COMP4161 S2/2017
Advanced Topics in Software Verification

Assignment 3

This assignment starts on Tuesday, 2017-10-10 and is due on Monday, 2017-10-23, 23:59h. We will accept Isabelle .thy files only. In addition to this pdf document, please refer to the provided Isabelle template for the definitions and lemma statements.

The assignment is take-home. This does NOT mean you can work in groups. Each submission is personal. For more information, see the plagiarism policy: https://student.unsw.edu.au/plagiarism

Submit using `give` on a CSE machine: `give cs4161 a3 a3.thy`

For all questions, you may prove your own helper lemmas, and you may use lemmas proved earlier in other questions. You can also use automated tool like sledghammer. If you can't finish an earlier proof, use *sorry* to assume that the result holds so that you can use it if you wish in a later proof. You won't be penalised in the later proof for using an earlier true result you were unable to prove, and you'll be awarded part marks for the earlier question in accordance with the progress you made on it.

1 Huffman Code (65 marks)

A Huffman code is an algorithm for lossless compression. Given a distribution of the relative frequency of each character, the algorithm assigns the most frequent character the shortest encoding. The encoding, a stream of bits, is a so-called variable-length prefix code. A code is a prefix code when no no two symbols are encoded as a prefix of the other.

As an example for a Huffman encoding, consider the sequence of symbols *abcdaa*. The frequency of the occurrence of the symbol a is 3, while that of each other symbol is 1. In this case, Huffman code will encode a with the shortest code (e.g. *True*), and all other symbols with longer codes, starting with *False*.

In this assignment we will define Huffman encoding and decoding in Isabelle, and prove, that under the right conditions, decoding an encoded sequence of symbols will yield back the original sequence of symbols.

As the first step of computing a Huffman code, we define a function *freq-list* that computes the frequency of each symbol in a given sequence. We leave the alphabet of symbols open as a type variable $'a$. The input sequence to this function is not necessarily exactly the text that we will later encode, it is merely a text corpus that has the expected alphabet and distribution of letters in this alphabet.

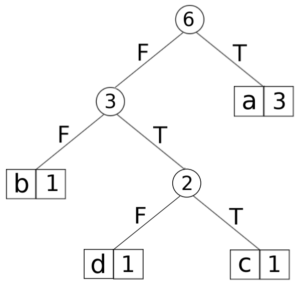
(a) Define the function *freq-of* that produces a list of pairs, in which the first component is the symbol and the second component the number of times the symbol occurs in the input list. (3 marks)
Test function with a few examples. For concrete test cases, you can use the type *string* in Isabelle, which is defined as *char list*. Concrete strings are written with two single quotes, e.g. *''abc''*.

(b) Prove that each character is mentioned only once in the result of *freq-of*. (5 marks)

Given a frequency list of symbols, we can then construct the so-called Huffman tree. The paths in the Huffman tree will give us the code for each symbol while ensuring the prefix-property and keeping track of the frequency of symbols.

datatype *'a htree = Leaf 'a int | Branch 'a htree 'a htree*

The template gives the full definition of the Huffman tree construction. It first sorts the frequency list, converts it into a list of *Leaf* trees, and then keeps merging sub-trees in this list by weight (frequency) until the list only contains one element. This element is the resulting Huffman tree.



To turn the tree into a map of symbols to codes, we traverse the tree from the root, adding *False* to the output when we go left, and *True* when we go right. See function *code-list* in the template, which returns a list of pairs where the first component is the symbol, and the second component is the code (*bool list*) for that symbol. The function *code-map* flips the first and second components of such a list, so we can also translate codes back into symbols.

The Isabelle library function *map-of* turns such $('a \times 'b)$ *list*s into functions $'a \Rightarrow 'b$ *option*, also called *map* from $'a$ to $'b$.

(c) Given a map *'a ⇒ bool list option*, write a function *encoder* that turns a list of symbols into code (a list of bool). You can assume that the map has a translation for all characters of the input. (3 marks)

(d) The template gives a definition for the corresponding decoder. It consumes one bit of input at a time, checking if the input consumed so far yields a valid code or not. If yes, it emits the corresponding symbol, if no, it keeps accumulating input.

