JOHANNES KEPLER
UNIVERSITY LINZ

Author
Thomas Michlmayr
12202348

Submission
RISC
Research Institute for Symbolic Computation

Thesis Supervisor
A.Univ.-Prof. DI Dr. Wolfgang Schreiner

September 2026

FORMAL MODELLING OF CHANNEL- AND SOURCE-CODING ALGORITHMS FROM CODING THEORY

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Technische Mathematik

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Kurzfassung

Diese Bachelorarbeit verfolgt das Ziel, ausgewählte Konzepte und Verfahren der Informationstheorie in der Sprache des mathematischen Model Checkers RISCAL (RISC Algorithm Language) mathematisch zu spezifizieren und zu implementieren. Hierfür werden Verfahren aus dem Bereich des Channel-Codings zur Fehlererkennung und -korrektur übermittelter digitaler Nachrichten sowie Verfahren des Source-Codings zur Datenkomprimierung betrachtet. Die entwickelten Algorithmen und ihre Eigenschaften werden in RISCAL formal spezifiziert und auf endlichen Datentypen verifiziert. RISCAL ermöglicht dabei die vollständige Überprüfung aller möglichen Eingangsparameter und liefert im Falle einer Verletzung der spezifizierten Eigenschaften entsprechende Gegenbeispiele. Anhand verschiedener Verfahren aus dem Channel- und Source-Coding wird gezeigt, wie Konzepte der Codierungstheorie formal in RISCAL modelliert und verifiziert werden können.

Abstract

This bachelor's thesis aims to mathematically specify and implement selected concepts and techniques from information theory in the language of the mathematical model checker RISCAL (RISC Algorithm Language). To this end, techniques from channel coding for the detection and correction of errors in transmitted digital messages, as well as techniques from source coding for data compression, are considered. The developed algorithms and their properties are formally specified and verified in RISCAL using finite data types. RISCAL enables the exhaustive verification of all possible input parameters and provides counterexamples if a specified property is violated. Using various channel- and source-coding techniques, this thesis demonstrates how concepts from coding theory can be formally modelled and verified in RISCAL.

Contents

1 Introduction and Background **1**

2 State of the Art **3**
2.1 Coding Theory Fundamentals 3
2.2 Channel Coding 6
2.3 Source Coding 6
2.4 Formal Verification 7
2.5 Model Checking 7
2.6 RISCAL 8

3 Channel Coding **10**
3.1 Parity Bit, Even Weight- and Repetition Code 10
3.2 Identifying Transmission Errors 15
3.3 Hamming Code 16

4 Source Coding **23**
4.1 Run Length Encoding 23
4.2 Variable-Size Codes, Prefix Codes 34
4.3 Golomb Code 36
4.4 Shannon-Fano Coding, Huffman Coding 46
4.5 String Compression, Dictionary Method, Lempel-Ziv LZ77/LZ1 . . 55

5 Conclusion **66**

Bibliography **67**

Appendix A **67**
A.1 Even Weight Code 68
A.2 Hamming Code 70
A.3 Extension 72
A.4 Run Length Encoding 74
A.5 Golomb Coding 79
A.6 Shannon-Fano Coding 84
A.7 Lempel-Ziv 88

Chapter 1

Introduction and Background

Today's world is increasingly getting more digitalized. In fact, about six billion people are using the internet and other digitalized technologies. A big keyword regarding this technology is data. In our daily lives, everyone needs to transmit data over the internet, send and receive messages on smartphones, or save data on storage devices. Just as problems can occur when sending physical mail through the post office, transmission problems can also occur in the digital world. To guarantee correct transmission, theoretical and practical solutions had to be found. Correctness can be ensured in different ways with advantages and disadvantages. From this problem, in 1948 Coding theory emerged, with Claude Shannon generally regarded as one of its founders, introducing the concepts of entropy and channel capacity [1].

To get more concrete, exchanging data can be understood by considering as a communication system which consists of a source, transmitter, channel, receiver and destination. While data in the source can be letters, the transmitter uses the data as input for "encoding-functions". Via the channel, on which noises can disturb the transmission, the encoded information gets transmitted to the receiver. The receiver has to reverse the operations of the transmitter to finally restore the original data [1]. This layout lays the foundation of the mathematical way of approaching the problem. Coding Theory covers two main topics: ***channel coding*** and ***source coding***. The first focuses on the already mentioned task of reconstructing data after noises appeared in the channel. This happens with redundancy which establishes error-detection or -correction codes [2]. The second topic reasons about reducing the size or redundancy of the original data, hence fewer data have to be transmitted; this is also called compression. With the compressed data, the original one can still be restored [3, 4].

Introducing such algorithms comes together with formally verifying their validity. Proving such algorithms corresponds to proving properties in first-order logic (FOL). Since first-order logic is undecidable on infinite domains, there is no algorithm which can determine the logical correctness of the output for every input, hence this way of proving is not completely automatizable. However, there exists another way of guaranteeing the correctness of algorithms. Instead of proving on infinite domains, we may restrict ourselves to finite ones. In these domains, every execution of an algorithm or system can be tested for every input from the finite domain. The finite domain is called a model and the procedure is called ***formal model checking*** [5]. In detail, this procedure takes a model/system description and a logical formula which describes what the execution should satisfy, e.g. correct decompression. Then it will be checked if every possible execution of the

given algorithm/system is indeed a model for the logical formula; if yes, the algorithm is valid.

For model checking, the tool RISCAL (RISC Algorithmic Language) is very suitable [6, 7]. RISCAL provides a language where algorithms and logical properties can be described with logical formulas in first-order logic. RISCAL then allows to automatically check the validity of models of fixed size via going through the whole state space. There are also options to see every possible logical state of input and output when checking. RISCAL also produces counterexamples when algorithms are not valid [6].

Furthermore, RISCAL supports the verification of algorithms over state spaces of arbitrary size with the help of loop invariants which must hold before and after every loop iteration. From these, RISCAL generates verification conditions which imply the general validity of the algorithm [6].

RISCAL can also translate logical formulas into quantifier-free ones and check them with external SMT (Satisfiability Modulo Theories) solvers, which enhances the efficiency of the verification [6].

The contribution of this thesis is the formal modelling and implementation of selected channel- and source-coding techniques in RISCAL. The mathematical concepts underlying these techniques are translated into RISCAL specifications, together with executable implementations of the corresponding algorithms. Their relevant properties, such as correct encoding and decoding, error detection, error correction, and lossless compression, are formulated as logical specifications and checked using RISCAL. Thus, the thesis investigates how existing concepts from coding theory can be formally modelled and verified using RISCAL.

The considered channel-coding techniques include parity-based codes, repetition codes, and Hamming codes. The source-coding part considers, among others, run-length encoding, scalar quantization, variable-size and prefix codes, Golomb coding, Shannon-Fano coding, Huffman coding, and Lempel-Ziv-based dictionary methods. The implementations and specifications developed in this thesis are available at the RISC website

> https://www.risc.jku.at/research/formal/software/RISCAL/papers/bachelorthesis-Michlmayr.zip.

The rest of this thesis is structured as follows. Chapter 2 introduces the fundamentals of information theory, channel and source coding, as well as formal verification, model checking, and RISCAL. Chapter 3 focuses on channel coding and discusses error detection and correction using linear codes, including their implementation in RISCAL. Chapter 4 focuses on source coding and discusses various methods of data compression. Chapter 5 concludes the thesis by summarizing the main results and findings of the previous chapters. Appendix A contains code examples for the techniques discussed in the thesis.

Chapter 2

State of the Art

This chapter aims to describe the general development of coding theory and its fundamentals, consisting of the different types of coding, formal verification of algorithms, the concept of model checking and the formal language RISCAL used in this thesis.

2.1 Coding Theory Fundamentals

Coding theory has its roots in the 1940s and 1950s. Back then, there were message communication services like telegraphy and it became of interest to ensure a reliable way of data transmission, i.e., a certain message robustness against external influences and ways of error detection and correction. In 1948, Claude Elwood Shannon published in his paper ***"A mathematical theory of communication"*** [1] the core concepts of Information Theory. Hence, a communication system was defined as the path which data have to travel from an information source to the destination (see Figure 2.1). This section and the following two are based on Shannon's paper.

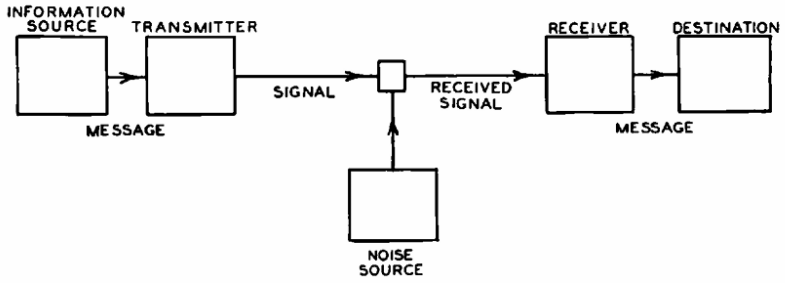

Figure 2.1: Concept of a Communication System [1]

At first, a message gets generated in the ***Information Source*** block. Then this message is prepared and sent into the ***Transmitter*** block. There the input is edited for a reliable transmission as we will see later (when discussing the coding and encoding of data). Also the message is converted into a medium/signal, e.g., electrical voltage or current, which fits best for transmission. Now this signal travels via the ***Channel*** to the ***Receiver*** block (the channel represents a wire, radio waves, etc). The Receiver has to apply the inverse operations to the received signal to restore the originally sent message. This message is now available at the ***Destination*** block. In the real world, data transmission suffers from external influences, called ***noise***, which occur while

the signal travels through the channel. This means that noise is added to our signal in the channel indicated by the ***noise*** block.

Communication systems are split into three categories: ***discrete, continuous and mixed***. Discrete means that messages and signals are discrete pieces of information, like letters as messages and bit sequences as signals. In continuous systems, messages and signals are represented as functions. Mixed ones combine the previous definitions. This thesis focuses purely on discrete communication systems because of later restrictions on finite domains for verification theory and algorithms.

Shannon argued that for discrete systems sequences of symbols from a finite set (like letters from the alphabet) are transmitted. First he thought about how much information can be sent via the channel. Therefore, consider the number of signals $N(T)$, which can be sent with duration T. Hence, $\log(N(T))$ bits are needed to identify each possible signal.

Definition 2.1.1. *Let* $C = \lim_{T\to\infty} \frac{\log(N(T))}{T}$ *where* $N(T)$ *is the number of signals of duration* T. *Then* C *is called* ***(channel) capacity****. The capacity is measured in bits per time unit.*

If all symbols have equal duration, the capacity can be interpreted as bits per symbol.

For now it is determined how much information can be sent via a certain channel. The information source needs to be analyzed, too. According to Shannon, such sources can be understood as stochastic processes, which produce sequences of symbols from finite domains (e.g., letters from the alphabet). Each symbol occurs with a certain probability. To determine the information, a mathematical measure $H(p_1, p_2, ..., p_n)$ is introduced, where $(p_1, p_2, ..., p_n)$ are the probabilities of n possible events. The following properties **(PROP1)** must be satisfied:

- H must be continuous in p_i for every i;
- if $p_i = \frac{1}{n}$, then H must be a monotonically increasing function in n;
- H must be additive for successive independent choices, i.e., the total information of a two-stage process equals the information of the first stage plus the expected information of the second one (e.g., see Figure 2.2).

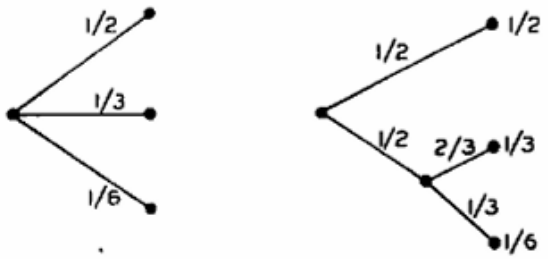

Figure 2.2: Additive Property of measure H with example $H(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}) = H(\frac{1}{2}, \frac{1}{2}) + \frac{1}{2}H(\frac{2}{3}, \frac{1}{3})$ [1]

The second property of **(PROP1)** indicates that equally likely events increase the general information of the source.

Definition 2.1.2. *Let* $H(p_1, ..., p_n) = -\sum_{i=1}^{n} p_i \log(p_i)$ *. Then* H *is the only function which satisfies (**PROP1**). We call H **Entropy** and sometimes write* H *for* $H(p_1, ..., p_n)$ *and* $H(x)$ *for the entropy of random variable x.*

Entropy helps to make statements about how much information is contained in a message, how uncertain an event is, and how many possible outcomes exist.

The following properties **(PROP2)** hold for the entropy:

- $H \geqslant 0$.
- $H = 0 \iff$ every p_i but one is equal to zero. Hence H vanishes when the outcome is certain.
- if the p_i are equally distributed ($p_i = \frac{1}{n}$), it holds $H = \log(n)$ and H admits its maximum. This indicates the most uncertain situation.
- Assume event x with m possibilities and event y with n. Then $p(i, j)$ is the probability of the joint occurrence of i for the first event and j for the second. Then the following holds:
  - $H(x) = -\sum_{i,j} p(i, j) \log(\sum_j p(i, j))$
  - $H(y) = -\sum_{i,j} p(i, j) \log(\sum_i p(i, j))$
  - $H(x, y) = -\sum_{i,j} p(i, j) \log(p(i, j))$
  - $H(x, y) \leqslant H(x) + H(y)$ where equality holds when x and y are independent, i.e., $p(i, j) = p(i)p(j)$.
- Assume two events x and y. The probability that y has value j under the assumption that x has value i is given by $p_i(j) = \frac{p(i,j)}{\sum_j p(i,j)}$. The average entropy of y for each possible x is known as **conditional entropy of y** given by $H_x(y) = -\sum_{i,j} p(i, j) \log(p_i(j))$.
- From the previous properties it follows that $H(y) \geqslant H_x(y)$.

Finally, the theory is expanded from a single symbol to a sequence G_N of symbols of length N. Its entropy is defined as $H = \lim_{N\to\infty} \frac{G_N}{N}$ and is also called **entropy rate**.

The size of symbols, which are not really needed (which are **redundant**), is also of interest:

Definition 2.1.3. *Let* $R(x) = H_{max} - H(x)$*, where* x *is a random variable,* $H_{max} = \log_2(N)$ *is the maximum entropy for* N *possible values of* x*, and* $H(x)$ *is the entropy of* x*.* $R(x)$ *is called the **redundancy of R(x)** and indicates how much information per symbol is not needed compared to the maximum possible information.*

This theory lays the foundation for coding theory and establishes the connection between entropy and channel capacity. With these characteristic values, it can be discussed which information can be transmitted with a certain entropy on channels with certain channel capacity. These fundamentals are important for algorithms of data compression (**source coding**) and error detection and -correction (**channel coding**), which we want to specify and validate in this thesis. In the next sections, source coding and channel coding will be explained further.

2.2 Channel Coding

We consider transmissions, where noises can disturb the sent messages, in particular noises which alter messages completely randomly every time. The transmission happens in a **noisy channel**. This theory investigates the possibility of correct transmission and data reconstruction (correction) of noise-disturbed messages.

First, the entropy can be considered. $H(x)$ represents the entropy of the source, $H(y)$ the entropy of the output. In the optimal case without noise $H(x) = H(y)$ should hold. The joint entropy is written as $H(x, y)$; we have $H(x, y) = H(x) + H_x(y) = H_y(x) + H(y)$.

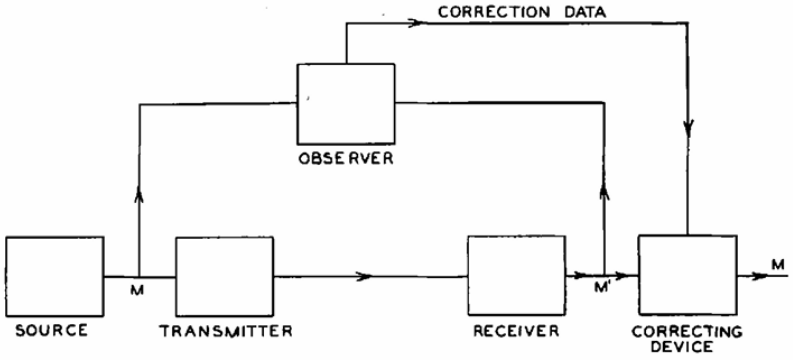

Figure 2.3: Correcting Data [1]

Receivers of noise-perturbed messages can no longer determine the transmitted message with certainty, therefore Shannon recalls for the conditional entropy $H_y(x)$ to quantify the uncertainty introduced by noise and calls it **equivocation**. Hence, he defines the actual rate of transmission as $R = H(x) - H_y(x)$ which indicates the average amount of information transmitted correctly. Furthermore, Shannon proved that error correction is possible, as long as the correction channel's capacity is greater than or equal to $H_y(x)$ and that $H_y(x)$ must be sent for error correction. Therefore, the channel capacity of noisy channels can be redefined as $C = max(H(x) - H_y(x))$.

Finally, Shannon proves the existence of coding schemes providing certain correctness. Assuming H as entropy per second and C as channel capacity, $H \leqslant C$ implies the existence of a coding system such that transmissions contain only an arbitrarily small number of errors. If $H > C$, lossless transmission is not possible. $H - C$ indicates the lower bound of equivocation. It is only possible to encode the source such that the equivocation is $H - C + \epsilon$ for arbitrarily small ϵ.

2.3 Source Coding

With the established definitions of capacity and entropy, the efficiency of channels bothered Shannon. Therefore he explained based on the English language, how much information is needed, such that a sentence can be understood uniquely (e.g., how many letters of words are needed, such that the receiver of these words understands them). Hence, this theory, source

coding, deals with efficient compression of source information and efficient decompression at the receiver in a **noiseless channel**. This means, that we are trying to minimize transmitted symbols, such that the channel with capacity C is used in the best way possible. Data which can be compressed consist of **redundant** information.

First, the whole process can be realized with the help of functions. Therefore, we define the **encoding function** $E : S \to C'$ and **decoding function (or inverse)** $D : C' \to S$ where S is the set of source sequences and C' the set of channel sequences. A reliable correct compression and decompression must satisfy $D(E(x)) = x, \forall x \in S$.

Considering the ratio $\frac{C}{H}$, it can be determined how many source symbols can be transmitted per channel symbol on average. Shannon proved that transmitting with a rate greater than this ratio is impossible. This means $H \leqslant C$ must hold for a lossless transmission. For sufficiently large sequence lengths, coding schemes can be constructed with rates which approach $\frac{C}{H}$ arbitrarily closely.

2.4 Formal Verification

Formal verification is the process of proving or disproving formally specified properties of computer programs or computing systems. Here, we reason (based on **mathematical logic**) about all possible behaviours of (a model) of the program/system instead of empirical testing sample executions. Program/system specifications can be expressed precisely in **first-order logic (FOL)** [6]. In particular, the specification of a program/function/algorithm can be described by a pair of logical formulas, the precondition and the postcondition. This approach can be also applied to the verification of coding algorithms.

However, proving properties in FOL comes with certain difficulties. The reason is the undecidability of first-order logic, which means that there is no algorithm which is able to decide the logical validity of every formula, hence proving in FOL is not completely automatizable. In order to be capable of automated verification, one may restrict the consideration to **finite models**, because FOL on finite domains is decidable [6].

2.5 Model Checking

Model checking is a technique for verifying programs/systems that operate on finite domains by checking all reachable states with respect to given specific properties [6].

The big advantage of model checking is that it is completely automatizable. Furthermore, while going through the state space, counterexamples of executions can be found that violate the specified properties. However, since model checking is a state-driven technique, the number of possible states of complex models increases very quickly, which causes a longer processing time. Therefore, optimized techniques based on symbolic representation or the use of SMT-solvers may be used.

2.6 RISCAL

The RISC Algorithm Language (**RISCAL**) is a tool for the formal verification of theorems and algorithms, which has been developed at the Research Institute for Symbolic Computation (RISC). RISCAL is designed especially for verifying **finite** models via model checking. Therefore, the language uses a variant of FOL and set theory that restrict the types of variables to finite domains. RISCAL algorithms look like typical code in programming languages. When verifying an algorithm, RISCAL evaluates the whole space of input values and checks the execution for every possible value. States which do not satisfy expected conditions represent counterexamples that are automatically reported. RISCAL also features efficient model checking by the use of external SMT solvers like Z3, where formal properties are translated into quantifier-free formulas which can be automatically decided by the solvers [6].

This thesis presents a substantial amount of RISCAL code; therefore, its fundamentals are explained below.

Constants defined in the form techniques *val [constantname] : [domain/type]* are parameters for the model. New datatypes can be created by *type typename = ...* where ... can stand for the type of numbers up to a certain size or for datatypes like arrays of certain size and type. The following example shows the definition of a **constant N**, which is from the natural numbers, the definition of a **type Bit**, which is a natural number up to 1, and the definition of a **type Word**, which is an array of N values of type Bit:

```
1    val N: ℕ;
2    type Bit = ℕ[1];
3    type Word = Array[N, Bit];
```

RISCAL distinguishes between functions *fun* and procedures *proc*, which provide different styles for describing computations. Functions follow a functional programming style and are defined by a single expression whose evaluation determines the return value. They may contain local definitions using *let x = t1 in t2* and can also be defined recursively. This style is particularly well suited for concise mathematical specifications, where the result can be expressed directly as a formula or recursive definition. They will be indicated by

fun [name]([parameters]) : [return type] = ...

```
1    fun parityWord(w: Word, i: Nat[N]): Bit =
2    if i = 0 then w[0]
3    else parityWord(w,i-1) ^ w[i];
```

Procedures, in contrast, follow an imperative programming style. They consist of local variables declared with *var x := T*, assignments that modify the program state, and control structures such as conditionals and loops. This allows algorithms to be described step by step and makes procedures especially suitable for implementations that iteratively process data. They will be indicated by

proc [name](parameters) : [return type] [code] return...

```
    proc encode(w: Word): Codeword
    {
        var ret: Codeword;
        ...
        return ret;
    }
```

In general, functions are preferable whenever an algorithm can be expressed directly as an expression, whereas procedures are more appropriate for algorithms that naturally rely on state changes or iterative computations.

Logical specifications of theories can be stated by predicates which are introduced by the *pred* keyword;

```
    pred validCodeword(c: Codeword) ⇔ parityCodeword(c,N) = 0;
```

With the help of *requires* and *ensures* functions and procedures can be annotated by logical conditions: *requires* indicates the logical condition that the parameters should fulfill. *ensures* indicates the logical condition that the output should satisfy:

```
    proc encode(w: Word): Codeword
    requires validWord(word);
    ensures validCodeword(result);
    {
        ...
    }
```

theorem enables the formalization of logical theorems, e.g., the correctness of de- and encoding:

```
    theorem decodingCorrect(w: Word) ⇔ decode(encode(w)) = w;
```

For more details on the RISCAL homepage, see the tutorial and reference manual [8].

Chapter 3

Channel Coding

Reliable transmission over noisy communication channels requires mechanisms for detecting and correcting transmission errors. This chapter introduces the basic concepts of error-correcting codes and presents Hamming codes as fundamental examples of linear block codes.

3.1 Parity Bit, Even Weight- and Repetition Code

When talking about channel coding, parity-bit theory is fundamental as it is the foundation for more advanced techniques.

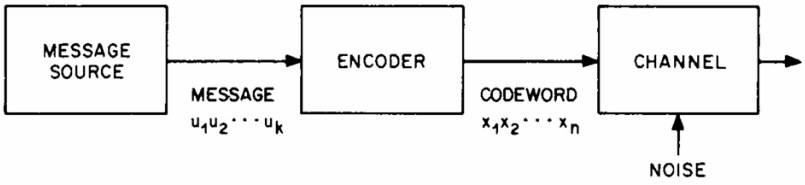

Figure 3.1: Transmitting and encoding messages [9]

Figure 3.1 illustrates the transmission process of a message u which is built by k symbols $u = u_1, u_2, ..., u_k$. The encoded message has n symbols, known as **length**, therefore the encoder appends $n - k$ **parity bits** such that we receive a codeword $x = x_1, x_2, ..., x_n$ after encoding our message. The encoded message consists of the original k message symbols and the $n - k$ parity bits. The core principle for detecting errors is, that the equation

$$H \begin{pmatrix} x_1 \\ ... \\ x_n \end{pmatrix} = 0 \quad \textbf{(PEQ)}$$

holds. H denotes the $(n - k) \times n$ **parity-check matrix** which is defined as $H = [A|I_{n-k}]$, where A is a $(n - k) \times k$ matrix and I_{n-k} is the $(n - k) \times (n - k)$ identity matrix. For binary coding, the entries of codewords and all associated matrices are elements of $\mathbb{F}_2 = \{0, 1\}$. For q-ary coding, they are elements of the finite field $\mathbb{F}_q$. For now, binary coding is considered, if not explicitly mentioned otherwise. The components of A depend on the expectations of the algorithm (we will come to this later). To achieve error detection, the additional parity bits have to be set correctly, i.e., with given A the equation **(PEQ)** must be solved for the

unknown parity bit variables. Hence, these equations for each component are called **parity check equations** [9].

The parity check equations provide a complete characterization of valid codewords. We therefore define the code as the set of all vectors satisfying these equations.

Definition 3.1.1. *Let H be a binary parity-check matrix. The corresponding **code** C is defined as* $C = \{x \in \mathbb{F}_2^n : Hx^T = 0\}$.

