Discrete Mathematics and Functional Programming

Revised Section 6.12. Extended example: Huffman encoding

Thomas VanDrunen

July 28, 2015

6.12 Extended example: Huffman encoding

The information we store and use with computers each day—whether text, images, audio, or numerical data—must be decomposed into a form suitable for electronic media. It must be *digitized*. It is no surprise, then, that much research has gone into efficient binary encoding of different kinds of information. If we can store information with as few bits as possible, that same information will require less storage space and, probably more importantly, can be transmitted faster and with less bandwidth.

Text is a simple example. We represent text digitally by representing each letter, number, and punctuation symbol with a binary code. By *code* here we really mean *encoding scheme*, that is, an association between characters and bit sequences. Standard encoding schemes for text are ASCII, which uses 7-bit sequences for Latin characters, and Unicode, which uses 16-bit sequences and includes alphabets from around the world.

We will simplify things a bit. Suppose we wanted to transmit the string `RIFFRAFF`. Since this involves only four distinct characters—`A`, `F`, `I`, and `R`—we can make our own just-for-this-purpose encoding scheme that uses only two bits per character:

`A`	00
`F`	01
`I`	10
`R`	11

11	10	01	01	11	00	01	01
`R`	`I`	`F`	`F`	`R`	`A`	`F`	`F`

The entire string requires sixteen bits, which is how much we would need for a single character if we were using Unicode. We would like to find an encoding scheme for a given message that will encode that message with as few bits as possible.

Make sure you notice the difference between a message and an encoding scheme. We are working on optimizing the encoding scheme, measured by how small it makes the encoding of the message. To be useful, we presumably would need to transmit a representation of the encoding scheme itself, but we will ignore that for now. Also, by *encoding* we do not mean *encryption*. Our goal here is efficiency, not secrecy.

Our little encoding scheme is similar to ASCII and Unicode in that all assume that each character has the same number of bits. Let us now jettison that assumption; why not give characters that occur more frequently codes with *fewer* bits than rarer characters? Since `F` occurs four times and `R` twice, we can make a *variable-length* scheme that assigns them single-bit codes, while retaining two-bit codes for the other characters (still enough for a shave and a haircut).

A	01
F	0
I	10
R	1

1	10	0	0	1	01	0	0
R	I	F	F	R	A	F	F

The message now takes only ten bits—or so it seems. The problem is that the boundaries between the letters are not part of the information sent or stored. The bit sequence `1100010100` is ambiguous, alternately interpreted as

A	01
F	0
I	10
R	1

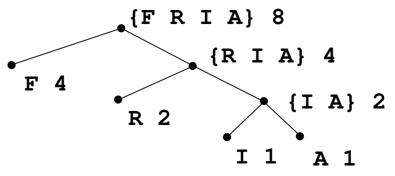
1	1	0	0	01	0	10	0
R	R	F	F	A	F	I	F

This problem is not inherent to variable-length codes, though. We can make an unambiguous variable-length code by making sure that no character has an encoding that is a *prefix* of another character's encoding, as in the code above `0` for `F` is a prefix of `01` for `A`. So, trying again, here is a prefix-free variable-length encoding scheme for this example:

A	111
F	0
I	110
R	10

10	110	0	0	10	111	0	0
R	I	F	F	R	A	F	F

The message now takes 14 bits—only a slight win over the fixed-length version of this example, but the benefits increase with larger messages containing more distinct characters. Our goal in this section to discover a way to build an optimal encoding scheme for a given message—and to encode and decode messages with such encoding schemes.



We move our attention from how we represent the encoded message to how we will represent the encoding itself. Instead of thinking of it as a table in which we can look up the code for a character or the character for a code, imagine the encoding scheme as a full binary tree. Each leaf will store or represent a character. The code for that character will be a description of the path from the root of the tree to the leaf holding that character. Consider the tree at the left, which represents the same encoding scheme as the most recent table. Let `0` indicate taking a the branch of a left child and `1` indicate the right. The route from the root to the leaf containing `I` is right, right, left, which matches the code for `I`, `110`.

The internal nodes contain strings indicating the characters found in the subtrees rooted there. So if we want to encode the character `R`, we start from the root and notice that the root's right child contains `R` in its string. We go right, and `1` is the first bit in the encoded form of the character. Next we notice that that node (`R I A`) has a left child that contains `R`, so we go left. Now we have reached the leaf for `R` and have found the code for `R` to be `10`.

Additionally, the leaves contain the frequency of their character's occurrences in the message and internal nodes contain the sum of the frequencies of the characters in the subtree. These frequencies are not used in encoding or decoding the message, but we will use them in building the tree itself.