Note that both *encoder* and *decoder* are so far independent of Huffman trees. They merely expect coding maps for single characters and translate them into functions for lists of characters. They will only behave correctly if the maps they get as input are inverse to each other, and if the code is indeed a prefix code. The domain *dom* of a map is the set of all inputs for which the output is not *None*.

Prove the following lemma:

$\llbracket$*is-inv mp mp'*; *unique-prefix mp'*; *set xs* $\subseteq$ *dom mp*; $[] \notin$ *dom mp'*$\rrbracket$
$\Longrightarrow$ *decoder mp'* [] (*encoder mp xs*) = *xs*

The template lists a few helper lemmas that you can, but do not have to prove to get there. (15 marks)

(e) Show that the inverse of a map can be constructed by swapping the first and second components of the pairs in the list, if the lists are distinct in the first and second components: (5 marks)

$\llbracket$*distinct* (*map snd xs*); *distinct* (*map fst xs*)$\rrbracket$
$\Longrightarrow$ *is-inv* (*map-of xs*) (*map-of* (*map* (λ(*a*, *b*). (*b*, *a*)) *xs*))

(f) Write a function *letters-of* that for any *'a htree* returns the set of symbols *'a* that is stored in its leaves. (3 marks)

(g) Write a function *distinct-tree*, analogous to *distinct* on lists, that decides whether the letters of an *'a htree* are distinct from each other. (3 marks)

(h) Show that *code-list* turns any tree with with distinct letters into a list with distinct symbols in the first component. (5 marks)

(i) Show that if the characters of a frequency list are distinct, so are the letters of the corresponding Huffman tree. (10 marks)

(j) Prove that the *code-list* and *code-map* lists, when turned into maps, satisfy the conditions of the decoder lemma when the input is a Huffman tree whose letters cover the input string and whose alphabet contains at least two separate letters. (13 marks)

2 C verification: primality test (35 marks)

The function *is-prime-2* below tests whether a given number n is a prime or not.

```
unsigned int is_prime_2(unsigned int n)
{
  /* Numbers less than 2 are not prime. */
  if (n < 2) {
    return 0;
  }

  /* 2 is the only even number that is prime */
  if (n % 2 == 0) {
    return (n == 2);
  }

  /* Find the first non-trivial factor of 'n'. */
  unsigned int i = 3;
  while (n % i != 0) {
    i+=2;
  }

  /* If the first factor found is 'n', it is a prime */
  return (i == n);
}
```

We will verify that *is-prime-2* correctly computes the primality of n. In other words, we will show that *is-prime-2* returns *1* if and only if the input n is a prime and that the computation always terminates.

(a) Define a predicate *is-prime-inv* which states the invariant of the while loop in the *is-prime-2* function. (4 marks)

(b) Define a measure *is-prime-measure* which returns a natural number that strictly decreases each loop iteration. (3 marks)

(c) Using AutoCorres ("*apply wp*"), show *is-prime-2* is correct. To prove this, first introduce the following lemmas:

(c1) *prime-2-or-odd* which states that a prime number n is either equal to *2* or an odd number. (3 marks)

(c2) *is-prime-precond-implies-inv* which states that the invariant holds when you first enter the loop; (3 marks)

(c3) *is-prime-body-obeys-inv* which states that the invariant holds between loop iterations; (5 marks) (c4) *is-prime-body-obeys-measure* which states that the measure decreases between loop iterations; (5 marks)

(c5) *is-prime-body-implies-no-overflow* which states that the loop invariant implies there is no overflow (4 marks)

(c6) *is-prime-inv-implies-postcondition* which states that the invariant implies the function's post-condition when the loop finally finishes. (4 marks)

(d) Finally, use these lemmas to complete the proof. (4 marks)

Hints

- While C uses 32-bit words (of Isabelle type `word32`), AutoCorres uses a technique called *word abstraction*, allowing you to reason using `nat`'s instead. This comes at the price of being obliged to prove that arithmetic operations don't overflow `UINT_MAX` (i.e., $2^{32} - 1$).
- The tactic "`apply arith`" may help to solve 'obvious' proofs involving arithmetic. Additionally, you may find *sledgehammer* to also be effective at proofs that seem obvious.
- Feel free to modify any of the definitions or lemmas in the template. The goal is to prove the function is correct: if changing some of the provided lemmas helps you achieve this, don't be afraid to do it.