*Furthermore, a code is called **linear**, if for all codewords $x, y \in C$, the sum $x+y$ is also a codeword in C, and for every field element c, cx is also a codeword in C.*

For these x, y, and c it holds $H(x+y)^T = Hx^T + Hy^T = 0$ and $H(cx)^T = cHx^T = 0$.

This thesis discusses linear codes, if not explicitly mentioned otherwise.

We are now considering the **even weight code**. Here, the codeword with length n must contain an even number of ones by adding just one parity bit. Hence, $n - k = 1$ holds. Therefore, H is just a row vector as its components only represent a single equation. For even weight, H consists of components with value 1, since all bits have to be added to check for an even number of ones.

Another way of using parity bits is the **repetition code**. Here, single bits are encoded by sending each bit n times, thus $k = 1$ is considered. H is defined as

$$H = \begin{vmatrix} 1 & 1 & 0 & \dots & 0 \\ 1 & 0 & 1 & 0 & 0 \\ \dots & \dots & \dots & 1 & 0 \\ 1 & 0 & \dots & \dots & 1 \end{vmatrix}.$$

This means that the codeword consists either only of ones or only of zeroes, i.e., the same symbol is transmitted multiple times [9].

Considering codewords, the **generator matrix G** can be derived from **(PEQ)** and easy transformations as $G = [I_k | -A^T]$ which results in $x = uG$. The codewords are just all linear combinations of the rows of G. For such linear codes, there are always 2^k codewords, if H has $n-k$ linearly independent rows, where k is called the **dimension**. This gives a good characterization for codes [9]:

Definition 3.1.2. *Let C be a linear code with length n and dimension k. Then C is called a $[n,k]-$ code with **rate** $R = \frac{k}{n}$. The code consists of 2^k codewords.*

There is no unique generator matrix for a single code [9].

It is of interest, how efficient such coding techniques are and for what they can be used especially, for instance questions like "How many errors can be detected?", "How many errors can be corrected" or simply the search for codes with better rates are common. For answering, a few more concepts are needed.

We now consider the error vector $e = y - x = e_1 e_2 ... e_n$, where y is the received vector and x the codeword.

For correct identification and detection of errors, codewords can be analyzed further. For comparing messages, especially codewords, the amount of different bits can be checked:

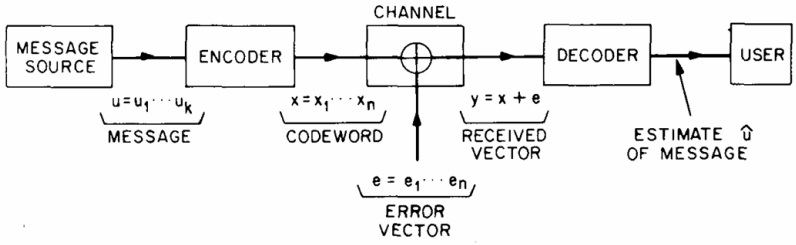

Figure 3.2: Encoding and decoding considering errors in transmission [9]

Definition 3.1.3. *The number of positions with different values between two vectors* x *and* y *is called* ***Hamming distance*** *denoted by* $\mathrm{dist}(x, y)$.

Single messages/vectors can be also characterized by the number of ones:

Definition 3.1.4. *The number of nonzero positions of a vector* x *is called* ***Hamming weight*** *denoted by* $wt(x)$.

Considering the previous definitions, it follows that for the error vector $x - y$ between the vectors x and y, it holds that $\mathrm{dist}(x, y) = wt(x - y)$.

Theorem 3.1.1. *Let* x *and* y *be binary vectors. The intersection* $x * y$ *of* x *and* y *is defined as* $x * y = (x_1 y_1, ..., x_n y_n)$*, i.e., the component* $(x * y)_i$ *is* 1 *if* x_i *and* y_i *are both* 1*. Then* $wt(x + y) = wt(x) + wt(y) - 2wt(x * y)$.

By Theorem 3.1.1 it follows $wt(x+y) = (wt(x)+wt(y)) \bmod 2$. Hence, for the weight of the sum the following observation can be made:

- $wt(x + y)$ is even if $wt(x)$ is even and $wt(y)$ is even.
- $wt(x + y)$ is odd if $wt(x)$ is even and $wt(y)$ is odd (or vice versa).
- $wt(x + y)$ is even if $wt(x)$ is odd and $wt(y)$ is odd.

Thus it follows, that in a linear code either every codeword has even weight or half of the codewords consists of words with even weight and the other of words have odd weight **(Weight-Theorem)** [9].

While the Hamming distance compares two individual codewords, a code C should also have a characteristic distance. Therefore, the minimum distance of a code is defined as the smallest Hamming distance between any two distinct codewords:

Definition 3.1.5. *Let* C *be a linear code. The* ***minimum distance*** *of* C *is defined as* $\mathrm{mindist}(C) := \min_{x,y \in C, x \neq y} \mathrm{dist}(x, y) = \min_{x,y \in C, x \neq y} wt(x - y)$.

The previous characterization of codes Definition 3.1.2 can be extended:

Definition 3.1.6. *Let* C *be a linear code with length* n*, dimension* k *and* $\mathrm{mindist}(C) = d$*. Then* C *is called a* $[n, k, d]-$ *code.*

Thus, every pair of valid codewords differs in at least d positions. Intuitively, a larger minimum distance results in easier distinction between two codewords [9]. The minimum distance determines important properties of a code:

Theorem 3.1.2. *The minimum distance of a linear code is the minimum weight of any nonzero codeword.*

Hence, determining the minimum distance does not require comparing every pair of codewords. Instead, it is sufficient to inspect the weights of all nonzero codewords [9].

Finally, the minimum distance determines the error-correcting capability of a code:

Theorem 3.1.3. *A linear code* C *with* $\mathrm{mindist}(C) = d$ *can correct* $\lfloor \frac{d-1}{2} \rfloor$ *errors. Furthermore, if* d *is even, the code can* ***simultaneously*** *detect* $\frac{d}{2}$ *errors and correct* $\frac{d-2}{2}$ *errors.*

The following theorem summarizes the general relationship between the minimum distance of a code and its error-correcting capability:

Theorem 3.1.4. *A linear code* C *can* ***simultaneously*** *correct at most* a *errors and detect up to* b *errors with* $b > a$ *if and only if* $\mathrm{mindist}(C) \geqslant a + b + 1$.

In particular, every code with minimum distance d *can detect up to* $d - 1$ *errors.*

This theory can be implemented in RISCAL. In this code example, even weight codes with length n and one parity bit are considered.

```
val K: Nat;
val N = K+1;
type Bit = Nat[1];
type Message = Array[K,Bit];
type Codeword = Array[N,Bit];
```

At first, we define the types *Bit*, *Message* and *Codeword* where *K* indicates the bit size the original message has and *N* the according codeword size. Later executions are based on *K=24*.

```
fun xor(a: Bit, b: Bit): Bit =
if a = b then 0 else 1;

fun parityMessage(m: Message, i: Nat[K-1]): Bit =
if i = 0 then m[0] else
xor(parityMessage(m,i-1), m[i]);

fun parityCodeword(c: Codeword, i: Nat[N-1]): Bit =
if i = 0 then c[0] else
xor(parityCodeword(c,i-1), c[i]);
```

By logical XOR operations, the parity of bit sequences can be evaluated. Therefore, a *xor* function has been established. For checking the parity of messages and codewords, separate recursive functions have been implemented.

```
pred validCodeword(c: Codeword) ⇔ parityCodeword(c,N-1) = 0;

pred singleError(c1: Codeword, c2: Codeword) ⇔
     ∃i:Nat[N-1].
     c1[i] ≠ c2[i] ∧
     ∀(j:Nat[N-1] with j ≠ i. c1[j] = c2[j]);
```

The predicates *validCodeword* and *singleError* indicate the properties, if encoded messages are valid and if a single error occurred during data transmission.

```
    proc encode(m: Message): Codeword
    ensures validCodeword(result);
    {
        var c: Codeword := Array[N,Bit](0);
        for var i:Nat[K] := 0; i < K; i := i+1 do
        c[i] := m[i];
        c[N-1] := parityMessage(m,K-1);
        return c;
    }

    proc decode(c: Codeword): Message
    requires validCodeword(c);
    {
        var m: Message := Array[K,Bit](0);
        for var i:Nat[K] := 0; i < K; i := i+1 do
        m[i] := c[i];
        return m;
    }
```

The procedure *encode* shows the simple encoding process for even weight codes. It ensures, that the process produces a valid codeword via the predicate *validCodeword*.

The execution of RISCAL checks that for all inputs that satisfy the stated precondition the procedure *encode* returns a result that satisfies the stated postcondition:

```
    Executing encode(Array[ℤ]) with all 16777216 inputs inputs.
    Execution completed for ALL inputs (196242 ms, 16777216 checked,
    0 inadmissible).
```

decode decodes the codeword by extracting the bits coming before the parity bit. Here, *validCodeword* is required, which is ensured by the encoding procedure.

```
    theorem DecodeEncode(m: Message) ⇔ decode(encode(m)) = m;
    theorem DetectSingleBitError(m: Message, c: Codeword) ⇔
            singleError(encode(m),c) ⇒ ~validCodeword(c);
```

In the end, we want to establish theorems, namely *DecodeEncode*, which shows the validity of the encoding theory, and *DetectSingleBitError*, which shows that every single-error noise will be detected. RISCAL checks for all values of the parameters that these theorems hold:

```
    Executing DecodeEncode(Array[ℤ]) with all 16777216 inputs.
    Execution completed for ALL inputs (327631 ms, 16777216 checked,
    0 inadmissible).

    Executing DetectSingleBitError(Array[ℤ],Array[ℤ]) with all
    33554432 inputs.
    Execution completed for ALL inputs (309154 ms, 33554432 checked,
    0 inadmissible).
```

For the last theorem, `K=12` has been used to achieve a faster execution time.

3.2 Identifying Transmission Errors

To identify the positions of transmission errors, the concept of cosets is introduced.

Definition 3.2.1. *Let C be a linear code and let $a \in \mathbb{F}_2^n$. The **coset** of C determined by a is the set*

$$a + C = \{ a + x : x \in C \}.$$

*The vector a is called a **coset representative**.*

The family of all distinct cosets partitions $\mathbb{F}_2^n$.

Cosets are similar to equivalence classes in mathematics. Two vectors belong to the same coset if and only if their difference lies in C. Consequently, the cosets form a partition of $\mathbb{F}_2^n$. Every vector belongs to exactly one coset, and any two cosets are either disjoint or identical [9].

Since every received vector y belongs to exactly one coset, there exists a unique coset representative a_i such that $y \in a_i + C$, where a_i denotes the representative of the i-th coset. In decoding, the representatives are chosen to be the minimum-weight vectors of their respective cosets. Let $z \in C$ be the transmitted codeword. Thus, the error vector is $e = y - z$. Since $y = a_i + x$ for some codeword x, it follows that $e = y - z = a_i + x - z = a_i + x'$ with $x' = x - z \in C$. Hence, the error vector belongs to the same coset $a_i + C$ as the received vector. Since every received vector belongs to exactly one coset, all possible error vectors associated with y lie in this unique coset. As error vectors with smaller Hamming weight are more likely to occur, decoding selects the vector of minimum Hamming weight within the coset (**maximum likelihood decoding**). These minimum-weight representatives are called **coset leaders**. The representatives a_i introduced above are precisely these coset leaders [9].

A useful representation of all cosets is the **standard array**, whose first row contains all codewords of the code C, beginning with the zero codeword. Each subsequent row represents one coset and starts with its coset leader. To decode a received vector y, the decoder locates y in the standard array and selects the coset leader of the corresponding row as the estimated error vector. By Theorem 3.1.3, this decoding strategy is correct whenever the number of transmission errors does not exceed $\lfloor \frac{d-1}{2} \rfloor$ [9].

Instead of searching the standard array directly, each coset can be identified by its **syndrome**. For a received vector y, the syndrome is defined as $S = Hy^T$, where S is an $(n-k)$ dimensional column vector. Since every codeword x satisfies $Hx^T = 0$, it follows from $y = x + e$ that $S = Hy^T = H(x + e)^T = Hx^T + He^T = He^T$. Thus, the syndrome depends only on the error vector and not on the transmitted codeword. For binary codes, it is obtained by adding modulo 2 the columns of H corresponding to the error positions. Consequently, every coset has a unique syndrome, and two vectors belong to the same coset if and only if they have the same syndrome [9].

As discussed above, decoding via the maximum likelihood method might not always be successful. Although the coset leader is the most likely error vector within a coset, transmission errors of larger Hamming weight may also occur. Consequently, the decoder may decode the received vector incorrectly, resulting in a decoding error. This raises the question of how likely such decoding failures are. Let P_{err} be the probability of a decoding error of code C. Let $t = \lfloor \frac{d-1}{2} \rfloor$ denote the error-correcting capability of a code with $\mathrm{mindist}(C) = d$. For codes with distance $d = 2t + 1$ or $d = 2t + 2$, every error vector of weight at most t is the unique minimum-weight vector of its coset and is therefore a coset leader. Hence, $\alpha_i = \binom{n}{i}$ for $0 \leqslant i \leqslant t$, where α_i denotes the number of coset leaders of weight i. This yields the following upper bound for P_{err} with small channel error probability $p < \frac{1}{2}$ [9] :

$$P_{err} \leqslant 1 - \sum_{i=0}^{t} \binom{n}{i} p^i (1-p)^{n-i} - \alpha_{t+1} p^{t+1} (1-p)^{n-t-1} \leqslant 1 - \sum_{i=0}^{t} \binom{n}{i} p^i (1-p)^{n-i}$$

The upper bound becomes an equality if $\alpha_i = 0$ for $i > t$, i.e., if there are no coset leaders of weight greater than t. Codes with this property are called **perfect**. For such codes, every error vector of weight at most t is the unique minimum-weight vector of its coset and is therefore decoded correctly. Error vectors of larger weight, however, cannot be corrected with certainty.[9]

This theory is important for the next class of linear codes, **Hamming Codes**.

3.3 Hamming Code

Hamming codes form a family of linear block codes that were specifically designed for error detection and correction. They are among the most important examples of binary linear codes and provide an efficient method for correcting transmission errors. The standard Hamming codes are designed to correct a single transmission error. Error locations can be identified using the syndrome, making syndrome decoding particularly efficient. Furthermore, Hamming codes are perfect codes.

With the established connection between syndrome and columns of the parity-check matrix H, the columns of H are chosen to be nonzero and pairwise distinct. For H with r rows, there are $2^r - 1$ columns/vectors with length r. Therefore, we get a Hamming code with length $n = 2^r - 1$. The columns of H represent numbers from 1 to n in binary representation. This motivates the definition of Hamming codes [9]:

Definition 3.3.1. *For $r \geqslant 2$, a binary (standard) **Hamming code** H_r is the linear code whose parity-check matrix H consists of all distinct nonzero binary column vectors of length r. Hence, H_r has length $n = 2^r - 1$, dimension $k = 2^r - 1 - r$ and $\mathrm{mindist}(H_r) = 3$.*

Permuting the columns of H does not change the code parameters or its error-correcting capability. Thus, two codes are called **equivalent** if one is obtained from the other by a permutation of the coordinate positions. So, Hamming codes are unique up to equivalence [9].

Let H_i denote the i-th column of H, corresponding to the binary representation of the integer i. Let there be a single error at position k, the error vector is $e = e_k$. Due to $S = He^T$ and $He^T = H_k$ the syndrome equals the k-th column of H, so the error position is obtained immediately [9].

We now implement a [7,4,3] Hamming code in RISCAL: For the matrix we can use

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

```
type Bit = Nat[1];
type Message = Array[4,Bit];
type Codeword = Array[7,Bit];
```

We are using the same types as in the even weight code example with the only difference of fixed sizes for *Message* and *Codeword*, since we want to work with a specific Hamming code.

```
fun xor(a: Bit, b: Bit): Bit =
if a = b then 0 else 1;
```

The *xor* function is inherited from the previous example.

```
fun p1(m: Message): Bit =
xor(xor(m[0],m[1]),m[3]);

fun p2(m: Message): Bit =
xor(xor(m[0],m[2]),m[3]);

fun p3(m: Message): Bit =
xor(xor(m[1],m[2]),m[3]);
```

As mentioned in the beginning of the chapter, the parity bits are appended as the last $n - k$ bits for matrices in the form of $H' = [A|I_{n-k}]$. Comparing H' with H, we notice the parity-bit-columns $I_{n-k}]$ of H' are columns 1, 2 and 4 of H, therefore, the parity bits must be now at position $1, 2$ and 4. This is implemented via the functions *p1*, *p2*, *p3*.

```
pred validCodeword(c: Codeword) ⇔
c[0] = xor(xor(c[2],c[4]),c[6]) ∧
c[1] = xor(xor(c[2],c[5]),c[6]) ∧
c[3] = xor(xor(c[4],c[5]),c[6]);

pred singleError(c1: Codeword, c2: Codeword) ⇔
     ∃i:Nat[6].
     c1[i] ≠ c2[i] ∧ ∀(j:Nat[6] with j ≠ i. c1[j] = c2[j]);
```

The predicate *validCodeword* checks now the various parity checks. For simulating exactly one bit errors the predicate *singleError* is implemented. Two codewords are analyzed for a different bit in just one position.

```
proc encode(m: Message): Codeword
ensures validCodeword(result);
{
    var c: Codeword := Array[7,Bit](0);
    c[0] := p1(m);
    c[1] := p2(m);
    c[2] := m[0];
    c[3] := p3(m);
```

```
        c[4] := m[1];
        c[5] := m[2];
        c[6] := m[3];
        return c;
    }

    proc decode(c: Codeword): Message
    requires validCodeword(c);
    {
        var m: Message := Array[4,Bit](0);
        m[0] := c[2];
        m[1] := c[4];
        m[2] := c[5];
        m[3] := c[6];
        return m;
    }
```

The *encode* procedure is plugging the various calculated bits together to a codeword, which is ensured to be valid by the predicate. *decode* is picking only the data bits to restore the message. For decoding a valid codeword is required again. *encode* satisfies its specification:

```
    Executing encode(Array[ℤ]) with all 16 inputs.
    Execution completed for ALL inputs (23 ms, 16 checked,
    0 inadmissible).
```

```
    proc syndrome(c: Codeword): Array[3,Bit]
    {
        var s: Array[3,Bit]:= Array[3,Bit](0);
        s[0] := xor(xor(xor(c[0],c[2]),c[4]),c[6]);
        s[1] := xor(xor(xor(c[1],c[2]),c[5]),c[6]);
        s[2] := xor(xor(xor(c[3],c[4]),c[5]),c[6]);
        return s;
    }

    fun syndromeValue(s: Array[3,Bit]): Nat[7] = 4*s[2] + 2*s[1] + s[0];

    fun syndromeValueOfCodeword(c: Codeword): Nat[7] =
    4 * xor(xor(xor(c[3],c[4]),c[5]),c[6])
    + 2 * xor(xor(xor(c[1],c[2]),c[5]),c[6])
    + xor(xor(xor(c[0],c[2]),c[4]),c[6]);
```

Since the syndrome is $S = Hc^T$ for a codeword c, *syndrome* forms the calculation of the components of S. With *syndromeValue* and *syndromeValueOfCodeword* the concrete bit position, where an error occurred, can be identified.

```
    proc correct(c: Codeword): Codeword
    {
        var r: Codeword := Array[7,Bit](0);
        for var i:Nat[7] := 0; i < 7; i := i+1 do r[i] := c[i];
        var s: Array[3,Bit]:= Array[3,Bit](0);;
        s := syndrome(c);
        var pos: Nat[7];
        pos := syndromeValue(s);
        if pos > 0 then r[pos-1] := xor(r[pos-1],1);
```

```
        return r;
    }
```

The procedure *correct* calculates the syndrome and corrects error bits.

```
    theorem DecodeEncode(m: Message) ⇔ decode(encode(m)) = m;
    theorem DetectSingleError(m: Message,c: Codeword) ⇔
            singleError(encode(m),c) ⇒ syndromeValueOfCodeword(c) > 0;
    theorem CorrectSingleError(m: Message,c: Codeword) ⇔
            singleError(encode(m),c) ⇒ correct(c) = encode(m);
```

The theorem *DecodeEncode* shows the property, when decoding an encoded message (without errors), the original message is obtained. *DetectSingleError* is about the fact, that if a single error occurs, the syndrome must be bigger than 0. Thus Hamming is able to detect a single error. *CorrectSingleError* shows the capability of fixing an error, if only one occurred.

The theorems hold:

```
    Executing DecodeEncode(Array[ℤ]) with all 16 inputs.
    Execution completed for ALL inputs (38 ms, 16 checked,
    0 inadmissible).

    Executing DetectSingleError(Array[ℤ],Array[ℤ]) with all 2048 inputs.
    Execution completed for ALL inputs (844 ms, 2048 checked,
    0 inadmissible).

    Executing CorrectSingleError(Array[ℤ],Array[ℤ]) with all 2048 inputs.
    Execution completed for ALL inputs (795 ms, 2048 checked,
    0 inadmissible).
```

```
    pred doubleError(c1: Codeword, c2: Codeword) ⇔
         ∃i:Nat[6], j:Nat[6]. i ≠ j ∧ c1[i] ≠ c2[i] ∧ c1[j] ≠ c2[j]
         ∧ ∀(k:Nat[6] with k ≠ i ∧ k ≠ j. c1[k] = c2[k]);

    theorem DetectsDoubleError(m: Message,c: Codeword) ⇔
            doubleError(encode(m),c) ⇒ ~validCodeword(c);
```

Since the $[7,4,3]$ Hamming code is also capable of detecting 2 errors, the predicate *doubleError* is useful to check this property. Further, *DetectsDoubleError* checks, if two error occur, the codeword must be invalid.

```
    Executing DetectsDoubleError(Array[ℤ],Array[ℤ]) with all 2048 inputs.
    Execution completed for ALL inputs (1705 ms, 2048 checked,
    0 inadmissible).
```

The theory of channel coding also yields possibilities to construct new codes from old ones. There exists different methods [9]:

Let C be an $[n, k, d]$ binary code. The method of **extending** a code C to C' consists of adding zeroes at the end of codewords with even weight and ones at the end of codewords with odd weight. Therefore, the new parity check equation $x_1 +$

$\ldots + x_{n+1} = 0$ is satisfied. The new parity check matrix H' is now:

$$H' = \begin{pmatrix} 1 & 1 & . & . & . & 1 \\ H_{1,1} & H_{1,2} & . & . & H_{1,n} & 0 \\ . & . & . & . & . & . \\ H_{n-k,1} & H_{n-k,2} & . & . & H_{n-k,2} & 0 \end{pmatrix}$$

where $H_{i,j}$ denotes the entry of the previous $(n-k) \times n$ check matrix H in row i and column j. The distances between all different codewords are even. If C's minimum distance d is **odd**, the minimum distance of C' is $d+1$. Therefore, $[n+1, k, d+1]$ codes with higher error detection capability can be constructed. For example, standard Hamming codes H_r with $mindist(H_r) = 3$ can be extended to **extended Hamming codes** H'_r with $mindist(H'_r) = 4$.

