Noname manuscript No.
(will be inserted by the editor)

A Library for Formalization of Linear Error-correcting Codes

Reynald Affeldt · Jacques Garrigue · Takafumi Saikawa

the date of receipt and acceptance should be inserted later

Abstract Error-correcting codes add redundancy to transmitted data to ensure reliable communication over noisy channels. Since they form the foundations of digital communication, their correctness is a matter of concern. To enable trustful verification of linear error-correcting codes, we have been carrying out a systematic formalization in the Coq proof-assistant. This formalization includes the material that one can expect of a university class on the topic: the formalization of well-known codes (Hamming, Reed-Solomon, Bose-Chaudhuri-Hocquenghem) and also a glimpse at modern coding theory. We demonstrate the usefulness of our formalization by extracting a verified decoder for low-density parity-check codes based on the sum-product algorithm. To achieve this formalization, we needed to develop a number of libraries on top of Coq's Mathematical Components. Special care was taken to make them as reusable as possible so as to help implementers and researchers dealing with error-correcting codes in the future.

1 Towards a Formal Coding Theory

Error-correcting codes are a well-established field of applied mathematics. They are a necessary technology to ensure reliable storage and communication of information.

This is a pre-print of an article published in the Journal of Automated Reasoning. The final authenticated version is available online at: `https://doi.org/10.1007/s10817-019-09538-8`.

R. Affeldt
National Institute of Advanced Industrial Science and Technology (AIST)
E-mail: reynald.affeldt at aist.go.jp

J. Garrigue
Nagoya University
E-mail: garrigue at math.nagoya-u.ac.jp

T. Saikawa
Nagoya University
E-mail: tscompor at gmail.com

They need to be efficient from two perspectives: they are expected to be robust, recovering from errors using only limited extra storage; they are also expected to be fast, using clever encoding and decoding algorithms on the fly. For the first perspective, information theory provides a framework to prove stochastically how resilient codes are to errors, and their theoretical limits. For the second one, a wide range of algorithms have been developed for various applications, whose correctness relies on the mathematical properties of the codes.

To illustrate the various applications of error-correcting codes, let us walk through the concrete codes we will be dealing with in this paper. Hamming codes are the most famous error-correcting codes. Though they were developed at the time of punch cards, they are still in use in modern memory hardware. Reed-Solomon codes are used in electronic storage devices (compact disc, digital versatile disc, hard disk drive, solid-state drive, flash memory) to correct burst errors associated with media effects, but also in satellite communication and Quick Response (QR) codes. The applications of Bose-Chaudhuri-Hocquenghem codes are similar to Reed-Solomon codes. Satellite communications and cellular phones have recently been using more modern error-correcting codes such as low-density parity-check codes. Nowadays, low-density parity-check codes are commonly used for data storage (hard disk and solid-state drives), wireless communications (e.g., IEEE 802.16e, IEEE 802.11n), video broadcasting (DVB-S2, 8K Ultra-high-definition television broadcasting), 10GBASE-T Ethernet, etc. The main reason for the existence of many codes is the variety of applications with different requirements such as the accepted rate of errors or the required speed of execution.

The development of error-correcting codes and their analysis rely on a wide range of mathematical theories. Classical error-correcting codes (such as Hamming, Reed-Solomon, Bose-Chaudhuri-Hocquenghem) use linear algebra. The construction of more recent codes (such as low-density parity-check codes) is using insights from graph theory in addition to linear algebra. All these codes belong to the class of linear error-correcting codes. More generally, the qualitative evaluation of error-correcting codes uses probability theory and information theory.

Given the importance of the applications of error-correcting codes and the variety of mathematical theories at work, the validity of proofs is a matter of concern. The development of codes based on graph theory provides a good example of such a concern. Low-density parity-check codes (which are based on graphs) were discovered in the early sixties [Gallager, 1962] but lacked a practical implementation. Interest about codes based on graphs was renewed in the nineties with the invention of Turbo codes. Even though good performance could be observed empirically, they have long been lacking rigorous mathematical proofs [Gowers, 2008, VII.6]. Low-density parity-check codes regained attention in the nineties but then again there have long been only partial answers about the quality of decoders [Richardson and Urbanke, 2001, §I]. This situation has motivated a new body of works known as *modern coding theory*. Research in modern coding theory has actually been so active that today correctness guarantees for cutting-edge error-correcting codes are scattered in scientific publications. To quote the standard textbook on modern coding theory, "[t]here are nearly as many flavors of iterative decoding systems—and graph-

ical models to denote them—as there are researchers in the field" [Richardson and Urbanke, 2008, Preface].

Our goal is to provide a formalization of the theory of error-correcting codes that makes it possible to tackle formalization challenges such as modern coding theory and support the formal verification of concrete implementations. Indeed, a researcher or an implementer willing to perform formal verification of a concrete error-correcting code first needs as a prerequisite to provide a formal specification of what a code is supposed to achieve. But, to the best of our knowledge, no such formal specification is available yet. The main reason might be that the formalization of error-correcting codes is a difficult undertaking. The first difficulty is the construction of a comprehensive library that encompasses the required mathematical theories (probabilities, graphs, linear algebra, etc.). The second difficulty is that monographs and research papers on error-correcting codes do not provide details for the non-expert reader. Meanwhile, teaching material does not lend itself well to formalization, in comparison with well-structured algebra textbooks. In practice, one often finds prose definitions that look incomplete without the accompanying examples, algorithms written in prose, hypotheses about the model that appear during the course of proofs, etc.

Still, there is previous work that we can take advantage of to formalize error-correcting codes. The MATHCOMP library [Mahboubi and Tassi, 2016] provides big operators to formalize combinatorial results, a substantial formalization of linear algebra, and tools to reason about graphs. The formalization of the foundational theorems of information theory [Affeldt et al, 2014] provides us with basic definitions about communication channels and probabilities.

The work we present in this paper is an attempt at a systematic and reusable formalization of error-correcting codes inside a proof-assistant. Our contributions can be summarized as follows:

- Our formalization covers the material that one can expect of a university class: basic definitions of linear error-correcting codes, specification of decoders, concrete applications to well-known codes (Hamming, Reed-Solomon, Bose-Chaudhuri-Hocquenghem), and a glimpse at modern coding theory (with the formal verification of sum-product decoding). This leads us to augment textbook definitions with their implicit assumptions and complete partial proofs. This also includes a formal account of notational practice, in particular for the *summary operator* used in modern coding theory (see Sect. 8.2). Our formalization can therefore be regarded as a firm assessment of the foundations of the theory.
- Our formalization features reusable libraries. For example, we provide libraries for the generic layers of linear codes and cyclic codes and use these libraries to prove more concrete codes. In particular, this results in sharing a large part of the proofs for Reed-Solomon and Bose-Chaudhuri-Hocquenghem codes. For modern coding theory, we needed libraries to reason about *Tanner graphs*, which are bipartite graphs used to represent codes (see Sect. 8.1). These libraries contain several technical contributions. For example, we needed to instrument MATHCOMP's Euclid's algorithm for computing the GCD of polynomials in order to decode polynomial codes (see Sect. 5.2).

– We demonstrate that our formalization can be used to develop formally-verified decoders. Our illustrative experiment consists in the extraction of OCaml code for the sum-product algorithm to decode low-density parity-check codes (see Sect. 9.2).

The significance of these contributions is two-fold. First, this means that this library provides a sound basis to understand existing error-correcting codes and formalize new ones, eventually allowing to solve the two gaps we described above, both about the reliability of mathematical proofs in this field, and the verification of implementations. Second, this provides a concrete example of using verification to "sanitize" (a part of) a field, i.e., ensure that all hypotheses are explicitly given, with no hidden assumptions, filling in the subtle details of many proofs and definitions. This indicates the possibility of applying the well-developed method of formalized mathematics to a broader domain of applied mathematics.

