, 1988.