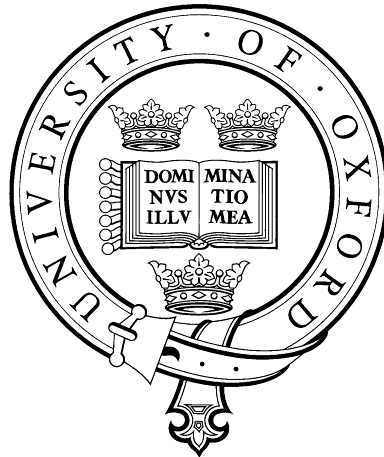
A Formal Treatment of Lossless Data Compression Algorithms

Barney Stratford

Oxford University Computing Laboratory



A thesis submitted for the degree of Doctor of Philosophy

December 13, 2005

Abstract

Since its inception, data compression has been practised mostly as an experimental science. Although this thesis continues that trend to some extent, its main emphasis is on formal derivations of compression algorithms and proofs of their correctness. Such a mathematical approach has not been taken before, and we have found that it has yielded significant dividends in the form of faster compression.

Modern compression schemes can be viewed as a combination of a statistical model and an entropy coder. The method developed in this thesis consists of a PPM (Prediction by Partial Matching) model with arithmetic coding. Other methods, including dictionary-based algorithms and the Burrows-Wheeler Transform, are discussed briefly.

Arithmetic coding can be seen as a generalisation of Huffman coding, with fewer restrictions and better compression. However, the algorithm is quite difficult to analyse and understand, and it's very easy to make a mistake that would render the program incorrect. Our formal approach simplifies the explanation, and gives us confidence in the final software.

Prediction by Partial Matching (PPM) represents the state of the art in statistical modelling. Many of its variants outperform all other known methods. Its major drawback is that it is very slow, and requires temporary storage space linear in the size of the input. A number of the design decisions, while intuitively sensible, are not backed up by any theory. We aimed to justify our version of PPM by using Bayesian statistics. Although this approach

did not entirely succeed, there was some significant progress towards the target.

Our derivations are carried out using notation drawn from the functional programming language Haskell. Haskell provides a number of advantages over the more traditional imperative languages, although all programs are given in C in an appendix.

Acknowledgments

First and foremost, I would like to thank my supervisor, Richard Bird. His enthusiasm for functional programming provided the inspiration for this work, and his support and encouragement throughout the research and writing of this thesis have been invaluable. He agreed to help me even though he was officially on sabbatical for most of the first year, and he has read and provided helpful comments on early drafts of many parts of this thesis. I would have been much worse off without his help.

The latter part of my work was particularly difficult, and I went through a fairly long period when nothing seemed to work properly. Thanks to Irina Voiculescu, David Stirzaker and Alan Grafen for generously giving their time to help me when everything I tried was going wrong.

Thanks to the Algebra of Programming Group for listening to early versions of my ideas and providing helpful feedback and suggestions for improvement.

Ann Morley, Matt Harvey and my parents have been a constant support throughout the writing of this thesis. Matt has also read and commented on a final draft of this work. Without their encouragement, it would have been so much more unpleasant.

Thank you all.

Contents

Abstract **i**

Acknowledgments **iii**

1 Introduction **1**

1.1 History and Applications of Data Compression 1

1.2 How Data Compression Works 3

1.2.1 Lossy and Lossless Compression 3

1.2.2 The Compromises That Must Be Made 5

1.3 Various Data Compression Schemes 6

1.3.1 Statistical Methods: Huffman Coding and Arithmetic Coding 7

1.3.2 Dictionary-Based Methods: LZ77, LZ78 and LZW 8

1.3.3 Other Methods: The Burrows-Wheeler Transform 9

1.4 Thesis Aims 13

1.4.1 The Need for Formal Derivation of Compression Algorithms 13

1.4.2 Intended Audience 13

2 Mathematical Preliminaries **15**

2.1 Functional Programming 15

2.1.1 The Haskell Type System . . . 16
2.1.2 Function Evaluation . . . 19
2.1.3 Constructors . . . 20
2.2 Interval Operations . . . 21
2.3 Integer Arithmetic . . . 23
2.4 String Operations . . . 24

3 Anatomy of an Arithmetic Coder **25**
3.1 An Overview of Arithmetic Coding . . . 25
3.2 A Definition of Arithmetic Coding . . . 28
3.3 Input and Output . . . 32
3.4 Data Compression with Arithmetic Coding . . . 39
3.5 Rational-Valued Models . . . 41
3.6 Approximate Arithmetic Coding . . . 43
3.7 Conclusion . . . 49

4 Some Arithmetic Coding Schemes **50**
4.1 Introduction . . . 50
4.2 A First Attempt . . . 50
4.2.1 Shrinking Intervals . . . 51
4.2.2 The Complete Algorithm . . . 55
4.2.3 A Problem . . . 57
4.3 Witten, Neal and Cleary's Coder . . . 58
4.3.1 Representing Intervals . . . 59
4.3.2 Re-Normalisation . . . 61
4.3.3 Re-Normalisation in the Decoder . . . 64

4.3.4 The Interval Approximation . . . 65
4.3.5 Putting it all Together . . . 67
4.3.6 The Underflow Problem Again . . . 71
4.4 A Fast, General-Purpose Coder . . . 71
4.4.1 Re-Normalisation . . . 74
4.4.2 Approximate Interval Shrinking . . . 78
4.4.3 The Faster Interval Approximation . . . 82
4.4.4 The Complete Coder . . . 86
4.5 Analysis of the New Coder's Performance . . . 86

5 The PPM Model 91

5.1 Introduction . . . 91
5.2 Prediction by Partial Matching . . . 92
5.2.1 Which Context? . . . 94
5.2.2 Implementing PPM with Suffix Trees and DAWGs . . . 95
5.2.3 Theme and Variations . . . 97
5.3 Statistical Estimation . . . 97
5.3.1 Maximum Likelihood Estimation . . . 98
5.3.2 Bayesian Estimation . . . 98
5.4 A New PPM Model . . . 100
5.4.1 The Axiomatisation . . . 100
5.4.2 A Simplification . . . 104
5.5 Towards Bayesian PPM . . . 111

A The New Coder in C 112

A.1 Introduction . . . 112

A.2 The Arithmetic Coding Library 113
A.2.1 ac.h 113
A.2.2 encode.c 114
A.2.3 decode.c 117
A.2.4 lookup.c 120
A.3 Example Compression Software 121
A.3.1 main_encode.c 121
A.3.2 main_decode.c 122
A.3.3 model.h 123
A.3.4 model.c 123

B Some Standard Haskell Functions **126**

C Proofs of Theorems **128**
C.1 Interval Operations 128
C.2 Arithmetic Coding Proofs 136

Bibliography **140**

Chapter 1

Introduction

1.1 History and Applications of Data Compression

In recent years, the amount of data needing to be stored and transmitted on computer systems has soared. To compensate, storage media have increased in size and extra network bandwidth has been provided, but our demand for on-the-spot information remains insatiable. There comes a time when it is no longer cost-effective to build bigger computers and networks, and that is when data compression becomes necessary.

Data compression has been in use for centuries—it dates from long before the invention of modern digital computers [32]. A scheme invented by the freed slave Marcus Tullius Tiro was used in the Roman Senate, whereby discussion was noted down in compressed form, and then expanded into longhand later when time was less pressing. Tiro's method bears many resemblances to the most modern of compression schemes: more common messages are compressed to a greater extent than less common messages; the context in which a word or phrase appears is used to predict its probability; and the method was developed to overcome a problem of limited bandwidth. With the onset of the Dark Ages, however, Tiro's method

was lost and shorthand, along with most of civilisation, entered a period of stagnation.

A more recent example of data compression comes from the Morse Code (invented by Alfred Vail [67] in 1844 as a development of Morse's earlier ideas). Using the dot and dash symbols to represent the letters, the code gives very nearly the best possible communication speed for English text. The protocols used by the telegraph operators of old would have heartened a latter-day network nerd! Even today it has not become obsolete, since a Morse signal can be recovered in the very worst of conditions, and it can be used to send messages to the remotest parts of the world by radio at low cost.

A modern application of data compression technology is in speech recognition. As human utterances go flying past a computer's microphone like bullets from a machine-gun, it is inevitable that mistakes will be made in interpretation. By using a statistical model of speech, we can predict what sounds might be appearing next, and so can reduce the error rate.

Data compression can help increase the execution speed of microprocessors. When a conditional branch instruction is encountered, a processor will typically have to flush all of the partially-executed instructions from its pipeline. This costs a great deal of time, particularly in a tight loop. To reduce this cost, branch prediction is employed: we take a guess as to which way the flow of execution will go, and only flush the pipeline when that guess is wrong. Once more, data compression can assist us with the prediction process [7].

By predicting upcoming letters, compression algorithms have been used as the basis for efficient input devices for people with special needs. Of particular note is the Dasher system [22], which is based on arithmetic coding. It can be controlled using almost any muscle of the body, by mouse gestures, or even by glancing at the screen. Because of the fact that misspellings occur with lower frequency than correct spellings, the system actually makes it harder to misspell words, and the resultant decrease in such errors was apparent

in objective testing. After training, the speed of data input was comparable to the use of a keyboard, even for users with limited mobility.

There have even been suggestions that data compression is, in a sense, the same thing as artificial intelligence [40]: the more intelligent a machine is, the better it will be able to predict what letters are about to occur in a file to be compressed, and therefore it will achieve increased compression.

1.2 How Data Compression Works

The rigorous mathematical study of data compression was initiated by Shannon's famous papers on information theory [58]. With his Noiseless Coding Theorem, he proved that there was a limit on how well data could be compressed[1]. Huffman coding provided an example of a compression algorithm that came very close to Shannon's limit, and this was further improved upon with the advent of arithmetic coding, which (in theory) can achieve that limit.

1.2.1 Lossy and Lossless Compression

If we are to store a file in less space, we will clearly have to throw some information away. We can classify the data in a file as either useful, redundant or irrelevant. Redundant information is that which can be reconstructed or deduced from the useful information, while irrelevant information is simply that which can be lost irretrievably without ill-effect.

Clearly, we can't throw away any of the useful information, or our data will be damaged.

[1] This fact is often forgotten by people who should know better, for example [27, 14]. A simple counting argument shows that no compression scheme can compress every possible string into something shorter without losing some information.

However, we can remove the redundant information with no problem at all, and in certain situations we may wish to throw away irrelevant information as well. This is the means by which compression algorithms do their work. If only redundant information is discarded, then the result is lossless compression: we can exactly reconstruct the redundant information, so the decoded data is identical to the original file. If the irrelevant information is thrown away as well then we have lossy compression, since there may be some distortion in the decoded data. We can't reconstruct the irrelevant data, but that doesn't matter because it is irrelevant.

Natural languages provide many examples of these ideas. For example, English text has redundancy in its letters: 'E' occurs with probability 10%, while 'Z' occurs only 0.2% of the time. There is also redundancy in the words, since certain parts of speech can occur only in certain contexts, following rules defined by the grammar. The context in which a sentence is uttered affords us yet more redundancy.

This incredible level of redundancy means that we can recover a spoken word in a crowded room, we can delete letters or misspell words, and we can even violate the normal rules of grammar and still be understood:

- The itndened mnieang is eisaly receroved evne fi hte spilgelns hvae bene mnglead.
- W cn dscrd mst f th vwls wth n prblm.
- _ __ g____ __ ___ c____a.
- This sentence no verb.

Interestingly, that isn't the whole story, since some sentences that are perfectly well-formed are nonetheless nonsense:

- Don't look at me in that tone of voice—it smells a funny colour.

and some utter nonsense has acquired a meaning of its own:

- All your base are belong to us.

Some gibberish (such as "Jabberwocky" by Lewis Carroll) even "seems to fill [one's] head with ideas" [12].

All of this redundancy means that English text is a prime target for data compression. Current methods will typically compress text down to about 2 bits per character, although our limited understanding of how human language *really* works surely indicates that this figure will be improved upon one day.

An example of lossy compression comes from the storage of music. A sound wave is a continuous function of time, and so there is significant redundancy when this is encoded into a file. Furthermore, there is also irrelevant information, since some sounds are too high-pitched for us to hear. These can simply be discarded. The decoded sound is almost—but not exactly—the same as the original, and the differences are too small to notice.

1.2.2 The Compromises That Must Be Made

There are four measures of how well a data compression algorithm performs: compression speed, decompression speed, size of the compressed file, and (for lossy compression) the quality of the encoded data. Following Moffat's lead [45], we use the term *efficient* for algorithms that achieve a high speed, and *effective* for those that achieve a high compression ratio.

It is usually impossible to build an algorithm that is both very efficient and very effective; instead a tradeoff is required. Algorithms such as LZW are fast at encoding, extremely fast at decoding, but aren't terribly effective. By contrast, arithmetic coders tend to be much slower, but much more effective.

What kind of algorithm is to be used depends entirely on the application. When a file

is to be sent over a network, for example, fast decompression is usually more of a concern than the size of the compressed data. When backing up a large quantity of data, however, it's usually more important to ensure good compression ratios. Since backed-up data will (hopefully) never be needed, decompression speed is less of an issue.

1.3 Various Data Compression Schemes

Data compression algorithms can be broadly classified into one of two kinds: statistical methods and dictionary based methods.

Statistical methods, as the name implies, rely on the use of a statistical model of the data being encoded. The model predicts the distribution of the forthcoming letters in the data stream, and feeds the predictions to an entropy coder. Entropy coders use fewer bits to represent a very likely message than to represent a very unlikely message, thereby minimising the expected length of the compressed data stream.

Dictionary algorithms work in a totally different way. They maintain a data structure containing some of the substrings of the part of the input data that has already been encoded. When a previously-observed substring occurs, a pointer into the data structure is stored instead of the full substring. Because the pointers usually take less space than the original substring, data compression is achieved.

The dictionary-based methods tend to be much faster than the statistical methods, particularly at decoding, but their compression effectiveness is markedly less.

1.3.1 Statistical Methods: Huffman Coding and Arithmetic Coding

Huffman coding was discovered by David A. Huffman in 1951. He found it impossible to prove that any of the then-known coding algorithms will always produce a minimum-length symbol code. The simple reason for this is that it isn't true. Instead, he produced a variant of the Shannon-Fano encoding that does have this desirable property.

A Huffman coder works by creating a binary tree whose leaves are labelled with the symbols to be encoded. We begin with a list of all the trees consisting of a single leaf, together with the probabilities of occurrence of the corresponding letters. We then remove the two least likely trees in our list and give them a common parent node. The resulting larger tree then has a probability that is the sum of the probabilities of the two original trees. The new tree is added back to our list. We repeat this process until there is only a single tree in our list. This is called the Huffman tree.

To encode a symbol, we give a path through the (binary) tree from the root to the appropriate leaf. Similarly, we can decode by following the given path to see what symbol is at that leaf. Because less likely symbols have a greater depth in the Huffman tree, they have longer code words, and so more likely symbols can have shorter codewords. The expected length of a codeword is minimised and so compression is achieved.

Arithmetic coding is seen by some authors as a generalisation of Huffman's method, although this is by no means a universally-held opinion. It is discussed in much more detail in the following chapters. The main disadvantage of Huffman coding is that it always requires at least one bit to represent a letter (even if that letter occurs with near certainty) whereas arithmetic coding can cope with fractions of a bit of information. A further advantage to arithmetic coding is that it doesn't require the use of the Huffman tree, and therefore saves

the time that would be spent building and maintaining it. This makes it much more suitable for adaptive encoding—see Chapter 3.

When Huffman coding was invented, it precipitated the near-instantaneous death of Shannon-Fano encoding, since there were no good reasons to continue using the older, inferior method. By contrast, arithmetic coding has failed to displace Huffman coding in spite of its many strengths. This has been caused by its encumbrance with a number of software patents with extremely restrictive licence terms. Fortunately, these patents are now nearing the ends of their lives—check with your friendly lawyer before using the technique yourself! Even once the patents are gone, it is likely that Huffman coding will continue to be used for some significant time. So many file formats rely on it nowadays that replacing it will be a major undertaking.

1.3.2 Dictionary-Based Methods: LZ77, LZ78 and LZW

The dictionary-based methods do not rely on an explicit statistical model of the text; instead, they maintain a data structure representing the part of the text that has been encoded, and they refer back to that structure to see what the next few letters in the data stream are. Precisely how this happens depends on the algorithm in question.

It is not intended to give a full description of these methods here, as they are not discussed in the rest of the thesis. Instead, we just give a flavour of how such algorithms work.

The first of the dictionary-based methods was developed by Abraham Lempel and Jacob Ziv [71], and became known as LZ77. In this method, we search through the already-encoded part of the data stream to find the longest prefix of the data remaining to be encoded. If there is such repetition, then we simply give the index into the already-encoded data where this prefix can be found, together with the length of the prefix. Finally, we give the letter following the repeated section of text.

The decoder simply copies bytes from the locations indicated in order to reconstruct the original file. This simplicity means that the decoder can be extremely fast.

Various kinds of data structure can be used to search for repeated text. One of the faster ones is a suffix tree, although many implementations of the algorithm simply search in a finite-length circular queue, ignoring any repetitions that occurred in the distant past.

The main problem with LZ77 is that there is still a large amount of redundancy in the encoded data stream, since a particular substring may have occurred more than once. LZ78 [72] relies on maintaining an explicit dictionary of previously-seen substrings: instead of recording substrings by their start position and length, an offset into the dictionary is given. Since there is only one dictionary entry for each previously-seen substring, there is reduced duplication of information, and so better compression can be achieved.

LZW [65] is a further refinement of the method. In both LZ77 and LZ78, the encoded data stream contains references to previously-occurring substrings, together with a single byte that is encoded literally. LZW does away with the literal byte. Instead, we begin with a dictionary that contains all single-byte substrings, and the encoded data stream contains references to this expanded dictionary.

There are many variations on this basic theme, including LZSS, LZFG, LZRW, and others.

1.3.3 Other Methods: The Burrows-Wheeler Transform

No discussion of data compression methods would be complete without mentioning the Burrows-Wheeler Transform. The method was first discovered by David Wheeler in 1983, but it was not until 1994 that he and Mike Burrows published a technical report [11] outlining the method. It was still largely unknown until Mark Nelson published an article [48] in Dr Dobb's Journal in 1996, after which papers concerning the algorithm started to appear in

1	B	A	R	B	A	R	A
2	A	B	A	R	B	A	R
3	R	A	B	A	R	B	A
4	A	R	A	B	A	R	B
5	B	A	R	A	B	A	R
6	R	B	A	R	A	B	A
7	A	R	B	A	R	A	B

1	A	B	A	R	B	A	R
2	A	R	A	B	A	R	B
3	A	R	B	A	R	A	B
4	B	A	R	A	B	A	R
5	B	A	R	B	A	R	A
6	R	A	B	A	R	B	A
7	R	B	A	R	A	B	A

Figure 1.1: The rotations (left) and sorted rotations (right) of "BARBARA".

many other places.

The interest generated by this method is not really surprising: it is an exceptionally beautiful algorithm, and compression software based on it can perform almost as well as the state of the art with a much lower cost in time and space.

The *rotation* of a string of length n is the result of removing the last letter and putting it on the front of the $n-1$ letters remaining. For example, the rotation of the string "BARBARA" is "ABARBAR". This can be further rotated, resulting in "RABARBA", "ARABARB", etc. The nth rotation will be identical to the original string.

To compute the Burrows-Wheeler transform of our string, we sort the n rotations into lexicographic order and take the last letter of each of the sorted rotations. Figure 1.1 shows that the Burrows-Wheeler Transform of "BARBARA" is "RBBRAAA".

In order to decode the message uniquely, we will also need to know that the original message occurs in the 5th line of the table of sorted rotations.

Notice that we can take the sorted table, rotate it, and sort it again to get back to where we started (see Figure 1.2). This is the basis of the reverse transform. Initially, we know only the BWT of the original string, so we can enter this into our table, as in Figure 1.3. We apply the rotation to all the elements in the table, and then we sort the result. Since we

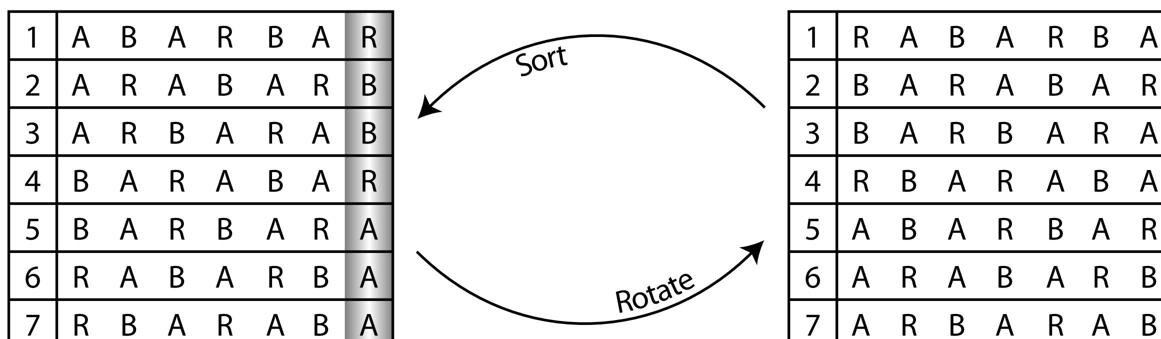


Figure 1.2: Rotating and sorting the table

know that rotating followed by sorting has no effect on the sorted table, we know that the shaded column is unaffected by these operations.

We continue rotating and sorting until the whole of the original table has been recovered. Looking at the 5th line of the recovered table, you can see the original message, “BARBARA”.

We have glossed over many of the details of the algorithm: in particular, we can avoid building the whole Burrows-Wheeler table. These shortcuts are discussed more fully in [48, 41, 33, 37].

So why is this method useful for data compression? The reason is that letters in a string are usually heavily influenced by nearby letters. For example, in English, the string “here” is usually preceded by a space or a “t”. When the rotations are sorted, those that begin with “here” are likely to end with a space or a “t”, and so there will be lots of spaces and “t”s close together in the transformed text. In fact, the BWT of a text contains many long runs of the same letter. For example, the BWT of part of this chapter contains the string “aaaattttttttTnnnswSnsttttstrrr”, with long runs of identical letters. This is much easier to compress than the original string. Typically [18], the BWT will be followed by move-to-front encoding or run-length encoding, followed by an entropy encoder.

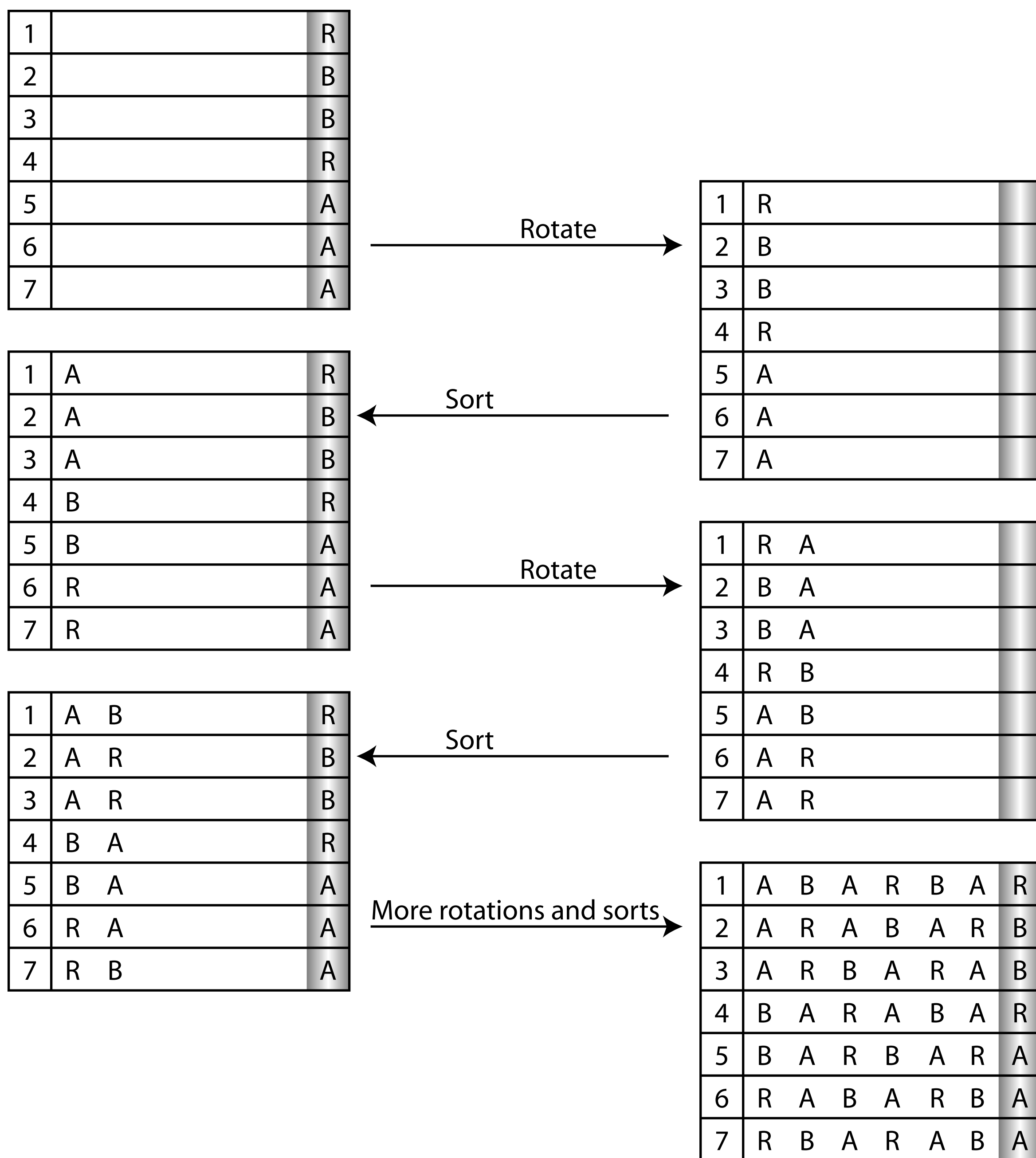


Figure 1.3: Computing the inverse BWT of “RBBRAAA”.

1.4 Thesis Aims

1.4.1 The Need for Formal Derivation of Compression Algorithms

Reading the data compression literature, one frequently encounters references to experimental results that show that one algorithm outperforms another on specific examples of text, but there is often little in the way of formal proof that any particular algorithm is the *best possible* for any particular situation. Indeed, there is not usually even a statement of what assumptions an algorithm makes of the string being encoded.

Starting from a definition of arithmetic coding, we derive a framework for practical arithmetic coding algorithms. The standard Witten, Neal and Cleary coder [70] falls within this theoretical framework, and we will also see how this formal work leads quite naturally to a significantly faster coding algorithm that would have been hard to spot by informal methods.

The second part of this thesis concerns the PPM algorithm, which is used to build a statistical model of the data being encoded. We attempt to start from some sensible assumptions about our data and derive a PPM-like algorithm from those assumptions. Although it proved impossible to carry the reasoning through to the intended result, we have nonetheless made some useful progress for others to build on.

1.4.2 Intended Audience

Data compression is an area that sits on the boundary between several different disciplines. It is both computer science and statistics; both a theoretical and an experimental field of study. Many in the data compression community are unfamiliar with the functional programming paradigm, while the functional programmers have not published many papers on data compression or statistical algorithms [6]. We intend to cater for both of these groups,

although we will be assuming that the reader has basic knowledge of both probability and programming.

For those that prefer experimentation to theory, the algorithms developed are refined to C code in Appendix A.

Chapter 2

Mathematical Preliminaries

2.1 Functional Programming

All the derivations in this thesis are carried out using the conceptual framework provided by functional programming. The great advantage of this is that it allows us to concentrate on *what* is being computed, rather than *how* it is computed. Functional programming languages do not have the concept of state, thereby removing a complication that frequently arises in the analysis of imperative programs. Issues of how a computation is carried out can be deferred until after an algorithm has been thoroughly studied and understood.

Execution of a program in a functional language is performed by evaluating an expression. By contrast, traditional imperative languages are executed by performing a series of instructions. Church proved that the two models of computation give rise to precisely the same set of computable functions.

In this thesis, we will be using a syntax loosely based on the functional programming language called Haskell. We use this language mainly as a specification tool, before refining our programs to C code later on. Equivalent programs can usually be executed much faster

in C than in Haskell, but the downside is that C has many more pitfalls: in particular, bugs that manifest themselves in C programs usually cause compiler errors when the equivalent program is written in Haskell, thereby saving a great deal of time and effort tracking down the offending lines of source.

By using these methods, we hope to get the best of both worlds: the ease of reasoning in Haskell, together with the speed of C.

In this section, we give only a brief overview of Haskell: for a fuller introduction, consult [5] or [54]. For further discussion of the rationale behind the use of Haskell, see [55].

2.1.1 The Haskell Type System

Haskell is a strongly-typed language, meaning that a variable can hold data of only one type. Attempting to use data of one type where we expect to see a different type will cause a compiler error. For example, the following Haskell code will not work:

```
xs :: [Integer]   -- xs is declared to be a list of integers.
xs = 0            -- 0 is an integer, not a list of integers. This is an error.


ys :: [Integer]
ys = [0]          -- [0] is the list containing the integer 0, so this is fine.
```

Haskell has all the usual types that one might expect to see in a programming language, including Char, Integer, Float and Bool.

One of Haskell's most important data-types is the list. As we have seen, lists are indicated by square brackets. Additionally, strings (lists of characters) can be indicated by using quotes.