Outline We first start with a background section (Sect. 2) on the MATHCOMP library and information theory. We then explain our formalization by interleaving the presentation of the resulting library with concrete use-cases.

We formalize basic definitions and generic results about linear error-correcting codes in Sect. 3. We illustrate them with a formalization of Hamming codes in Sect. 4. In particular, we provide a concrete encoder and decoder and express the error rate in terms of a closed formula.

Second, we formalize in Sect. 5 basic definitions and results about codes whose codewords are better seen as polynomials. This includes in particular the Euclidean algorithm for decoding. We apply the resulting library to Reed-Solomon codes in Sect. 6 and to Bose-Chaudhuri-Hocquenghem codes in Sect. 7 where we explain in particular their decoding.

Third, we formalize in Sect. 8 the basics of modern coding theory. In particular, we formalize the key properties of the sum-product algorithm, the standard algorithm for efficient decoding of low-density parity-check codes. We apply our library of modern coding theory in Sect. 9. First, we formalize sum-product decoding in the simple case of the binary erasure channel. Second, we formalize a concrete implementation of the sum-product algorithm for the binary symmetric channel, making our work the first formal verification of a decoding algorithm for a recent class of error-correcting codes.

We review related work in Sect. 10, discuss future work in Sect. 11, and conclude in Sect. 12.

We do not assume prior knowledge of error-correcting codes and tried to make this paper as self-contained as possible. Yet, this cannot be a replacement for a textbook on error-correcting codes [MacWilliams and Sloane, 1977; McEliece, 2002; Richardson and Urbanke, 2008], in particular when it comes to intuitions.

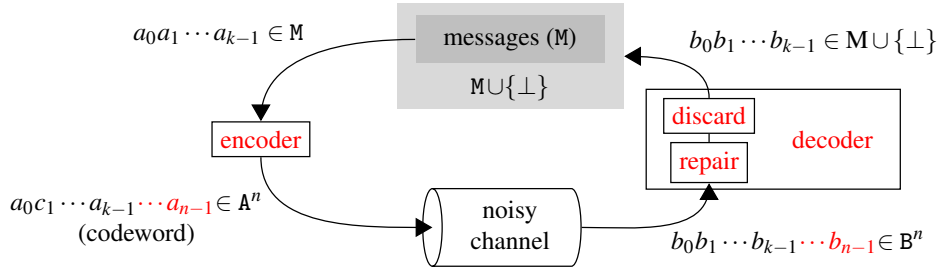

Fig. 1: The setting of Error-correcting Codes

2 Background

2.1 The MATHCOMP Library

The MATHCOMP library [Mahboubi and Tassi, 2016] is a collection of formalized mathematical definitions and lemmas built on top of the COQ proof-assistant [The Coq Development Team, 2018] that was originally developed to formalize the odd order theorem [Gonthier et al, 2013]. It contains in particular a formalization of linear algebra with matrices and polynomials, which are at the heart of the theory of error-correcting codes. In this paper, we use a number of objects from the MATHCOMP library. It is not our purpose to explain them in details. For quick reference, we have extracted from the documentation succinct explanations about most notations in Tables 1, 2, and 3. Other notations are either self-explanatory or will be explained when introduced.

2.2 Formalization of Codes in Information Theory and Probabilities

In Sect. 2.2.1, we explain the basic setting of error-correcting codes, including how we model probabilities, using our formalization from previous work [Affeldt et al, 2014]. In Sect. 2.2.2, we extend this framework with a formalization of aposteriori probabilities.

2.2.1 Channels and Codes in Information Theory

The most generic definition of a code is as a *channel code*: a pair of encoder/decoder functions with a finite type `M` for the message pieces to be encoded. Encoded message pieces (*codewords*) are represented by row-vectors over a finite alphabet `A` (denoted by `'rV[A]_n`). Codewords go through a noisy channel. The decoder (that may fail) is fed with the outputs of the channel that are also represented by row-vectors (possibly over a different[1] alphabet `B`):

```
Definition encT := {ffun M → 'rV[A]_n}.
Definition decT := {ffun 'rV[B]_n → option M}.
```

[1] The input and output alphabets are not the same for example in the case of the *binary erasure channel* that replaces some bits with an *erasure*.

```
Record code := mkCode { enc : encT ; dec : decT }.
```

The action of the encoder on messages and of the decoder on the channel outputs is depicted in Fig. 1. Decoding is furthermore decomposed into a first phase that "repairs" codewords altered by noise, and a "discard" phase that removes the redundant symbols introduced by the encoder. We come back to this aspect of the formalization later on.

We represent probability distributions by the type `dist`. It is a dependent record with a function from some finite type to non-negative reals and a proof that its outputs sum to one.

```
Record dist := mkDist {
  pmf :> A →R+ ; (* →R+ is a notation *)
  pmf1 : \sum_(a in A) pmf a == 1 :> R}.
```

The first field is interpreted as a coercion from probability distributions to functions thanks to the notation `:>`. In the second field, the notation[2] `\sum_(i in E) e i` sums the reals `e i` for all `i` in `E`. The notation `==` is for Boolean equality (see Table 3) whose both sides are interpreted as reals thanks to the notation `:> R`. Hereafter, `{dist T}` is a notation for the type of distributions `dist T` that additionally hides a function that checks whether `T` is a finite type.

We represent probability events by finite sets. Given a distribution `P`, the probability of an event `{set A}` is formalized as follows:

```
Definition Pr P (E : {set A}) := \sum_(a in E) P a.
```

A (discrete) noisy channel is modeled by a *stochastic matrix*: a matrix whose rows form probability distributions. In Coq, we formalize such a matrix by a function from the input alphabet `A` to probability distributions over the output alphabet `B`, hence the following notation:

```
Notation "'Ch(A, B)" := (A → dist B).
```

Traditionally, channels are ranged over by W.

The most common type of channel is the *binary symmetric channel*, where `A` and `B` are both the set $\{0,1\}$, and whose only possible error is the flipping of a bit, with probability p. That is $W = \begin{bmatrix} 1-p & p \\ p & 1-p \end{bmatrix}$.

Communication of n characters is thought of as happening over the nth extension of the channel, i.e., a channel whose input and output are row-vectors. In this paper, we deal with *discrete memoryless channels* (DMCs). It means that the output probability of a character does not depend on preceding inputs. In this case, the definition of the nth extension of a channel `W` boils down to a probability mass function that associates to an input vector `x` the following distribution of output vectors:

```
Definition f (x : 'rV[A]_n) :=
  [ffun y : 'rV[B]_n ⇒ \prod_(i < n) W '(y_i | x_i)].
```

[2] The MathComp library provides the same notation for sums, it is generic but cannot be used directly with the reals from the Coq standard library. One first needs to show that they satisfy the basic properties of appropriate algebraic structures and declare the latter as `Canonical`. We have chosen not to do that because it makes automatic tactics for reals such as `field` and `lra` inoperative. Our notation is therefore just a specialization of the generic notation provided by MathComp.

where `x_i` represents the `i`th element of the vector `x` and `W '(b | a)` is a Coq notation for `W a b`; it is to match the traditional pencil-and-paper writing $W(b|a)$ of the probability of having b out of the channel W knowing that the input was a. Hereafter, the Coq notation `W ''(y | x)` (in pencil-and-paper proofs, one overloads the notation $W(y|x)$) is the probability for the DMC of `W` that an input `x` is output as `y`.

