Haskell Demo Program: Huffman Coding

Josh Myer

February 24, 2008

This file is an example of literate Haskell programming. Along with this, there are *HUnit* test cases and a cabal file.

On to the code. First, the usual setup...

```
module Huffman where
import Control.Arrow
import List (sort, group, length, take)
import Data.Array (Array, array, index, (!))
import Data.Word (Word8)
import qualified Data.ByteString as B
import Data.Bits (testBit, shiftL)
```

This is the tree structure used in our program. There are the two intuitive cases (Leaf and Node), as well as a third NullTree case. This one is there as a sentinel value of sorts. It ensures that creating a tree always succeeds, even if the input list is empty.

If we're given an empty list, we can't know how to create a dummy "NULL" value Leaf node. Therefore, we must explicitly create our own NullTree.

Since Tree is polymorphic, we must know what sorts of values we are being created with, and what the "NULL" value is. If we're given an empty list of values to encode, we can't create either a Leaf (since it requires some a to hang off it) or a Node (since it requires two Trees, which requires creating four new Leafs).

```
data Tree a =
  NullTree |   -- empty tree type
  Leaf Int a |
  Node Int (Tree a) (Tree a)
  deriving (Show, Read)
```

Defining an ordering on trees makes the Huffman algorithm much easier to implement, so we do so here.

Note that we're missing one typical feature of Huffman coding: we don't assume that our Leaf values are ordered, so we can't order beyond weights. This makes the library more general, but causes extra run-time, since we have to be broader when searching for a given node.

```
treeNodeWeight :: Tree a → Int
treeNodeWeight NullTree = 0
treeNodeWeight (Leaf n x) = n
treeNodeWeight (Node n x y) = n
instance Eq (Tree a) where
  x ≡ y = (treeNodeWeight x)Prelude. ≡ (treeNodeWeight y)
instance Ord (Tree a) where
compare x y = Prelude.compare weight_x weight_y where
  weight_x = treeNodeWeight x
  weight_y = treeNodeWeight y
```

Now that we've defined our *Tree* type, we can move on creating the tree. The algorithm is fairly straightforward: given a sequence of tokens, count them and create *Leaf* nodes of each. In this list of *Tree*s, find the two smallest elements and replace them with a new *Node* containing them. Repeat this until only one element is left in the list.

Within the resultant tree, the Huffman code is encoded within the branching structure. The code for any given symbol is the path from the root node to it, where the left branches are encoded as '0' and the right as '1'.

```
  -- Combine two tree nodes into one
treeNodeCombine :: Tree a → Tree a → Tree a
treeNodeCombine l r = Node n l r where
  n = ((treeNodeWeight l) + (treeNodeWeight r))
  -- Recursively reduce a tree down to its largest element
joinTree :: [Tree a] → Tree a
joinTree ([]) = NullTree   -- Error case, shouldn't happen.
joinTree (t0 : []) = treeNodeCombine t0 NullTree
joinTree (t0 : t1 : []) = treeNodeCombine t0 t1
joinTree (t0 : t1 : ts) = (joinTree ∘ sort) newtree where
  newtree = ((treeNodeCombine t0 t1) : ts)
  -- Given a list of symbols, count the frequency of each
freqCount :: Ord a ⇒ [a] → [(Int, a)]
freqCount = map (length &&& head) ∘ group ∘ sort
  -- Given an input sequence, create a new tree
mkTree :: Ord a ⇒ [a] → Tree a
mkTree l = t where
  t = joinTree t_nodes
  t_nodes = (map (λ(n, c) → Leaf n c) ∘ freqCount) l
```

Now that we can create a *Tree*, we can use it to encode a string. We do this by first creating a dictionary mapping each symbol to its bitstring representation. Once we have that dictionary, encoding a list of symbols is as easy as mapping each one to its bitstring, then concatenating them all.

```
  -- Our actual bit type...
data Bit = L | R deriving (Eq, Show)
```

```
  -- One entry in our alphabet
data AlphabetEntry a = AlphabetEntry [Bit] a deriving (Eq, Show)
type Alphabet a = [AlphabetEntry a]
  -- Given a tree, make its forward dictionary, mapping a -¿ [Bits]
mkEncoding :: Tree a → Alphabet a
mkEncoding = mkEncodingCore [ ]
  -- The core of the encoding generator
mkEncodingCore :: [Bit] → Tree a → Alphabet a
mkEncodingCore b NullTree = [ ]
mkEncodingCore b (Leaf n c) =
  [AlphabetEntry b c]
mkEncodingCore b (Node n l r) =
  encl ++ encr where
  encl = mkEncodingCore (b ++ [L]) l
  encr = mkEncodingCore (b ++ [R]) r
  -- Look up the bits for a single alphabet entry
lookupEncoding :: Eq a ⇒ Alphabet a → a → [Bit]
lookupEncoding [ ] _ = [ ]
lookupEncoding ((AlphabetEntry bits c) : ab) x
  | x ≡ c = bits
  | otherwise = lookupEncoding ab x
  -- Given an input list, create a tree and encode the list to a bit
  -- array
encodeListToBits :: Ord a ⇒ [a] → (Tree a, [Bit])
encodeListToBits l = (t, b) where
  t = mkTree l
  enc = mkEncoding t
  b = concat $ map (lookupEncoding enc) l
```

After we create a bitstring ([*Bit*]), we want to decode it. To do this, we start at the root of the tree and follow the directions given in the bitstring. A '0' means branch left, a '1' means branch right. If we ever get to a *Leaf*, we've just decoded that symbol, so we emit it, and reset the state to the root of the tree and continue the parse.

