

# 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 corresponding 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 explore the verification of 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 propose.

## 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 literature such as a formal definition Huffman's algorithm in [4] or a formally verified implementation of compression/decompression function pair for deflate in [3].

## Our approach

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

## Contributions

The main contributions of this report are the following:

1. a formally verified implementation of an encode function for prefix-free codes
2. a formally verified implementation of an decode function for prefix-free codes
3. a formally verified implementation of a prefix-free code generator

## 2 Implementation

Our code is available in the following repository:

<https://github.com/samuelchassot/FormalVerification-Project>

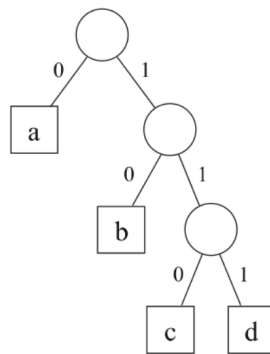
and build instructions are in file `README.md` in the top-level directory.

## 3 Prefix-free codes

We define a code as a 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 corresponding full binary tree can be seen in figure 1.



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

## 4 Datatypes

We represent a string (the type of an input of the pipeline) as `List[Char]` as it is easier to define recursion using lists rather than the `String` type directly. Similarly we define an encoded binary string (the type of an output of the pipeline) as `List[Boolean]`.

To represent the binary trees used to represent the prefix-free codes, we define the following case classes:

```
sealed abstract class Tree
case class InnerNode(t1: Tree, t2: Tree) extends Tree
case class Leaf(w: BigInt, c: Char) extends Tree

type Forest = List[Tree]
```

A `Forest` is a list of trees that is used in the process of generation of the prefix-free code. We define two functions on `Trees`:

- `isSubTree(t: Tree, st: Tree)`: boolean value indicating if `st` is a subtree of `t`.
- `isSameTree(t1: Tree, t2: Tree)`: equality between trees) are reflexive and transitive.

We then prove that these two relations are transitive and reflexive.

## 5 Encoder

### 5.1 "Encodability"

To define the `encode` function, we first need to define its parameters, output and preconditions. The `encode` function takes as input a `List[Char]` and a `Tree` and produces a `List[Boolean]`. It requires that the `Tree` is in fact a `InnerNode` instance. Willing to encode something with only a `Leaf` would not make sense as the bits are added to the output binary string when an edge of the tree is followed. It also requires that the `List[Char]` is non-empty and that each of its character is uniquely encodable by the tree. To be uniquely encodable means that the character appears exactly once in the leaves of the tree. If the character does not appear in the tree, we simply cannot encode it and if it appears more than once, the behavior of the function is undefined.

We so define the following function that return true if and only if the given character appears only once in the tree's leaves:

```
def canEncodeCharUniquely(t: Tree, c: Char): Boolean
```

### 5.2 Encode a character

We first define a function `encodeChar(t: Tree, c: Char): List[Boolean]` which encodes the given `Char` using the given `Tree`. This function also requires the `Tree` to be an instance of `InnerNode` and the `Char` to be uniquely encodable.

This function is defined recursively:

- it first checks which of the left or right child can encode the character (using the `canEncodeCharUniquely` function),
- if the subtree that can encode the character is a `Leaf`, it returns `List(False)` if it is the left child or `List(True)` if it is the right,
- if the subtree is a `InnerNode`, it performs a recursive call on this subtree and appends the result to either `List(False)` or `List(True)` depending on the left or right again.

### 5.3 Encode a string

## 6 Decoder

### 6.1 "Decodability"

### 6.2 Decode a string

## 7 Prefix-free code generator

## 8 Main theorem

## 9 Future work

Huffman's algorithm was the starting point of this project although 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.

## 10 Conclusions

**Acknowledgements.** The authors of this report would like to thank Viktor Kunčák for proposing such course at EPFL and offering to get some experience with a personal project they truly enjoyed to work on. The authors 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 Kunčák. 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.