Implementation and verification of encoder, decoder and generator for prefix-free codes

Samuel Chassot ✉
EPFL

Daniel Filipe Nunes Silva ✉
EPFL

Abstract

We propose an implementation in the Scala programming language of an encoder, a decoder as well as a naive prefix-free code generator. These implementations are formally proven for correctness thanks to the verification framework Stainless [2]. Those three functions can be chained to form a pipeline as a whole on a given input string s, i.e. generate a correspoding prefix-free code c, use c to encode s as e and finally decode e with c as d. Correctness is therefore defined as $s == d$.

2012 ACM Subject Classification Software and its engineering → Software verification and validation

Keywords and phrases formal verification, prefix-free codes, Huffman Coding

1 Introduction

The aim of this project is to come up with a verified implementation of an encoder and decoder pair as well as a prefix-free code generator in the scope of a personal project for the Formal Verification course (CS-550) at EPFL.

In [1], J. C. Blanchette presents a formal proof of the optimality for the Huffman's algorithm. The latter generates a prefix-free code for a string such that the encoding of the encoded string has minimal length. As proposed in the conclusion of Blanchette's article, we explored the the verification on encode/decode functions.

In this report, we start with some background knowledge related to prefix-free codes. Then, we go through the verified implementation of the pipeline we came up with.

Prior work

To the best of our knowledge, this is the first time an encode decode function pair is formally verified. However we can find many examples of similar exercises in the litterature such as formally defining Huffman's algorithm in [4] or implementing formally verified compression decompression function pair for deflate in [3].

Our approach

We first considered the whole class of prefix-free codes and not only the optimal ones generates by Huffman's algorithm. Then, we came up with basic datatypes and implementations plain Scala code before adding *requires* and *ensurings* from Stainless in order to validate the main correctness theorem Finally, we also implemented and verified an naive prefix-free code generator in order to complete the whole pipeline.

Contributions

The main contributions of this report are the following:
1. formally verified of an encode function for prefix-free codes
2. formally verified of an decode function for prefix-free codes
3. formally verified of a prefix-free code generator

2 Implementation

Our code is available in the following repository:

`https://github.com/samuelchassot/FormalVerification-Project`

and build instructions are in file `README.md` in the top-level directory.

3 Prefix-free codes

We define a code as map from a list of characters formed from a string, the symbols, to a list of bits, the code words. The fact that a code is prefix-free means that there is no full code word in this map that is a prefix of an other code word, i.e. 01 is a prefix of the code word 0111.

Those codes can be represented as full binary trees. To determine the code word of a symbols located in the leaves, it suffices to find the path from the root down to the corresponding leaf assuming that taking the left child represents the 0 bit and the right child represents the 1 bit.

'a': 0, 'b': 10, 'c': 110, 'd': 111 is a prefix-free code and the correponding full binary tree can be seen in figure 1.

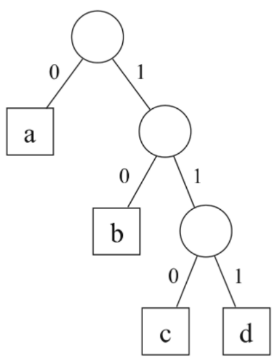

Figure 1 A full binary tree representing a prefix-free code.

4 Datatypes

TODO

5 Encode

TODO

6 Decode

TODO

7 Prefix-free code generator

TODO

8 Future work

Huffman's algorithm was the starting point of this project altough we did not clearly exploit it. We restricted ourselves to the broader class of prefix-free codes and how to generate some without worrying about its optimality as in [1].

As an improvement, we suggest to extend our current implementation of naive prefix-free code generator to an actual formally verified implementation of Huffman's algorithm. This is surely more challenging to prove since Huffman's algorithm requires sorting the forest of tree at each iteration. Ensuring the forest properties such as the characters that can be uniquely decoded with it, is not trivial.

9 Conclusions

TODO

Acknowledgements. The autors of this report would like to thank Viktor Kunčak for proposing such course at EPFL and offering to get some experience with a personal project they truly enjoyed to work on. The autors thank Jad Hamza, who was often able to provide precious help for the progress of the project despite the lack of context.

References

1 Jasmin Christian Blanchette. Proof pearl: Mechanizing the textbook proof of huffman's algorithm. *J. Autom. Reason.*, 43(1):1–18, June 2009. `doi:10.1007/s10817-009-9116-y`.

2 Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. *Proc. ACM Program. Lang*, (OOPSLA), November 2019. `doi:https://doi.org/10.1145/3360592`.

3 Christoph-Simon Senjak and Martin Hofmann. An implementation of deflate in coq, 2016. `arXiv:1609.01220`.

4 L. Théry. Formalising huffman's algorithm. 2004.