Note that this doesn't mention what happens when we hit an erroneous term; I'm not certain what the best approach to that is just yet.

```
  -- Given a list of bits and a tree, decode the message
decodeFromBits :: Tree a → [Bit] → [a]
decodeFromBits t b = decodeFromBitsCore t b t [ ]
decodeFromBitsCore :: Tree a → [Bit] → Tree a → [a] → [a]
decodeFromBitsCore _ _ NullTree _ = [ ]
decodeFromBitsCore t bs (Leaf n c) ret =
  decodeFromBitsCore t bs t (c : ret)
decodeFromBitsCore _ [ ] _ ret = reverse ret
decodeFromBitsCore t (L : bs) (Node n l r) ret =
  decodeFromBitsCore t bs l ret
decodeFromBitsCore t (R : bs) (Node n l r) ret =
  decodeFromBitsCore t bs r ret
```

Now, we want to provide our callers with a convenient way to get a raw *ByteStream* suitable for writing out to a compressed file. The following code implements the ByteStream wrangling required, though probably not in the ideal fashion. It is, in fact, quite ugly code.

```
data BitBuffer = BitBuffer Int [Word8]
bitBit :: Bit → Word8
bitBit L = 0
bitBit R = 1
  -- Ensure that a BitBuffer is properly packed before it gets written
finalizeBitBuffer :: BitBuffer → B.ByteString
finalizeBitBuffer (BitBuffer _ []) = B.empty
finalizeBitBuffer (BitBuffer 0 cs) = B.pack $ reverse (0 : cs)
finalizeBitBuffer (BitBuffer pos (c : cs)) =
  B.pack $ reverse (n_bits : last_byte : cs) where
  n_bits = fromIntegral pos :: Word8
  last_byte = shiftL c (8 − pos)
outputBits :: [Bit] → B.ByteString
outputBits bs = finalizeBitBuffer $ oBCore (BitBuffer 0 []) bs
oBCore :: BitBuffer → [Bit] → BitBuffer
oBCore ob [] = ob
oBCore (BitBuffer 0 cs) (b : bs) = oBCore newbuf bs where
  newbuf = BitBuffer 1 (newc0 : cs)
  newc0 = (bitBit b)
oBCore (BitBuffer pos (c : cs)) (b : bs) = oBCore newbuf bs where
  newbuf = BitBuffer pos′ newc
  pos′ = (pos + 1) ‘mod‘ 8
  newc = (newc0 : cs)
  newc0 = c ∗ 2 + (bitBit b)
```

With that in place, we can now offer *encodeList*, the real encoding function, which takes in a sequence and returns a *Tree* and a buffer, suitable for writing to a file.

```
  -- Given an input list, create a tree and encode the list to a buffer
encodeList :: Ord a ⇒ [a] → (Tree a, B.ByteString)
encodeList l = (t, buf) where
  (t, bits) = encodeListToBits l
  buf = outputBits bits
```

Of course, once you have binary output, you need corresponding binary input.

```
  -- break out the bits in a Word8
word8ToBits :: Word8 → [Bit]
word8ToBits c = [if testBit c (7 − n) then R else L | n ← [0 . . 7]]
  -- Now, create a lookup table to speed things up
mkWord8BitsTable :: Array Word8 [Bit]
mkWord8BitsTable = array (0, 255) [(w, (word8ToBits w)) | w ← [0 . . 255]]
  -- Load a ByteString intoa [Bit]
inputBits :: B.ByteString → [Bit]
  -- inputBits bs = concat $ ((map word8ToBits) . B.unpack) bs
inputBits bs = bitstrings where
  table = mkWord8BitsTable
  l_bs = fromIntegral $ B.length bs
  pos_final = fromIntegral $ B.last bs
  n_bits = pos_final + (8 ∗ (l_bs − 2))
  bitstrings_raw = map (table!) $ B.unpack bs
  bitstrings = take n_bits $ concat bitstrings_raw
decodeBuffer :: Tree a → B.ByteString → [a]
decodeBuffer t bs = decodeFromBits t (inputBits bs)
```

Finally, as I was writing this code, I occasionally had need to view the *Tree* as something that looked more like a tree and less like LISP. To do so, I could write lots of ugly code to format ASCII trees, or I could make really beautiful `dot` diagram. This function creates a string containing the source for a nice tree in `dot`'s input format.

Don't know what `dot` is? `http://www.graphviz.org` is the homepage for the project.

```
  -- Tree helper function: generate a graphviz dot file depicting it
dotTree :: Show a ⇒ Tree a → String → String
dotTree t [ ] = dotTree t "T"
dotTree t name = header ⧺ body ⧺ footer where
  header = "digraph \"" ⧺ name ⧺ "\" {\n"
  body = dotTreeBody t name
  footer = "\n}"
  -- Tree helper function: generate the body of the digraph
dotTreeBody :: Show a ⇒ Tree a → String → String
dotTreeBody (Leaf n x) name =
  name ⧺ " [label=\"" ⧺ label ⧺ "\"]\n" where
  label = (show x) ⧺ "/" ⧺ (show n)
dotTreeBody (Node n x y) name =
  mynode ⧺ links ⧺ bodies where
  mynode = name ⧺ " [style=filled,color=lightgrey];\n"
  namel = name ⧺ "l"
  namer = name ⧺ "r"
  linkl = name ⧺ " -> " ⧺ namel ⧺ ";\n"
```

```
linkr = name ++ " -> " ++ namer ++ ";\n"
links = linkl ++ linkr
bodyl = dotTreeBody x namel
bodyr = dotTreeBody y namer
bodies = bodyl ++ bodyr
```