$xs :: \text{String}$

$xs = \text{“Hello”}$

$ys :: [\text{Char}]$

$ys = [\text{‘H’}, \text{‘e’}, \text{‘l’}, \text{‘l’}, \text{‘o’}]$ -- Same thing, with the same type, as xs

Lists have a concatenation operator, denoted $+\!\!+$, where

$$[x_0, \ldots, x_a] +\!\!+ [y_0, \ldots, y_b] = [x_0, \ldots, x_a, y_0, \ldots, y_b].$$

As a useful shorthand, we will write $x : xs$ to mean $[x] +\!\!+ xs$.[1] The empty list is written $[\,]$.

One thing that is novel to seasoned C programmers[2] is that functions are first-class objects in Haskell. They can be passed as parameters to other functions, and they can be manipulated in various ways. The type of functions is represented with the $\to$ symbol:

$\mathsf{plusone} :: (\text{Integer} \to \text{Integer}) \to (\text{Integer} \to \text{Integer})$

$\mathsf{plusone}\ f = g$

$\quad \textbf{where}\ g\ x = f\ (x + 1)$

Equivalently, we could write

$\mathsf{plusone} :: (\text{Integer} \to \text{Integer}) \to (\text{Integer} \to \text{Integer})$

$\mathsf{plusone}\ f\ x = f\ (x + 1)$

There are two ways to pass multiple arguments to a function in Haskell. The one that will be more familiar is to use a *tuple*:

[1]In actual fact, Haskell has : as its primitive operator, from which $+\!\!+$ is defined. For our purposes, this minor detail can be overlooked.

[2]Perl programmers may be more familiar with this, since Perl has apparently borrowed this, and several other features, from Haskell.

$$\mathsf{times} :: (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}$$

$$\mathsf{times}\ (x, y) = x * y$$

However, since functions are first-class objects, we can also say

$$\mathsf{times} :: \text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})$$

$$\mathsf{times}\ x\ y = x * y$$

which is to say that $\mathsf{times}\ x$ is itself a function taking a single parameter, y. The type of times can equivalently be written without the brackets:

$$\mathsf{times} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}$$

This mechanism is extremely powerful, for it allows us to combine functions together in all kinds of interesting ways. For example, it is perfectly sensible to evaluate

$$\mathsf{plusone}\ (\mathsf{times}\ 2)\ x$$

which is the same as $2 * (x + 1)$.

Functions can be composed with the $\circ$ operator:

$$(f \circ g)\ x = f\ (g\ x)$$

In this thesis, we will be referring to some types that are not present in Haskell as standard. This is because we are using Haskell as the basis of a language for derivation of programs, not as a programming language in itself. They are:

- Symbol: Used to represent arbitrary symbols. A symbol may be a character, or a group of characters, or even something that can't be represented with the Char datatype.
- Interval: Intervals in the real number line.
- Real: Arbitrary real numbers.
- Digit: The type of digits in some specified number base. First introduced in Section 3.3.

2.1.2 Function Evaluation

In Haskell, functions are evaluated by performing a series of *reductions*. At each step, an expression is reduced, by substitution, to something simpler. The computation halts when no further reductions are possible.

2.1.1 Example. Two of the standard Haskell functions, head and repeat are defined as

$$\mathsf{head}\ (x : xs) = x$$

$$\mathsf{repeat}\ x = x : \mathsf{repeat}\ x$$

The expression head (repeat 42) can be evaluated as follows:

$$\begin{aligned} & \mathsf{head}\ (\mathsf{repeat}\ 42) \\ = & \ \{\text{Definition of } \mathsf{repeat}\} \\ & \mathsf{head}\ (42 : \mathsf{repeat}\ 42) \\ = & \ \{\text{Definition of } \mathsf{head}\} \\ & 42 \end{aligned}$$

□

It is interesting to note that many traditional imperative languages would have choked on the previous program, for repeat 42 is an infinite list. An important aspect of Haskell is its *lazy semantics*: expressions are evaluated only as much as is necessary. In this case, it was not necessary to completely evaluate repeat 42 before passing it to head. It's quite OK to produce an infinite data structure, as long as most of its data is not used.

Haskell provides us with a wide variety of standard functions that are the building blocks for creating programs. For reference, many of these are described in Appendix B.

2.1.3 Constructors

Haskell provides a special kind of function, called a *constructor*, for representing the values of a datatype. A constructor differs from other functions in that it can't perform any computation or be reduced further; an expression involving a constructor is already in its simplest form. In Haskell, constructors are distinguished by the fact that their names begin with a capital letter. Since constructors can't be reduced any further, we can extract the values of their arguments by a process of pattern-matching.

2.1.2 Example. The Either α β datatype has two constructors, $\mathsf{Left} :: \alpha \rightarrow \text{Either } \alpha\ \beta$ and $\mathsf{Right} :: \beta \rightarrow \text{Either } \alpha\ \beta$. This datatype is often used to represent computations that may return an error. The values $\mathsf{Left}\ x$ represent the error x, while values of the form $\mathsf{Right}\ y$ represent a successful computation that returns y.

To illustrate how constructors can be used in pattern-matching, we define a function called show.

$$\mathsf{show} :: \text{Either String Bool} \rightarrow \text{String}$$

$$\mathsf{show}\ (\mathsf{Left}\ x) = \text{“Left ”} +\!\!+\ x$$

$$\mathsf{show}\ (\mathsf{Right\ False}) = \text{“Right False”}$$

$$\mathsf{show}\ (\mathsf{Right\ True}) = \text{“Right True”}$$

□

Some of the datatypes that we will encounter, including intervals and real numbers, will be considered initially as abstract entities. When we wish to write programs that manipulate elements of these types we will have to declare constructors and specify the abstract values that are represented by the constructors. For example, a list of digits could be used to represent an abstract real number.

2.2 Interval Operations

In order to be able to build and use arithmetic coders easily, we will need to develop a little of the mathematics of intervals. The aim of this section is to present the operations that we will be using, and to discover some of their properties. The proofs of these results are unilluminating, so they have been relegated to Appendix C.1.

2.2.1 Definition. For real numbers a and b such that $a < b$, we use the notation $\langle a,\, b\rangle$ to represent the interval[3] $\{x \in \mathbb{R} : a \leq x < b\}$. We let $\mathcal{I} = \{\langle a,\, b\rangle : a, b \in \mathbb{R} \wedge a < b\}$. □

Of course, the set $\mathcal{I}$ is not very interesting without some operations, and so we will give it some.

2.2.2 Definition. For any $\langle l_1,\, r_1\rangle$, $\langle l_2,\, r_2\rangle$ and $\langle l,\, r\rangle$ in $\mathcal{I}$, let

- $\langle l_1,\, r_1\rangle \otimes \langle l_2,\, r_2\rangle = \langle l_1 + l_2(r_1 - l_1),\, l_1 + r_2(r_1 - l_1)\rangle$
- $\langle l,\, r\rangle^{-1} = \left\langle \dfrac{-l}{r-l},\, \dfrac{1-l}{r-l} \right\rangle$ □

Suppose that I and J are any two intervals. If we translate and stretch the real line so that $\langle 0,\, 1\rangle$ is mapped to I, then J will be mapped to $I \otimes J$ and I^{-1} will be mapped to $\langle 0,\, 1\rangle$. Elastic bands provide a useful mental image.

2.2.3 Proposition. Let $I, J, K \in \mathcal{I}$. Then

- $I \otimes J \in \mathcal{I}$
- $I^{-1} \in \mathcal{I}$
- $I \otimes \langle 0,\, 1\rangle = \langle 0,\, 1\rangle \otimes I = I$
- $I \otimes I^{-1} = I^{-1} \otimes I = \langle 0,\, 1\rangle$

[3] The standard notation for such intervals is $[a, b)$. We have avoided this notation owing to the possible confusion between interval delimiters, list brackets and parentheses.

- $I \otimes (J \otimes K) = (I \otimes J) \otimes K$

In short, $\mathcal{I}$ is a *group* with identity $\langle 0,\, 1\rangle$ and operations $\otimes$ and $^{-1}$. □

When we translate and stretch the real line like this, we move not only intervals, but also single points. We can therefore extend our definition of $\otimes$ to act on real numbers as well as intervals.

2.2.4 Definition. For $\langle l,\, r\rangle \in \mathcal{I}$ and $x \in \mathbb{R}$, let $\langle l,\, r\rangle \otimes x = l + x(r - l)$. □

2.2.5 Proposition. Let $I, J \in \mathcal{I}$ and let $x \in \mathbb{R}$. Then

- $\langle 0,\, 1\rangle \otimes x = x$
- $(I \otimes J) \otimes x = I \otimes (J \otimes x)$ □

To put it another way, Proposition 2.2.5 says that $\mathcal{I}$ has a *group action* on $\mathbb{R}$, and it is for this reason that we have chosen to use the $\otimes$ symbol both for the group operation and the group action.

2.2.6 Definition. For $\langle l,\, r\rangle \in \mathcal{I}$, define

- $\mathsf{left}\ \langle l,\, r\rangle = l$
- $\mathsf{right}\ \langle l,\, r\rangle = r$
- $\mathsf{width}\ \langle l,\, r\rangle = r - l$ □

2.2.7 Proposition. Let $I, J \in \mathcal{I}$. Then

- $\mathsf{left}\ (I \otimes J) = I \otimes \mathsf{left}\ J$
- $\mathsf{right}\ (I \otimes J) = I \otimes \mathsf{right}\ J$.
- $\mathsf{width}\ (I \otimes J) = \mathsf{width}\ I \times \mathsf{width}\ J$ □

As well as having some pleasing algebraic properties, $\mathcal{I}$ behaves well in a set-theoretic sense.

2.2.8 Proposition. Let I, X and Y be in $\mathcal{I}$, and let $x \in \mathbb{R}$. Then

- $X \subseteq Y \iff I \otimes X \subseteq I \otimes Y$
- $x \in X \iff I \otimes x \in I \otimes X$ □

The kinds of mathematical structures described in this section occur in many other places as well. For a fun introduction to the area, see [49].

2.3 Integer Arithmetic

We will need to use some results about the floor and ceiling functions, $x \mapsto \lfloor x \rfloor$ and $x \mapsto \lceil x \rceil$. These are used when we come to discuss incremental decoding, as well as when we discuss speed improvements to the new algorithm. Many published proofs involving arithmetic coding have been made more difficult than necessary because the authors were unaware of these useful characterisations.

2.3.1 Lemma. (Rule of Floors and Ceilings) Let $x \in \mathbb{R}$ and $n \in \mathbb{Z}$. Then

- $n \leq \lfloor x \rfloor \iff n \leq x$
- $\lfloor x \rfloor < n \iff x < n$.
- $n < \lceil x \rceil \iff n < x$
- $\lceil x \rceil \leq n \iff x \leq n$.
- $-\lfloor x \rfloor = \lceil -x \rceil$.
- $\lfloor -x \rfloor = -\lceil x \rceil$.

Proof. Follows directly from the definitions: $\lfloor x \rfloor$ is the largest integer that is less than or equal to x and $\lceil x \rceil$ is the smallest integer that is greater than or equal to x. □

There is one final observation that we will use: if n and m are integers, then $n \leq m \iff n < m + 1$. We refer to this as the "Plus 1 Rule".

2.4 String Operations

There are a couple of notations that will be used for string operations.

- $xs \preceq ys$ if xs is a prefix of ys.
- $xs = ys$ if xs is equal to ys.
- $xs \preceq_n ys$ if xs is a substring of ys that occurs at position n.
- $xs =_n ys$ if xs is the longest substring of ys that occurs at position n.

More formally,

- $xs \preceq_n ys \iff xs \preceq \mathsf{drop}\ n\ ys \wedge n \leq \mathsf{length}\ ys$, and
- $xs =_n ys \iff xs = \mathsf{drop}\ n\ ys \wedge n \leq \mathsf{length}\ ys$.

Chapter 3

Anatomy of an Arithmetic Coder

3.1 An Overview of Arithmetic Coding

A *message* can be considered as a finite list of symbols drawn from some alphabet. In arithmetic coding, each message is represented by an interval of real numbers in $\langle 0, 1\rangle$. Messages containing more information are represented by narrower intervals; the empty message is always encoded as $\langle 0, 1\rangle$.

Every interval that represents some message xs can be partitioned into disjoint subintervals, one for each letter x of our alphabet. The subintervals are then taken to represent the messages $xs + [x]$. By repeating this partitioning process, we can encode messages of any required length.

To decode, all we need to know is a *single* element e of the interval representing the message. Suppose that xs is any message whose corresponding interval I contains e. The messages $xs + [x]$ will be represented by disjoint subsets of I, only one of which contains e. We can use this fact to find successively longer messages whose corresponding interval contains e. Eventually, we will have recovered the whole message.

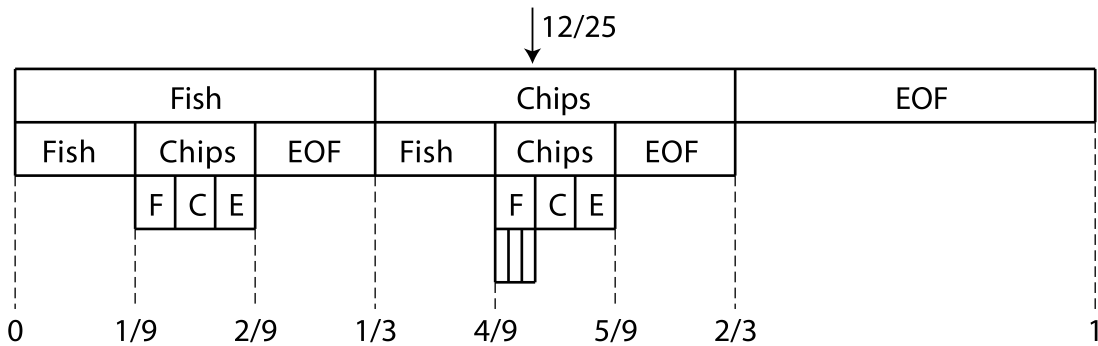


Figure 3.1: The Arithmetic Coder of Example 3.1.1.

The partitions used by these algorithms are fixed at the design stage, so both the encoder and decoder will know how to partition each interval. The particular choice of partition used is known as a *model*, and is discussed further in the next section.

3.1.1 Example. Suppose that our alphabet consists of the three symbols "Fish", "Chips" and "EOF". Every interval that represents a message xs is partitioned into thirds, with the lowest third representing the message $xs ++ [\mathrm{Fish}]$, the middle third representing the message $xs ++ [\mathrm{Chips}]$, and the highest third representing the message $xs ++ [\mathrm{EOF}]$. The EOF symbol occurs at the end of every message we wish to encode, and nowhere else.

Suppose that we need to encode the message $[\mathrm{Fish}, \mathrm{Chips}, \mathrm{EOF}]$. The only message whose encoding we know straight away is the empty message, $[\,]$, which is encoded as $\langle 0,\ 1\rangle$.

By using our model, we see that $[\,] ++ [\mathrm{Fish}]$ is encoded as $\langle 0,\ {}^1\!/_3\rangle$. The middle third of this, or $\langle {}^1\!/_9,\ {}^2\!/_9\rangle$, is the interval representing $[\mathrm{Fish}] ++ [\mathrm{Chips}]$. Finally, $[\mathrm{Fish}, \mathrm{Chips}] ++ [\mathrm{EOF}]$ is represented by the top third of $\langle {}^1\!/_9,\ {}^2\!/_9\rangle$, or $\langle {}^5\!/_{27},\ {}^6\!/_{27}\rangle$, and our complete message has been encoded.

Suppose now that a friend has encoded a message, and we are told that its corresponding interval contains ${}^{12}\!/_{25}$. We know straight away that $[\,]$ is a message whose corresponding interval, $\langle 0,\ 1\rangle$, contains ${}^{12}\!/_{25}$.

As before, $\langle 0,\ 1\rangle$ is partitioned into thirds, and the subset containing $^{12}/_{25}$ is $\langle ^1/_3,\ ^2/_3\rangle$. This corresponds to the message $[\,] +\!\!+ [\text{Chips}]$. The middle third of $\langle ^1/_3,\ ^2/_3\rangle$ is $\langle ^4/_9,\ ^5/_9\rangle$, representing the message $[\text{Chips}] +\!\!+ [\text{Chips}]$, and this too contains $^{12}/_{25}$. The message $[\text{Chips}, \text{Chips}] +\!\!+ [\text{Fish}]$ is represented by the interval $\langle ^{12}/_{27},\ ^{13}/_{27}\rangle$, which also contains $^{12}/_{25}$; as does $\langle ^{38}/_{81},\ ^{39}/_{81}\rangle$, representing $[\text{Chips}, \text{Chips}, \text{Fish}] +\!\!+ [\text{EOF}]$. This message ends in EOF, and so is the same as the message that our friend encoded. We have successfully decoded the message. □

The arithmetic decoder continues to emit symbols until the whole message has been recovered. The question then arises of how the decoder can know when it has decoded a complete message. The commonest method in the literature is to use of a special EOF symbol, as was the case in the preceding example. Another common choice is to record the length of the message and decode precisely that many symbols. Which method is used is largely a matter of taste. The first method may be more useful when the length of the message to be compressed is not known in advance, while the second might be better if one wishes to check in advance that there is sufficient space to decode the entire message. Throughout this thesis, we follow the literature and assume that all messages end in the EOF character.

Although the idea behind arithmetic coding is strikingly simple, practical implementations have earned themselves a deserved reputation for being difficult to understand and reason about. Perhaps as a result of this, there have been only a few attempts (eg. [6]) to produce a rigorous proof of the method's correctness. This chapter aims to rectify this omission by producing a formal derivation of the method. Along the way we find ourselves able to create a much faster algorithm than Witten, Neal and Cleary's original program [70]. (Experimental results proving this claim are included in Section 4.5.) Previous attempts at speed improvements [45] have come at a high cost in the algorithm's reduced effectiveness.

3.2 A Definition of Arithmetic Coding

A *model* is an association between symbols and disjoint intervals in $\langle 0,\, 1\rangle$. Formally, we treat it as an abstract type, with functions

$$\mathsf{adapt} :: \text{Model} \to \text{Symbol} \to \text{Model}$$

$$\mathsf{lookup} :: \text{Model} \to \text{Symbol} \to \text{Interval}$$

$$\mathsf{symbol} :: \text{Model} \to \text{Real} \to \text{Symbol}$$

satisfying, for all models m and for all symbols x and y,

1. $\mathsf{lookup}\ m\ x \subseteq \langle 0,\, 1\rangle$
2. $\mathsf{symbol}\ m\ e = x \iff e \in \mathsf{lookup}\ m\ x$

Given a symbol x, we can find the associated interval by evaluating $\mathsf{lookup}\ m\ x$. Given a real number e, we can find the associated symbol by evaluating $\mathsf{symbol}\ m\ e$. These functions are the mainstay of the encoder and decoder.

If the interval I represents the message xs, and the model m is to be used for encoding the next letter x, then the interval $I \otimes \mathsf{lookup}\ m\ x$ represents the message $xs +\!\!+ [x]$. This is the mechanism behind the arithmetic encoder. Observe that the intervals $I \otimes \mathsf{lookup}\ m\ x$ are disjoint subsets of I (by Proposition 2.2.8).

If x is as yet unknown, but we have that $e \in I \otimes \mathsf{lookup}\ m\ x$, then Proposition 2.2.8 says that $I^{-1} \otimes e \in \mathsf{lookup}\ m\ x$. By the definition of a model, therefore, $x = \mathsf{symbol}\ m\ (I^{-1} \otimes e)$. We can use this fact to invert arithmetic coding.

Having encoded or decoded a symbol, we will generally want to *adjust* the model for subsequent symbols. This is the purpose of adapt, which enables the compression algorithm to adapt itself to more closely fit the kind of data being encoded. For this reason, our models are generally known as *adaptive models* in the literature. Authors also refer to *static models* in which no adaptation can occur, although these have limited usefulness. Static models are

often used with Huffman coding, since each alteration to the model requires the Huffman tree to be re-built from scratch—a slow process. The theory developed in this thesis absolutely requires the use of an adaptive model, and so we use only the more general concept.

3.2.1 Example. Returning to the food example of the previous section, we can define a model, $\mathsf{foodmodel}$, by

$\mathsf{lookup\ foodmodel}\ x$

$\quad |\ x == \text{Fish} \quad = \langle 0,\ 1/3 \rangle$

$\quad |\ x == \text{Chips} = \langle 1/3,\ 2/3 \rangle$

$\quad |\ x == \text{EOF} \quad = \langle 2/3,\ 1 \rangle$

$\mathsf{symbol\ foodmodel}\ e$

$\quad |\ e \in \langle 0,\ 1/3 \rangle \quad = \text{Fish}$

$\quad |\ e \in \langle 1/3,\ 2/3 \rangle = \text{Chips}$

$\quad |\ e \in \langle 2/3,\ 1 \rangle \quad = \text{EOF}$

$\mathsf{adapt\ foodmodel}\ x = \mathsf{foodmodel}$

If I is the interval representing the message xs, then

$$I \otimes \mathsf{lookup\ foodmodel}\ \text{Fish} = I \otimes \langle 0,\ 1/3 \rangle\ ,$$

which is the lowest third of I. This is the interval representing the message $xs ++ [\text{Fish}]$, as in Example 3.1.1. Similarly, $I \otimes \mathsf{lookup\ foodmodel}\ \text{Chips}$ is the middle third of I, and $I \otimes \mathsf{lookup\ foodmodel}\ \text{EOF}$ is the top third of I. □

To carry out the actual encoding of messages, we define new functions, encode and decode. They are constructed in precisely the manner discussed earlier.

$$\mathsf{encode} :: \text{Model} \to \text{Interval} \to [\text{Symbol}] \to \text{Interval}$$
$$\mathsf{encode}\ m\ I\ [\,] \qquad = I$$
$$\mathsf{encode}\ m\ I\ (x : xs) = \mathsf{encode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ xs$$

$$\mathsf{decode} :: \text{Model} \to \text{Interval} \to \text{Real} \to [\text{Symbol}]$$
$$\mathsf{decode}\ m\ I\ e = x : xs$$
$$\quad \textbf{where}\ x = \mathsf{symbol}\ m\ (I^{-1} \otimes e)$$
$$\qquad\quad xs = \mathsf{decode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ e$$

In this definition, I is the interval representing the part of the message that has already been encoded, and m is the model to be used for the remainder of the encoding.

3.2.2 Proposition. Let m be a model, and let I be an interval. Let xs be a message, and let e be a real number. If $e \in I$, then

$$e \in \mathsf{encode}\ m\ I\ xs \iff xs \preceq \mathsf{decode}\ m\ I\ e.$$

Proof. To prove the result, we will need the auxiliary fact that $\mathsf{encode}\ m\ I\ xs \subseteq I$. We use induction on the length of xs. The base case follows immediately from the definition of encode. The inductive step is as follows:

$$\mathsf{encode}\ m\ I\ (x : xs)$$
$$= \quad \{\,\text{Definition}\,\}$$
$$\mathsf{encode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ xs$$
$$\subseteq \quad \{\,\text{Induction}\,\}$$

$$I \otimes \mathsf{lookup}\ m\ x$$

$\subseteq$ { Since $\mathsf{lookup}\ m\ x \subseteq \langle 0,\ 1\rangle$, using Proposition 2.2.8 }

$$I$$

We use induction on the length of xs for the main result. The base case is trivial. For the inductive step, we reason thus:

$$e \in \mathsf{encode}\ m\ I\ (x : xs)$$

$\Leftrightarrow$ { Definition of encode }

$$e \in \mathsf{encode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ xs$$

$\Leftrightarrow$ { Since $\mathsf{encode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ xs \subseteq I \otimes \mathsf{lookup}\ m\ x$ }

$$e \in \mathsf{encode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ xs \wedge e \in I \otimes \mathsf{lookup}\ m\ x$$

$\Leftrightarrow$ { Induction }

$$xs \preceq \mathsf{decode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ e \wedge e \in I \otimes \mathsf{lookup}\ m\ x$$

$\Leftrightarrow$ { Proposition 2.2.8 }

$$xs \preceq \mathsf{decode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ e \wedge I^{-1} \otimes e \in \mathsf{lookup}\ m\ x$$

$\Leftrightarrow$ { Definition of a model }

$$xs \preceq \mathsf{decode}\ (\mathsf{adapt}\ m\ x)\ (I \otimes \mathsf{lookup}\ m\ x)\ e \wedge x = \mathsf{symbol}\ m\ (I^{-1} \otimes e)$$

$\Leftrightarrow$ { Definition of decode }

$$x : xs \preceq \mathsf{decode}\ m\ I\ e$$ □

3.3 Input and Output

We have seen that the output from an arithmetic coder is an interval. Most of the time, this is not useful. Instead, we need to store our encoded data as a sequence of letters from some finite alphabet. For convenience, and without loss of generality, we will assume that the alphabet used is $\{0, 1, \ldots, \mathsf{base} - 1\}$, where base is a fixed constant. The letters in this alphabet will be given the following type:

$$\textbf{type}\ \text{Digit} = \text{Integer}\ (0, 1, \ldots, \mathsf{base} - 1).$$

In Witten, Neal and Cleary's coder, $\mathsf{base} = 2$, and the list of bits has to be packed into bytes before it can be written to disk. This introduces an avoidable (and significant) overhead: we can take $\mathsf{base} = 256$ and write bytes directly. This is the first optimisation that leads to the improved coder.

To perform the translation between intervals and lists of digits, we introduce two new functions, output and input:

$$\mathsf{output} :: \text{Interval} \rightarrow [\text{Digit}]$$
$$\mathsf{input} :: [\text{Digit}] \rightarrow \text{Real}.$$

and our compression and decompression algorithms will be given by

$$\mathsf{compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}]$$
$$\mathsf{compress}\ m\ xs = (\mathsf{output} \circ \mathsf{encode}\ m\ I)\ xs$$

$$\mathsf{decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$$
$$\mathsf{decompress}\ m\ ys = (\mathsf{decode}\ m\ I \circ \mathsf{input})\ ys$$

for some fixed I. (It turns out that we want $I = \langle 0,\ 1 \rangle$. See section 3.4.) For any model m,

and for any message xs, we require[1] that

$$xs \preceq (\mathsf{decompress}\ m \circ \mathsf{compress}\ m)\ xs.$$

By Proposition 3.2.2, we can achieve this by requiring that $(\mathsf{input} \circ \mathsf{output})\ I \in I$ for all intervals $I \subseteq \langle 0,\ 1\rangle$.

One obvious way to map lists of digits to real numbers in $\langle 0,\ 1\rangle$ is by using the standard radix notation. This immediately suggests a suitable definition for input.

3.3.1 Definition. We define

$$\begin{aligned}
\mathsf{input}\ [\,] &= 0\\
\mathsf{input}\ (d : ds) &= (d + \mathsf{input}\ ds)/\mathsf{base}
\end{aligned}$$

□

At present there is not enough information to derive a unique matching output function: instead, some creativity will be needed. For example, taking $\mathsf{base} = 10$ and $I = \langle 0.25,\ 0.5\rangle$, we could take $\mathsf{output}\ I = [3]$, $[4]$, or even $[2, 5, 0]$. Since we are interested in data compression, it is natural to let $\mathsf{output}\ I$ be a *minimal-length* list of digits such that $(\mathsf{input} \circ \mathsf{output})\ I \in I$—in this example, we could take $\mathsf{output}\ I = [3]$.

In practice, it is useful to have the length of $\mathsf{output}\ I$ depend only on the width of I, and not on $\mathsf{left}\ I$ or $\mathsf{right}\ I$: this enables us to concatenate other information to the compressed data stream without the need for any messy re-positioning of file pointers. For example, the intervals $\langle 0.99,\ 1\rangle$ and $\langle 0.9,\ 0.91\rangle$ both have the same width, 0.01. However, the shortest number in $\langle 0.99,\ 1\rangle$ is 0.99, while the shortest number in $\langle 0.9,\ 0.91\rangle$ is 0.9.

So how long does $\mathsf{output}\ I$ need to be?

---

[1] The decoder produces some extra rubbish after completing the decoding, and so we can't require that $xs = (\mathsf{decompress}\ m \circ \mathsf{compress}\ m)\ xs$ just yet. That's what the special EOF symbol is for.

3.3.2 Lemma. Let $0 < w \leq 1$, and let $n = \lceil -\log_{\mathsf{base}} w \rceil$. Then every interval $I \subseteq \langle 0, 1\rangle$ of width w contains a number that is at most n digits long.

Proof. There are base^n numbers in $\langle 0, 1\rangle$ that are up to n digits long. Each is separated from its nearest neighbours by a distance of $1/\mathsf{base}^n$. If I contained no numbers that were n digits long, then those numbers would be separated by more than the width of I: that is, $1/\mathsf{base}^n > w$. We show that actually $1/\mathsf{base}^n \leq w$, and so I contains at least one number that is up to n digits long.

$$-\log_{\mathsf{base}} w \leq \lceil -\log_{\mathsf{base}} w \rceil$$

$\Leftrightarrow$ { Definition of n }

$$-\log_{\mathsf{base}} w \leq n$$

$\Leftrightarrow$ { Arithmetic }

$$1/\mathsf{base}^n \leq w$$

This proves the claim. □

3.3.3 Lemma. Let $0 < w \leq 1$, and let $n = \lceil -\log_{\mathsf{base}} w \rceil$. Then there is an interval $I \subseteq \langle 0, 1\rangle$ of width w whose *shortest* number is n digits long.

