CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Deflate-like compressor in HOL

Creating executable binaries using CakeML

Master's thesis in Computer Science - Algorithms, Languages and Logic

Eric Carlsson
Li Rönning

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

A Deflate-like compressor in HOL

Creating executable binaries using CakeML

Eric Carlsson
Li Rönning

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

A Deflate-like compressor in HOL
Creating executable binaries using CakeML
Eric Carlsson
Li Rönning

Supervisor: Magnus Myreen, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

A Deflate-like compressor in HOL
Creating executable binaries using CakeML
Eric Carlsson, Li Rönning
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Deflate is a well-known file format specification of lossless compression, with implementations in software like gzip and zip. Proving that a compressor following the Deflate algorithm has the property of being entirely reversible is of interest, as Deflate is widely used. The aim of this project is to produce verified binaries for the Deflate algorithm using the interactive theorem prover HOL4 and the CakeML compiler. The implementation consists of three main parts: the LZSS compression algorithm, the Huffman encoding algorithm, and the Deflate algorithm. Since the focus was to properly implement Deflate, the majority of our effort was put into developing the algorithm itself. Due to time constraints however, our implementation reached an executable implementation of a Deflate-like algorithm. While the Deflate algorithm has been implemented and verified before, the work described here is the first to produce an executable binary of a compression specification.

Keywords: Deflate, Huffman encoding, LZSS compression, formal verification, interactive theorem prover, HOL, CakeML.

Acknowledgements

First, we would like to thank our supervisor Magnus Myreen for the support and ideas he has provided throughout the project. We would also like to thank our examiner Carl-Johan Seger who has provided us with valuable feedback. Furthermore, we extend our appreciation to the entire HOL4/CakeML community for their informative, patient and quick answer to our questions. Lastly, we are grateful to Alexander Cox for allowing us to continue from where he left off with his LZSS implementation and verification.

Eric Carlsson, Li Rönning
Gothenburg, June 2022

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Contributions . . . 2
1.2 Background . . . 2
1.2.1 HOL4 . . . 2
1.2.2 CakeML . . . 4

2 A Simple Compressor **5**
2.1 Background . . . 5
2.1.1 Dictionary compression . . . 5
2.1.1.1 Static dictionary compression . . . 6
2.1.1.2 Semi-adaptive dictionary compression . . . 6
2.1.1.3 Adaptive dictionary compression . . . 6
2.1.1.4 Parsing strategies . . . 7
2.2 A simple static dictionary compressor . . . 7
2.2.1 Compressor . . . 9
2.2.2 Dictionary . . . 11
2.3 Problems . . . 12

3 Deflate **13**
3.1 Background . . . 13
3.1.1 Compression in Deflate . . . 13
3.1.2 Huffman encoding in Deflate . . . 14
3.1.3 Block format . . . 15
3.1.3.1 Uncompressed block . . . 16
3.1.3.2 Fixed Huffman block . . . 16
3.1.3.3 Dynamic Huffman block . . . 17
3.2 Implementation . . . 18
3.2.1 Encoding Deflate . . . 18
3.2.1.1 Encoding the compressed data . . . 20
3.2.1.2 Transmitting the trees for decoding . . . 21
3.2.1.3 Run-length encoding (RLE) . . . 22
3.2.1.4 Transmitting the code length alphabet . . . 22
3.2.2 Decoding Deflate . . . 22

3.2.3 Main compression functions 24
3.3 Evaluation of Deflate binary 25

4 LZSS **27**
4.1 Background 27
4.1.1 LZSS by Alexander Cox 27
4.1.1.1 LZSS implementation 27
4.1.1.2 Ringbuffer implementation 29
4.2 Implementation 30
4.2.1 Changes made to Cox's implementation 30
4.2.2 LZSS using Ringbuffer 31

5 Huffman encoding **33**
5.1 Background 33
5.1.1 Huffman encoding 33
5.1.2 Fixed Huffman encoding 35
5.1.3 Canonical Huffman codes 36
5.2 Fixed encoding 37
5.2.1 Constructing the tree 37
5.3 Dynamic encoding 38
5.3.1 Building a tree 38
5.3.2 Building a canonical Huffman tree 40
5.4 Encoding and decoding 40

6 Discussion **41**
6.1 Related work 41
6.2 Future work 42
6.3 Testing the binary 43
6.4 Lessons learned 44
6.5 Conclusion 44

Bibliography **45**

List of Figures

1.1 Example HOL4 script file, exampleScript.sml, with a basic `Datatype`, `Definition` and `Theorem`. 3
1.2 The goal coupled to the theorem `sweden_is_large` in the HOL REPL. 3
1.3 The result of proving the goal for the theorem `sweden_is_large` in the HOL REPL. 3

2.1 Functions used by compress and decompress and their dependencies. . 10

3.1 A Huffman tree containing the encoding for the characters and lengths in "Hello! Hi! (6,11)". 15
3.2 The general structure of a block created by a Deflate compressor. . . 16
3.3 The block format for an uncompressed block (BTYPE=00). 16
3.4 The internal block format for the fixed Huffman block (BTYPE=01). 16
3.5 The internal block format for the dynamic Huffman block (BTYPE=10). 17
3.6 Definition of `encode_block`. 19

5.1 Step 1. Sort the leaf nodes is descending order. 34
5.2 Step 2. Combine the leaf nodes with the lowest counts and add the parent to the sorted leaf nodes. 34
5.3 Step 3. Combine the two nodes with the lowest counts and add the parent to the sorted nodes. 34
5.4 Step 4. Combine the two nodes with the lowest counts and add their parent to the sorted nodes. 35
5.5 The result is a Huffman tree for the text "bbbbbaaaccdde". 35
5.6 A canonical Huffman tree representing the code lengths `[2,2,2,3,3]`. 36

List of Tables

2.1 Dictionary for a short compression example. 8
2.2 Compression of the text "We all live in a yellow submarine, yellow submarine, yellow submarine" using the dictionary from Table 2.1 . . 8

3.1 Evaluation of the compression ratio for our Deflate encoder binary compared against `zip`. 26
3.2 Evaluation of the execution time for our Deflate encoder binary compared against `zip`. 26

4.1 Example of compression state to showcase recursive backreferences in `matchLengthRB` where the overline represents the first index and the underline represents the seconds index. 31

5.1 The predefined code lengths for the fixed Huffman tree used in Deflate [1]. 37

1

Introduction

Compression is a widely used technique to optimise data storage and transmission. In situations such as these, lossless compression is of particular interest as it comes with the guarantee of being entirely reversible. The reversible feature means that we can optimise the handling of the data and still interpret it after decompression.

Two algorithms are needed to achieve compression. A compression algorithm that takes advantage of redundancy in the input to compress the data, and a decompression algorithm that interprets the compressed data and restores it to its former shape. The format the data takes after compression will not be legible unless decompressed. From here on, a compressor will refer to a compression scheme consisting of both a compression and a decompression algorithm.

Deflate [1] is a well-known file format specification of lossless compression, with implementations in software like gzip [2] and zip [3]. It specifies the use of LZ77 compression and Huffman encoding, with extra details on how the combination should work.

Since Deflate is found in well-known software, proving that a compressor following the Deflate algorithm has the property of being entirely reversible is of interest. Proving the correctness of Deflate can be achieved by formal verification using tools such as HOL4.

Implementations of Deflate have been verified before. Senjak and Hoffman [4] constructed a verified implementation of Deflate using the interactive theorem prover Coq [5]. Huffman encoding has been formally verified separetly from Deflate both by Blanchette [6], with the use of Isabelle/HOL [7], and Théry [8], who also used Coq.

In contrast to Senjak and Hofmann's [4] verified implementation, we produce an executable binary of our implementation. If they wished to compile their implementation into a binary, they would need to extract their program to another host language, like OCaml, and then compile the binary. However, if the translation and compilation processes are not verified, the produced binary would not be verified either. We construct a Deflate binary by implementing a Deflate algorithm in HOL4, and then compiling it using the verified CakeML compiler inside the logic of HOL4.

1.1 Contributions

This project is, to our knowledge, the first that produces an executable binary for a non-trivial compression and decompression specification. We created binaries for two different compressors. One is a simple static dictionary compressor, and the other is a more comprehensive adaptive compressor that to a large extent follows the Deflate algorithm defined in RFC 1951 [1].

The compression and decompression algorithms are defined in HOL4 and then compiled using the CakeML compiler. The tools HOL4 and CakeML will be explained in Section 1.2. We explain how compression works, with a focus on dictionary compression, and construct an example compressor to highlight important properties and potential problems (Chapter 2).

The implementation of a Deflate algorithm was the main focus of this project (Chapter 3). We also constructed implementations for the two main mechanisms of Deflate, LZSS compression (Chapter 4) and Huffman encoding (Chapter 5). Part of the implementation of LZSS compression was begun by Alexander Cox, a student at Australian National University, ANU.

1.2 Background

To understand this project, it is necessary to understand the tools that it is built for and relies upon. These tools are HOL4 and CakeML.

1.2.1 HOL4

Higher-Order-Logic 4, HOL4 for short, is a proof assistant, also known as an interactive theorem prover, specialised in higher-order logic. Using an interactive theorem prover when constructing proofs makes the development easier as proofs strictly adhere to the formal rules of the logic and can be mechanically checked by a computer [9]. When using built-in tools and theorem provers, most easy proofs can be automatically proved, and support can be given when tackling complex proofs.

When using HOL4, all development is done in script files ending with the suffix "Script.sml" [9]; the syntax is a close derivative from SML (Standard Meta Language), which is a modern dialect of ML [10]. These files consist of definitions that contain SML like code, data types that allow for the creation of custom algebraic data types, and theorems with logical propositions and tactics to prove them correct. For an example that demonstrates a basic `Datatype`, `Definition` and `Theorem`, see Figure 1.1.

```
open preamble;
val _ = new_theory "example";

Datatype:
  scandinavianCountries =
      Sweden
    | Denmark
    | Norway
End

Definition getPopulation_def:
  getPopulation Sweden  = 10400000 ∧
  getPopulation Denmark =  5800000 ∧
  getPopulation Norway  =  5500000
End

Theorem sweden_is_large:
  ∀ country. country = Sweden  ⇒  10000000 ≤ getPopulation country
Proof
  Cases_on ‘country’
  \\ simp[getPopulation_def]
QED

val _ = export_theory();
```

Figure 1.1: Example HOL4 script file, exampleScript.sml, with a basic `Datatype`, `Definition` and `Theorem`.

Those coming from a traditional functional programming background may find that `Definition` and `Datatype` act as they would in most conventional functional programming languages. It is with `Theorem` and tactics where HOL4 differs. A `Theorem` allows the developer to define propositions using logic and create proofs regarding various properties for their `Definition`. However, simply writing the propositions does not prove the theorems true, and this is where tactics come in. Tactics are functions applied to theorems that transform them in various ways to prove that they indeed hold. A `Theorem` statement can be sent into HOL, thus creating a goal, as shown in Figure 1.2.

```
Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:
     ∀country. country = Sweden ⇒ 10000000 ≤ getPopulation country
```

Figure 1.2: The goal coupled to the theorem `sweden_is_large` in the HOL REPL.

The goal is then solved by the tactics found between `Proof` and `QED` in Figure 1.1. The tactic `Cases_on ‘country‘` enumerates all different cases for the variable `country` and creates a subgoal for each. Here "\\" is a connective that applies the following tactic to each subgoal the previous tactic creates. With the use of `simp[getPopulation_def]`, each subgoal is simplified with regard to the given definition. The result that HOL4 managed to prove the goal is shown in Figure 1.3.

```
Initial goal proved.
⊢ ∀country. country = Sweden ⇒ 10000000 ≤ getPopulation country: proof
```

Figure 1.3: The result of proving the goal for the theorem `sweden_is_large` in the HOL REPL.

Theorems can be seen as an alternative method to tests for creating guarantees that certain functions behave as expected of them. Beyond the typical theorem there are also termination proofs. When creating a `Definition`, HOL4 has to be sure that the defined function will eventually terminate. HOL4 attempts to prove that it terminates using a naive strategy. If it fails, the developer has to provide a termination proof, i.e. show that the definition will eventually terminate.

