Chapter II.9
Compression / Decompression

Introduction

Something about the aim of this chapter.

LZW compression

General compression methods make use of the redundancy present in most of the source texts. The idea is to replace parts of such texts by codes that occupy less space. It is often desirable that, during encoding, no information gets lost, that is, it should be possible to reconstruct the original source text from its encoded representation. Note that this property is not always demanded, especially when dealing with data representing sounds or images the loss of data might be acceptable as long as it does not lead to a significant decline of quality (e.g. JPEG compression, see also ???).

The *LZW* compression algorithm maintains a table that associates code values with substrings occurring in the input text. This *LZW* table satisfies the so-called *prefix property*: if the table contains a certain string s, all the prefixes of s are also present. The construction of an *LZW* table is done as follows. The initial table contains all possible singleton strings. Then, the input data is taken one character at a time to determine the longest string that already exists in the table. The output is the corresponding code value and the process is repeated on the remainder of the input data. At the same time, the table is extended with a new entry consisting of the matched string extended with the next character from the input, and a new code value.

To illustrate this compression method, consider the string *mississippi* and an initial *LZW* table containing

[(i,0),(m,1),(p,2),(s,3)]

The longest initial part of the input that matches a table entry is the string *m*. The output is 1, and moreover, the string *mi* is added to the table which then looks as follows

[(i,0),(m,1),(p,2),(s,3),(mi,4)]

The process is repeated on the remainder of the input, being *ississippi*. It is easy to verify that, when all input data has been consumed, this has led to the output

1,0,3,3,5,7,2,2,0

containing two items less than the input. Furthermore, the *LZW* table produced by this process consists of

[(i,0),(m,1),(p,2),(s,3),(mi,4),(is,5),(ss,6),(si,7),(iss,8),(sip,9),(pp,10),(pi,11)]

The resulting codes are written to a file, preferably in such a way that they occupy a minimal amount of space. It is clear that the sizes of the code values depend on the size of the *LZW* table: if the table grows, code sizes will increase. In the previous example, the 4 different input characters can be represented by 2 bits, thus the whole input string by 22 bits. The output codes require 4 bits for each code, yielding 36 bits in total (actually, it is 32 bits, for, one can use 3 bits as long as the table contains less than 8 elements).

In the Clean version of the *LZW* algorithm we take 8 bits characters as input, and use a fixed table size of 4096 entries resulting in 12 bits code values (we refrain from using variably sized codes). This means that 2 output codes actually produce 3 characters. (The 4 most-significant bits of each code are packed into a single character, placed between the two remaining 8-bit least-significant parts.) For example, the encoded representation of the example source text (in hexadecimal form)

6D 69 73 73 101 103 70 70 69

results in the following output sequence.

6D 00 69 73 00 73 01 11 03 70 00 70 69

Observe that we have used the 8 bit ASCII value to encode characters, and hence, the first new table entry will receive 256 (100 hexadecimal) as code value.

When encoding a large source file this may lead to an overflow of the *LZW* table. In that case one can either decide to stop adding new entries to the table or to clear (parts of) of it. In section ... we will discuss both possibilities, starting with the former one.

LZW compression in Clean

For representing the *LZW* table we adopt the tree-like data structures from ???, i.e.

```
LZWTable = Node Char Int LZWTable LZWTable LZWTable | Leaf
```

Each element of the lzw table (corresponding to some string s) consists of the a code value, the last character of s (say c) , and three subtrees: one tree containing all extensions of s and two subtrees containing extensions of s without c starting with a character that is less than, respectively, greater than c. Returning to our example text, the new parts of the lzw table, after processing the input, look as follows.

======== plaatje =========

Note that there is some freedom in the way the initial table (containing the 256 singleton strings) is built. For the best performance, however, it is necessary that it is well balanced.

Exercise. Define a functional initial_lzw_table return a well balanced initial table.