Proof. Let $I = \langle 1 - w, 1\rangle$. The highest number with only $n-1$ digits is $1 - \mathsf{base}^{-(n-1)}$. We show that $1 - \mathsf{base}^{-(n-1)} < \mathsf{left}\ I$, and so I contains no number of $n - 1$ digits or fewer.

$$\log_{\mathsf{base}} w < \lfloor \log_{\mathsf{base}} w \rfloor + 1$$

$\Leftrightarrow$ { Lemma 2.3.1 }

$$\log_{\mathsf{base}} w < -(\lceil -\log_{\mathsf{base}} w \rceil - 1)$$

$\Leftrightarrow$ { Definition of n }

$$\log_{\mathsf{base}} w < -(n-1)$$

$\Leftrightarrow$ { Arithmetic }

$$1 - \mathsf{base}^{-(n-1)} < 1 - w$$

$\Leftrightarrow$ { $\mathsf{left}\ I = 1 - w$ }

$$1 - \mathsf{base}^{-(n-1)} < \mathsf{left}\ I$$

By the previous lemma, I contains at least one number that is at most n digits long, and this reasoning shows that I contains no numbers that have fewer than n digits. The shortest number in I is therefore exactly n digits long. □

By the previous two lemmas, we need to ensure that

$$\mathsf{length}\ (\mathsf{output}\ I) = \lceil -\log_{\mathsf{base}} w \rceil$$

if the interval I has width w. Returning to our example involving intervals of width 0.01, we see that $\lceil -\log_{10} 0.01 \rceil = 2$, so $\mathsf{output}\ \langle 0.99,\ 1 \rangle = [9,9]$, while $\mathsf{output}\ \langle 0.9,\ 0.91 \rangle = [9,0]$.

One of the effects of $\mathsf{output}\ I$ will be to convert a real number in I to a list of digits, and we define a new function, digits, to do this. Once again, we use our intuitive understanding of the radix notation.

$$\begin{aligned}
&\mathsf{digits} :: \text{Real} \rightarrow [\text{Digit}] \\
&\mathsf{digits}\ x = d : ds \\
&\qquad \textbf{where}\ d \ = \lfloor x \times \mathsf{base} \rfloor \\
&\qquad\qquad\quad ds = \mathsf{digits}\ (x \times \mathsf{base} - d)
\end{aligned}$$

For example, $1/7 = 0.142857142857\ldots$, while $\mathsf{digits}\ 1/7 = [1,4,2,8,5,7,1,4,2,8,5,7,\ldots]$ (assuming $\mathsf{base} = 10$).

For each interval $I \subseteq \langle 0,\, 1\rangle$, we have to find a real number $x \in I$ whose digits we can take to be the result of evaluating $\mathsf{output}\ I$. Which number x to use is largely a matter of choice.

3.3.4 Lemma. Let $I \subseteq \langle 0,\, 1\rangle$, and let $n = \lceil -\log_{\mathsf{base}}(\mathsf{width}\ I)\rceil$. Let

$$x = \frac{\lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil - 1}{\mathsf{base}^n}.$$

Then $x \in I$.

Proof. Let $w = \mathsf{width}\ I$. Then

$$-\log_{\mathsf{base}} w \leq \lceil -\log_{\mathsf{base}} w \rceil$$

$\Leftrightarrow$ { Arithmetic }

$$\mathsf{base}^{-\log_{\mathsf{base}} w} \leq \mathsf{base}^{\lceil -\log_{\mathsf{base}} w \rceil}$$

$\Leftrightarrow$ { Arithmetic and definition of n }

$$1/w \leq \mathsf{base}^n$$

$\Leftrightarrow$ { Arithmetic }

$$-w \times \mathsf{base}^n \leq -1$$

$\Rightarrow$ { Since $\mathsf{right}\ I \times \mathsf{base}^n \leq \lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil$ }

$$\mathsf{right}\ I \times \mathsf{base}^n - w \times \mathsf{base}^n \leq \lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil - 1$$

$\Leftrightarrow$ { Arithmetic }

$$(\mathsf{right}\ I - w) \times \mathsf{base}^n \leq \lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil - 1$$

$\Leftrightarrow$ { Since $\lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil - 1 < \mathsf{right}\ I \times \mathsf{base}^n$ }

$$(\mathsf{right}\ I - w) \times \mathsf{base}^n \leq \lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil - 1 < \mathsf{right}\ I \times \mathsf{base}^n$$

$\Leftrightarrow$ { Arithmetic }

$$\mathsf{right}\ I - w \leq \frac{\lceil \mathsf{right}\ I \times \mathsf{base}^n \rceil - 1}{\mathsf{base}^n} < \mathsf{right}\ I$$

$\Leftrightarrow$ { Definition of x and w }

$$\mathsf{left}\ I \leq x < \mathsf{right}\ I$$

$\Leftrightarrow$ { Set theory }

$$x \in I$$ □

Observe that $x \times \mathsf{base}^n$ is an integer. Therefore, the only non-zero digits of x occur in the first n places. To specify x exactly, it is only necessary to give the first n digits, since the rest are all zero. We can put this more formally by saying that

$$x = \mathsf{input}\ (\mathsf{take}\ n\ (\mathsf{digits}\ x)).$$

This suggests a suitable definition for output.

3.3.5 Definition. Let $I \subseteq \langle 0,\ 1\rangle$. We define

$\mathsf{output}\ I = \mathsf{take}\ n\ (\mathsf{digits}\ x)$

where $n = \lceil -\log_{\mathsf{base}} w \rceil$

$w = \mathsf{width}\ I$

$x = \lceil \mathsf{right}\ I \times \mathsf{base}^n - 1 \rceil / \mathsf{base}^n$ □

3.3.6 Proposition. With these definitions of input and output, $(\mathsf{input} \circ \mathsf{output})\ I \in I$ for all intervals $I \subseteq \langle 0,\ 1\rangle$.

Proof. This follows easily:

$(\mathsf{input} \circ \mathsf{output})\ I$

$= \{\text{Definition of } \mathsf{output}\}$

$\mathsf{input}\ (\mathsf{take}\ n\ (\mathsf{digits}\ x))$

$= \{\text{Previous observation}\}$

x

$\in \{\text{Lemma 3.3.4}\}$

I □

As they stand, our input and output functions are not as useful as they might be for the subsequent analysis. In order to reason about these functions, we will need to explore how they interact with the interval operations given in Chapter 2, and to express them in a more friendly form.

3.3.7 Proposition. Let

$$\mathsf{interval} :: \text{Digit} \rightarrow \text{Interval}$$
$$\mathsf{interval}\ d = \langle d/\mathsf{base},\ (d+1)/\mathsf{base}\rangle$$

Let d be a digit, ds be a list of digits, and $I \subseteq \langle 0,\ 1\rangle$. Then

$$\mathsf{input}\ (d : ds) = \mathsf{interval}\ d \otimes \mathsf{input}\ ds$$

and

$$\mathsf{output}\ (\mathsf{interval}\ d \otimes I) = d : \mathsf{output}\ I$$

Proof. Given in Section C.2 □

3.4 Data Compression with Arithmetic Coding

The aim of any data compression algorithm is to minimise the expected length of the compressed file. This is achieved by ensuring that very likely messages get compressed into shorter output sequences than very unlikely messages. The amount of output produced depends upon the width of the interval produced by the encoder; this in turn depends upon the choice of model used. A good choice of model is crucial for the success of arithmetic coding.

Since we don't know in advance what message is to be compressed, we can consider it to be a random variable, XS. The expected length of the compressed file—the quantity that we wish to minimise—is then given by the following expression:

$$\begin{aligned}
& \mathbb{E}\ (\mathsf{length}\ (\mathsf{compress}\ m\ XS)) \\
=\ & \{\ \text{Definition of } \mathsf{compress}\ \} \\
& \mathbb{E}\ ((\mathsf{length} \circ \mathsf{output} \circ \mathsf{encode}\ m\ I)\ XS) \\
=\ & \{\ \text{Definition of expectation}\ \} \\
& \sum_{xs} \mathbb{P}\ (xs = XS) \times (\mathsf{length} \circ \mathsf{output} \circ \mathsf{encode}\ m\ I)\ xs \\
\approx\ & \{\ \text{Definition 3.3.5}\ \} \\
& \sum_{xs} \mathbb{P}\ (xs = XS) \times (-\log_{\mathsf{base}} \circ \mathsf{width} \circ \mathsf{encode}\ m\ I)\ xs
\end{aligned}$$

By a standard result[2], this is minimised precisely when

$$\mathsf{width}\ (\mathsf{encode}\ m\ I\ xs) = \mathbb{P}\ (xs = XS)$$

for all messages xs. Recall that we are assuming that all messages end with the special EOF symbol that occurs nowhere else. Therefore, $xs = XS \Leftrightarrow xs \preceq XS$, and so this equation is

[2] For a proof, see [66], page 5.

satisfied whenever

$$\mathsf{width}\ (\mathsf{encode}\ m\ I\ xs) = \mathbb{P}\ (xs \preceq XS) \qquad (*)$$

We have to design a model so that $(*)$ holds. We calculate the width of the interval representing an encoded message, and we derive equation $(*)$ as a corollary to this result. Note that foldl is a standard Haskell function, defined in Appendix B, and that foldl adapt m ys gives us the model in use after the string ys has been processed.

3.4.1 Theorem. Suppose that, for all messages ys and symbols x,

$$\mathsf{width}\ (\mathsf{lookup}\ (\mathsf{foldl\ adapt}\ m\ ys)\ x) = \mathbb{P}\ (ys +\!\!+ [x] \preceq XS \mid ys \preceq XS)\,.$$

Then, for all messages xs and ys,

$$\mathsf{width}\ (\mathsf{encode}\ (\mathsf{foldl\ adapt}\ m\ ys)\ I\ xs) = \mathsf{width}\ I \times \mathbb{P}\ (ys +\!\!+ xs \preceq XS \mid ys \preceq XS)\,.$$

Proof. Given in Section C.2. □

3.4.2 Corollary. Suppose that, for all messages ys and symbols x,

$$\mathsf{width}\ (\mathsf{lookup}\ (\mathsf{foldl\ adapt}\ m\ ys)\ x) = \mathbb{P}\ (ys +\!\!+ [x] \preceq XS \mid ys \preceq XS)\,.$$

Then, taking $I = \langle 0,\, 1\rangle$,

$$\mathsf{width}\ (\mathsf{encode}\ m\ I\ xs) = \mathbb{P}\ (xs \preceq XS)$$

for all complete messages xs.

Proof. Take $ys = [\,]$ in Theorem 3.4.1. □

We now know how to build a model that minimises the expected length of the arithmetic coder's output, and we have also completed the definition of compress and decompress by finding the fixed interval I:

$$\mathsf{compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}]$$
$$\mathsf{compress}\ m\ xs = (\mathsf{output} \circ \mathsf{encode}\ m\ \ \langle 0,\, 1\rangle)\ xs$$

$$\mathsf{decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$$
$$\mathsf{decompress}\ m\ ys = (\mathsf{decode}\ m\ \ \langle 0,\, 1\rangle \circ \mathsf{input})\ ys$$

The key to using arithmetic coding for data compression is to produce an accurate model for the data that is being encoded. This is fairly easy to do; for example, we could maintain a count of the frequencies with which different letters make an appearance, and base our probability estimates on that.

However, designing a really *good* model invariably requires a combination of clever algorithms and a lot of computation. There is always a tradeoff between efficiency and effectiveness. This is still a very active area of data compression research.

At last, we know how to use arithmetic coding to achieve data compression. There are still a number of issues that need to be resolved, the most important of which is the need to get away from using unimplementable real-valued arithmetic.

3.5 Rational-Valued Models

As things stand, our definition of a model is of limited practical value. We are using real arithmetic when integer arithmetic would suffice (and real arithmetic can't be implemented on a finite computer). Furthermore, no real-world models would require arbitrary real numbers. Instead, we will always assume that the intervals returned by the lookup function have

rational endpoints:

$$\mathsf{lookup}\ m\ x = \langle n_1/d,\, n_2/d\rangle$$

for some integers n_1, n_2 and d. We can assume without loss of generality that d depends only on the model m and not on the letter x being encoded.

The symbol function can also be adapted to work with integers. Observe that

$$e \in \langle n_1/d,\, n_2/d\rangle \iff \lfloor d \times e\rfloor/d \in \langle n_1/d,\, n_2/d\rangle\,.$$

For any real number e, it is easy to calculate an integer $n = \lfloor d \times e \rfloor$ such that

$$\mathsf{symbol}\ m\ e = \mathsf{symbol}\ m\ (n/d).$$

3.5.1 Definition. A model is *rational-valued* if there are functions

$$\begin{aligned} \mathsf{denominator} &:: \mathrm{Model} \to \mathrm{Integer} \\ \mathsf{int_lookup} &:: \mathrm{Model} \to \mathrm{Symbol} \to (\mathrm{Integer}, \mathrm{Integer}) \\ \mathsf{int_symbol} &:: \mathrm{Model} \to \mathrm{Integer} \to \mathrm{Symbol} \end{aligned}$$

such that (taking $d = \mathsf{denominator}\ m$)

- $\mathsf{int_lookup}\ m\ x = (n_1, n_2) \implies \mathsf{lookup}\ m\ x = \langle n_1/d,\, n_2/d\rangle$
- $\mathsf{int_symbol}\ m\ n = x \implies \mathsf{symbol}\ m\ (n/d) = x$

for all models m, for all letters x, and for all integers n.

By contrast, we will refer to the arbitrary models discussed in previous sections as *real-valued* models. □

An easy consequence of this definition is that

$$\mathsf{int_lookup}\ m\ x = (n_1, n_2) \wedge \mathsf{int_symbol}\ m\ n = y \implies (x = y \Leftrightarrow n_1 \le n < n_2).$$

3.5.2 Example. The food model, discussed at the beginning of the chapter, is a rational-valued model. We take

$$\mathsf{denominator\ foodmodel} = 3$$

$$\mathsf{int_lookup\ foodmodel}\ \mathrm{Fish} = (0, 1)$$
$$\mathsf{int_lookup\ foodmodel}\ \mathrm{Chips} = (1, 2)$$
$$\mathsf{int_lookup\ foodmodel}\ \mathrm{EOF} = (2, 3)$$

$$\mathsf{int_symbol\ foodmodel}\ 0 = \mathrm{Fish}$$
$$\mathsf{int_symbol\ foodmodel}\ 1 = \mathrm{Chips}$$
$$\mathsf{int_symbol\ foodmodel}\ 2 = \mathrm{EOF}$$

□

All the models we consider in this thesis will be rational-valued. By representing rational numbers as a pair of integers, we find that our arithmetic coder could, in principle, be implemented as it now stands. It would, however, be horribly inefficient, requiring integers of unbounded size. In order to derive a practical algorithm, we need to make a number of approximations.

3.6 Approximate Arithmetic Coding

Our current arithmetic coding algorithm requires the use of arbitrary-precision arithmetic. This greatly slows our compression and decompression algorithms, and costs a great deal of memory. Therefore, we would like to introduce some alterations to our algorithm to enable us to use only finite-precision integers to represent intervals. Since we can't represent all possible intervals using such integers (there aren't enough integers) we will have to make some approximations.

Of course, an approximate arithmetic coder will be slightly less effective than the exact version, but this is a small price to pay for practicality.

The problem with building an approximate arithmetic coder is that we have to change the definition of a coder: there is no mention of approximations in the work that has gone before. It looks like our earlier work will have to be done all over again to allow for the approximations that we're about to make. In fact, we will find that we can express approximate coders in terms of exact coders, and so our previous effort is saved.

In the exact coder, we used the $\otimes$ operator to perform exact arithmetic on real-valued models. For the approximate coder, we will introduce a new operator, $\boxtimes_d$, that performs approximate arithmetic on rational-valued models with denominator d. The program we are aiming for will look like:

$$\begin{aligned}
&\mathsf{approx_encode} :: \text{Model} \to \text{Interval} \to [\text{Symbol}] \to \text{Interval} \\
&\mathsf{approx_encode}\ m\ I\ [\,] \qquad\quad = I \\
&\mathsf{approx_encode}\ m\ I\ (x : xs) = \mathsf{approx_encode}\ (\mathsf{adapt}\ m\ x)\ (I \boxtimes_d \mathsf{int_lookup}\ m\ x)\ xs \\
&\qquad \textbf{where}\ d = \mathsf{denominator}\ m
\end{aligned}$$

$$\begin{aligned}
&\mathsf{approx_decode} :: \text{Model} \to \text{Interval} \to \text{Real} \to [\text{Symbol}] \\
&\mathsf{approx_decode}\ m\ I\ e = x : xs \\
&\qquad \textbf{where}\ x = \mathsf{int_symbol}\ m\ (I^{-1} \boxtimes_d e) \\
&\qquad\qquad\quad xs = \mathsf{approx_decode}\ (\mathsf{adapt}\ m\ x)\ (I \boxtimes_d \mathsf{int_lookup}\ m\ x)\ e \\
&\qquad\qquad\quad\ d = \mathsf{denominator}\ m
\end{aligned}$$

Of course, we would like to have that

$$I \boxtimes_d \mathsf{int_lookup}\ m\ x \approx I \otimes \mathsf{lookup}\ m\ x$$

or, more generally,

$$I \boxtimes_d (n_1,\, n_2) \approx I \otimes \langle n_1/d,\, n_2/d \rangle$$

so that the approximate coder is almost as effective as the exact version.

3.6.1 Definition. An *interval approximation* consists of an operator, $\boxtimes_d$, satisfying:

- $I \boxtimes_d (n_1, n_2) \subseteq I$, and
- $e \in I \boxtimes_d (n_1, n_2) \iff n_1 \leq I^{-1} \boxtimes_d e < n_2$

whenever $0 \leq n_1 < n_2 \leq d$. □

Just as $\otimes$ could act on both intervals and real numbers, so $\boxtimes_d$ can also. $I \boxtimes_d (n_1, n_2)$ will be an interval, while $I^{-1} \boxtimes_d e$ will be an integer.

3.6.2 Example. An example of an interval approximation is given by

$$I \boxtimes_d (n_1, n_2) = I \otimes \langle n_1/d,\, n_2/d \rangle \,.$$

This "approximation" is, in fact, identical to the exact algorithm. As such, it isn't terribly useful, but it illustrates the point.

Since $\langle n_1/d,\, n_2/d \rangle \subseteq \langle 0,\, 1 \rangle$, the first part of the definition for an interval approximation is satisfied (by Proposition 2.2.8).

For the second part, we observe that

$$e \in I \boxtimes_d (n_1, n_2)$$

$\Leftrightarrow$ { Definition of $\boxtimes_d$ }

$$e \in I \otimes \langle n_1/d,\, n_2/d \rangle$$

$\Leftrightarrow$ { Proposition 2.2.8 }

$$I^{-1} \otimes e \in \langle n_1/d,\, n_2/d \rangle$$

$\Leftrightarrow$ { Set theory }

$$n_1/d \leq I^{-1} \otimes e < n_2/d$$

$$\Leftrightarrow \quad \{\,\text{Arithmetic}\,\}$$

$$n_1 \leq d \times (I^{-1} \otimes e) < n_2$$

$$\Leftrightarrow \quad \{\,\text{Rule of Floors}\,\}$$

$$n_1 \leq \lfloor d \times (I^{-1} \otimes e) \rfloor < n_2$$

and so we can take

$$I^{-1} \boxtimes_d e = \lfloor d \times (I^{-1} \otimes e) \rfloor.$$

This makes $\boxtimes_d$ an interval approximation. □

Unfortunately, we can't blindly attach an interval approximation into the arithmetic coding algorithm in the way we've described, or all of our previous careful work is invalidated; we don't even have a proof that the approximate encoding can be reversed! What we can do, however, is to adjust our model slightly so that the new model just *happens* to agree with the approximation. This adjustment will be carried out by the function

$$\mathsf{approximate} :: \text{Model} \rightarrow \text{Interval} \rightarrow \text{Model}$$

and we will require that

$$I \otimes \mathsf{lookup}\ (\mathsf{approximate}\ m\ I)\ x = I \boxtimes_d \mathsf{int_lookup}\ m\ x,$$

$$\mathsf{symbol}\ (\mathsf{approximate}\ m\ I)\ (I^{-1} \otimes e) = \mathsf{int_symbol}\ m\ (I^{-1} \boxtimes_d e), \text{ and}$$

$$\mathsf{adapt}\ (\mathsf{approximate}\ m\ I)\ x = \mathsf{approximate}\ (\mathsf{adapt}\ m\ x)\ (I \boxtimes_d \mathsf{int_lookup}\ m\ x),$$

for then, by substituting into the definition of encode and decode, we see that

$$\mathsf{encode}\ (\mathsf{approximate}\ m\ I)\ I\ xs = \mathsf{approx_encode}\ m\ I\ xs, \text{ and}$$

$$\mathsf{decode}\ (\mathsf{approximate}\ m\ I)\ I\ e = \mathsf{approx_decode}\ m\ I\ e,$$

and so we can express the approximate coder in terms of the exact one. Consequently, all our previous results will still hold, including its invertibility and correctness.

We can achieve this by taking

$$\mathsf{lookup}\ (\mathsf{approximate}\ m\ I)\ x = I^{-1} \otimes (I \boxtimes_d \mathsf{int_lookup}\ m\ x), \text{ and}$$

$$\mathsf{symbol}\ (\mathsf{approximate}\ m\ I)\ e = \mathsf{int_symbol}\ m\ (I^{-1} \boxtimes_d (I \otimes e)).$$

It is necessary to check that our approximate model really is a model according to the formal definition. The proof shows why it was necessary to define interval approximations in the way we have. Once we have the proof, then we can see that all the other results we've proved apply as much to approximate arithmetic coders as they do to exact ones.

3.6.3 Theorem. If m is a rational-valued model and I is an interval, then $\mathsf{approximate}\ m\ I$ is also a model.

Proof. By the definition of an interval approximation,

$$I \boxtimes_d \mathsf{int_lookup}\ m\ x \subseteq I$$

$\Leftrightarrow$ { Proposition 2.2.8 }

$$I^{-1} \otimes (I \boxtimes_d \mathsf{int_lookup}\ m\ x) \subseteq I^{-1} \otimes I = \langle 0,\ 1 \rangle$$

$\Leftrightarrow$ { Definition of $\mathsf{approximate}$ }

$$\mathsf{lookup}\ (\mathsf{approximate}\ m\ I)\ x \subseteq \langle 0,\ 1 \rangle$$

This is the first half of the definition of a model.

We check that the second part of the definition of a model holds.

$$e \in \mathsf{lookup}\ (\mathsf{approximate}\ m\ I)\ x$$

$\Leftrightarrow$ { Definition of $\mathsf{approximate}$ }

$$e \in I^{-1} \otimes (I \boxtimes_d \mathsf{int_lookup}\ m\ x)$$

$\Leftrightarrow$ { Proposition 2.2.8 }

$$I \otimes e \in I \boxtimes_d \mathsf{int_lookup}\ m\ x$$

$\Leftrightarrow$ { Taking $\mathsf{int_lookup}\ m\ x = (n_1, n_2)$ }

$$I \otimes e \in I \boxtimes_d (n_1, n_2)$$

$\Leftrightarrow$ { Definition of interval approximation }

$$n_1 \leq I^{-1} \boxtimes_d (I \otimes e) < n_2$$

$\Leftrightarrow$ { Since m is a rational-valued model }

$$\mathsf{int_symbol}\ m\ (I^{-1} \boxtimes_d (I \otimes e)) = x$$

$\Leftrightarrow$ { Definition of $\mathsf{approximate}$ }

$$\mathsf{symbol}\ (\mathsf{approximate}\ m\ I)\ e = x$$

Therefore, $\mathsf{approximate}\ m\ I$ really is a model (although not necessarily rational-valued).

□

Finally, the approximate compression and decompression algorithms can be specified as

$$\mathsf{approx_compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}]$$

$$\mathsf{approx_compress}\ m\ xs = (\mathsf{output} \circ \mathsf{approx_encode}\ m\ \langle 0,\ 1\rangle)\ xs$$

$$\mathsf{approx_decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$$

$$\mathsf{approx_decompress}\ m\ ys = (\mathsf{approx_decode}\ m\ \langle 0,\ 1\rangle \circ \mathsf{input})\ ys$$

3.7 Conclusion

In this chapter, we have seen how to build an arithmetic coder. There are a number of details that need to be added before we get a complete, implementable algorithm. In particular, we need to decide on:

- a model for the data that is to be encoded, and
- an interval approximation.

By considering arithmetic coding in the most general setting possible, we have made ourselves aware of all possible options. A thorough exploration of all possibilities enables us to derive the promised faster arithmetic coder.

This chapter has considered all the things that arithmetic coders have in common. In the next, we explore their differences.

Chapter 4

Some Arithmetic Coding Schemes

4.1 Introduction

We've seen how to build arithmetic coders in theory; we now have to make them work in practice. We begin by giving a naïve coder that has a number of serious drawbacks. We show how these problems can be solved, resulting in Witten, Neal and Cleary's arithmetic coder—the canonical example that is referred to in most papers on the subject. Finally, a novel arithmetic coder is introduced that runs significantly faster than those presently in use.

4.2 A First Attempt

In this section, we introduce our first attempt at a practical arithmetic coder. The word "attempt" is used deliberately, since it turns out that this coder doesn't work. However, most of the interesting coders are based on it, and they take various different approaches to fixing the problems that we encounter.

The major obstacle to practical arithmetic coding is the use of arbitrary-precision num-

bers, both for representing the interval I and the real number e in our encoding and decoding functions. The space required to process a message is therefore unbounded, and the time required is prohibitive. Perhaps the most natural way around this is to approximate intervals and reals using only fixed-precision numbers. To do this, we introduce new constructors for the Interval and Real types:

$$\mathsf{Naive_int} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Interval}$$
$$\mathsf{Naive_int}\ a\ b \equiv \left\langle a/\mathsf{base}^N,\ b/\mathsf{base}^N \right\rangle$$

$$\mathsf{Naive_real} :: \text{Integer} \rightarrow [\text{Digit}] \rightarrow \text{Real}$$
$$\mathsf{Naive_real}\ n\ ys \equiv (n + \mathsf{input}\ ys)/\mathsf{base}^N$$

where N is some fixed constant. We will require that $0 \leq a < b \leq \mathsf{base}^N$ and $0 \leq n < \mathsf{base}^N$. These conditions have an important consequence, which is discussed in Section 4.2.3

Since Naive_int and Naive_real are constructors, they can't be reduced to other values in the course of evaluation; instead, they are taken to *represent* intervals and real numbers. The relationship between $\mathsf{Naive_int}\ a\ b$ and $\left\langle a/\mathsf{base}^N,\ b/\mathsf{base}^N \right\rangle$ is one of equivalence, not equality, and the same goes for Naive_real.

4.2.1 Shrinking Intervals

If we are to data-refine all our intervals to pairs of integers, then we will need to choose an interval approximation so that

$$\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2) = \mathsf{Naive_int}\ a'\ b'$$

for some a' and b'. Further, these integers will be bounded by base^N, and so have finite precision.

4.2.1 Theorem. Let

$$\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2) = \mathsf{Naive_int}\ \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor$$

and

$$(\mathsf{Naive_int}\ a\ b)^{-1} \boxtimes_d \mathsf{Naive_real}\ n\ ys = \left\lfloor \frac{d(n-a+1)-1}{b-a} \right\rfloor.$$

Then $\boxtimes_d$ is an interval approximation. Further,

$$\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2) \approx \mathsf{Naive_int}\ a\ b \otimes \langle n_1/d,\, n_2/d \rangle\,,$$

in the sense that

$$0 \leq \mathsf{left}\ (\mathsf{Naive_int}\ a\ b \otimes \langle n_1/d,\, n_2/d \rangle) - \mathsf{left}\ (\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2)) < 1/\mathsf{base}^N$$

and

$$0 \leq \mathsf{right}\ (\mathsf{Naive_int}\ a\ b \otimes \langle n_1/d,\, n_2/d \rangle) - \mathsf{right}\ (\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2)) < 1/\mathsf{base}^N.$$

Proof. Suppose that $0 \leq n_1 < n_2 \leq d$. Then we can see by some simple arithmetic that

$$a \leq \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor \leq \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor \leq b,$$

and so $\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2) \subseteq \mathsf{Naive_int}\ a\ b$. This is the first part of the definition of an interval approximation (Definition 3.6.1).

We can use Definition 3.6.1 and the formula for $\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2)$ to derive the value of $(\mathsf{Naive_int}\ a\ b)^{-1} \boxtimes_d \mathsf{Naive_real}\ n\ ys$.

$$\mathsf{Naive_real}\ n\ ys \in \mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2)$$

$$\Leftrightarrow \quad \{\text{Definition of } \boxtimes_d\}$$

$$\mathsf{Naive_real}\ n\ ys \in \mathsf{Naive_int}\ \lfloor a + n_1(b-a)/d \rfloor\ \lfloor a + n_2(b-a)/d \rfloor$$

$\Leftrightarrow$ { Definition of $\mathsf{Naive_int}$ and $\mathsf{Naive_real}$ }

$$\lfloor a + n_1(b-a)/d \rfloor / \mathsf{base}^N \leq (n + \mathsf{input}\ ys)/\mathsf{base}^N < \lfloor a + n_2(b-a)/d \rfloor / \mathsf{base}^N$$

$\Leftrightarrow$ { Arithmetic }

$$\lfloor a + n_1(b-a)/d \rfloor \leq n + \mathsf{input}\ ys < \lfloor a + n_2(b-a)/d \rfloor$$

$\Leftrightarrow$ { Rule of Floors }

$$\lfloor a + n_1(b-a)/d \rfloor \leq \lfloor n + \mathsf{input}\ ys \rfloor < \lfloor a + n_2(b-a)/d \rfloor$$

$\Leftrightarrow$ { Since $\mathsf{input}\ ys \in \langle 0,\, 1 \rangle$ }

$$\lfloor a + n_1(b-a)/d \rfloor \leq n < \lfloor a + n_2(b-a)/d \rfloor$$

$\Leftrightarrow$ { Plus 1 Rule }

$$\lfloor a + n_1(b-a)/d \rfloor < n + 1 \leq \lfloor a + n_2(b-a)/d \rfloor$$

$\Leftrightarrow$ { Rule of Floors[1] }

$$a + n_1(b-a)/d < n + 1 \leq a + n_2(b-a)/d$$

$\Leftrightarrow$ { Arithmetic }

$$n_1(b-a) < d(n-a+1) \leq n_2(b-a)$$

$\Leftrightarrow$ { Plus 1 Rule }

$$n_1(b-a) \leq d(n-a+1) - 1 < n_2(b-a)$$

$\Leftrightarrow$ { Arithmetic }

$$n_1 \leq \frac{d(n-a+1)-1}{b-a} < n_2$$

[1]Observe that the Rule of Floors would be inapplicable without the previous application of the Plus 1 Rule.

$$\Leftrightarrow \quad \{\text{Rule of Floors}\}$$

$$n_1 \leq \left\lfloor \frac{d(n-a+1)-1}{b-a} \right\rfloor < n_2$$

$$\Leftrightarrow \quad \{\text{Definition of } \boxtimes_d\}$$

$$n_1 \leq (\mathsf{Naive_int}\ a\ b)^{-1} \boxtimes_d \mathsf{Naive_real}\ n\ ys < n_2$$

and so $\boxtimes_d$ really is an interval approximation.