Finally, the quality of a code `c` for a given channel `W` is measured by its *error rate*. It is defined using the *conditional error rate* (notation: `e(W, c)`):

```
Definition ErrRateCond (W : 'Ch(A, B)) c m :=
  Pr (W ''(| enc c m)) (preimC (dec c) m).
```

The notation `W ''(| enc c m)` above is for the distribution with probability mass function `fun y ⇒ W ''(y | enc c m)`, i.e., the distribution of outputs corresponding to the codeword `(enc c m)` sent over the DMC of `W`. The set `preimC (dec c) m` is the complement of the pre-image of `m`, i.e., the set of outputs `y` that do not decode to `m` . The error rate is defined by the average of the conditional error rates over the set of messages (notation: `echa(W, c)`):

```
Definition CodeErrRate (W : 'Ch(A, B)) c :=
  1 / INR #|M| * \sum_(m in M) e(W, c) m.
```

The function `INR` injects natural numbers into real numbers.

2.2.2 Aposteriori Probability

Probabilities are used to specify the correctness of decoders that use probabilities, such as the sum-product algorithm (see Sect. 8.3).

First, we define the notion of *aposteriori probability*: the probability that an input was sent knowing that some output was received. It is defined via the Bayes rule from the probability that an output was received knowing that some input was sent. For an input distribution P and a channel W, the aposteriori probability[3] of an input x given the output y is:

$$P^W(x|y) \stackrel{def}{=} \frac{P(x)W(y|x)}{\sum_{x' \in A^n} P(x')W(y|x')}.$$

We formalize aposteriori probabilities with the following probability mass function `f`:

```
(* Module PosteriorProbability *)
Definition den := \sum_(x in 'rV_n) P x * W ''(y | x).
Definition f x := P x * W ''(y | x) / den.
```

The function `f` is indeed the probability mass function of a distribution because its denominator is strictly positive and that it sums to 1. Proving that the denominator is not zero requires an hypothesis. This additional hypothesis should not be regarded as a technical hindrance: in fact, it expresses the natural condition that, since `y` was received, then necessarily a suitable `x` (i.e., such that `P x` and `W ''(y | x)` are both not equal to `0`) was sent beforehand. The denominator being non-zero is thus equivalent to the `receivable` condition:

[3] This definition is specialized to communication over a noisy channel and is sufficient for our purpose in this paper. It can also be recast in a more general setting with conditional probabilities using an appropriate joint distribution [Affeldt et al, 2019].

```
Definition receivable y := [∃ x, (P x ≠ 0) && (W ``(y | x) ≠ 0)].
```

In Coq, we denote aposteriori probabilities by `P ``^^ W (x | y)` where `y` is a vector together with the proof that it is `receivable`.

When P is a uniform distribution over a set C of row-vectors, one can show that for any $x \in C$, $P^W(x|y) = KW(y|x)$ for an appropriate constant K (for any channel W, provided that y is receivable).

Second, the probability that the n_0th bit of the input is set to b (0 or 1) given the output y is defined by the *marginal aposteriori probability* (K' is chosen so that it is indeed a probability):

$$P^W_{n_0}(b|y) \stackrel{def}{=} K' \sum_{x \in \mathbb{F}_2^n \; x_{n_0}=b} P^W(x|y)$$

In Coq, we will denote this probability by `P '_ n0 ``^^ W (b | y)` where `y` is a vector together with the proof that it is `receivable`. See our implementation [Infotheo, 2019] for complete formal definitions.

3 A Formal Setting for Linear ECCs

3.1 Linear ECC as a Vector Space

The simplest mathematical definition of a linear error-correcting code (hereafter, linear ECC) is as a finite dimensional vector space over some finite field, whose vectors represent potentially erroneous data, and a specified subspace of correct data corresponding to codewords, so that a linear combination of codewords is still a codeword. We express codewords as concrete row-vectors, and their set is called the *codebook*. In the MathComp library, `vectType R` is a type for finite dimensional vector spaces over `R`, and `{vspace vT}` is the type of subspaces of `vT`, where `vT` has a `vectType` structure. We therefore define linear ECCs as the following type `Lcode0.t` (inside a module `Lcode0` to have a namespace):

```
(* Module Lcode0 *)
Definition t (F : finFieldType) n := {vspace 'rV[F]_n}.
```

The natural `n` is referred to as the *length* of the code. The *dimension* of the code is the dimension of the subspace. When `F` is the finite field $\mathbb{F}_2$, we talk about binary codes.

In practice, one often defines a linear ECC using its *parity-check matrix*, i.e., the matrix whose rows correspond to the checksum equations that codewords fulfill. More precisely, a linear ECC can be defined as the kernel of the *syndrome* function, i.e., the function $y \mapsto (Hy^T)^T$ where H is the parity-check matrix:

```
Definition syndrome (H : 'M[F]_(m, n)) (y : 'rV_n) := (H *m y^T)^T.
```

We can use the fact that this function is linear (proof term `hom_syndrome` below) to define the corresponding ECC using the `lker` function of the MathComp library:

```
Definition kernel : Lcode0.t F n := lker hom_syndrome.
```

So, codewords become vectors whose syndrome is 0 and when the parity-check matrix is full rank, the dimension of the code is $n - m$ (where m is the number of rows of the parity-check matrix):

```
Lemma dim_kernel (Hm : \rank H = m) (mn : m ≤ n) : \dim kernel = n - m.
```

Finally, we formalize the notion of *minimum distance* of a code. This is the minimum distance between any two codewords. For this definition to make sense, it is important to ensure that the code has at least two codewords, which is often implicit in textbooks. The distance in question is the *Hamming distance*: `dH x y` is the number of different elements between the vectors `x` and `y`. It is best formalized using the *Hamming weight* `wH`, the number of non-zero elements in a vector:

```
Definition wH v := count (fun x ⇒ x ≠ 0) (tuple_of_row v).
Definition dH u v := wH (u - v).
```

A code has at least two codewords when it is not *trivial*, i.e., it is not reduced to the singleton with the null vector (this is sufficient because of linearity):

```
Definition not_trivial := ∃ cw, (cw ∈ C) && (cw ≠ 0).
```

We formally define the minimum distance by using the functions `xchoose` and `arg_min` from MATHCOMP. Given a proof of existence, `xchoose` returns a witness. We use it to define a non-zero codeword from a proof term (`C_not_trivial`) that establishes that `C` is a non-trivial linear ECC:

```
Definition non_0_cw := xchoose C_not_trivial.
```

Given a predicate `P`, an element `a` that satisfies `P`, and a natural-valued function `f`, `arg_min` returns an element that satifies `P` and minimizes `f`. We use it to define a non-zero codeword with minimum weight, whose weight defines the *minimum distance* of the code:

```
Definition min_wH_cw :=
  arg_min non_0_cw [pred cw | (cw ∈ C) && (wH cw ≠ 0)] (@wH F n).
Definition min_dist := wH min_wH_cw.
```

The length, the dimension, and the minimum distance of a code are important to discuss its quality. In particular, given a length and a dimension, it is better for a code to have the largest possible minimum distance, so that two codewords are more easily told apart. Such codes are called *maximum-distance separable* and are defined as follows:

```
Definition maximum_distance_separable := (min_dist == n - \dim C + 1).
```

This is because the length, the dimension, and the minimum distance of a code are related by the *singleton bound*:

```
Lemma singleton_bound : min_dist ≤ n - \dim C + 1.
```

Section 6 provides Reed-Solomon codes as an example of maximum-distance separable codes.

3.2 Linear ECCs with Coding and Decoding Functions

In practice, a linear ECC is not only a vector space but also a pair of encoding and decoding functions to be used with a channel (see Fig. 1). We found it useful to isolate a set of abstract requirements about the encoder and the decoder in order to formalize generic lemmas about decoding. These requirements correspond to hypotheses about the setting of ECCs that are not emphasized (not to say implicit) in textbooks. In the following, we assume a channel with input alphabet `A` and output alphabet `B`.