1.2.2 CakeML

Verified applications refer to programs and functions that have been proven to follow a certain formal specification [11]. Once a program is verified, it can be guaranteed to follow the proven specifications.

CakeML can refer to many things, a functional programming language, an ecosystem of proofs and verified applications defined in HOL4 [12], but most notably, a verified compiler. Beyond just accepting the syntax of the CakeML programming language, it can compile HOL4 definitions into binaries. Due to the verified nature of the compiler, the resulting binary's behaviour is guaranteed to conform to the proved properties of the definition used as input.

To create an executable binary for the HOL4 definition `testFunction_def` two files have to be created: `testProgScript.sml` and `testCompileScript.sml`. The file `testProgScript.sml` defines the binary's functionality through the definition `main_function_def`, where `testFunction_def` in turn is called. The input and output of the binary is expected to be of a special type, *mkstring*, then the necessary parsing and formatting is handled in `testProgScript.sml`. The result is a properly configured HOL4 definition using tools provided by CakeML. The configured definition is then imported into `testCompileScript.sml`, which defines the compilation of the binary.

2

A Simple Compressor

This chapter covers background for compression and a small static dictionary compressor implemented using HOL4. Lastly, we present inherent problems and drawbacks with this type of dictionary.

2.1 Background

Compression algorithms can be grouped into two different groups, lossy and lossless compression, and can be used in different scenarios for different kinds of media [13]. For example, a certain amount of loss can be acceptable when compressing images or video as long as the content is still understandable. Whereas with text, a loss is unacceptable as the contents may no longer be legible, or the behaviour of the contents may potentially change.

Several methods to achieve compression exist; these include statistical modelling or using a dictionary to lookup text matches [13]. Statistical modelling uses probabilistic models of the text to predict the likelihood of the following character or substring. This likelihood is then used to decide upon a codeword, and a similar process is used upon decompression. Dictionary compressors compress by exchanging substrings in the text for codes based on their dictionary. If a compressor with the static dictionary {hello=a, world=b} compressed the string "hello world", then the result would be "a b". This procedure can easily be reversed by flipping the dictionary and repeating the process. For Deflate, knowledge of dictionary compressors is necessary. The rest of this chapter explains dictionary compressors using a simple instance.

2.1.1 Dictionary compression

The main difference among dictionary compression algorithms is how they handle their dictionary. Bell, Witten and Cleary [13] divided dictionary compressors into three different categories; static, semi-adaptive and adaptive. These range from not adapting the dictionary the compressor uses to the input to basing the dictionary entirely on the input.

2.1.1.1 Static dictionary compression

Static compressors use the same dictionary for all texts [13], which has its drawbacks, as the model can match very poorly with the text it is compressing. There are also cases where these compressors do well; for example, if we know in what setting the compressor will be used, the model can be shaped to work well for that setting. However, creating a new dictionary for every scenario in which compression is desired would be unreasonable.

One basic form of static dictionary compression is using digram coding, where digram refers to pairs of characters [14]. The dictionary would be a collection of digrams, e.g. “ab”, “bc”, and “cd”. Digram dictionaries are typically built on top of a character set. For example, the ASCII character set has 96 printable characters and is typically stored in 8-bits and thus has room for 256 values. The 160 unused values can then be used to represent digrams. Instead of just digrams, the values could also represent triplets of characters called trigrams or general strings called n-grams.

In each iteration during digram compression, the first two characters of the text are inspected to check if there are any matches in the dictionary. If a match is found, then the corresponding code is used; otherwise, the first character is coded and the procedure is repeated.

2.1.1.2 Semi-adaptive dictionary compression

The natural progression from static compression is semi-adaptive compression, where the dictionary is generated from the input [13]. A drawback is that semi-adaptive compressors require two passes over the text, the first to generate the dictionary and the second to compress the text. Furthermore, the generated dictionary also has to be transmitted with the message in order to decompress it [13]. However, in static compression, the dictionary is known beforehand since it remains the same for all inputs.

2.1.1.3 Adaptive dictionary compression

In 1977 Lempel and Ziv [15] created a compression algorithm known as the LZ77 compression algorithm. Instead of using pointers to a dictionary, Lempel and Ziv introduce a “sliding window” that acts as both dictionary and lookahead. The lookahead refers to the text to compress next, so a lookahead of size 258 would contain the next 258 uncompressed characters. The algorithm works by taking strings from the lookahead and comparing them against the contents of the dictionary. If a match is found, the string is replaced with a triplet representing length, distance and a character. Where the distance is how far back the start of the match is in the dictionary, the length is how long the match is, and the character is used if no match was found.

The LZ77 algorithm is the foundation of adaptive compression, and many variations of the original LZ77 compression algorithm exist, each reflecting its own design decision. Some may have an infinite dictionary or a longer lookahead, giving them

more opportunities for better compression at the cost of compression speed and required working memory. Depending on the task and resources available different variations would be more suitable.

2.1.1.4 Parsing strategies

One problem shared among static, semi-adaptive and adaptive compressors is how to substitute substrings using their dictionaries, known as the parsing problem [13]. The fastest and most straightforward approach is greedy parsing which attempts to find the longest matching substring from the start of the text. However, due to its greedy nature, it is not necessarily optimal [16].

There have been several optimal parsing strategies proposed, examples include Schuegraf and Heaps [16], and Storer and Szymanski [17], but where they gain in compression, they lose in time efficiency. The optimally compressed files have been shown only to achieve a few per cent more compression while being two to three times slower to perform. Another limitation is that optimal parsing strategies expect the entire text to be available upon parsing. However, this is not always the case and restricts where compression may be used. For these reasons, greedy parsing is the most commonly used option.

2.2 A simple static dictionary compressor

We will present an example compressor with a static dictionary to showcase how compression can be done in practice. The compressor uses greedy parsing and attempts to find the longest match from the start of the unprocessed text. It will compress the following lyrics of the song Yellow Submarine by The Beatles [18]:

> "We all live in a yellow submarine, yellow submarine, yellow submarine"

The compression will be done by substituting phrases in the text against matches in the dictionary in Table 2.1.

As we can see in Table 2.1, the dictionary is a list of pairs, each containing a phrase and a code, sorted in order of descending length of the phrases. It is sorted to ensure that the compressor always finds the longest match possible. To visualise all the matched characters, we will use underscores to signify the whitespace at the end of a phrase.

The compressor goes from the start of the dictionary and compares the dictionary phrase, "submarine,_", against the text we want to compress. The following phrase is tested if the current does not match, steadily progressing through the dictionary. The search through the dictionary ends when a match is found, and then the corresponding code is added to the compressed result. The matched text is removed from the input, and the procedure is then repeated until all of the input has been compressed. The iterations of compression for the lyrics can be seen in Table 2.2 and yields the compressed string "abdcefgfgfi".

Table 2.1: Dictionary for a short compression example.

Phrase	Code
submarine,_	g
submarine	i
yellow_	f
live_	d
all_	b
We_	a
in_	c
a_	e

Table 2.2: Compression of the text "We all live in a yellow submarine, yellow submarine, yellow submarine" using the dictionary from Table 2.1

Result	Match	Remaining input
		We all live in a yellow submarine, yellow submarine, yellow submarine
a	We_	all live in a yellow submarine, yellow submarine, yellow submarine
ab	all_	live in a yellow submarine, yellow submarine, yellow submarine
abd	live_	in a yellow submarine, yellow submarine, yellow submarine
abdc	in_	a yellow submarine, yellow submarine, yellow submarine
abdce	a_	yellow submarine, yellow submarine, yellow submarine
abdcef	yellow_	submarine, yellow submarine, yellow submarine
abdcefg	submarine,_	yellow submarine, yellow submarine
abdcefgf	yellow_	submarine, yellow submarine
abdcefgfg	submarine,_	yellow submarine
abdcefgfgf	yellow_	submarine
abdcefgfgfi	submarine	

2.2.1 Compressor

This simple static dictionary compressor consists of two main functions, `compress_main` and `decompress_main`. First, we have `compress_main`, this function takes in a string s and checks if we get the same result after decompressing the compressed value of s. If it does, we return the compressed value of s with the prefix "Compressed: " to indicate success. Otherwise, we return the original string s with the prefix "Uncompressed: ". These prefixes are necessary for the decompressor in order for it to know what do to with the input it is given.

```
compress_main s ≝
  if decompress (compress s) = s then
    strcat "Compressed: " (compress s)
  else strcat "Uncompressed: " s
```

The function `decompress_main` reverses the actions done by `compress_main` by checking what prefix the input string s has and then either decompressing the input or returning the input string without the prefix. The `decompress_main` function can be seen here:

```
decompress_main s ≝
  if "Compressed: " ≼ s then
    decompress (drop (strlen "Compressed: ") s)
  else drop (strlen "Uncompressed: ") s
```

These main functions act like wrapper functions with checks for `compress` and `decompress`. Like with the main functions, `decompress` reverses the action done by `compress`. The functions `compress` and `decompress` can be seen below.

```
compress s ≝ tab_sub s lorem_dict

decompress s ≝ tab_sub s (flip_alist lorem_dict)
```

As this is a static dictionary compressor, it uses a table containing all characters and some substrings of an example text as its dictionary. The function `lorem_dict` creates an association list, a list of pairs, where the first value is the uncompressed value and the second is its compressed counterpart. The function is called `lorem_dict` because it uses Lorem ipsum, a standard placeholder text, to create some substrings to include in the dictionary.

The functions `compress` and `decompress` are remarkably similar as they both rely on `tab_sub` to replace substrings in s with matches found in the table/dictionary. The `tab_sub` function calls `find_match` and replaces the match with its compressed counterpart if one is found by `find_match`. Then it drops the match from the string we look for matches in, concatenates the replacement and repeats until we have an empty string. If a match was not found, the compression is cancelled prematurely. These functions can be found in Figure 2.1. In `decompress`, the order of the pairs in the dictionary is flipped to find matches based on the compressed data and replace them with the uncompressed counterpart.

```
tab_sub s tab ≝
 if s = "" then ""
 else
  let (match, value) = find_match s tab
  in
   if match = "" then ""
   else strcat value (tab_sub (drop (strlen match) s) tab)

find_match s [] ≝ ("", [])
find_match s ((k, v) :: ts) ≝
 if k ≼ s then (k, v) else find_match s ts

lorem_dict ≝
 create_fixed_dict
  "Lorem ipsum dolor sit amet, consectetur adipiscing elit."

flip_alist [] ≝ []
flip_alist ((x, y) :: t) ≝ (y, x) :: flip_alist t
```

Figure 2.1: Functions used by compress and decompress and their dependencies.

2.2.2 Dictionary

There are requirements that the dictionary has to meet for the compression algorithm to work. When the dictionary meets these requirements, we say that the table is well-formed. The requirements are:

- the table contains at least a value for each of the 256 characters,
- the codes are prefix-free and,
- the table is sorted after matches in descending order.

These requirements mean that regardless of what text is used as input for the compression function, we can compress it if the text consists of the basic 256 characters. The prefix-free codes make decompression easier as the codes are guaranteed not to contain any ambiguity. For example, if a is encoded as 10, b as 100, and c as 0, how do we know if 100 is decoded as ac or b? We call the code for a a prefix to the code for b. That codes are prefix-free means that these situations do not occur as all codes are distinguishable from each other. Furthermore, the dictionary is sorted in descending order, which means that the longest matches will be found first. The order is vital due to the greedy parsing strategy, as higher compression ratios are reached if longer matches are compressed.

The compressor uses an association list as its dictionary with which it decides what each match should be exchanged to. The base of the association list consists of all the basic 256 characters. It is thereby guaranteed that anything containing these base characters can be compressed.

In addition to this, other substrings are generated from a string that acts as the model for the dictionary. These substrings are then entered into the dictionary as matches, alongside the base 256 characters. This implementation enters the first sentence of Lorem ipsum to `create_fixed_dict`, as the model string.