For the last claim, it is easy to see that

$$\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2)$$

$$= \quad \{\text{Definition of } \boxtimes_d\}$$

$$\mathsf{Naive_int}\ \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor$$

$$\equiv \quad \{\text{Definition of } \mathsf{Naive_int}\}$$

$$\left\langle \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor \Big/ \mathsf{base}^N,\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor \Big/ \mathsf{base}^N \right\rangle$$

and

$$\mathsf{Naive_int}\ a\ b \otimes \langle n_1/d,\ n_2/d \rangle$$

$$\equiv \quad \{\text{Definition of } \mathsf{Naive_int}\}$$

$$\left\langle a/\mathsf{base}^N,\ b/\mathsf{base}^N \right\rangle \otimes \langle n_1/d,\ n_2/d \rangle$$

$$= \quad \{\text{Arithmetic}\}$$

$$\left\langle \left(a + \frac{n_1(b-a)}{d} \right) \Big/ \mathsf{base}^N,\ \left(a + \frac{n_2(b-a)}{d} \right) \Big/ \mathsf{base}^N \right\rangle$$

from which we see that

$$\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2) \approx \mathsf{Naive_int}\ a\ b \otimes \langle n_1/d,\ n_2/d \rangle\ .$$

as claimed. □

4.2.2 The Complete Algorithm

Whatever architecture we are using, there will be some operations that are more costly than others. The way most computers today are built, fixed-precision addition and subtraction are very cheap, multiplication rather less so, and division is to be avoided as much as possible. Arbitrary-precision arithmetic is usually prohibitively expensive for practical use, and often has to be implemented in software.

Our choice of interval approximation means that we get a reasonably good encoding, while avoiding those operations that are too expensive. In particular, we use only addition, multiplication and integer division, all of which are usually implemented in fast hardware. The use of an approximation has enabled us to bridge the gap from theory to practice.

The complete algorithms will be given by

$\mathsf{naive_compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}]$

$\mathsf{naive_compress}\ m\ xs = \mathsf{output}\ (\mathsf{approx_encode}\ m\ \langle 0,\ 1\rangle\ \ xs)$

$\mathsf{naive_decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$

$\mathsf{naive_decompress}\ m\ ys = \mathsf{approx_decode}\ m\ \langle 0,\ 1\rangle\ \ (\mathsf{input}\ ys)$

$\mathsf{naive_encode} :: \text{Model} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow [\text{Symbol}] \rightarrow \text{Interval}$

$\mathsf{naive_encode}\ m\ a\ b\ xs = \mathsf{approx_encode}\ m\ I\ xs$

$\quad \textbf{where}\ I = \mathsf{Naive_int}\ a\ b$

$\mathsf{naive_decode} :: \text{Model} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$

$\mathsf{naive_decode}\ m\ a\ b\ n\ ys = \mathsf{approx_decode}\ m\ I\ e$

$\quad \textbf{where}\ I = \mathsf{Naive_int}\ a\ b$

$\quad\quad\quad\quad e = \mathsf{Naive_real}\ n\ ys$

except that we wish to give these functions in terms of naive_encode and naive_decode, not approx_encode and approx_decode.

We can easily see that[2]

$$\langle 0,\ 1\rangle = \mathsf{Naive_int}\ 0\ \mathsf{base}^N$$

and

$$\mathsf{input}\ ys = \mathsf{Naive_real}\ (\mathsf{base}^N \times \mathsf{input}\ (\mathsf{take}\ N\ ys))\ (\mathsf{drop}\ N\ ys)$$

and therefore our naïve compression and decompression algorithms can be given as

$$\mathsf{naive_compress} :: \mathrm{Model} \to [\mathrm{Symbol}] \to [\mathrm{Digit}]$$
$$\mathsf{naive_compress}\ m\ xs = \mathsf{output}\ (\mathsf{naive_encode}\ m\ a\ b\ xs)$$
$$\quad \textbf{where}\ a = 0$$
$$\qquad b = \mathsf{base}^N$$

$$\mathsf{naive_decompress} :: \mathrm{Model} \to [\mathrm{Digit}] \to [\mathrm{Symbol}]$$
$$\mathsf{naive_decompress}\ m\ ys = \mathsf{naive_decode}\ m\ a\ b\ n\ (\mathsf{drop}\ N\ ys)$$
$$\quad \textbf{where}\ a = 0$$
$$\qquad b = \mathsf{base}^N$$
$$\qquad n = \mathsf{base}^N \times \mathsf{input}\ (\mathsf{take}\ N\ ys)$$

If we substitute the definitions of approx_encode, approx_decode, Naive_int, Naive_real and $\boxtimes_d$ into naive_encode and naive_decode, we complete the derivation of the naïve encoder and decoder:

[2] Recall that base^N is the maximum integer that the encoder and decoder can handle.

$$
\begin{aligned}
&\mathsf{naive_encode}\ m\ a\ b\ [\,] = \mathsf{Naive_int}\ a\ b \\
&\mathsf{naive_encode}\ m\ a\ b\ (x : xs) = \mathsf{naive_encode}\ (\mathsf{adapt}\ m\ x)\ a'\ b'\ xs \\
&\quad \textbf{where}\ (n_1, n_2) = \mathsf{int_lookup}\ m\ x \\
&\quad\quad d = \mathsf{denominator}\ m \\
&\quad\quad a' = a + \lfloor n_1(b-a)/d \rfloor \\
&\quad\quad b' = a + \lfloor n_2(b-a)/d \rfloor
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{naive_decode}\ m\ a\ b\ n\ ys = x : xs \\
&\quad \textbf{where}\ x = \mathsf{int_symbol}\ m\ \lfloor (d(n-a+1)-1)/(b-a) \rfloor \\
&\quad\quad xs = \mathsf{naive_decode}\ (\mathsf{adapt}\ m\ x)\ a'\ b'\ n\ ys \\
&\quad\quad (n_1, n_2) = \mathsf{int_lookup}\ m\ x \\
&\quad\quad d = \mathsf{denominator}\ m \\
&\quad\quad a' = a + \lfloor n_1(b-a)/d \rfloor \\
&\quad\quad b' = a + \lfloor n_2(b-a)/d \rfloor
\end{aligned}
$$

These algorithms would be easy to implement in practice, given a suitable model. All arithmetic is performed with finite-precision integers, using reasonably inexpensive operations. There's just one little snag.

4.2.3 A Problem

The major drawback of this algorithm is that the length of the message is severely limited. As the message is encoded and a and b approach each other, the accuracy of the coding reduces further and further, and eventually a and b will collide. If this were to happen, then $\mathsf{Naive_int}\ a\ b = \varnothing$, which is not allowed. The encoded message would be lost, and the coder's state would no longer be well-defined. In practice, it would probably crash or spew out infinite rubbish.

Formally speaking, the problem has been caused because there is no guarantee that $\mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2)$ is in the domain of $\mathsf{Naive_int}$, and so this needs to be proven.

4.2.2 Example. Suppose that $n_1 = 0$, $n_2 = 1$ and $b - a < d$. Then

$$
\begin{aligned}
& \mathsf{Naive_int}\ a\ b \boxtimes_d (n_1, n_2) \\
= & \quad \{ \text{Definition of } \boxtimes_d \} \\
& \mathsf{Naive_int}\ \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor \\
= & \quad \{ \text{Definitions of } n_0 \text{ and } n_1 \} \\
& \mathsf{Naive_int}\ \lfloor a \rfloor\ \left\lfloor a + \frac{(b-a)}{d} \right\rfloor \\
= & \quad \{ \text{Since } b - a < d \} \\
& \mathsf{Naive_int}\ a\ a
\end{aligned}
$$

Therefore, if we attempt to shrink $\mathsf{Naive_int}\ a\ b$, we will get a collision. □

In fact, the conditions necessary to ensure that collisions cannot occur are so strict that the naïve coder is unusable. We haven't really succeeded in limiting the precision required for the encoder, as we now require N to have a very large value to prevent collisions. Furthermore, the value of N puts an upper bound on the amount of information that can be encoded at once using this coder. This is known as the *underflow problem*, and it was first solved by Witten, Neal and Cleary in 1987.

4.3 Witten, Neal and Cleary's Coder

In the long history of arithmetic coding, Witten, Neal and Cleary's coder was the first to be widely used. Their paper [70] included an implementation of the algorithm in C, making

it easy for people to experiment with and use. We can derive their coder using the work of the previous chapter. It also overcomes the drawbacks of the naïve coder. The result of this coder is always a list of bits: $\mathsf{base} = 2$.

4.3.1 Representing Intervals

The insight that enabled Witten et al. to produce a usable algorithm was *incremental encoding.* They observed that whenever a and b approach each other sufficiently closely, then at least one of the following conditions will hold[3].

4.3.1 Conditions.

- $\mathsf{Naive_int}\ a\ b \subseteq \langle 0,\ 1/2 \rangle = \mathsf{interval}\ 0$ — ie. $0 \leq a < b \leq 1/2 \times 2^N$
- $\mathsf{Naive_int}\ a\ b \subseteq \langle 1/2,\ 1 \rangle = \mathsf{interval}\ 1$ — ie. $1/2 \times 2^N \leq a < b \leq 2^N$
- $\mathsf{Naive_int}\ a\ b \subseteq \langle 1/4,\ 3/4 \rangle$ — ie. $1/4 \times 2^N \leq a < b \leq 3/4 \times 2^N$ □

This suggested to them that they should use the following constructors to represent intervals and real numbers instead:

$$\mathsf{Wnc_int} :: [\text{Digit}] \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Interval}$$

$$\mathsf{Wnc_int}\ zs\ c\ a\ b \equiv \mathsf{foldr}\ (\otimes)\ \langle 0,\ 1 \rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4 \rangle^c \otimes \mathsf{Naive_int}\ a\ b$$

$$\mathsf{Wnc_real} :: [\text{Digit}] \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow [\text{Digit}] \rightarrow \text{Real}$$

$$\mathsf{Wnc_real}\ zs\ c\ n\ ys \equiv \mathsf{foldr}\ (\otimes)\ \langle 0,\ 1 \rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4 \rangle^c \otimes \mathsf{Naive_real}\ n\ ys$$

which is to say that

$$\begin{aligned} &\mathsf{Wnc_int}\ [z_0, z_1, \ldots, z_k]\ c\ a\ b \equiv \\ &\quad \mathsf{interval}\ z_0 \otimes \mathsf{interval}\ z_1 \otimes \ldots \otimes \mathsf{interval}\ z_k \otimes \\ &\quad \underbrace{\langle 1/4,\ 3/4 \rangle \otimes \langle 1/4,\ 3/4 \rangle \otimes \ldots \otimes \langle 1/4,\ 3/4 \rangle}_{c \text{ times}} \otimes \mathsf{Naive_int}\ a\ b \end{aligned}$$

[3] Recall that the interval function was defined on page 38.

and

$$\mathsf{Wnc_real}\ [z_0, z_1, \ldots, z_k]\ c\ n\ ys \equiv$$

$$\mathsf{interval}\ z_0 \otimes \mathsf{interval}\ z_1 \otimes \ldots \otimes \mathsf{interval}\ z_k \otimes$$

$$\underbrace{\langle {}^1\!/\!_4,\ {}^3\!/\!_4 \rangle \otimes \langle {}^1\!/\!_4,\ {}^3\!/\!_4 \rangle \otimes \ldots \otimes \langle {}^1\!/\!_4,\ {}^3\!/\!_4 \rangle}_{c \text{ times}} \otimes \mathsf{Naive_real}\ n\ ys.$$

As in the naïve coder, we require that $0 \leq a < b \leq \mathsf{base}^N$. Also like the naïve coder, our compression and decompression algorithms will be defined as

$$\mathsf{wnc_compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}]$$

$$\mathsf{wnc_compress}\ m\ xs = \mathsf{output}\ (\mathsf{approx_encode}\ m\ \langle 0,\ 1 \rangle\ \ xs)$$

$$\mathsf{wnc_decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$$

$$\mathsf{wnc_decompress}\ m\ ys = \mathsf{approx_decode}\ m\ \langle 0,\ 1 \rangle\ \ (\mathsf{input}\ ys)$$

$$\mathsf{wnc_encode} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow$$
$$[\text{Symbol}] \rightarrow \text{Interval}$$

$$\mathsf{wnc_encode}\ m\ zs\ c\ a\ b\ xs = \mathsf{approx_encode}\ m\ I\ xs$$
$$\quad \textbf{where}\ I = \mathsf{Wnc_int}\ zs\ c\ a\ b$$

$$\mathsf{wnc_decode} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow$$
$$\text{Integer} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$$

$$\mathsf{wnc_decode}\ m\ zs\ c\ a\ b\ n\ ys = \mathsf{approx_decode}\ m\ I\ e$$
$$\quad \textbf{where}\ I = \mathsf{Wnc_int}\ zs\ c\ a\ b$$
$$\qquad\qquad e = \mathsf{Wnc_real}\ zs\ c\ n\ ys$$

Once again, we will aim to express all of these functions in terms of wnc_encode and wnc_decode only, eliminating references to approx_encode and approx_decode.

4.3.2 Re-Normalisation

Unlike in the naïve coder, there is not a unique representation of most intervals. For example, the interval $\langle 1/2,\ 3/4\rangle$ can be represented in several ways:

- $\mathsf{Wnc_int}\ []\ 0\ (1/2 \times \mathsf{base}^N)\ (3/4 \times \mathsf{base}^N)$
- $\mathsf{Wnc_int}\ []\ 1\ (1/2 \times \mathsf{base}^N)\ (\mathsf{base}^N)$
- $\mathsf{Wnc_int}\ [1,0]\ 0\ 0\ (\mathsf{base}^N)$

among others.

Which representation of an interval should we use? We choose the representation that ensures that none of Conditions 4.3.1 above holds—in this example, we represent $\langle 1/2,\ 3/4\rangle$ as $\mathsf{Wnc_int}\ [1,0]\ 0\ 0\ (\mathsf{base}^N)$, since $\mathsf{Naive_int}\ 0\ (\mathsf{base}^N)$ is a subset of none of $\langle 0,\ 1/2\rangle$, $\langle 1/2,\ 1\rangle$, and $\langle 1/4,\ 3/4\rangle$.

Suppose we are faced with an interval $\mathsf{Wnc_int}\ zs\ c\ a\ b$, where $\mathsf{Naive_int}\ a\ b$ is sufficiently narrow that at least one of Conditions 4.3.1 holds. Then we have to *re-normalise* it, meaning that we find an alternative representation of the interval in which a and b are widely-separated.

The simplest case to consider is when $\mathsf{Naive_int}\ a\ b \subseteq \langle 1/4,\ 3/4\rangle$, for then

$$
\begin{aligned}
& \mathsf{Wnc_int}\ zs\ c\ a\ b \\
\equiv\ & \{\text{Definition of } \mathsf{Wnc_int}\} \\
& \mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^c \otimes \mathsf{Naive_int}\ a\ b \\
\equiv\ & \{\text{Definition of } \mathsf{Naive_int}\} \\
& \mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^c \otimes \\
& \quad \left\langle a/\mathsf{base}^N,\ b/\mathsf{base}^N\right\rangle \\
=\ & \{\text{Arithmetic}\}
\end{aligned}
$$

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^c \otimes \langle 1/4,\ 3/4\rangle \otimes$$
$$\left\langle 2a/\mathsf{base}^N - 1/2,\ 2b/\mathsf{base}^N - 1/2\right\rangle$$

$= \{$ Arithmetic $\}$

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^{c+1} \otimes$$
$$\left\langle 2a/\mathsf{base}^N - 1/2,\ 2b/\mathsf{base}^N - 1/2\right\rangle$$

$\equiv \{$ Definition of $\mathsf{Naive_int}$, since $0 \leq 2a - 1/2 \times \mathsf{base}^N < 2b - 1/2 \times \mathsf{base}^N \leq \mathsf{base}^N$ $\}$

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^{c+1} \otimes$$
$$\mathsf{Naive_int}\ (2a - 1/2 \times \mathsf{base}^N)\ (2b - 1/2 \times \mathsf{base}^N)$$

$\equiv \{$ Definition of $\mathsf{Wnc_int}$ $\}$

$$\mathsf{Wnc_int}\ zs\ (c+1)\ (2a - 1/2 \times \mathsf{base}^N)\ (2b - 1/2 \times \mathsf{base}^N)$$

$= \{$ Arithmetic, since $\mathsf{base} = 2$ $\}$

$$\mathsf{Wnc_int}\ zs\ (c+1)\ (2a - 2^{N-1})\ (2b - 2^{N-1})$$

Note that $\mathsf{Naive_int}\ (2a - 2^{N-1})\ (2b - 2^{N-1})$ has twice the width of $\mathsf{Naive_int}\ a\ b$, and so the risk of collision has been reduced.

In the case where $\mathsf{Naive_int}\ a\ b \subseteq \langle 0,\ 1/2\rangle$, the reasoning is similar, but with a very important extra step:

$$\mathsf{Wnc_int}\ zs\ c\ a\ b$$

$\equiv \{$ Definition of $\mathsf{Wnc_int}$ $\}$

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^c \otimes \mathsf{Naive_int}\ a\ b$$

$\equiv \{$ Definition of $\mathsf{Naive_int}$ $\}$

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^c \otimes \left\langle a/\mathsf{base}^N,\ b/\mathsf{base}^N\right\rangle$$

$=$ { Arithmetic }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^{c} \otimes \langle 0,\ 1/2\rangle \otimes \left\langle 2a/\mathsf{base}^{N},\ 2b/\mathsf{base}^{N}\right\rangle$$

$=$ { Since $\langle 1/4,\ 3/4\rangle \otimes \langle 0,\ 1/2\rangle = \langle 1/4,\ 1/2\rangle = \langle 0,\ 1/2\rangle \otimes \langle 1/2,\ 1\rangle$ }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 0,\ 1/2\rangle \otimes \langle 1/2,\ 1\rangle^{c} \otimes \left\langle 2a/\mathsf{base}^{N},\ 2b/\mathsf{base}^{N}\right\rangle$$

$=$ { Definition of interval }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ (zs \mathbin{+\!\!+} 0 : \mathsf{replicate}\ c\ 1)) \otimes \left\langle 2a/\mathsf{base}^{N},\ 2b/\mathsf{base}^{N}\right\rangle$$

$\equiv$ { Definition of Naive_int, since $0 \leq 2a < 2b \leq \mathsf{base}^{N}$ }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ (zs \mathbin{+\!\!+} 0 : \mathsf{replicate}\ c\ 1)) \otimes \mathsf{Naive_int}\ (2a)\ (2b)$$

$\equiv$ { Definition of Wnc_int }

$$\mathsf{Wnc_int}\ (zs \mathbin{+\!\!+} 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2a)\ (2b)$$

The case in which Naive_int $a\ b \subseteq \langle 1/2,\ 1\rangle$ is almost identical:

$$\mathsf{Wnc_int}\ zs\ c\ a\ b$$

$\equiv$ { Definition of Wnc_int and Naive_int }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^{c} \otimes \left\langle a/\mathsf{base}^{N},\ b/\mathsf{base}^{N}\right\rangle$$

$=$ { Arithmetic }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^{c} \otimes \langle 1/2,\ 1\rangle \otimes \left\langle 2a/\mathsf{base}^{N} - 1,\ 2b/\mathsf{base}^{N} - 1\right\rangle$$

$=$ { Since $\langle 1/4,\ 3/4\rangle \otimes \langle 1/2,\ 1\rangle = \langle 1/2,\ 3/4\rangle = \langle 1/2,\ 1\rangle \otimes \langle 0,\ 1/2\rangle$ }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/2,\ 1\rangle \otimes \langle 0,\ 1/2\rangle^c \otimes$$
$$\left\langle 2a/\mathsf{base}^N - 1,\ 2b/\mathsf{base}^N - 1\right\rangle$$

$\equiv$ { Definition of Naive_int, interval and Wnc_int }

$$\mathsf{Wnc_int}\ (zs +\!\!+ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2a - \mathsf{base}^N)\ (2b - \mathsf{base}^N)$$

$=$ { Since $\mathsf{base} = 2$ }

$$\mathsf{Wnc_int}\ (zs +\!\!+ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2a - 2^N)\ (2b - 2^N)$$

Therefore, in *any* case where a and b approach each other too closely, we have an escape route: we can use a better representation of the same interval to separate a and b and continue with the encoding.

4.3.3 Re-Normalisation in the Decoder

Whenever we re-normalise the interval in the decoder, we will also have to adjust the values of n and ys, since Wnc_int $zs\ c\ a\ b$ and Wnc_real $zs\ c\ n\ ys$ both depend on the values of zs and c.

The reasoning is not difficult. We consider first the case where Naive_int $a\ b \subseteq \langle 1/4,\ 3/4\rangle$. Then

$$\mathsf{Wnc_real}\ zs\ c\ n\ ys$$

$\equiv$ { Definition of Wnc_real }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^c \otimes \mathsf{Naive_real}\ n\ ys$$

$=$ { Proposition 2.2.3 }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle 1/4,\ 3/4\rangle^{c+1} \otimes \langle 1/4,\ 3/4\rangle^{-1} \otimes \mathsf{Naive_real}\ n\ ys$$

$=$ { Arithmetic }

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle {}^1\!/\!_4,\ {}^3\!/\!_4\rangle^{c+1} \otimes (2 \times \mathsf{Naive_real}\ n\ ys - {}^1\!/\!_2)$$

$=\ \{$ Arithmetic $\}$

$$\mathsf{foldr}\ (\otimes)\ \langle 0,\ 1\rangle\ (\mathsf{map\ interval}\ zs) \otimes \langle {}^1\!/\!_4,\ {}^3\!/\!_4\rangle^{c+1} \otimes \mathsf{Naive_real}\ (2n - {}^1\!/\!_2\mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

$\equiv\ \{$ Definition of $\mathsf{Wnc_real}$ $\}$

$$\mathsf{Wnc_real}\ zs\ (c+1)\ (2n - {}^1\!/\!_2\mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

$=\ \{$ Since $\mathsf{base} = 2$ $\}$

$$\mathsf{Wnc_real}\ zs\ (c+1)\ (2n - 2^{N-1} + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

The other cases are very similar, and we see that

$$\begin{aligned}\mathsf{Wnc_real}\ zs\ c\ n\ ys &= \mathsf{Wnc_real}\ (zs +\!\!+\ 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2n + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)\\ &= \mathsf{Wnc_real}\ (zs +\!\!+\ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2n - 2^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)\end{aligned}$$

Whenever zs and c are altered by the re-normalisation process, therefore, we can update n and ys to match.

4.3.4 The Interval Approximation

As we did with the naïve coder, we wish to choose our interval approximation so that

$$(\mathsf{Wnc_int}\ zs\ c\ a\ b) \boxtimes_d (n_1,\ n_2) = \mathsf{Wnc_int}\ zs'\ c'\ a'\ b'$$

for some values of zs', c', a' and b'. We shrink intervals in exactly the same way as we did with the naïve coder, but we always re-normalise first since that reduces the risk of collisions.

$$\mathsf{Wnc_int}\ zs\ c\ a\ b \boxtimes_d (n_1,\, n_2)$$
$$\mid \mathsf{Naive_int}\ a\ b \subseteq \langle 0,\, 1/2\rangle$$
$$= \mathsf{Wnc_int}\ (zs +\!\!+ 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2a)\ (2b) \boxtimes_d (n_1,\, n_2)$$
$$\mid \mathsf{Naive_int}\ a\ b \subseteq \langle 1/2,\, 1\rangle$$
$$= \mathsf{Wnc_int}\ (zs +\!\!+ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2a - 2^N)\ (2b - 2^N) \boxtimes_d (n_1,\, n_2)$$
$$\mid \mathsf{Naive_int}\ a\ b \subseteq \langle 1/4,\, 3/4\rangle$$
$$= \mathsf{Wnc_int}\ zs\ (c+1)\ (2a - 2^{N-1})\ (2b - 2^{N-1}) \boxtimes_d (n_1,\, n_2)$$
$$\mid \textbf{otherwise}$$
$$= \mathsf{Wnc_int}\ zs\ c\ \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor$$

Interval approximations come in two parts, and we now have to consider the part that is used by the decoder. As before, we re-normalise first. If we assume that $\mathsf{Wnc_int}\ zs\ c\ a\ b$ has been re-normalised, then

$$\mathsf{Wnc_real}\ zs\ c\ n\ ys \in \mathsf{Wnc_int}\ zs\ c\ a\ b \boxtimes_d (n_1,\, n_2)$$

$\Leftrightarrow$ { Definition of $\boxtimes_d$ }

$$\mathsf{Wnc_real}\ zs\ c\ n\ ys \in \mathsf{Wnc_int}\ zs\ c\ \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor$$

$\Leftrightarrow$ { Definitions of Wnc_int and Wnc_real }

$$\mathsf{Naive_real}\ n\ ys \in \mathsf{Naive_int}\ \left\lfloor a + \frac{n_1(b-a)}{d} \right\rfloor\ \left\lfloor a + \frac{n_2(b-a)}{d} \right\rfloor$$

$\Leftrightarrow$ { Result from previous section }

$$n_1 \le \left\lfloor \frac{d(n-a+1)-1}{b-a} \right\rfloor < n_2$$

Therefore, we take

$$(\mathsf{Wnc_int}\ zs\ c\ a\ b)^{-1} \boxtimes_d \mathsf{Wnc_real}\ zs\ c\ n\ ys = \left\lfloor \frac{d(n-a+1)-1}{b-a} \right\rfloor$$

and this gives us our interval approximation

$(\mathsf{Wnc_int}\ zs\ c\ a\ b)^{-1} \boxtimes_d \mathsf{Wnc_real}\ zs\ c\ n\ ys$

$\quad |\ \mathsf{Naive_int}\ a\ b \subseteq \langle 0,\ 1/2\rangle$

$\qquad = (\mathsf{Wnc_int}\ (zs +\!\!+ 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2a)\ (2b))^{-1} \boxtimes_d$

$\qquad\qquad \mathsf{Wnc_real}\ (zs +\!\!+ 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2n + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$

$\quad |\ \mathsf{Naive_int}\ a\ b \subseteq \langle 1/2,\ 1\rangle$

$\qquad = (\mathsf{Wnc_int}\ (zs +\!\!+ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2a - 2^N)\ (2b - 2^N))^{-1} \boxtimes_d$

$\qquad\qquad \mathsf{Wnc_real}\ (zs +\!\!+ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2n - 2^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$

$\quad |\ \mathsf{Naive_int}\ a\ b \subseteq \langle 1/4,\ 3/4\rangle$

$\qquad = (\mathsf{Wnc_int}\ zs\ (c+1)\ (2a - 2^{N-1})\ (2b - 2^{N-1}))^{-1} \boxtimes_d$

$\qquad\qquad \mathsf{Wnc_real}\ zs\ (c+1)\ (2n - 2^{N-1} + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$

$\quad |\ \textbf{otherwise}$

$$= \left\lfloor \frac{d(n-a+1)-1}{b-a} \right\rfloor$$

4.3.5 Putting it all Together

We have all the parts needed to build an arithmetic coder: a data refinement for representing intervals, another one for reals, and an interval approximation. When we put these parts together, we obtain the complete Witten, Neal and Cleary coder. We simply substitute the definitions into wnc_compress and wnc_decompress. The result is given in figures 4.1–4.3.

In practical implementations, the variable zs is stored as a file on disk. Appending information to a file is cheap, and, by Proposition 3.3.7, $\mathsf{output}\ (\mathsf{Wnc_int}\ zs\ c\ a\ b) = zs +\!\!+ \mathsf{output}\ (\mathsf{Wnc_int}\ [\,]\ c\ a\ b)$. Therefore, finishing the encoding is also a cheap operation, consisting of appending a few more bits to the file we've created.