Encoder Let `C` be a linear ECC of length `n` over a finite field `A` (as defined in Sect. 3.1) and let `M` be a set of messages (`M` has type `finType` which is a type with finitely many elements). An encoder is an injective function from `M` whose image is a subset of `C`:

```
(* Module Encoder *)
Record t (C : Lcode0.t A n) (M : finType) : Type := mk {
  enc :> encT A M n ;
  enc_inj : injective enc ;
  enc_img : enc @: M \subset C }.
```

Recall that `encT` is an abbreviation for `{ffun M → 'rV[A]_n}` (see Sect. 2.2.1).

Decoder A decoder is a function that takes some channel output and returns a message from which this output could have arisen. A decoder can possibly fail. We gave it the type `decT`, an abbreviation for `{ffun 'rV[B]_n → option M}` in Sect. 2.2.1. Formally, it actually helps to be more precise and to decompose decoding in two phases: a first phase that "repairs" the channel output by turning it into a valid codeword, and a second phase that "discards" the checksum-part of the codeword (recall Fig. 1). Since the second phase is conceptually easy, it is often left implicit in textbooks. In our formalization, a decoder is therefore the composition of a *repair* function and a *discard* function, resp. of types `repairT` and `discardT`:

```
Definition repairT (B A : finType) n := {ffun 'rV[B]_n → option 'rV[A]_n}.
Definition discardT (A : finType) n (M : finType) := 'rV[A]_n → M.
```

This leads us to the following definition of a decoder:

```
(* Module Decoder *)
Record t (C : Lcode0.t A n) (M : finType) : Type := mk {
  repair :> repairT B A n ;
  repair_img : oimg repair \subset C ;
  discard : discardT A n M ;
  dec : decT B M n := [ffun x ⇒ omap discard (repair x)] }.
```

where `oimg` returns the set-image of a (partial) function.

Linear ECC with Encoder and Decoder A linear ECC with coding and decoding functions consists of a vector space with an encoder and a decoder that are *compatible*, i.e., such that the discard function cancels the encoder function on the set of codewords:

```
(* Module Lcode *)
Record t : Type := mk {
  lcode0_of :> Lcode0.t A n ;
  enc : Encoder.t lcode0_of M ;
  dec : Decoder.t B lcode0_of M ;
  compatible : cancel_on lcode0_of (Encoder.enc enc) (Decoder.discard dec) }.
```

3.3 Example: Repetition Codes and Linear ECCs in Systematic Form

Let us illustrate the definitions of the previous section with repetition codes and introduce the presentation in terms of *systematic form*.

The r-repetition code encodes one bit by replicating it r times. It has therefore two codewords: $\overbrace{00\cdots0}^{r}$ and $\overbrace{11\cdots1}^{r}$. The parity-check matrix can be defined as $H = [A \,||\, 1]$ where A is the column vector of $r-1$ 1's, and 1 the identity matrix. The reader can check that the equation $Hc^T = 0$ corresponds to the set of equations $c_0 \oplus c_{i+1} = 0$, so that we obtain the desired two codewords.

Decoding codewords from a repetition code can be achieved by *majority vote*, i.e., by returning the codeword $bb\cdots b$ if there is a majority of bits b and fail otherwise:

```
Definition majority_vote r (s : seq 'F_2) : option 'rV['F_2]_r :=
  let cnt := N(1 | s) in
  if r./2 < cnt then Some (const_mx 1)
  else if (r./2 == cnt) && ¬ odd r then None
  else Some 0.
```

This function actually just repairs a codeword that was altered. To complete decoding, one still needs to discard the superfluous $r-1$ bits.

When the parity-check matrix of a linear ECC is put in the form $[A \,||\, 1]$, it is said to be in systematic form and A is called the *check symbol matrix*. The advantage of a parity-check matrix in systematic form is that the corresponding encoder can be simply defined as the matrix multiplication by $G = \left[1 \,||\, -A^T\right]$ where G is called the *generator matrix*. More generally, let A be a $(n-k)\times k$-matrix, $H = [A \,||\, 1]$ and $G = \left[1 \,||\, -A^T\right]$. Then H is the parity-check matrix of a code of length `n` and dimension `k` with the (injective) encoding function $x \mapsto x \times G$. Last, the discard operation can be performed by multiplication by the matrix $[1 \,||\, 0]^T$. The systematic form is illustrated by repetition codes in our implementation [Infotheo, 2019, file `repcode.v`] and with Hamming codes in Sect. 4.4.

3.4 The Variety of Decoding Procedures

There exist several strategies to decode the channel output. We here provide formal definitions and basic lemmas for the most common ones.

Minimum Distance decoding chooses the closest codeword in terms of Hamming distance. Let us assume a set of codewords `C` and a repair function `f` (of type `repairT`, see Sect. 3.2). When `f` repairs an output `y` to a codeword `x`, then there is no other codeword `x'` that is closer to `y`:

```
Definition MD_decoding :=
  ∀ y x, f y = Some x → ∀ x', x' ∈ C → dH x y ≤ dH x' y.
```

The main property of Minimum Distance decoding is that it can correct $\lfloor \frac{d_{\min}-1}{2} \rfloor$ errors where $d_{\min}$ is the minimum distance (defined in Sect. 3.1). Let `f` be a repair function that implements Minimum Distance decoding and whose image is a subset of a linear ECC `C` that is not trivial. Then we can show that `f` can decode up to `mdd_err_cor` errors:

```
Definition mdd_err_cor C_not_trivial := (min_dist C_not_trivial).-1./2.
Lemma mddP x y : f y ≠ None → x ∈ C →
  dH x y ≤ mdd_err_cor C_not_trivial → f y = Some x.
```

Bounded Distance decoding decodes a message to its original codeword as long as it does not deviate more than a given bound. Let `f` be a repair function, `C` a linear ECC, and `t` a bound. The function `f` implements Bounded Distance decoding when the following holds (notation: `t.-BDD(C, f)`):

```
Definition BD_decoding t :=
  ∀ c e, c ∈ C → wH e ≤ t → f (c + e) = Some c.
```

Maximum Likelihood decoding decodes to the codeword that is the most likely to have been sent according to the definition of the channel. In other words, given C an ECC, a Maximum Likelihood decoder is such that its repair function f satisfies $W(y|f(y)) = \max_{x' \in C} W(y|x')$ for any received message y. Observe that this definition is talking about messages y that are *receivable*, i.e., that have a non-zero probability of having been sent and of going through the channel. This is an example of implicit assumption on which textbooks do not insist but that is important to complete formal proofs. Put formally, let us assume a channel `W`, a linear ECC `C`, and an input distribution `P`. A function `f` performs Maximum Likelihood decoding when it satisfies:

```
Definition ML_decoding :=
  ∀ y : P.-receivable W,
  ∃ x, f y = Some x ∧ W ``(y | x) = \rmax_(x' in C) W ``(y | x').
```

The type `P.-receivable W` denotes `receivable` vectors, i.e., it hides the assumption that the message `y` has a non-zero probability, which happens to be the same hypothesis that was needed to define aposteriori probabilities in Sect. 2.2.2.