The function `extract_keys` is used to create the base keys and keys based on s. It then sorts them on the longest strings and zips them with codes with fixed length, which the function `gen_fix_codes` create.

$$
\begin{aligned}
&\mathsf{create\_fixed\_dict}\ s \stackrel{\mathsf{def}}{=} \\
&\quad \mathsf{let}\ keys\ =\ \mathsf{qsort}\ (\lambda\, x\ y.\ \mathsf{strlen}\ y\ <\ \mathsf{strlen}\ x)\ (\mathsf{extract\_keys}\ s) \\
&\quad \mathsf{in} \\
&\qquad \mathsf{zip}\ (keys, \mathsf{gen\_fix\_codes}\ (\mathsf{length}\ keys)) \\
\\
&\mathsf{extract\_keys}\ s\ \stackrel{\mathsf{def}}{=}\ \mathsf{base\_keys}\ +\!\!+\ \mathsf{extract\_substrings\_n}\ s\ 6
\end{aligned}
$$

The functions `extract_substrings_n` find all substrings up to length 6, and this could of course be changed to some other integer. This collection of generated substrings represents the majority of our dictionary and enables the compressor to find other matches than just the basic characters.

The dictionary only enables matches for the substrings found in the given input. If the text to be compressed is very different from the input used to generate the

dictionary, then the level of compression may be low compared to a more similar text.

2.3 Problems

This example takes the form of a static dictionary compressor, which means that the dictionary is not adapted to the input. Using a static dictionary can work well in cases where the compressor is supposed to work on specific kinds of input since the model can be adapted for those inputs. However, it does not work well for general compression as the performance of the compressor is highly varying and often not as good as an adaptive dictionary.

A significant drawback of the `decompress_main` function is that it will think that any string starting with "Compressed: " can be decompressed. The `decompress` function can process the string, but in practice, there will either be no matches in the table used, or the matches will create a new string that is not legible.

The implementation of this compressor is naive and as a result has poor performance. The definition `find_match` uses `IS_PREFIX`, which has a complexity of $O(m)$, for each key-value pair in the dictionary. In the worst case, `find_match` would have the complexity $O(n^2)$ to decode a single character. The complete algorithm would then be even more complex, possibly having cubic complexity.

3

Deflate

This chapter explains the background necessary to understand Deflate as well as how LZ compression and Huffman encoding are incorporated. Then our implementation of a Deflate-like algorithm is demonstrated.

3.1 Background

Deflate [1] is a well-known file format specification from 1996 that, among other things, explains how to use LZ compression and Huffman encoding together to achieve compression.

In Deflate [1], the compression algorithm does not have as high requirements as the decompression algorithm. A compression algorithm is compliant with Deflate as long as it stays within the boundaries defined by the file format. It could use one of three possible compression modes or a sliding window of half the standard size. However, a decompression algorithm needs to be able to decompress the output of any compression algorithm that yields a correct Deflate file format.

Deflate specifies a file format and is, as such, particular about the ordering of bits and bytes. All bytes part of a multi-byte number are stored with the least-significant byte first. For example, the following two bytes `{0000 0010}` `{0000 1000}` represent the value 520 in binary, the first byte represents the number 512 and the second byte represents the number 8. These two bytes are stored with the second byte first according to Deflate, i.e. `{0000 1000}` `{0000 0010}`. Bits are ordered in both most-significant and least-significant bit first, depending on what the bits represent. If the bits represent a prefix-free code, then most-significant bit first is used; otherwise, least-significant bit first is used.

3.1.1 Compression in Deflate

The Deflate file format is closely related to LZ compression [1]. LZ compression uses substitution with backreferences [13], also called length-distance pairs, to compress text. These backreferences refer to where a substring occurred earlier in the text. Depending on what LZ algorithm is used, these can take different forms. An example is the text “Hello! Hi! Hello!” that would be compressed to “Hello! Hi! (6,11)”. Here “(6,11)” is a backreference where 6 states the length of the substituted phrase

and 11 the distance back to where the phrase starts. The result of a successful LZ compression is a list of text literals and backreferences.

The result from the compression is represented using two alphabets [1], one for the literals and lengths and the other for the distances. The literal/length alphabet spans the interval 0-287, where 0-255 represents the literal byte values, 256 is a reserved stop signal (`End of block`), and 257-285 represents length codes. The two last codes, 286 and 287, are unused. The distance alphabet spans the interval 0-29.

In Deflate, the length of a substituted phrase may not exceed 258, and the distance may not be longer than 32768 [1]; to represent these values with few codes, Deflate has a solution using additional bits. There are 29 length codes used to represent lengths between 3 and 258. For this to work, 0-5 additional bits are added after the length code. There are 30 distance codes used to represent distances between 1 and 32768. For the distance codes, 0-13 additional bits are added. In the RFC [1] there is one table for the lengths and one for the distances, these tables can be used to translate lengths and distances to their codes with additional bits.

The backreference "(6,11)" is converted using the tables in Deflate the following way. The length 6 is represented by the length code 260 using 0 extra bits, resulting in "260()" where the parentheses represent the extra bits. The distance of 11 is represented by the distance code 6 with 2 extra bits. The distance code 6 with two extra bits corresponds to the span 9-12, which means that the distance 9 is represented by "6(00)", 10 by "6(10)", 11 by "6(01)" and 12 by "6(11)". Note that the additional bits are stored with the least-significant bit first. The final result after converting the backreference "(6,11)" is "260() 6(01)".

The literals are converted to numbers and the new representation of the string "Hello! Hi! Hello!" is the following: "72,101,108,108,111,33,32,72,105,33,32,260(),6(01)".

3.1.2 Huffman encoding in Deflate

Huffman encoding is a method to construct minimal prefix-free codes to represent a set of elements [19]. From the text "Hello! Hi! (6,11)" the Huffman tree seen in Figure 3.1 can be generated; this tree represents the literal/length alphabet. The distance Huffman tree only has one value to represent, the distance 11, and therefore only has to use one bit to encode the value, {11:1}. The numbers are represented as characters in the tree to avoid misunderstandings, e.g. (*e*:1100) instead of the numerical representation (101:1100). The tree can then be used to encode and decode, in this case exchanging "H" for 00, "e" for 1100 etc. Important to note is that Deflate does not allow codes with a length longer than 15 or shorter than 1 [1].

The encodings that are found in the tree in Figure 3.1, enable us to store the string in bit format. The result of encoding "Hello! Hi! (6,11)" is:

00 1100 01 01 1110 101 100 00 1101 101 100 1111 101

The last 3 bits of the sequence, 101, are for the distance where the first 1 is the distance code and the two following bits are the two extra bits.

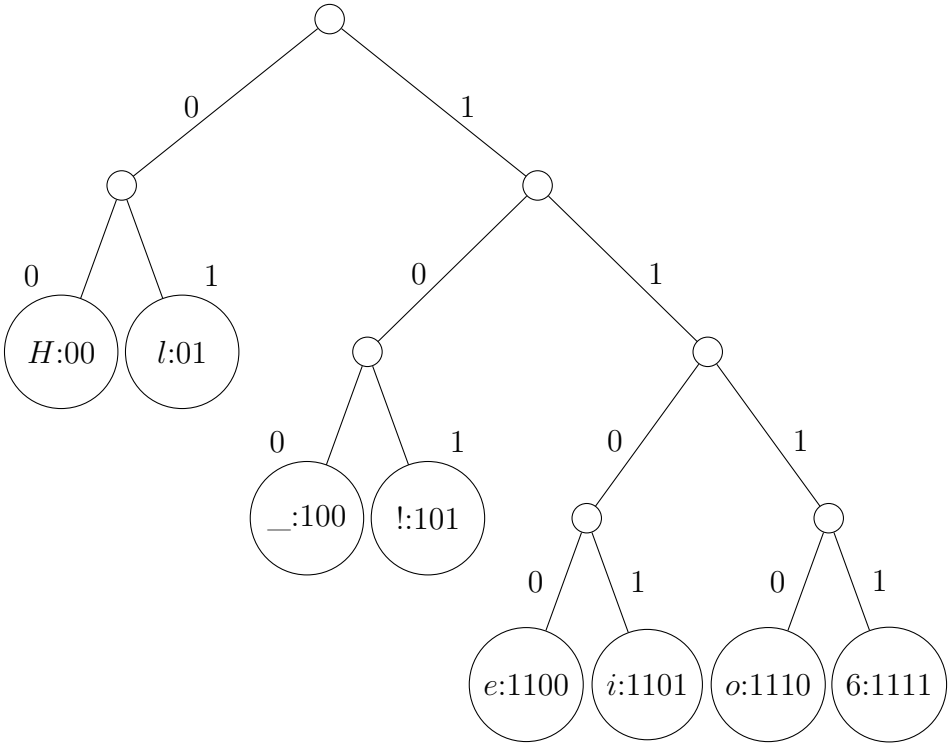

Figure 3.1: A Huffman tree containing the encoding for the characters and lengths in "Hello! Hi! (6,11)".

The Huffman tree needs to be stored with the message to be able to decode it later, this creates certain requirements for the creation of the tree, these are explained further in Chapter 5. When the requirements are met, only the code lengths from the Huffman tree need to be stored as they are sufficient in themselves to restore the Huffman tree. An important remark is that if an element in the alphabet is not included in the tree then it is denoted with the code length 0. Furthermore, if elements at the end of the alphabet have the code length 0 then they are excluded from the list as they can be inferred. Using this method, the Huffman trees for two alphabets used to encode LZ compression would at most require 315 code lengths, which corresponds to the literal/length and distance alphabets concatenated.

For example, in the excerpt "...40004002..." from the code length alphabet built from the example in Figure 3.1, the first 4 represents the length for the code representing e, and the following zeroes represent f, g and h which are not present in the Huffman tree. The next 4 represents i, and the two following zeroes j and k then the 2 represents l. If the values after 2 are all zero then all of them would be discarded.

3.1.3 Block format

The Deflate compression algorithm outputs a sequence of blocks, where each block has a block header followed by the block's data, as seen in Figure 3.2. There is no upper limit for the size of a compressed block, however, an uncompressed block has an upper limit of 65,535 bytes.

Each block has a block header that consists of two parts, BFINAL and BTYPE [1]. The first bit of the block header, BFINAL, signifies if the block is the final block. The next two bits, BTYPE, indicate what type of compression was used to compress the data. Then depending on the type, the block data may take different forms, as they need to contain different types of information.

potential block	BFINAL	BTYPE	the block data	potential block
...	1 bit	2 bits	varies	...

Figure 3.2: The general structure of a block created by a Deflate compressor.

The BTYPE bits can have the following values; 00, 01, 10 or 11. These codes correspond to an uncompressed block (00), a block compressed using a fixed Huffman tree (01) and a block compressed using a dynamic Huffman tree (10) [1]. A fixed tree refers to a predefined tree while a dynamic tree refers to a tree constructed from the input it is to be applied on. The last code, 11, is reserved, if it occurs there may have been corruption of the data or some other error may have occurred.

3.1.3.1 Uncompressed block

In the case of an uncompressed block, as seen in Figure 3.3, the bits between BTYPE and the next byte are ignored. The number of ignored bits can vary, since the previous block does not need to end on an aligned byte. The two next bytes, LEN, entails the number of data bytes stored in the block; and the block after that, NLEN, is the bit-complement to LEN and acts as a checksum. After LEN and NLEN comes LEN number of bytes containing the uncompressed data.

ignored	LEN	NLEN	uncompressed data
varies	2 bytes	2 bytes	LEN bytes

Figure 3.3: The block format for an uncompressed block (BTYPE=00).

3.1.3.2 Fixed Huffman block

A block which contains data compressed using the fixed Huffman tree has the BFINAL and BTYPE flags and then the compressed data follows. The length of the compressed data is unknown since it is compressed using Huffman trees and symbols may be compressed with a variable number of bits. To indicate the end of the block, the value 256 is appended to the end of the data, and when encountered the next block can be read. In the case of the fixed Huffman block, we do not need to transmit the alphabets as the fixed Huffman trees never change, and the decompressor can generate them. The structure can be seen in Figure 3.4.

compressed data	End of block
varies	varies

Figure 3.4: The internal block format for the fixed Huffman block (BTYPE=01).