Here is an ML datatype for Huffman trees.

```
- datatype huffTree =
=        Leaf of char * int
=      | Internal of huffTree * huffTree * string * int;
```

Recall that `char` is the ML type for values of a single character. A character literal, for example the character `A`, is written `#"A"`.

Now we will implement a system to build a tree based on letter frequencies in a text, encode a text using the tree, and decode the text, also using the tree. First, a few convenience functions for analyzing and synthesizing trees. We wish to be able to find the frequency of a node, whatever it is; similarly to find all the symbols at a node; to make a tree from two nodes; and to print the tree in a way somewhat readable.

```
- fun freq(Leaf(sym, f)) = f
=   | freq(Internal(left, right, sym, f)) = f;

val freq = fn : huffTree -> int

- fun symbols(Leaf(sym, f)) = str(sym)
=   | symbols(Internal(left, right, sym, f)) = sym;

val symbols = fn : huffTree -> string

- fun makeTree(left, right) =
=      Internal(left, right, symbols(left) ^ symbols(right),
=               freq(left) + freq(right));

val makeTree = fn : huffTree * huffTree -> huffTree

- fun printTree(Leaf(sym, f)) =
=         print("Leaf(\"" ^ str(sym) ^ "\", " ^
=                Int.toString(f) ^ ")")
=   | printTree(Internal(left, right, syms, f)) =
=         (print("makeTree("); printTree(left);
=          print(", "); printTree(right); print(")"));

val printTree = fn : huffTree -> unit
```

The above makes use of the standard ML function `str` which converts a char to a string of length 1.

Suppose we want to encode the text `ANNIKA ISAAC AND SILAS VANDRUNEN`. Spaces count as characters. We want to count up the frequency of the various letters and make a list of pairs containing a character and its frequency. We will need to iterate over the characters in the string and add them to such a list.

Suppose we have such a list, possibly empty, and want to account for another character. Either the character already appears in the list (in which case we would increment its frequency), or the character does not yet appear (in which case we would add it, with frequency 1). In building a function to do this, we break this into three cases: the list is empty, the list begins with the letter we are adding, and the list begins with a different character.

```
- fun addCharToFreqList(c, []) = [(c, 1)]
=   | addCharToFreqList(c, (x, f)::rest) =
=     if c = x then (x,f+1)::rest
=              else (x,f)::addCharToFreqList(c, rest);


Warning: calling polyEqual
val addCharToFreqList = fn
     : ''a * (''a * int) list -> (''a * int) list
```

The ML interpreter cannot determine that we mean a list of *characters*, but that is not a problem. Now we write two functions that will apply this to every

Tangent: Talking with the eyes

In Alexandre Dumas's novel *The Count of Monte Cristo*, a character named Noirtier de Villefort suffers a stroke and is left completely paralyzed, able to control only his eyes and eyelids. With these alone he develops a means of communicating his basic needs to his granddaughter, for example closing his eyes to say "yes" and blinking repeatedly to say "no."

More complicated thoughts could be expressed through these building blocks. At one point, when his granddaughter sensed he wanted something,

> . . . she began to recite all the letters of the alphabet, starting with *a*, until she got to *n*, smiling and watching the invalid's face; at *n*, Noirtier indicated: "Yes."
>
> "So!" Valentine said. "Whatever you want starts with *n*. And what do we want after *n*? Na, ne, ni, no. . . "
>
> "Yes, yes, yes," the old man said.
>
> Valentine went to fetch a dictionary, which she put on a reading stand in front of Noirtier. She opened it and when she saw that he was looking attentively at the pages, she ran her finger up and down the columns. At the word *notary*, Noirtier signalled her to stop.

After Valentine teaches the method of communication to the notary, Noirtier is able even to rewrite his will. The principle of systems like the Huffman encoding are not so different: as a series of yeses and noes directs the search of a dictionary, so here a series of bits describes a path in a tree.

letter in a list of characters and finally convert a message to a list of character-frequency pairs. Recall that `explode` takes a string and returns a list of the characters in the string.

```
- fun charListToFreqList([]) = []
=   | charListToFreqList(c::rest) =
=        addCharToFreqList(c, charListToFreqList(rest));

val charListToFreqList = fn : ''a list -> (''a * int) list

- fun msgToFreqList(msg) = charListToFreqList(explode(msg));

val msgToFreqList = fn : string -> (char * int) list

- val namePairs = msgToFreqList("ANNIKA ISAAC AND SILAS VANDRUNEN");

 val namePairs =
  [(#"N",6),(#"E",1),(#"U",1),(#"R",1),(#"D",2),(#"A",7),
   (#"V",1),(#" ",4),(#"S",3),(#"L",1),(#"I",3),(#"C",1),...]
        : (char * int) list
```