Maximum Likelihood decoding is desirable because it achieves the smallest error rate among all the possible decoders [Infotheo, 2019, Lemma `ML_smallest_err_rate`]. Still, it is possible to achieve Maximum Likelihood decoding via Minimum Distance decoding. This is for example the case when the channel `W` is a binary symmetric channel with error probability strictly less than $\frac{1}{2}$. In this case, for any linear ECC `C`, repair function `f`, and input distribution `P`, Minimum Distance decoding implies Maximum Likelihood decoding:

```
Lemma MD_implies_ML : p < 1/2 → MD_decoding [set cw in C] f →
  (∀ y, f y ≠ None) → ML_decoding W C f P.
```

Maximum Aposteriori Probability decoding decodes to messages that maximize the aposteriori probability. A decoder `dec` implements Maximum Aposteriori Probability decoding when the following holds:

```
Definition MAP_decoding := ∀ y : P.-receivable W,
  ∃ m, dec y = Some m ∧ P `^^ W (m | y) = \rmax_(m in C) (P `^^ W (m | y)).
```

Maximum Aposteriori Probability decoding is desirable because it achieves Maximum Likelihood decoding:

```
Lemma MAP_implies_ML : MAP_decoding W C dec P → ML_decoding W C dec P.
```

Maximum Posterior Marginal decoding is similar to Maximum Aposteriori Probability decoding: it decodes to messages such that each bit maximizes the marginal aposteriori probability.

In the rest of this paper, we illustrate these definitions in due time by providing concrete examples of decoders (a Minimum Distance decoder in Sect. 4.3, two examples of Bounded Distance decoders in Sections 7.3 and 6.4, and one example of Maximum Posterior Marginal decoder in Sect. 8.3).

4 Use-case: Formalization of Hamming Codes and Their Properties

Hamming codes were the first linear ECCs to detect *and* correct errors. They were discovered by R. W. Hamming [Hamming, 1950] and are still in use today. In this section, we formalize their theory using the libraries presented above. The formalization presented in this section is a revision of previous work [Asai, 2014, Sect. 5], [Affeldt and Garrigue, 2015, Sect. 4].

4.1 Formal Definition of Hamming Codes

A Hamming code of length $n = 2^m - 1$ (with $m \geq 2$) is a linear ECC defined by a parity-check matrix whose columns consist of all non-zero words of length m. The dimension of such a Hamming code is $k = 2^m - m - 1$, i.e., one adds m extra bits for error checking. For example, the Hamming code with $n = 7$ and $k = 4$ is $\begin{bmatrix} 0&0&0&1&1&1&1 \\ 0&1&1&0&0&1&1 \\ 1&0&1&0&1&0&1 \end{bmatrix}$.

Formally, for any `m` (and `n` defined appropriately), we can define the parity-check matrix of an Hamming code using a function `cV_of_nat` that builds column vectors with the binary representation of natural numbers (e.g., for the matrix above, `cV_of_nat 3 1` returns the first column vector, `cV_of_nat 3 2` the second, etc.):

```
(* Module Hamming *)
Definition PCM := \matrix_(i < m, j < n) (cV_of_nat m j.+1 i 0).
Definition code : Lcode0.t _ n := kernel PCM.
```

4.2 Minimum Distance of Hamming Codes

One can establish the minimum distance of Hamming codes by analyzing the parity-check matrix. One first shows that there are no codewords of weights 1 and 2 while there is a codeword of weight 3, namely $7 \times 2^{n-3} = (1110\cdots0)_2$. Hamming codes are therefore not trivial and their minimum distance is 3:

```
Lemma hamming_not_trivial : not_trivial (Hamming.code m).
Lemma hamming_min_dist : min_dist hamming_not_trivial = 3.
```

As a consequence, Hamming codes can correct 1-bit errors by Minimum Distance decoding (by the Lemma `mddP` of Sect. 3.4).

4.3 Minimum Distance Decoding for Hamming Codes

The procedure of decoding for Hamming codes is typically the first example of non-trivial decoding algorithm one finds in a textbook (e.g., [MacWilliams and Sloane, 1977; McEliece, 2002]). It goes as follows. Let us consider some channel output `y` with at most one error. To decode `y`, we compute its syndrome. If the syndrome is non-zero, then it is actually the binary representation of the index of the bit to flip back. The function `hamming_err` computes the index `i` of the bit to correct as well as a vector to repair the error (`nat_of_rV`/`rV_of_nat` perform conversions between natural numbers and their binary representation in terms of row-vectors of bits). The function `hamming_repair` fixes the error by adding this vector:

```
Definition hamming_err y :=
  let i := nat_of_rV (syndrome (Hamming.PCM m) y) in
  if i is 0 then 0 else rV_of_nat n (2 ^ (n - i)).
Definition hamming_repair : repairT _ _ n :=
  [ffun y ⇒ Some (y + hamming_err y)].
```

We can show that the `hamming_repair` function implements Minimum Distance decoding (defined in Sect. 3.4):

```
Lemma hamming_MD_decoding :
  MD_decoding [set cw in Hamming.code m] hamming_repair.
```

Therefore, it is also a Maximum Likelihood decoding (under the condition of application of Lemma `MD_implies_ML` from Sect. 3.4).

4.4 The Encoding and Discard Functions for Hamming Codes

We now complete the formalization of Hamming codes by providing the encoding and discard functions (as in Fig. 1). For that purpose, we define the systematic form of Hamming codes.

Hamming Codes in Systematic Form Modulo permutation of the columns, the parity-check matrix of Hamming codes can be transformed into systematic form $H = [A \,||\, 1]$ (as in the example about repetition codes in Sect. 3.3). This provides us with a generating matrix $G = \left[1 \,||\, -A^T\right]$. For illustration in the case of the $(7,4)$-Hamming code, we have:

$$H_{7,4} = \left[\begin{array}{ccc||ccc} 0&1&1&1&0&0\\ 1&0&1&1&1&0\\ 1&1&0&1&0&0 \end{array}\right], \qquad G_{7,4} = \left[\begin{array}{cccc||ccc} 1&0&0&0&0&1&1\\ 0&1&0&0&1&0&1\\ 0&0&1&0&1&1&0\\ 0&0&0&1&1&1&1 \end{array}\right].$$

Let `systematic` be the column permutation that turns `Hamming.PCM` into H. The parity-check matrix in systematic form is formalized as follows:

```
(* Module SysHamming *)
Definition H := col_perm systematic (Hamming.PCM m).
```

From `H`, we can derive a generating matrix `G` (using an intermediate matrix `A`):

```
Definition A :=
  lsubmx (castmx (erefl, esym (subnK (Hamming.m_len m’))) H).
Definition G :=
  castmx (erefl, subnK (Hamming.m_len m’)) (row_mx 1%:M (- A)^T).
```

In `A`, `lsubmx` extracts the left sub-matrix, more precisely the first m columns, where m is taken from the type of the expression headed by `castmx`. In `A`, `castmx` casts a matrix with n columns to a matrix with $m+(n-m)$ columns (and the opposite direction in `G`), with `subnK (Hamming.m_len m’)` a proof that $n = m+(n-m)$. In `G`, `row_mx` does a juxtaposition.

The discard function in systematic form can be succinctly defined by multiplication by the (transpose of the) matrix $[1\,||\,0]$:

```
Definition mx_discard : ’M[’F_2]_(n - m, n) :=
  castmx (erefl, subnK (Hamming.m_len m’)) (row_mx 1%:M 0).
```

These casts are required to accommodate the dependent types used by MATHCOMP matrices, which themselves avoid having to prove that indices are within range. They can be ignored by the reader because they have no computational contents. However, they require manual input when composing matrices (for example using `row_mx`) or defining sub-matrices (for example when using `lsubmx`). Matrices defined with `castmx` are also the source of extra proof-steps. Typically, when we want to address the (i,j)-th element of a cast matrix, we need to look for the (i',j')-th elements of the uncast matrix, where i and i' (resp. j and j') are indices with the same value but with different types defined using MATHCOMP "ordinals". Fortunately, the MATHCOMP library provides good support to deal with ordinals and moreover we are here mostly dealing with the restricted class of columns' manipulations.