3.1.3.3 Dynamic Huffman block

A block which contains data compressed using dynamic Huffman trees has the most complicated structure of the compression types. The structure of the block is visualised in Figure 3.5. The reason it is more complicated is that the Huffman trees need to be transmitted along with the compressed data in contrast to the uncompressed and fixed Huffman blocks.

`NLIT`	`NDIST`	`NCLEN`	`CLEN alphabet`	`Encoded code lengths`	`Compressed data`	`End of block`
5 bits	5 bits	4 bits	3*`NCLEN` bits	varies	varies	varies

Figure 3.5: The internal block format for the dynamic Huffman block (BTYPE=10).

Two Huffman trees are used, one for the literal/length alphabet and one for the distance alphabet. These are stored as lists of code lengths after each other with the distance alphabet placed last. The literal/length alphabet has a minimum length of 257, all literals and the stop signal, and `NLIT` indicates how many length codes are used, i.e. how long is the list of code lengths. Similarly, `NDIST` indicates how many distance codes are used. This is relevant since code lengths of 0 at the end of both can be discarded.

Deflate [1] has a specialised Run-Length Encoding (RLE) used to compress the combined list of code lengths. In general, RLE compresses by making use of repetitions; for example, the sequence "000012333333333" could be represented as "('0',4)('1',1)('2',1)('3',9)". The Deflate RLE alphabet spans the interval 0-18, where 0-15 represent the code lengths 0-15 and the codes 16-18 represent repetitions. Code 16 uses two additional bits to repeat the previous code length 3-6 times. The codes 17 and 18 are used to encode repeating zeroes where 17 encodes 3-10 repeating zeroes using three additional bits and 18 encodes 11-138 zeroes using seven additional bits. Using Deflate's RLE, the code lengths "000012333333333" are converted to "17(100) 1 2 3 16(01) 16(00)" where the parentheses indicate the additional bits.

A Huffman tree for the RLE alphabet is constructed and then used to encode the compressed list of code lengths, resulting in the part of the block corresponding to `Encoded code lengths` (Figure 3.5). The Huffman tree is stored in a special order specified by Deflate [1]:

16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

The reason for this order is that code lengths closer to the end are more likely to have the code length zero. If that is the case they can be left out, which saves space. The value `NCLEN` then indicates how long the list of code lengths is, i.e. 18 minus how many code lengths could be left out. The list of code lengths, `CLEN alphabet`, stores each code length using 3 bits.

Similarly to the block with fixed Huffman trees, the compressed data is placed last with its end indicated by `End of block`.

3.2 Implementation

First, we present how the three different block types are encoded into binary format. Afterwards, the finer details of the encoding are demonstrated. Then we demonstrate our Deflate decoder in a similar manner. Lastly, the main functions and the simple theorem are described.

3.2.1 Encoding Deflate

All block-level logic when encoding the input as binary is handled by `encode_block`, such as the input data and the compression mode of each block. While the RFC [1] states that blocks may be of arbitrary size we decided to cap the length to 16,383 bytes, similar to the ZLib implementation of Deflate [20]. If all of the input did not fit inside one block, then `encode_block` (Figure 3.6) is called recursively with the remainder of the input and adds a new block after the current.

Three conditions are inspected when determining the compression mode of the block: was there any compression performed, are the codes from the Huffman tree too long and how long is the input for the block. To check these three conditions, the block input is compressed using `LZSS_compress` which performs compression using backreferences (Section 4.1.1.1). Then two Huffman trees are generated, to encompass all unique elements in the compressed result, along with their respective lexicographically sorted list of code lengths. Lastly, the two trees are compressed and encoded using a third Huffman tree. The third tree is generated to encode the two first trees and Deflate has set an upper limit of seven for the length of its codes.

If no compression was done then the uncompressed mode is used. Fixed compression is used when the input is shorter and we set all messages shorter than 100 characters to be compressed using the fixed compression mode. Dynamic compression has the restriction that no Huffman code may have a length greater than 15, if this is the case then fixed is used. If none of the previous conditions are triggered, then the block is compressed using dynamic compression.

```
encode_block "" ≝ []
encode_block (string v v1) ≝
 let block_input = take 16383 (string v v1);
 s' = drop 16383 (string v v1);
 lzList = LZSS_compress block_input;
 (lenList, distList) = split_len_dist lzList [] [];
 (len_tree, len_alph) = unique_huff_tree (256 :: lenList);
 (dist_tree, dist_alph) = unique_huff_tree distList;
 longest_code1 =
  foldl (λ e x. if e < x then x else e) 0
   (len_alph ++ dist_alph);
 (lendist_alph_enc, clen_tree, clen_alph, _) =
  encode_rle (len_alph ++ dist_alph);
 longest_code2 =
  foldl (λ e x. if e < x then x else e) 0 clen_alph;
 BFINAL = if s' = "" then [T] else [F];
 enc =
  if length dist_tree = 0 then
   [F; F] ++ encode_uncompressed lenList
  else if
   strlen block_input < 100 ∨ 15 < longest_code1 ∨
   7 < longest_code2
  then
   [F; T] ++ encode_fixed lzList
  else
   [T; F] ++
   encode_dynamic lzList len_tree len_alph dist_tree
    dist_alph clen_alph lendist_alph_enc
 in
  BFINAL ++ enc ++ encode_block s'
```

Figure 3.6: Definition of `encode_block`.

In the case that no compression was possible the input data is simply copied into the block. The length of the copied data resides in `LEN` with `NLEN` validating the value of `LEN`. The most complicated part of this mode is that 5 bits are to be appended merely to fill up the byte the block header began.

```
encode_uncompressed s ≝
 let rest_of_first_byte = [T; T; T; T; T];
 LEN = pad0 16 (tn2bl (length s));
 NLEN = map (λ b. if b ⟺ T then F else T) LEN;
 input = flat (map (λ n. pad0 8 (tn2bl n)) (reverse s))
 in
  rest_of_first_byte ++ LEN ++ NLEN ++ input
```

The fixed compression mode makes use of predefined Huffman trees and as such only returns the compressed data ending with the end of block signal.

```
encode_fixed lzList ≝
 deflate_encoding lzList fixed_len_tree fixed_dist_tree ++
 encode_single_huff_val fixed_len_tree 256
```

In the function `enocde_dynamic` the arguments *len_alph* and *dist_alph*, which are alternative representations of the Huffman tree, are appended together. The new combined representation is encoded using RLE, `encode_rle`, with a new Huffman tree, *clen_tree* and *clen_alph*. The new tree is then encoded into binary using `encode_clen_alph`. The lists of code lengths: *len_alph, dist_alph* and *clen_alph*, are of variable length since zeroes at the end could be trimmed off. To know how many code lengths to read during decoding, each list's length is recorded in the variables *NLIT*, *NDIST* and *NCLEN*. All of these values are then appended to each other as per the structure specified in Figure 3.5.

```
encode_dynamic lzList len_tree len_alph dist_tree dist_alph
 clen_alph lendist_alph_enc ≝
 let (NCLEN_num, CLEN_bits) = encode_clen_alph clen_alph;
 NLIT = pad0 5 (tn2bl (max (length len_alph) 257 − 257));
 NDIST = pad0 5 (tn2bl (length dist_alph − 1));
 NCLEN = pad0 4 (tn2bl (NCLEN_num − 4));
 header_bits = NLIT ++ NDIST ++ NCLEN
 in
  header_bits ++ CLEN_bits ++ lendist_alph_enc ++
  deflate_encoding lzList len_tree dist_tree ++
  encode_single_huff_val len_tree 256
```

3.2.1.1 Encoding the compressed data

The only difference between the encoding of the data in fixed and dynamic is the use of either predefined trees or trees generated from the input.

Each element, either a literal or a backreference, is encoded sequentially. In the case of a literal, then the value of the literal is extracted and encoded using the provided

literal/length Huffman tree. For a backreference, the length of the match and the distance back are encoded on their own using the two trees and tables provided by the RFC [1].

```
encode_LZSS lz len_tree dist_tree ≝
 case lz of
  Lit c ⇒ encode_single_huff_val len_tree (ORD c)
 | LenDist (len, dist) ⇒
  let enc_len =
   encode_LZSS_table len find_level_in_len_table len_tree;
  enc_dist =
   encode_LZSS_table dist find_level_in_dist_table
    dist_tree
  in
   enc_len ++ enc_dist
```

3.2.1.2 Transmitting the trees for decoding

To transmit the tree, the code length of each element in the alphabet is calculated. This includes all elements, even those not present in the current message. The list of code lengths can then be used to reconstruct a tree representing the elements and their respective code length.

The code lengths are sorted lexicographically to ensure that each length can be coupled to its corresponding element. To ensure that we generate code length for all elements, `gen_zero_codes` is used to instantiate a base list. The list has all elements in the alphabet and each element is associated with an empty list signifying its code. This list is sorted lexicographically upon construction.

The function `fill_assoc_list` is then used to fill out the base list with values from a Huffman tree. If an element occurs in the Huffman tree then the code in the base list is updated with the code found in the Huffman tree. The wrapper function `complete_assoc_list` sets up `gen_zero_codes` and `fill_assoc_list` to create a list with all elements in the alphabets and their respective codes for this encoded message.

```
complete_assoc_list ls ≝
 let max_val = foldl (λ a (b, _). if a < b then b else a) 0 ls;
 gs = gen_zero_codes max_val;
 as = qsort (λ (a, _) (b, _). a < b) ls
 in
  fill_assoc_list gs as
```

Lastly, `len_from_codes` calculates the length of each code. The list of code lengths

can now be added to the encoded message.

```
len_from_codes [] ≝ []
len_from_codes ((n, bl) :: ns) ≝ length bl :: len_from_codes ns

all_lens as ≝ len_from_codes (complete_assoc_list as)
```

3.2.1.3 Run-length encoding (RLE)

RLE is applied since the list of code lengths is prone to repeating values. First, repeating values are found and translated to Deflate's format. A Huffman tree, `nclen_tree`, is generated from the values and is then used to encode the repeating values.

```
encode_rle [] ≝ ([], [], [], [])
encode_rle (l :: ls) ≝
 let repeat = find_repeat ls l 1;
 (repeat_tree, repeat_alph) = unique_huff_tree (map fst repeat);
 enc = encode_rle_aux repeat repeat_tree
 in
  (enc, repeat_tree, repeat_alph, repeat)
```

3.2.1.4 Transmitting the code length alphabet

The first step in encoding the Huffman `nclen_tree` into binary is to rearrange its list of code lengths, `nclen_alph`, so that we then can trim off potential zeroes found at the end. Then we count the length of the list and store the result in `NCLEN`. Lastly, each value in `nclen_alph` is converted to a 3-bit value.

```
encode_clen_alph alph ≝
 let alph =
  trim_zeroes_end (encode_clen_pos (PAD_RIGHT 0 19 alph));
 CLEN_bits = flat (map (λ a. pad0 3 (tn2bl a)) alph);
 NCLEN = length alph
 in
  (NCLEN, CLEN_bits)
```

3.2.2 Decoding Deflate

Decoding a block starts with the decoder reading the block header, `BFINAL` and `BTYPE`, to understand if this is the last block to read as well as what compression

mode to use for the current block.