During decoding, the variables zs and c have no effect whatsoever, and so can be removed from the algorithm. They were necessary for the formal derivation, as they ensure that the

$$
\begin{aligned}
&\mathsf{wnc_compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}] \\
&\mathsf{wnc_compress}\ m\ xs = \mathsf{output}\ (\mathsf{wnc_encode}\ m\ zs\ c\ a\ b\ xs) \\
&\qquad \textbf{where}\ zs = [\,] \\
&\qquad\qquad\quad c = 0 \\
&\qquad\qquad\quad a = 0 \\
&\qquad\qquad\quad b = 2^N
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{wnc_decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}] \\
&\mathsf{wnc_decompress}\ m\ ys = \mathsf{wnc_decode}\ m\ zs\ c\ a\ b\ n\ (\mathsf{drop}\ N\ ys) \\
&\qquad \textbf{where}\ zs = [\,] \\
&\qquad\qquad\quad c = 0 \\
&\qquad\qquad\quad a = 0 \\
&\qquad\qquad\quad b = 2^N \\
&\qquad\qquad\quad n = 2^N \times \mathsf{input}\ (\mathsf{take}\ N\ ys)
\end{aligned}
$$

Figure 4.1: The definitions of wnc_compress and wnc_decompress

$$
\begin{aligned}
&\mathsf{wnc_encode}\ m\ zs\ c\ a\ b\ [\,] = \mathsf{Wnc_int}\ zs\ c\ a\ b \\
&\mathsf{wnc_encode}\ m\ zs\ c\ a\ b\ (x : xs) \\
&\quad \mid 0 \leq a < b \leq 1/2 \times 2^N \\
&\qquad = \mathsf{wnc_encode}\ m\ (zs +\!\!+ 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2a)\ (2b)\ (x : xs) \\
&\quad \mid 1/2 \times 2^N \leq a < b \leq 2^N \\
&\qquad = \mathsf{wnc_encode}\ m\ (zs +\!\!+ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2a - 2^N)\ (2b - 2^N)\ (x : xs) \\
&\quad \mid 1/4 \times 2^N \leq a < b \leq 3/4 \times 2^N \\
&\qquad = \mathsf{wnc_encode}\ m\ zs\ (c+1)\ (2a - 2^{N-1})\ (2b - 2^{N-1})\ (x : xs) \\
&\quad \mid \textbf{otherwise} \\
&\qquad = \mathsf{wnc_encode}\ (\mathsf{adapt}\ m\ x)\ zs\ c\ a'\ b'\ xs \\
&\qquad \textbf{where}\ (n_1, n_2) = \mathsf{int_lookup}\ m\ x \\
&\qquad\qquad d = \mathsf{denominator}\ m \\
&\qquad\qquad a' = a + \lfloor n_1(b-a)/d \rfloor \\
&\qquad\qquad b' = a + \lfloor n_2(b-a)/d \rfloor
\end{aligned}
$$

Figure 4.2: The definition of wnc_encode

$$
\begin{aligned}
&\mathsf{wnc_decode}\ m\ zs\ c\ a\ b\ n\ ys\\
&\quad \mid 0 \le a < b \le {}^1\!/_2 \times 2^N\\
&\qquad = \mathsf{wnc_decode}\ m\ (zs +\!\!+\ 0 : \mathsf{replicate}\ c\ 1)\ 0\ (2a)\ (2b)\\
&\qquad\qquad (2n + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)\\
&\quad \mid {}^1\!/_2 \times 2^N \le a < b \le 2^N\\
&\qquad = \mathsf{wnc_decode}\ m\ (zs +\!\!+\ 1 : \mathsf{replicate}\ c\ 0)\ 0\ (2a - 2^N)\ (2b - 2^N)\\
&\qquad\qquad (2n - 2^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)\\
&\quad \mid {}^1\!/_4 \times 2^N \le a < b \le {}^3\!/_4 \times 2^N\\
&\qquad = \mathsf{wnc_decode}\ m\ zs\ (c+1)\ (2a - 2^{N-1})\ (2b - 2^{N-1})\\
&\qquad\qquad (2n - 2^{N-1} + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)\\
&\quad \mid \mathbf{otherwise}\\
&\qquad = x : xs\\
&\qquad \mathbf{where}\ x = \mathsf{int_symbol}\ m\ \lfloor (d(n-a+1)-1)/(b-a) \rfloor\\
&\qquad\qquad xs = \mathsf{wnc_decode}\ (\mathsf{adapt}\ m\ x)\ zs\ c\ a'\ b'\ n\ ys\\
&\qquad\qquad (n_1, n_2) = \mathsf{int_lookup}\ m\ x\\
&\qquad\qquad d = \mathsf{denominator}\ m\\
&\qquad\qquad a' = a + \lfloor n_1(b-a)/d \rfloor\\
&\qquad\qquad b' = a + \lfloor n_2(b-a)/d \rfloor
\end{aligned}
$$

Figure 4.3: The definition of wnc_decode

decoder and encoder are always in the same state at the same time. (In the literature, it is said that the encoder and decoder are "in lockstep".)

4.3.6 The Underflow Problem Again

The Witten, Neal and Cleary coder hasn't managed to vanquish the underflow problem completely, but has reduced it to an acceptable level. Example 4.2.2 still applies, but it is now always the case that $b - a > 1/4 \times 2^N$, and so we can prevent disasters by ensuring that $1/4 \times 2^N \geq d = \mathsf{denominator}\ m$. This can be done by using a sufficiently large value of N together with models whose denominators are bounded. Unlike in the naïve coder, we don't have to choose a really huge value for N, and the length of file that we can encode is no longer limited by underflow.

We've also lied slightly about the use of finite-precision arithmetic. The variable c can grow arbitrarily large in pathological cases. However, the probability of this occurring is vanishingly small and it can be ignored in practice.

4.4 A Fast, General-Purpose Coder

The fast arithmetic coder is presented in much the same manner as the Witten, Neal and Cleary coder: we give a data refinement that is used to represent intervals, and then install it into the compression algorithm.

One of the reasons for the increased speed of the fast coder is that it produces a list of bytes ($\mathsf{base} = 256$), whereas Witten's coder produced a list of bits. Lists of bits have to be packed into bytes before they can be written to disk, and this consumes a surprising amount of time.

The use of integer division in the Witten, Neal and Cleary coder also costs significantly.

We give a novel interval approximation that avoids the need for any divisions or multiplications. In fact, we only need to use addition, subtraction and bit shifts. Moffat et al. [45] attempted a similar idea, but their speedup relied on the fact that they were running their coder on a machine that did multiplication and division in software. As soon as hardware-based arithmetic is used, their advantage vanishes. By contrast, the algorithm presented here outperforms even hardware-based arithmetic without a significant cost in the compression's effectiveness.

Other coders that have used similar methods to increase their speed include the Q-coder and the Z-coder [53, 8]. These are all limited to binary alphabets, whereas the method presented here can cope with alphabets of any size.

The constructors used are

$$\mathsf{Fast_int} :: [\text{Digit}] \rightarrow \text{Digit} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Interval}$$

$$\mathsf{Fast_int}\ zs\ z\ c\ a\ b \equiv \mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{Naive_int}\ a\ b$$

$$\mathsf{Fast_real} :: [\text{Digit}] \rightarrow \text{Digit} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow [\text{Digit}] \rightarrow \text{Real}$$

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys \equiv \mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{Naive_real}\ n\ ys$$

In the Witten, Neal and Cleary coder, we required that $0 \leq a < b \leq \mathsf{base}^N$. Here, we relax this condition somewhat: $0 \leq a < b \leq 2 \times \mathsf{base}^N$ and $0 < b - a \leq \mathsf{base}^N$. As in the Witten, Neal and Cleary coder, zs is the list of digits that have been output from the coder, and c is used to prevent underflow. However, we now have the additional variable, z. There are some circumstances in which the coder may wish to revise the last digit that it produced, and so we keep that digit separately in z.

As with the previous coders, we would like to define:

$\mathsf{fast_compress} :: \text{Model} \rightarrow [\text{Symbol}] \rightarrow [\text{Digit}]$

$\mathsf{fast_compress}\ m\ xs = \mathsf{output}\ (\mathsf{approx_encode}\ m\ \langle 0, 1\rangle\ xs)$

$\mathsf{fast_decompress} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$

$\mathsf{fast_decompress}\ m\ ys = \mathsf{approx_decode}\ m\ \langle 0, 1\rangle\ (\mathsf{input}\ ys)$

$\mathsf{fast_encode} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow \text{Digit} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow$
$\quad [\text{Symbol}] \rightarrow \text{Interval}$

$\mathsf{fast_encode}\ m\ zs\ z\ c\ a\ b\ xs = \mathsf{approx_encode}\ m\ I\ xs$
$\quad \textbf{where}\ I = \mathsf{Fast_int}\ zs\ z\ c\ a\ b$

$\mathsf{fast_decode} :: \text{Model} \rightarrow [\text{Digit}] \rightarrow \text{Digit} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \rightarrow$
$\quad \text{Integer} \rightarrow [\text{Digit}] \rightarrow [\text{Symbol}]$

$\mathsf{fast_decode}\ m\ zs\ z\ c\ a\ b\ n\ ys = \mathsf{approx_decode}\ m\ I\ e$
$\quad \textbf{where}\ I = \mathsf{Fast_int}\ zs\ z\ c\ a\ b$
$\qquad\qquad e = \mathsf{Fast_real}\ zs\ z\ c\ n\ ys$

This doesn't quite work, however, since there's no way to represent the interval $\langle 0, 1\rangle$ in terms of Fast_int, and so no way to express fast_compress in terms of fast_encode. Instead, note that

$$\mathsf{output}\ (\mathsf{encode}\ m\ \langle 0, 1\rangle\ xs)$$

$= \{\text{Definition of } \mathsf{tail}\}$

$$\mathsf{tail}\ (0 : \mathsf{output}\ (\mathsf{encode}\ m\ \langle 0, 1\rangle\ xs))$$

$= \{\text{Proposition 3.3.7}\}$

$$\mathsf{tail}\ (\mathsf{output}\ (\mathsf{interval}\ 0 \otimes \mathsf{encode}\ m\ \langle 0, 1\rangle\ xs))$$

$= \{$ Property of $\mathsf{encode}\}$

$$\mathsf{tail}\ (\mathsf{output}\ (\mathsf{encode}\ m\ (\mathsf{interval}\ 0)\ xs))$$

$\approx \{$ Property of $\mathsf{approx_encode}\}$

$$\mathsf{tail}\ (\mathsf{output}\ (\mathsf{approx_encode}\ m\ (\mathsf{interval}\ 0)\ xs))$$

and so we take

$$\mathsf{fast_compress} :: \text{Model} \to [\text{Symbol}] \to [\text{Digit}]$$
$$\mathsf{fast_compress}\ m\ xs = \mathsf{tail}\ (\mathsf{output}\ (\mathsf{approx_encode}\ m\ (\mathsf{interval}\ 0)\ xs))$$

$$\mathsf{fast_decompress} :: \text{Model} \to [\text{Digit}] \to [\text{Symbol}]$$
$$\mathsf{fast_decompress}\ m\ ys = \mathsf{approx_decode}\ m\ (\mathsf{interval}\ 0)\ (\mathsf{input}\ (0 : ys))$$

4.4.1 Re-Normalisation

As before, we will want to express $\mathsf{Fast_int}\ zs\ z\ c\ a\ b$ in a normal form to prevent $\mathsf{Naive_int}\ a\ b$ from becoming too small, although the normal form looks somewhat different to that in the Witten, Neal and Cleary coder. In this case, re-normalisation happens whenever

$$\mathsf{width}\ (\mathsf{Naive_int}\ a\ b) \leq 1/\mathsf{base},$$

which occurs precisely when $b - a \leq \mathsf{base}^{N-1}$.

When we normalise, we aim to express $\mathsf{Naive_int}\ a\ b$ in the form

$$\mathsf{Naive_int}\ a\ b = \mathsf{interval}\ z' \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

for some z', $a_{z'}$ and $b_{z'}$ such that $0 \leq a_{z'} < b_{z'} \leq 2 \times \mathsf{base}^N$ and $0 < b_{z'} - a_{z'} \leq \mathsf{base}^N$.

An elementary calculation shows that

$$\mathsf{Naive_int}\ a\ b = \mathsf{interval}\ z' \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\Leftrightarrow$ { Proposition 2.2.3 }

$$(\mathsf{interval}\ z')^{-1} \otimes \mathsf{Naive_int}\ a\ b = \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\Leftrightarrow$ { Definition of interval and Naive_int }

$$\langle z'/\mathsf{base},\ (z'+1)/\mathsf{base}\rangle^{-1} \otimes \left\langle a/\mathsf{base}^N,\ b/\mathsf{base}^N\right\rangle \equiv \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\Leftrightarrow$ { Interval arithmetic }

$$\left\langle (a \times \mathsf{base} - z' \times \mathsf{base}^N)/\mathsf{base}^N,\ (b \times \mathsf{base} - z' \times \mathsf{base}^N)/\mathsf{base}^N\right\rangle \equiv \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\Leftrightarrow$ { Definition of Naive_int }

$$\mathsf{Naive_int}\ (a \times \mathsf{base} - z' \times \mathsf{base}^N)\ (b \times \mathsf{base} - z' \times \mathsf{base}^N) = \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\Leftrightarrow$ { Naive_int is a constructor }

$$a_{z'} = a \times \mathsf{base} - z' \times \mathsf{base}^N \quad \text{and} \quad b_{z'} = b \times \mathsf{base} - z' \times \mathsf{base}^N$$

We also require that $0 \leq a_{z'} < 2 \times \mathsf{base}^N$, which is always the case when $0 \leq a_{z'} < \mathsf{base}^N$. Therefore,

$$0 \leq a_{z'} < \mathsf{base}^N$$

$\Leftrightarrow$ { Previous discussion }

$$0 \leq a \times \mathsf{base} - z' \times \mathsf{base}^N < \mathsf{base}^N$$

$\Leftrightarrow$ { Arithmetic }

$$z' \leq a/\mathsf{base}^{N-1} < z' + 1$$

$\Leftrightarrow$ { Definition of floor }

$$z' = \lfloor a/\mathsf{base}^{N-1} \rfloor$$

and so we have derived the values of z', $a_{z'}$ and $b_{z'}$.

We now split into cases. When $z' < \mathsf{base} - 1$, then

$$\mathsf{Fast_int}\ zs\ z\ c\ a\ b$$

$\equiv$ { Definition }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{Naive_int}\ a\ b$$

$=$ { Definition }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{interval}\ z' \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\equiv$ { Definition }

$$\mathsf{Fast_int}\ (zs +\!\!+ z : \mathsf{replicate}\ c\ (\mathsf{base} - 1))\ z'\ 0\ a_{z'}\ b_{z'}$$

When $z' > \mathsf{base} - 1$, we get a very similar result:

$$\mathsf{Fast_int}\ zs\ z\ c\ a\ b$$

$\equiv$ { Definition }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{Naive_int}\ a\ b$$

$=$ { Definition }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{interval}\ z' \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$=$ { Since $\mathsf{interval}\ z \otimes \mathsf{interval}\ z' = \mathsf{interval}\ (z + 1) \otimes \mathsf{interval}\ (z' - \mathsf{base})$ for all z }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ (z + 1))\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ 0)^c \otimes \mathsf{interval}\ (z' - \mathsf{base}) \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\equiv$ { Definition }

$$\mathsf{Fast_int}\ (zs +\!\!+ (z + 1) : \mathsf{replicate}\ c\ 0)\ (z' - \mathsf{base})\ 0\ a_{z'}\ b_{z'}$$

Finally, consider the case $z' = \mathsf{base} - 1$.

$$\mathsf{Fast_int}\ zs\ z\ c\ a\ b$$

$\equiv$ { Definition }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{Naive_int}\ a\ b$$

$=$ { Definition }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{interval}\ z' \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$=$ { Since $z' = \mathsf{base} - 1$ }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^{c+1} \otimes \mathsf{Naive_int}\ a_{z'}\ b_{z'}$$

$\equiv$ { Definition }

$$\mathsf{Fast_int}\ zs\ z\ (c+1)\ a_{z'}\ b_{z'}$$

We also have to consider what happens in the decoder during re-normalisation. This is rather simpler than in the previous arithmetic coder.

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys$$

$\equiv$ { Definition of $\mathsf{Fast_real}$ }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{Naive_real}\ n\ ys$$

$\equiv$ { Interval arithmetic and definition of $\mathsf{Naive_real}$ }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base} - 1))^c \otimes \mathsf{interval}\ z' \otimes (\mathsf{interval}\ z')^{-1} \otimes ((n + \mathsf{input}\ ys)/\mathsf{base}^N)$$

$=$ { Arithmetic }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base}-1))^c \otimes \mathsf{interval}\ z' \otimes$$
$$((n \times \mathsf{base} - z' \times \mathsf{base}^N + \mathsf{head}\ ys + \mathsf{input}\ (\mathsf{tail}\ ys))/\mathsf{base}^N)$$

$\equiv$ { Definition of Naive_real }

$$\mathsf{foldr}\ (\otimes)\ (\mathsf{interval}\ z)\ (\mathsf{map\ interval}\ zs) \otimes (\mathsf{interval}\ (\mathsf{base}-1))^c \otimes \mathsf{interval}\ z' \otimes$$
$$\mathsf{Naive_real}\ (n \times \mathsf{base} - z' \times \mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

$\equiv$ { Definition of Fast_real }

$$\mathsf{Fast_real}\ zs\ z\ (c+1)\ (n \times \mathsf{base} - z' \times \mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

Therefore, when $z' = \mathsf{base} - 1$,

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys = \mathsf{Fast_real}\ zs\ z\ (c+1)$$
$$(n \times \mathsf{base} - z' \times \mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

by the earlier discussion. Similarly, when $z' < \mathsf{base} - 1$,

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys = \mathsf{Fast_real}\ (zs +\!\!+\ z : \mathsf{replicate}\ c\ (\mathsf{base}-1))\ z'\ 0$$
$$(n \times \mathsf{base} - z' \times \mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

and finally,

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys = \mathsf{Fast_real}\ (zs +\!\!+\ (z+1) : \mathsf{replicate}\ c\ 0)\ (z' - \mathsf{base})\ 0$$
$$(n \times \mathsf{base} - z' \times \mathsf{base}^N + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$$

when $z' > \mathsf{base} - 1$.

4.4.2 Approximate Interval Shrinking

In the Witten, Neal and Cleary coder, we require to calculate $a' = a + \lfloor n_1(b-a)/d \rfloor$ and $b' = a + \lfloor n_2(b-a)/d \rfloor$, where $0 \le n_1 < n_2 \le d$ and, to prevent the underflow problem rearing its ugly head, $d \le b - a$. We give an approximation [20, 23, 51, 52] to

$$\left\lfloor \frac{n_i(b-a)}{d} \right\rfloor \quad \text{when} \quad 0 \le n_i \le d \le b-a$$

where $n_i = n_1$ or n_2.

Let $k \in \mathbb{N}$ be such that $1/2(b-a) < 2^k d \leq b-a$, that is $k = \lfloor \log_2(w/d) \rfloor$. Then

$$\left\lfloor \frac{n_i(b-a)}{d} \right\rfloor = \left\lfloor \frac{(2^k n_i)(b-a)}{(2^k d)} \right\rfloor$$

and so we can assume without loss of generality that $1/2(b-a) < d \leq b-a$. Now,

$$1/2(b-a) < d \leq (b-a)$$

$\Leftrightarrow$ { Arithmetic }

$$1 \leq \frac{b-a}{d} < 2$$

$\Rightarrow$ { Since $0 \leq x - \lfloor x \rfloor < 1$ for all x }

$$1 \leq \frac{n_i(b-a)}{d} - \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor + \frac{b-a}{d} < 3$$

$\Leftrightarrow$ { Arithmetic }

$$1 + \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor \leq \frac{(n_i+1)(b-a)}{d} < 3 + \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor$$

$\Leftrightarrow$ { Rule of Floors }

$$1 + \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor \leq \left\lfloor \frac{(n_i+1)(b-a)}{d} \right\rfloor < 3 + \left\lfloor \frac{n(b-a)}{d} \right\rfloor$$

$\Leftrightarrow$ { Arithmetic }

$$1 \leq \left\lfloor \frac{(n_i+1)(b-a)}{d} \right\rfloor - \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor < 3$$

$\Leftrightarrow$ { Arithmetic }

$$\left\lfloor \frac{(n_i+1)(b-a)}{d} \right\rfloor - \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor = 1 \text{ or } 2$$

Observe that

$$\sum_{n_i=0}^{d-1} \left(\left\lfloor \frac{(n_i+1)(b-a)}{d} \right\rfloor - \left\lfloor \frac{n_i(b-a)}{d} \right\rfloor \right) = \left\lfloor \frac{d(b-a)}{d} \right\rfloor - \left\lfloor \frac{0(b-a)}{d} \right\rfloor = b-a.$$

But each of the terms in this sum evaluates to either 1 or 2. Suppose that p of them evaluate to 1 and q of them evaluate to 2. There are d terms in total. Therefore,

$$1p + 2q = b - a \text{ (writing the above sum in a different way) and}$$
$$p + q = d \text{ (since the total number of terms is } d\text{).}$$

This pair of equations has solution

$$p = 2d - (b - a), \text{ and}$$
$$q = (b - a) - d.$$

We therefore make the following approximation:

$$\left\lfloor \frac{(n_i + 1)(b - a)}{d} \right\rfloor - \left\lfloor \frac{n_i(b - a)}{d} \right\rfloor \approx \begin{cases} 1 & \text{if } n_i \geq (b - a) - d \\ 2 & \text{if } n_i < (b - a) - d \end{cases}$$

Note that there are $p = 2d - (b - a)$ values of n_i for which this approximation is 1, and $q = (b - a) - d$ for which it is 2. Furthermore, we can easily see that

$$\left\lfloor \frac{n_i(b - a)}{d} \right\rfloor \approx \begin{cases} n_i + (b - a) - d & \text{if } n_i \geq (b - a) - d \\ 2n_i & \text{if } n_i < (b - a) - d \end{cases}$$

Since this approximation requires at most a small constant number of additions, subtractions and bit-shifts, it is clear that it will be much quicker to evaluate than the original function, which used multiplication and division. In effect, we have approximated integer division by putting a kink into the graph: see Figure 4.4. This approximation also lends itself to hardware implementations [20] of arithmetic coding, since multiplication and division require a significantly larger quantity of silicon than these simple operations.

The cost is that the compression effectiveness is reduced by about 1%. This compares favourably with the cost of 21% that Moffat was prepared to accept in his improved coder [45].

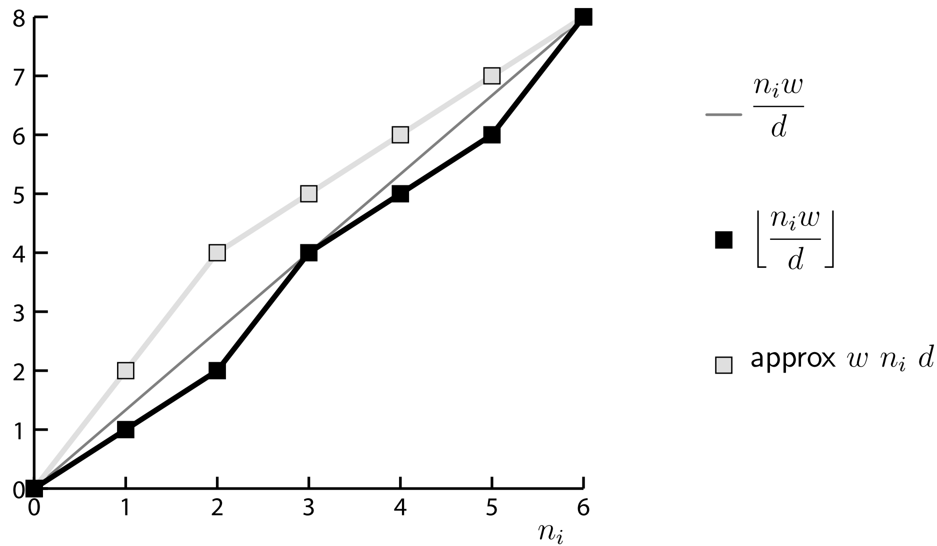


Figure 4.4: Integer Division and Approximate Division with $d = 6$ and $w = 8$

4.4.1 Proposition. Let

$\mathsf{approx} :: \text{Integer} \to \text{Integer} \to \text{Integer} \to \text{Integer}$

$\mathsf{approx}\ w\ n_i\ d$

$\quad |\ 2^k n_i \geq w - 2^k d = 2^k n_i + w - 2^k d$

$\quad |\ 2^k n_i < w - 2^k d = 2(2^k n_i)$

$\qquad$ **where** $1/2 w < 2^k d \leq w$

$\mathsf{unapprox} :: \text{Integer} \to \text{Integer} \to \text{Integer} \to \text{Integer}$

$\mathsf{unapprox}\ w\ n\ d$

$\quad |\ n \geq 2(w - 2^k d) = \lfloor 2^{-k}(n - w + 2^k d) \rfloor$

$\quad |\ n < 2(w - 2^k d) = \lfloor 2^{-k}(1/2 n) \rfloor$

$\qquad$ **where** $1/2 w < 2^k d \leq w$

Then

$$\left\lfloor \frac{n_i(b-a)}{d} \right\rfloor \approx \mathsf{approx}\ (b-a)\ n_i\ d$$

and

$$\mathsf{approx}\ w\ n_i\ d \le n \iff n_i \le \mathsf{unapprox}\ w\ n\ d.$$

Proof. The first part follows directly from the previous discussion.

Suppose that $\mathsf{approx}\ w\ n_i\ d \le n$. There are three cases to consider:

- $2^k n_i \ge w - 2^k d$. Then $2^k n_i + w - 2^k d \le n$, so $n \ge 2(w - 2^k d)$ and $n_i \le 2^{-k}(n - w + 2^k d)$. By the Rule of Floors, therefore $n_i \le \mathsf{unapprox}\ w\ n\ d$.
- $2^k n_i < w - 2^k d$ and $n \ge 2(w - 2^k d)$. Then $2(2^k n_i) \le n$, so $n_i \le 2^{-k}(1/2 n)$. Also, $1/2 n \ge w - 2^k d$, so $n - 1/2 n \le n - w - 2^k d$. Therefore, $n_i \le 2^{-k}(n - w + 2^k d)$. The Rule of Floors gives us that $n_i \le \mathsf{unapprox}\ w\ n\ d$.
- $2^k n_i < w - 2^k d$ and $n < 2(w - 2^k d)$. Then $2(2^k n_i) \le n$, so $n_i \le 2^{-k}(1/2 n)$. Once again, the Rule of Floors gives us that $n_i \le \mathsf{unapprox}\ w\ n\ d$.

The proof that $n_i \le \mathsf{unapprox}\ w\ n\ d \Rightarrow \mathsf{approx}\ w\ n_i\ d \le n$ is very similar. $\square$

Not only is our approximation reversible, but the operations required to reverse it are also extremely cheap. We can therefore make a very fast decoder as well.

4.4.3 The Faster Interval Approximation

Using the work of the previous sections, we can build the interval approximation used by the faster coder. As always, we want it to take the form

$$\mathsf{Fast_int}\ zs\ z\ c\ a\ b \boxtimes_d (n_1, n_2) = \mathsf{Fast_int}\ zs'\ z'\ c'\ a'\ b'$$

for some zs', z', c', a' and b'. We also want it to be able to re-normalise its interval parameter. Putting together all of the results about re-normalisation, along with with the interval-shrinking approximation, gives us the interval approximation of Figures 4.5 and 4.6.

$$
\begin{aligned}
&\mathsf{Fast_int}\ zs\ z\ c\ a\ b \boxtimes_d (n_1,\ n_2)\\
&\quad |\ b - a \le \mathsf{base}^{N-1}\ \wedge\ z' < \mathsf{base} - 1\\
&\qquad = \mathsf{Fast_int}\ zs_0\ z'\ 0\ a_{z'}\ b_{z'} \boxtimes_d (n_1,\ n_2)\\
&\quad |\ b - a \le \mathsf{base}^{N-1}\ \wedge\ z' > \mathsf{base} - 1\\
&\qquad = \mathsf{Fast_int}\ zs_1\ (z' - \mathsf{base})\ 0\ a_{z'}\ b_{z'} \boxtimes_d (n_1,\ n_2)\\
&\quad |\ b - a \le \mathsf{base}^{N-1}\ \wedge\ z' = \mathsf{base} - 1\\
&\qquad = \mathsf{Fast_int}\ zs\ z\ (c+1)\ a_{z'}\ b_{z'} \boxtimes_d (n_1,\ n_2)\\
&\quad |\ \mathbf{otherwise}\\
&\qquad = \mathsf{Fast_int}\ zs\ z\ c\ (a + \mathsf{approx}\ (b-a)\ n_1\ d)\ (a + \mathsf{approx}\ (b-a)\ n_2\ d)\\
&\quad \mathbf{where}\ a_{z'} = a \times \mathsf{base} - z' \times \mathsf{base}^N\\
&\qquad\quad b_{z'} = b \times \mathsf{base} - z' \times \mathsf{base}^N\\
&\qquad\quad z' = \lfloor a / \mathsf{base}^{N-1} \rfloor\\
&\qquad\quad zs_0 = zs \mathbin{+\!\!+} z : \mathsf{replicate}\ c\ (\mathsf{base} - 1)\\
&\qquad\quad zs_1 = zs \mathbin{+\!\!+} (z+1) : \mathsf{replicate}\ c\ 0
\end{aligned}
$$

Figure 4.5: The interval approximation used in the fast arithmetic coder

$(\mathsf{Fast_int}\ zs\ z\ c\ a\ b)^{-1} \boxtimes_d \mathsf{Fast_real}\ zs\ z\ c\ n\ ys$

$\mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' < \mathsf{base} - 1$

$= (\mathsf{Fast_int}\ zs_0\ z'\ 0\ a_{z'}\ b_{z'})^{-1} \boxtimes_d$

$\mathsf{Fast_real}\ zs_0\ z'\ 0\ (n_{z'} + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$

$\mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' > \mathsf{base} - 1$

$= (\mathsf{Fast_int}\ zs_1\ z'\ 0\ a_{z'}\ b_{z'})^{-1} \boxtimes_d$

$\mathsf{Fast_real}\ zs_1\ (z' - \mathsf{base})\ 0\ (n_{z'} + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$

$\mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' = \mathsf{base} - 1$

$= (\mathsf{Fast_int}\ zs\ z\ (c+1)\ a_{z'}\ b_{z'})^{-1} \boxtimes_d$

$\mathsf{Fast_real}\ zs\ z\ (c+1)\ (n_{z'} + \mathsf{head}\ ys)\ (\mathsf{tail}\ ys)$

$\mid$ **otherwise**

$= \mathsf{unapprox}\ (b - a)\ (n - a)\ d$

where $a_{z'} = a \times \mathsf{base} - z' \times \mathsf{base}^N$

$b_{z'} = b \times \mathsf{base} - z' \times \mathsf{base}^N$

$n_{z'} = n \times \mathsf{base} - z' \times \mathsf{base}^N$

$z' = \lfloor a/\mathsf{base}^{N-1} \rfloor$

$zs_0 = zs \mathbin{+\!\!+} z : \mathsf{replicate}\ c\ (\mathsf{base} - 1)$

$zs_1 = zs \mathbin{+\!\!+} (z+1) : \mathsf{replicate}\ c\ 0$

Figure 4.6: The interval approximation used in the fast arithmetic coder

4.4.2 Theorem. Under the definition above, $\boxtimes_d$ is an interval approximation.

Proof. We check the definition of an interval approximation (Definition 3.6.1). Since we have that $0 \leq n_1 < n_2 \leq d$, the definition of approx gives us that

$$a \leq a + \mathsf{approx}\ (b-a)\ n_1\ d < a + \mathsf{approx}\ (b-a)\ n_2\ d \leq b$$

from which we conclude that

$$\mathsf{Naive_int}\ (a + \mathsf{approx}\ (b-a)\ n_1\ d)\ (a + \mathsf{approx}\ (b-a)\ n_2\ d) \subseteq \mathsf{Naive_int}\ a\ b$$

and so

$$\mathsf{Fast_int}\ zs\ z\ c\ (a + \mathsf{approx}\ (b-a)\ n_1\ d)\ (a + \mathsf{approx}\ (b-a)\ n_2\ d) \subseteq \mathsf{Fast_int}\ zs\ z\ c\ a\ b.$$

Therefore, whenever $\mathsf{Fast_int}\ zs\ z\ c\ a\ b$ is in its normal form,

$$\mathsf{Fast_int}\ zs\ z\ c\ a\ b \boxtimes_d (n_1,\ n_2) \subseteq \mathsf{Fast_int}\ zs\ z\ c\ a\ b,$$

and whenever it is not in its normal form, then the results of Section 4.4.1 show that the same conclusion holds. This is the first part of the definition of an interval approximation.

