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

Samuel Chassot
EPFL

Daniel Filipe Nunes Silva
EPFL

Abstract

Prefix-free codes are widely used in communication systems, e.g. in compression algorithms combined with Huffman's algorithm. In this work, we propose an implementation in the Scala programming language of an encoder decoder pair as well as a naive prefix-free code generator. Such implementations can be critical due to the fundamental role they play. Therefore, our 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 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 of 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 litterature such as a formal definition Huffman's algorithm in [4] or a formally verified implemention 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 vanilla Scala before adding *requires* and *ensurings* from Stainless in order to validate the main correctness theorem. Finally, we also implement and verify an 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 decode pair for prefix-free codes
2. 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-PrefixFreeCodes`

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 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, e.g. 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.

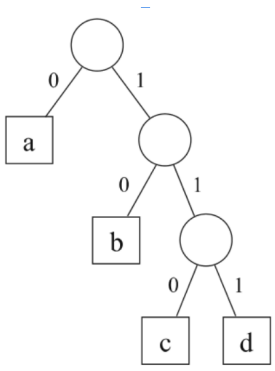

Figure 1 A full binary tree representing a prefix-free code.

4 Datatypes

The inputs of our pipeline are string stored as `List[Char]` since it is easier to define recursion on lists rather than on the `String` type directly with Stainless. Similarily, the outputs of our pipeline, an encoded binary string are defined as `List[Boolean]`.

We define the following case classes to describe the full binary tree used to represent prefix-free codes and `Forest`, which is a list of trees used for the generation of the prefix-free codes:

```
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]
```

On the one hand, we use `InnerNode`s representing the intermediates nodes from the root down to the leaves and they are fully defined by their two children. On the other hand, we have `Leaf` to describe the bottom nodes. Those contain the character `c` they represent as well as their correspoding weight `w`, i.e. how many time it appears in the corresponding string.

Moreover, we define two functions on `Tree`s that are proven to be reflexive and transitive:

- `isSubTree(t: Tree, st: Tree)`: boolean value indicating if `st` is a subtree of `t`.
- `isSameTree(t1: Tree, t2: Tree)`: boolean value indicating if two trees are the same, i.e. all the nodes are the same.

5 Encoder

5.1 "Encodability"

Before implementing the `encode` function, we need to define the notion of encodability. For a character to be encodable with a given tree, it must appear exactly once in its leaves. 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 therefore define the following function that returns true if and only if the given character appears exactly once in the tree's leaves:

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

When encoding, we also require that the tree is an `InnerNode` because encoding with a single `Leaf` is not properly defined. In fact, there is no point in encoding a string that contains only one unique character.

5.2 Encode a character

We first define the following :

```
def {encodeChar(t: Tree, c: Char): List[Boolean]}
```

It is used to encode the given `Char` recursively using the given `Tree`. This function also requires the `Tree` to be an instance of `InnerNode` and the `Char` to be uniquely encodable :

1. It first checks which of the left or right child of the tree contains the character to encode
2. if the child is a `Leaf`, it returns `List(False)` if it is the left child or `List(True)` if it is the right,
3. if the child is a `InnerNode`, it performs a recursive call on this subtree and appends the output to either `List(False)` or `List(True)`.

For the first point to be valid, we define the following lemma: if a tree can encode uniquely a character, then exactly one of its children can. We then know that only one branch of the condition is valid at a time.

The function has a postcondition asserting that the produced list of bits is actually decodable and that, if it is decoded, it returns the exact same character that is to be encoded and all bits of its encoding are consumed.

5.3 Encode a string

We first define the following :

```
def encode(t: Tree, s: List[Char]): List[Boolean]
```

It requires the `Tree` to be a `InnerNode` and the `List[Char]` to be non-empty and that each of its character is uniquely encodable with the tree.

We now define the `encode` function itself. It is defined recursively as follows:

1. If there is only one char in the input list, call `encodeChar` and returns the result.
2. Else, call `encodeChar` on the head and concatenate the result with the result of a recursive call on the tail.

The postcondition asserts that the produced list of bits is decodable using the same tree and that, if decoded, gives back the same string as the one received as input.

To prove this postcondition, we have to define the following lemmas:

1. Assert that `encodeChar` produces a list of bits that is decodable and gives the right result back.
2. Assert that if we can decode two lists of bits, we can still decode the concatenation of the two.
3. Assert that if we can decode a list of bits, then we can correctly decode at least one character from it and that we can still decode correcty the remaining bits.

6 Decoder

6.1 "Decodability"

Similarily to the notion of encodability, we define the notion of decodability. Namely, the two following functions:

```
def canDecodeAtLeastOneChar(t: Tree, bs: List[Boolean]): Boolean
def canDecode(s: Tree, bs: List[Boolean])
             (implicit t: Tree): Boolean
```

The first one returns true if it is possible to decode at least one character from the given list of bits using the given tree. It goes down the tree from the root and follows the proper edge recursively given the head bit of the list and consumes it. If it reaches a leaf it returns true. This means that given the bits to decode a character from, you could actually find a path from the root of the tree to a character. If it does not reach a leaf, it is eventually stuck in "the middle of the tree" with no more bits to be consumed. In this case, the function returns false since the provided bits were not enough to decode at least one character from.