$$
\begin{array}{l}
\textsf{decode\_block}\ (b_0 :: b_1 :: b_2 :: bl) \stackrel{\text{def}}{=} \\
\quad \textsf{let}\ (enc, drop\_bits) = \\
\quad\quad \textsf{if}\ (b_1 \iff \textsf{F}) \wedge (b_2 \iff \textsf{F})\ \textsf{then decode\_uncompressed}\ bl \\
\quad\quad \textsf{else if}\ (b_1 \iff \textsf{F}) \wedge (b_2 \iff \textsf{T})\ \textsf{then decode\_fixed}\ bl \\
\quad\quad \textsf{else if}\ (b_1 \iff \textsf{T}) \wedge (b_2 \iff \textsf{F})\ \textsf{then decode\_dynamic}\ bl \\
\quad\quad \textsf{else}\ (\text{“”}, 0) \\
\quad \textsf{in} \\
\quad\quad \textsf{if}\ b_0 \iff \textsf{T}\ \textsf{then}\ enc \\
\quad\quad \textsf{else strcat}\ enc\ (\textsf{decode\_block}\ (\textsf{drop}\ drop\_bits\ bl)) \\
\textsf{decode\_block}\ [] \stackrel{\text{def}}{=} \text{“”} \\
\textsf{decode\_block}\ [v] \stackrel{\text{def}}{=} \text{“”} \\
\textsf{decode\_block}\ [v;\ v_4] \stackrel{\text{def}}{=} \text{“”}
\end{array}
$$

In the case that uncompressed mode is used, the next 5 bits are discarded. Then the next four bytes are read as two two-byte numbers, `LEN` and `NLEN`, which tell us how many characters have been stored in the block. That many characters are then copied to the output and the block is correctly decoded.

$$
\begin{array}{l}
\textsf{decode\_uncompressed}\ bl \stackrel{\text{def}}{=} \\
\quad \textsf{let}\ bl' = \textsf{drop}\ 5\ bl; \\
\quad LEN = \textsf{TBL2N}\ (\textsf{take}\ 16\ bl'); \\
\quad bl'' = \textsf{drop}\ 16\ bl'; \\
\quad NLEN = \textsf{TBL2N}\ (\textsf{take}\ 16\ bl'') \\
\quad \textsf{in} \\
\quad\quad (\textsf{read\_literals}\ (\textsf{drop}\ 16\ bl'')\ LEN, 5 + 2 \times 16 + LEN \times 8)
\end{array}
$$

For blocks using fixed compression, the decoding of literals and backreferences begin directly after the block header is read. The first step makes use of the predefined Huffman trees to determine if the first binary-encoded value is a literal or a backreference. If it is a literal then we drop the match from the input data and start over from the first step. However, if what was found is a backreference then we may have to read some more bits to find the correct length value. Then a similar process is performed for the distance of the backreference. The decoding ends when the end of block value is found.

$$
\begin{array}{l}
\textsf{decode\_fixed}\ bl \stackrel{\text{def}}{=} \\
\quad \textsf{let}\ (lzList, drop\_bits) = \\
\quad\quad \textsf{deflate\_decoding}\ bl\ \textsf{fixed\_len\_tree fixed\_dist\_tree}\ []\ 0; \\
\quad res = \textsf{LZSS\_decompress}\ lzList \\
\quad \textsf{in} \\
\quad\quad (res, drop\_bits)
\end{array}
$$

Dynamic blocks are yet again the most complicated of the lot. First the values `NLIT`, `NDIST` and `NCLEN` are read with 5, 5 and 4 bits respectively. Then `CLEN alphabet` is read by reading 3-bit numbers `NCLEN` number of times. The binary encoding is reversed and the Huffman tree for the code lengths, `nclen_tree`, is restored.

The tree and the sum of `NLIT` and `NDIST` are provided to the run-length decoder so that it can decode the literal/length alphabet as well as the following distance alphabet. From the two alphabets, the Huffman trees `len_tree` and `dist_tree` are recreated. From here, the process is identical to that of decoding a fixed block with the exception that dynamically created prefix trees are used instead of predefined ones.

$$
\begin{array}{l}
\mathsf{decode\_dynamic}\ bl \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{let}\ (NLIT', NDIST', NCLEN', bl) = \mathsf{read\_dyn\_header}\ bl; \\
\quad (clen\_tree', bits) = \mathsf{decode\_clen}\ bl\ NCLEN'; \\
\quad bl' = \mathsf{drop}\ bits\ bl; \\
\quad (len\_dist\_alph', bits') = \\
\quad\quad \mathsf{decode\_rle}\ bl'\ (NLIT' + NDIST')\ clen\_tree'; \\
\quad bl'' = \mathsf{drop}\ bits'\ bl'; \\
\quad len\_alph' = \mathsf{take}\ NLIT'\ len\_dist\_alph'; \\
\quad dist\_alph' = \mathsf{drop}\ NLIT'\ len\_dist\_alph'; \\
\quad len\_tree' = \mathsf{canonical\_codes}\ len\_alph'; \\
\quad dist\_tree' = \mathsf{canonical\_codes}\ dist\_alph'; \\
\quad (lzList', drop\_bits'') = \\
\quad\quad \mathsf{deflate\_decoding}\ bl''\ len\_tree'\ dist\_tree'\ []\ 0; \\
\quad res = \mathsf{LZSS\_decompress}\ lzList' \\
\quad \mathsf{in} \\
\quad\quad (res, 5 + 5 + 4 + bits + bits' + drop\_bits'')
\end{array}
$$

3.2.3 Main compression functions

In our implementation, we use a list of booleans to represent bits. This representation only works within our algorithm and cannot be used as input or output in the binary. For this reason, two wrapper definitions, `deflate_encode` and `deflate_decode`, were defined which convert a list of booleans to a string using `bl2s` and `s2bl`.

$$\mathsf{deflate\_encode}\ s \stackrel{\mathsf{def}}{=} \mathsf{bl2s}\ (\mathsf{encode\_block}\ s)$$

$$\mathsf{deflate\_decode}\ s \stackrel{\mathsf{def}}{=} \mathsf{decode\_block}\ (\mathsf{s2bl}\ s)$$

We employ the same structure for our main functions as used for the simple compressor in Chapter 2. The main drawback with this is that the result from the Deflate encoding will have the prefix *"Compressed: "* and thus not be Deflate compliant.

$$
\begin{array}{l}
\mathsf{deflate\_encode\_main}\ s \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if}\ \mathsf{deflate\_decode}\ (\mathsf{deflate\_encode}\ s) = s\ \mathsf{then} \\
\quad\quad \mathsf{strcat}\ \text{“Compressed: ”}\ (\mathsf{deflate\_encode}\ s) \\
\quad \mathsf{else}\ \mathsf{strcat}\ \text{“Uncompressed: ”}\ s \\
\\
\mathsf{deflate\_decode\_main}\ s \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{if}\ \text{“Compressed: ”} \preccurlyeq s\ \mathsf{then} \\
\quad\quad \mathsf{deflate\_decode}\ (\mathsf{drop}\ (\mathsf{strlen}\ \text{“Compressed: ”})\ s) \\
\quad \mathsf{else}\ \mathsf{drop}\ (\mathsf{strlen}\ \text{“Uncompressed: ”})\ s
\end{array}
$$

3.3 Evaluation of Deflate binary

To evaluate how often the compression succeeded, we created a python script that generated random byte arrays of lengths between 100 and 20 000 bytes. These arrays were then sent as input to the `deflate_encode_main` binary. We generated 1 000 random arrays and all inputs resulted in compressed data with the prefix "Compressed:". The random inputs were compressed to roughly 100% of their original size as expected of compressed random data.

To test how well the binary performs in realistic situations which usually have more redundancy, we compressed various texts and files. Furthermore, the same files were compressed with `zip` [3], version 3.0-10 [21], which builds upon Deflate, The result of the tests can be seen in Table 3.1 and Table 3.2 on the next page. The text *report.txt* is our thesis report in .txt format.

Our implementation of Deflate seems to yield compression comparative to that of `zip` for smaller files. However, the execution times were several magnitudes slower.

Table 3.1: Evaluation of the compression ratio for our Deflate encoder binary compared against `zip`.

File	File size	Compression ratio `deflate`	Compression ratio `zip`
Yellow submarine lyrics	1 254 B	32%	42%
ChessBoard.png	1 579 B	22%	30%
Lorem Ipsum 20k letters	20 072 B	34%	31%
Lorem Ipsum 30k letters	30 098 B	32%	30%
Lorem Ipsum 40k letters	40 138 B	33%	29%
rfc1951.txt	36 944 B	34%	30%
eduroam-linux.py	43 289 B	35%	31%
zlib/deflate.c	81 628 B	31%	31%
report.txt	99 701 B	37%	32%

Table 3.2: Evaluation of the execution time for our Deflate encoder binary compared against `zip`.

File	File size	Execution time `deflate`	Execution time `zip`
Yellow submarine lyrics	1 254 B	45ms	2.5ms
ChessBoard.png	1 579 B	47ms	2.3ms
Lorem Ipsum 20k letters	20 072 B	2590ms	3.2ms
Lorem Ipsum 30k letters	30 098 B	4745ms	4.0ms
Lorem Ipsum 40k letters	40 138 B	6354ms	4.6ms
rfc1951.txt	36 944 B	7828ms	4.0ms
eduroam-linux.py	43 289 B	10680ms	3.8ms
zlib/deflate.c	81 628 B	22439ms	5.4ms
report.txt	99 701 B	36934ms	8.6ms

4

LZSS

This chapter is about LZSS compression, and here we demonstrate the implementation of our LZSS compression algorithm. The implemented LZSS algorithm is a part of our of Deflate algorithm implementation which was previously introduced in Chapter 3.

4.1 Background

While Deflate refers to LZ77, the differences between LZ77 and LZSS are minor, and LZSS can still perform compression compatible with the Deflate algorithm. One of the differences is that while LZ77 always returns a triplet of distance, length and a character, LZSS instead returns a literal character or a distance length pair. The other change is that LZSS has a lower bound of 3 regarding the length of matches. The lower bound ensures that the length distance pair takes less space than the values the pair encodes.

4.1.1 LZSS by Alexander Cox

Alexander Cox, a MSc student at ANU Canberra Australia, had begun on an implementation of LZSS compression. His work consisted of proofs and definitions covering an LZSS compressor and decompressor as well as a verified Ringbuffer data structure. In his initial implementation of LZSS, Cox used standard lists as the sliding window and later pivoted to using the newly developed Ringbuffer. The change of data structures was still an ongoing process and the Ringbuffer was for that reason never fully integrated.

4.1.1.1 LZSS implementation

Cox's work regarding LZSS consists of a compressor, with the auxiliary functions `matchLength`, `getMatch`, `LZmatch` and `LZinit`, and a decompressor with the helper function `resolveDistLen`.

To understand the compressor, the auxiliary functions must first be understood. The function `matchLength` counts the number of equal elements in two lists, from the start of the lists until the first mismatch. It is then used in `getMatch` which takes two lists and finds the longest segment in the first list which matches the start

of the second list.

In LZSS compression, the first list acts as our dictionary and the second list is the lookahead which has the next characters to be compressed. The return value of `getMatch` is a length-distance pair, *(l,d)*, where l is the length of the match and d is the distance from the start of the list to where the match begins.

The function `LZmatch` takes the pair returned from `getMatch` and formats it as the datatype `LZSS`. The datatype is either the pair representing a backreference or a literal containing the next character in the lookahead. The compressor, `LZcomp`, works by calling `LZmatch` to find matches and iteratively work through text to be compressed.

```
LZcomp s split bufSize lookSize ≝
 if length s ≤ split ∨ s = [] ∨ bufSize = 0 ∨ lookSize = 0 then
  []
 else
  let match =
   LZmatch (take split s) (take lookSize (drop split s));
  len =
   case match of
    None ⇒ 1
   | Some (Lit v1) ⇒ 1
   | Some (LenDist (ml, v4)) ⇒ max ml 1;
  bufDrop = split + len − bufSize;
  recurse =
   LZcomp (drop bufDrop s) (split + len − bufDrop) bufSize
    lookSize
  in
   case match of None ⇒ recurse | Some m ⇒ m :: recurse
```

The decompressor, `LZdecompress`, works by unpacking literals and resolving length-distance pairs. The length-distance pairs are then handled by the function `resolveLenDist`, which extracts the substring that the pair refers back to.

```
LZdecompress de [] ≝ de
LZdecompress de (next :: t) ≝
 let newde =
  case next of
   Lit a ⇒ snoc a de
  | LenDist ld ⇒
   case resolveLenDist de ld of None ⇒ de | Some s ⇒ de ++ s
 in
  LZdecompress newde t
```

$$\textsf{resolveLenDist}\ []\ v_0 \stackrel{\textsf{def}}{=} \textsf{None}$$
$$\textsf{resolveLenDist}\ (v_2 :: v_3)\ (l, d) \stackrel{\textsf{def}}{=}$$
$$\textsf{if length}\ (v_2 :: v_3) < d \vee \textsf{length}\ (v_2 :: v_3) < l \vee d < 1 \vee l < 1\ \textsf{then None}$$
$$\textsf{else Some}\ (\textsf{take}\ l\ (\textsf{drop}\ (\textsf{length}\ (v_2 :: v_3) - d)\ (v_2 :: v_3)))$$

One limitation of the decompressor is that it can only function in direct subsequent application with its counterpart, this also applies to the compressor. This is since neither of them converts the input to strings or other types that can be displayed. Both of the functions either return or take in some form of the internal datatype which holds literals and/or length-distance pairs.