This can be also implemented in RISCAL:

```
    val K: Nat;
    val N = K;
    val NXT = N+1;

    type Bit = Nat[1];
    type Codeword = Array[N,Bit];
    type ExtendedCodeword = Array[NXT,Bit];
```

For the executions of procedures, *K=5* has been chosen again.

```
    fun xor(a: Bit, b: Bit): Bit =
    if a = b then 0 else 1;
```

```
    fun parity(c: Codeword, i: Nat[N-1]): Bit =
    if i = 0 then c[0] else xor(parity(c,i-1), c[i]);
```

```
    fun extendedBit(c: Codeword, i: Nat[N]): Bit =
    if i = N then parity(c,N-1) else c[i];

    fun parityExtended(c: Codeword, i: Nat[N]): Bit =
    if i = 0 then extendedBit(c,0) else
    xor(parityExtended(c,i-1), extendedBit(c,i));
```

extendedBit returns a bit of the extended codeword. The parity function is also updated for the extended case.

```
    pred validExtension(c: Codeword) ⇔ parityExtended(c,N) = 0;
```

validExtension is the property, that every new codeword has even parity.

```
    proc extend(c: Codeword): ExtendedCodeword
    ensures ∀i:Nat[N-1]. result[i] = c[i];
    ensures result[N] = parity(c,N-1);
    {
        var e: ExtendedCodeword := Array[NXT,Bit](0);
        for var i:Nat[N] := 0; i < N; i := i+1 do
        e[i] := c[i];
        e[N] := parity(c,N-1);
        return e;
    }
```

```
    Executing extend(Array[ℤ]) with all 32 inputs.
    Execution completed for ALL inputs (126 ms, 32 checked,
    0 inadmissible).
```

```
    theorem ExtendingAddsParityBit(c: Codeword) ⇔
            extendedBit(c,N) = parity(c,N-1);

    theorem ExtendingPreservesCodeword(c: Codeword, i: Nat[N-1])
    ⇔ extendedBit(c,i) = c[i];

    theorem ExtendingCorrect(c: Codeword) ⇔ validExtension(c);
```

ExtendingAddParityBit is about the correctness of the parity bit. *ExtendingPreservesCodeword* indicates that previous codewords must still be feasible. Theorem *ExtendingCorrect* checks if the extended codeword has even parity. These theorems can also be checked by RISCAL with *K=22*:

```
    Executing ExtendingAddsParityBit(Array[ℤ]) with all 4194304 inputs.
    Execution completed for ALL inputs (41655 ms, 4194304 checked,
    0 inadmissible).

    Executing ExtendingPreservesCodeword(Array[ℤ],ℤ) with all
    92274688 inputs.
    Execution completed for ALL inputs (120299 ms, 92274688 checked,
    0 inadmissible).

    Executing ExtendingCorrect(Array[ℤ]) with all 4194304 inputs.
    Execution completed for ALL inputs (56516 ms, 4194304 checked,
    0 inadmissible).
```

```
    fun distanceFrom(c1: Codeword, c2: Codeword, i: Nat[N-1]):
    Nat[N] =
    if i = 0 then if c1[0] = c2[0] then 0 else 1
    else distanceFrom(c1,c2,i-1) + (if c1[i] = c2[i] then 0 else 1);

    fun distance(c1: Codeword, c2: Codeword): Nat[N] =
    if N = 0 then 0 else distanceFrom(c1,c2,N-1);

    fun extendedDistance(c1: Codeword, c2: Codeword): Nat[NXT] =
    if extendedBit(c1,N) = extendedBit(c2,N) then distance(c1,c2)
    else distance(c1,c2) + 1;
```

To ensure the minimum distance property of extending, the function *distance* measures the distance between two codewords. *distanceFrom* and *extendedDistance* are for getting the hamming distance for codewords and extended ones.

```
    pred minimumDistance(d: Nat[N]) ⇔
         ∀(c1:Codeword, c2:Codeword with c1 ≠ c2.
         distance(c1,c2) ≥ d) ∧ ∃(c1:Codeword, c2:Codeword.
         c1 ≠ c2 ∧ distance(c1,c2) = d);

    pred extendedMinimumDistance(d: Nat[NXT]) ⇔
         ∀(c1:Codeword, c2:Codeword with c1 ≠ c2.
```

```
        extendedDistance(c1,c2) ≥ d)
        ∧ ∃(c1:Codeword, c2:Codeword. c1 ≠ c2 ∧
        extendedDistance(c1,c2) = d);
```

To determine the minimum distance of a code, the pairwise Hamming distances between distinct codewords are considered. The functions *minimumDistance* and *extendedMinimumDistance* perform this check by verifying whether a given value is the minimum distance of the respective code.

```
theorem ExtendingEvenDistance(c1: Codeword, c2: Codeword)
requires c1 ≠ c2 ∧ distance(c1,c2) % 2 = 0;
⇔ extendedDistance(c1,c2) = distance(c1,c2);

theorem ExtendingOddDistance(c1: Codeword, c2: Codeword)
requires c1 ≠ c2 ∧ distance(c1,c2) % 2 = 1;
⇔ extendedDistance(c1,c2) = distance(c1,c2) + 1;

theorem MinimumDistanceAfterExtension(d: Nat[N])
requires minimumDistance(d) ∧ d % 2 = 1;
⇔ extendedMinimumDistance(d+1);
```

ExtendingEvenDistance checks, if the distance between two codewords remains still unchanged, if it was even already. *ExtendingOddDistance* checks, if the distance between two codewords increases by one, if it was odd. Finally, *MinimumDistanceAfterExtension* checks for a general minimum distance of a code, if it will be increased by one, if it was odd. The properties of extending hold by the RISCAL checking process. We use *K=12* for the first two theorems and *K=9* for the last theorem:

```
Executing ExtendingEvenDistance(Array[ℤ],Array[ℤ]) with all
16777216 inputs.
Execution completed for ALL inputs (132999 ms, 8384512 checked,
8392704 inadmissible).

Executing ExtendingOddDistance(Array[ℤ],Array[ℤ]) with all
16777216 inputs.
Execution completed for ALL inputs (133537 ms, 8388608 checked,
8388608 inadmissible).

Executing MinimumDistanceAfterExtension(ℤ) with all 10 inputs.
Execution completed for ALL inputs (15283 ms, 1 checked,
9 inadmissible).
```

There are more ways, like **puncturing**, **expurgating by removing codewords**, and others. We only wanted to have a glimpse into this theory [10].

Chapter 4

Source Coding

In this chapter, we are interested in reducing the redundancy of data (**compression**) to save memory capacity or in general use an efficient way of representation. Characteristic values like entropy and redundancy introduced in Chapter 2 are important.

4.1 Run Length Encoding

The process of compressing data (or streams in general) has a flow similar to channel coding: original data, also denoted as **raw data**, are received by an encoder (or **compressor**), which compresses it. A decoder (or **decompressor**) then decodes the compressed data and restores the original data.

Compression methods can be characterized according to different properties [10]:

- **nonadaptive**: the compression algorithm stays the same for all input data.
- **adaptive**: the compression algorithm examines the input data and modifies its operations or parameters in response to the data.
- **lossless**: the decoder is able to **fully restore** the original input data without losing any information. This is important for source codes of programs.
- **lossy**: the decoder is **not able to fully restore** the original input data. This is fine, e.g. for compressing pictures.
- **cascaded**: data get compressed successively by two or more encoders.
- **perceptive**: the encoder compresses in a lossy way such that, e.g., human perception is still able to obtain good information (e.g., music).
- **symmetrical**: en- and decoder are using basically the same algorithm but in the other direction.
- **asymmetrical**: the encoder and decoder differ significantly; one may be substantially more complex than the other.

To assess methods, different indicators are used in practice [10]:

Definition 4.1.1. *The **compression ratio** $\mathrm{C_r}(\mathrm{in}, \mathrm{out})$ of an input stream in and output stream out is defined as $\mathrm{C_r}(\mathrm{in}, \mathrm{out}) = \frac{size(\mathrm{in})}{size(\mathrm{out})}$. It is also called **bpb (bits per bit/bits per pixel) or bitrate**.*

Definition 4.1.2. *The **compression factor** $C_f(in, out)$ is defined as $C_f(in, out) = \frac{size(in)}{size(out)}$. The compression factor is the inverse of the compression ratio.*

As input data, we often consider text, images or bit streams.

Definition 4.1.3. *Single elements of input data are called **data items** or **data symbols**.*

An easy introduction to source coding is the method of **Run Length Encoding (RLE)**. Here, sequences of the data are checked for consecutive appearances of data items. First, we consider text as input, e.g., the sequence *'a bbb c'* has data items *'a'* with 1 consecutive appearance, *'b'* with 3 and *'c'* with 1. The number of consecutive appearances can be described as follows:

Definition 4.1.4. *The number n of consecutive appearances of data items is denoted as the **run length**, **repetition count**, or **run**.*

Segments can be represented by tuples, e.g., *'bbb'* as $(3, b)$. Now, this sequence can be stored differently: segments of consecutive items in the sequence are represented by an **escape character** (this might be the symbol *'@'*) followed by the run length and the data item itself. The example sequence *'a bbb c'* could be stored as *'a @3b c'*. However, as this example shows, that RLE is not always useful, because we have the same amount of data items after compression. In fact, there are more problems with RLE. The problems and possible solutions for RLE are [10]:

- If run lengths are too short, the compressed data could be equally big as before or even bigger. Therefore, runs should only be encoded if their lengths are large enough to compensate for the additional encoding overhead.
- The escape character may itself occur in the input data. Therefore, an unambiguous escape mechanism or another representation has to be used.
- In the particular text-RLE scheme described by Salomon [10], the repetition count is represented by one byte and is therefore limited in size. A count of 0 represents three repetitions, allowing a stored count of 255 to represent a run of 258 identical characters.

Figure 4.1 shows possible methods of RLE text (de-)compression:

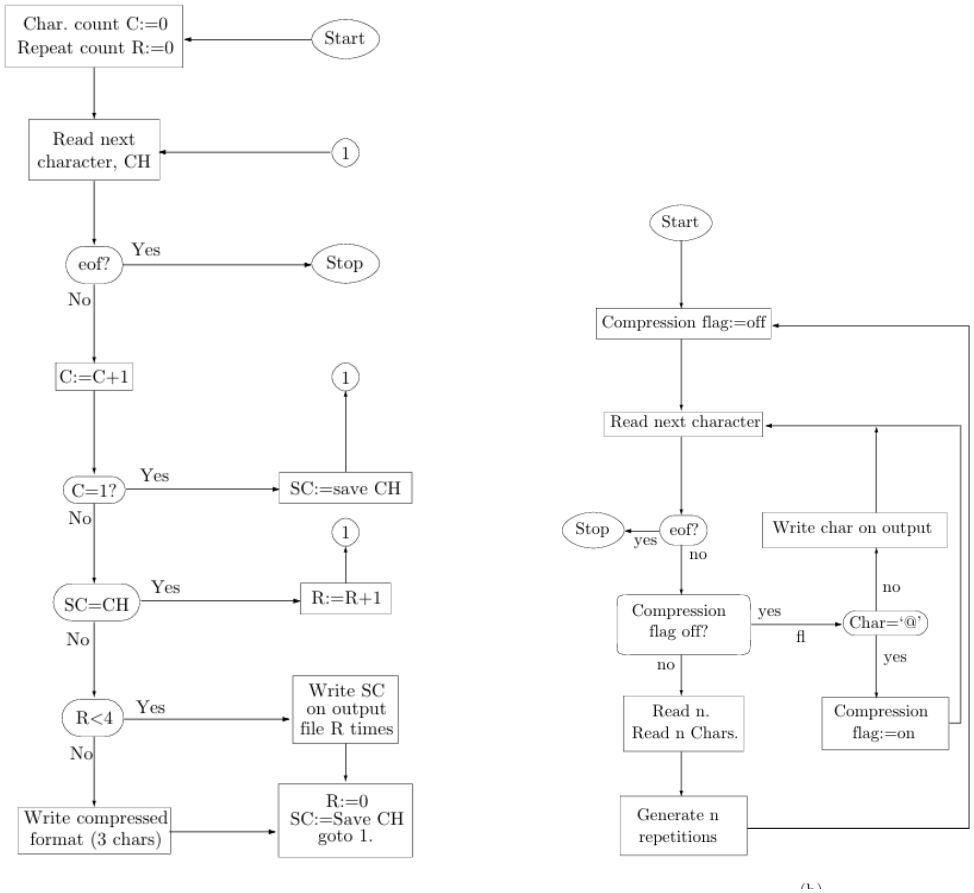

Figure 4.1: RLE encoding and decoding flow charts [10]

The left side shows the encoding process. Encoding needs two counters, C for counting the position and R for counting consecutive repetitions of a character. C is basically only needed in the beginning of the procedure. The first character is assigned to variable SC. Then the next character CH is read. If CH = SC, then the same character occurred directly again and the repetition counter is incremented. Then the next character is read. This loop is sustained until a different character than SC is read. When this happens, the repetition counter is checked. If R is big enough (4 in the figure), a compressed format of the character sequence is written on the output and R is reset and CH is assigned to SC. Otherwise, the character is written R-times on the output. This process remains until the end of file (EOF) is reached.

The right side explains the decoding process. It needs a compression flag, which indicates, when compressed data is currently the input. The flag is initially set to *false*. The first character is read. If the flag is *false*, the character is checked for the escape character (here '@'). If not, the character is just written and the next character is read; otherwise, the compression flag is set to *true* and the loop starts again by reading the next character, which is the run length n. Since the flag is activated, the next character is written n times. The compression flag is deactivated and the loop starts again. This process remains until EOF is reached.

We assume that the encoder writes **RLE-tokens** to the output. Each token

contains a compression flag, the run length, and the corresponding symbol. Symbols with run lengths of at least 3 are compressed. This is realized in RISCAL:

```
    val N:Nat;
    val A:Nat;

    type Symbol = Nat[A];
    type Index = Nat[N-1];
    type Text = Array[N,Symbol];
    type Count = Nat[N];
    type ElementCount = Nat[N];
    type Element = Record [compressed: Bool,count: Count,symbol: Symbol];
    type EncodedText = Record [length: ElementCount,
         elements: Array[N,Element]];
```

The constants *N* and *A* define the size of the model. *N* is the fixed length of the input text, while *A* determines the range of possible symbols. At first, the parameters are chosen as *N=4* and *A=2* for the first executions. The type *Symbol* therefore represents one symbol of the text, and *Index* represents a valid position in a text. The type *Text* is an array of *N* symbols and represents the input text. The type *Count* is used for run lengths, while *ElementCount* is used for the number of encoded elements. The type *Element* represents one RLE-token. Its field *compressed* indicates whether the token represents a compressed run, *count* stores the number of repetitions, and *symbol* stores the corresponding symbol. The type *EncodedText* then represents the complete encoded text. Its field *length* specifies how many elements are actually used, while *elements* stores the corresponding RLE-tokens.

```
    pred run(t:Text,start:Index,count:Count,s:Symbol) ⇔ start + count
         <= N ∧ ∀j:Count with j < count. t[start + j] = s;

    pred completeRun(t:Text,start:Index,count:Count,s:Symbol) ⇔
         run(t,start,count,s) ∧ (start = 0 ∨ t[start - 1] ≠ s) ∧
         (start + count = N ∨ t[start + count] ≠ s);

    pred runStartsAt(t:Text,start:Index,s:Symbol) ⇔ t[start] = s ∧
         (start = 0 ∨ t[start - 1] ≠ s);
```

The predicate *run* describes a sequence of consecutive occurrences of the same symbol. The parameter *start* specifies the first position of the sequence, *count* its length, and *s* the symbol occurring in the sequence. The condition *start + count <= N* ensures that the complete sequence lies within the text. The quantified condition then requires every position of the sequence to contain *s*. The predicate *completeRun* extends *run* by requiring that the sequence cannot be extended to the left or to the right. Thus, if *start* is not the first position, the preceding symbol must differ from *s*. Similarly, if the run does not end at position *N*, the following symbol must differ from *s*. Consequently, *completeRun* describes the complete maximal run of a symbol. The predicate *runStartsAt* checks whether a new run of symbol *s* starts at position *start*. The symbol at *start* must be *s*, while the preceding symbol, if one exists, must be different.

```
    pred validElement(e:Element) ⇔ (e.compressed = false ∧ e.count = 1)
```

```
          ∨ (e.compressed = true ∧ e.count >= 3);

     pred validEncoded(e:EncodedText) ⇔ ∀(i:Index with i < e.length.
          validElement(e.elements[i])) ∧ e.length > 0;

     pred decodedLengthBounded(e: EncodedText) ⇔
          (sum i: Index. if i < e.length then e.elements[i].count
          else 0) <= N;
```

The predicate *validElement* defines the two possible forms of an RLE-token. If *compressed* is false, the element represents exactly one symbol and therefore has *count = 1*. If *compressed* is true, the element represents a compressed run and therefore has *count >= 3*. This directly reflects the chosen RLE rule that runs of length one or two are stored as individual symbols, while runs of at least three symbols are compressed. The predicate *validEncoded* applies *validElement* to every element that is actually used in an *EncodedText*. The condition *e.length > 0* additionally ensures that an encoded text contains at least one element. The predicate *decodedLengthBounded* limits the total number of symbols represented by an *EncodedText* to at most *N*. This is necessary because the decoded representation has to fit into the fixed-size type *Text*.

```
     fun decodedLength(e:EncodedText):Nat[N]
     requires validEncoded(e) ∧ decodedLengthBounded(e); =
     sum i:Index. if i < e.length then e.elements[i].count else 0;

     fun elementStart(e:EncodedText,i:Index):Nat[N]
     requires validEncoded(e) ∧ decodedLengthBounded(e) ∧ i < e.length; =
     sum j:Index. if j < i then e.elements[j].count else 0;
```

The function *decodedLength* computes the number of symbols represented by an *EncodedText* after decoding. It sums the *count* of every used element. The function *elementStart* determines the first position in the decoded text that belongs to element *i*. This position is obtained by adding the counts of all preceding elements. For example, if the first element represents three symbols, the second element starts at decoded position 3.

```
     pred elementContains(e:EncodedText,i:Index,p:Nat[N]) ⇔
          validEncoded(e) ∧ decodedLengthBounded(e) ∧ i < e.length ∧
          elementStart(e,i) <= p ∧
          p < elementStart(e,i) + e.elements[i].count;

     pred encodingCoversText(t:Text,e:EncodedText) ⇔ validEncoded(e) ∧
          decodedLengthBounded(e) ∧ decodedLength(e) = N;

     pred elementEncodes(t:Text,e:EncodedText,i:Index) ⇔ i < e.length ∧
          ∀p:Index. elementContains(e,i,p) ⇒ t[p] = e.elements[i].symbol;

     pred encodes(t:Text,e:EncodedText) ⇔ encodingCoversText(t,e) ∧
          ∀i:Index with i < e.length. elementEncodes(t,e,i);
```

The predicate *elementContains* determines whether position *p* of the decoded text belongs to element *i*. The position must be greater than or equal to *elementStart(e,i)* and smaller than the end position obtained by

adding the element's *count*. The predicate *encodingCoversText* specifies that an encoded text represents a complete input text. In particular, *decodedLength(e) = N* requires the decoded representation to have exactly the same length as a *Text*. The predicate *elementEncodes* establishes the connection between an encoded element and the original text. For every position *p* contained in element *i*, the corresponding symbol in the input text must equal *e.elements[i].symbol*. Finally, *encodes* combines these conditions. It requires the encoded representation to cover the complete text and requires every encoded element to represent the correct symbols. Thus, *encodes(t,e)* expresses that *e* is a valid encoding of *t*.

```
    fun runLength(t:Text,start:Index,s:Symbol):Count
    requires runStartsAt(t,start,s);
    ensures completeRun(t,start,result,s);
    = choose c:Count with completeRun(t,start,c,s);
```

The function *runLength* returns the length of the complete run starting at *start*. The precondition *runStartsAt(t,start,s)* guarantees that a run of symbol *s* starts at this position. The expression *choose c* selects a value satisfying *completeRun*. Since *completeRun* describes the complete maximal run, the corresponding value of *c* is uniquely determined. This function satisfies its specification:

```
    Executing runLength(Array[ℤ],ℤ,ℤ) with all 972 inputs.
    Execution completed for ALL inputs (331 ms, 243 checked,
    729 inadmissible).
```

```
    proc encodeCompressedRun(count:Count,s:Symbol):Element
    requires count >= 3;
    ensures result.compressed = true ∧ result.count = count ∧
    result.symbol = s;
    {
        var result : Element = <<compressed: true, count: count,
            symbol: s>>;
        return result;
    }

    proc encodeSingleRun(t:Text,start:Index):EncodedText
    requires start < N;
    ensures result.length = 1 ∧ result.elements[0].compressed = false ∧
    result.elements[0].count = 1 ∧ result.elements[0].symbol = t[start];
    {
        var emptyElement:Element = <<compressed: false, count: 0,
            symbol: 0>>;
        var elements:Array[N,Element] = Array[N,Element](emptyElement);

        elements[0] := <<compressed: false, count: 1, symbol: t[start]>>;
        var result:EncodedText = <<length: 1, elements: elements>>;

        return result;
    }
```

```
    Executing encodeCompressedRun(ℤ,ℤ) with all 15 inputs.
    Execution completed for ALL inputs (18 ms, 6 checked,
```

```
    9 inadmissible).

    Executing encodeSingleRun(Array[ℤ],ℤ) with all 324 inputs.
    Execution completed for ALL inputs (291 ms, 324 checked,
    0 inadmissible).
```

The procedure *encodeCompressedRun* constructs a single compressed *Element*. Its precondition requires *count >= 3*, and its postcondition guarantees that the resulting element stores the given count and symbol with *compressed = true*. The procedure *encodeSingleRun* handles a run of length one. Since the run is not compressed, the procedure creates an element with *compressed = false*, *count = 1*, and the symbol at position *start*. This element is then stored as the only used element of the resulting *EncodedText*.

```
proc encodeDoubleRun(t:Text,start:Index): EncodedText
requires start + 1 < N ∧ t[start] = t[start + 1];
ensures result.length = 2 ∧ result.elements[0].compressed = false ∧
result.elements[0].count = 1 ∧ result.elements[0].symbol = t[start]
∧ result.elements[1].compressed = false ∧
result.elements[1].count = 1 ∧
result.elements[1].symbol = t[start + 1];
{
    var emptyElement: Element = <<compressed: false, count: 0,
        symbol: 0>>;
    var elements: Array[N, Element] = Array[N, Element]
        (emptyElement);

    elements[0] := <<compressed: false, count: 1, symbol: t[start]>>;
    elements[1] := <<compressed: false, count: 1,
        symbol: t[start + 1]>>;

    var result: EncodedText = <<length: 2, elements: elements>>;
    return result;
}

proc encodeCompressedRunText(count:Count,s:Symbol):EncodedText
requires count >= 3;
ensures result.length = 1 ∧ result.elements[0].compressed = true ∧
result.elements[0].count = count ∧ result.elements[0].symbol = s;
{
    var emptyElement:Element = <<compressed: false, count: 0,
        symbol: 0>>;
    var elements:Array[N,Element] = Array[N,Element](emptyElement);
        elements[0] := encodeCompressedRun(count,s);

    var result:EncodedText = <<length: 1, elements: elements>>;
    return result;
}
```

```
    Executing encodeDoubleRun(Array[ℤ],ℤ) with all 324 inputs.
    Execution completed for ALL inputs (257 ms, 81 checked,
    243 inadmissible).

    Executing encodeCompressedRunText(ℤ,ℤ) with all 15 inputs.
    Execution completed for ALL inputs (40 ms, 6 checked,
```

```
    9 inadmissible).
```

The procedure *encodeDoubleRun* handles a run of exactly two equal symbols. Its precondition requires the two positions to be within the text and requires both symbols to be equal. Since a run of length two is not compressed, *encodeDoubleRun* creates two elements, each with *compressed = false* and *count = 1*. The procedure *encodeCompressedRunText* converts a compressed *Element* into an *EncodedText*. It calls *encodeCompressedRun* and stores the resulting element at position 0. The resulting *EncodedText* therefore has exactly one used element.

```
proc encodeOneRun(t:Text,start:Index,count:Count):EncodedText
requires start < N ∧ count >= 1 ∧ start + count <= N ∧
(count = 1 ∨ count >= 3 ∨ t[start] = t[start + 1]);
{
    var result:EncodedText;

    if count = 1 then result := encodeSingleRun(t,start);
    else if count = 2 then result := encodeDoubleRun(t,start);
    else result := encodeCompressedRunText(count,t[start]);

    return result;
}
```

The procedure *encodeOneRun* is the central procedure for encoding a single run. For *count = 1*, it calls *encodeSingleRun*. For *count = 2*, it calls *encodeDoubleRun*. For *count >= 3*, it calls *encodeCompressedRunText*. Thus, *encodeOneRun* directly implements the rule that only runs of at least three equal symbols are compressed.

```
proc appendEncodedText(left: EncodedText,right: EncodedText):
     EncodedText
requires validEncoded(left) ∧ validEncoded(right) ∧
decodedLengthBounded(left) ∧ decodedLengthBounded(right) ∧
left.length + right.length <= N;
ensures result.length = left.length + right.length;
{
    var result: EncodedText = left;

    for var j: Index := 0; j < right.length; j := j + 1 do
    result.elements[result.length + j] := right.elements[j];

    result.length := left.length + right.length;
    return result;
}

proc appendRun(e:EncodedText,t:Text,start:Index,count:Count):
     EncodedText
requires validEncoded(e) ∧ decodedLengthBounded(e) ∧ start < N ∧
count >= 1 ∧ start + count <= N ∧
run(t,start,count,t[start]) ∧
e.length + encodeOneRun(t,start,count).length <= N;
ensures result.length = e.length +
encodeOneRun(t,start,count).length;
{
```

```
        var run:EncodedText = encodeOneRun(t,start,count);
        var result:EncodedText = appendEncodedText(e,run);

        return result;
    }
```

In the following, the model parameter *N* has been reduced to 3, to accelerate the RISCAL executions.

```
    Executing appendEncodedText(Record[length:ℤ,
    elements:Array[Record[compressed:Bool,count:ℤ,
    symbol:ℤ]]],Record[length:ℤ,elements:Array[Record[compressed:Bool,
    count:ℤ,symbol:ℤ]]]) with all 3057647616 inputs.
    PARALLEL execution with 12 threads (output disabled).
    Execution completed for ALL inputs (5460051 ms, 13436928 checked,
    3044210688 inadmissible).

    Executing appendRun(Record[length:ℤ,elements:Array[Record
    [compressed:Bool,count:ℤ,symbol:ℤ]]],
    Array[ℤ],ℤ,ℤ) with all 17915904 inputs.
    PARALLEL execution with 12 threads (output disabled).
    Execution completed for ALL inputs (38213 ms, 370656 checked,
    17545248 inadmissible).
```

The procedure *appendEncodedText* appends all used elements of *right* to *left*. The loop copies the elements one by one to the first unused positions of *left*. Afterwards, *result.length* is increased by *right.length*. The procedure *appendRun* combines the encoding of one run with the construction of the complete encoded text. It first calls *encodeOneRun* to obtain the encoded representation of the run beginning at *start*. It then calls *appendEncodedText* to append this representation to the already constructed *EncodedText e*.

```
    proc encodeFirstRun(t:Text,start:Index):EncodedText
    requires runStartsAt(t,start,t[start]);
    ensures result.length >= 1;
    {
        var count:Count = runLength(t,start,t[start]);
        var result:EncodedText = encodeOneRun(t,start,count);

        return result;
    }

    type Position = Nat[N];

    fun toIndex(p:Position):Index
    requires p < N;
    = p;

    pred encodedPrefix(t:Text,start:Position,e:EncodedText) ⇔
         validEncoded(e) ∧ decodedLengthBounded(e) ∧ start <= N ∧
         decodedLength(e) = start;
```

```
    Executing encodeFirstRun(Array[ℤ],ℤ) with all 324 inputs.
    Execution completed for ALL inputs (301 ms, 243 checked,
    81 inadmissible).
```

The procedure *encodeFirstRun* encodes the first run of the input text. It calls *runLength* to determine the length of this run and then passes the result to *encodeOneRun*. The type *Position* is defined separately from *Index* because the recursive encoder also needs to represent the position N, which is one position beyond the last valid array index. In contrast, *Index* only represents positions from 0 to N − 1. The function *toIndex* converts a valid *Position* into an *Index*. Its precondition *p < N* guarantees that the conversion is only performed for positions that are valid array indices. The predicate *encodedPrefix* describes the state of the recursive encoder. It requires that *e* is a valid encoded text and that its decoded length is exactly *start*. Therefore, *e* represents the already processed prefix of the input text.

```
multiple proc encodeFrom(t:Text,start:Position,e:EncodedText):
EncodedText
requires encodedPrefix(t,start,e) ∧ (start = N ∨
runStartsAt(t,start,t[start]));
ensures result.length >= e.length;
{
    var result:EncodedText = e;

    if start < N then
    {
        var index:Index = toIndex(start);
        var count:Count = runLength(t,index,t[index]);

        result := appendRun(result,t,index,count);
        if start + count < N then result := encodeFrom(t,
            start + count,result);
    }
    return result;
}

proc encode(t:Text):EncodedText
requires N > 0;
ensures encodes(t,result);
{
    var result:EncodedText = encodeFirstRun(t,0);
    var count:Count = runLength(t,0,t[0]);

    if count < N then result := encodeFrom(t,count,result);

    return result;
}
```

```
Executing encodeFrom(Array[ℤ],ℤ,Record[length:ℤ,elements:Array
[Record[compressed:Bool,count:ℤ,symbol:ℤ]]]) with all
5971968 inputs.
Execution completed for ALL inputs (14584 ms, 82377 checked,
5889591 inadmissible).

Executing encode(Array[ℤ]) with all 27 inputs.
Execution completed for ALL inputs (71 ms, 27 checked,
0 inadmissible).
```

The recursive procedure *encodeFrom* continues the encoding at position *start*.