For the second part, we see that (again, assuming that $\mathsf{Fast_int}\ zs\ z\ c\ a\ b$ is in its normal form)

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys \in \mathsf{Fast_int}\ zs\ z\ c\ a\ b \boxtimes_d (n_1,\ n_2)$$

$\Leftrightarrow$ { Definition of $\boxtimes_d$ and fact that $\mathsf{normalise}\ I = I$ }

$$\mathsf{Fast_real}\ zs\ z\ c\ n\ ys \in \mathsf{Fast_int}\ zs\ z\ c\ (a+\mathsf{approx}\ (b-a)\ n_1\ d)\ (a+\mathsf{approx}\ (b-a)\ n_2\ d)$$

$\Leftrightarrow$ { Definitions of $\mathsf{Fast_int}$ and $\mathsf{Fast_real}$ }

$$\mathsf{Naive_real}\ n\ ys \in \mathsf{Naive_int}\ (a + \mathsf{approx}\ (b-a)\ n_1\ d)\ (a + \mathsf{approx}\ (b-a)\ n_2\ d)$$

$\Leftrightarrow$ { Definitions of $\mathsf{Naive_real}$ and $\mathsf{Naive_int}$ }

$$(a + \mathsf{approx}\ (b-a)\ n_1\ d)/\mathsf{base}^N \leq (n + \mathsf{input}\ ys)/\mathsf{base}^N <$$
$$(a + \mathsf{approx}\ (b-a)\ n_2\ d)/\mathsf{base}^N$$

$\Leftrightarrow$ { Arithmetic and the Rule of Floors }

$$\mathsf{approx}\ (b-a)\ n_1\ d \leq n - a < \mathsf{approx}\ (b-a)\ n_2\ d$$

$\Leftrightarrow$ { Proposition 4.4.1 }

$$n_1 \leq \mathsf{unapprox}\ (b-a)\ (n-a)\ d < n_2$$

$\Leftrightarrow$ { Definition of $\boxtimes_d$ }

$$n_1 \leq (\mathsf{Fast_int}\ zs\ z\ c\ a\ b)^{-1} \boxtimes_d \mathsf{Fast_real}\ zs\ z\ c\ n\ ys < n_2$$

and so this really is an interval approximation. For the cases where $\mathsf{Fast_int}\ zs\ z\ c\ a\ b$ is not in its normal form, we can use the results of Section 4.4.1. □

4.4.4 The Complete Coder

We are ready to put all the parts of the faster coder together in much the same way as we did for the Witten, Neal and Cleary coder. The result is the programs given in figures 4.7–4.9.

Following the lead of [70], we have provided a C implementation of this algorithm in Appendix A.

4.5 Analysis of the New Coder's Performance

Table 4.1 shows the performances of the various arithmetic coders discussed so far, measured on a SunBlade 100 machine with a 500MHz Sparc 9 processor running Solaris. In addition, Moffat's coder [45] is tested, as that outperforms Witten's coder when the machine carries

$$
\begin{aligned}
&\mathsf{fast_compress} :: \mathrm{Model} \rightarrow [\mathrm{Symbol}] \rightarrow [\mathrm{Digit}]\\
&\mathsf{fast_compress}\ m\ xs = \mathsf{output}\ (\mathsf{fast_encode}\ m\ zs\ z\ c\ a\ b\ xs)\\
&\qquad \mathbf{where}\ zs = [\,]\\
&\qquad\qquad\quad z = 0\\
&\qquad\qquad\quad c = 0\\
&\qquad\qquad\quad a = 0\\
&\qquad\qquad\quad b = \mathsf{base}^N
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fast_decompress} :: \mathrm{Model} \rightarrow [\mathrm{Digit}] \rightarrow [\mathrm{Symbol}]\\
&\mathsf{fast_decompress}\ m\ ys = \mathsf{fast_decode}\ m\ zs\ z\ c\ a\ b\ n\ (\mathsf{drop}\ N\ ys)\\
&\qquad \mathbf{where}\ zs = [\,]\\
&\qquad\qquad\quad z = 0\\
&\qquad\qquad\quad c = 0\\
&\qquad\qquad\quad a = 0\\
&\qquad\qquad\quad b = \mathsf{base}^N\\
&\qquad\qquad\quad n = \mathsf{base}^N \times \mathsf{input}\ (\mathsf{take}\ N\ ys)
\end{aligned}
$$

Figure 4.7: The definitions of fast_compress and fast_decompress

$$\mathsf{fast_encode}\ m\ zs\ z\ c\ a\ b\ [\,] = \mathsf{Fast_int}\ zs\ z\ c\ a\ b$$

$$\mathsf{fast_encode}\ m\ zs\ z\ c\ a\ b\ (x : xs)$$

$$\mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' < \mathsf{base} - 1$$

$$= \mathsf{fast_encode}\ m\ (zs +\!\!+\ z : \mathsf{replicate}\ c\ (\mathsf{base} - 1))\ z'\ 0\ a_{z'}\ b_{z'}\ (x : xs)$$

$$\mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' > \mathsf{base} - 1$$

$$= \mathsf{fast_encode}\ m\ (zs +\!\!+\ (z+1) : \mathsf{replicate}\ c\ 0)\ (z' - \mathsf{base})\ 0\ a_{z'}\ b_{z'}\ (x : xs)$$

$$\mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' = \mathsf{base} - 1$$

$$= \mathsf{fast_encode}\ m\ zs\ z\ (c+1)\ a_{z'}\ b_{z'}\ (x : xs)$$

$$\mid \mathbf{otherwise}$$

$$= \mathsf{fast_encode}\ (\mathsf{adapt}\ m\ x)\ zs\ z\ c\ a'\ b'\ xs$$

$$\mathbf{where}\ (n_1, n_2) = \mathsf{int_lookup}\ m\ x$$

$$d = \mathsf{denominator}\ m$$

$$a' = a + \mathsf{approx}\ (b - a)\ n_1\ d$$

$$b' = a + \mathsf{approx}\ (b - a)\ n_2\ d$$

$$z' = \lfloor a / \mathsf{base}^{N-1} \rfloor$$

$$a_{z'} = a \times \mathsf{base} - z' \times \mathsf{base}^N$$

$$b_{z'} = b \times \mathsf{base} - z' \times \mathsf{base}^N$$

Figure 4.8: The definition of fast_encode

$$
\begin{aligned}
&\mathsf{fast_decode}\ m\ zs\ z\ c\ a\ b\ n\ ys\\
&\quad \mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' < \mathsf{base} - 1\\
&\qquad = \mathsf{fast_decode}\ m\ (zs +\!\!+\ z : \mathsf{replicate}\ c\ (\mathsf{base} - 1))\ z'\ 0\ a_{z'}\ b_{z'}\ n_{z'}\ (\mathsf{tail}\ ys)\\
&\quad \mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' > \mathsf{base} - 1\\
&\qquad = \mathsf{fast_decode}\ m\ (zs +\!\!+\ (z+1) : \mathsf{replicate}\ c\ 0)\ (z' - \mathsf{base})\ 0\ a_{z'}\ b_{z'}\ n_{z'}\ (\mathsf{tail}\ ys)\\
&\quad \mid b - a \leq \mathsf{base}^{N-1}\ \wedge\ z' = \mathsf{base} - 1\\
&\qquad = \mathsf{fast_decode}\ m\ zs\ z\ (c+1)\ a_{z'}\ b_{z'}\ n_{z'}\ (\mathsf{tail}\ ys)\\
&\quad \mid \mathbf{otherwise}\\
&\qquad = x : xs\\
&\quad\ \ \mathbf{where}\ x = \mathsf{int_symbol}\ m\ (\mathsf{unapprox}\ (b-a)\ (n-a)\ d)\\
&\qquad\qquad xs = \mathsf{fast_decode}\ (\mathsf{adapt}\ m\ x)\ zs\ c\ a'\ b'\ n\ ys\\
&\qquad\qquad (n_1, n_2) = \mathsf{int_lookup}\ m\ x\\
&\qquad\qquad d = \mathsf{denominator}\ m\\
&\qquad\qquad a' = a + \mathsf{approx}\ (b-a)\ n_1\ d\\
&\qquad\qquad b' = a + \mathsf{approx}\ (b-a)\ n_2\ d\\
&\qquad\qquad z' = \lfloor a / \mathsf{base}^{N-1} \rfloor\\
&\qquad\qquad a_{z'} = a \times \mathsf{base} - z' \times \mathsf{base}^{N}\\
&\qquad\qquad b_{z'} = b \times \mathsf{base} - z' \times \mathsf{base}^{N}\\
&\qquad\qquad n_{z'} = n \times \mathsf{base} - z' \times \mathsf{base}^{N} + \mathsf{head}\ ys
\end{aligned}
$$

Figure 4.9: The definition of fast_decode

Encoding	Encoded Size	Encoding Time	Decoding Time
None	3,251,493 bytes		
Witten	1,795,707 bytes	4.181 s	4.378 s
Moffat	2,180,261 bytes	3.583 s	4.712 s
Fast	1,818,799 bytes	1.335 s	1.710 s

Table 4.1: Comparison of the Different Arithmetic Coders

out multiplications in software. The test file is the concatenation of the files in the Calgary Corpus [68]. Each test was repeated 10 times, and the results were averaged to reduce random fluctuations. The fast coder produces an output file that is about 1% larger than the output of Witten's coder, while encoding and decoding take only 32% and 39% of the time respectively. The model used in all cases was identical to that given in Witten's original paper [70]. It is interesting to note that the time required for modelling in the encoder was 0.814 seconds, meaning that the faster encoder runs at about 6.5 times the speed when modelling time is discounted.

Chapter 5

The PPM Model

5.1 Introduction

We have seen how to build an efficient arithmetic coder. In order to produce a complete data compression algorithm, we require a good model for the data that is being encoded. Having seen a part of the input stream, the model is used to estimate the probability distribution for the next (as yet, unseen) letter so that the arithmetic coder can produce an encoding that is of minimal expected length. In cases where we have a great deal of knowledge about the data being encoded (say a stochastic grammar or something similar [24, 56]) then this can be fairly simple.

Unfortunately, we can't usually specify in advance what kind of information is to be encoded. Users might choose to compress something written in any language and with any encoding of the characters. The information might be buried inside markup or transformed in a myriad of other ways. Languages themselves come in many dialects and change with time.

Faced with a moving target, we have to design a model that can learn about the infor-

mation being encoded, and base its future predictions on past observations.

Over the years, several methods for achieving this have been proposed, including Dynamic Markov Coding [15] and the fiendishly clever Burrows-Wheeler Transform [11, 48]. Of all these techniques, we choose to focus upon Prediction by Partial Matching (PPM)—the current state of the art in data compression. It is amenable to the kind of formal work that is the emphasis of this thesis, and there are several ad-hoc choices that must be made. There are several variations on the theme, and it is unclear that any of them are better than any others. The justification for the use of one variant or another is usually experimental. By producing a formal analysis of the algorithm, we hope to do away with all of this ad-hockery, or at least make it clear what assumptions are being made.

5.2 Prediction by Partial Matching

The simplest of all adaptive models is called the *zero-order model*. We simply count the frequencies of the letters, and use that to make our prediction about the forthcoming letters. For example, if we had seen the letter A 35 times and the letter B 65 times in the string we are compressing, then we might predict that the probability that the next letter is an A is about 35%.

Using this method, we can compress English text down to around 4 or 5 bits ber symbol—a saving of 40–50% compared to uncompressed text in a 256-letter alphabet. We can do rather better than this, however, by looking at the previous letter to help predict the next letter. In English, the letter Q is almost always followed by a U, but occasionally by A or I. The probability that a U occurs is normally about 3%, but after a Q it jumps to about 97%. Clearly, the context is useful in predicting what the next letter will be.

The idea behind PPM is to look at the previous few letters in the input stream (the

context), and use them to predict what the next letter is going to be. For each context, we keep a tally of how many times each letter has been seen in that context.

5.2.1 Definition. Let

$$\mathsf{count} :: [\text{Symbol}] \to [\text{Symbol}] \to \text{Integer}$$

$$\mathsf{count}\ [\,]\ \ [\,] = 1$$

$$\mathsf{count}\ ys\ [\,] = 0$$

$$\mathsf{count}\ ys\ xs$$

$$\mid ys \preceq xs = \mathsf{count}\ ys\ (\mathsf{tail}\ xs) + 1$$

$$\mid ys \not\preceq xs = \mathsf{count}\ ys\ (\mathsf{tail}\ xs)$$

That is, $\mathsf{count}\ ys\ xs$ returns the number of times ys occurs as a substring of xs. □

Suppose that XS is the (random) string that we are trying to compress, and that we have observed that $xs \preceq XS$. Then we choose a *context* that is a suffix of xs, and we estimate that

$$\mathbb{P}\ (xs +\!\!+ [x] \preceq XS \mid xs \preceq XS) = \frac{\mathsf{count}\ (context +\!\!+ [x])\ xs}{\mathsf{count}\ context\ xs - 1}.$$

(Observe that $\sum_x \mathsf{count}\ (context +\!\!+ [x])\ xs = \mathsf{count}\ context\ xs - 1$ since *context* is a suffix of xs. This accounts for the -1 term in the denominator of the above expression.)

5.2.2 Example. Suppose that xs = "MISSISSIPPI", and that we take $context$ = "I". Then

- $\mathsf{count}\ context\ xs = 4$
- $\mathsf{count}\ (context +\!\!+ [\text{'S'}])\ xs = 2$
- $\mathsf{count}\ (context +\!\!+ [\text{'P'}])\ xs = 1$.

Therefore, we estimate that

- $\mathbb{P}$ ("MISSISSIPPI" $+\!\!+$ ['S'] $\preceq XS \mid$ "MISSISSIPPI" $\preceq XS) = 2/3$, and
- $\mathbb{P}$ ("MISSISSIPPI" $+\!\!+$ ['P'] $\preceq XS \mid$ "MISSISSIPPI" $\preceq XS) = 1/3$. □

5.2.1 Which Context?

The trouble with this scheme is that it is not clear which context would be the best to use. In the previous example, we could have taken $context = [\,]$, in which case we estimate that $\mathbb{P}$ (“MISSISSIPPI” $+\!\!+$ [‘S’] $\preceq XS$ | “MISSISSIPPI” $\preceq XS$) $= {}^4\!/_{11}$. If we were to choose to take $context =$ “IPPI”, then we wouldn’t have any information at all on which to base our estimate.

There is an even deeper problem than this, however. If we use a short context, then our estimates of the probabilities will be more accurate, but we might not be using all the available information in the input stream. If we use a longer context, then the estimates will be less accurate, but we might be better able to use the information we’ve just seen in the input stream. Where should we draw the line? Unfortunately, there seems to be no answer that stands up to rigorous scrutiny. Instead, we take the intuitively-sensible option of blending together the various estimates.

This still leaves us with one further problem: what if we come across a symbol that has never been seen [69] in the current context? We can’t assign it a zero probability, as that would upset the arithmetic coder. In practice, we add an extra symbol to our alphabet, called the *escape symbol*. Whenever we attempt to encode a letter that has not been seen in the current context, an escape symbol is encoded and the model switches to a shorter context before trying again. Whenever the decoder sees an escape symbol in the encoded data stream, it becomes aware that the next letter has never been seen in the present context, and so switches to the next shortest context before attempting to decode the letter. If the letter still hasn’t been seen, then further escape symbols can be emitted. The trouble is that the escape character then has to be assigned an estimated probability and there are many ways to do this, again with no theoretical justification.

These various setbacks have spawned a large number of different PPM variants, including PPMA, PPMB, PPMC, PPMD, PPMZ, and others [10]. Then there's PPM*, in which the contexts are allowed to grow arbitrarily long, and that too comes in many different forms [13].

5.2.2 Implementing PPM with Suffix Trees and DAWGs

The space requirements of the PPM algorithm are quite large. Fortunately, we don't have to store the full list of contexts and probabilities in memory, as we can use *suffix trees* [64] instead. The context-switching mechanism of PPM bears a marked resemblance to Ukkonen's algorithm [62], and indeed this algorithm can be used as the basis for an implementation of PPM.

A suffix tree of a string xs is an edge-labelled tree. Each node in the tree can be labelled with the concatenation of the edge-labels on the path from the root to the node. For each suffix ys of xs, there is exactly one node whose label has ys as a prefix. Additionally, no node is allowed to have a single child. An example is given in Figure 5.1.

A suffix tree often contains some additional information. For each non-leaf node xs, it is useful to maintain a pointer to the node $\mathsf{tail}\ xs$, known as a *suffix link*, as this information is useful when an escape symbol is encountered. The suffix links are also of vital importance in Ukkonen's algorithm. They are marked with a dashed line in Figure 5.1.

A closely related data structure is the Directed Acyclic Word Graph (DAWG), in which nodes that have identically-labelled descendants are merged together. Since the descendants are identical, there is no need to store them repeatedly, and so the DAWG provides a further saving of space. An example of a DAWG is given in Figure 5.2.

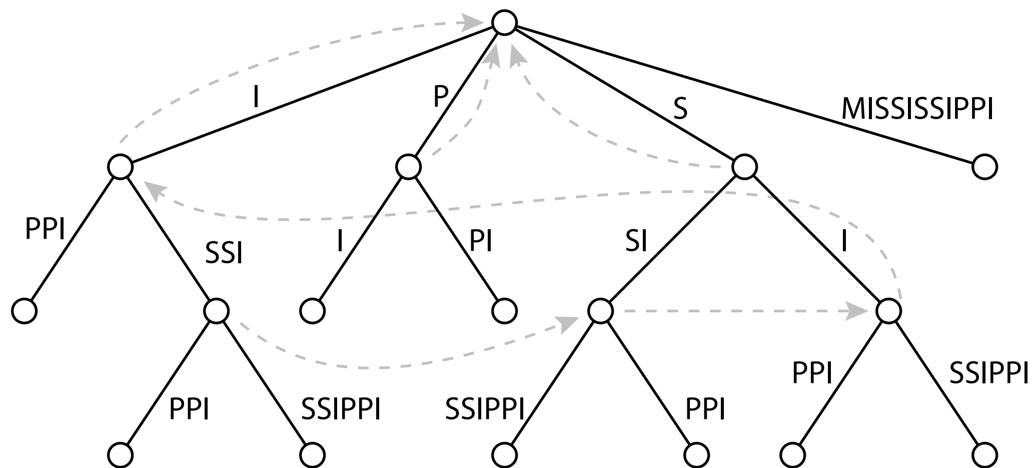


Figure 5.1: The suffix tree of the string “MISSISSIPPI”.

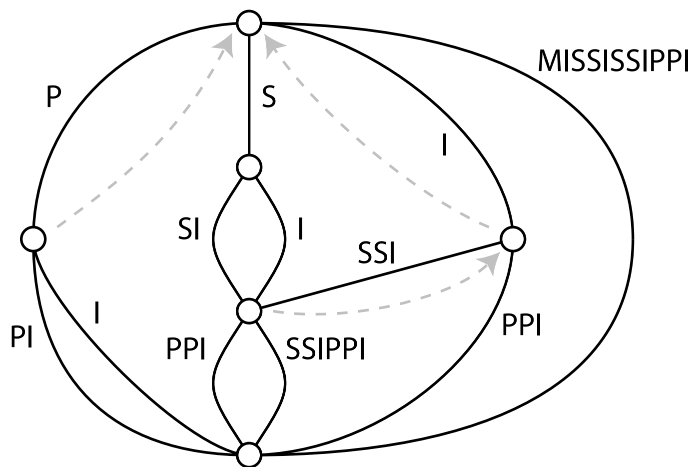


Figure 5.2: The DAWG of the string “MISSISSIPPI”.

5.2.3 Theme and Variations

What probability should we assign to the escape symbol when we are modelling with PPM? This is not an easy question to answer, as we are attempting to estimate the probability of an event that we have never observed in the current context. To give a flavour of the various solutions that have been proposed for this problem, we will give details of three PPM variants.

In PPMA, the escape symbol gets assigned a count of 1.

PPMB is almost identical to PPMA, with the exception that it assigns a probability to each letter only after it has been seen at least twice in the present context. When a letter has not been seen at least twice, then we switch to a shorter context, emit an escape symbol, and try again.

In PPMC, the escape symbol is treated exactly like any other symbol: we keep a tally of how many times it has occurred in each context and use that to predict the probability of it recurring.

5.3 Statistical Estimation

There are many situations in which we would like to know the value of some variable that cannot be measured directly or is too difficult to measure directly. For example, it is not possible to measure the probability that a given coin shows heads when tossed. All we can do is to toss it a few times, and *estimate* the probability based on the observed outcomes.

In data compression, it is impossible to know in advance the probability of a given letter occurring in a given context, and so, for similar reasons, we have to employ estimation techniques.

5.3.1 Maximum Likelihood Estimation

Often, one of the simplest ways of estimating the value of some parameter is by using the maximum-likelihood method. Suppose that we have made some observation O, and we wish to estimate the value of parameter P. Then we choose the estimate, p, of P that maximises the probability of O occurring: that is, we choose p to maximise

$$\mathbb{P}\left(O \mid P = p\right).$$

5.3.1 Example. Suppose we have a file containing some bits, and we observe that there are n zeros and m ones in the file. We wish to estimate the probability that a bit is a zero, and this is our parameter P.

The probability of getting n zeros and m ones is given by the Binomial Distribution:

$$\mathbb{P}\left(n \text{ zeros and } m \text{ ones} \mid P = p\right) = \frac{(n+m)!}{n!\,m!} \times p^n \times (1-p)^m.$$

We can show that this expression is maximised when $p = n/(n+m)$, and so this is our estimate of the probability P that a zero occurs. □

5.3.2 Bayesian Estimation

Bayesian statistics [4] takes a rather different approach. We assume that our parameter P has some known distribution of values, called the *prior distribution*. We then make the observation O, and use Bayes's Theorem to update our prior distribution in the light of our newly-gained knowledge from the observation O. This gives us a new distribution on P, called the *posterior distribution*.

5.3.2 Example. We return to the file of the previous example, and use Bayesian methods to estimate the value of P. As our prior distribution, we might assume that P is uniform on $\langle 0,\, 1\rangle$. Then the prior distribution's density function is:

$$f_P\,(p) = \begin{cases} 1 & \text{if } 0 \leq p < 1 \\ 0 & \text{otherwise} \end{cases}$$

By Bayes's Theorem, the posterior distribution given the observation O is:

$$\begin{aligned} f_{P|O}\,(p) &= \frac{\mathbb{P}\,(O \mid P = p) \times f_P\,(p)}{\mathbb{P}\,(O)} \\ &= \frac{\mathbb{P}\,(O \mid P = p) \times f_P\,(p)}{\int \mathbb{P}\,(O \mid P = p) \times f_P\,(p)\mathrm{d}p} \end{aligned}$$

Once again, we calculate $\mathbb{P}\,(n \text{ zeros and } m \text{ ones} \mid P = p)$ by using the Binomial Distribution. The result of this calculation (by a standard method) is that

$$f_{P|O}\,(p \mid n \text{ zeros and } m \text{ ones}) = \frac{(n+m+2)!}{(n+1)!\,(m+1)!} \times p^n \times (1-p)^m,$$

a function with a sharp spike around $n/(n+m)$.

Notice that the Bayesian estimate is not a single number, but rather a distribution. If we need a number, then we can take the expectation of P, which in this case is $\mathbb{E}\,(P) = (n+1)/(n+m+2)$. □

The choice of prior is often subjective, depending on such things as one's state of mind. For example, when someone hands us a coin, we might assume that it's reasonably fair, or we might be paranoid and assume that it's highly biased. In the latter case, after tossing the coin and finding that it shows heads, we are inclined to believe that the coin will only ever show heads. Since the prior is subjective, Bayesian methods are somewhat controversial, but they do enable us to state precisely what assumptions are being made.

5.4 A New PPM Model

Our description of the PPM algorithm has highlighted a number of difficulties. The most significant is the difficulty in assigning probabilities to the escape sequences. A number of different methods have been proposed, all of which lack theoretical justification. They are distinguished purely by experimental results.

In order to stand any chance of deriving a PPM algorithm, it will be necessary to state precisely the assumptions we are making of the strings being encoded. The attempted axiomatisation led to a beautiful piece of mathematics that proved difficult to work with. Although it was not possible to derive a PPM algorithm from scratch using this method, some useful results were gained nonetheless.

5.4.1 The Axiomatisation

The intention behind this axiomatisation is to produce a Bayesian version of the PPM algorithm. We begin by exploring the relationships between the probabilities of occurrence of the various substrings of the message being encoded.

Our most important assumption will be that the future is similar to the past. Without this, it would be impossible (or at least very difficult) to estimate probabilities of letters in any meaningful way. When we've seen a sample of the data being encoded, then this stationarity assumption ensures that the data will continue to behave in the manner we've already seen.

We will also be assuming that the contents of a message do not help us to predict where the end of the message is. This seems like a fair assumption, given that we have no prior knowledge about the message that is being encoded.

These axioms can be stated formally as follows.

5.4.1 Axiom. We assume that the random message XS to be encoded has the following properties for all strings xs and for all natural numbers n and m:

- $\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \mathbb{P}\ (xs \preceq_m XS \mid [\,] \preceq_m XS)$, and
- $\mathbb{P}\ (xs =_n XS \mid [\,] \preceq_n XS) = (1 - z) \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS)$

where $1 - z = \mathbb{P}\ ([\,] =_n XS \mid [\,] \preceq_n XS)$. □

5.4.2 Proposition. The axioms are equivalent to

$$\begin{aligned} z \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) &= \sum_x \mathbb{P}\ ([x] +\!\!+ xs \preceq_n XS \mid [\,] \preceq_n XS) \\ &= \sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS) \end{aligned}$$

for all strings xs.