The main function of our encoding algorithm, called lzw_encode, takes the source file (represented as a list of characters), the *LZW* table and a counter indicating the next free code value as input. It yields the resulting codes as a list of integers, by repeatedly performing encode_string to the input data.

```
lzw_encode :: [Char] LZWTable Int -> [Int]
lzw_encode [] code_table next_new_code
    = []
lzw_encode input code_table next_new_code
    = [next_code : lzw_encode rest_input new_code_table (inc next_new_code)]
where
    (next_code, rest_input, new_code_table)
        = encode_string input 0 code_table next_new_code
```

The function encode_string traverses the *LZW* tree using the input characters as direction indicators. As soon as it hits on a leaf a new entry is created in the *LZW* tree. The result of encode_string is the code value of the longest matching substring, the remainder of the input and the extended *LZW* tree.

```
encode_string :: [Char] Int LZWTable Int -> (Int, [Char], LZWTable)
encode_string [] prev_code code_table new_code
    = (prev_code, [], Leaf)
encode_string [next_char : rest_input] prev_code Leaf new_code
    | new_code < MaxEntries
        = (prev_code, [next_char : rest_input],
                Node next_char new_code Leaf Leaf Leaf)
        = (prev_code, [next_char : rest_input], Leaf)
encode_string [next_char : rest_input] prev_code
        (Node this_char this_code ext left right) new_code
    | next_char < this_char
        = (code_left, input_left,
                Node this_char this_code ext tab_left right)
        with
        (code_left, input_left, tab_left)
            = encode_string [next_char : rest_input] prev_code left new_code
    | next_char > this_char
        = (code_right, input_right,
                Node this_char this_code ext left tab_right)
        with
        (code_right, input_right, tab_right)
            = encode_string [next_char : rest_input] prev_code right new_code
    | otherwise
        = (code_ext, input_ext, Node this_char this_code tab_ext left right)
        with
        (code_ext, input_ext, tab_ext)
            = encode_string rest_input this_code ext new_code
```

Finally, all produced value are written to a file by calling the function list_to_file. Since every output code is split into two parts (one 8 bit, and one 4 bit part) list_to_file is parameterized with the (still to be written) remainder of the previous write operation. To indicate that the remainder is empty, it is set to -1.

```
list_to_file :: [Int] Int *File -> *File
list_to_file [] rest_code file
    | rest_code < 0
        = file
        = fwritec (toChar rest_code) file
list_to_file [next_code : next_codes] rest_code file
    | rest_code < 0
        = list_to_file next_codes ((next_code >> 4) bitand 0xf0)
            (fwritec (toChar next_code) file)
        = list_to_file next_codes (-1) (fwritec (toChar next_code)
            (fwritec (toChar comb_code) file))
        with
            comb_code = rest_code bitor (next_code >> 8)
```

Decompression

A key feature of *LZW* encoding is that the resulting *LZW* table is not needed when the encoded text is decompressed. The correspondence between code values and strings can be determined solely by examining the list of encoded output values of the compression algorithm. Consider, for example the situation as it occurs at the beginning of the input

text (of encoded data). The first value will always be a character (for, the initial *LZW* table contained only characters). One also knows that this character forms the prefix of the string belonging to the first code value (i.e. 256). The next input value (which is either a character or the code 256) forms the string of code value 257, except that again the last character is still unknown. However, the missing last character of the previous code, 256, is known right now: it is the first character of the string of code 257. In the same way, the strings of all subseqent codes can be determined.

Based on the above observation, the decoding algorithm can be specified in a few lines of Clean code. The function decode takes a list of code values as input and yields the corresponding decoded data. The translation of code value into strings is done with the aid of a string table: a strict array of strings (i.e. {! [Char]}) in which the i-th element corresponds to the string with code value i. The construction of this string table occurs on-the-fly. For this reason, decode is supplied with some additional arguments: one argument (of type [Char]) is the output of decode itself (indeed, this leads to a cycle!) of which an appropriate prefix is taken and put into the string table (the fourth argument of decode) at position next_code (the third argument).

```
decode :: [Int] [Char] !Int  !*{! [Char]} -> [Char]
decode [] output next_code strings
    = []
decode [lzw_code : codes] output next_code strings
    | next_code == MaxEntries
        = string_of_code ++ decode codes [] next_code sel_strings
    | otherwise
        = string_of_code ++ decode codes (drop size_of_code output)
                    (inc next_code) new_strings
    with
        new_strings = { sel_strings & [next_code] =
                                take (inc size_of_code) output }
        size_of_code = length string_of_code
where
    (string_of_code, sel_strings) = uselect strings lzw_code
```