The predicate *encodedPrefix* guarantees that the current *EncodedText e* already represents the prefix ending at this position. If *start < N*, *encodeFrom* converts the position to an *Index*, determines the length of the next run using *runLength*, and adds the encoded run using *appendRun*. If further symbols remain, *encodeFrom* calls itself recursively with the position *start + count*. The procedure *encode* is the entry point of the encoder. It starts with *encodeFirstRun* at position 0. It then determines the length of the first run with *runLength*. If this run does not already cover the complete text, *encode* calls *encodeFrom* for the remaining part of the text. Most importantly, the postcondition *encodes(t,result)* explicitly states that the result produced by *encode* is an encoding of the input text. This provides the formal connection between the input *Text* and the resulting *EncodedText*.

```
    pred decodedSymbol(e:EncodedText,i:Index,p:Nat[N],s:Symbol) ⇔
         elementContains(e,i,p) ∧ s = e.elements[i].symbol;

    pred elementAtPosition(e:EncodedText,i:Index,p:Nat[N]) ⇔
         elementContains(e,i,p);

    pred decodedAt(e:EncodedText,p:Nat[N],s:Symbol) ⇔ ∃i:Index.
         elementAtPosition(e,i,p) ∧ decodedSymbol(e,i,p,s);
```

The predicate *decodedSymbol* states that symbol *s* is the symbol represented at position *p* by element *i*. This is the case when *elementContains* establishes that position *p* belongs to element *i* and *s* is equal to the symbol stored in that element. The predicate *elementAtPosition* directly expresses that element *i* contains position *p*. It is defined using *elementContains* so that the positional relationship can be used explicitly when defining *decodedAt*. The predicate *decodedAt* combines both relationships. It states that there exists an encoded element containing position *p* whose stored symbol is *s*. Thus, *decodedAt* specifies the result of decoding one particular position.

```
    proc decodeAt(e: EncodedText,p: Nat[N]): Symbol
    requires validEncoded(e) ∧ decodedLengthBounded(e) ∧
    p < decodedLength(e);
    ensures decodedAt(e,p,result);
    {
        var result: Symbol;

        choose result: Symbol with decodedAt(e,p,result);

        return result;
    }

    proc decode(e:EncodedText):Text
    requires validEncoded(e) ∧ decodedLengthBounded(e) ∧
    decodedLength(e) = N;
    ensures ∀p:Index. decodeAt(e,p) = result[p];
    {
        var result:Text = Array[N,Symbol](0);
        for var p:Position := 0; p < N; p := p + 1 do
        result[toIndex(p)] := decodeAt(e,toIndex(p));

        return result;
    }
```

```
    Executing decodeAt(Record[length:ℤ,elements:Array[Record
    [compressed:Bool,count:ℤ,symbol:ℤ]]],ℤ) with all 221184 inputs.
    Execution completed for ALL inputs (2178 ms, 7425 checked,
    213759 inadmissible).

    Executing decode(Record[length:ℤ,elements:Array[Record
    [compressed:Bool,count:ℤ,symbol:ℤ]]]) with all 55296 inputs.
    Execution completed for ALL inputs (2362 ms, 1755 checked,
    53541 inadmissible).
```

The procedure *decodeAt* decodes one position of an *EncodedText*. Its precondition guarantees that the encoded text is valid and that *p* is a valid position in its decoded representation. The expression *choose result* selects the symbol satisfying *decodedAt(e,p,result)*. Since each decoded position belongs to exactly one encoded element, the corresponding symbol is uniquely determined. The procedure *decode* reconstructs the complete *Text*. It initializes an array of length *N* and iterates over all positions using *p*. For each position, *decodeAt* determines the corresponding symbol, which is then stored in *result*. The postcondition of *decode* states that for every valid position *p*, the value returned by *decodeAt(e,p)* is equal to *result[p]*. Thus, every position of the reconstructed text is defined by the decoding specification.

```
    theorem decodeEncode(t:Text) ⇔ decode(encode(t)) = t;
```

The theorem *decodeEncode* states that encoding and subsequently decoding any input text reproduces the original text exactly. It therefore establishes the correctness of the complete RLE model by connecting the explicitly defined procedures *encode* and *decode*. RISCAL checks for all possible texts this property for *N=7* and *A=3*:

```
    Executing decodeEncode(Array[ℤ]) with all 16384 inputs.
    Execution completed for ALL inputs (123919 ms, 16384 checked,
    0 inadmissible)
```

There are more variants of run length encoding, like **relative encoding** or special techniques for compressing images with RLE [10].

4.2 Variable-Size Codes, Prefix Codes

We now investigate methods, which produce bitpatterns of different sizes for symbols.

Definition 4.2.1. *A **code** is a mapping from data items to other representations or values. These representations or values are called **codewords** or sometimes simply **codes**.*

Considering data inputs where the symbols do not appear with equal probability, an effective way to compress the input is to map symbols which occur more frequently to shorter (smaller) codes. The following example shows

the effectiveness [10]: Consider symbols a_1, a_2, a_3 and a_4 with $p_1(a_1) = 0.49$, $p_2(a_2) = 0.25$, $p_3(a_3) = 0.25$ and $p_4(a_4) = 0.01$. The entropy is $H(p_1, p_2, p_3, p_4) \approx 1.57$ bits per symbol, indicating the theoretical lower bound on the average number of bits required to represent a symbol. Using $a_1 \rightarrow 00$, $a_2 \rightarrow 01$, $a_3 \rightarrow 10$, $a_4 \rightarrow 11$ as code yields a redundancy $R = \log_2(4) - 1.57 = 0.43$. To reduce the redundancy, variable-size codes can be used by assigning shorter codes to more frequent symbols, e.g., $a_1 \rightarrow 1$, $a_2 \rightarrow 01$, $a_3 \rightarrow 010$, $a_4 \rightarrow 001$. Since the codewords now have different lengths, the average codeword length must be used to calculate the redundancy: $R = 1 \cdot 0.49 + 2 \cdot 0.25 + 3 \cdot 0.25 + 3 \cdot 0.01 - 1.57 = 0.2$. Thus, the redundancy has been reduced from 0.43 to 0.20 bits per symbol. These codes are called **variable-size codes**.

Definition 4.2.2. ***Variable-size codes*** *are codes, which allow codes with different lengths.*

Variable-size codes can also bear problems in decompression. The decoder only receives a sequence of binary values. Hence, it may be unclear which segments correspond to individual codewords, e.g., the sequence *0101* could be understood as *01, 01* or *010, 1*. This occurs when one codeword is also the prefix of another codeword. Here, the mapping of a_2 is also the beginning of the mapping of a_3. Such codes are called **ambiguous**. If the variable-size code is defined as $a_1 \rightarrow 1$, $a_2 \rightarrow 01$, $a_3 \rightarrow 000$, $a_4 \rightarrow 001$ the decoding problem is avoided. This code now satisfies the so called **prefix property** and can therefore be **unambiguously** decoded [10].

Definition 4.2.3. *A code satisfies the* ***prefix property****, if the beginnings of codes do not consist of other codes.*

However, using variable-size codes does not necessarily result in optimal compression. Algorithms such as Huffman coding can be used to construct efficient prefix codes.

Definition 4.2.4. ***Prefix codes*** *are variable-size codes which satisfy the prefix property.*

The binary representation does not satisfy the prefix property. For this representation it is also a problem that the number of bits and therefore the code size can only be determined if the size of integers is known in advance [10].

A useful form of representation is the unary representation, which defines **unary codes** [10].

Definition 4.2.5. ***Unary codes*** *represent a positive integer n by $(n-1)$ repetitions of one bit, terminated by the other bit. For example, one possible convention is to represent n by $(n-1)$ ones followed by a 0. The opposite convention, using $(n-1)$ zeros followed by a 1, is also possible.*

Theorem 4.2.1. *Unary codes are prefix codes, since the termination bit marks the end of each codeword. Therefore, no unary codeword can be a prefix of another codeword.*

Unary coding maps each integer to a bit sequence whose length increases linearly with the integer. Therefore, large numbers are represented by long codewords, which can be inefficient.

We are now interested in finding efficient prefix codes.

4.3 Golomb Code

Now, the derivation of the so-called **Golomb Code** is explained:

We consider input data whose values are not uniformly distributed, for example, binary data processed using RLE. More specifically, we consider binary sequences in which 0s occur more frequently than 1s. We can therefore consider the run lengths of consecutive 0s. Let p denote the probability of occurrence of the bit 0. Then, the probability of observing a run of n consecutive 0s terminated by a 1 is $P(N = n) = p^n(1 - p)$. This corresponds to a *geometric distribution* where shorter runs are more likely than longer ones. Therefore, the code should follow the principle that more likely values should be assigned shorter codewords, whereas less likely values should be assigned longer codewords [10].

We therefore seek a code that is lossless, uniquely decodable, and adaptable to different probability distributions through a parameter. Intuitively, one could assign a different codeword length to each possible value n according to its probability. However, this would require constructing a separate code for each specific probability distribution. Instead, we construct a **parameterized prefix code**: a family of codes whose structure is determined by a parameter m (**Golomb parameter**). This family forms the **Golomb code**. The parameter m can be understood as defining blocks of consecutive integers: $[0, ..., m - 1], [m, ..., 2m - 1], [2m, ..., 3m - 1], ...$. For a given value n, m determines the size of the blocks and therefore the block in which n is located. The choice of m affects the codeword lengths: small values of m lead to short initial codewords whose lengths increase rapidly, whereas large values of m lead to longer initial codewords whose lengths increase more slowly. m can be adjusted regarding the statistical properties of the input data [10].

For RLE, the choice of m can be related to the probability p of the most frequent symbol. If p is large, long sequences of consecutive 0s are more likely, resulting in larger run lengths. Therefore, a larger value of m is appropriate. For small p, the runs tend to be shorter, so smaller values of m are sufficient. For small p, the runs are shorter and smaller m are sufficient. For geometrically distributed values, there exists a value of m for which the average codeword length is minimized. Thus, m should not be chosen arbitrarily, but should reflect the statistical properties of the input data. For RLE, an intuitive choice is to relate m to the distribution of the run lengths. In particular, the median can be used as a practical choice for m: approximately half of the runs are shorter than m, while the other half are at least m long [10].

We can now construct the Golomb code. How can an arbitrary non-negative integer n be encoded? We perform Euclidean division by m, which decomposes n into two values: Hence, $n = q \cdot m + r$, $q = \lfloor \frac{n}{m} \rfloor$, $r = n - q \cdot m$ with $0 \leqslant r < m$. Thus, n can be represented by the pair (q, r). The quotient q indicates the m-block in

which n is located, while the remainder r indicates the position within this block [10].

The quotient q is encoded using unary coding. Hence, the length of this part of the codeword increases linearly with q. This is appropriate for geometrically distributed values, since larger values of n are less likely and generally result in larger quotients. The unary representation is uniquely decodable because its termination bit marks the end of the quotient [10].

The remainder can only assume m possible values: $0, 1, ..., m-1$. We need a binary representation for m different values. With c bits it is possible to represent 2^c different values. Thus, we require $2^c \geqslant m$. The smallest number c satisfying this condition is $c = \lceil \log_2(m) \rceil$. There is a case distinction [10]:

If m is a power of 2, $m = 2^k$, the remainder can be represented with exactly k bits, which marks the special case of Golomb codes [10].

If m is not a power of 2, the m possible remainder values are encoded using a shortened binary representation called **truncated binary coding**. This avoids unused bit patterns that would occur if all values were represented using c bits. Some values are therefore encoded using $c-1$ bits, while the remaining values are encoded using c bits. The number of values using $c-1$ bits is determined by the threshold $t = 2^c - m$ [10]. If the remainder r is smaller than the threshold, i.e. $r < t$, it is encoded directly using $c-1$ bits. If $r \geqslant t$, the remainder is shifted by adding the threshold, resulting in $r' = r + t$. This shifted value r' is then encoded using c bits. Thus, the threshold determines both which values use the shorter representation and by how much the remaining values are shifted before their binary representation is generated. For example, for $m = 5$, we have $c = 3$ and $t = 3$. Therefore, the values 0, 1, and 2 are encoded using $c - 1 = 2$ bits as 00, 01, and 10. The remaining values 3 and 4 are shifted by the threshold, resulting in $3 + 3 = 6$ and $4 + 3 = 7$, which are represented using 3 bits as 110 and 111. This avoids the unused bit patterns that would occur if all values were encoded using three bits.

The encoding of n consists of 5 steps:

1. Choose m based on the statistical properties of the input data, for example, based on the probability parameter p or, intuitively for RLE, on the median of the run lengths.
2. Calculate $q = \lfloor \frac{n}{m} \rfloor$, $r = n - q \cdot m$ with $0 \leqslant r < m$ and $c = \lceil \log_2(m) \rceil$.
3. Use unary coding for q.
4. Use truncated binary coding for r.
5. Concatenate the unary code of q with the truncated binary code of r to obtain the final codeword.

The decoder must know the value of m used by the encoder.

Decoding consists of:

1. Read the unary code until its termination bit to identify the quotient q.
2. Calculate $c = \lceil \log_2(m) \rceil$. Together with m, this determines the truncated binary representation used for the remainder r.
3. Read the truncated binary code according to the rule determined by m to identify the remainder r.

4. Reconstruct n via $n = q \cdot m + r$.

The codeword can therefore be decoded unambiguously: the unary termination bit identifies the end of the quotient part, while the value of m determines how the following truncated binary bits are interpreted as the remainder. Finally, the original integer is reconstructed as $n = qm + r$.

This can be implemented in RISCAL:

```
    val N: Nat;
    val M: Nat;
    val K: Nat;

    type Number = Nat[N];
    type Parameter = Nat[M];
    type Auxiliary = Nat[K];
```

N specifies the maximum value of an input number, *M* the maximum value of the Golomb parameter, and *K* the maximum size used for bit sequences and auxiliary values. Based on these bounds, three different types are introduced. *Number* represents the numbers that are encoded, *Parameter* represents the possible Golomb parameters, and *Auxiliary* is used for internal values such as bit positions, lengths, offsets, and intermediate values. The values used for the following executions are *N=61*, *M=6*, and *K=62*.

```
    pred validParameter(m: Parameter) ⇔ m > 0;
    fun quotient(n: Number, m: Parameter): Nat[N]
    requires validParameter(m); = n / m;
    fun remainder(n: Number, m: Parameter): Nat[N]
    requires validParameter(m); = n - quotient(n,m) * m;

    theorem divisionCorrect(n: Number, m: Parameter)
    requires validParameter(m); ⇔ n = quotient(n,m) * m
        + remainder(n,m);
    theorem remainderCorrect(n: Number, m: Parameter)
    requires validParameter(m); ⇔ remainder(n,m) < m;

```

```
    Executing divisionCorrect(ℤ,ℤ) with all 434 inputs.
    Execution completed for ALL inputs (39 ms, 372 checked,
    62 inadmissible).

    Executing remainderCorrect(ℤ,ℤ) with all 434 inputs.
    Execution completed for ALL inputs (40 ms, 372 checked,
    62 inadmissible).
```

Since the Golomb parameter must be positive, *validParameter* is introduced to restrict the parameter to valid values. The functions *quotient* and *remainder* implement the Euclidean division of an input number *n* by the Golomb parameter *m*. They calculate quotient and remainder as already explained. Thus, every input number is represented as $n = qm+r$ with $0 \leqslant r < m$. The theorems *divisionCorrect* and *remainderCorrect* formally verify these two properties.

```
    pred validAuxiliary() ⇔ M + 1 <= K;

```

```
    fun halfCeil(x: Parameter): Auxiliary
    requires validParameter(x);
    requires validAuxiliary();
    = (x + 1) / 2;

    fun bitLength(m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    = if m = 1 then 0
    else 1 + bitLength(halfCeil(m));

    fun threshold(m: Parameter): Nat[N*N]
    requires validParameter(m);
    requires validAuxiliary();
    = 2 ^ bitLength(m) - m;

    pred powerOfTwo(m: Parameter)
    requires validParameter(m);
    requires validAuxiliary();
    ⇔ 2 ^ bitLength(m) = m;
```

The type *Auxiliary* is used for internal values such as bit positions and lengths. Therefore, *validAuxiliary* requires *M+1≤K* so that the required auxiliary values can be represented. The function *halfCeil* calculates $\lceil \frac{x}{2} \rceil$ using integer division. It is used by *bitLength* to recursively determine the number of bits required for a value of the Golomb parameter. For a parameter *m*, *bitLength* calculates $c = \lceil \log_2(m) \rceil$. The special case *m=1* terminates the recursion, since no bits are required to represent the only possible remainder $r = 0$. For truncated binary coding, the threshold *t*=$2^c - m$ is required. It separates the remainder values into those represented using $c-1$ bits and those represented using c bits. This value is calculated by *threshold*. Finally, *powerOfTwo* checks whether *m* is a power of two. In this case, $2^c = m$, so all possible remainder values can be represented using exactly c bits and no truncated binary transformation is necessary.

```
    fun remainderLength(r: Auxiliary, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m; =
    if powerOfTwo(m) then bitLength(m)
    else if r < threshold(m) then bitLength(m) - 1
    else bitLength(m);

    fun shiftedRemainder(r: Auxiliary, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m;
    requires r >= threshold(m);
    = r + threshold(m);
```

The function *remainderLength* determines the number of bits required to encode a remainder *r*. If *m* is a power of two, all remainders have the same length c. Otherwise, the threshold $t = 2^c - m$ determines the length. For *r<t*, the remainder is encoded using $c - 1$ bits. For *r≥t*, it is encoded using c bits.

Thus, *remainderLength* determines the exact length of the truncated binary part before the bits are generated. For the remainders that require c bits, the value itself cannot simply be encoded directly. Instead, truncated binary coding shifts these values by the threshold. The function *shiftedRemainder* therefore calculates $r' = r + t$. This shifted value can then be represented using c bits.

```
type Bit = Nat[1];
type BitSequence = Array[K, Bit];
type Codeword = Record[bits: BitSequence,length: Auxiliary];
```

A single bit is represented by the type *Bit*, which is restricted to the values 0 and 1. A *BitSequence* is then defined as an array containing K bits.

The actual Golomb codeword, however, usually contains fewer than K bits. Therefore, storing only the bit array would not be sufficient to determine which positions belong to the encoded value. The type *Codeword* is consequently defined as a record containing both the complete bit array and its actual *length*.

The first *length* positions of *bits* represent the actual codeword. The remaining positions only contain the initial values of the array and are not part of the encoded value. This representation allows all codewords to use the same fixed-size array while still storing their individual lengths.

```
fun binaryBit(n: Auxiliary): Bit = n % 2;
proc binaryStep(n: Auxiliary, i: Auxiliary, offset: Auxiliary,
     a: BitSequence): BitSequence
requires offset + i < K;
{
    return a with [offset + i] = binaryBit(n);
}
```

The binary representation is constructed using the functions *binaryBit* and *binaryStep*. The function *binaryBit* returns the least significant bit of a number by calculating its remainder modulo 2. Therefore, it returns 0 for even numbers and 1 for odd numbers. The procedure *binaryStep* writes this bit to a specified position in the *BitSequence*. The parameter i specifies the relative position within the binary representation, while *offset* specifies where this representation starts in the complete bit sequence. The use of an offset is important for the Golomb code because the binary representation of the remainder is not necessarily located at the beginning of the bit sequence. It follows the unary representation of the quotient. Therefore, the same binary procedure can be used for both the truncated binary value and other binary representations at arbitrary positions.

```
proc binary(n: Auxiliary, l: Auxiliary, offset: Auxiliary,
     a: BitSequence): BitSequence
requires l <= K;
requires n < 2 ^ l;
requires offset + l <= K;
{
    var currentN: Auxiliary := n;
    var result: BitSequence := a;
    var i: Auxiliary := l;

    while i > 0 do
```

```
        {
            result := binaryStep(currentN,i - 1,offset,result);
            currentN := currentN / 2;
            i := i - 1;
        }
        return result;
    }
```

The procedure *binary* constructs the binary representation of a value using exactly l bits. The parameter *offset* determines the position at which these bits are stored in the *BitSequence*. The precondition $n < 2^l$ ensures that n can be represented using at most l bits. The additional condition $offset + l \leqslant K$ ensures that all positions used for the representation are within the available *BitSequence*. The binary representation is constructed by repeatedly extracting the least significant bit using *binaryBit*. After processing a bit, the current value is divided by 2, which removes the least significant bit. Since the bits are obtained from least significant to most significant, they are written to the positions from right to left. This results in the usual left-to-right binary representation in the final sequence.

```
    fun truncatedValue(r: Auxiliary, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m; =
    if powerOfTwo(m) then r
    else if r < threshold(m) then r
    else shiftedRemainder(r,m);
```

The function *truncatedValue* determines the numerical value that is passed to *binary* when encoding a remainder. The result depends on the truncated binary representation defined by the Golomb parameter m. If m is a power of two, the remainder can be represented directly using *bitLength(m)* bits. The same applies to a non-power-of-two parameter if the remainder is smaller than the threshold t. In both cases, the original remainder r is therefore returned unchanged. For the remaining remainders, where $r \geqslant t$, the value has to be shifted. In this case, *shiftedRemainder* calculates *r'=r+t*. The resulting value is then represented in binary.

```
    proc truncatedBinary(r: Auxiliary, m: Parameter, offset: Auxiliary,
         a: BitSequence): BitSequence
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m;
    requires offset + remainderLength(r,m) <= K;
    {
        return binary(truncatedValue(r,m),remainderLength(r,m),offset,a);
    }
```

The procedure *truncatedBinary* combines the previously defined functions for truncated binary encoding. First, *truncatedValue* determines which numerical value has to be encoded. Then, *remainderLength* determines the number of bits required for this value. Finally, *binary* writes the value using exactly this number of bits at the specified *offset*. The procedure

therefore connects the mathematical truncated binary definition with the actual construction of the bit sequence.

```
    fun unaryBit(q: Number, i: Auxiliary): Bit
    requires i <= q; =
    if i < q then 1 else 0;
```

For the quotient part of the Golomb code, unary coding is used. A quotient q is represented by q one-bits followed by a zero The function *unaryBit* determines the bit that has to be written at position i. As long as $i<q$, the function returns 1. At position $i=q$, it returns 0, which terminates the unary representation.

```
    proc unaryStep(q: Number, i: Auxiliary, a: BitSequence): BitSequence
    requires i <= q;
    requires i < K;
    {
        return a with [i] = unaryBit(q,i);
    }
```

The procedure *unaryStep* writes the bit calculated by *unaryBit* to position i of the *BitSequence*. It separates the calculation of an individual unary bit from the construction of the complete unary representation.

```
    proc unary(q: Number, a: BitSequence): BitSequence
    requires q < K;
    {
        var result: BitSequence := a;
        var i: Auxiliary := 0;

        while i <= q do
        {
            result := unaryStep(q, i, result);
            i := i + 1;
        }
        return result;
    }
```