Proof. Simply chase definitions.

$$z \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \sum_x \mathbb{P}\ ([x] +\!\!+ xs \preceq_n XS \mid [\,] \preceq_n XS)$$

$\Leftrightarrow$ { Arithmetic }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \frac{\sum_x \mathbb{P}\ ([x] +\!\!+ xs \preceq_n XS \mid [\,] \preceq_n XS)}{z}$$

$\Leftrightarrow$ { Since $1 - z = \mathbb{P}\ ([\,] =_n XS \mid [\,] \preceq_n XS)$ }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \frac{\sum_x \mathbb{P}\ ([x] +\!\!+ xs \preceq_n XS \mid [\,] \preceq_n XS)}{\mathbb{P}\ ([\,] \neq_n XS \mid [\,] \preceq_n XS)}$$

$\Leftrightarrow$ { Since $[\,] \preceq_n XS \Rightarrow ([\,] \preceq_{n+1} XS \Leftrightarrow [\,] \neq_n XS)$ }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \frac{\sum_x \mathbb{P}\ ([x] +\!\!+ xs \preceq_n XS \mid [\,] \preceq_n XS)}{\mathbb{P}\ ([\,] \preceq_{n+1} XS \mid [\,] \preceq_n XS)}$$

$\Leftrightarrow$ { Partitioning the event $xs \preceq_{n+1} XS$ }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \frac{\mathbb{P}\ (xs \preceq_{n+1} XS \mid [\,] \preceq_n XS)}{\mathbb{P}\ ([\,] \preceq_{n+1} XS \mid [\,] \preceq_n XS)}$$

$\Leftrightarrow$ { Conditional probability }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \mathbb{P}\ (xs \preceq_{n+1} XS \mid [\,] \preceq_{n+1} XS)$$

$\Leftrightarrow$ { Property of the natural numbers }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \mathbb{P}\ (xs \preceq_m XS \mid [\,] \preceq_m XS)$$

This is the first axiom.

$$z \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS)$$

$\Leftrightarrow$ { Arithmetic }

$$\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS) = \sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS) + (1 - z) \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS)$$

$\Leftrightarrow$ { Partitioning the event $xs \preceq_n XS$ }

$$\sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS) + \mathbb{P}\ (xs =_n XS \mid [\,] \preceq_n XS) = \sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS) + (1 - z) \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS)$$

$\Leftrightarrow$ { Arithmetic }

$$\mathbb{P}\ (xs =_n XS \mid [\,] \preceq_n XS) = (1 - z) \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS)$$

This gives us the second axiom. □

When performing calculations, we'll usually use the form of the axioms given by Proposition 5.4.2 rather than as they were initially stated, as the calculations are slightly less unpleasant that way. Proposition 5.4.2 still gives a fairly complicated relationship between the probability estimates that we're going to want to make, however. In order to simplify things, we will explore a little deeper.

5.4.3 Theorem. Let $1 \leq l \in \mathbb{N}$, and let $z \in \mathbb{R}$. Let $\mathsf{f} :: \text{String} \rightarrow \text{Real}$ be any function satisfying

$$\sum_x \mathsf{f}\ ([x] +\!\!+\ xs) = \sum_x \mathsf{f}\ (xs +\!\!+\ [x])$$

whenever $\mathsf{length}\ xs = l - 1$. Then there is a unique function $\mathsf{f}' :: \text{String} \rightarrow \text{Real}$ such that

- $\mathsf{length}\ xs = l \Longrightarrow \mathsf{f}'\ xs = \mathsf{f}\ xs$
- $\mathsf{length}\ xs < l \Longrightarrow z \times \mathsf{f}'\ xs = \sum_x \mathsf{f}'\ ([x] +\!\!+\ xs) = \sum_x \mathsf{f}'\ (xs +\!\!+\ [x])$.

Proof. We define f' by

$$\begin{aligned}
&\mathsf{f}'\ xs \\
&\quad \mid \mathsf{length}\ xs = l \quad = \mathsf{f}\ xs \\
&\quad \mid \mathsf{length}\ xs < l \quad = (\textstyle\sum_x \mathsf{f}'\ ([x] +\!\!+\ xs))/z
\end{aligned}$$

Clearly, this is the only f' for which the conclusion can hold. We have to show that $z \times \mathsf{f}'\ xs = \sum_x \mathsf{f}'\ (xs +\!\!+\ [x])$. To do this, we use induction on $l - \mathsf{length}\ xs$.

Base Case. For the base case, we take $l - \mathsf{length}\ xs = 1$.

$$\begin{aligned}
& \sum_x \mathsf{f}'\ (xs +\!\!+\ [x]) \\
= \ & \{\ \text{Since } \mathsf{length}\ (xs +\!\!+\ [x]) = l\ \} \\
& \sum_x \mathsf{f}\ (xs +\!\!+\ [x]) \\
= \ & \{\ \text{Property of } \mathsf{f}\ \} \\
& \sum_x \mathsf{f}\ ([x] +\!\!+\ xs) \\
= \ & \{\ \text{Since } \mathsf{length}\ ([x] +\!\!+\ xs) = l\ \} \\
& \sum_x \mathsf{f}'\ ([x] +\!\!+\ xs) \\
= \ & \{\ \text{Definition of } \mathsf{f}'\ \}
\end{aligned}$$

$$z \times \mathsf{f}'\ xs$$

Inductive Step. We proceed thus:

$$\sum_x \mathsf{f}'\ (xs +\!\!+ [x])$$

$=$ { Definition of f' }

$$\sum_x \sum_{x'} \mathsf{f}'\ ([x'] +\!\!+ xs +\!\!+ [x])/z$$

$=$ { The summations are finite }

$$\sum_{x'} \sum_x \mathsf{f}'\ ([x'] +\!\!+ xs +\!\!+ [x])/z$$

$=$ { Induction }

$$\sum_{x'} \mathsf{f}'\ ([x'] +\!\!+ xs)$$

$=$ { Definition of f' }

$$z \times \mathsf{f}'\ xs$$ □

Once we have constructed a function f such that $\mathbb{P}\ (xs \preceq_n XS \mid xs \preceq_n [\,]) = \mathsf{f}\ xs$ for all xs of length l, we automatically have that $\mathbb{P}\ (xs \preceq_n XS \mid xs \preceq_n [\,]) = \mathsf{f}'\ xs$ for all xs of length up to l. We don't have to carry out any extra verification to ensure that this is the case.

5.4.2 A Simplification

We can simplify things even more than this. We have that

$$\sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS) = z \times \mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS)$$

$\Rightarrow$ { Arithmetic }

$$\sum_x \frac{\mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS)}{\mathbb{P}\ (xs \preceq_n XS \mid [\,] \preceq_n XS)} = z$$

$\Rightarrow$ { Conditional probability }

$$\sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid xs \preceq_n XS) = z$$

It would be far easier to specify a distribution satisfying our axioms by giving the probabilities $\mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid xs \preceq_n XS)$ rather than $\mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS)$, since the verification that $\sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid xs \preceq_n XS) = z$ is so much easier than verifying that $\sum_x \mathbb{P}\ (xs +\!\!+ [x] \preceq_n XS \mid [\,] \preceq_n XS) = \sum_x \mathbb{P}\ ([x] +\!\!+ xs \preceq_n XS \mid [\,] \preceq_n XS)$. Fortunately, we can do exactly this, although the proof requires a certain amount of linear algebra.

5.4.4 Definition. We write the matrix $\mathbf{P}$ as $(p_{j,i})$ if the number in the jth row and the ith column of $\mathbf{P}$ is $p_{j,i}$. We say that $\mathbf{P}$'s columns sum to z if $\sum_j p_{j,i} = z$ for all columns i. □

5.4.5 Theorem. Let $\mathbf{P} = (p_{j,i})$ be an $m \times m$ matrix whose columns sum to z. For each i, let

$$p_i = \det \begin{pmatrix} p_{0,0} - z & p_{0,1} & \cdots & p_{0,i-1} & p_{0,i+1} & \cdots & p_{0,m} \\ p_{1,0} & p_{1,1} - z & \cdots & p_{1,i-1} & p_{1,i+1} & \cdots & p_{1,m} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ p_{i-1,0} & p_{i-1,1} & \cdots & p_{i-1,i-1} - z & p_{i-1,i+1} & \cdots & p_{i-1,m} \\ p_{i+1,0} & p_{i+1,1} & \cdots & p_{i+1,i-1} & p_{i+1,i+1} - z & \cdots & p_{i+1,m} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots \\ p_{m,0} & p_{m,1} & \cdots & p_{m,i-1} & p_{m,i+1} & \cdots & p_{m,m} - z \end{pmatrix}.$$

Then

$$\mathbf{P} \begin{pmatrix} p_0 \\ \vdots \\ p_m \end{pmatrix} = z \begin{pmatrix} p_0 \\ \vdots \\ p_m \end{pmatrix}.$$

Proof. Let

$$(\mathbf{P} - z\mathbf{I}) \begin{pmatrix} p_0 \\ \vdots \\ p_m \end{pmatrix} = \begin{pmatrix} q_0 \\ \vdots \\ q_m \end{pmatrix}.$$

Then, for each j,

$$q_j = \left(\sum_{i \neq j} p_{j,i} \times p_i \right) + (p_{j,j} - z) p_j.$$

We simplify this expression for each j, aiming to show that $q_j = 0$.

One of the properties of a determinant is that you can add one row to another without altering the value of the determinant. Adding to row j the values of all other rows in p_i therefore gives another formula for p_i when $j \neq i$:

$$p_i = \det \begin{pmatrix} p_{0,0} - z & \cdots & p_{0,i-1} & p_{0,i+1} & \cdots & p_{0,m} \\ \vdots & & \vdots & \vdots & & \vdots \\ \sum_{k \neq i} p_{k,0} - z & \cdots & \sum_{k \neq i} p_{k,i-1} - z & \sum_{k \neq i} p_{k,i+1} - z & \cdots & \sum_{k \neq i} p_{k,m} - z \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{i-1,0} & \cdots & p_{i-1,i-1} - z & p_{i-1,i+1} & \cdots & p_{i-1,m} \\ p_{i+1,0} & \cdots & p_{i+1,i-1} & p_{i+1,i+1} - z & \cdots & p_{i+1,m} \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{m,0} & \cdots & p_{m,i-1} & p_{m,i+1} & \cdots & p_{m,m} - z \end{pmatrix}$$

but $\mathbf{P}$'s columns sum to z, so we can simplify this:

$$p_i = \det \begin{pmatrix} p_{0,0} - z & \cdots & p_{0,i-1} & p_{0,i+1} & \cdots & p_{0,m} \\ \vdots & & \vdots & \vdots & & \vdots \\ -p_{i,0} & \cdots & -p_{i,i-1} & -p_{i,i+1} & \cdots & -p_{i,m} \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{i-1,0} & \cdots & p_{i-1,i-1} - z & p_{i-1,i+1} & \cdots & p_{i-1,m} \\ p_{i+1,0} & \cdots & p_{i+1,i-1} & p_{i+1,i+1} - z & \cdots & p_{i+1,m} \\ \vdots & & \vdots & \vdots & & \vdots \\ p_{m,0} & \cdots & p_{m,i-1} & p_{m,i+1} & \cdots & p_{m,m} - z \end{pmatrix}.$$

By the method of minor determinants, we see that

$$p_{j,i} \times p_i = \det \begin{pmatrix} p_{0,0} - z & \cdots & p_{0,i-1} & 0 & p_{0,i+1} & \cdots & p_{0,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ -p_{i,0} & \cdots & -p_{i,i-1} & 0 & -p_{i,i+1} & \cdots & -p_{i,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ p_{i-1,0} & \cdots & p_{i-1,i-1} - z & 0 & p_{i-1,i+1} & \cdots & p_{i-1,m} \\ 0 & \cdots & 0 & p_{j,i} & 0 & \cdots & 0 \\ p_{i+1,0} & \cdots & p_{i+1,i-1} & 0 & p_{i+1,i+1} - z & \cdots & p_{i+1,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ p_{m,0} & \cdots & p_{m,i-1} & 0 & p_{m,i+1} & \cdots & p_{m,m} - z \end{pmatrix}.$$

It is well-known that swapping two rows of a determinant changes the sign of one of the

rows, so we swap rows i and j:

$$p_{j,i} \times p_i = \det \begin{pmatrix} p_{0,0} - z & \cdots & p_{0,i-1} & 0 & p_{0,i+1} & \cdots & p_{0,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & p_{j,i} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ p_{i-1,0} & \cdots & p_{i-1,i-1} - z & 0 & p_{i-1,i+1} & \cdots & p_{i-1,m} \\ p_{i,0} & \cdots & p_{i,i-1} & 0 & p_{i,i+1} & \cdots & p_{i,m} \\ p_{i+1,0} & \cdots & p_{i+1,i-1} & 0 & p_{i+1,i+1} - z & \cdots & p_{i+1,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ p_{m,0} & \cdots & p_{m,i-1} & 0 & p_{m,i+1} & \cdots & p_{m,m} - z \end{pmatrix}.$$

This gives us the value of $p_{j,i} \times p_i$ for each i and j. The remaining term in the formula for q_j is $(p_{j,j} - z)p_j$, and we can easily show that

$$(p_{j,j} - z)p_j = \det \begin{pmatrix} p_{0,0} - z & \cdots & p_{0,j-1} & 0 & p_{0,j+1} & \cdots & p_{0,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ p_{j-1,0} & \cdots & p_{j-1,j-1} - z & 0 & p_{j-1,j+1} & \cdots & p_{j-1,m} \\ 0 & \cdots & 0 & p_{j,j} - z & 0 & \cdots & 0 \\ p_{j+1,0} & \cdots & p_{j+1,j-1} & 0 & p_{j+1,j+1} - z & \cdots & p_{j+1,m} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ p_{m,0} & \cdots & p_{m,j-1} & 0 & p_{m,j+1} & \cdots & p_{m,m} - z \end{pmatrix}$$

We are now in a position to evaluate q_j. By the method of minor determinants, we see that

$$q_j = \det \begin{pmatrix} p_{0,0} - z & p_{0,1} & \cdots & p_{0,m} \\ p_{1,0} & p_{1,1} - z & \cdots & p_{1,m} \\ \vdots & \vdots & & \vdots \\ p_{m,0} & p_{m,1} & \cdots & p_{m,m} - z \end{pmatrix},$$

or, to put it more succinctly, $q_j = \det(\mathbf{P} - z\mathbf{I})$. Since $\mathbf{P}$ has columns summing to z, z is an eigenvalue, and so $\det(\mathbf{P} - z\mathbf{I}) = 0$. Therefore, $q_j = 0$ for all j, and the result follows. □

5.4.6 Corollary. Let

$$p_{j,i} = \mathbb{P}\left([i, j] \preceq_n XS \mid [i] \preceq_n XS\right),$$

and let p_i be as in Theorem 5.4.5. If

$$\mathbb{P}\left([i] \preceq_n XS \mid [\,] \preceq_n XS\right) = z \times \frac{p_i}{\sum_k p_k}$$

then the axioms are satisfied.

Proof. By our earlier discussion,

$$\sum_j \mathbb{P}\left([i, j] \preceq_n XS \mid [i] \preceq_n XS\right) = z$$

and so $\sum_j p_{j,i} = z$. In other words, $\mathbf{P} = (p_{j,i})$ has columns summing to z, and Theorem 5.4.5 applies.

Therefore,

$$\sum_j \mathbb{P}\left([i, j] \preceq_n XS \mid [\,] \preceq_n XS\right)$$

$= \{$ Conditional probability $\}$

$$\sum_j \mathbb{P}\left([i, j] \preceq_n XS \mid [i] \preceq_n XS\right) \times \mathbb{P}\left([i] \preceq_n XS \mid [\,] \preceq_n XS\right)$$

$= \{$ Assumption $\}$

$$\sum_j p_{j,i} \times \mathbb{P}\left([i] \preceq_n XS \mid [\,] \preceq_n XS\right)$$

$= \{$ Since $\sum_j p_{j,i} = z$ $\}$

$$z \times \mathbb{P}\left([i] \preceq_n XS \mid [\,] \preceq_n XS\right)$$

and

$$\sum_i \mathbb{P}\ ([i,j] \preceq_n XS \mid [\,] \preceq_n XS)$$

$= \{$ Conditional probability $\}$

$$\sum_i \mathbb{P}\ ([i,j] \preceq_n XS \mid [i] \preceq_n XS) \times \mathbb{P}\ ([i] \preceq_n XS \mid [\,] \preceq_n XS)$$

$= \{$ Assumption $\}$

$$\sum_i z \times \frac{p_i \times p_{j,i}}{\sum_k p_k}$$

$= \{$ Theorem 5.4.5 $\}$

$$z \times \left(z \times \frac{p_j}{\sum_k p_k}\right)$$

$= \{$ Definition $\}$

$$z \times \mathbb{P}\ ([j] \preceq_n XS \mid [\,] \preceq_n XS)$$

Finally, we can easily see that

$$\sum_i \mathbb{P}\ ([i] \preceq_n XS \mid [\,] \preceq_n XS) = z \times \mathbb{P}\ ([\,] \preceq_n XS \mid [\,] \preceq_n XS)$$

and we're done. □

Suppose we wish to calculate $\mathbb{P}\ ([i,j] \preceq_n XS \mid [\,] \preceq XS)$ when we know the value of $\mathbb{P}\ ([i,j,k] \preceq_n XS \mid [i,j] \preceq XS)$ for all i, j and k. Then we can use Theorem 5.4.5 for that too, by using an alphabet consisting of *pairs* of symbols. For example, the string "Hello" could be represented as ["He", "el", "ll", "lo"], and

$$\mathbb{P}\ (\text{"Hel"} \preceq_n XS \mid \text{"He"} \preceq_n XS) = \mathbb{P}\ ([\text{"He"}, \text{"el"}] \preceq_n XS' \mid [\text{"He"}] \preceq_n XS')\,.$$

Theorem 5.4.5 can also be used to give an explicit formula for the stationary distribution of a stationary Markov chain (contrary to the claim in several statistics textbooks that this

is not possible). Just take $z = 1$ to ensure that $\mathbf{P}$ is a stochastic matrix. Although the system we've set up looks a little bit like a stationary Markov process, it isn't. In particular, we allow the prediction of the next symbol to depend on events that happened at arbitrarily distant positions in the string.

5.5 Towards Bayesian PPM

The main aim of this work was to produce a Bayesian version of PPM, thereby making explicit the assumptions that are being made of the string being compressed. This would enable us to rigorously derive the algorithm from scratch. The main problem was in finding a suitable family of priors for our probabilities $\mathbb{P}\left(xs +\!\!+ [x] \preceq_n XS \mid xs \preceq_n XS\right)$. In the case where $xs = [\,]$, this has already been done: we use a Dirichlet distribution [39]. However, in more complicated cases it is not clear how to build these priors in a way that is consistent with our axioms.

We have seen how the PPM algorithm works in its currently-used form, and have attempted to produce a formal axiomatisation of the method, in the hope of rigorously deriving a similar algorithm. A certain amount of progress has been made in that direction, although more remains to be done—in particular, we will need to find a well-behaved prior distribution before we can begin to apply Bayesian methods.

Appendix A

The New Coder in C

A.1 Introduction

The arithmetic coding program given here is based on the fast coder described in Section 4.4. It is optimised for readability as well as speed, and is intended to be used as a general-purpose arithmetic coding library—experience has shown that it's very hard to get arithmetic coding algorithms to run correctly, so it is much better to reuse tried and tested code!

It contains a few differences from the formally-derived algorithm, intended to make it more useful in practical applications. In particular, the coder adds a few dummy bytes to the compressed file that aren't present in the original algorithm. These ensure that the file pointer is left immediately after the end of the compressed data stream when encoding and decoding. We can therefore concatenate more data onto the end of a compressed file if we so choose, and not have to reposition the pointer after we've finished reading the compressed data. This convenience is not present in the Witten, Neal and Cleary coder.

Before using this code, the reader should be aware that arithmetic coding was covered by a number of patents. Many of these have now expired, since the method is over 20 years

old; others may still be in force. You need to perform your own checks. None of the novel ideas in this paper are covered by additional patents, so you can use them as you wish.

These files can be downloaded from

`http://web.comlab.ox.ac.uk/oucl/research/pdt/ap/ac/coder.zip`.

A.2 The Arithmetic Coding Library

A.2.1 ac.h

The programmers' interface for this arithmetic coder is fairly self-explanatory. The denominator used by `AC_Encode` and `AC_Decode` is that passed to the *previous* call to those functions. Initially, the denominator is set to 1, so you will almost certainly want to say `AC_Encode (0, 1, new_denom)` before beginning the encoding of the file, and equivalently `value = AC_Decode (0, 1, new_denom)` before decoding.

In order to ensure correct operation, both the compressor and decompressor must make matching calls to `AC_Encode` and `AC_Decode`. If the compression software makes a call to `AC_Encode (low, high, new_denom)` then the decompressor must make a matching call to `AC_Decode (low, high, new_denom)`.

```
/*
 *  ac.h
 *  By Barney Stratford
 *  Version 14, 4th May 2005
 */

#ifndef AC_H
#define AC_H

#define MAX_DENOM 65536
```

```
void AC_InitialiseEncoder (FILE *compressed);
void AC_InitialiseDecoder (FILE *compressed);

void AC_Encode (register long low, register long high, long new_denom);
long AC_Decode (register long low, register long high, long new_denom);

void AC_FinaliseEncoder (void);
void AC_FinaliseDecoder (void);

#endif
```

A.2.2 encode.c

```
/*
 *  encode.c
 *  By Barney Stratford
 *  Version 14, 4th May 2005
 */

#include <assert.h>
#include <stdio.h>
#include "ac.h"

#define BITS 16

static FILE *file = NULL;
static long left, kink, bits;
static long denom;
extern const int ac_shift[];

static void write_byte (long byte);

void AC_InitialiseEncoder (FILE *compressed)
{
#ifdef DEBUG
    assert (compressed != NULL);
    assert (file == NULL);
#endif
```

```
    /* Initialise state */
    file = compressed;
    left = 0;
    kink = 0;
    bits = BITS + 8;
    denom = 1;
}

void AC_Encode (register long low, register long high, long new_denom)
{
    register long width;

#ifdef DEBUG
    assert (1 <= new_denom);
    assert (new_denom <= MAX_DENOM);
    assert (0 <= low);
    assert (low < high);
    assert (high <= denom);
    assert (file != NULL);
#endif

    /* Shrink the interval */
    low <<= bits;
    high <<= bits;
    if (low < kink) low <<= 1;
    else low += kink;
    if (high < kink) high <<= 1;
    else high += kink;
    left += low;
    width = high - low;

    /* If necessary, write out some bytes */
    while (width <= 1L << BITS)
    {
        write_byte (left >> BITS);
        left = (left & ((1L << BITS) - 1)) << 8;
        width <<= 8;
    }
```

```
    /* Adjust the denominator */
    if (new_denom <= 256)
        bits = BITS + ac_shift[new_denom] - ac_shift[width >> BITS];
    else
        bits = BITS - 8 + ac_shift[new_denom >> 8] - ac_shift[width >> BITS];
    if (new_denom << bits > width) bits--;
    kink = width - (new_denom << bits);
    denom = new_denom;
}

void AC_FinaliseEncoder (void)
{
#ifdef DEBUG
    assert (file != NULL);
#endif

    /* Write out any remaining data in the buffer */
    write_byte (left >> BITS);
    write_byte (256);
    write_byte (0);
    write_byte (0);
    file = NULL;
}

static void write_byte (long byte)
{
    static int buffer = 0, carry = 0;

    if (byte < 255)
    {
        putc (buffer, file);
        while (carry)
        {
            putc (255, file);
            carry--;
        }
        buffer = byte;
    }
```

```
    else if (byte > 255)
    {
        putc (buffer + 1, file);
        while (carry)
        {
            putc (0, file);
            carry--;
        }
        buffer = byte - 256;
    }
    else carry++;
}
```

A.2.3 decode.c

```
/*
 *  decode.c
 *  By Barney Stratford
 *  Version 14, 4th May 2005
 */

#include <stdio.h>
#include <assert.h>
#include "ac.h"

#define BITS 16

static FILE *file = NULL;
static long left, kink, bits;
static long denom, value;
extern const int ac_shift[];


static long read_byte (void);
```

```
void AC_InitialiseDecoder (FILE *compressed)
{
#ifdef DEBUG
    assert (compressed != NULL);
    assert (file == NULL);
#endif


    /* Initialise state */
    file = compressed;
    read_byte ();
    left = (read_byte () << 16) + (read_byte () << 8) + read_byte ();
    kink = 0;
    bits = BITS + 8;
    denom = 1;
    value = 0;
}



long AC_Decode (register long low, register long high, long new_denom)
{
    register long width;


#ifdef DEBUG
    assert (1 <= new_denom);
    assert (new_denom <= MAX_DENOM);
    assert (0 <= low);
    assert (low <= value);
    assert (value < high);
    assert (high <= denom);
    assert (file != NULL);
#endif
```

```
    /* Shrink the interval */
    low <<= bits;
    high <<= bits;
    if (low < kink) low <<= 1;
    else low += kink;
    if (high < kink) high <<= 1;
    else high += kink;
    left -= low;
    width = high - low;

    /* If necessary, read in some bytes */
    while (width <= 1L << BITS)
    {
        left = (left << 8) + read_byte ();
        width <<= 8;
    }

    /* Adjust the denominator */
    if (new_denom <= 256)
        bits = BITS + ac_shift[new_denom] - ac_shift[width >> BITS];
    else
        bits = BITS - 8 + ac_shift[new_denom >> 8] - ac_shift[width >> BITS];
    if (new_denom << bits > width) bits--;
    kink = width - (new_denom << bits);
    denom = new_denom;

    /* Find the next symbol */
    if (left < kink << 1) return (value = (left >> 1) >> bits);
    else return (value = (left - kink) >> bits);
}

void AC_FinaliseDecoder (void)
{
#ifdef DEBUG
    assert (file != NULL);
#endif
    file = NULL;
}
```

```
static long read_byte (void)
{
    int byte = getc (file);
    return (byte == EOF ? 0 : byte);
}
```

A.2.4 lookup.c

This lookup table satisfies `256 <= n << ac_shift[n] && n << ac_shift[n] < 512` whenever `0 < n && n <= 256`. We use a lookup table for this task because it's faster than performing the calculation.

```
/*
 *  lookup.c
 *  By Barney Stratford
 *  Version 14, 4th May 2005
 */
const int ac_shift[] =
{
    9, 8, 7, 7, 6, 6, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5,
    4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    0
};
```

A.3 Example Compression Software

For completeness, we will give an example of the use of the arithmetic coding library given in the previous section.

A.3.1 main_encode.c

```
/*
 *  main_encode.c
 *  By Barney Stratford
 *  Version 14, 3rd December 2005
 */

#include <stdio.h>
#include "ac.h"
#include "model.h"

int main (void)
{
    int letter;
    long low, high, new_denom;

    new_denom = Model_Initialise ();
    AC_InitialiseEncoder (stdout);
    AC_Encode (0, 1, new_denom);
    do
    {
        letter = getchar ();
        Model_Encode (&low, &high, &new_denom, letter);
        AC_Encode (low, high, new_denom);
    } while (letter != EOF);
    AC_FinaliseEncoder ();
    Model_Finalise ();

    return 0;
}
```

A.3.2 main_decode.c

```
/*
 *  main_decode.c
 *  By Barney Stratford
 *  Version 14, 3rd December 2005
 */

#include <stdio.h>
#include "ac.h"
#include "model.h"

int main (void)
{
    int letter;
    long low, high, value, new_denom;

    new_denom = Model_Initialise ();
    AC_InitialiseDecoder (stdin);
    value = AC_Decode (0, 1, new_denom);
    do
    {
        letter = Model_Decode (&low, &high, &new_denom, value);
        value = AC_Decode (low, high, new_denom);
        if (letter != EOF) putchar (letter);
    } while (letter != EOF);
    AC_FinaliseDecoder ();
    Model_Finalise ();

    return 0;
}
```

A.3.3 model.h

```
/*
 *  model.h
 *  By Barney Stratford
 *  Version 14, 3rd December 2005
 */

#ifndef __MODEL
#define __MODEL

long Model_Initialise (void);
void Model_Encode (long *low, long *high, long *new_denom, int letter);
int  Model_Decode (long *low, long *high, long *new_denom, long value);
void Model_Finalise (void);

#endif
```

A.3.4 model.c

We give the very simplest example of a model: a static model. The files available for download also include an adaptation of the models given in Witten, Neal and Cleary's paper [70] so that fair comparisons can be made.

```
/*
 *  model.c
 *  By Barney Stratford
 *  Version 14, 3rd December 2005
 *
 *  Cumulative frequency counts sourced from "Arithmetic Coding for
 *  Data Compression", Communications of the ACM, 30(6):520--540,
 *  June 1987, by Ian H. Witten, Radford M. Neal and John G. Cleary.
 */

#include <stdio.h>
#include "ac.h"
#include "model.h"
```

```
#define NUM_SYMBOLS 257

const int frequencies[NUM_SYMBOLS + 1] =
{
       0,    1,    2,    3,    4,    5,    6,    7,
       8,    9,   10,  134,  135,  136,  137,  138,
     139,  140,  141,  142,  143,  144,  145,  146,
     147,  148,  149,  150,  151,  152,  153,  154,
     155, 1391, 1392, 1413, 1422, 1425, 1426, 1451,
    1466, 1468, 1470, 1472, 1473, 1552, 1571, 1631,
    1632, 1647, 1662, 1670, 1675, 1679, 1686, 1691,
    1695, 1699, 1705, 1708, 1710, 1711, 1712, 1713,
    1714, 1715, 1739, 1754, 1776, 1788, 1803, 1813,
    1822, 1838, 1854, 1862, 1868, 1880, 1903, 1916,
    1927, 1941, 1942, 1956, 1984, 2013, 2019, 2022,
    2033, 2034, 2037, 2038, 2039, 2040, 2041, 2042,
    2045, 2046, 2537, 2622, 2795, 3027, 3771, 3898,
    4008, 4301, 4719, 4725, 4764, 5014, 5153, 5582,
    6028, 6139, 6144, 6532, 6907, 7438, 7590, 7647,
    7744, 7756, 7857, 7862, 7864, 7865, 7867, 7870,
    7871, 7872, 7873, 7874, 7875, 7876, 7877, 7878,
    7879, 7880, 7881, 7882, 7883, 7884, 7885, 7886,
    7887, 7888, 7889, 7890, 7891, 7892, 7893, 7894,
    7895, 7896, 7897, 7898, 7899, 7900, 7901, 7902,
    7903, 7904, 7905, 7906, 7907, 7908, 7909, 7910,
    7911, 7912, 7913, 7914, 7915, 7916, 7917, 7918,
    7919, 7920, 7921, 7922, 7923, 7924, 7925, 7926,
    7927, 7928, 7929, 7930, 7931, 7932, 7933, 7934,
    7935, 7936, 7937, 7938, 7939, 7940, 7941, 7942,
    7943, 7944, 7945, 7946, 7947, 7948, 7949, 7950,
    7951, 7952, 7953, 7954, 7955, 7956, 7957, 7958,
    7959, 7960, 7961, 7962, 7963, 7964, 7965, 7966,
    7967, 7968, 7969, 7970, 7971, 7972, 7973, 7974,
    7975, 7976, 7977, 7978, 7979, 7980, 7981, 7982,
    7983, 7984, 7985, 7986, 7987, 7988, 7989, 7990,
    7991, 7992, 7993, 7994, 7995, 7996, 7997, 7998,
    7999, 8000
};
```

```
long Model_Initialise (void)
{
    return frequencies[NUM_SYMBOLS];
}

void Model_Encode (long *low, long *high, long *new_denom, int letter)
{
    if (letter == EOF) letter = NUM_SYMBOLS - 1;
    *low = frequencies[letter];
    *high = frequencies[letter + 1];
    *new_denom = frequencies[NUM_SYMBOLS];
}

int Model_Decode (long *low, long *high, long *new_denom, long value)
{
    int letter;

    for (letter = 0; frequencies[letter + 1] <= value; letter++);
    *low = frequencies[letter];
    *high = frequencies[letter + 1];
    *new_denom = frequencies[NUM_SYMBOLS];
    if (letter == NUM_SYMBOLS - 1) letter = EOF;
    return letter;
}

void Model_Finalise (void)
{}
```

Appendix B

Some Standard Haskell Functions

drop removes the first few elements from a list.

$$\mathsf{drop} :: \text{Integer} \rightarrow [\alpha] \rightarrow [\alpha]$$

$$\mathsf{drop}\ 0\ xs = xs$$

$$\mathsf{drop}\ (n+1)\ (x : xs) = \mathsf{drop}\ n\ xs$$

foldl combines list items together starting from the left.

$$\mathsf{foldl} :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$

$$\mathsf{foldl}\ f\ e\ [\,] = e$$

$$\mathsf{foldl}\ f\ e\ (x : xs) = \mathsf{foldl}\ f\ (f\ e\ x)\ xs$$

foldr combines list items together starting from the right.

$$\mathsf{foldr} :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$$

$$\mathsf{foldr}\ f\ e\ [\,] = e$$

$$\mathsf{foldr}\ f\ e\ (x : xs) = f\ x\ (\mathsf{foldr}\ f\ e\ xs)$$

head returns the first element of a list.

$$\mathsf{head} :: [\alpha] \to \alpha$$

$$\mathsf{head}\ (x : xs) = x$$

map applies a function to all elements in a list.

$$\mathsf{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$

$$\mathsf{map}\ f\ [\,] \qquad = [\,]$$

$$\mathsf{map}\ f\ (x : xs) = f\ x : \mathsf{map}\ f\ xs$$

$\mathsf{replicate}$ produces a list whose elements are all identical.

$$\mathsf{replicate} :: \text{Integer} \to \alpha \to [\alpha]$$

$$\mathsf{replicate}\ 0\ x \qquad = [\,]$$

$$\mathsf{replicate}\ (n+1)\ x = x : \mathsf{replicate}\ n\ x$$

tail removes the first element of a list.

$$\mathsf{tail} :: [\alpha] \to [\alpha]$$

$$\mathsf{tail}\ (x : xs) = xs$$

take leaves behind only the first few elements of a list.

$$\mathsf{take} :: \text{Integer} \to [\alpha] \to [\alpha]$$

$$\mathsf{take}\ 0\ xs \qquad\qquad = [\,]$$

$$\mathsf{take}\ (n+1)\ (x : xs) = x : \mathsf{take}\ n\ xs$$

Appendix C

Proofs of Theorems

C.1 Interval Operations

For completeness, we give the proofs of the group-theoretic results about $\mathcal{I}$ that were deferred from Section 2.2. They don't add much to the discussion, and can safely be skipped if you're not interested.