While Cox may not have finished the integration of Ringbuffers in his work, there were some completed functions. Cox had defined the function `matchLengthRB` to match the functionality of `matchLength` using a Ringbuffer instead of lists. It functions by having two indexes, where the first is lower than the second and the second is placed at the split index, where the dictionary ends and the lookahead starts. If the values at the indexes match then they both increment and check the next value, continuing until they do not match or the split index reaches the end.

Upon inspection of his code, we discovered that there were some bugs related to the compressor. When we tested `LZcomp`, we discovered that the literals were encoded incorrectly, which meant that sometimes `LZComp` returned the wrong characters. Furthermore, it sometimes cut the string short and thus did not return a properly compressed output.

4.1.1.2 Ringbuffer implementation

Cox finished large parts of an LZSS implementation using lists and proved it correct. Most literature discussing LZ compression refers to a sliding window that is used as the buffer, and these lists are a naive imitation. Thus, to enable a more performant LZSS algorithm, Cox created a Ringbuffer data structure.

A Ringbuffer, or circular buffer, is a mutable fixed-size array that functions as if the start and end of the array are connected. To achieve this property, it has to store three values; the array buffer, the write index and the current size.

When an element is added to the buffer it is written to the position of the write index, after which the write index increments. If three values are added to the buffer the write index will be moved three spaces forward. When enough elements have been added to fill the buffer, the write index will wrap around to the start of the buffer and begin to overwrite the first elements that were added. These properties make it suitable for handling data streams.

There are two ways to create a Ringbuffer in Cox's implementation. If the contents for the buffer already exists then `rb_of_list` can be used to generate a Ringbuffer of the same size as content used as input. There is also the option to create a

Ringbuffer from scratch using `empty_rb` which takes two arguments, the size and the default value. Size is necessary since the array is a fixed size and the default value is important as it sets the type for the Ringbuffer.

Cox created several utility functions around the Ringbuffer. Most of these define standard behaviour when handling an array-like structure, for example, head, tail and append. There is also a function to get the list of values from a defined ringbuffer, called `list_of_ringBuffer`. This function takes into account the write index and size to extract only the elements that have been added and does not include the default values.

With the exception of functions based on `rbUPDATE` all other definitions have been proven to function as expected by Cox.

4.2 Implementation

Our main contributions to Cox work include integrating the Ringbuffer with the already constructed `LZCompress` method and some minor fixes to get everything working correctly. There was no documentation or notes from when his work stopped so our understanding of the implementation and potential bugs were discovered by reading the code and testing it with various input.

4.2.1 Changes made to Cox's implementation

This section explains the changes we made to Cox's work presented in Section 4.1.1.

When we received Cox's work there were two versions of the main compression function, one which according to comments was "old and broken but has stuff proved about it" and a new implementation with no proofs yet. We ignored the old and broken version and mainly worked with the new definition when attempting to create a functioning LZ compressor.

As previously mentioned, we noticed that there were issues with the implementation by Cox. One issue was that literals were encoded as other characters, which had to do with `LZmatch` using the wrong list. This happened when no match of size three or greater was found, then the first character in the lookahead should be returned as a literal. However, instead of the first character in the lookahead, it returned the first character of the dictionary. For example, if the dictionary is "abbbbb" and the lookahead is "cdd", then the function `LZmatch` returned the literal "a" instead of the literal "c".

The second issue caused the output to be cut short, the reason for this was found in the function `LZcomp`. This had to do with the parameter `split` that kept track of the index which divided the buffer between dictionary and lookahead. It worked as expected until it had filled up the buffer. When the list is filled, some of the first elements have to be dropped which essentially moves the list backwards. This calculation was missing for the split variable and as such it skipped ahead at double the speed and ended the compression prematurely by reaching the end of the list.

4.2.2 LZSS using Ringbuffer

This section demonstrates how the implementation of LZSS using Cox's Ringbuffer data structure works.

The functions `getMatchRB`, `LZmatchRB` and `LZcompRB` were created to mirror the algorithm implemented by Cox with the exception that a Ringbuffer is used instead of lists. The implementation was easier to read than the original since much of the complexity was moved into the functions interacting with the Ringbuffer. Theorems were not used to verify the correctness of our implementation but rather rigorous testing against the list implementation to ensure equal functionality.

We discovered that `matchLengthRB` allowed for recursive backreferences which later caused issues in the decompression. This was enabled due a lack of boundary checks that allowed the first index to enter the lookahead. An example can be seen in Table 4.1 where a match of length 10 is found. However, part of that match depends on the lookahead. When the decompressor attempts to decompress the backreference it depends on itself to find its value and it fails. We decided to add a boundary check to `matchLengthRB` to ensure that recursive backreferences are avoided. Another option would have been to change the decompressor, so that it could handle recursive backreferences.

Table 4.1: Example of compression state to showcase recursive backreferences in `matchLengthRB` where the overline represents the first index and the underline represents the seconds index.

Buffer:	"hello"	Lookahead:	"hellohello"
Ringbuffer:	"$\overline{hello\underline{hello}}\underline{hello}$"		
Resulting match:	(5,10)		

5

Huffman encoding

This Chapter discusses Huffman encodings, namely dynamic and fixed Huffman encoding, and their use in compression. Dynamic Huffman encoding creates a Huffman tree from its input, while fixed Huffman encoding uses a predefined Huffman tree. The implementations of fixed and dynamic Huffman encoding presented in this Chapter are both part of our Deflate algorithm seen in Chapter 3.

5.1 Background

An encoding is a way to represent data, and many encoding schemes have been developed to accommodate different expectations or requirements of the data. Examples include Morse code [22], which has been used within telecommunication to encode the English alphabet with short and long signals, and ASCII, which has been used to store a base set of characters in an 8-bit format [23]. The primary purpose of these encoding schemes was to standardise how to transmit and store messages in their given medium.

5.1.1 Huffman encoding

In 1952, Huffman [19] wrote a paper about creating prefix-free minimum redundancy codes, i.e. codes that encode a text with as few bits as possible. Minimum redundancy refers to how no further bits can be removed and that the codes are optimal in the amount of bits used. Creating the codes starts by calculating probabilities for all characters, depending on the frequency they occur in a text. He then builds a tree bottom-up by combining the two lowest probabilities under a parent node which also gets a value, the combined probability of its children. The parent node is then added to the list of probabilities that can be combined. This process repeats until all characters are present in the tree.

This methodology [19] creates a prefix-free code tree, where the leaf nodes contain the values to be encoded, and each left and right edge has values connected to them, e.g. '0' and '1'. The code for each leaf can be found by concatenating the values found on each edge while traversing the tree from the root node to the leaf node. Due to the tree's structure, no code will be the prefix to any other code. A Huffman tree is a prefix-free tree constructed regarding the frequency with which each character occurs in a text.

We will now go through an example of building a Huffman tree from the text "bbbbbaaaccdde". The occurrences of each character in the text is counted, this yields the following counts; $a = 3$, $b = 5$, $c = 2$, $d = 2$, $e = 1$. From the frequencies, we create the leaf nodes of the tree, sorted in descending order of their counts, as in Figure 5.1.

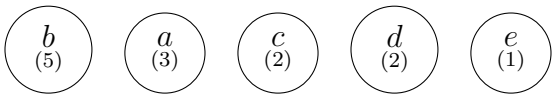

Figure 5.1: Step 1. Sort the leaf nodes is descending order.

From the leaf nodes, we, step by step, combine the two with the lowest count by giving them a common parent node. In the first step, we combine the nodes d and e. The parent node of d and e is given their combined count of 3 and added to the list. Figure 5.2 contains the result of this step.

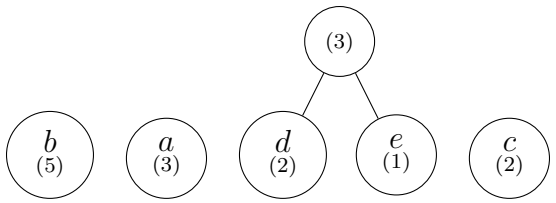

Figure 5.2: Step 2. Combine the leaf nodes with the lowest counts and add the parent to the sorted leaf nodes.

The two lowest counts are now 3 and 2; we, therefore, combine the parent node of d and e with leaf node c, with the result in Figure 5.3.

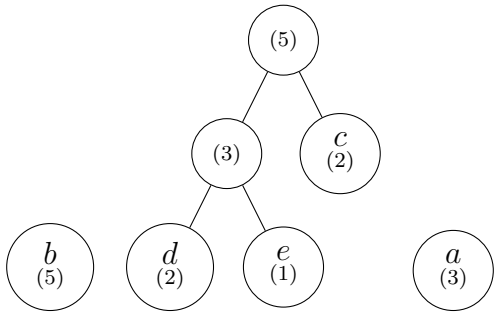

Figure 5.3: Step 3. Combine the two nodes with the lowest counts and add the parent to the sorted nodes.

Now the two lowest counts are 5 and 3, which means leaf node b and leaf node a are combined with a parent that gets the count 8, as seen in Figure 5.4.

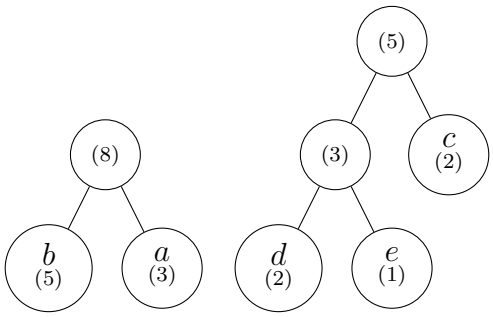

Figure 5.4: Step 4. Combine the two nodes with the lowest counts and add their parent to the sorted nodes.

Lastly, we only have two nodes left to join. The Huffman tree is complete after the two nodes are joined, see Figure 5.5. Each character needs an encoding, and it can be generated by first giving all left paths in the tree the value 0 and all right paths 1. The encoding is then found by traversing the tree from the root node to the leaf. So the encoding for c is 11 when we follow the path to c in the final tree.

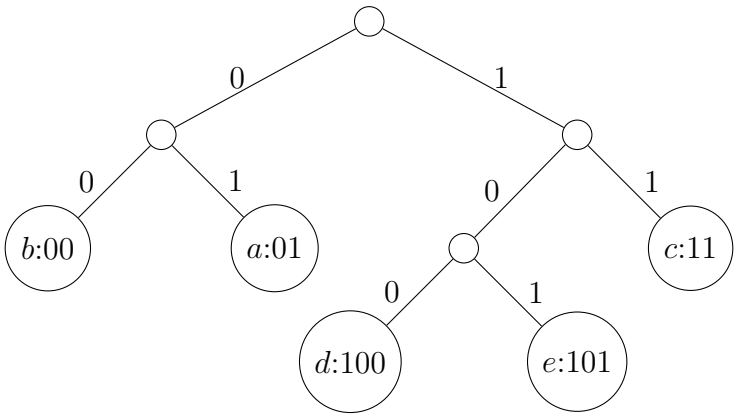

Figure 5.5: The result is a Huffman tree for the text "bbbbbaaaccdde".

