Mathematisch-Naturwissenschaftliche Fakultät

Wilhelm-Schickard-Institut für Informatik

Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

SS 2017

Benjamin Dietrich

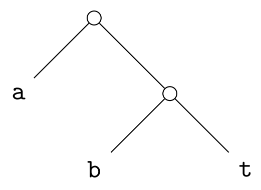
Assignment #3

Submission Deadline: Thu, 18.5.2017

Exercise 1: Encoding and Decoding of Huffman Codes (10 Points)

Every day, billion of electronic messages are transmitted. To find a compact binary coding for such messages is a major task in order to save network traffic.

This exercise is about *Huffman Codes*, a classic approach to find an optimal compact translation of messages (sequences of characters) to *bit* sequences. Messages can be *encoded* and *decoded* using binary trees. Take for example:



The tree represents a coding for the three characters `'a'`, `'b'` and `'t'`. Each character can be described by the by the *path* from the tree's root to the leaf with the character. This path is recorded by the decision to go either left (`L`) or right (`R`) at each node of the path:

Zeichen	*Code*
'a'	L
'b'	RL
't'	RR

We *encode* a message by concatenation of its characters' path-codes. For example, message `"abba"` is encoded to the bit sequence `LRLRLL` which is represented in our program by a list `[L,R,L,R,L,L] :: [Bit]`. Thus, `L` and `R` are constructors of a simple sum type `Bit` which is defined as follows:

```
data Bit = L | R
           deriving (Eq, Show)
```

In the opposite direction, *decoding* also proceeds along the tree-code. Assume that we received an encoded bit sequence `RLLRRRRLRR`. In order to decode it, we follow the given path until we find a leaf (i.e. a character) – which is `'b'` in this example. Restarting from the root, we repeat this procedure, to get a textual interpretation for the rest of the bits, too. Finally, we have a decoded cleartext message `"battat"`.

1. Define an algebraic data type `HuffTree` to represent trees which can be used for Huffman coding.
2. Write a function

   ```
   encodeMessage :: HuffTree -> String -> [Bit]
   ```

   which encodes a message of type `String` using a given tree-code of type `HuffTree`.
3. Also write a function

   ```
   decodeMessage :: HuffTree -> [Bit] -> String
   ```

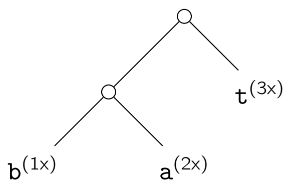
   to decode a given bit-sequence.
4. In the lecture, we've discussed the identity `(fromList . toList) xs == xs`. A similar identity holds regarding the functions `encodeMessage` and `decodeMessage`. Fomulate that identity.

Exercise 2: Optimal Huffman Trees (10 Points)

In the previous exercise we encoded and decoded messages along tress (of type `HuffTree`), based on the assumption that these trees are given before the encoding starts. However, a further interesting question is, how an *optimal* coding (i.e. tree) can be constructed, such that for a given cleartext message `m`, the encoded bit sequence `encodeMessage m` is of minimal length.

Since Huffman codings are built of path descriptions, the solution for the problem is to count the *frequency* of each character in the cleartext message. Based on this knowledge it is possible to build a tree – called *Huffman tree* – which contains the most frequent character next to the root and all other characters, in descending frequency order, further down in the tree. Thus, more frequent characters need less space for coding, while less frequent need more.

With this approach the following Huffman tree is derived for the cleartext word `"battat"` (character frequency annotated in in parentheses):



Based on this tree the encoded bit sequence for `"battat"` is LLLRRRLRR which is one bit shorter than the sequence we had in the previous exercise.

The tree is built in two steps:

1. Count the frequency of each character in the given cleartext message and keep them in an association list of character/frequency tuples (type `[(Char,Integer)]`).
2. Build the tree bottom-up out of subtrees: Start with one tree for each character (so far, each tree consists only of one leaf). Take the two trees with the lowest frequency and merge them into a new tree. The frequency of a tree is the sum of the frequencies of his subtrees. Repeat the last step until all trees are merged into one – the Huffman tree.

Note: It is allowed and might be useful to introduce additional data types and local functions.

Implement the described algorithm in a function

```
codeOf :: String -> HuffTree
```

which returns the optimal Huffman code for a given cleartext message.