The procedure *unary* constructs the complete unary representation of the quotient. It starts at position 0 and repeatedly calls *unaryStep* until position q is reached. Thus, positions 0 through $q-1$ contain 1, while position q contains the terminating 0. The precondition $q<K$ guarantees that the last required position is inside the *BitSequence*.

```
    fun unaryLength(q: Number): Auxiliary
    requires q < K;
    = q + 1;

    fun golombLength(n: Number, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    = unaryLength(quotient(n,m)) + remainderLength(remainder(n,m),m);
```

The function *unaryLength* calculates the length of the unary representation. Since a quotient q is represented by q one-bits followed by one zero, its length is

q+1 . The function *golombLength* then calculates the total length of the Golomb code. A Golomb code consists of the unary representation of the quotient followed by the truncated binary representation of the remainder. Therefore,

L = *unaryLength*(q) + *remainderLength*(r,m) . This length is required when constructing the final *Codeword* and is also used to ensure that the complete codeword fits into the available K positions.

```
proc golomb(n: Number, m: Parameter): Codeword
requires validParameter(m);
requires validAuxiliary();
requires golombLength(n,m) <= K;
{
    var bits: BitSequence := Array[K,Bit](0);
    bits := unary(quotient(n,m), bits);
    bits := truncatedBinary(remainder(n,m), m,
        unaryLength(quotient(n,m)), bits);
    var codeword: Codeword = <<bits: bits,
        length: golombLength(n,m)>>;
    return codeword;
}
```

The procedure *golomb* combines all previously defined components to construct a complete Golomb codeword. First, a *BitSequence* of size *K* is initialized with zeros. The quotient $q = \lfloor \frac{n}{m} \rfloor$ is then encoded using *unary* and written starting at position 0. Next, the remainder $r = n - qm$ is encoded using *truncatedBinary*. Its *offset* is set to *unaryLength(q)*. Consequently, the truncated binary representation starts immediately after the unary representation. Finally, the resulting bit sequence and the actual length calculated by *golombLength* are stored in a *Codeword*.

```
fun decodeUnaryRec(bits: BitSequence, length: Auxiliary,
i: Auxiliary, count: Auxiliary): Auxiliary
requires i < length;
requires count <= i; =
if bits[i] = 0 then count
else if i + 1 = length then count + 1
else decodeUnaryRec(bits,length,i + 1,count + 1);

fun decodeUnary(bits: BitSequence, length: Auxiliary): Auxiliary
requires length > 0;
= decodeUnaryRec(bits,length,0,0);
```

The unary part is decoded using *decodeUnary*. The actual traversal of the bit sequence is implemented recursively by *decodeUnaryRec*. The parameter *i* represents the current position in the sequence, while *count* stores the number of one-bits that have been encountered so far. If the current bit is 0, the unary representation is complete and *count* is returned. If the current bit is 1, the function moves to the next position and increments *count*. The additional case for the last available bit being 1 allows the function to terminate even if no terminating zero occurs within the given codeword length. The function *decodeUnary* serves as the interface for this recursive implementation and starts the recursion at position 0 with a count of 0.

```
fun truncatedShort(x: Auxiliary,m: Parameter): Bool
```

```
    requires validParameter(m);
    requires validAuxiliary();
    requires x < 2 ^ bitLength(m);
    = x < threshold(m);
```

The function *truncatedShort* determines whether the initial $c - 1$ bits of a truncated binary representation form one of the short values. A value is a short representation exactly if *x<t,* where t is the threshold calculated by *threshold*.

```
    fun truncatedDecodedValue(x: Auxiliary,b: Bit,m: Parameter):
    Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires x < 2 ^ bitLength(m);
    requires x >= threshold(m);
    = 2 * x + b - threshold(m);
```

If the initial $c - 1$ bits do not represent a short value, one additional bit has to be read. The function *truncatedDecodedValue* reconstructs the original remainder using *r=2x+b-t,* where *x* is the value represented by the first $c - 1$ bits, *b* is the additional bit, and t is the threshold.

```
    fun decodeBits(bits: BitSequence,offset: Auxiliary,
    length: Auxiliary): Auxiliary
    requires length <= 3;
    requires offset + length <= K; =
    if length = 0 then 0
    else if length = 1 then bits[offset]
    else if length = 2 then bits[offset] * 2 + bits[offset + 1]
    else bits[offset] * 4 + bits[offset + 1] * 2 + bits[offset + 2];
```

The function *decodeBits* interprets a sequence of bits as a binary number. The cases for lengths 0, 1, 2, and 3 are explicitly defined. For example, a sequence of three bits is interpreted as $b_0 \cdot 2^2 + b_1 \cdot 2^1 + b_2$. The current implementation restricts the length to at most three bits. Therefore, *decodeBits* is sufficient for the parameter range used in the current implementation, but does not represent a fully general binary decoder for arbitrary values of *K*.

```
    fun decodeTruncatedBinary(bits: BitSequence,m: Parameter,
    offset: Auxiliary): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires offset + bitLength(m) <= K; =
    if powerOfTwo(m) then decodeBits(bits,offset,bitLength(m))
    else if truncatedShort(decodeBits(bits,offset,bitLength(m) - 1),m)
    then decodeBits(bits,offset,bitLength(m) - 1)
    else truncatedDecodedValue(decodeBits(bits,offset,
    bitLength(m) - 1), bits[offset + bitLength(m) - 1],m);
```

The function *decodeTruncatedBinary* combines the individual functions required to decode the truncated binary remainder. If *m* is a power of two, the remainder is represented using exactly *c = bitLength(m)* bits, so these bits are decoded directly. For a non-power-of-two parameter, the first $c - 1$ bits

are decoded first. The function *truncatedShort* then determines whether this value is below the threshold. If it is, these $c-1$ bits already represent the original remainder. Otherwise, one additional bit is read and *truncatedDecodedValue* is used to reconstruct the original remainder.

```
    pred validCodeword(codeword: Codeword) ⇔ codeword.length > 0 ∧
         decodeUnary(codeword.bits,codeword.length) <= N;
```

The predicate *validCodeword* defines basic conditions under which a codeword can be passed to the decoder. Its length must be greater than zero because *decodeUnary* requires at least one bit. Furthermore, the decoded quotient must be within the range of the type *Number*.

```
    fun decode(codeword: Codeword,m: Parameter): Number
    requires validParameter(m);
    requires validAuxiliary();
    requires validCodeword(codeword);
    requires decodeUnary(codeword.bits, codeword.length) * m
    +decodeTruncatedBinary(codeword.bits,m,
    unaryLength(decodeUnary(codeword.bits,
    codeword.length))) <= N;
    = decodeUnary(codeword.bits,codeword.length) * m
    + decodeTruncatedBinary(codeword.bits,m,
    unaryLength(decodeUnary(codeword.bits,codeword.length)));
```

The function *decode* reconstructs the original number from a valid *Codeword*. First, *decodeUnary* determines the quotient q. The position of the truncated binary part is then obtained using *unaryLength(q)*. The corresponding remainder r is decoded using *decodeTruncatedBinary*. Finally, the original number is reconstructed according to the division equation $n = qm + r$. The additional precondition ensures that the reconstructed value is within the range of *Number*, so that the result can be returned as a value of type *Number*.

```
    theorem decodeEncode(n: Number, m: Parameter)
    requires validParameter(m);
    requires validAuxiliary();
    requires golombLength(n,m) <= K;
    ⇔ decode(golomb(n,m),m) = n;
```

The formal verification is expressed by the theorem *decodeEncode*. It states that, for every valid input whose codeword fits into the available *K* positions, *decode* (*golomb(n,m),m)=n*. Thus, *decodeEncode* formally verifies that encoding and subsequent decoding returns the original input number. RISCAL also verifies the correctness:

```
    Executing decodeEncode(ℤ,ℤ) with all 434 inputs.
    Execution completed for ALL inputs (99 ms, 372 checked,
    62 inadmissible).
```

```
    pred isPrefix(a: Codeword,b: Codeword) ⇔ a.length <= b.length  ∧
         ∀i:Nat[K-1]. i < a.length ⇒  a.bits[i] = b.bits[i];
    pred prefixFree(a: Codeword,b: Codeword) ⇔ ¬isPrefix(a,b) ∧
         ¬isPrefix(b,a);

    theorem golombPrefixFree(n1: Number,n2: Number,m: Parameter)
```

```
    requires validParameter(m);
    requires validAuxiliary();
    requires n1 ≠ n2;
    requires golombLength(n1,m) <= K;
    requires golombLength(n2,m) <= K;
    ⇔ prefixFree(golomb(n1,m),golomb(n2,m));
```

```
    Executing golombPrefixFree(ℤ,ℤ,ℤ) with all 26908 inputs.
    Execution completed for ALL inputs (4195 ms, 22692 checked,
    4216 inadmissible).
```

The predicate *isPrefix* defines whether one codeword is a prefix of another codeword. First, the length of *a* must be less than or equal to the length of *b*. This ensures that a longer codeword cannot be a prefix of a shorter one. The second condition compares the bits of the two codewords. For every position *i* smaller than the length of *a*, the corresponding bits of both codewords must be equal. Since only the first *length* positions represent the actual codeword, the remaining positions of the fixed-size bit arrays are ignored. The predicate *prefixFree* uses *isPrefix* to define the prefix-free property. It requires that neither codeword is a prefix of the other. Therefore, both possible directions are checked. This also means that two equal codewords are not considered prefix-free, since an equal codeword is a prefix of itself. The theorem *golombPrefixFree* applies this property to the codewords generated by *golomb*. The requirement $n_1 \neq n_2$ ensures that two different input numbers are considered. The requirements *validParameter* and *validAuxiliary* ensure that the parameter and auxiliary bounds are valid. Additionally, the two *golombLength* requirements ensure that both generated codewords fit into the available K positions. Under these conditions, the theorem states that the two generated Golomb codewords satisfy *prefixFree*. Thus, the theorem formally verifies that codewords generated for two different input values cannot be prefixes of one another.

4.4 Shannon-Fano Coding, Huffman Coding

A further prefix code is the **Shannon-Fano code**. It was one of the first methods developed for constructing good variable-size codes. The principle is easy to understand [10]:

The input consists of n symbols with known probabilities of occurrence p_i.

1. Sort the symbols in descending order of their probabilities. The sorted symbols are now considered as a list/set.
2. Divide the list into two subsets such that the sums of their probabilities are as equal as possible.
3. Append a 0 to the codewords of all symbols in one subset and a 1 to the codewords of all symbols in the other subset.
4. Recursively repeat this procedure for each subset until every subset contains only a single symbol.

It is easy to notice the satisfaction of the prefix property.

The resulting code depends on how evenly the symbols can be divided during the splitting process. Shannon-Fano coding is particularly efficient, when the symbols have probabilities of (negative) powers of 2 [10].

The recursive splitting process can be represented as a binary tree. Each internal node represents a subset of symbols that is further divided into two subsets. The two resulting branches are assigned the bits 0 and 1, respectively. The splitting continues until each leaf of the tree represents exactly one symbol. The codeword of each symbol is then obtained by following the path from the root to its corresponding leaf and concatenating the bits along this path. The code is constructed **top-down**, from the leftmost to the rightmost bit. During decoding, the tree is traversed from the root to the corresponding leaf [10].

A related method is **Huffman coding**, which generally produces codes with a lower average codeword length than Shannon-Fano coding. Here, the codewords are generated using a binary tree. In contrast to Shannon-Fano coding, the tree is constructed **bottom-up**, starting with the individual symbols and successively combining them into larger groups. The codes are built from the rightmost to the leftmost bit [10].

Huffman coding consists of the following steps:

The input consists of n symbols with known probabilities of occurrence p_i.

1. Sort the symbols in descending order of their probabilities. The sorted symbols are now considered as a list/set.
2. Select the two symbols or subtrees with the lowest probabilities. Add their probabilities and create an auxiliary symbol representing their combination. Remove the two selected elements from the list and insert the auxiliary symbol with the combined probability.
3. Repeat this procedure until only one auxiliary symbol remains, representing the entire set of symbols and forming the root of the tree.
4. Assign a bit of 1 to one branch and a bit of 0 to the other branch of each pair of branches. The assignment of bits to the branches is arbitrary.
5. The codeword for each symbol is obtained by following the path from the root to its corresponding leaf and concatenating the assigned bits along the path.

The resulting structure forms again a binary tree. The original symbols are located at the leaves, while the auxiliary symbols form the internal nodes. The Huffman tree itself is constructed bottom-up by repeatedly combining the two symbols or subtrees with the lowest probabilities. The tree is then traversed to determine the codewords of the symbols. During decoding, the tree is traversed from the root to the corresponding leaf [10].

Since the two methods are similar, we only consider the RISCAL implementation for the Shannon-Fano code. We assume that the ordering is already done and that all probabilities are positive:

```
1     val N: Nat;
2     val F: Nat;
```

```
3
4       type Symbol = Nat[N-1];
5       type Frequency = Nat[F];
6       type Frequencies = Array[N, Frequency];
```

The global values *N* and *F* define the number of symbols and the maximum frequency, respectively. The type *Symbol* represents a symbol by its array index. Since the frequency array contains N entries, valid symbol indices range from 0 to $N - 1$. The type *Frequency* represents the frequency of a symbol and is bounded by F. The array type *Frequencies* stores one frequency for each of the N symbols. The elements of the *Frequencies* array represent the frequencies of the corresponding symbols, e.g., element 0 represents the frequency of symbol 0. The parameters *N=4* and *F=6* are used for all except one procedure which will be mentioned later explicitly.

```
1       pred sorted(f: Frequencies) ⇔
2            ∀i:Symbol, j:Symbol. i < j ⇒ f[i] ≥ f[j];
3       pred validInput(f: Frequencies) ⇔ N ≥ 2 ∧
4            ∀i:Symbol. f[i] > 0 ∧ sorted(f);
```

The predicate *sorted* specifies that the frequencies are ordered in non-increasing order. For every pair of symbol indices *i* and *j* with *i<j*, the frequency at position *i* must therefore be greater than or equal to the frequency at position *j*. The predicate *validInput* combines the requirements for a valid Shannon-Fano input. At least two symbols must be present, every frequency must be positive, and the frequencies must be sorted. The sorting requirement is used because the subsequent partitioning operates directly on contiguous ranges of the already ordered frequency array.

```
1       type Total = Nat[N*F];
```

The type *Total* represents the range of possible sums of frequencies. Since there are N frequencies and every frequency is bounded by F, a total frequency sum is bounded by $N \cdot F$.

```
1       pred validSplit(start: Nat[N], end: Nat[N], k: Nat[N]) ⇔
2            start < k ∧ k < end;
```

The predicate *validSplit* determines whether a split position k is valid for a range $[start, end)$. The conditions *start<k* and *k<end* ensure that the split lies strictly inside the range. Consequently, both resulting subranges $[start, k)$ and $[k, end)$ are non-empty. A split at either boundary would leave one of the two subsets empty and is therefore not considered valid.

```
1       fun sumrange(f: Frequencies, start: Nat[N], end: Nat[N]): Total
2       requires start <= end; =
3       if start = end then 0
4       else f[start] + sumrange(f, start+1, end);
5
6       fun splitDifference(f: Frequencies,start: Nat[N],
7       end: Nat[N],k: Nat[N]): Total
8       requires start < k ∧ k < end;
9       requires start <= end; =
10      if sumrange(f,start,k) >= sumrange(f,k,end) then sumrange(f,start,k)
```

```
    - sumrange(f,k,end)
    else sumrange(f,k,end) - sumrange(f,start,k);
```

The function *sumrange* computes the sum of all frequencies in the half-open range $[\mathrm{start}, \mathrm{end})$. The recursion terminates when *start=end*. In this case the range is empty and its sum is zero. Otherwise, the frequency at position start is added and the function continues with the remaining range $[\mathrm{start}+1, \mathrm{end})$. The precondition *start* $\leqslant$ *end* ensures that the range is not reversed. The function *splitDifference* compares the frequency sums of the two subsets created by a valid split position *k*. The first subset is $[\mathrm{start}, \mathrm{k})$ and the second subset is $[\mathrm{k}, \mathrm{end})$. It returns the absolute difference between their frequency sums. Thus, a smaller value of *splitDifference* means that the two resulting subsets have more similar total frequencies.

```
    pred optimalSplit(f: Frequencies,start: Nat[N],end: Nat[N],
         k: Nat[N]) ⇔ validSplit(start,end,k) ∧
         ∀j:Nat[N]. validSplit(start,end,j)
         ⇒ splitDifference(f,start,end,k)
         ≤ splitDifference(f,start,end,j);

    pred firstOptimalSplit(f: Frequencies,start: Nat[N],end: Nat[N],
         k: Nat[N]) ⇔ optimalSplit(f,start,end,k) ∧
         ∀j:Nat[N]. validSplit(start,end,j) ∧ j < k
         ⇒ ¬optimalSplit(f,start,end,j);
```

The predicate *optimalSplit* specifies that *k* is an optimal split if it is a valid split and its *splitDifference* is less than or equal to the difference of every other valid split position. This corresponds to the Shannon-Fano objective of dividing the current range into two subsets whose total frequencies are as close as possible. If several split positions have the same minimum difference, *firstOptimalSplit* resolves this ambiguity by selecting the leftmost optimal split. More precisely, it requires that there is no smaller valid split position *j<k* that is also optimal. Thus, "first" means the optimal split with the smallest index k, i.e., the split that is furthest to the left within the current range. This makes the selection deterministic even when several split positions are equally good.

```
    proc findSplit(f: Frequencies,start: Nat[N],end: Nat[N]): Nat[N]
    requires validInput(f);
    requires start + 1 < end;
    ensures firstOptimalSplit(f,start,end,result);
    {
        var ret: Nat[N] := 0;

        for var k:Nat[N] := start+1; k < end ∧ ret = 0; k := k+1 do
        {
            if firstOptimalSplit(f,start,end,k) then ret := k;
        }
        return ret;
    }
```

```
    Executing findSplit(Array[ℤ],ℤ,ℤ) with all 60025 inputs.
    Execution completed for ALL inputs (706 ms, 756 checked,
    59269 inadmissible).
```

The procedure *findSplit* searches for the first optimal split position of a given range. The precondition *start+1<end* ensures that the range contains at least two symbols. Since the range is half-open, a range containing exactly one symbol has $\mathrm{end} = \mathrm{start} + 1$ and therefore does not satisfy this condition. Such a range is already a leaf and must not be split further. For a range with at least two symbols, there is at least one possible integer split position satisfying $\mathrm{start} < \mathrm{k} < \mathrm{end}$. The procedure starts with *k=start+1* and examines the possible split positions from left to right. Once a position satisfying *firstOptimalSplit* is found, it is stored in *ret*. The loop condition *ret=0* prevents further updates after a split has been found. The postcondition guarantees that the returned position is the first optimal split of the given range.

```
    type Bit = Nat[1];

    fun splitBit(start: Nat[N],end: Nat[N],k: Nat[N],i: Nat[N]): Bit
    requires start < k ∧ k < end;
    requires start <= i ∧ i < end; =
    if i < k then 1 else 0;
```

The type *Bit* represents a single binary value and is restricted to 0 and 1. The function *splitBit* determines the bit assigned to a symbol during a split. The preconditions require *k* to be a valid split of $[\mathrm{start}, \mathrm{end})$ and require *i* to belong to this range. If *i<k*, the symbol belongs to the first subset $[\mathrm{start}, \mathrm{k})$ and receives the bit 1. Otherwise, it belongs to the second subset $[\mathrm{k}, \mathrm{end})$ and receives the bit 0. This implements the chosen convention that the first, respectively left, subset receives 1 and the second, respectively right, subset receives 0.

```
    type Code = Array[N, Bit];
    type CodeLength = Nat[N];
    type CodeData = Record [code: Code,length: CodeLength];
```

The type *Code* represents the storage for a codeword as an array of *N* bits. The type *CodeLength* stores the actual number of bits belonging to the codeword. The fixed-size array is therefore used as a buffer, while *length* determines which prefix of this buffer represents the actual codeword. The record type *CodeData* combines the bit buffer and its actual length into a single object.

```
    proc appendBit(code: Code,length: CodeLength,bit: Bit): Code
    requires length < N;
    {
        var result:Code := code;
        result[length] := bit;
        return result;
    }
```

The procedure *appendBit* appends a bit to an existing code buffer. The new bit is written at the position specified by *length*. The precondition *length<N* ensures that this position is inside the allocated code array. The procedure does not change the previously stored bits and returns the resulting code buffer.

```
    fun extendCode(data: CodeData,bit: Bit): CodeData
    requires data.length < N; =
    data with .code = appendBit(data.code, data.length, bit)
    with .length = data.length + 1;
```

```
5
6       fun extendSplitCode(data: CodeData,start: Nat[N],end: Nat[N],
7       k: Nat[N],i: Nat[N]): CodeData
8       requires start < k ∧ k < end;
9       requires start <= i ∧ i < end;
10      requires data.length < N;
11      = extendCode(data, splitBit(start,end,k,i));
```

The function *extendCode* extends a complete *CodeData* object. It calls *appendBit* to add the new bit at the current end of the codeword and increases the stored length by one. The function *extendSplitCode* combines this operation with the Shannon-Fano split. It determines the bit assigned to symbol *i* by calling *splitBit* and then extends the current codeword using *extendCode*.

```
1       type Codes = Array[N, CodeData];
2
3       pred validCodes(codes: Codes,start: Nat[N],end: Nat[N]) ⇔
4            ∀i:Nat[N-1], j:Nat[N-1]. ((start <= i ∧ i < end) ∧
5            (start <= j ∧j < end)) ⇒ codes[i].length = codes[j].length;
6
7       pred validCodeRange(codes: Codes,start: Nat[N],end: Nat[N]) ⇔
8            validCodes(codes,start,end) ∧ ∀i:Nat[N-1]. (start <= i ∧
9            i < end) ⇒ codes[i].length <= N - (end - start);
```

The type *Codes* stores one *CodeData* object for each symbol. The predicate *validCodes* requires all codewords belonging to the same range $[\mathrm{start}, \mathrm{end})$ to have the same current length. This reflects the structure of the recursive Shannon-Fano procedure: all symbols within a range have undergone the same number of splits before that range is processed further. The predicate *validCodeRange* extends this condition by additionally restricting the maximum current code length within the range. For every symbol in $[\mathrm{start}, \mathrm{end})$, the current code length must satisfy *codes[i].length* $\leqslant$ *N - (end-start)*. This ensures that the codewords are not already too long for the remaining recursive partitioning of the range. Since each further split adds exactly one bit, the condition guarantees that sufficient capacity remains for all splits that can still occur within the current range. Thus, *validCodeRange* combines the requirement of equal code lengths from *validCodes* with a bound on the current code length. This condition is used as the precondition of *encodeRange* to ensure that only valid intermediate code states are passed to the recursive encoding procedure.

```
1       proc applySplit(codes: Codes,start: Nat[N],end: Nat[N],
2       k: Nat[N]): Codes
3       requires validCodes(codes,start,end);
4       requires start < k ∧ k < end;
5       requires ∀i:Nat[N-1]. (start <= i ∧ i < end) ⇒ codes[i].length < N;
6       ensures ∀i:Nat[N-1]. start <= i ∧ i < end
7               ⇒ result[i] = extendSplitCode(codes[i],start,end,k,i);
8       ensures ∀i:Nat[N-1]. ¬(start <= i ∧ i < end) ⇒ result[i] = codes[i];
9       ensures validCodes(result,start,k);
10      ensures validCodes(result,k,end);
11      {
12          var result:Codes := codes;
```

```
13
14          for var i:Nat[N] := start; i < end; i := i+1 do
15          {
16              result[i] := extendSplitCode(result[i],start,end,k,i);
17          }
18          return result;
19      }
```

For this procedure, N is reduced to 3 to achieve faster execution time.

```
1       Executing applySplit(Array[Record[code:Array[ℤ],length:ℤ]],
2       ℤ,ℤ,ℤ) with all 2097152 inputs.
3       Execution completed for ALL inputs (8284 ms, 15360 checked,
4       2081792 inadmissible).
```

The procedure *applySplit* applies one Shannon-Fano split to all symbols in the range $[start, end)$. The first precondition requires that all codewords in the current range have the same length. The second requires k to be a valid split position. The third ensures that all codewords in the range still have sufficient capacity for one additional bit. For every symbol i in the range, *extendSplitCode* determines the appropriate split bit and appends it to the current codeword. Codewords outside the range are not modified. The first two postconditions describe these changes explicitly: codewords inside the range are extended according to the split, while codewords outside the range remain unchanged. The remaining postconditions guarantee that the two resulting subranges $[start, k)$ and $[k, end)$ each satisfy *validCodes*. Thus, after a split, the invariant required for recursively processing either child range is preserved.

```
1       pred validRange(start: Nat[N], end: Nat[N]) ⇔  start < end;
2       pred leafRange(start: Nat[N], end: Nat[N]) ⇔
3            validRange(start,end) ∧ start + 1 = end;
```

The predicate *validRange* defines a non-empty range by requiring *start<end*. The predicate *leafRange* identifies a range containing exactly one symbol. Because the range is half-open, the condition *start+1=end* means that the only index contained in the range is *start*. Such a range represents a leaf of the Shannon-Fano partitioning process and therefore does not require another split.

```
1       proc encodeRange(f: Frequencies,codes: Codes,start: Nat[N],
2       end: Nat[N]): Codes
3       requires validInput(f);
4       requires validRange(start,end);
5       requires validCodeRange(codes,start,end);
6       {
7           var result:Codes := codes;
8           var k:Nat[N] := 0;
9
10          if leafRange(start,end) then result := codes;
11          else
12          {
13              k := findSplit(f,start,end);
14              result := applySplit(codes,start,end,k);
15              result := encodeRange(f,result,start,k);
16              result := encodeRange(f,result,k,end);
```

```
        }
        return result;
    }
```