Once we have a list with accurate frequency counts for each character, we want to turn those character-frequency pairs into leaves of the tree. The function `makeLeafSet` converts a list of pairs into a list of leaves, sorting the leaves in increasing order of frequency as it goes. It makes use of the function `adjoinSet` which acts like the cons operator except that it sorts as it goes. Writing the function `adjoinSet` is left as a project.

```
- fun makeLeafSet([]) = []
=   | makeLeafSet((sym, f)::morePairs) =
=       adjoinSet(Leaf(sym, f), makeLeafSet(morePairs));

val makeLeafSet = fn : (char * int) list -> huffTree list

- val leafList = makeLeafSet(namePairs);

val leafList =
  [Leaf (#"K",1),Leaf (#"C",1),Leaf (#"L",1),Leaf (#"V",1),
   Leaf (#"R",1),Leaf (#"U",1),Leaf (#"E",1),Leaf (#"D",2),
   Leaf (#"I",3),Leaf (#"S",3),Leaf (#" ",4),Leaf (#"N",6),
   Leaf (#"A",7)]
       : huffTree list
```

```
[ K 1   C 1   L 1   V 1   R 1   U 1   E 1   D 2   I 3   S 3   _ 4   N 6   A 7 ]
```

Right now we have a big set of trivial trees. We will maintain this set as a set of trees, but at each step will will merge two of them together to make a bigger tree until we are left with a single tree, which will be our encoding.

A tree that meets our goal of minimizing the message encoding will have the most frequently used characters closest to the root and the least frequently used at the end of the longest sequence of links. Therefore our strategy is to merge smallest-frequency trees. Our list is sorted by frequency, so the first two items will have a frequency at least as small as all other items in the list. The function `oneMerge` takes a list of huffTree nodes and merges the first two into a new tree.

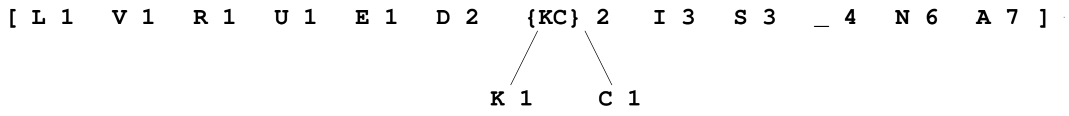
```
- fun oneMerge(x::y::rest) = adjoinSet(makeTree(x,y), rest);

Warning: match nonexhaustive
          x :: y :: rest => ...

val oneMerge = fn : huffTree list -> huffTree * huffTree list

- oneMerge(leafList);

val it =
  [Leaf (#"L",1),Leaf (#"V",1),Leaf (#"R",1),Leaf (#"U",1),
   Leaf (#"E",1),Leaf (#"D",2),Internal (Leaf (#"K",1),
   Leaf (#"C",1),"KC",2),Leaf (#"I",3),Leaf (#"S",3),
   Leaf (#" ",4),Leaf (#"N",6),Leaf (#"A",7)]
   : huffTree list
```


[L 1   V 1   R 1   U 1   E 1   D 2   {KC} 2   I 3   S 3   _ 4   N 6   A 7]
K 1
C 1

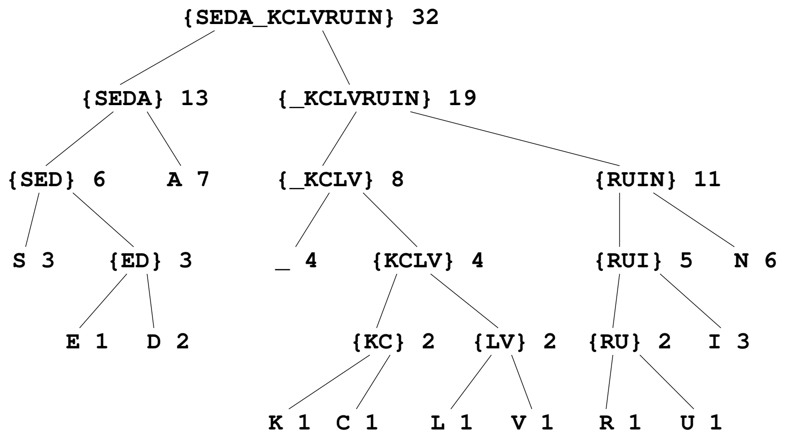
The function `successiveMerge` repeats this process until we have a single tree. It, too, is left as a project. We can also combine `successiveMerge` with `makeLeafSet` to form a function that will generate a tree given a list of character-frequency pairs:

```
- fun generateHuffTree(pairs) =
=           successiveMerge(makeLeafSet(pairs));

val generateHuffTree = fn : (char * int) list -> huffTree

- val nameTree = generateHuffTree(namePairs);
```

```
val nameTree =
    (Internal (Internal #,Leaf #,"SEDA",13),
     Internal (Internal #,Internal #," KCLVRUIN",19),
      "SEDA KCLVRUIN",32)
  : huffTree
```