The second one returns true only if we can decode the whole list of bits, i.e. we can decode one or more characters and when the list of bits is empty, we effectively reached a leaf. It is defined recursively and uses `canDecodeAtLeastOneChar`.

Then, we prove a few lemmas related to those two functions, the most important ones being the following:

- `canDecodeImpliesCanDecodeAtLeastOneChar`: proves that, if we can decode the whole list of bits, then we can decode at least one character from it.
- `canDecodeImpliesCanDecodeTailAfterOneCharDecoded`: proves that, if we can decode the whole list of bits, then if we decode the first character, we can still decode the remaining bits.

6.2 Decode a character

We first define the following :

```
def decodeChar(t: Tree, bs: List[Boolean]): (Char, List[Boolean])
```

It produces a tuple containing the decoded `Char` and the remaining `List[Boolean]` to be decoded. As preconditions, it requires that the given `Tree` is in fact an `InnerNode` and that we can decode at least one character from the given list of bits using the given tree.

Concerning the implementation, it is defined recursively: going down the tree consuming a bit from the list at each step, when `False` it calls recursively on the left child or on the right one when `True` until it reaches a leaf. At this point, it returns the character that is in the leaf and the remaining list of bits.

6.3 Decode a string

We first define the following :

```
def decode(t: Tree, bs: List[Boolean]): List[Char]
```

The output corresponds to the list of characters obtained by decoding the list of bits using the given tree. It has no postcondition but some preconditions. Like `decodeChar`, it requires the given `Tree` to be an `InnerNode` but also that we can decode the entire given list of bits using the given tree.

Concerning the implementation, it first calls `decodeChar` and if the remaining list of bits in the resulting tuple is empty, it returns `List(c)` where `c` is the character in the tuple. If the remaining bits list is not empty, it calls itself recursively on this list and returns `c :: res` where res is the list produced by the recursive call.

The lemmas are called in the implementation to ensure that the preconditions of the calls to `decodeChar` and to `decode` are indeed valid.

7 Prefix-free code generator

To complete the pipeline, we implement an algorithm that produces a prefix-free code for a string. It is a naive algorithm in the sense that it does not produce an optimal such that the encoded string is the shortest possible given the input.

It takes as parameters a `List[Char]` and works as following:

1. A function produces a `List[(Char, Int)]` where each character of the input list appears once and the integer in the tuple is the number of times this character occurs in the input list. We call this list the occurences.
2. A function takes the occurences and produces a `Forest` of leaves where each character appears in one leaf with the number of occurences as weight.
3. The tree is then produced by recursively merging the two first trees of the forest into a new `InnerNode` and calling the function recursively on the resulting `Forest` until only one tree is left.

To implement this algorithm, we use three functions:

```
def generateOccurrences(chars: List[Char])
                       (implicit s: List[Char]): List[(Char,
                                                       BigInt)]
def generateForest(s: List[Char]): Forest
def naivePrefixFreeCode(f: Forest)(implicit s: List[Char]): Tree
```

These three functions have sufficient pre- and postconditions ensuring during the whole process that, each character of the input list in uniquely encodable by the produced tree. This proof is not trivial and requires to prove that the list of characters is indeed preserved during the whole process but without any duplicates.

8 Main theorem

The main theorem we prove in the end is the following:

```
    def decodeEncodedString(s: List[Char]): Unit = {
        require(removeDuplicates(s).length > 1)
    }.ensuring(_ => {
        val t = generatePrefixFreeCode(s)
        val e = encode(t, s)
        val d = decode(t, e)

        s == d
    })
```

It states that, if we take the input string, generate a prefix-free code tree, use it to encode the list and decode the produced encoding, we indeed get back the original list.

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.

We could also explore how to adapt our work to different kind of codes. In fact, how could this work with the broader class of uniquely decodable codes, although prefix-free codes are already a large family of codes ? The main challenge would probably be to come up with an elegant way of representing such codes. In our case, representing prefix-free codes as full binary trees gave us a nice advantages, e.g. they allow recursion which is natural to work with in Stainless.

10 Conclusion

We propose a formally verified implementation in Scala of an encode-decode functions pair for prefix-free code. To complete the pipeline, we also propose a formally verified prefix-free code generator. We verify the whole pipeline, that means that, given a string represented by a `List[Char]`, it can generate the prefix-free code, encode the string and decode the encoded output correctly.

The most difficult part of this project is to figure out how to break problems into pieces small enough so that Stainless can be used to prove them. It is frustrating to see how trivial the solution appears but how difficult it is to find it in the first place.

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.