The Complete Encoder and Decoder Functions of Hamming Codes Using the column permutation `systematic` the other way around, we can produce the discard function and the generating matrix corresponding to the original `Hamming.PCM`:

```
Definition mx_discard := col_perm systematic^-1 (SysHamming.mx_discard m’).
Definition discard : discardT _ n _ := fun y ⇒ y *m mx_discard^T.
Definition GEN := col_perm systematic^-1 (SysHamming.G m’).
```

Coupled with the `hamming_repair` function from Sect. 4.3, `GEN` and `discard` provide us with a complete definition of Hamming encoders and decoders:

```
Definition channel_code := mkCode
  [ffun t ⇒ t *m GEN] [ffun x ⇒ omap discard (hamming_repair _ x)].
```

4.5 Error Rate of Hamming Codes over the Binary Symmetric Channel

We can now use our formalization of Hamming codes to recover standard results. For example, we can show, in the case of a binary symmetric channel `W` with probability `p`, that the error rate (see Sect. 2.2.1) of Hamming codes can be expressed as a closed formula:

```
Lemma hamming_error_rate : p < 1/2 →
  echa(W, channel_code) =
    1 - ((1 - p) ^ n) - INR n * p * ((1 - p) ^ (n - 1)).
```

where `channel_code` is the channel code built in the previous sections.

5 Formalization of Cyclic Codes and Euclidean Decoding

In this section, we formalize the basic tools to deal with cyclic codes, whose ease of encoding has made the most studied of all codes [MacWilliams and Sloane, 1977, p. 188].

The goal of Sect. 5.1 is to explain the *key equation*. It is a relation between the syndrome (see Sect. 3) and two polynomials called the locator and the evaluator polynomials that are crafted in such a way that decoding amounts to solving the key equation (note that in the context of cyclic codes, polynomials are used to represent row-vectors). In Sect. 5.2, we explain a standard algorithm to solve the key equation: the Euclidean algorithm for decoding. The Euclidean algorithm for decoding applies in particular to *cyclic codes*. We therefore complete this section with the formalization of cyclic codes as well as polynomial codes, which is a larger class of codes (see Sect. 5.3).

5.1 Locator, Evaluator, Syndrome Polynomials, and the Key Equation

In this section, we first formalize the polynomials that form the key equation, and then formalize the key equation itself. The polynomials in question are the (error-)locator polynomials (Sect. 5.1.1), the (error-)evaluator polynomials (Sect. 5.1.2), and the syndrome polynomials (Sect. 5.1.3). In this section, we provide generic definitions; concrete instantiations can be found in Sections 7 and 6.

5.1.1 Locator Polynomials

The *support set* of a vector `e` is the set of indices `i` such that `i`th component of `e` is not zero:

```
Definition supp : {set 'I_n} := [set i | e_i ≠ 0].
```

In the following, `a` is a vector of n values that defines the code; we are dealing with cyclic codes that can be defined succinctly because they present some regularity.

The *locator polynomial* of a vector `e` is a polynomial whose roots give the index of the errors, i.e., the elements of the output vector that make the checksum fails. It

is denoted by σ(a, e). We formalize it as the polynomial errloc a (supp e) where errloc is defined as follows:

```
Definition errloc (a : 'rV[F]_n) (E : {set 'I_n}) : {poly F} :=
  \prod_(i in E) (1 - a_i *: 'X).
```

The zeros of the error-locator polynomial are the points a_i^{-1} with i belonging to E; since we take E = supp e, this means that the zeros of the error-locator polynomial identifies the elements of the support set of e.

The *ith punctured locator* polynomial σ(a, e, i) [McEliece, 2002, p. 239] is then simply defined by errloc a (supp e :\ i). It is used in the next section to define evaluator polynomials.

5.1.2 Evaluator Polynomials

The *evaluator polynomial* of a vector e is a polynomial from which it is possible to compute the correct value of an erroneous element of e. The evaluator polynomial ω(f, a, e)[4] is defined using the ith punctured locator polynomial:

```
Definition erreval (f a : 'rV[F]_n) e :=
  \sum_(i in supp e) e_i * f_i *: σ(a, e, i).
```

The two vectors of n values a and f define the code. The vector a is the same as for the locator polynomial. The vector f comes from a generalization thanks to which one can accommodate Reed-Solomon and Bose-Chaudhuri-Hocquenghem codes.

The fact that evaluator polynomials make it possible to compute the correct value of an erroneous element is explained by the following lemma:

```
Lemma erreval_vecE i : i ∈ supp e →
  e_i * f_i = - a_i * ω(f, a, e).[a_i^-1] / σ(a, e)^`().[a_i^-1].
```

The MATHCOMP notation p.[x] is for the evaluation of a polynomial p at point x and ^`() is for the formal derivative of a polynomial (see Table 3). The equation looks as follows in mathematical notation:

$$e_i f_i = -a_i \frac{\omega_{f,a,e}(a_i^{-1})}{\sigma'_{a,e}(a_i^{-1})}.$$

This lemma indicates that to discover the vector e (and thus decode), we only need to compute ω(f, a, e) and σ(a, e). Computing these two polynomials is the purpose of the Euclidean algorithm for decoding.

Last, decoding using the Euclidean algorithm will require the locator polynomial σ(a, e) and evaluator polynomial ω(f, a, e) to be relatively prime (see Sect. 5.2.3):

```
Lemma coprime_errloc_erreval : coprimep σ(a, e) ω(f, a, e).
```

For this lemma to hold, we assume that a represents non-zero, pairwise distinct elements, that the support of e is smaller than or equal to t, and that f is non-zero on the support of e.

[4] The evaluator polynomial is also denoted by η in the literature.

5.1.3 Syndrome Polynomials

In Sect. 3.1, we defined the notion of syndrome as a row-vector. However, syndromes are sometimes more conveniently seen as polynomials. We formalize *syndrome polynomials* as follows:

```
Definition syndrome_coord (i : nat) (y : 'rV_n) :=
  \sum_(j < n) y_j * b_j * a_j ^+ i.
Definition syndromep r y := \poly_(i < r) (syndrome_coord i y).
```

More precisely, they are the polynomials corresponding to the (row-vector) syndromes of the so-called *Generalized Reed-Solomon codes*, i.e., linear ECCs with the following parity-check matrix:

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ a_0 & a_1 & \cdots & a_{n-1} \\ a_0^2 & a_1^2 & \cdots & a_{n-1}^2 \\ \vdots & \vdots & \ddots & \vdots \\ a_0^{r-1} & a_1^{r-1} & \cdots & a_{n-1}^{r-1} \end{bmatrix} \begin{bmatrix} b_0 & & & \\ & b_1 & & 0 \\ & & \ddots & \\ & 0 & & b_{n-2} \\ & & & & b_{n-1} \end{bmatrix}.$$

We call the matrix on the left the r-Vandermonde matrices of the vector a of size n, formalized as follows:

```
Definition vander_gen (r : nat) := \matrix_(i < r, j < n) a_j ^+ i.
```

In MATHCOMP, the above matrix product thus becomes:

```
(* Module GRS *)
Definition PCM r : 'M_(r, n) := vander_gen a r *m diag_mx b.
```

In the definition of the evaluator polynomial in Sect. 5.1.2, the vector `f` that was presented as a generalization makes it possible to define evaluator polynomials for any Generalized Reed-Solomon codes.