The procedure *encodeRange* implements the recursive part of the Shannon-Fano encoding process for a given range. The precondition *validInput* ensures that the frequency array satisfies the requirements of the algorithm, while *validRange* ensures that the current range is non-empty. The precondition *validCodeRange* additionally ensures that the codewords belonging to the current range have a consistent length and that their current length is compatible with the remaining recursive partitioning of the range. If the range is a *leafRange*, it contains exactly one symbol and therefore does not require another split. In this case, *encodeRange* leaves the code array unchanged. Otherwise, *findSplit* determines the first optimal split position. The procedure *applySplit* then appends the corresponding split bit to every codeword in the current range. The resulting code array is subsequently passed to two recursive calls of *encodeRange*, one for the left subrange $[\mathrm{start}, \mathrm{k})$ and one for the right subrange $[\mathrm{k}, \mathrm{end})$. The additional condition represented by *validCodeRange* is important for the recursive calls. Each split increases the code length by one. Therefore, the procedure must only be called for intermediate code states that still have sufficient capacity for the remaining possible splits. This prevents invalid intermediate states from being passed to a recursive call. The recursion continues until every range contains exactly one symbol.

```
    proc makeEmptyCodes(): Codes
    {
        var emptyCode: Code := Array[N,Bit](0);
        var emptyCodeData: CodeData = <<code: emptyCode, length: 0>>;
        var result: Codes := Array[N,CodeData](emptyCodeData);
        return result;
    }
```

The procedure *makeEmptyCodes* creates the initial code representation used by the encoder. First, an array of N bits is initialized with zeros. This array is combined with a length of zero to form an empty *CodeData* object. The resulting *CodeData* is then used to initialize an array containing one empty codeword for each of the N symbols. Thus, all symbols start with an empty codeword before the recursive Shannon-Fano partitioning begins.

```
    pred nonEmptyCodes(codes: Codes) ⇔ ∀i:Nat[N-1]. codes[i].length > 0;
    pred isPrefix(a: CodeData,b: CodeData) ⇔
         a.length <= b.length ∧
         ∀i:Nat[N-1]. i < a.length ⇒ a.code[i] = b.code[i];
    pred prefixFree(codes: Codes) ⇔
         ∀i:Nat[N-1], j:Nat[N-1]. i ≠ j ⇒ ¬isPrefix(codes[i],codes[j]);
```

The predicate *nonEmptyCodes* requires every codeword to have a positive length. This guarantees that no symbol is represented by an empty codeword after encoding. The predicate *isPrefix* determines whether one codeword is a prefix of another. First, the length of the first codeword must not exceed the length of the second. Then, every bit belonging to the first codeword must equal the corresponding bit of the second codeword. Only the actual codeword bits

are compared; unused positions of the fixed-size code buffers are ignored. The predicate *prefixFree* requires that for every two different symbols, neither codeword is a prefix of the other. This formally expresses the prefix-free property of the generated Shannon-Fano code.

```
    proc encode(f: Frequencies): Codes
    requires validInput(f);
    ensures nonEmptyCodes(result);
    ensures prefixFree(result);
    {
        var codes:Codes := makeEmptyCodes();
        return encodeRange(f,codes,0,N);
    }
```

```
    Executing encode(Array[ℤ]) with all 2401 inputs.
    Execution completed for ALL inputs (418 ms, 126 checked,
    2275 inadmissible).
```

The procedure *encode* combines the previously defined components to construct the complete Shannon-Fano code. First, *makeEmptyCodes* creates an empty codeword for every symbol. The complete symbol range $[0, N)$ is then passed to *encodeRange*. This recursively partitions the range and extends the codewords until each range contains exactly one symbol. The postcondition *nonEmptyCodes* guarantees that every resulting codeword contains at least one bit. The postcondition *prefixFree* guarantees that no generated codeword is a prefix of another generated codeword.

```
    fun sameCode(a: CodeData, b: CodeData): Bool =
    a.length = b.length ∧
    ∀i:Nat[N-1]. i < a.length ⇒ a.code[i] = b.code[i];

    pred containsCode(codes: Codes, code: CodeData) ⇔
         ∃i:Symbol. sameCode(codes[i], code);

    proc decodeSymbol(codes: Codes, code: CodeData): Symbol
    requires code.length > 0;
    requires containsCode(codes, code);
    {
        var result:Symbol := 0;

        for var i:Nat[N] := 0; i < N; i := i+1 do
        {
            if sameCode(codes[i], code) then
            result := i;
        }
        return result;
    }

    theorem decodeEncode(f: Frequencies) ⇔
            validInput(f) ⇒ ∀i:Symbol. decodeSymbol(encode(f),
            encode(f)[i]) = i;
    theorem encodedCodesAreUnique(f: Frequencies) ⇔
            validInput(f) ⇒ ∀i:Symbol, j:Symbol. i
            ≠ j ⇒ ¬sameCode(encode(f)[i], encode(f)[j]);
```

The function *sameCode* determines whether two *CodeData* objects represent the same actual codeword. First, their stored lengths must be equal. Then, for every position within that length, the corresponding bits must be equal. The remaining positions of the fixed-size *Code* arrays are deliberately ignored. Therefore, two different buffer contents can still represent the same codeword if their actual lengths and relevant bits are equal. The predicate *containsCode* specifies whether a given codeword occurs in a *Codes* array. It is satisfied if there exists at least one symbol index *i* such that the code stored at *codes[i]* represents the same actual codeword as the supplied *code*, according to *sameCode*. The procedure *decodeSymbol* determines the symbol index corresponding to a given codeword. The first precondition requires the supplied codeword to be non-empty. The second requires that the codeword actually occurs in the supplied *Codes* array. The procedure then iterates over all symbol indices. For every position *i*, *sameCode* compares the stored codeword with the supplied codeword. If they are equal, the corresponding symbol index is stored in *result*. The procedure therefore performs the reverse lookup from a codeword to its associated symbol index. The theorem *decodeEncode* specifies the correspondence between the encoding and decoding procedures. For every valid input and every symbol index *i*, the codeword stored at position *i* in the result of *encode* must be decoded back to the same symbol index *i*. The theorem *encodedCodesAreUnique* describes the uniqueness of Shannon-Fano codes. For all combinations between frequencies and symbols RISCAL checks the correctness of the theorems for *N=8* and *F=6*:

```
    Executing decodeEncode(Array[ℤ]) with all 5764801 inputs.
    Execution completed for ALL inputs (88745 ms, 5764801 checked,
    0 inadmissible).

    Executing encodedCodesAreUnique(Array[ℤ]) with all 5764801 inputs.
    Execution completed for ALL inputs (441377 ms, 5764801 checked,
    0 inadmissible).
```

4.5 String Compression, Dictionary Method, Lempel-Ziv LZ77/LZ1

Compared to previous compression techniques, we now focus on encoding whole ”strings” of symbols instead of individual symbols. This approach is called **string compression**. The basic idea is that it might be more efficient to analyze combinations of symbols that form strings rather than encoding individual symbols. In text files, these strings can, for example, be English words or other frequently occurring sequences. Such strings can be stored in a **dictionary**, which contains strings together with their corresponding codes or references. If a string occurs in the input and is contained in the dictionary, it can be represented by its corresponding code or reference [10]. For suitable data containing recurring strings, string-based compression can achieve better results than methods that encode individual symbols, such as Huffman coding. In **adaptive dictionary methods**, new strings can be added to the dictionary as they are encountered, allowing them to be represented by references later in the input. The effectiveness of string-based and dictionary-based methods

depends on the input data and, in particular, on whether recurring strings can be found. This idea forms the basis for the dictionary methods discussed in this chapter [10].

To illustrate why encoding strings can be more efficient than encoding individual symbols, we consider two symbols a_1 and a_2 with probabilities $p(a_1) = 0.8$ and $p(a_2) = 0.2$. Instead of considering the symbols individually, we now consider strings consisting of two symbols. There are four possible two-symbol strings: $[a_1a_1, a_1a_2, a_2a_1, a_2a_2]$. Assuming independent occurrences of the symbols, their probabilities are $p(a_1a_1) = 0.64$, $p(a_1a_2) = 0.16$, $p(a_2a_1) = 0.16$ and $p(a_2a_2) = 0.04$. Thus, instead of two individual symbols, we now have four possible two-symbol strings with different probabilities. We can now apply Huffman coding to these four two-symbol strings: $a_1a_1 \to 0, a_1a_2 \to 11, a_2a_1 \to 100, a_2a_2 \to 101$. Thus, a_1a_1, which consists of two original symbols, requires only 1 bit, while a_1a_2, a_2a_1, and a_2a_2 require 2, 3, and 3 bits, respectively. The average code length is $0.64 \cdot 1 + 0.16 \cdot 2 + 0.16 \cdot 3 + 0.04 \cdot 3 = 1.56 \,\mathrm{bits\ per\ two\ symbol\ string}$. Since each string contains two original symbols, this corresponds to 0.78 bit/symbol. For comparison, encoding the individual symbols requires 1 bit per symbol in this two-symbol alphabet. Thus, in this example, considering two-symbol strings reduces the average code length. If the string length is increased, more possible strings arise, and their probabilities become increasingly different from each other. This allows the variable-size coding to exploit the difference between frequent and infrequent strings more effectively. In Salomon's example, the average code size per original symbol decreases as the string length increases: from 1 bit for individual symbols, to 0.78 bits for two-symbol strings, and to 0.728 bits for three-symbol strings [10].

A dictionary compression procedure for data bytes could be as follows [10]:

1. Read the input file and prepare a list of different bytes read, together with their number of occurrences.
2. Order the list in descending order of frequency. The list can have at most 256 elements.
3. The list becomes the dictionary, which is written to the compressed file, preceded by its 1-byte integer length.
4. Read the input data again and locate each byte in the dictionary to determine its index. The index is written to the compressed file, preceded by 3 bits that represent the index's length. 000 indicates a length of 1 bit.

Decompression is done as follows:

1. Read the size and dictionary from the compressed file.
2. Decode the compressed data by reading the first 3 bits to identify how many bits have to be read for the index.
3. Use the index to find the original symbol in the dictionary.

This method yields a compression ratio of $C_r = 0.9375$. The decoding process is fast, but the disadvantage lies in the encoding process, since the input file must be read twice and searching for bytes in the dictionary is also slow because, for example, binary search is not possible [10].

We can now explain the well-known method of **Lempel-Ziv 77 (LZ77)**. It builds on the ideas of the dictionary compression procedure explained above. Here, the input file is not fully read and transformed into a dictionary as the first step, instead, only a certain part of the input is considered. The part that serves as the dictionary is called the **search buffer**. It is continuously shifted: new inputs from the file is added, while old data are removed from the buffer. The shifting process shifts data from the **look-ahead buffer**, which contains the still not encoded data, into the search buffer. This process can be understood as a window that slides across the input data. The window consists of the search buffer on the left and the look-ahead buffer on the right. Typically, the search buffer is several thousands of bytes long, whereas the look-ahead buffer consists of only a few tens of bytes. The encoder takes the first symbol a_i from the look-ahead buffer and searches the search buffer from right to left for the same symbol. If a match is found, its distance from the current position is recorded as the **offset** o. It then checks how many subsequent symbols match in the search buffer and the look-ahead buffer. The number of matching symbols is called the **length** l. After finding a match, the encoder continues searching further to the left for other occurrences of a_i. The encoder then creates a **token** (o, l, a_{i+l}) for the longest match. If all matches have the same length, it selects the last match found. a_{i+l} is the symbol immediately following the matched string; this makes the encoding easier. The token is written to the output file. The window is then shifted by $l + 1$ positions to the right: l positions for the matched string and one position for the next symbol. If a_i is not found in the search buffer, the token has an offset and length of 0. Such tokens are not good for compression [10].

The decoder has a buffer as big as the encoder's window. For decoding, a token is taken. The match is found in the buffer and then written on the output followed by the third component of the token. The match and the third component are then shifted into the buffer. The decoding process is much simpler than the encoding process; hence, LZ77 is asymmetric. This variant assumes that repeated patterns in the input occur within relatively short distances [10].

We now implement LZ77 in RISCAL:

```
    val N: Nat;
    val A: Nat;
    val S: Nat;
    val L: Nat;

    type Symbol = Nat[A];
    type Input = Array[N, Symbol];
    type Index = Nat[N-1];
```

The parameters *N*, *A*, *S*, and *L* define the finite model of the LZ77 encoder. The parameter *N* specifies the length of the input sequence, while *A* determines the finite alphabet from which the input symbols are taken. The type *Symbol* represents an individual input symbol as an element of *Nat[A]*. The type *Input* represents the complete input sequence as a fixed-size array of *N* symbols. The parameter *S* specifies the size of the search buffer, which contains already processed input symbols. The parameter *L* specifies the size of the look-ahead buffer, which contains the part of the input that is currently being encoded.

The type *Index* represents a valid position within the input array and therefore ranges from 0 to *N*-1. The parameters are kept symbolic rather than fixing concrete buffer and input sizes. This allows the procedures and their properties to be checked for all finite instances satisfying the respective bounds of the model. In the beginning, *N=8*, *A=3*, *S=3*, and *L=2* are used.

```
fun searchStart(p: Index): Index = if p < S then 0 else p - S;
pred inSearchBuffer(q: Index, p: Index) ⇔
     searchStart(p) <= q ∧ q < p;
pred validMatchLength(p: Index, q: Index, k: Nat[N]) ⇔
     k + 1 < L ∧ p + k < N ∧ q + k < N;
```

The function *searchStart* determines the first position of the search buffer for the current input position *p*. If fewer than *S* symbols have been processed, the search buffer starts at position 0. Otherwise, its lower boundary is *p* - *S*. Thus, the search buffer contains at most *S* previously processed symbols. The predicate *inSearchBuffer* determines whether a position *q* belongs to the search buffer associated with *p*. The condition $searchStart(p) \leqslant q < p$ ensures that *q* is within the current search buffer and lies strictly before the current position *p*. The upper bound is exclusive because *p* marks the first position of the look-ahead buffer and therefore does not belong to the search buffer. The half-open interval is also useful for the subsequent matching procedure, since it gives a direct characterization of all positions at which a match may start. The predicate *validMatchLength* specifies the conditions under which another symbol may be included in a match. The first condition limits the match to the look-ahead buffer, while the remaining two conditions ensure that both compared positions remain within the input sequence.

```
proc matchLength(input: Input, q: Index, p: Index): Nat[N]
requires inSearchBuffer(q, p);
{
    var k: Nat[N] := 0;

    while (validMatchLength(p, q, k) ∧
    input[q + k] = input[p + k]) do
    {
        k := k + 1;
    };
    return k;
}
```

The procedure *matchLength* determines the length of the match between a position *q* in the search buffer and the current position *p*. The precondition requires *q* to be a valid position in the current search buffer. This reflects the LZ77 principle that a match is searched for in the already processed part of the sliding window. The local variable *k* stores the current match length. It is initially zero. As long as *validMatchLength* holds and the symbols at positions *q* + *k* and *p* + *k* are equal, *k* is increased by one. When one of these conditions no longer holds, *k* represents the length of the matching sequence.

```
type Match = Record [q: Index,length: Nat[N]];

pred isLongestMatch(input: Input, p: Index, q: Index,
     length: Nat[N]) ⇔
```

```
          length = 0 ∨ (inSearchBuffer(q, p) ∧
          forall r: Index with inSearchBuffer(r, p).
          matchLength(input, r, p) <= length);
    pred isLeftmostLongestMatch(input: Input, p: Index,
          q: Index, length: Nat[N]) ⇔
          length = 0 ∨ (inSearchBuffer(q, p) ∧
          forall r: Index with inSearchBuffer(r, p).
          (matchLength(input, r, p) = length ∧ r < q) ⇒ false);
```

The type *Match* stores the position *q* at which a match was found and its corresponding *length*. The predicate *isLongestMatch* specifies that a match of length *length* is at least as long as every other match starting in the current search buffer. The case $length = 0$) represents the situation in which no positive-length match exists. The predicate *isLeftmostLongestMatch* specifies the tie-breaking rule. If several matches have the same maximum length, the selected position must not have another match of the same length at a smaller position *r*. Consequently, the selected match is the leftmost among the longest matches.

```
    proc bestMatch(input: Input, p: Index): Match
    ensures isLongestMatch(input, p, result.q, result.length);
    ensures isLeftmostLongestMatch(input, p, result.q, result.length);
    {
        var best: Match = <<q: 0, length: 0>>;
        var q: Index := p;

        while (q > searchStart(p)) do
        {
            q := q - 1;
            var length: Nat[N] := matchLength(input, q, p);
            if (length >= best.length) then
            {
                best.q := q;
                best.length := length;
            }
        };
        return best;
    }
```

```
    Executing bestMatch(Array[ℤ],ℤ) with all 524288 inputs.
    Execution completed for ALL inputs (6588 ms, 524288 checked,
    0 inadmissible).
```

The procedure *bestMatch* searches the current search buffer for the best match for the look-ahead buffer beginning at position *p*. The local record *best* stores the best match found so far. The variable *q* is initialized with *p*. In every iteration it is first decreased by one. Consequently, the search proceeds from right to left through the search buffer. For every candidate position, *matchLength* determines the corresponding match length. If the newly determined *length* is greater than or equal to *best.length*, the current candidate replaces the previous best match. The use of $>=$ is important for the tie-breaking behavior: when two matches have the same length, the later candidate replaces the previous one. Since the search proceeds from right to left, this results in the leftmost position being retained among matches of equal length. The first

postcondition guarantees that the returned *result.q* and *result.length* describe a longest match. The second postcondition guarantees the specified leftmost tie-breaking property. Thus, the postconditions formally specify the two essential selection properties of *bestMatch*.

```
    type Token = Record [offset: Nat[N],length: Nat[N],
         nextSymbol: Symbol,isEnd: Bool];
    pred hasNextSymbol(p: Index, length: Nat[N]) ⇔ p + length < N;
```

The type *Token* represents one LZ77 output token. The field *offset* stores the distance from the current position to the beginning of the selected match. The field *length* stores the length of the matching sequence. The field *nextSymbol* stores the symbol immediately following the match. The additional field *isEnd* is used to represent the special case in which a match reaches the end of the input and therefore has no following symbol. This field is an extension of the three-component LZ77 token representation used here to handle end-of-input within the fixed formal model. The predicate *hasNextSymbol* determines whether a symbol exists after a match of length *length* starting at position *p*. A following symbol exists exactly when $p + length$ is still smaller than *N*.

```
    proc makeToken(input: Input, p: Index): Token
    {
        var best: Match = bestMatch(input, p);
        var token: Token = <<offset: 0, length: 0, nextSymbol: 0,
            isEnd: false>>;

        if (best.length = 0) then
        {
            token.offset := 0;
        }
        else
        {
            token.offset := p - best.q;
        }

        token.length := best.length;

        if (hasNextSymbol(p, best.length)) then
        {
            token.nextSymbol := input[p + best.length];
            token.isEnd := false;
        }
        else
        {
            token.nextSymbol := 0;
            token.isEnd := true;
        }
        return token;
    }
```

The procedure *makeToken* converts the result of *bestMatch* into an LZ77 token. If no positive-length match exists, *offset* is explicitly set to zero. This represents the literal-symbol case using the conventional form $(0, 0, nextSymbol)$. If a match exists, *offset* is calculated as the distance between the current position *p* and the selected match position *best.q*.

The selected match length is then stored in *token.length*. The subsequent handling of *nextSymbol* distinguishes between an ordinary token and a final token. If *hasNextSymbol* holds, the symbol immediately following the match is stored in *nextSymbol* and *isEnd* remains false. Otherwise, the match reaches the end of the input. In this case, *isEnd* is set to true and *nextSymbol* is assigned the value zero only as a placeholder, since no actual input symbol exists after the match.

```
    type Position = Nat[N];
    type Tokens = Array[N, Token];
    type Encoding = Record[tokens: Tokens,count: Nat[N]];
    type PositionResult = Record [position: Position,valid: Bool];
```

The type *Tokens* provides storage for up to *N* tokens. The type *Encoding* combines this fixed-size storage with the field *count*, which records how many entries of *tokens* actually belong to the encoding. The separate type *Position* is used for positions that may also reach *N*. This is important for the encoder because after processing the final part of the input, the position can become exactly *N*, whereas an *Index* only represents positions from 0 to $N-1$. The record type *PositionResult* combines a calculated *position* with a Boolean field *valid*. It is used by *tokenPosition* to distinguish between a valid position and a position calculation that has already become invalid. This allows the validity of an encoding to be checked without relying on positions outside the modeled range.

```
    pred validToken(token: Token, p: Position) ⇔ ((token.length = 0 ∧
         token.offset = 0) ∨ (token.length > 0 ∧ token.offset > 0
         ∧ token.offset <= p ∧ p - token.offset + token.length <= N)) ∧
         ((token.isEnd ∧ p + token.length = N) ∨ (token.isEnd = false ∧
         p + token.length < N));

    pred validEndPosition(encoding: Encoding, i: Nat[N]) ⇔
         i < encoding.count ∧ (i = encoding.count - 1 ∨
         encoding.tokens[i].isEnd = false);

    fun tokenPosition(encoding: Encoding, i: Nat[N]): PositionResult
    requires encoding.count > 0; =
    if i = 0 then <<position: 0, valid: true>>
    else
        if tokenPosition(encoding, i - 1).valid then
            if tokenPosition(encoding, i - 1).position +
            encoding.tokens[i - 1].length + 1 <= N then
            <<position: tokenPosition(encoding, i - 1).position +
            encoding.tokens[i - 1].length + 1, valid: true>>
            else <<position: 0, valid: false>>
        else <<position: 0, valid: false>>;

    pred encodingReachesEnd(encoding: Encoding) ⇔ encoding.count > 0 ∧
         ((encoding.tokens[encoding.count - 1].isEnd ∧
         tokenPosition(encoding, encoding.count - 1).position +
         encoding.tokens[encoding.count - 1].length = N) ∨
         (encoding.tokens[encoding.count - 1].isEnd = false ∧
         tokenPosition(encoding, encoding.count - 1).position +
         encoding.tokens[encoding.count - 1].length + 1 = N));

```

```
    pred validEncodingSpec(encoding: Encoding) ⇔
         if encoding.count = 0 then false
         else (forall i: Nat[N] with i < encoding.count.
         tokenPosition(encoding, i).valid ∧
         validToken(encoding.tokens[i],
         tokenPosition(encoding, i).position) ∧
         validEndPosition(encoding, i)) ∧ encodingReachesEnd(encoding);
```

The predicate *validToken* specifies the conditions that an individual token must satisfy at a given input position *p*. Two cases are distinguished. If *token.length* is zero, the token represents the literal case and its *offset* must also be zero. Otherwise, the token represents a match and its offset must be positive, may not exceed the current position, and must refer to a source range that remains within the modeled input. The second part of *validToken* checks the end position of the token. If *isEnd* is true, the match must end exactly at position *N*. If *isEnd* is false, a following symbol must still exist, so the end of the match must remain strictly before *N*. The predicate *validEndPosition* ensures that a token marked with *isEnd* can occur only as the last token of the encoding. All preceding tokens must therefore have *isEnd* set to false. The function *tokenPosition* reconstructs the input position at which the *i*-th token starts. The first token starts at position zero. For every subsequent token, the position of the previous token is advanced by its match length plus one, corresponding to the matched sequence and its following symbol. The function returns both the calculated position and a validity flag. If the preceding position is invalid or the next position would exceed *N*, the result is marked as invalid. Finally, the predicate *encodingReachesEnd* specifies that the encoding must cover the complete input. If the final token has *isEnd* set to true, its match must end exactly at *N*. Otherwise, the final token must be followed by its *nextSymbol*, so its position plus its match length and one additional symbol must equal *N*. The predicate *validEncodingSpec* combines the individual validity conditions into one declarative specification of a valid encoding. An encoding with no tokens is rejected. For every token position within the encoding, the corresponding position must be valid, the token itself must satisfy *validToken*, and the token must satisfy the restriction imposed by *validEndPosition*. In addition, *encodingReachesEnd* must hold. Thus, a valid encoding is required not only to consist of individually valid tokens, but also to represent the complete input up to position *N*.

```
    proc validEncoding(encoding: Encoding): Bool
    requires encoding.count > 0;
    ensures result ⇔ validEncodingSpec(encoding);
    {
        var i: Nat[N] := 0;
        var valid: Bool := true;

        while (i < encoding.count ∧ valid) do
        {
            var positionResult: PositionResult =
            tokenPosition(encoding, i);

            if positionResult.valid then
            {
                if (validToken(encoding.tokens[i],
```

```
                positionResult.position) ∧  validEndPosition(encoding,
                i)) then
                {
                    i := i + 1;
                }
                else
                {
                    valid := false;
                }
            }
            else
            {
                valid := false;
            }
        };
        return valid ∧ encodingReachesEnd(encoding);
    }
```