Now we use this tree to encode the message. The function `encodeSymbol` takes a character and a Huffman tree and finds the code for that symbol by traversing the tree. It returns a list of bits (which we define below in a datatype), that serves as a record of the search and, thus, the encoding for that character. This function is left as a project. Once it is written, we can use it to encode an entire list of characters into a list of bit sequences. Recall that `explode` takes a string and returns a list of the characters in the string.

```
- datatype bit = One | Zero;

datatype bit = One | Zero

- fun encodeList([], tree) = []
=   | encodeList(fst::rest, tree) =
=        encodeSymbol(fst, tree)@encodeList(rest, tree);

val encodeList = fn : char list * huffTree -> bit list

- fun encode(msg, tree) = encodeList(explode(msg), tree);

val encode = fn : string * huffTree -> bit list

- val msg = encode("ANNIKA ISAAC AND SILAS VANDRUNEN",
=      nameTree);
```

```
val msg = [Zero,One,One,One,One,One,One,One,One,One,Zero,...]
: bit list
```

Notice our message is a simple list of bits, not a list of list of bits. We do not show the boundaries of the letters in our message.

Decoding the message, a final part of the project, uses the bit sequence to traverse the tree. When it hits a leaf, it adds a new character to the result and goes back to the top of the tree.

```
- decode(msg, nameTree);

val it = "ANNIKA ISAAC AND SILAS VANDRUNEN" : string
```

This section was adapted from Abelson and Sussman[1].

Project

6.A Finish the following function `adjoinSet`, which takes a tree and a list of trees and places the tree in the right place in the list to maintain that list as sorted by frequency.

```
fun adjoinSet(x, []) = [x]
  | adjoinSet(x, fst::rest) =
     if freq(x) < freq(fst)
     then ??
     else ??;
```

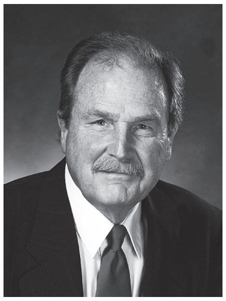
6.B Finish the function `successiveMerge`. It should make use of `oneMerge` to do each step of the merging, but it should also make use of your `adjoinSet` to maintain sortedness. In other words, you may assume in writing `successiveMerge` that any list your function receives is already sorted, but this means that any recursive call to `successiveMerge` must pass a list that is also sorted. Remember also that `adjoinSet` returns a list of pairs.

```
fun successiveMerge([x]) = x
  | successiveMerge(x::y::rest) = ??;
```

Biography: David Huffman, 1925–1999

David Huffman was a computer scientist who made important contributions to information theory. After serving in the Navy in World War II, Huffman attended graduate school at MIT. While there he took a certain course that gave students the option of writing a term paper to get out of the final exam. It was in that term paper that Huffman presented his most famous result, the encoding scheme that bears his name. Although widely used in industry, the encoding was never patented, leading some to observe that his “main compensation was dispensation from a final exam” [30].

Huffman taught at MIT and the University of California Santa Cruz, where he helped found the computer science department. His later work included research into paper folding and computational origami.



Project, continued

6.C Finish the function encodeSymbol. It uses the case and option ML language features described in Section 2.5.

```
fun encodeSymbol(sym, tree) =
 let
  fun encodeSymbol1(sy, Leaf(st, f)) =
                  ??
    | encodeSymbol1(sy,Internal(left,right,
                                 st,f)) =
      case (encodeSymbol1(sy, left),
            encodeSymbol1(sy, right)) of
         (NONE, NONE) => ??
       | (NONE, SOME bits) => ??
       | (SOME bits, x) => ??;
 in
   valOf(encodeSymbol1(sym,tree))
 end;
```

(This all assumes that every character in the message actually occurs somewhere in the tree. If not, encodeSymbol1 will return NONE, and the line valOf(encodeSymbol1(sym, tree)) will crash. Furthermore, the pattern (SOME bits, x) assumes x will always be NONE.)

6.D Finish the function decode. It makes use of a function called implode, which is a standard ML function and does the opposite of what explode does: it turns a list of characters into a string.

```
fun chooseBranch(Zero,
          Internal(left, right, st, f)) =
      left
  | chooseBranch(One,
          Internal(left, right, st, f)) =
      right;


fun decode(bits, tree) =
 let
  fun decode1([], currentBranch) = []
    | decode1(b::rest, currentBranch) =
       let val nextBranch = ?? in
         case nextBranch of
            Leaf(sym, w) => ??
          | Internal(left, right, syms, w)
             => ??
       end;
 in
  implode(decode1(bits, tree))
 end;
```