5.1.4 The Key Equation

The *key equation* is a relation between the locator polynomial (Sect. 5.1.1), the evaluator polynomial (Sect. 5.1.2), and the syndrome polynomial (Sect. 5.1.3). It is intended to be solved for the locator and evaluator polynomials (resp. σ and ω), so as to apply the characterization lemma of Sect. 5.1.2 to perform decoding. The key equation reads as

$$\sigma \mathscr{S} \equiv \omega \pmod{X^r}$$

where r is the size of codewords. Formally:

```
Lemma GRS_key_equation r :
  Sigma * GRS.syndromep a b r y = Omega + GRS_mod r * 'X^r.
```

where `Sigma` and `Omega` are appropriate locator and evaluator polynomials, and `GRS_mod` is some polynomial. We will instantiate this generic key equation for Reed-Solomon in Sect. 6.3 and for Bose-Chaudhuri-Hocquenghem in Sect. 7.2.2.

Solving the key equation is the purpose of the Euclidean algorithm explained in the next section.

5.2 The Euclidean Algorithm for Decoding and the Key Equation

The Euclidean algorithm for decoding solves the key equation (seen in Sect. 5.1.4) for the locator and the evaluator polynomials, given the syndrome polynomial. This approach [Sugiyama et al, 1975] to decoding applies to several codes (Reed-Solomon, Bose-Chaudhuri-Hocquenghem, Goppa).

We explain our formalization of the Euclidean algorithm for decoding in Sections 5.2.1 and 5.2.2. The main result is the lemma of Sect. 5.2.3 that shows that the Euclidean algorithm solves the key equation.

5.2.1 The Polynomials Computed by the Euclidean Algorithm for Decoding

The Euclidean algorithm involves four sequences of polynomials: r_i, q_i, u_i, v_i. The sequence r_i is defined by iterated modulo (recall from Table 3 that `%%` stands for the remainder of the pseudo-division of polynomials):

```
(* Module Euclid *)
Fixpoint r_i :=
  if i is j.+1 then if j is k.+1 then r_k %% r_j
                    else r_1
  else r_0.
```

The initialization polynomials are r_0 and r_1. The polynomial q_i is defined similarly but with iterated division; we therefore have $r_i = q_{i+2}r_{i+1} + r_{i+2}$. The sequence u_i (resp. v_i) is defined such that $u_{i+2} = -q_{i+2}u_{i+1} + u_i$ (resp. $v_{i+2} = -q_{i+2}v_{i+1} + v_i$) with $(u_0, u_1) = (1, 0)$ (resp. $(v_0, v_1) = (0, 1)$).

The sequences above satisfy a number of relations. Let us just introduce the ones that are directly useful to prove the main (forthcoming) lemma that explains decoding using the Euclidean algorithm. First, the sequences u_i, v_i, and r_i satisfy the following relations [McEliece, 2002, Sect. 9.4, Table 9.2 (C, D)]:

```
Lemma vu i : v_i.+1 * u_i - v_i * u_i.+1 = (- 1) ^+ i.
Lemma ruv i : r_i = u_i * r_0 + v_i * r_1.
```

Second, the size of r_i polynomials (i.e., $1 + \deg(r_i)$ if $r_i \neq 0$, see Table 3) is strictly decreasing (as long as it does not reach 0):

```
Lemma ltn_size_r i : 1 ≤ i → r_i ≠ 0 → size r_i.+1 < size r_i.
```

The lemma is here stated for $i \geq 1$; the case $i = 0$ depends on the initialization polynomials r_0 and r_1.

5.2.2 The Stopping Condition of the Euclidean Algorithm

We have seen in the previous section that the size of the polynomials r_i is strictly decreasing. In this section, we are concerned with the index `stop` at which this size becomes smaller than some value. The outputs of the Euclidean algorithm are the polynomials $r_{\texttt{stop}}$ and $v_{\texttt{stop}}$; to anticipate a little, we will see that $r_{\texttt{stop}}$ and $v_{\texttt{stop}}$ will turn out to be the sought evaluator and locator polynomials.

Let us assume some `t` such that `t < size` r_0. The Euclidean algorithm stops when `size` r_i is smaller than `t`. We formalize this condition using the following predicate, which holds for indices `k` such that `t < size` r_i forall `i` $\leq$ `k`:

```
Definition euclid_cont := [pred k | [∀ i : 'I_k.+1, t < size r_i]].
```

We use this predicate to define the largest index such that `euclid_cont` holds:

```
Lemma ex_euclid_cont : ∃ k, euclid_cont k.
Lemma euclid_cont_size_r : ∀ k, euclid_cont k → k ≤ size r_0.
Definition stop' := ex_maxn ex_euclid_cont euclid_cont_size_r.
```

The construct `ex_maxn` comes from MATHCOMP and returns the largest natural number such that some property holds. Thus, the desired index `stop` is `stop'.+1`, the first index at which `euclid_cont` does not hold anymore.

While the size of r_k is strictly decreasing, one can show that the sum of the sizes of $v_{k.+1}$ and r_k is constant [McEliece, 2002, Sect. 9.4, Table 9.2 (F)]:

```
Lemma relationF k : k < stop → (size v_{k.+1}).-1 + (size r_k).-1 = (size r_0).-1.
```

(Under the hypotheses `t < size` r_0 and `size` $r_1 \leq$ `size` r_0).

Suppose that `stop` has been defined w.r.t. to some value `t`. We can then deduce the following lemma [McEliece, 2002, Sect. 9.4, Lemma 2]:

```
Lemma euclid_lemma p t : p + t = size r_0 →
  let stop := stop t r_0 r_1 in size v_stop ≤ p ∧ size r_stop ≤ t.
```

Proof Using the Lemma `relationF`.

5.2.3 How to Solve the Key Equation using the Euclidean Algorithm

Let us assume that we have polynomials V, R, r_0, and r_1 (with $V \neq 0$ and $r_1 \neq 0$) that satisfy a key equation $V r_1 \equiv R \pmod{r_0}$ and such that the sum of the sizes of V and R is smaller than or equal to the size of r_0. Then we can show that there is a non-zero polynomial k such that $v_{stop} = kV$ and $r_{stop} = kR$ [McEliece, 2002, Sect. 9.4, Theorem 9.5]. In the event that V and R are relatively prime, we can even show that k is actually the scalar $\frac{V(0)}{v_{\mathtt{stop}}(0)}$:

```
Lemma solve_key_equation_coprimep p t :
  V * r_1 = R + U * r_0 →
  size V ≤ p → size R ≤ t →
  p + t = size r_0 →
  coprimep V R →
  let stop := stop t r_0 r_1 in
  ∃ k, k ≠ 0 ∧ v_stop = k *: V ∧ r_stop = k *: R.
```

Proof Using the Lemmas `vu`, `ruv`, and `euclid_lemma`.

5.3 Formalization of Polynomial and Cyclic ECCs

We conclude this section with a formalization of cyclic codes. Beforehand, we formalize the more general class of polynomial codes. In this context, a codeword $[c_0; c_1; \cdots; c_{n-1}]$ is better seen as the polynomial $c_0 + c_1 X + \cdots + c_{n-1} X^{n-1}$. The conversion is handled by the MATHCOMP functions `rVpoly` and `poly_rV` (see Table 3).

A *polynomial code* is a linear ECC whose codebook features a *polynomial generator*. Given a codebook, a polynomial generator is a polynomial that divides (without remainder) all the codewords, i.e., a polynomial that satisfies the following predicate:

```
Definition is_pgen := [pred g | [∀ x, (x ∈ C) == (g %| rVpoly x)]].
```

Let `'pgen[C]` be the set of polynomial generators.