For this procedure, N is reduced to 3 and A to 2 to lower the execution time of RISCAL executions due to higher possible inputs.

```
    Executing validEncoding(Record[tokens:Array[Record[offset:ℤ,
    length:ℤ,
    nextSymbol:ℤ,isEnd:Bool]],count:ℤ]) with all 3538944 inputs.
    Execution completed for ALL inputs (14442 ms, 2654208 checked,
    884736 inadmissible).
```

The procedure *validEncoding* checks whether a complete encoding satisfies the formal validity specification. Its precondition requires the encoding to contain at least one token. The postcondition states that the returned Boolean value is equivalent to the declarative predicate *validEncodingSpec*. The variable *i* identifies the token currently being checked, while *valid* stores the result of the validity check. For each token, *tokenPosition* first determines its starting position. If this position is valid, both *validToken* and *validEndPosition* are evaluated. If one of these conditions fails, the encoding is marked as invalid. After all tokens have been processed, the procedure additionally checks *encodingReachesEnd*. Consequently, the procedure accepts only encodings in which all tokens are valid and the final token reaches the end of the input.

```
    proc encode(input: Input): Encoding
    ensures validEncoding(result);
    {
        var emptyToken: Token = <<offset: 0, length: 0, nextSymbol: 0,
            isEnd: false>>;
        var emptyTokens: Tokens = Array[N, Token](emptyToken);
        var result: Encoding = <<tokens: emptyTokens, count: 0>>;
        var p: Position := 0;

        while (p < N) do
        {
            var token: Token = makeToken(input, p);

            result.tokens[result.count] := token;
            result.count := result.count + 1;
```

```

            if (token.isEnd) then
            {
                p := N;
            }
            else
            {
                p := p + token.length + 1;
            }
        };
        return result;
    }
```

```
    Executing encode(Array[ℤ]) with all 65536 inputs.
    Execution completed for ALL inputs (59425 ms, 65536 checked,
    0 inadmissible).
```

The procedure *encode* processes the input from left to right. The variable *p* denotes the first position of the part of the input that has not yet been encoded. For every iteration, *makeToken* creates a token for the current position. The token is stored at the next free position in *result.tokens*, and *result.count* is increased accordingly. If *isEnd* is false, the encoder advances by *token.length* plus one. The first part corresponds to the matched sequence, while the additional position corresponds to *nextSymbol*. Thus, the next iteration begins immediately after the complete portion represented by the token. If *isEnd* is true, the token represents a final match without a following symbol. In this case, *p* is set directly to *N*, terminating the encoding process. The postcondition additionally specifies that the resulting encoding satisfies *validEncoding*. Thus, the encoder does not only construct an encoding, but formally guarantees that its result fulfills the defined validity conditions.

```
    type Output = Array[N, Symbol];
```

The type *Output* represents the reconstructed input sequence using the same fixed-size representation as *Input*.

```
    proc decode(encoding: Encoding): Output
    requires encoding.count > 0;
    requires validEncoding(encoding);
    {
        var emptyOutput: Output = Array[N, Symbol](0);
        var output: Output := emptyOutput;
        var p: Position := 0;
        var i: Nat[N] := 0;

        while (i < encoding.count) do
        {
            var token: Token = encoding.tokens[i];
            var source: Position := p - token.offset;
            var j: Nat[N] := 0;

            while (j < token.length) do
            {
                output[p + j] := output[source + j];
```

```
                j := j + 1;
            };
            if (token.isEnd) then
            {
                p := N;
            }
            else
            {
                output[p + token.length] := token.nextSymbol;
                p := p + token.length + 1;
            };
            i := i + 1;
        };
        return output;
    }
```

The procedure *decode* reconstructs the original input from a valid encoding. The two preconditions require the encoding to contain at least one token and to satisfy *validEncoding*. The decoder therefore operates only on encodings whose token structure and positions have already been verified. The variable *i* identifies the current token, while *p* denotes the position at which the decoded representation is currently extended. For each token, *source* identifies the beginning of the referenced sequence by subtracting the token's *offset* from the current output position. The inner loop copies *token.length* symbols from the source position to the current output position. The copying is performed one symbol at a time. This is important for LZ77 because the source and destination ranges may overlap. A symbol written earlier during the same copying operation can therefore become the source for a subsequent symbol. If *isEnd* is false, *nextSymbol* is written immediately after the copied match and *p* is advanced by the match length plus one. If *isEnd* is true, the token represents the final match without a following symbol. In this case no additional symbol is written and *p* is set directly to *N*.

```
    theorem decodeEncode(input: Input) ⇔ decode(encode(input)) = input;
```

An encoding is considered correct if applying *decode* to the result of *encode* reproduces the original *input* exactly. The theorem *decodeEncode* states this property for every modeled input. It therefore connects the complete encoding and decoding procedures and provides the final correctness statement of the formalization. The theorem holds:

```
    Executing decodeEncode(Array[ℤ]) with all 65536 inputs.
    Execution completed for ALL inputs (113977 ms, 65536 checked,
    0 inadmissible).
```

Chapter 5

Conclusion

This thesis investigated the formal modelling and verification of selected techniques from channel and source coding using the RISC Algorithm Language (RISCAL). Various coding algorithms were mathematically specified and implemented, including error-detecting and error-correcting codes as well as different data compression techniques. Their properties were expressed as formal specifications and verified over finite domains using RISCAL.

The results show that RISCAL is a suitable tool for formally modelling and verifying coding algorithms. The finite nature of the considered data structures makes it possible to exhaustively check the specified properties for all possible inputs of a given model size. This was demonstrated, for example, by verifying the encoding, decoding, error detection, and error correction properties of the implemented Hamming code. Similar specifications were used for the considered source-coding techniques.

At the same time, the implementation showed that formal modelling requires careful consideration of possible states and input combinations. Although the mathematical specification of many algorithms is relatively straightforward, special cases and invalid inputs have to be explicitly addressed by predicates, preconditions, and other restrictions. Furthermore, the number of possible input combinations can grow rapidly with the size and number of parameters of the underlying data structures. This can significantly increase the computational effort required for exhaustive verification, as observed in the source-coding implementations.

Overall, the work demonstrates that coding-theoretic algorithms can be expressed and formally verified in RISCAL. The combination of mathematical specifications, executable algorithms, and exhaustive verification provides a systematic way to investigate the correctness of these algorithms for finite models.

Bibliography

[1] C. E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal*, 27, 3, 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[2] R. Hill. 1986. *A First Course in Coding Theory. Oxford Applied Mathematics and Computing Science Series*. Clarendon Press. ISBN: 9780198538035. https://global.oup.com/academic/product/a-first-course-in-coding-theory-9780198538035?cc=at&lang=en&#.

[3] K. Sayood. 2006. *Introduction to Data Compression. EBSCO ebook academic collection*. Elsevier Science. ISBN: 9780126208627. DOI: 10.1016/B978-0-12-620862-7.X5000-7.

[4] Thomas M. Cover. 2006. *Elements of Information Theory*. (2. ed.). *A Wiley-Interscience publication*. Wiley-Interscience. ISBN: 9780471241959. DOI: 10.1002/047174882X.

[5] Edmund Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. The MIT Press. ISBN: 978-0-262-03270-4. https://mitpress.mit.edu/9780262032704/model-checking/.

[6] Wolfgang Schreiner. 2023. *Concrete Abstractions: Formalizing and Analyzing Discrete Theories and Algorithms with the RISCAL Model Checker*. Springer. ISBN: 978-3-031-24936-5. DOI: 10.1007/978-3-031-24934-1.

[7] Wolfgang Schreiner. 2019. *The RISC Algorithm Language (RISCAL)*. Research Institute for Symbolic Computation (RISC, Johannes Kepler Universiy. https://www.risc.jku.at/research/formal/software/RISCAL.

[8] Wolfgang Schreiner. 2017. *The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual(Version1.0)*. Available at [7]. Research Institute for Symbolic Computation (RISC), Johannes Kepler Universiy.

[9] F. Jessie MacWilliams and N. J. A. Sloane. 1977. The Theory of Error-Correcting Codes. In (*North-Holland Publishing Company*). Elsevier. ISBN: 978-0-444-85010-2. DOI: 10.1016/S0924-6509(08)70521-X.

[10] David Salomon. 2004. *Data compression : The Complete Reference*. (3rd ed.). Springer. ISBN: 978-0-387-21832-8. DOI: 10.1007/b97635.

Appendix A

A.1 Even Weight Code

```
    val K: Nat;
    val N = K+1;

    type Bit = Nat[1];
    type Message = Array[K,Bit];
    type Codeword = Array[N,Bit];

    fun xor(a: Bit, b: Bit): Bit =
    if a = b then 0
    else 1;

    fun parityMessage(m: Message, i: Nat[K-1]): Bit =
    if i = 0 then m[0]
    else xor(parityMessage(m,i-1), m[i]);

    fun parityCodeword(c: Codeword, i: Nat[N-1]): Bit =
    if i = 0 then c[0]
    else xor(parityCodeword(c,i-1), c[i]);

    pred validCodeword(c: Codeword) ⇔ parityCodeword(c,N-1) = 0;

    pred singleError(c1: Codeword, c2: Codeword) ⇔ ∃i:Nat[N-1].
         c1[i] ≠ c2[i] ∧ ∀(j:Nat[N-1] with j ≠ i. c1[j] = c2[j]);

    proc encode(m: Message): Codeword
    ensures validCodeword(result);
    {
        var c: Codeword := Array[N,Bit](0);

        for var i:Nat[K] := 0; i < K; i := i+1 do
        c[i] := m[i];

        c[N-1] := parityMessage(m,K-1);

        return c;
    }

    proc decode(c: Codeword): Message
    requires validCodeword(c);
    {
        var m: Message := Array[K,Bit](0);

        for var i:Nat[K] := 0; i < K; i := i+1 do
```

```
44          m[i] := c[i];
45
46          return m;
47      }
48
49      theorem DecodeEncode(m: Message) ⇔ decode(encode(m)) = m;
50      theorem DetectSingleBitError(m: Message, c: Codeword) ⇔
51              singleError(encode(m),c) ⇒ ~validCodeword(c);
```

A.2 Hamming Code

```
type Bit = Nat[1];

type Message = Array[4,Bit];
type Codeword = Array[7,Bit];

fun xor(a: Bit, b: Bit): Bit =
if a = b then 0 else 1;

fun p1(m: Message): Bit =
xor(xor(m[0],m[1]),m[3]);

fun p2(m: Message): Bit =
xor(xor(m[0],m[2]),m[3]);

fun p3(m: Message): Bit =
xor(xor(m[1],m[2]),m[3]);

pred validCodeword(c: Codeword) ⇔ c[0] = xor(xor(c[2],c[4]),c[6]) ∧
     c[1] = xor(xor(c[2],c[5]),c[6]) ∧ c[3] =
     xor(xor(c[4],c[5]),c[6]);

proc encode(m: Message): Codeword
ensures validCodeword(result);
{
    var c: Codeword := Array[7,Bit](0);

    c[2] := m[0];
    c[4] := m[1];
    c[5] := m[2];
    c[6] := m[3];

    c[0] := p1(m);
    c[1] := p2(m);
    c[3] := p3(m);

    return c;
}

proc syndrome(c: Codeword): Array[3,Bit]
{
    var s: Array[3,Bit]:= Array[3,Bit](0);

    s[0] := xor(xor(xor(c[0],c[2]),c[4]),c[6]);

    s[1] := xor(xor(xor(c[1],c[2]),c[5]),c[6]);

    s[2] := xor(xor(xor(c[3],c[4]),c[5]),c[6]);

    return s;
}

fun syndromeValue(s: Array[3,Bit]): Nat[7] =
4*s[2] + 2*s[1] + s[0];
```

```
54
55      pred singleError(c1: Codeword, c2: Codeword) ⇔ ∃i:Nat[6].
56           c1[i] ≠ c2[i] ∧ ∀(j:Nat[6] with j ≠ i. c1[j] = c2[j]);
57
58      proc correct(c: Codeword): Codeword
59      {
60          var r: Codeword := Array[7,Bit](0);
61
62          for var i:Nat[7] := 0; i < 7; i := i+1 do
63          r[i] := c[i];
64
65          var s: Array[3,Bit]:= Array[3,Bit](0);
66
67          s := syndrome(c);
68
69          var pos: Nat[7];
70
71          pos := syndromeValue(s);
72
73          if pos > 0 then r[pos-1] := xor(r[pos-1],1);
74
75          return r;
76      }
77
78      proc decode(c: Codeword): Message
79      requires validCodeword(c);
80      {
81          var m: Message := Array[4,Bit](0);
82
83          m[0] := c[2];
84          m[1] := c[4];
85          m[2] := c[5];
86          m[3] := c[6];
87
88          return m;
89      }
90
91      fun syndromeValueOfCodeword(c: Codeword): Nat[7] =
92          4 * xor(xor(xor(c[3],c[4]),c[5]),c[6]) +
93          2 * xor(xor(xor(c[1],c[2]),c[5]),c[6]) +
94          xor(xor(xor(c[0],c[2]),c[4]),c[6]);
95
96
97      theorem DecodeEncode(m: Message) ⇔ decode(encode(m)) = m;
98      theorem DetectSingleError(m: Message,c: Codeword) ⇔
99              singleError(encode(m),c) ⇒ syndromeValueOfCodeword(c) > 0;
100     theorem CorrectSingleError(m: Message,c: Codeword) ⇔
101             singleError(encode(m),c) ⇒ correct(c) = encode(m);
102     pred doubleError(c1: Codeword, c2: Codeword) ⇔ ∃i:Nat[6], j:Nat[6].
103          i ≠ j ∧ c1[i] ≠ c2[i] ∧ c1[j] ≠ c2[j] ∧
104          ∀(k:Nat[6] with k ≠ i ∧ k ≠ j. c1[k] = c2[k]);
105     theorem DetectsDoubleError(m: Message,c: Codeword) ⇔
106             doubleError(encode(m),c) ⇒ ~validCodeword(c);
```

A.3 Extension

```
    val K: Nat;
    val N = K;
    val NXT = N+1;

    type Bit = Nat[1];
    type Codeword = Array[N,Bit];
    type ExtendedCodeword = Array[NXT,Bit];

    fun xor(a: Bit, b: Bit): Bit =
    if a = b then 0 else 1;

    fun parity(c: Codeword, i: Nat[N-1]): Bit =
    if i = 0 then c[0]
    else xor(parity(c,i-1), c[i]);

    fun extendedBit(c: Codeword, i: Nat[N]): Bit =
    if i = N then parity(c,N-1)
    else c[i];

    fun parityExtended(c: Codeword, i: Nat[N]): Bit =
    if i = 0 then extendedBit(c,0)
    else xor(parityExtended(c,i-1), extendedBit(c,i));

    pred validExtension(c: Codeword) ⇔ parityExtended(c,N) = 0;

    proc extend(c: Codeword): ExtendedCodeword
    ensures ∀i:Nat[N-1]. result[i] = c[i];
    ensures result[N] = parity(c,N-1);
    {
        var e: ExtendedCodeword := Array[NXT,Bit](0);

        for var i:Nat[N] := 0; i < N; i := i+1 do e[i] := c[i];

        e[N] := parity(c,N-1);

        return e;
    }

    theorem ExtendingAddsParityBit(c: Codeword) ⇔
            extendedBit(c,N) = parity(c,N-1);

    theorem ExtendingPreservesCodeword(c: Codeword, i: Nat[N-1]) ⇔
            extendedBit(c,i) = c[i];

    theorem ExtendingCorrect(c: Codeword) ⇔ validExtension(c);

    fun distanceFrom(c1: Codeword, c2: Codeword, i: Nat[N-1]): Nat[N] =
    if i = 0 then
    if c1[0] = c2[0] then 0 else 1
    else
    distanceFrom(c1,c2,i-1) + (if c1[i] = c2[i] then 0 else 1);

    fun distance(c1: Codeword, c2: Codeword): Nat[N] =
```

```
54      if N = 0 then 0
55      else distanceFrom(c1,c2,N-1);
56
57      fun extendedDistance(c1: Codeword, c2: Codeword): Nat[NXT] =
58      if extendedBit(c1,N) = extendedBit(c2,N) then distance(c1,c2)
59      else distance(c1,c2) + 1;
60
61      pred minimumDistance(d: Nat[N]) ⇔ ∀(c1:Codeword,
62           c2:Codeword with c1 ≠ c2. distance(c1,c2) ≥ d) ∧
63           ∃(c1:Codeword, c2:Codeword. c1 ≠ c2 ∧ distance(c1,c2) = d);
64
65      pred extendedMinimumDistance(d: Nat[NXT]) ⇔ ∀(c1:Codeword,
66           c2:Codeword with c1 ≠ c2. extendedDistance(c1,c2) ≥ d) ∧
67           ∃(c1:Codeword, c2:Codeword. c1 ≠ c2 ∧
68            extendedDistance(c1,c2) = d);
69
70      theorem ExtendingEvenDistance(c1: Codeword, c2: Codeword)
71      requires c1 ≠ c2 ∧ distance(c1,c2) % 2 = 0; ⇔
72      extendedDistance(c1,c2) = distance(c1,c2);
73
74      theorem ExtendingOddDistance(c1: Codeword, c2: Codeword)
75      requires c1 ≠ c2 ∧ distance(c1,c2) % 2 = 1; ⇔
76      extendedDistance(c1,c2) = distance(c1,c2) + 1;
77
78      theorem MinimumDistanceAfterExtension(d: Nat[N])
79      requires minimumDistance(d) ∧ d % 2 = 1; ⇔
80      extendedMinimumDistance(d+1);
```

A.4 Run Length Encoding

```
val N:Nat;
val A:Nat;

type Symbol = Nat[A];
type Index = Nat[N-1];
type Text = Array[N,Symbol];
type Count = Nat[N];
type ElementCount = Nat[N];
type Element = Record [compressed: Bool,count: Count,symbol: Symbol];
type EncodedText = Record [length: ElementCount,
     elements: Array[N,Element]];

pred run(t:Text,start:Index,count:Count,s:Symbol) ⇔
     start + count <= N ∧
     ∀j:Count with j < count. t[start + j] = s;

pred completeRun(t:Text,start:Index,count:Count,s:Symbol) ⇔
     run(t,start,count,s) ∧ (start = 0 ∨
     t[start - 1] ≠ s) ∧ (start + count = N ∨ t[start + count] ≠ s);

pred runStartsAt(t:Text,start:Index,s:Symbol) ⇔ t[start] = s ∧
     (start = 0 ∨ t[start - 1] ≠ s);

pred validElement(e:Element) ⇔ (e.compressed = false ∧ e.count = 1)
     ∨ (e.compressed = true ∧ e.count >= 3);

pred validEncoded(e:EncodedText) ⇔ ∀(i:Index with i < e.length.
     validElement(e.elements[i])) ∧ e.length > 0;

pred decodedLengthBounded(e: EncodedText) ⇔ (sum i: Index.
     if i < e.length then e.elements[i].count else 0) <= N;

fun decodedLength(e:EncodedText):Nat[N]
requires validEncoded(e) ∧ decodedLengthBounded(e); =
sum i:Index. if i < e.length then e.elements[i].count else 0;

fun elementStart(e:EncodedText,i:Index):Nat[N]
requires validEncoded(e) ∧ decodedLengthBounded(e) ∧ i < e.length; =
sum j:Index. if j < i then e.elements[j].count else 0;

pred elementContains(e:EncodedText,i:Index,p:Nat[N]) ⇔
     validEncoded(e) ∧ decodedLengthBounded(e) ∧
     i < e.length ∧ elementStart(e,i) <= p ∧ p < elementStart(e,i)
     + e.elements[i].count;

pred encodingCoversText(t:Text,e:EncodedText) ⇔ validEncoded(e) ∧
     decodedLengthBounded(e) ∧ decodedLength(e) = N;

pred elementEncodes(t:Text,e:EncodedText,i:Index) ⇔ i < e.length ∧
     ∀p:Index. elementContains(e,i,p) ⇒ t[p] = e.elements[i].symbol;

pred encodes(t:Text,e:EncodedText) ⇔ encodingCoversText(t,e) ∧
     ∀i:Index with i < e.length. elementEncodes(t,e,i);
```

```
54
55      fun runLength(t:Text,start:Index,s:Symbol):Count
56      requires runStartsAt(t,start,s);
57      ensures completeRun(t,start,result,s);
58      = choose c:Count with completeRun(t,start,c,s);
59
60      proc encodeCompressedRun(count:Count,s:Symbol):Element
61      requires count >= 3;
62      ensures result.compressed = true ∧ result.count = count ∧
63      result.symbol = s;
64      {
65          var result : Element = <<compressed: true, count: count,
66              symbol: s>>;
67          return result;
68      }
69
70      proc encodeSingleRun(t:Text,start:Index):EncodedText
71      requires start < N;
72      ensures result.length = 1 ∧ result.elements[0].compressed = false ∧
73      result.elements[0].count = 1 ∧ result.elements[0].symbol = t[start];
74      {
75          var emptyElement:Element = <<compressed: false, count: 0,
76              symbol: 0>>;
77          var elements:Array[N,Element] = Array[N,Element](emptyElement);
78          elements[0] := <<compressed: false, count: 1, symbol: t[start]>>;
79          var result:EncodedText = <<length: 1, elements: elements>>;
80
81          return result;
82      }
83
84      proc encodeDoubleRun(t:Text,start:Index): EncodedText
85      requires start + 1 < N ∧ t[start] = t[start + 1];
86      ensures result.length = 2 ∧ result.elements[0].compressed = false ∧
87      result.elements[0].count = 1 ∧
88      result.elements[0].symbol = t[start] ∧
89      result.elements[1].compressed = false ∧
90      result.elements[1].count = 1 ∧
91      result.elements[1].symbol = t[start + 1];
92
93      {
94          var emptyElement: Element = <<compressed: false, count: 0,
95              symbol: 0>>;
96          var elements: Array[N, Element] = Array[N, Element]
97              (emptyElement);
98          elements[0] := <<compressed: false, count: 1, symbol: t[start]>>;
99          elements[1] := <<compressed: false, count: 1,
100             symbol: t[start + 1]>>;
101
102         var result: EncodedText = <<length: 2, elements: elements>>;
103         return result;
104     }
105
106     proc encodeCompressedRunText(count:Count,s:Symbol):EncodedText
107     requires count >= 3;
108     ensures result.length = 1 ∧ result.elements[0].compressed = true ∧
```

```
    result.elements[0].count = count ∧ result.elements[0].symbol = s;
    {
        var emptyElement:Element = <<compressed: false, count: 0,
            symbol: 0>>;
        var elements:Array[N,Element] = Array[N,Element](emptyElement);
        elements[0] := encodeCompressedRun(count,s);

        var result:EncodedText = <<length: 1, elements: elements>>;
        return result;
    }

    proc encodeOneRun(t:Text,start:Index,count:Count):EncodedText
    requires start < N ∧ count >= 1 ∧ start + count <= N ∧ (count = 1 ∨
     count >= 3 ∨ t[start] = t[start + 1]);
    {
        var result:EncodedText;

        if count = 1 then result := encodeSingleRun(t,start);
        else if count = 2 then result := encodeDoubleRun(t,start);
        else result := encodeCompressedRunText(count,t[start]);

        return result;
    }

    proc appendEncodedText(left: EncodedText,right: EncodedText):
         EncodedText
    requires validEncoded(left) ∧ validEncoded(right) ∧
    decodedLengthBounded(left) ∧ decodedLengthBounded(right)
    ∧ left.length + right.length <= N;
    ensures result.length = left.length + right.length;
    {
        var result: EncodedText = left;

        for var j: Index := 0; j < right.length; j := j + 1 do
        result.elements[result.length + j] := right.elements[j];

        result.length := left.length + right.length;
        return result;
    }

    proc appendRun(e:EncodedText,t:Text,start:Index,count:Count):
         EncodedText
    requires validEncoded(e) ∧ decodedLengthBounded(e) ∧ start < N ∧
    count >= 1 ∧ start + count <= N ∧
    run(t,start,count,t[start]) ∧
    e.length + encodeOneRun(t,start,count).length <= N;
    ensures result.length = e.length +
    encodeOneRun(t,start,count).length;
    {
        var run:EncodedText = encodeOneRun(t,start,count);
        var result:EncodedText = appendEncodedText(e,run);

        return result;
    }

