Seif Haridi

Assignment 3, CS2104: Compression

Examination date: Third Assignment
currently scheduled 2003-10-06

Problem Description

This lab will develop the Huffman compression algorithm, invented by David A. Huffman in 1952.

Compression algorithms transform a *raw* representation of data into a *compressed* representation of data, where the compressed representation requires less space. Smaller data can be communicated faster and more of it fits on disks.

The Huffman algorithm is a *lossless* compression method: the raw data can *exactly* be reconstructed from the compressed data. This is in contrast to *lossy* compression methods for pictures (such as JPEG), video (such as MPEG), or sound (such as MP3). In these domains some loss in quality due to compression is acceptable: the hope is that the loss will be unnoticed by human perception.

Even though Huffman is a lossless algorithm, it is an important component in many other compression methods (even in lossy methods) such as in JPEG and MPEG files and in everyday fax machines.

Lab Purpose. The main purpose of this lab is to familiarize you with the concept of *abstract data types*. You will learn how to separate *interface* and *implementation* of a data type and to organize abstract data types into modules.

You will be forced not to break the abstraction barrier of the abstract data type as we offer routines that allow you to check whether your implementation is correct and also routines to compress and decompress files. These software components insist on the interface of the abstract data type.

Another purpose is to explore programming techniques for trees and lists (recursion, accumulators, . . .).

You need to understand the following sections of Chapter 3: Section 3.4.6 on trees, Section 3.6 on higher order Programming, Section 3.7 on abstract data types, in particular understand the concept of secure abstract data types in Section 3.7.5, and Section 3.9.3 on software components.

Huffman Compression

The Huffman algorithm compresses data based on frequency: replace frequently occurring data by shorter sequences of bits and less frequent data by relatively

longer sequences of bits. Bit sequences are assigned to data by a binary tree called *Huffman tree*.

Compressing Bytes. In the following we assume that the raw data is a sequence of bytes, that is integers between `0` and `255`. Typically, bytes are encoded by bit sequences that are represented by a *fixed-size* sequence of exactly eight bits. For example, the bit sequence `00100101` represents the byte `37`:

$$\begin{aligned} &0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &\quad = 2^5 + 2^2 + 2^1 = 32 + 4 + 1 = 37 \end{aligned}$$

Variable-Size Encoding. Huffman compression uses a *variable-size* encoding for bytes. Frequently used bytes are assigned shorter bit sequences while less frequently used bytes are assigned longer bit sequences.

Assume that byte `37` has higher frequency than byte `129` (that is, it occurs more often in the raw byte sequence). If we choose byte `37` to be encoded by the bit sequence `0` and byte `129` by `10`, we can represent the sequence of bytes

```
[37 129 129 129 37]
```

by the bit sequence

`0 10 10 10 0`

instead of the uncompressed bit sequence

`00100101 10000001 10000001 10000001 00100101`

Here, we can compress 40 bits (five bytes with eight bits each) to 8 bits!

Prefix Property. We must be able to reconstruct the raw data from the compressed data. The bit sequences for the different bytes have to be chosen such that they are unambiguous.

For example, if we would have chosen bit sequences `0` for `37` and `00` for `129`, the bit sequence `000` could be decompressed either to `[37 37 37]`, `[37 129]`, or `[129 37]`. The problem with this choice of bit sequences is that `0` is a prefix of `00`. A bit sequence b_1 is a prefix of a bit sequence b_2, if b_2 starts with all the bits of b_1.

The Huffman algorithm constructs bit sequences with two properties:

- Unambiguous: a bit sequence for byte b is never a prefix of a bit sequence for byte b' where $b \neq b'$.
- Shortest: the bit sequence of a byte b will be shorter than, or a most equal to, that of any byte b' with lower frequency.

You already know telephone numbers as a particular numbering system which is also unambiguous as it comes to prefixes of numbers.

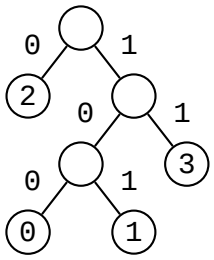

Figure 1: A Huffman Tree.

Structure of Huffman Compression. To compress a byte sequence into a bit sequence, the Huffman algorithm performs the following steps:

1. Compute the frequency for each byte.

   We will use the following byte sequence (represented as list) as an example:

   ```
   [1 3 2 0 2 3 2 2 3 1 2 3]
   ```

   The following *frequency map* maps each byte to its frequency

Byte	Frequency
0	1
1	2
2	5
3	4

2. Based on the frequencies, construct a Huffman tree.

   Figure 1 shows a Huffman tree for our example. To find the bit sequence for a byte, start at the root node and find your way to the byte. Each time you choose a left branch, add a `0` to the bit sequence. Each time you move right, add a `1` to the bit sequence.