Decoding an encoded text can be done by calling lzw_decode which, applies decode to the proper initial values.

```
lzw_decode :: File *File -> *File
lzw_decode in_file out_file
    = list_to_file decoded_file out_file
where
    decoded_file = decode (read_code (-1) f_in)
                        decoded_file FirstLZWCode initial_strings
```

Here read_code is essentially the inverse of the function list_to_file presented in the previous section, and initial_strings creates the initial string table in which the first 256 elements are set to the singleton strings.

Performance results

Before we try to improve our encoding program we will do some measurements just to be sure that these improvements are necessary indeed. In general, the lack of reference material often makes it hard to determine the (relative) efficiency of functional programs. In our case, however, is was not difficult to write a straightforward C version of encoding algorithm that uses the same data structure as in the Clean program. We have compared this C program (of which the listing can be found in the appendix) with the encoding algorithm written in Clean not only to see how fast the latter one is but also to determine a limit for our optimisations, for, we do not expect to obtain a program that will be (much) faster the former one. The performance figures shown in the table below are obtained by running each program on two different source files: paper1 (53Kb troff source) and book1 (769 Kb plain ASCII file). These files were taken from the Calgary Text Compression Corpus. They are intended to serve as benchmark files for testing compression methods.

File	C	Clean
paper1	0.15	1.4/1.9
book1	1.3	18/29

CPU times are specified in seconds; each CPU time of the Clean program is divided into execution time and garbage collection time (ET/GC).

These figures clearly show that the Clean program is performing poorly, indicating that it not (yet) really suited for being used in practice. As a first step, we will propose e few minor optimisations and measure their effects on the performance. In all cases the book1 file is taken as input data for the algorithm.

1. Adding strictness annotations to a) function arguments and b) also to the tree data structure. Obviously, it is harmless to make all function of the compression programs strict all of their arguments. Many of these strict arguments were not found by the strictness analyser.
2. Splitting the function encode_string into two functions encode_string and encode_char_and_substring. In comparison with encode_string the latter functions gets two additional arguments, namely the next input character and the rest of the input. This avoids the rebuilding of and the pattern matching on the input whenever the left or the right branch of the *LZW* tree is chosen.
3. Reading and writing occurs directly on files instead of using intermediate lists.
4. The first 256 entries (corresponding to the singleton strings) are not stored in the *LZW* tree. Instead we use a hash table of size 256 in which the first character of the input data serves as a hash value. The entries of the hash table are *LZW* trees, thus, the entry at character c corresponds to all strings starting with c. The Clean he type of hash table is:
```
HashTable :== { ! LZWTable }
```
5. Combining 1 to 4.
6. Using a separate look up function once the *LZW* table is filled. Note that from that moment the table does not change anymore. The function encode_string, however, keeps on updating it, which produces lots of (unnecessary) garbage. This optimisation is done in combination with 5.

Optimisation	1a	1b	2	3	4	5	6
ET/GCT	9/12	8/8	18/28	7/9	11/18	4/4	2.6/0

These figures are already quite satisfactory: the final solution is only 2 times slower than the C version.