2.2.3 Proposition. Let $I, J, K \in \mathcal{I}$. Then

- $I \otimes J \in \mathcal{I}$
- $I^{-1} \in \mathcal{I}$
- $I \otimes \langle 0,\ 1\rangle = \langle 0,\ 1\rangle \otimes I = I$
- $I \otimes I^{-1} = I^{-1} \otimes I = \langle 0,\ 1\rangle$
- $I \otimes (J \otimes K) = (I \otimes J) \otimes K$

In short, $\mathcal{I}$ is a group with identity $\langle 0,\ 1\rangle$ and operations $\otimes$ and $^{-1}$.

Proof. Throughout the proof, let $\langle l_I,\ r_I\rangle = I$, $\langle l_J,\ r_J\rangle = J$ and $\langle l_K,\ r_K\rangle = K$.

We begin by checking that $\mathcal{I}$ is closed under the operations:

$I \in \mathcal{I} \wedge J \in \mathcal{I}$

$\Rightarrow$ { Definition of $\mathcal{I}$ }

$l_I < r_I \wedge l_J < r_J$

$\Rightarrow$ { Arithmetic }

$0 < r_I - l_I \wedge l_J < r_J \wedge -l_I < 1 - l_I$

$\Rightarrow$ { Arithmetic }

$l_I + l_J(r_I - l_I) < l_I + r_J(r_I - l_I) \wedge -l_I/(r_I - l_I) < (1 - l_I)/(r_I - l_I)$

$\Rightarrow$ { Definition of $\mathcal{I}$ }

$\langle l_I + l_J(r_I - l_I),\, l_I + r_J(r_I - l_I)\rangle \in \mathcal{I} \wedge \langle -l_I/(r_I - l_I),\, (1 - l_I)/(r_I - l_I)\rangle \in \mathcal{I}$

$\Rightarrow$ { Definition of $\otimes$ and $^{-1}$ }

$\langle l_I,\, r_I\rangle \otimes \langle l_J,\, r_J\rangle \in \mathcal{I} \wedge \langle l_I,\, r_I\rangle^{-1} \in \mathcal{I}$

$\Rightarrow$ { Definition of I and J }

$I \otimes J \in \mathcal{I} \wedge I^{-1} \in \mathcal{I}$

We check that the identity has the required properties:

$I \otimes \langle 0,\, 1\rangle$

$=$ { Definition }

$\langle l_I + 0(r_I - l_I),\, l_I + 1(r_I - l_I)\rangle$

$=$ { Arithmetic }

$\langle l_I,\, r_I\rangle$

$=$ { Definition }

I

$\langle 0,\, 1\rangle \otimes I$

$=$ { Definition }

$\langle 0 + l_I(1-0),\, 0 + r_I(1-0)\rangle$

$=$ { Arithmetic }

$\langle l_I,\, r_I\rangle$

$=$ { Definition }

I

We check that the inverse operation $^{-1}$ works correctly:

$I \otimes I^{-1}$

$=$ { Definition }

$\langle l_I,\, r_I\rangle \otimes \langle -l_I/(r_I - l_I),\, (1-l_I)/(r_I - l_I)\rangle$

$=$ { Definition }

$\langle l_I + -l_I(r_I - l_I)/(r_I - l_I),\, l_I + (1-l_I)(r_I - l_I)/(r_I - l_I)\rangle$

$=$ { Arithmetic }

$\langle 0,\, 1\rangle$

$I^{-1} \otimes I$

$=$ { Definition }

$$\langle -l_I/(r_I - l_I),\, (1 - l_I)/(r_I - l_I)\rangle \otimes \langle l_I,\, r_I\rangle$$

$= \{$ Definition $\}$

$$\langle -l_I/(r_I - l_I) + l_I/(r_I - l_I),\, -l_I/(r_I - l_I) + r_I/(r_I - l_I)\rangle$$

$= \{$ Arithmetic $\}$

$$\langle 0,\, 1\rangle$$

Finally, we check that $\otimes$ is associative.

$$I \otimes (J \otimes K)$$

$= \{$ Definitions $\}$

$$\langle l_I,\, r_I\rangle \otimes (\langle l_J,\, r_J\rangle \otimes \langle l_K,\, r_K\rangle)$$

$= \{$ Definition of $\otimes$ $\}$

$$\langle l_I,\, r_I\rangle \otimes \langle l_J + l_K(r_J - l_J),\, l_J + r_K(r_J - l_J)\rangle$$

$= \{$ Definition of $\otimes$ $\}$

$$\langle l_I + (l_J + l_K(r_J - l_J))(r_I - l_I),\, l_I + (l_J + r_K(r_J - l_J))(r_I - l_I)\rangle$$

$= \{$ Definition of $\otimes$ $\}$

$$\langle l_I + l_J(r_I - l_I),\, l_I + r_J(r_I - l_I)\rangle \otimes \langle l_K,\, r_K\rangle$$

$= \{$ Definition of $\otimes$ $\}$

$$(\langle l_I,\, r_I\rangle \otimes \langle l_J,\, r_J\rangle) \otimes \langle l_K,\, r_K\rangle$$

$= \{$ Definitions $\}$

$$(I \otimes J) \otimes K$$

Therefore, as claimed, $\mathcal{I}$ is a group. □

2.2.5 Proposition. Let $I, J \in \mathcal{I}$ and let $x \in \mathbb{R}$. Then

- $\langle 0,\, 1\rangle \otimes x = x$
- $(I \otimes J) \otimes x = I \otimes (J \otimes x)$

Proof. Let $\langle l_I,\, r_I\rangle = I$ and $\langle l_J,\, r_J\rangle = J$. Then the identity behaves correctly:

$$\langle 0,\, 1\rangle \otimes x$$

$$= \{\text{Definition}\}$$

$$0 + x(1-0)$$

$$= \{\text{Arithmetic}\}$$

$$x$$

and the order in which operations are performed is irrelevant:

$$I \otimes (J \otimes x)$$

$$= \{\text{Definitions}\}$$

$$\langle l_I,\, r_I\rangle \otimes (\langle l_J,\, r_J\rangle \otimes x)$$

$$= \{\text{Definition of } \otimes\}$$

$$\langle l_I,\, r_I\rangle \otimes (l_J + x(r_J - l_J))$$

$$= \{\text{Definition of } \otimes\}$$

$$l_I + l_J(r_I - l_I) + x(r_J - l_J)(r_I - l_I)$$

$$= \{\text{Definition of } \otimes\}$$

$$\langle l_I + l_J(r_I - l_I),\, l_I + r_J(r_I - l_I)\rangle \otimes x$$

$= \{$ Definition of $\otimes$ $\}$

$$(\langle l_I,\, r_I\rangle \otimes \langle l_J,\, r_J\rangle) \otimes x$$

$= \{$ Definitions $\}$

$$(I \otimes J) \otimes x$$

Therefore, the claimed properties really do hold. □

2.2.7 Proposition. Let $I, J \in \mathcal{I}$. Then

- $\mathsf{left}\ (I \otimes J) = I \otimes \mathsf{left}\ J$
- $\mathsf{right}\ (I \otimes J) = I \otimes \mathsf{right}\ J$.
- $\mathsf{width}\ (I \otimes J) = \mathsf{width}\ I \times \mathsf{width}\ J$

Proof. As always, let $\langle l_I,\, r_I\rangle = I$ and $\langle l_J,\, r_J\rangle = J$. Then

$$I \otimes J$$

$= \{$ Definitions $\}$

$$\langle l_I,\, r_I\rangle \otimes \langle l_J,\, r_J\rangle$$

$= \{$ Definition of $\otimes$ $\}$

$$\langle l_I + l_J(r_I - l_I),\, l_I + r_J(r_I - l_I)\rangle$$

$= \{$ Definition of $\otimes$ $\}$

$$\langle \langle l_I,\, r_I\rangle \otimes l_J,\, \langle l_I,\, r_I\rangle \otimes r_J\rangle$$

$= \{$ Definitions $\}$

$$\langle I \otimes \mathsf{left}\ J,\ I \otimes \mathsf{right}\ J\rangle$$

from which the first two results follow.

$$\mathsf{width}\ (I \otimes J)$$

$= \{$ Definitions $\}$

$$\mathsf{width}\ (\langle l_I,\ r_I\rangle \otimes \langle l_J,\ r_J\rangle)$$

$= \{$ Definition of $\otimes$ $\}$

$$\mathsf{width}\ \ \langle l_I + l_J(r_I - l_I),\ l_I + r_J(r_I - l_I)\rangle$$

$= \{$ Definition of width $\}$

$$(l_I + r_J(r_I - l_I)) - (l_I + l_J(r_I - l_I))$$

$= \{$ Arithmetic $\}$

$$(r_I - l_I)(r_J - l_J)$$

$= \{$ Definition of width $\}$

$$\mathsf{width}\ \ \langle l_I,\ r_I\rangle \times \mathsf{width}\ \ \langle l_J,\ r_J\rangle$$

$= \{$ Definitions $\}$

$$\mathsf{width}\ I \times \mathsf{width}\ J$$ □

2.2.8 Proposition. Let I, X and Y be in $\mathcal{I}$, and let $x \in \mathbb{R}$. Then

- $X \subseteq Y \iff I \otimes X \subseteq I \otimes Y$
- $x \in X \iff I \otimes x \in I \otimes X$

Proof. Suppose that $I = \langle l_I, r_I \rangle$, $X = \langle l_X, r_X \rangle$ and $Y = \langle l_Y, r_Y \rangle$. Then

$$X \subseteq Y$$

$\Leftrightarrow$ { Definitions and set theory }

$$l_Y \leq l_X \quad \text{and} \quad r_X \leq r_Y$$

$\Leftrightarrow$ { Arithmetic, since $l_I < r_I$ }

$$l_I + l_Y(r_I - l_I) \leq l_I + l_X(r_I - l_I) \quad \text{and} \quad l_I + r_X(r_I - l_I) \leq l_I + r_Y(r_I - l_I)$$

$\Leftrightarrow$ { Set theory }

$$\langle l_I + l_X(r_I - l_I), l_I + r_X(r_I - l_I) \rangle \subseteq \langle l_I + l_Y(r_I - l_I), l_I + r_Y(r_I - l_I) \rangle$$

$\Leftrightarrow$ { Definition of $\otimes$ }

$$\langle l_I, r_I \rangle \otimes \langle l_X, r_X \rangle \subseteq \langle l_I, r_I \rangle \otimes \langle l_Y, r_Y \rangle$$

$\Leftrightarrow$ { Definitions }

$$I \otimes X \subseteq I \otimes Y$$

and

$$x \in X$$

$\Leftrightarrow$ { Definitions and set theory }

$$l_X \leq x < r_X$$

$\Leftrightarrow$ { Arithmetic, since $l_I < r_I$ }

$$l_I + l_X(r_I - l_I) \leq l_I + x(r_I - l_I) < l_I + r_X(r_I - l_I)$$

$\Leftrightarrow$ { Set theory }

$$l_I + x(r_I - l_I) \in \langle l_I + l_X(r_I - l_I),\, l_I + r_X(r_I - l_I)\rangle$$

$\Leftrightarrow$ { Definition of $\otimes$ }

$$\langle l_I,\, r_I\rangle \otimes x \in \langle l_I,\, r_I\rangle \otimes \langle l_X,\, r_X\rangle$$

$\Leftrightarrow$ { Definitions }

$$I \otimes x \in I \otimes X$$

□

C.2 Arithmetic Coding Proofs

3.3.7 Proposition. Let

$\mathsf{interval} :: \mathrm{Digit} \to \mathrm{Interval}$

$\mathsf{interval}\ d = \langle d/\mathsf{base},\ (d+1)/\mathsf{base}\rangle$

Let d be a digit, ds be a list of digits, and $I \subseteq \langle 0,\, 1\rangle$. Then

$$\mathsf{input}\ (d : ds) = \mathsf{interval}\ d \otimes \mathsf{input}\ ds$$

and

$$\mathsf{output}\ (\mathsf{interval}\ d \otimes I) = d : \mathsf{output}\ I$$

Proof. The first expression follows immediately from the definition of $\otimes$.

We now prove the second. As in the definition of output, let

$$\begin{aligned}
n &= \lceil -\log_{\mathsf{base}}(\mathsf{width}\ I)\rceil \\
n' &= \lceil -\log_{\mathsf{base}}(\mathsf{width}\ (\mathsf{interval}\ d \otimes I))\rceil \\
x &= (\lceil \mathsf{right}\ I \times \mathsf{base}^n\rceil - 1)/\mathsf{base}^n \\
x' &= (\lceil \mathsf{right}\ (\mathsf{interval}\ d \otimes I) \times \mathsf{base}^{n'}\rceil - 1)/\mathsf{base}^{n'}
\end{aligned}$$

Then

$$n'$$

$$= \{\text{Definition of } n'\}$$

$$\lceil -\log_{\mathsf{base}}(\mathsf{width}\ (\mathsf{interval}\ d \otimes I))\rceil$$

$$= \{\text{Proposition 2.2.7}\}$$

$$\lceil -\log_{\mathsf{base}}(\mathsf{width}\ I/\mathsf{base})\rceil$$

$$= \{\text{Arithmetic}\}$$

$$\lceil -\log_{\mathsf{base}}(\mathsf{width}\ I) + 1\rceil$$

$$= \{\text{Definition of } n\}$$

$$n + 1$$

and

$$x'$$

$$= \{\text{Definition of } x' \text{ and preceding result}\}$$

$$(\lceil \mathsf{right}\ (\mathsf{interval}\ d \otimes I) \times \mathsf{base}^{(n+1)}\rceil - 1)/\mathsf{base}^{(n+1)}$$

$$= \{\text{Proposition 2.2.7}\}$$

$$(\lceil \mathsf{interval}\ d \otimes \mathsf{right}\ I \times \mathsf{base}^{(n+1)}\rceil - 1)/\mathsf{base}^{(n+1)}$$

$$= \{\text{Definition of } \otimes\}$$

$$(\lceil (d + \mathsf{right}\ I) \times \mathsf{base}^{n}\rceil - 1)/\mathsf{base}^{(n+1)}$$

$$= \{d \times \mathsf{base}^n \text{ is an integer}\}$$

$$(d \times \mathsf{base}^n + \lceil(\mathsf{right}\ I \times \mathsf{base}^n\rceil - 1)/\mathsf{base}^{(n+1)}$$

$= \{$ Definition of x $\}$

$$(d + x)/\mathsf{base}$$

Clearly,

$$\mathsf{digits}\ ((d + x)/\mathsf{base}) = d : \mathsf{digits}\ x$$

whenever d is a digit and $x \in \langle 0,\ 1\rangle$. Therefore,

$$\mathsf{output}\ (\mathsf{interval}\ d \otimes I)$$

$= \{$ Definition of output $\}$

$$\mathsf{take}\ n'\ (\mathsf{digits}\ x')$$

$= \{$ Preceding results $\}$

$$\mathsf{take}\ (n + 1)\ (d : \mathsf{digits}\ x)$$

$= \{$ Definition of take $\}$

$$d : \mathsf{take}\ n\ (\mathsf{digits}\ x)$$

$= \{$ Definition of output $\}$

$$d : \mathsf{output}\ I$$

□

3.4.1 Theorem. Suppose that, for all messages xs and symbols x,

$$\mathsf{width}\ (\mathsf{lookup}\ (\mathsf{foldl}\ \mathsf{adjust}\ m\ xs)\ x) = \mathbb{P}\ (xs +\!\!+ [x] \preceq XS \mid xs \preceq XS)$$

and

$$\mathsf{width}\ (\mathsf{lookup}\ (\mathsf{foldl}\ \mathsf{adjust}\ m\ xs)\ \mathrm{EOF}) = \mathbb{P}\ (xs = XS \mid xs \preceq XS)\,.$$

Then, for all messages xs and ys,

$$\mathsf{width}\ (\mathsf{encode}\ (\mathsf{foldl}\ \mathsf{adjust}\ m\ ys)\ xs) = \mathbb{P}\ (ys +\!\!+\ xs = XS \mid ys \preceq XS)\,.$$

Proof. Use induction on xs. For brevity, we write $m' = \mathsf{foldl}\ \mathsf{adjust}\ m\ ys$.

Case $[\,]$**.** In this case, $\mathsf{encode}\ m'\ [\,] = \mathsf{lookup}\ m'$ EOF, from which the result follows

Case $x : xs$**.** Once more, we chase definitions.

$$\mathsf{width}\ (\mathsf{encode}\ m'\ (x : xs))$$

$= \{$ Definition of encode $\}$

$$\mathsf{width}\ ((\mathsf{lookup}\ m'\ x) \otimes \mathsf{encode}\ (\mathsf{adjust}\ m'\ x)\ xs)$$

$= \{$ Proposition 2.2.7 $\}$

$$\mathsf{width}\ (\mathsf{lookup}\ m'\ x) \times \mathsf{width}\ (\mathsf{encode}\ (\mathsf{adjust}\ m'\ x)\ xs)$$

$= \{$ Definition of foldl $\}$

$$\mathsf{width}\ (\mathsf{lookup}\ m'\ x) \times \mathsf{width}\ (\mathsf{encode}\ (\mathsf{foldl}\ \mathsf{adjust}\ m\ (ys +\!\!+ [x]))\ xs)$$

$= \{$ Induction $\}$

$$\mathsf{width}\ (\mathsf{lookup}\ m'\ x) \times \mathbb{P}\ (ys +\!\!+ [x] +\!\!+ xs = XS \mid ys +\!\!+ [x] \preceq XS)$$

$= \{$ Assumption $\}$

$$\mathbb{P}\ (xs +\!\!+ [x] \preceq XS \mid xs \preceq XS) \times \mathbb{P}\ (ys +\!\!+ [x] +\!\!+ xs = XS \mid ys +\!\!+ [x] \preceq XS)$$

$= \{$ Probability theory $\}$

$$\mathbb{P}\ (ys +\!\!+ (x : xs) = XS \mid ys \preceq XS)$$

This proves the claim. $\square$

Bibliography

[1] Simon Ager. Shorthand. `http://www.omniglot.com/writing/shorthand.htm`.

[2] Arne Andersson and Steffan Nilson. Efficient implementation of suffix trees. *Software—Practice and Experience*, 25(2):129–141, February 1995.

[3] Timothy C. Bell. The Canterbury Corpus. `http://corpus.canterbury.ac.nz/`.

[4] José M. Bernardo and Adrian F. M. Smith. *Bayesian Theory*. Wiley, 1994.

[5] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, second edition, 1998.

[6] Richard Bird and Jeremy Gibbons. Arithmetic coding with folds and unfolds. In Johan Jeuring and Simon Peyton Jones, editors, Advanced Functional Programming 4, volume 2638 of LNCS. Springer-Verlag, 2003.

[7] Jeremy S. De Bonet. Data compression techniques for branch prediction. Unpublished article available from `http://www.debonet.com/research/publications/1999/DeBonet-BranchPrediction.pdf`, June 1999.

[8] Léon Bottou, Paul G. Howard, and Yoshua Bengio. The Z-coder adaptive binary coder. In *Proceedings of the Data Compression Conference*, pages 13–22, 1998.

[9] Suzanne Bunton. *On-Line Stochastic Processes in Data Compression.* PhD thesis, University of Washington, 1996.

[10] Suzanne Bunton. Semantically motivated improvements for PPM variants. *The Computer Journal*, 40(2/3):76–93, 1997.

[11] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Centre, 1994.

[12] Lewis Carroll. *Through the Looking-glass and What Alice Found There*. 1872.

[13] John G. Cleary and W.J. Teahan. Unbounded length contexts for PPM. *The Computer Journal*, 40(2/3):67–75, 1997.

[14] Michael L. Cole. Recursive data compression. US Patent Number 5,488,364.

[15] G. V. Cormack and R. N. S. Horspool. Data compression using dynamic Markov modelling. *The Computer Journal*, 30(6):541–550, 1987.

[16] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley, September 1991.

[17] Maxime Crochemore and Renaud Vérin. Direct construction of compact directed acyclic word graphs. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, pages 116–129, 1997.

[18] Sebastian Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software—Practice and Experience*, 32(2):99–111, 2002.

[19] P. Fenwick. A new data structure for cumulative probability tables. *Software—Practice and Experience*, 24(3):327–336, March 1994.

[20] Gennady Feygin, P. Glenn Gulak, and Paul Chow. Minimizing error and VLSI complexity in the multiplication free approximation of arithmetic coding. In *Proceedings of the Data Compression Conference*, March 1993.

[21] Geoffrey Grimmett and David Stirzaker. *Probability and Random Processes.* Oxford University Press, third edition, 1982.

[22] The Inference Group. Dasher. `http://www.inference.phy.cam.ac.uk/dasher/`.

[23] Hannes Hassler and Naofumi Takagi. Function evaluation by table look-up and addition. Technical Report KUIS-95-0003, Department of Information Science, Kyoto University, January 1995.

[24] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, first edition, October 1979. The second edition of this book has significantly reduced scope.

[25] Jason L. Hutchens and Michael D. Alder. Language acquisition and data compression, January 1997.

[26] Shunsuke Inenaga, Hiromasa Hoshino, Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa, Giancarlo Mauri, and Giulio Pavesi. On-line construction of compact directed acyclic word graphs. *Lecture Notes in Computer Science*, 2089:169–180, 2001.

[27] David C. James. Method for data compression. US Patent Number 5,533,051.

[28] J. Jiang. Novel design of arithmetic coding for data compression. *IEE Proceedings on Computers and Digital Techniques*, 142(6):419–424, November 1995.

[29] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *6th Symposium on Combinatorial Pattern Matching*, 1995.

[30] Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In *Proceedings of the Second Annual International Computing and Combinatorics Conference*, pages 219–230, 1996.

[31] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[32] Anita Kreitzman. The history of shorthand. Available from `http://www.ncraonline.org/about/history/shorthand.shtml`.

[33] Stefan Kurtz and Bernhard Balkenhol. Space efficient linear time computation of the Burrows and Wheeler-transformation.

[34] Steffan Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.

[35] Glen G. Langdon, Jr. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2):135–149, March 1984.

[36] N. Jesper Larsson. Extended application of suffix trees to data compression. In *Proceedings of the Data Compression Conference*, pages 190–199, 1996.

[37] N. Jesper Larsson. The context trees of block sorting compression. In *Proceedings of the Data Compression Conference*, pages 198–198, 1998.

[38] Haris Lekatsas and Wayne Wolf. Random access decompression using binary arithmetic coding. In *Proceedings of the Data Compression Conference*, pages 306–315, 1999.

[39] David J. C. MacKay and Linda C. Bauman Peto. A hierarchical Dirichlet language model. *Natural Language Engineering*, 1(3):1–19, 1994.

[40] Matthew Mahoney. Text compression as a test for artificial intelligence. In *AAAI/IAAI*, 1999.

[41] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[42] Alistair Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.

[43] Alistair Moffat. Critique of “Novel design of arithmetic coding for data compression”. *IEE Proceedings on Computers and Digital Techniques*, 144(6):394–396, November 1997.

[44] Alistair Moffat. An improved data structure for cumulative probablity tables. *Software—Practice and Experience*, 29(7):647–659, June 1999.

[45] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998.

[46] Mark Nelson. LZW data compression. *Dr. Dobb's Journal*, October 1989.

[47] Mark Nelson. Arithmetic coding + statistical modelling = data compression. *Dr Dobb's Journal*, February 1991.

[48] Mark Nelson. Data compression with the Burrows-Wheeler Transform. *Dr. Dobb's Journal*, September 1996.

[49] Peter M. Neumann, Gabrielle A. Stoy, and Edward C. Thompson. *Groups and Geometry*. Oxford University Press, 1994.

[50] Craig G. Nevill-Manning. *Inferring Sequential Structure*. PhD thesis, University of Waikato, May 1996.

[51] Stuart F. Oberman and Michael J. Flynn. Fast IEEE rounding for division by functional iteration. Technical Report CSL-TR-96-700, Computer Systems Laboratory, Stanford University, July 1996.

[52] Stuart F. Oberman and Michael J. Flynn. Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, August 1997.

[53] W. B. Pennebaker, J. L. Mitchell, Jr G. G. Langdon, and R. B. Arps. An overview of the basic principles of the Q-coder adaptive binary arithmetic coder. *IBM Journal of Research and Development*, 32(6):717–726, November 1988.

[54] John Peterson and Olaf Chitil. The Haskell Home Page. `http://www.haskell.org/`.

[55] John Peterson, Olaf Chitil, and Simon Peyton Jones. About Haskell. Available from `http://www.haskell.org/aboutHaskell.html`.

[56] György E. Révész. *Introduction to Formal Languages*. Dover, June 1991.

[57] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, second edition, 1998.

[58] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423,623–656, July and October 1948.

[59] Andreas Stolcke. *Bayesian Learning of Probabilistic Language Models*. PhD thesis, University of California at Berkeley, 1994.

[60] W.J. Teahan. Probability estimation for PPM. In *Proceedings of the New Zealand Computer Science Research Students' Conference*. University of Waikato, 1995.

[61] James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, December 1970.

[62] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, September 1995.

[63] Peter Walley. Inferences from multinomial data: Learning about a bag of marbles. *Journal of the Royal Statistical Society B*, 58(1):3–57, 1996.

[64] P. Weiner. Pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[65] Terry A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[66] Dominic Welsh. *Codes and Cryptography.* Oxford University Press, 1988.

[67] Morse code. From Wikipedia. `http://en.wikipedia.org/wiki/Morse_code`.

[68] Ian H. Witten and Timothy C. Bell. The Calgary Corpus. Available from `ftp://ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus`.

[69] Ian H. Witten and Timothy C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, July 1991.

[70] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987.

[71] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

[72] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.