```

```
    proc encodeFirstRun(t:Text,start:Index):EncodedText
    requires runStartsAt(t,start,t[start]);
    ensures result.length >= 1;
    {
        var count:Count = runLength(t,start,t[start]);
        var result:EncodedText = encodeOneRun(t,start,count);

        return result;
    }

    type Position = Nat[N];

    fun toIndex(p:Position):Index
    requires p < N;
    = p;

    pred encodedPrefix(t:Text,start:Position,e:EncodedText) ⇔
        validEncoded(e) ∧ decodedLengthBounded(e) ∧
        start <= N ∧ decodedLength(e) = start;

    multiple proc encodeFrom(t:Text,start:Position,e:EncodedText):
                  EncodedText
    requires encodedPrefix(t,start,e) ∧ (start = N ∨
    runStartsAt(t,start,t[start]));
    ensures result.length >= e.length;
    {
        var result:EncodedText = e;

        if start < N then
        {
            var index:Index = toIndex(start);
            var count:Count = runLength(t,index,t[index]);

            result := appendRun(result,t,index,count);

            if start + count < N then
            result := encodeFrom(t,start + count,result);
        }
        return result;
    }

    proc encode(t:Text):EncodedText
    requires N > 0;
    ensures encodes(t,result);
    {
        var result:EncodedText = encodeFirstRun(t,0);
        var count:Count = runLength(t,0,t[0]);

        if count < N then result := encodeFrom(t,count,result);

        return result;
    }

    pred decodedSymbol(e:EncodedText,i:Index,p:Nat[N],s:Symbol) ⇔
         elementContains(e,i,p) ∧ s = e.elements[i].symbol;
```

```
219
220     pred elementAtPosition(e:EncodedText,i:Index,p:Nat[N]) ⇔
221          elementContains(e,i,p);
222
223     pred decodedAt(e:EncodedText,p:Nat[N],s:Symbol) ⇔
224          ∃i:Index. elementAtPosition(e,i,p) ∧ decodedSymbol(e,i,p,s);
225
226     proc decodeAt(e: EncodedText,p: Nat[N]): Symbol
227     requires validEncoded(e) ∧ decodedLengthBounded(e) ∧
228     p < decodedLength(e);
229     ensures decodedAt(e,p,result);
230     {
231         var result: Symbol;
232
233         choose result: Symbol with decodedAt(e,p,result);
234
235         return result;
236     }
237
238
239     proc decode(e:EncodedText):Text
240     requires validEncoded(e) ∧ decodedLengthBounded(e) ∧
241     decodedLength(e) = N;
242     ensures ∀p:Index. decodeAt(e,p) = result[p];
243     {
244         var result:Text = Array[N,Symbol](0);
245         for var p:Position := 0; p < N; p := p + 1 do
246         result[toIndex(p)] := decodeAt(e,toIndex(p));
247
248         return result;
249     }
250
251     theorem decodeEncode(t:Text) ⇔ decode(encode(t)) = t;
```

A.5 Golomb Coding

```
    val N: Nat;
    val M: Nat;
    val K: Nat;

    type Number = Nat[N];
    type Parameter = Nat[M];
    type Auxiliary = Nat[K];

    pred validParameter(m: Parameter) ⇔ m > 0;

    fun quotient(n: Number, m: Parameter): Nat[N]
    requires validParameter(m); = n / m;

    fun remainder(n: Number, m: Parameter): Nat[N]
    requires validParameter(m); = n - quotient(n,m) * m;

    theorem divisionCorrect(n: Number, m: Parameter)
    requires validParameter(m); ⇔ n = quotient(n,m) * m +
    remainder(n,m);

    theorem remainderCorrect(n: Number, m: Parameter)
    requires validParameter(m); ⇔ remainder(n,m) < m;

    pred validAuxiliary() ⇔ M + 1 <= K;

    fun halfCeil(x: Parameter): Auxiliary
    requires validParameter(x);
    requires validAuxiliary();
    = (x + 1) / 2;

    fun bitLength(m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    = if m = 1 then 0
    else 1 + bitLength(halfCeil(m));

    fun threshold(m: Parameter): Nat[N*N]
    requires validParameter(m);
    requires validAuxiliary();
    = 2 ^ bitLength(m) - m;

    pred powerOfTwo(m: Parameter)
    requires validParameter(m);
    requires validAuxiliary();
    ⇔ 2 ^ bitLength(m) = m;

    fun remainderLength(r: Auxiliary, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m;
    = if powerOfTwo(m) then bitLength(m)
    else if r < threshold(m) then
    bitLength(m) - 1
```

```
    else bitLength(m);

    fun shiftedRemainder(r: Auxiliary, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m;
    requires r >= threshold(m);
    = r + threshold(m);

    type Bit = Nat[1];
    type BitSequence = Array[K, Bit];

    type Codeword = Record[bits: BitSequence,length: Auxiliary];

    fun binaryBit(n: Auxiliary): Bit = n % 2;

    proc binaryStep(n: Auxiliary, i: Auxiliary, offset: Auxiliary,
         a: BitSequence): BitSequence
    requires offset + i < K;
    {
        return a with [offset + i] = binaryBit(n);
    }

    proc binary(n: Auxiliary, l: Auxiliary, offset: Auxiliary,
         a: BitSequence): BitSequence
    requires l <= K;
    requires n < 2 ^ l;
    requires offset + l <= K;
    {
        var currentN: Auxiliary := n;
        var result: BitSequence := a;
        var i: Auxiliary := l;

        while i > 0 do
        {
            result := binaryStep(currentN,i - 1,offset,result);

            currentN := currentN / 2;
            i := i - 1;
        }

        return result;
    }

    fun truncatedValue(r: Auxiliary, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    requires r < m;
    = if powerOfTwo(m) then r
    else if r < threshold(m) then r
    else shiftedRemainder(r,m);

    proc truncatedBinary(r: Auxiliary, m: Parameter, offset: Auxiliary,
         a: BitSequence): BitSequence
    requires validParameter(m);
```

```
    requires validAuxiliary();
    requires r < m;
    requires offset + remainderLength(r,m) <= K;
    {
        return binary(truncatedValue(r,m),remainderLength(r,m),offset,a);
    }

    fun unaryBit(q: Number, i: Auxiliary): Bit
    requires i <= q;
    = if i < q then 1
    else 0;

    proc unaryStep(q: Number, i: Auxiliary, a: BitSequence): BitSequence
    requires i <= q;
    requires i < K;
    {
        return a with [i] = unaryBit(q,i);
    }

    proc unary(q: Number, a: BitSequence): BitSequence
    requires q < K;
    {
        var result: BitSequence := a;
        var i: Auxiliary := 0;

        while i <= q do
        {
            result := unaryStep(q, i, result);
            i := i + 1;
        }

        return result;
    }

    fun unaryLength(q: Number): Auxiliary
    requires q < K;
    = q + 1;

    fun golombLength(n: Number, m: Parameter): Auxiliary
    requires validParameter(m);
    requires validAuxiliary();
    = unaryLength(quotient(n,m)) + remainderLength(remainder(n,m),m);

    proc golomb(n: Number, m: Parameter): Codeword
    requires validParameter(m);
    requires validAuxiliary();
    requires golombLength(n,m) <= K;
    {
        var bits: BitSequence := Array[K,Bit](0);

        bits := unary(quotient(n,m), bits);
        bits := truncatedBinary(remainder(n,m), m,
            unaryLength(quotient(n,m)), bits);
        var codeword: Codeword = <<bits: bits,
            length: golombLength(n,m)>>;
```

```
164
165         return codeword;
166
167     }
168
169     fun decodeUnaryRec(bits: BitSequence, length: Auxiliary,
170         i: Auxiliary, count: Auxiliary): Auxiliary
171     requires i < length;
172     requires count <= i;
173     = if bits[i] = 0 then count
174     else if i + 1 = length then
175     count + 1
176     else
177     decodeUnaryRec(bits,length,i + 1,count + 1);
178
179     fun decodeUnary(bits: BitSequence, length: Auxiliary): Auxiliary
180     requires length > 0;
181     = decodeUnaryRec(bits,length,0,0);
182
183     fun truncatedShort(x: Auxiliary,m: Parameter): Bool
184     requires validParameter(m);
185     requires validAuxiliary();
186     requires x < 2 ^ bitLength(m);
187     = x < threshold(m);
188
189     fun truncatedDecodedValue(x: Auxiliary,b: Bit,m: Parameter):
190         Auxiliary
191     requires validParameter(m);
192     requires validAuxiliary();
193     requires x < 2 ^ bitLength(m);
194     requires x >= threshold(m);
195     = 2 * x + b - threshold(m);
196
197     fun decodeBits(bits: BitSequence,offset: Auxiliary,
198         length: Auxiliary): Auxiliary
199     requires length <= 3;
200     requires offset + length <= K;
201     = if length = 0 then 0
202     else if length = 1 then bits[offset]
203     else if length = 2 then bits[offset] * 2 + bits[offset + 1]
204     else bits[offset] * 4 + bits[offset + 1] * 2 + bits[offset + 2];
205
206     fun decodeTruncatedBinary(bits: BitSequence,m: Parameter,
207         offset: Auxiliary): Auxiliary
208     requires validParameter(m);
209     requires validAuxiliary();
210     requires offset + bitLength(m) <= K;
211     = if powerOfTwo(m) then decodeBits(bits,offset,bitLength(m))
212     else
213     if truncatedShort(decodeBits(bits,offset,bitLength(m) - 1),m) then
214     decodeBits(bits,offset,bitLength(m) - 1)
215     else
216     truncatedDecodedValue(decodeBits(bits,offset,bitLength(m) - 1),
217     bits[offset + bitLength(m) - 1],m);
218
```

```
219     pred validCodeword(codeword: Codeword) ⇔
220     codeword.length > 0 ∧
221     decodeUnary(codeword.bits,codeword.length) <= N;
222
223     fun decode(codeword: Codeword,m: Parameter): Number
224     requires validParameter(m);
225     requires validAuxiliary();
226     requires validCodeword(codeword);
227     requires decodeUnary(codeword.bits, codeword.length) *
228      m +decodeTruncatedBinary(codeword.bits,m,unaryLength(decodeUnary
229      (codeword.bits, codeword.length))) <= N;
230     =
231     decodeUnary(codeword.bits,codeword.length) * m +
232     decodeTruncatedBinary(codeword.bits,m,unaryLength(decodeUnary
233     (codeword.bits,codeword.length)));
234
235     theorem decodeEncode(n: Number, m: Parameter)
236     requires validParameter(m);
237     requires validAuxiliary();
238     requires golombLength(n,m) <= K;
239     ⇔ decode(golomb(n,m),m) = n;
240
241     pred isPrefix(a: Codeword,b: Codeword) ⇔
242          a.length <= b.length  ∧
243          ∀i:Nat[K-1]. i < a.length ⇒  a.bits[i] = b.bits[i];
244     pred prefixFree(a: Codeword,b: Codeword) ⇔
245          ¬isPrefix(a,b) ∧ ¬isPrefix(b,a);
246
247     theorem golombPrefixFree(n1: Number,n2: Number,m: Parameter)
248     requires validParameter(m);
249     requires validAuxiliary();
250     requires n1 ≠ n2;
251     requires golombLength(n1,m) <= K;
252     requires golombLength(n2,m) <= K; ⇔
253     prefixFree(golomb(n1,m),golomb(n2,m));
```

A.6 Shannon-Fano Coding

```
    val N: Nat;
    val F: Nat;

    type Symbol = Nat[N-1];
    type Frequency = Nat[F];
    type Frequencies = Array[N, Frequency];

    pred sorted(f: Frequencies) ⇔
         ∀i:Symbol, j:Symbol. i < j ⇒ f[i] ≥ f[j];

    pred validInput(f: Frequencies) ⇔
         N ≥ 2 ∧ ∀i:Symbol. f[i] > 0 ∧ sorted(f);

    type Total = Nat[N*F];

    pred validSplit(start: Nat[N], end: Nat[N], k: Nat[N]) ⇔
         start < k ∧ k < end;

    fun sumrange(f: Frequencies, start: Nat[N], end: Nat[N]): Total
    requires start <= end; =
    if start = end then
    0
    else
    f[start] + sumrange(f, start+1, end);

    fun splitDifference(f: Frequencies,start: Nat[N],end: Nat[N],
        k: Nat[N]): Total
    requires start < k ∧ k < end;
    requires start <= end; =
    if sumrange(f,start,k) >= sumrange(f,k,end) then
    sumrange(f,start,k) - sumrange(f,k,end)
    else sumrange(f,k,end) - sumrange(f,start,k);

    pred optimalSplit(f: Frequencies,start: Nat[N],end: Nat[N],k: Nat[N])
    ⇔ validSplit(start,end,k) ∧ ∀j:Nat[N]. validSplit(start,end,j) ⇒
    splitDifference(f,start,end,k) ≤ splitDifference(f,start,end,j);

    pred firstOptimalSplit(f: Frequencies,start: Nat[N],end: Nat[N],
         k: Nat[N]) ⇔
    optimalSplit(f,start,end,k) ∧ ∀j:Nat[N].
    validSplit(start,end,j) ∧ j < k ⇒ ¬optimalSplit(f,start,end,j);

    proc findSplit(f: Frequencies,start: Nat[N],end: Nat[N]): Nat[N]
    requires validInput(f);
    requires start + 1 < end;
    ensures firstOptimalSplit(f,start,end,result);
    {
        var ret: Nat[N] := 0;

        for var k:Nat[N] := start+1; k < end ∧ ret = 0; k := k+1 do
        {
            if firstOptimalSplit(f,start,end,k) then ret := k;
        }
```

```
54
55          return ret;
56      }
57
58      type Bit = Nat[1];
59
60      fun splitBit(start: Nat[N],end: Nat[N],k: Nat[N],i: Nat[N]): Bit
61      requires start < k ∧ k < end;
62      requires start <= i ∧ i < end; =
63      if i < k then 1
64      else 0;
65
66      type Code = Array[N, Bit];
67      type CodeLength = Nat[N];
68      type CodeData = Record [code: Code,length: CodeLength];
69
70      proc appendBit(code: Code,length: CodeLength,bit: Bit): Code
71      requires length < N;
72      {
73          var result:Code := code;
74
75          result[length] := bit;
76
77          return result;
78      }
79
80      fun extendCode(data: CodeData,bit: Bit): CodeData
81      requires data.length < N; =
82      data with .code = appendBit(data.code, data.length, bit)
83      with .length = data.length + 1;
84
85      fun extendSplitCode(data: CodeData,start: Nat[N],end: Nat[N],
86          k: Nat[N],i: Nat[N]): CodeData
87      requires start < k ∧ k < end;
88      requires start <= i ∧ i < end;
89      requires data.length < N; = extendCode(data,
90      splitBit(start,end,k,i));
91
92
93      type Codes = Array[N, CodeData];
94
95      pred validCodes(codes: Codes,start: Nat[N],end: Nat[N]) ⇔
96      ∀i:Nat[N-1], j:Nat[N-1]. ((start <= i ∧ i < end) ∧
97      (start <= j ∧ j < end)) ⇒ codes[i].length = codes[j].length;
98
99      pred validCodeRange(codes: Codes,start: Nat[N],end: Nat[N]) ⇔
100     validCodes(codes,start,end) ∧ ∀i:Nat[N-1].
101     (start <= i ∧ i < end) ⇒ codes[i].length <= N - (end - start);
102
103     proc applySplit(codes: Codes,start: Nat[N],end: Nat[N],k: Nat[N]):
104          Codes
105     requires validCodes(codes,start,end);
106     requires start < k ∧ k < end;
107     requires ∀i:Nat[N-1]. (start <= i ∧ i < end) ⇒ codes[i].length < N;
108     ensures ∀i:Nat[N-1]. start <= i ∧ i < end ⇒ result[i] =
```

```
109     extendSplitCode(codes[i],start,end,k,i);
110     ensures ∀i:Nat[N-1]. ¬(start <= i ∧ i < end) ⇒ result[i] = codes[i];
111     ensures validCodes(result,start,k);
112     ensures validCodes(result,k,end);
113     {
114         var result:Codes := codes;
115 
116         for var i:Nat[N] := start; i < end; i := i+1 do
117         {
118             result[i] := extendSplitCode(result[i],start,end,k,i);
119         }
120 
121         return result;
122     }
123 
124     pred validRange(start: Nat[N], end: Nat[N]) ⇔ start < end;
125     pred leafRange(start: Nat[N], end: Nat[N]) ⇔ validRange(start,end)
126          ∧ start + 1 = end;
127 
128     proc encodeRange(f: Frequencies,codes: Codes,start: Nat[N],
129          end: Nat[N]): Codes
130     requires validInput(f);
131     requires validRange(start,end);
132     requires validCodeRange(codes,start,end);
133     {
134         var result:Codes := codes;
135         var k:Nat[N] := 0;
136 
137         if leafRange(start,end) then result := codes;
138         else
139         {
140             k := findSplit(f,start,end);
141             result := applySplit(codes,start,end,k);
142 
143             result := encodeRange(f,result,start,k);
144             result := encodeRange(f,result,k,end);
145         }
146         return result;
147 
148     }
149 
150     proc makeEmptyCodes(): Codes
151     {
152         var emptyCode: Code := Array[N,Bit](0);
153 
154         var emptyCodeData: CodeData = <<code: emptyCode, length: 0>>;
155 
156         var result: Codes := Array[N,CodeData](emptyCodeData);
157 
158         return result;
159     }
160 
161     pred nonEmptyCodes(codes: Codes) ⇔ ∀i:Nat[N-1]. codes[i].length > 0;
162 
163     pred isPrefix(a: CodeData,b: CodeData) ⇔
```

```
164          a.length <= b.length ∧ ∀i:Nat[N-1]. i < a.length ⇒
165          a.code[i] = b.code[i];
166
167     pred prefixFree(codes: Codes) ⇔
168          ∀i:Nat[N-1], j:Nat[N-1]. i ≠ j ⇒ ¬isPrefix(codes[i],codes[j]);
169
170     proc encode(f: Frequencies): Codes
171     requires validInput(f);
172     ensures nonEmptyCodes(result);
173     ensures prefixFree(result);
174     {
175         var codes:Codes := makeEmptyCodes();
176
177         return encodeRange(f,codes,0,N);
178     }
179
180     fun sameCode(a: CodeData, b: CodeData): Bool =
181     a.length = b.length ∧ ∀i:Nat[N-1]. i < a.length ⇒ a
182     .code[i] = b.code[i];
183
184     pred containsCode(codes: Codes, code: CodeData) ⇔
185          ∃i:Symbol. sameCode(codes[i], code);
186
187     proc decodeSymbol(codes: Codes, code: CodeData): Symbol
188     requires code.length > 0;
189     requires containsCode(codes, code);
190     {
191         var result:Symbol := 0;
192
193         for var i:Nat[N] := 0; i < N; i := i+1 do
194         {
195             if sameCode(codes[i], code) then
196             result := i;
197         }
198
199         return result;
200     }
201
202     theorem decodeEncode(f: Frequencies) ⇔
203          validInput(f) ⇒ ∀i:Symbol. decodeSymbol(encode(f),
204          encode(f)[i]) = i;
205
206     theorem encodedCodesAreUnique(f: Frequencies) ⇔
207          validInput(f) ⇒ ∀i:Symbol, j:Symbol. i ≠ j ⇒
208          ¬sameCode(encode(f)[i], encode(f)[j]);
```

A.7 Lempel-Ziv

```
    val N: Nat;
    val A: Nat;

    type Symbol = Nat[A];
    type Input = Array[N, Symbol];

    val S: Nat;
    val L: Nat;

    type Index = Nat[N-1];

    fun searchStart(p: Index): Index = if p < S then 0 else p - S;

    pred inSearchBuffer(q: Index, p: Index) ⇔
         searchStart(p) <= q ∧ q < p;

    pred validMatchLength(p: Index, q: Index, k: Nat[N]) ⇔ k + 1 < L ∧
         p + k < N ∧ q + k < N;

    proc matchLength(input: Input, q: Index, p: Index): Nat[N]
    requires inSearchBuffer(q, p);
    {
        var k: Nat[N] := 0;

        while (validMatchLength(p, q, k) ∧ input[q + k] = input[p + k])
        do
        {
            k := k + 1;
        };

        return k;
    }

    type Match = Record [q: Index,length: Nat[N]];

    pred isLongestMatch(input: Input, p: Index, q: Index, length: Nat[N])
    ⇔ length = 0 ∨ (inSearchBuffer(q, p)
    ∧ forall r: Index with inSearchBuffer(r, p).matchLength(input, r, p)
     <= length);

    pred isLeftmostLongestMatch(input: Input, p: Index, q: Index,
    length: Nat[N]) ⇔
    length = 0 ∨ (inSearchBuffer(q, p) ∧ forall r: Index with
    inSearchBuffer(r, p).
    (matchLength(input, r, p) = length ∧ r < q) ⇒ false);

    proc bestMatch(input: Input, p: Index): Match
    ensures isLongestMatch(input, p, result.q, result.length);
    ensures isLeftmostLongestMatch(input, p, result.q, result.length);
    {
        var best: Match = <<q: 0, length: 0>>;


```

```
        var q: Index := p;

        while (q > searchStart(p)) do
        {
            q := q - 1;

            var length: Nat[N] := matchLength(input, q, p);

            if (length >= best.length) then
            {
                best.q := q;
                best.length := length;
            }
        };

        return best;
    }

    type Token = Record [offset: Nat[N],length: Nat[N],
         nextSymbol: Symbol,isEnd: Bool];

    pred hasNextSymbol(p: Index, length: Nat[N]) ⇔ p + length < N;

    proc makeToken(input: Input, p: Index): Token
    {
        var best: Match = bestMatch(input, p);

        var token: Token = <<offset: 0, length: 0, nextSymbol: 0,
            isEnd: false>>;

        if (best.length = 0) then
        {
            token.offset := 0;
        }
        else
        {
            token.offset := p - best.q;
        }

        token.length := best.length;

        if (hasNextSymbol(p, best.length)) then
        {
            token.nextSymbol := input[p + best.length];
            token.isEnd := false;
        }
        else
        {
            token.nextSymbol := 0;
            token.isEnd := true;
        }

        return token;
    }

```

```
    type Position = Nat[N];
    type Tokens = Array[N, Token];
    type Encoding = Record[tokens: Tokens,count: Nat[N]];
    type PositionResult = Record [position: Position,valid: Bool];

    pred validToken(token: Token, p: Position) ⇔
         ((token.length = 0 ∧ token.offset = 0) ∨
         (token.length > 0 ∧ token.offset > 0 ∧
         token.offset <= p ∧ p - token.offset + token.length <= N)) ∧
         ((token.isEnd ∧ p + token.length = N) ∨
          (token.isEnd = false ∧ p + token.length < N));

    pred validEndPosition(encoding: Encoding, i: Nat[N]) ⇔
         i < encoding.count ∧
         (i = encoding.count - 1 ∨ encoding.tokens[i].isEnd = false);

    fun tokenPosition(encoding: Encoding, i: Nat[N]): PositionResult
    requires encoding.count > 0; =
    if i = 0 then <<position: 0, valid: true>>
    else
    if tokenPosition(encoding, i - 1).valid then
        if tokenPosition(encoding, i - 1).position +
        encoding.tokens[i - 1].length + 1 <= N then
        <<position: tokenPosition(encoding, i - 1).position +
        encoding.tokens[i - 1].length + 1, valid: true>>
        else <<position: 0, valid: false>>
    else <<position: 0, valid: false>>;

    pred encodingReachesEnd(encoding: Encoding) ⇔
         encoding.count > 0 ∧
         ((encoding.tokens[encoding.count - 1].isEnd ∧
         tokenPosition(encoding, encoding.count - 1).position +
         encoding.tokens[encoding.count - 1].length = N) ∨
         (encoding.tokens[encoding.count - 1].isEnd = false ∧
         tokenPosition(encoding, encoding.count - 1).position +
         encoding.tokens[encoding.count - 1].length + 1 = N));

    pred validEncodingSpec(encoding: Encoding) ⇔
         if encoding.count = 0 then false
         else (forall i: Nat[N] with i < encoding.count.
         tokenPosition(encoding, i).valid ∧
         validToken(encoding.tokens[i],
         tokenPosition(encoding, i).position) ∧
         validEndPosition(encoding, i)) ∧ encodingReachesEnd(encoding);

    proc validEncoding(encoding: Encoding): Bool
    requires encoding.count > 0;
    ensures result ⇔ validEncodingSpec(encoding);
    {
        var i: Nat[N] := 0;
        var valid: Bool := true;

        while (i < encoding.count ∧ valid) do
        {
            var positionResult: PositionResult =
```

```
                tokenPosition(encoding, i);

            if positionResult.valid then
            {
                if (validToken(encoding.tokens[i],
                    positionResult.position) ∧
                    validEndPosition(encoding, i)) then
                {
                    i := i + 1;
                }
                else
                {
                    valid := false;
                }
            }
            else
            {
                valid := false;
            }
        };
        return valid ∧ encodingReachesEnd(encoding);
    }

    proc encode(input: Input): Encoding
    ensures validEncoding(result);
    {
        var emptyToken: Token = <<offset: 0, length: 0, nextSymbol: 0,
            isEnd: false>>;

        var emptyTokens: Tokens = Array[N, Token](emptyToken);

        var result: Encoding = <<tokens: emptyTokens, count: 0>>;

        var p: Position := 0;

        while (p < N) do
        {
            var token: Token = makeToken(input, p);

            result.tokens[result.count] := token;
            result.count := result.count + 1;

            if (token.isEnd) then
            {
                p := N;
            }
            else
            {
                p := p + token.length + 1;
            }
        };

        return result;
    }

```

```
    type Output = Array[N, Symbol];

    proc decode(encoding: Encoding): Output
    requires encoding.count > 0;
    requires validEncoding(encoding);
    {
        var emptyOutput: Output = Array[N, Symbol](0);

        var output: Output := emptyOutput;

        var p: Position := 0;
        var i: Nat[N] := 0;

        while (i < encoding.count) do
        {
            var token: Token = encoding.tokens[i];

            var source: Position := p - token.offset;
            var j: Nat[N] := 0;

            while (j < token.length) do
            {
                output[p + j] := output[source + j];
                j := j + 1;
            };

            if (token.isEnd) then
            {
                p := N;
            }
            else
            {
                output[p + token.length] := token.nextSymbol;
                p := p + token.length + 1;
            };

            i := i + 1;
        };

        return output;
    }

    pred encodingCorrect(input: Input) ⇔ decode(encode(input)) = input;

    theorem decodeEncode(input: Input) ⇔ decode(encode(input)) = input;
```

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Bachelorarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

_____

Thomas Michlmayr