The next step is to define a different, more compact representation of the *LZW* table by using arrays. In Clean one can define arrays of unboxed values (see also ...). An unboxed value is either a strict basic value or a record of unboxed values. For example, records of the following CodeElem type are unboxed


```
::  CodeElem = {    char    :: !Char,
                    left    :: !Int,
                    right   :: !Int,
                    extend  :: !Int
               }
```

We now define the *LZW* table as an array of unboxed CodeElems, i.e.

```
LZWTable :== {# CodeElem }
```

In this representation, the code value entry has become superfluous: it is equal to the index of the corresponding entry in the *LZW* table. By making the table unique, all updates can be done in place. To illustrate the effect of changing our basic data structure on the encoding algorithm we give the modified version of the function encode_char_and_substring; all other functions are adjusted in straightforward way.

```
NULL    :== (-1)

encode_char_and_substring :: !Char !File !File !Int
    !Int !*CodeTable !Int -> (!Int, !File, !Int, !*CodeTable)
encode_char_and_substring next_char rest_input input prev_code
    table_code table new_code
    | table_code == NULL
        =  (prev_code, input, new_code, { table & [new_code] = new_code_elem } )
        with
        new_code_elem = { char = next_char,left = NULL,
                          right = NULL,extend = NULL }
    | next_char < this_char
        = (code_left, file_left, table_code,
            { tab_left & [table_code] = { sel_code_elem & left = index_left }} )
        with
        (code_left, file_left, index_left, tab_left) =
            encode_char_and_substring next_char rest_input input
                        prev_code sel_code_elem.left sel_table new_code
    | next_char > this_char
        = (code_right, file_right, table_code,
        { tab_right & [table_code] = { sel_code_elem & right = index_right }} )
        with
        (code_right, file_right, index_right, tab_right)
            = encode_char_and_substring next_char rest_input input
                        prev_code sel_code_elem.right sel_table new_code
    | otherwise
        = (code_ext, file_ext, table_code,
            { tab_ext & [table_code] = { sel_code_elem & extend = index_ext }} )
        with
        (code_ext, file_ext, index_ext, tab_ext)
            = encode_substring rest_input table_code
                        sel_code_elem.extend sel_table new_code
where
    (sel_code_elem, sel_table) = uselect table table_code
    this_char = sel_code_elem.char
```

Observe that the leafs of the *LZW* tree are indicated by setting the index to NULL.

Running the resulting program on the test input takes 2.1 seconds (no garbage collection time).

Decompression

Just like our initial encoding algorithm, the decoding algorithm turns out to be very inefficient. For instance when applied to the encoded book1 file a heap of 10-11 Mb is needed for decompression. The main reason for this tremendous space leak is that the constructed string table might contain unevaluated entries. Remember that each entry is determined by taking a certain part of the output file (i.e. the decoded file being produced).

If the table contains entries that are never used (or not used for a long time) the corresponding entry contains a reference to the front part of output list. At worst, the whole output list is kept in the heap. Knowing that a list of characters takes up 12 bytes per element (the characters themselves are shared, hence only heap space for the list cells is needed), the decoded input file will occupy 9 MB of heap space. Adding this amount to the storage required by the rest of the string table explains the huge memory consumption. The chosen representation has another disadvantage: all strings are stored separately; even strings that are prefixes of others are copied. This becomes particularly annoying if the original source file contained al lot of repetitions, which is, for instance, the case with the 8Mb data file called aaaa used in our tests later on in the chapter. It appears that during the decoding space occupied by he string table is approximately 84 Mb (!).

Our next solution avoids both space leaks by using the input itself (i.e. the encoded file) instead of the produced output file to reconstruct the original source file. The decompression table is defined as follows.

```
:: DecTableElem =   {   first_char  :: !Char,
                        elem_code   :: !Int
                    }

:: DecTable     :== {# DecTableElem }
```

This table associates to each code value, say *cv*, its prefix code (which is either a character or a code value, but always stored as an integer) and a character first_char denoting the first character of *cv*'s source string. The latter is mainly used as optimisation for the decoding algorithm. The key routine is given by the function decode_code which, when applied to a code value, a code table and an output file, writes the string corresponding to that code value to the output file.

```
decode_code :: !Int  !DecTable !*File -> !*File
decode_code lzw_code code_table file
    | lzw_code < FirstLZWCode
        = fwritec (toChar lzw_code) file
    | otherwise
        = fwritec code_table.[inc lzw_code].first_char
            (decode_code code_table.[lzw_code].elem_code code_table file)
```

Here the purpose of the first_char field becomes clear: the last character of *cv*'s string (being the first character of *cv*+1's string) is now directly available.