5.1.2 Fixed Huffman encoding

Huffman encoding provides a method to generate an optimal encoding for a text given the probabilities of each symbol. However, there may be occasions where an encoding is desired, but creating a dynamic Huffman tree or transmitting the tree is considered unfeasible or counterproductive. In cases such as these, using a predefined Huffman tree, generated from a set of static probabilities, would solve the possible issues with the caveat that it would not necessarily provide an optimal encoding for each text [24].

One such example is in Deflate, where if the message to be compressed is short, then a fixed Huffman tree is used instead of generating a dynamic Huffman tree [1]. It is a compromise between using a more inefficient encoding, i.e. longer codes, against creating and transmitting a tree, i.e. appending a tree structure to the message.

5.1.3 Canonical Huffman codes

The need for canonical Huffman codes comes from the lack of determinism of the Huffman algorithm presented in Section 5.1.1. For example, if c and d where sorted the other way around in Figure 5.1, then the tree in Figure 5.5 would look different and the codes for c and d would change to 100 and 11 respectively.

Canonical Huffman codes place two restrictions upon the Huffman tree [25] to get a deterministic result. The first restriction is that elements with shorter codes are placed to the left of those with longer codes. The second restriction is that among elements with codes of the same length, those that come lexicographically first in the element set are placed to the left. The two restrictions guarantee that from a set of elements and frequencies, only one unique Huffman tree can be constructed. It is also assumed that the edges to the left are represented as 0s, and the edges to the right are represented as 1s.

Canonical Huffman codes are generally created by generating a standard Huffman tree and then adjusting the tree's codes according to the restrictions. This reconstruction procedure can be done by storing the tree and then recreating it from the stored information [1, 25].

To show how it works, consider the example shown in Figure 5.5 where the alphabet is "abcde" with the corresponding code lengths `CL = [2,2,2,3,3]`. The first step is to count how many codes there are of each length, giving us the result of `C = [0,0,3,2]`.

The next step is to find each length's initial value. The value for each length is calculated using the following formula $V_n = 2(V_{n-1} + C_{n-1})$ where V_n is initial value for a code of length n and C_n is the number of codes with length n. The initial value is $V_0 = 0$ which in turn gives the result `V = [0,0,0,6]`.

The last step is to build the canonical Huffman codes using the initial list of code lengths and the list of initial values. The list of code lengths is traversed and each code length, CL_i, finds its value using the following calculation, $CL_i = V_{C_i}$; $V_{C_i} = V_{C_i} + 1$;. The result is the list `[0,1,2,6,7]` which, when combined with the code length list and alphabet, yields a list of codes that can be visualised as the tree seen in Figure 5.6.

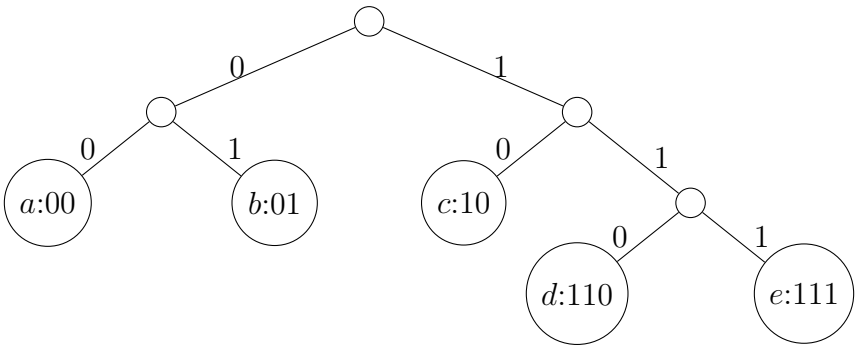

Figure 5.6: A canonical Huffman tree representing the code lengths `[2,2,2,3,3]`.

5.2 Fixed encoding

Deflate uses a fixed Huffman tree for one of its compression modes, and it is specified with a list of code lengths seen in Table 5.1, which is also presented in the RFC [1]. As the code lengths and the alphabet of the fixed Huffman tree are known, it can be generated using the method to create Canonical Huffman codes.

Table 5.1: The predefined code lengths for the fixed Huffman tree used in Deflate [1].

Symbol	Code Length
0 - 143	8
144 - 255	9
256 - 278	7
280 - 287	8

5.2.1 Constructing the tree

There are examples and pseudocode for how to implement Canonical Huffman codes in the RFC [1]. The functions `bl_count` and `next_code` are two of the functions defined in the specification of Deflate, and below we present our HOL4 implementation of them.

Given a list of code lengths, `bl_count` counts the number of occurrences of each code length. Each index larger than 0, corresponds to the length, and the contents at the index how many codes has that length. The function `bl_count` is a wrapper for `bl_count_aux` which performs the recursion to create the list.

$$\begin{aligned}
&\mathsf{bl_count_aux}\ 0\ []\ d \stackrel{\mathsf{def}}{=} [1]\\
&\mathsf{bl_count_aux}\ 0\ (l :: ls)\ d \stackrel{\mathsf{def}}{=} \mathsf{suc}\ l :: ls\\
&\mathsf{bl_count_aux}\ (\mathsf{suc}\ i)\ []\ d \stackrel{\mathsf{def}}{=} d :: \mathsf{bl_count_aux}\ i\ []\ d\\
&\mathsf{bl_count_aux}\ (\mathsf{suc}\ i)\ (l :: ls)\ d \stackrel{\mathsf{def}}{=} l :: \mathsf{bl_count_aux}\ i\ ls\ d
\end{aligned}$$

$$\mathsf{bl\_count}\ bl \stackrel{\mathsf{def}}{=} 0 :: \mathsf{TL}\ (\mathsf{foldl}\ (\lambda\, ls\ b.\ \mathsf{bl\_count\_aux}\ b\ ls\ 0)\ []\ bl)$$

The next step is to calculate the initial values for each length. First, `index_largest_nonzero` finds the longest code, *max*, in order to decide the number of initial values to generate. The function `next_code_aux` calculates the initial value of each length up to the length of *max*. The functions are then wrapped and setup by `next_code`.

$$\begin{aligned}
&\mathsf{next_code_aux}\ []\ prev \stackrel{\mathsf{def}}{=} [prev \times 2]\\
&\mathsf{next_code_aux}\ (x :: xs)\ prev \stackrel{\mathsf{def}}{=}\\
&\quad \mathsf{let}\ new = 2 \times (x + prev)\ \mathsf{in}\ new :: \mathsf{next_code_aux}\ xs\ new
\end{aligned}$$

$$\mathsf{next\_code}\ ls \stackrel{\mathsf{def}}{=} 0 :: \mathsf{next\_code\_aux}\ ls\ 0$$

In the implementation, bits are represented using booleans. The following functions convert between lists of booleans and natural numbers. The function `tbl2n` converts a list of booleans into a number, with the expectation that the most significant bit is the rightmost bit in the bool list, i.e. least-significant bit first. The function `TBL2N` is a wrapper that reverses the input for `tbl2n` and can therefore handle bool lists with most-significant bit first. A similar story can be said about `inv_tbl2n` and `TN2BL` which convert a number into a bool list.

Finally, `pad0` is used to pad codes with zeroes to reach their expected length. For example, if a code with length 5 and value 3 is to be converted into a code then `TN2BL` would return `[TT]` and `pad0` would be used to get the code with the correct length `[FFFTT]`.

The canonical codes are then calculated in `canonical_codes_aux`. Provided the initial list of code lengths representing the tree and the initial values for the different lengths, the code lengths are transformed into canonical codes using the formula discussed in 5.1.3. The code lengths and initial values are provided by `canonical_codes` which act as a wrapper function for the entire process.

```
canonical_codes_aux [] n nc ≝ []
canonical_codes_aux (0 :: ls) n nc ≝
 canonical_codes_aux ls (suc n) nc
canonical_codes_aux (suc v6 :: ls) n nc ≝
 case llokup nc (suc v6) of
  None ⇒ []
 | Some code ⇒
  (n, pad0 (suc v6) (tn2bl code)) ::
   canonical_codes_aux ls (suc n) (lupdate (suc code) (suc v6) nc)

canonical_codes ls ≝
 let nc = next_code (bl_count ls)
 in
  canonical_codes_aux ls 0 nc
```

5.3 Dynamic encoding

This section will explain the implementation of dynamic Huffman encoding. We describe how we implemented the Huffman algorithm and then how to transmit the constructed Huffman trees alongside the message. Then we demonstrate problems and solutions regarding how to create and transmit trees.

5.3.1 Building a tree

Building a Huffman tree starts by counting the frequency of each element. These frequencies are calculated by the function `get_freq` which expects a list of numbers as input. The reason for this type constraint is that the Deflate implementation converts characters to their numeral representation, and the lengths and distances are

also represented as numbers. The function `get_frequencies` is a wrapper function that sets up default parameter values before calling `get_freq`, and then returns a list of tuples containing the element and its frequency.

To build and handle a tree efficiently, we needed a new datatype. The datatype for the tree follows the structure of a standard binary tree and can be seen below.

$$\alpha\ \texttt{Tree} \ = \ \mathsf{Empty} \mid \mathsf{Leaf}\ \alpha \mid \mathsf{Node}\ (\alpha\ \texttt{Tree})\ (\alpha\ \texttt{Tree})$$

The next step is to transform the elements, in the list we got from `get_frequencies`, into Leaf nodes using `convert_frequencies`. When this is done, the list can be given as input to `create_tree`. Now the process is identical to the process described in the example in Section 5.1.1, i.e. the function `create_tree` combines the two elements with the lowest frequency under a parent node and does so until one node, the root, is left. The list is kept sorted during the process using `sort_frequencies`.

There are two special cases when creating a tree that are not handled by combining nodes. These are when the text we want to encode only has one element or is empty. In former case, it is encapsulated in a `Node` with an `Empty` to give the element a single bit code. The latter case, an empty input, results in an `Empty` tree. Both of these cases can easily be seen in the `create_tree` function below.

$$\begin{array}{l}
\mathsf{create_tree}\ ls \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{case}\ ls\ \mathsf{of} \\
\quad\quad [] \ \Rightarrow\ \mathsf{Empty} \\
\quad \mid [(\mathsf{Empty}, f)] \ \Rightarrow\ \mathsf{Empty} \\
\quad \mid [(\mathsf{Leaf}\ c, f)] \ \Rightarrow\ \mathsf{Node}\ (\mathsf{Leaf}\ c)\ \mathsf{Empty} \\
\quad \mid [(\mathsf{Node}\ ltr\ rtr, f)] \ \Rightarrow\ \mathsf{Node}\ ltr\ rtr \\
\quad \mid (c_1, f) :: (c_2, f_2) :: ls' \ \Rightarrow \\
\quad\quad \mathsf{let}\ nn \ = \ (\mathsf{Node}\ c_1\ c_2, f \ + \ f_2) \\
\quad\quad \mathsf{in} \\
\quad\quad\quad \mathsf{create_tree}\ (\mathsf{sort_frequencies}\ (nn :: ls'))
\end{array}$$

The function `build_huffman_tree` makes use of all functions presented so far to create a Huffman tree.

$$\begin{array}{l}
\mathsf{build_huffman_tree}\ s \stackrel{\mathsf{def}}{=} \\
\quad \mathsf{let}\ freqs \ = \\
\quad\quad \mathsf{sort_frequencies} \\
\quad\quad\quad (\mathsf{convert_frequencies}\ (\mathsf{get_frequencies}\ s)) \\
\quad \mathsf{in} \\
\quad\quad \mathsf{create_tree}\ freqs
\end{array}$$

From the tree, `get_huffman_codes` generates an association list with the elements

of the tree and their respective codes by traversing the tree.

```
get_huffman_codes tree code ≝
 case tree of
  Empty ⇒ []
 | Leaf c ⇒ [(c, code)]
 | Node ltr rtr ⇒
  let left = get_huffman_codes ltr (code ++ [F]);
  right = get_huffman_codes rtr (code ++ [T])
  in
   left ++ right
```

5.3.2 Building a canonical Huffman tree

All Huffman trees used in Deflate are expected to adhere to the restrictions of canonical Huffman codes. Due to this, `unique_huff_tree` was created to encapsulate the creation of a canonical Huffman tree. First, the Huffman tree is created; it is then rewritten as a list of code lengths from which `canonical_codes` constructs the corresponding canonical Huffman tree.

```
unique_huff_tree l ≝
 let huff_tree = build_huffman_tree l;
 assoc_list = get_huffman_codes huff_tree [];
 ls = all_lens assoc_list;
 cs = canonical_codes ls
 in
  (cs, ls)
```

5.4 Encoding and decoding

Encoding and decoding using the Huffman trees is a simple matter of replacing items for each other, both of which are based on the association list generated by the canonical Huffman codes method.