   The byte `0` has the compressed bit sequence `100`: starting from the root, go right, go left, and again go left and reach the leaf with byte `0`.

   The Huffman tree has the property that more frequent bytes are closer to the root than less frequent bytes. This results in shorter bit sequences for more frequent bytes.

3. Compression: replace each byte by the previously computed bit sequence. This is done by computing for each byte the bit sequence from the Huffman tree and then use this information to compress the input byte sequence.

   In our example, this leads to a *bitmap* as follows:

Byte	Bit Sequence
0	`100`
1	`101`
2	`0`
3	`11`

All in all, the compressed bit sequence is

```
10111010001100111010011
```

4. For decompression: replace the bit sequence by the bytes according to the Huffman tree.

1 Computing Frequency Maps

A frequency map `M` is implemented as a record with label `map` where its features are the bytes occurring in the input and the fields are the bytes' frequencies.

Develop a function `{FrequencyMap Xs}` which takes a list of bytes `Xs` as input and returns a frequency map for it.

For example, `{FrequencyMap [1 2 1 4]}` returns `map(1:2 2:1 4:1)`.

Implement a function `{Increment Xs M}` that takes a list of bytes `Xs` and a frequency map `M`. It recursively increments the frequency count for each byte in `Xs`. Initially, `Increment` is applied to the entire byte list and the empty map `map`.

One important step in computing an incremented frequency map is testing whether a byte has already been processed (so the new frequency must be the old frequency plus one) or not (then the new frequency is one). This can be done by using `AdjoinAt` together with `{CondSelect R F X}` which returns `R.F`, if the record `R` has the feature `F`, and `X` otherwise.

Store both functions in a file called `FrequencyMap.oz`

2 Computing Huffman Trees

The Huffman tree consists of nodes. A node can be either a leaf node or an inner node. A leaf node contains a byte and a frequency. An inner node contains a frequency and a left and a right subtree.

The Algorithm. The algorithm to construct the tree works in a bottom-up fashion on lists of trees. Initially, the algorithm starts with only leafs, where each leaf represents a distinct byte from the input byte sequence together with its frequency. The algorithm terminates with just one tree left which is the wanted Huffman tree.

The algorithm is as follows:

1. Compute the initial list of trees `Ts` from the frequency map.

2. Repeat recursively until only one tree is left in `Ts`, then return this tree.

   (a) Choose and remove the two trees `A` and `B` with lowest frequency (in the first step these will be two leaves, but as the algorithm proceeds also trees with multiple nodes are considered) from the list `Ts`.

(b) Add a new tree `AB` that has as its frequency the sum of frequencies of `A` and `B` and as subtrees `A` and `B` to the list of trees.

This is the step that makes the trees constructed by the algorithm satisfy the properties mentioned above: bytes with lower frequency are placed at the bottom of the tree. This is not only taken into consideration for individual bytes but also for subtrees representing an entire set of bytes.

The Tree Datatype. The first step is to define the abstract datatype for Huffman trees. Give implementations for the following interface:

- `{MakeNode F L R}` returns an inner node with frequency `F`, left subtree `L` and right subtree `R`.
- `{MakeLeaf B F}` returns a leaf node with byte `B` and frequency `F`.
- `{IsLeaf H}` tests whether the Huffman tree `H` is a leaf node.
- `{IsNode H}` tests whether the Huffman tree `H` is an internal node.
- `{Frequency H}` returns the frequency of `H`.
- `{Byte H}` returns the byte of a leaf node `H`.
- `{Left H}` returns left subtree of inner node `H`.
- `{Right H}` returns right subtree of inner node `H`.

Put all functions implementing the abstract datatype in a single file `Tree.oz`.

Computing the Initial Leaves. The initial list of leaves is computed from the frequency map.

Write a procedure `{InitLeaves M}` that returns a list of leaves, where each element corresponds to an entry in the frequency map `M`. Remember that you have to use the function `MakeLeaf` to construct a Huffman tree consisting of a single leaf only!

The function `{Record.toListInd R}` will come in handy here. It returns a list of feature-field pairs of the record `R`. For example,

```
{Record.toListInd map(1:2 2:1 4:1)}
```

returns `[1#2 2#1 4#1]`.

Constructing the Tree. Program a function `{Huffman M}` that returns a Huffman tree for the frequency map `M` according to the above algorithm.

You need an additional function `{FindLowest Hs}` that returns a pair `H#Hr` where `H` is the Huffman tree with lowest frequency in `Hs` and `Hr` contains the other elements of `Hs`.

You can either write a function `FindLowest` yourself or use the function given in Appendix A which is also available from the course website.