The other functions are defined straightforwardly. The only subtlety is the fact that the size code_table has to be MaxEntries + 1, for, decode_code uses for decoding code value MaxEntries the first character of entry MaxEntries + 1.

It should be clear that this decompression algorithm will run in constant heap space. To give you an impression of its efficiency: it only takes 1.5 seconds to decompress the encoded book1 file.

Compression results

Until now we were only interested in the efficiency of the encoding algorithm. The main question of this section will be: How well does our *LZW* encoding algorithm compress? An extensive comparison with other, popular methods is beyond the scope of this book, however, to give the reader an impression, we have compared our algorithm with the result of the standard Unix compress. It should be noted that the latter is a variant of *LZW* encoding, and that other compression methods (gzip, for instance) using different techniques give much better results.

The performance figures presented in the table below have been obtained by running both programs on five different files: paper1, book1, geo (102 Kb non-ASCII data file), cleanps (1.1 Mb PostScript file) and aaaa (8.4 Mb text file, consisting of a's only). The geo file was also taken from the Calgary set, the other two files were generated by ourselves.

file	paper1	book1	geo	cleanps	aaaa
compress	25	332	78	503	6
Clean	31.2	391	79	1000	6

All sizes are in Kb's

The results are quite disappointing, especially the compression of the PostScript file. In the remainder of the chapter we will discuss some improvements, examine the effort it takes to implement them and show their influence on the compression behaviour of th resulting program.

Improvements

The fact that, once the *LZW* table has been filled, it is fixed makes the compression algorithm quite rigid: it is not able anymore to anticipate on changes in the input. This becomes apparent when decoding the postscript file. This file starts with a (relatively large) header which is stored in the *LZW* table. This information, however, is rarely used in the rest of the file, which means that the largest part of the input data is redirected unchanged to the output file. An improvement of the algorithm is to throw away parts of the *LZW* table once it gets exhausted. We will use a simple criterion to decide whether an entry is removed, namely, entries whereof the code value were not used are freed; the others are maintained. Detection of such unused entries is easy: their extend field is NULL. However, the is a subtlety that has to be dealt with. It might be the case that an empty entry forms the connection between two non-empty entries in the table. This is illustrated inn the following picture.

Simply removing the empty entry would make it's successors unreachable. For this reason we introduce an extra phase during table clean up that removes all intermediate empty nodes from the *LZW* trees. The main function of this phase, called remove_empty_nodes, is given below.

```
remove_empty_nodes:: !Int  !*LZWTable -> (!Int, !*CodeTable)
remove_empty_nodestree_index table
    | tree_index == NULL
        = (NULL, table)
    | this_extend <> NULL
        = (tree_index, { extend_table & [tree_index]
            = { node & left = left_root, right = right_root,
                extend = extend_root }})
    | left_root == NULL
        = (right_root, extend_table)
    | right_root == NULL
        = (left_root, extend_table)
    | otherwise
        = (left_root, insert_in_right_branch right_root left_root extend_table)
where
    (node, node_table) = uselect table tree_index
```

```
    this_extend = node.extend

    (left_root, left_table)     = remove_empty_nodes node.left node_table
    (right_root, right_table)   = remove_empty_nodes node.right left_table
    (extend_root, extend_table) = remove_empty_nodes this_extend right_table

    insert_in_right_branch :: !Int !Int !*LZWTable -> !*CodeTable
    insert_in_right_branch new_right tree code_table
        | right_root == NULL
            = { node_table & [tree] = { node & right = new_right }}
        | otherwise
            = insert_in_right_branch new_right right_root node_table
    where
        (node, node_table) = uselect code_table tree
        right_root = node.right
```

Observe that the case in which an empty nodes contains non-empty left and right branches is treated special: to avoid that one these branches gets lost, the right tree is inserted in the (right part of) the left tree.