Encoding is the most simple of them, given an element to encode, `encode_single_huff_val` looks up and returns the corresponding code in the association list.

```
encode_single_huff_val ls s ≝
 let res = alookup ls s in case res of None ⇒ [] | Some b ⇒ b
```

Decoding is somewhat more cumbersome due to the variable length of the codes. Each code in the association list is checked if it is a prefix to the encoded list of bits, and once a match is found, the corresponding element is returned.

```
find_decode_match s [] ≝ None
find_decode_match s ((k, v) :: ts) ≝
 if v ≼ s then Some (k, v) else find_decode_match s ts
```

6

Discussion

This chapter presents lessons learned as well as related and future work, to then end the report with our conclusions.

6.1 Related work

There are few studies which aim to verify Deflate or other compression algorithms. We found one study where Deflate has been verified as a whole and a few where Huffman's algorithm has been verified independently. The studies which verified Huffman's algorithm used different interactive theorem provers.

In 2016, Senjak and Hofmann [4] verified an implementation of Deflate using the proof assistant Coq [5]. They created a formal specification of Deflate that can act as an alternative to the, at times, vaguely worded RFC [1]. Their implementation was found to reach competitive performance when compared to other implementations. They used program extraction to extract their code from Coq to OCaml to create an executable.

We differ from Senjak in that we create a executable binary which is guaranteed to have the behaviour from our Deflate implementation. We are able to create this binary since the CakeML compiler itself is verified; thus, the entire compilation chain is verified.

Huffman's algorithm was verified by Théry [8] in 2004 and Blanchette [6] in 2009. Théry performed their formalisation using Coq and went for a straightforward approach, where they defined the optimal encoding problem as a binary tree problem. Blanchette used Isabelle/HOL4 instead of Coq. While the general structure is similar to how Théry verified Huffman, they also used many custom induction rules to make use of Isabelle's automatic tactics to simplify the formalisation.

Another difference between our specification and the other studies is the comprehensive amount of the proofs. Senjak and Hofmann [4] rigorously proved their Deflate implementation. Both Théry [8] and Blanchette [6] have comprehensive proofs for their implementations of the Huffman algorithm.

Our implementation for constructing Huffman trees is what could be called the textbook method. There are, however, other methods which enable the creation of Huffman trees with specific properties. One such method, presented by Lamore and

Hirschberg [26], creates Huffman trees with length restricted codes. Restricting the length of codes is highly applicable to Deflate with its upper limit of 15 bits for a code.

6.2 Future work

For this project, we made some trade-offs due to time constraints. One is that we prioritised making a more thorough implementation over a verified implementation. The reason for this is that we believe a thorough implementation may be easier to write proofs for and hopefully does not need too much rewriting to satisfy Deflate fully. Therefore the implementation lacks proofs, and the addition of proofs would be a step towards both a more robust implementation but also a verified binary of Deflate. We also believe that a comprehensively proven implementation should be well done. If it is not well done or is missing key parts, then no matter how comprehensive the proofs, they will not have the same meaning.

Furthermore, there are improvements to be made for the implementation. One would be to introduce streams. Streams would enable us to read the input file on the go, which comes with the added benefit of not having to keep large files in working memory. This would work well for Deflate as it is a sequential algorithm, which means we do not need the entire input at one time.

Another improvement would be to replace the LZSS implementation used in Deflate with the one using Ringbuffers. This is partly because the intricate functionality of the Ringbuffer is extracted from the LZSS implementation to its own functions making the LZSS functions a lot easier to understand. Another reason is that we can make Ringbuffers mutable by using stateful monads. This could be an efficiency boost as we do not need to create new lists which are frequently exchanged; instead, only the necessary parts are updated.

Due to time constraints we chose to simplify Deflate by using fixed Huffman trees when the dynamic Huffman tree for lengths or distances created codes that were too long. However, there exists a modified version of Huffman that can reshape the original Huffman tree to adhere to a restriction for each code's length. This would have been an interesting option to pursue instead of our simplification, as it is vital that the Huffman codes for the literal/length tree and the distance tree does not exceed the length 15, and that the code length tree does not exceed the length 7.

One of the reasons our implementation is not Deflate compliant is how we handle LZSS backreferences. Deflate expects backreferences to be able to point from one block to another, our implementation does not allow backreferences between blocks. Furthermore, we decided to simplify LZSS by not allowing recursive backreferences, which are a feature of LZSS. However, due to the complexity of recursive backreferences, they were not included to simplify other development in the project. The aim was to incorporate recursive backreferences later, which is a point we, unfortunately, did not reach.

There are other simplifications we have made that comprimises the compliancy. In

our implementation, all data elements, including Huffman codes, are stored in the format of most-significant bit first. This made it easier to handle bit representations. However, the RFC [1] states that the Huffman codes should be stored starting with the most-significant bit, and the rest of the data elements should be stored with the least-significant bit first.

We also have a few other deviations from Deflate. One is because of how our Deflate implementation is verified; a prefix is always appended to the compressed data, which makes the block structure differ from Deflate. Another is that an uncompressed block `LEN` should start on an even byte. To accomplish this, a varying amount of bits may have to be added before `LEN`, but we always add 5 bits as if the block header, which is 3 bits long, is at the start of a byte. This is because the current implementation does not keep track of where in a byte it is.

In Deflate, it is allowed to implement a more restrictive compressor, as long as the output is of the Deflate file format. However, the deviations of our compressor alter the block format and as such they also apply to our decompressor as they would otherwise not work together.

6.3 Testing the binary

As could be seen in Table 3.2, we are not competitive with `zip` in terms of execution times. However, our foremost focus was to create an algorithm which generated output with the correct Deflate format. As the algorithm in this project is Deflate-like, and not a full fledged Deflate algorithm, we did not get to a level where focusing on performance was a priority.

Due to the efficiency limitation of our binary, the tests were performed on smaller files as it was not feasible to run them on larger files. This may in part be because the stateful Ringbuffer is not used in our implementation. Another reason may be that the method for decoding Huffman codes is fairly inefficient. A similar method for decoding characters is used in the compression example which yields a complexity of $O(n^2)$ for one character. Compounding this complexity with the many more complex processes that compose Deflate makes for an inefficient algorithm. To improve the implementation, an investigation of the complexity and improvements thereof could fix some of our efficiency problems.

We performed better when comparing the compression ratio. This can be seen in Table 3.1 where our compression ratio results are comparative to `zip`. However, there are details that need to be fixed before our implementation is Deflate compatible. One such change that could have a more significant impact on the compression ratio is to allow recursive backreferences. Another change that could impact the compression ratio, is to allow backreferences between blocks. This may have a greater impact when the files are larger, as larger files will create more blocks.

6.4 Lessons learned

When we started this project, we had assumed that we would spend quite a lot of time on the verification of our definitions. For this reason, in the beginning, we primarily sought and studied papers about the verification of algorithms related to Deflate. However, we spent most of the time during our thesis trying to understand and implement algorithms, not verifying them. Towards the end of the project, we found papers that have helped us understand the algorithms better. It would however have been helpful to have read them earlier in the project.

We also learned a lot from implementing both LZSS and Huffman separately and then attempting to join them. It gave us a deeper understanding of how the algorithms worked separately. Then we bumped into compatibility issues, when trying to join LZSS and Huffman. The issues we ran into are solved by Deflate and this gave us a deeper understanding of some of the problems Deflate solves.

Moreover, when studying and incorporating Cox's work, we encountered several challenges. There was a lack of documentation, which many times made it hard to understand what the purpose of functions were. It made it hard to distinguish between if we did not understand the intention of the code or if there perhaps was some bug in the code. This also caused us to be reluctant to make changes in the code, as there were not always proofs that indicated whether the function worked the same way or not after changing it.

Lastly, when using termination proofs, we realised that when studying what HOL4 asks us to prove, we were able to find bugs in our code. This experience was new to us, as otherwise an comprehensive amount of tests and finding edge cases would be needed to find the same issue. In this case, we discovered that when there were no matches in the table for the input, the function `find_match` in `tab_sub` would not terminate, which was an edge case we had not considered.

6.5 Conclusion

We have created binaries for a simple static compressor and a Deflate-like compressor specification. Due to time constraints, we did not manage to implement a fully Deflate compliant algorithm. However, reaching Deflate compliant binaries should be possible with some changes to the current implementation.

There is further work to be done regarding efficiency and verification. The current executable binaries are not efficient enough to be used in practice. This comes down to efficiency in the algorithms and data structures we used. Along with this, more verification needs to be done once the implementation is Deflate compliant.

Bibliography

[1] P. Deutsch, "Rfc1951: Deflate compressed data format specification version 1.3," 1996.

[2] P. Deutsch *et al.*, "Gzip file format specification version 4.3," 1996.

[3] P. Inc., ".zip file format specification," September 2012.

[4] C.-S. Senjak and M. Hofmann, "An implementation of deflate in coq," in *International Symposium on Formal Methods*, pp. 612–627, Springer, 2016.

[5] Coq, "The coq proof assistant." url: `https://coq.inria.fr/`. Accessed: 2021-11-30.

[6] J. C. Blanchette, "Proof pearl: Mechanizing the textbook proof of huffman's algorithm," *Journal of Automated Reasoning*, vol. 43, no. 1, pp. 1–18, 2009.

[7] T. N. L. C. Paulson and M. Wenzel, "Isabelle/hol: A proof assistant for higher-order logic," 2021. Accessed: 2021-11-30.

[8] L. Théry, "Formalising huffman's algorithm," tech. rep., Università degli Studi dell'Aquila, 2004. hal-02149909.

[9] HOL, "Interactive theorem prover." url: `https://hol-theorem-prover.org/`. Accessed: 2021-11-26.

[10] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The definition of standard ML: revised*. MIT press, 1997.

[11] P. Bjesse, "What is formal verification?," *ACM SIGDA Newsletter*, vol. 35, no. 24, pp. 1–es, 2005.

[12] CakeML, "A verified implementation of ml." url: `https://cakeml.org/`. Accessed: 2021-11-30.

[13] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys (CSUR)*, vol. 21, no. 4, pp. 557–591, 1989.

[14] P. Gage, "A new algorithm for data compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[15] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[16] E. J. Schuegraf and H. Heaps, "A comparison of algorithms for data base compression by use of fragments as language elements," *Information Storage and Retrieval*, vol. 10, no. 9-10, pp. 309–319, 1974.

[17] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM (JACM)*, vol. 29, no. 4, pp. 928–951, 1982.

[18] The Beatles, "Yellow submarine," 1966.

[19] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[20] madler, "madler/zlib: A massively spiffy yet delicately unobtrusive compression library.," May 2022.

[21] rpms, "rpms/zip: Add patch to fix format-security ftbfs (rhbz 1037412).," June 2022.

[22] T. E. of Encyclopaedia Britannica, "Morse code." ursl: `https://www.britannica.com/topic/Morse-Code`. Accessed: 2022-05-11.

[23] P. Christensson, "Character encoding." ursl: `https://techterms.com/definition/characterencoding`. Accessed: 2022-03-18.

[24] A. Moffat, "Huffman coding," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–35, 2019.

[25] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, 1964.

[26] L. L. Larmore and D. S. Hirschberg, "A fast algorithm for optimal length-limited huffman codes," *Journal of the ACM (JACM)*, vol. 37, no. 3, pp. 464–473, 1990.