Put all functions in a file `Huffman.oz`.

3 Compression

In order to make compression efficient, we first compute the *bitmap* described by the Huffman tree.

Computing Bitmaps. Develop a function `{Bitmap H}` that takes a Huffman tree (as defined by the abstract datatype) as input and returns a bitmap. The bitmap is implemented as a record with label `bits` that maps each byte in the tree `H` to its bit sequence represented as a list containing `0` and `1` as elements.

Use a similar technique as for computing frequency maps. Bitmaps are simpler as every byte occurs at most once in a Huffman tree.

The main difference to construction of the frequency map is that a Huffman tree is traversed recursively and simultaneously the appropriate bit sequences are computed. Define a function `{Traverse H Bs M}` where `H` is the Huffman tree to be traversed, `Bs` is the current sequence of bits, and `M` is the current map from bytes to bit sequences. Whenever `Traverse` recurses into a left subtree, a `0` is appended to the end of the bit sequence `Bs`. Similarly, a `1` is appended for the right subtree. If a leaf is reached, `Bs` yields the bit sequence for the leaf's byte.

Compression Proper. Now write the complete compressor function: `{HUF Xs}` returns a pair of Huffman tree for `Xs` (we will need the tree for decompression later) and the compressed bit sequence. The steps that `HUF` takes are as follows:

- Compute the frequency map `M`.
- Compute the Huffman tree `H` according to `M`.
- Compute the bitmap `B` according to `H`.
- Compute the bit sequence according to `Xs` and `B`.

Put both `Bitmap` and `HUF` into one file `HUF.oz`.

4 Decompression

In order to decompress a bit sequence, we also need the Huffman tree. Given a pair `HBs` of a Huffman tree `H` and a bit string `Bs` (as returned by `HUF`), the function `{UNHUF HBs}` returns the original uncompressed byte sequence.

`UNHUF` is simple: according to whether the next bit in the sequence is `0` or `1`, the left or right branch in the Huffman tree is chosen. If traversal reaches a leaf, we

have found the appropriate byte and recursion starts again from the full Huffman tree.

This can be best achieved by a nested function `Uncompress` which has access to the full Huffman tree `FullH` by an external reference:

```
fun {UNHUF HBs}
   case HBs of FullH#Bs then
      fun {Uncompress Bs H}
         ...
      end
   in
      {Uncompress Bs FullH}
   end
end
```

Put the function `UNHUF` into a file `UNHUF.oz`

Voluntary Trick Question. A more space-efficient way to store compressed information is to store the frequency map instead of the Huffman tree together with the bit sequence. Do you know why this is not sufficient?

5 Putting Everything Together

We offer functionality with which you can apply and test your Huffman compressor. These programs need your implementation of the abstract data type. They will use the Huffman tree abstract data type according to its interface.

You have to construct a *module* that contains the implementation of the abstract data type. A module is a record where the features give the interface names and the fields are the implementations (functions in our case). Construct the module as follows:

```
TreeModule = tree(makeNode:  MakeNode
                  makeLeaf:  MakeLeaf
                  isLeaf:    IsLeaf
                  frequency: Frequency
                  byte:      Byte
                  left:      Left
                  right:     Right)
```

Then download the file `MakeTest.ozf` from the course webpage and load the function `MakeTest` as follows:

```
MakeTest={Module.link [´MakeTest.ozf´]}.1.make
```

Call the function `MakeTest` with your `TreeModule` as argument and get a module `Test`:

```
Test = {MakeTest TreeModule}
```

The module `Test` offers the following functions and procedures:

- `Test.huf` and `Test.unhuf` are the functions `HUF` and `UNHUF` developed in Sections 3 and 4.
- `{Test.compress HUF InFile OutFile}` compresses the file with filename `InFile` (an atom) to a file with filename `OutFile` (an atom) using your compressor `HUF` (the function developed in Section 3).
- `{Test.decompress UNHUF InFile OutFile}` decompresses `InFile` to `OutFile` using your decompressor `UNHUF` (the function developed in Section 4).

A The Function `FindLowest`

```
local
   fun {Scan Hs I Ir}
      %% Hs is the list of Huffman trees
      %% I is the tree with so-far lowest frequency
      %% Ir are all trees already scanned and known
      %%    to have higher frequency than I
      case Hs
      of nil then I#Ir
      [] H|Hr then
         if {Frequency H}<{Frequency I} then
            %% Lower frequency tree!
            {Scan Hr H I|Ir}
         else
            {Scan Hr I H|Ir}
         end
      end
   end
in
   fun {FindLowest Hs}
      {Scan Hs.2 Hs.1 nil}
   end
end
```