Compacting of the table is done on two phases. During the first phase, an array of forwarding pointers is constructed (by calling set_forwading_pointers) indicating the new location of each non-empty entry of the table. During the second phase, the table is tamped down by, by calling compact_table, which moves all entries forward.

```
set_forwading_pointers :: !Int !*LZWTable !Int !*{# Int}
    -> (!*LZWTable, !Int, !*{# Int})
set_forwading_pointers elem_index code_table first_free forward
    | elem_index == MaxEntries
        = (code_table, first_free, forward)
    | elem.extend == NULL
        = set_forwading_pointers (inc elem_index) sel_table first_free
                { forward & [elem_index] = NULL }
    | elem_index > first_free
        = set_forwading_pointers (inc elem_index) sel_table new_next_free
                new_forward
        with
        (new_next_free, new_forward)
            = find_next_free (inc first_free) update_forward
        update_forward = { forward & [elem_index] = first_free }
    | otherwise
        =  set_forwading_pointers next_index sel_table next_index
                    { forward & [elem_index] = elem_index }
        with
            next_index =  inc elem_index

where
    (elem, sel_table) = uselect code_table elem_index

    find_next_free :: !Int !*{# Int} -> (!Int, !*{# Int})
    find_next_free free_cand forward
        | forw == 0 || forw > free_cand
            = find_next_free (inc free_cand) sel_table
        | otherwise
            = (free_cand, sel_table)
    where
        (forw, sel_table) = uselect forward free_cand

compact_table :: !Int !*LZWTable !*{# Int} -> (!*LZWTable, !*{# Int})
compact_table next_elem table forward
    | next_elem == MaxEntries
        = (table, forward)
    | forw == NULL
        = compact_table (inc next_elem) table sel_forward
    | otherwise
        = compact_table (inc next_elem)
```



```
            {sel_table & [forw] =
                    {   elem & left = new_left, right = new_right,
                        extend = new_ext , previous = new_previous }}
            forward_prev
        with
        (elem, sel_table)           = uselect table next_elem

        (new_left,forward_left)     = new_index elem.left sel_forward
        (new_right,forward_right)   = new_index elem.right forward_left
        (new_ext,forward_ext)       = new_index elem.extend forward_right
        (new_previous,forward_prev) = new_index elem.previous forward_ext
where
    (forw, sel_forward) = uselect forward next_elem

    new_index :: !Int !*{# Int} -> (!Int, !*{# Int})
    new_index old_index forward
        | old_index == NULL
            = (NULL, forward)
        | otherwise
            = (forw, sel_forward)
    where
        (forw, sel_forward) = uselect forward old_index
```

To save memory, the array of forwarding references is allocated only once at the program start. By offering it as an unique object, it be changed destructively, and reused each time the table is reorganised. The effects of this improvement on both the efficiency and the compression results is shown in the following table. We have also included the decompression times. The adjustment of the decoding algorithm such that it can cope with a changing *LZW* table is left as an exercise.

file	paper1	book1	geo	cleanps	aaa
size (Kb.)	26.5	378	78.5	543	6
compr time (sec.)	0.41	3.6	0.70	5.6	27
decompr time (sec.)	0.30	3.6	0.60	4.5	13.3

Further improvements

The compression algorithm can be improved further by increasing the maximum table size. This implies the code value sizes increase which will, of course, annul part of the gain obtained by using a larger table. However, as suggested in the introduction, one can use variably sized code values. A disadvantage of these value is that reading and writing if these becomes somewhat more involved.

Exercise Define a function write_code to write variably sized codes to an output file.

By taking a table size of 2^{15}, and by using variably sized code values it appears that one gets approximately the same compression results as with Unix compress.

Exercise Verify this statement by adjusting the encoding algorithm accordingly, and by running it on the example data.

Conclusion

We hebben weliswaar geen algemene methode gegeven voor het verbeteren van de efficientie van functionele programma's, echter wel aangetoond dat, uitgaande van een eenvoudige eerste versie middels relatief kleine stappen een, in eerste instantie erg inefficient programma, kan worden omgezet naar een programma met een acceptabele efficientie.

bla bla bla