A *cyclic code* is a linear ECC whose codebook is stable by right-cyclic shift. Let `rcs` be a function that performs a right-cyclic shift, i.e., that turns a vector $[c_0;\cdots;c_{n-2};c_{n-1}]$ into the vector $[c_{n-1};c_0;\cdots;c_{n-2}]$. Stability by right-cyclic shift can be defined by the following predicate:

```
Definition rcsP (C : {set 'rV[F]_n}) := ∀ x, x ∈ C → rcs x ∈ C.
```

We use the predicate `rcsP` to define cyclic codes as the following dependent record:

```
(* Module Ccode *)
Record t F n := mk {
  lcode0 :> Lcode0.t F n ;
  P : rcsP [set cw in lcode0] }.
```

For cyclic codes, a *cyclic generator* is defined as a non-zero polynomial of lowest degree in the code. Let `'cgen[C]` be the set of cyclic generators. For cyclic codes, the definition of cyclic generators and of polynomial generators coincide:

```
Lemma pgen_cgen (C : Ccode.t F n) (C_not_trivial : not_trivial C) (g : 'rV_n) :
  (rVpoly g ∈ 'pgen[[set cw in C]]) = (g ∈ 'cgen[C]).
```

We provide an example of a polynomial generator in Sect. 6.5.2 (in the case of Reed-Solomon codes) and an example of a cyclic code in Sect. 7.4.2 (in the case of Bose-Chaudhuri-Hocquenghem codes).

6 Use-case: Formalization of Reed-Solomon Codes

Reed-Solomon codes were discovered by I. S. Reed and G. Solomon [Reed and Solomon, 1960]. In this section, we formalize Reed-Solomon codes and apply the library for decoding with the Euclidean algorithm of Sect. 5 to produce a decoder. We define the code in Sect. 6.1 and the syndrome polynomials in Sect. 6.2. In Sect. 6.3, we establish the key equation for Reed-Solomon codes. In Sect. 6.4, we apply the Euclidean algorithm for decoding to Reed-Solomon codes. The other main properties of Reed-Solomon codes are summarized in Sect. 6.5.

6.1 Formal Definition of Reed-Solomon Codes

Let a be an element of a field F. One can define a Reed-Solomon parity-check matrix as the matrix

$$\left[(a^{i+1})^j\right]_{i\in[0,d),j\in[0,n)} = \begin{bmatrix} 1 & a & a^2 & \cdots & a^{n-1} \\ 1 & a^2 & a^4 & \cdots & a^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a^d & a^{2d} & \cdots & a^{d(n-1)} \end{bmatrix}.$$

The length of the code is n, and d is called the *redundancy* [McEliece, 2002, p. 254] because the above matrix corresponds to an encoder adding d symbols to messages. This matrix is a special case of the parity-check matrix for Generalized Reed-Solomon codes: in the definition seen in Sect. 5.1.3, instantiate the two vectors with the same vector $[1;a;a^2;\cdots;a^{n-1}]$ and the parameter r with d. Formally, the matrix above and the corresponding code is formalized as follows:

```
(* Module RS *)
Definition PCM : 'M[F]_(d, n) := \matrix_(i, j) (a ^+ i.+1) ^+ j.
Definition code : {vspace _} := kernel PCM.
```

6.2 Syndrome Polynomials for Reed-Solomon Codes

Let y be s vector of size n. The Reed-Solomon syndrome polynomial of y is $S = \sum_{i=1}^{2t-1} \hat{y}_i X^i$ where $\hat{y}_i = \sum_{j=0}^{n-1} y_j a_i^j$. The quantities $\hat{y}_i$'s are known as the *frequency-domain coordinates* of the *discrete Fourier transform* of y. We formalize this definition in a generic way so as to be able to reuse it for Bose-Chaudhuri-Hocquenghem codes.

If we see the vector y as a polynomial of degree $n-1$ and use a vector u to generalize definitions, we can formalize the frequency-domain coordinates $\hat{y}_i$ as follows:

```
Definition fdcoor u y i := (rVpoly y).[u_i].
```

This means that we evaluate the polynomial `rVpoly y` at point `u_i` (the notations are explained also in Table 3). In our implementation, we use the notation `y ^‘_(u, i)` for $\hat{y}_i$. This leads us to the following formal definition of syndrome polynomials:

```
Definition syndromep u y t := \poly_(k < t) y ^‘_(u, inord k.+1).
```

The function `inord` turns the natural `k.+1` into an “ordinal” (see Tables 2 and 3) that can be used as an index for vectors.

It is important to know the relation between the syndrome polynomial and the syndrome defined as a row-vector using the parity-check matrix (see Sect. 3.1). Let us introduce the following notation for Reed-Solomon syndrome polynomials:

```
Notation "'\RSsynp_(' u , y , d )" := (syndromep u y d).
```

One can check that the syndrome polynomial defined above is indeed equal to the syndrome of the Reed-Solomon parity-check matrix (modulo conversion between polynomials and row-vectors) for a well-chosen vector u:

```
Lemma syndrome_syndromep y :
  syndrome PCM y = poly_rV \RSsynp_(rVexp a n, y, d).
```

The expression `rVexp a n` corresponds to the vector $[1,a,a^2,...,a^{n-1}]$.

6.3 The Reed-Solomon Key Equation

The Reed-Solomon locator polynomials (notation: `σ(rVexp a n, y)`) are defined using the generic definition of Sect. 5.1.1.

The Reed-Solomon evaluator polynomials are obtained as an instantiation of the generic definition of Sect. 5.1.2:

```
Notation "'\RSomega_(' a , e )" := (erreval a a e).
```

Using above polynomials, the Reed-Solomon key equation is:

```
Lemma RS_key_equation y :
  σ(rVexp a n, y) * \RSsynp_(rVexp a n, y, t) =
  \RSomega_(rVexp a n, y) + - RS_mod y t * 'X^t.
```

where `t < n` and `RS_mod y t` is some polynomial. The proof is by instantiating the key equation for Generalized Reed-Solomon codes seen in Sect. 5.1.4.

6.4 Decoding using the Euclidean Algorithm

To perform decoding, we initialize the Euclidean algorithm (of Sect. 5.2) with $r_0 = X^d$ and r_1 being the syndrome polynomial of the received message. Then, the lemma of Sect. 5.2.3 gives us a scalar[5] $k \neq 0$ such that $v_{stop} = k\sigma$ and $r_{stop} = k\omega$. Since $\sigma(0) = 1$, we know moreover that $k = \frac{1}{v_{stop}(0)}$. This gives us the polynomials σ and ω (`s` and `w` below). Now that we have the locator and evaluator polynomials, we can use the Lemma `erreval_vecE` (of Sect. 5.1.2) to compute the error vector:

```
Definition RS_err y : {poly F} :=
  let r0 : {poly F} := 'X^d in
  let r1 := \RSsynp_(rVexp a n, y, d) in
  let vstop := v r0 r1 (stop (odd d + t) r0 r1) in
  let rstop := r r0 r1 (stop (odd d + t) r0 r1) in
  let s := vstop.[0]^-1 *: vstop in
  let w := vstop.[0]^-1 *: rstop in
  \poly_(i < n) (if s.[a^- i] == 0 then - w.[a ^- i] / s^`().[a ^- i] else 0).
```

We formalize the repair function in such a way that it always returns a codeword:

```
Definition RS_repair : repairT F F n := [ffun y ⇒
  if \RSsynp_(rVexp a n, y, d) == 0 then
    Some y
  else
    let ret := y - poly_rV (RS_err y) in
    if \RSsynp_(rVexp a n, ret, d) == 0 then Some ret else None].
```

We then formally prove that the repair function implements Bounded Distance decoding (defined in Sect. 3.4). More precisely, it can correct up to t errors, as long as $t \leq \lfloor \frac{d}{2} \rfloor$ (with $d < n$). For this purpose, we need a number of hypotheses: `F` is expected to be a finite field $\mathbb{F}_{q^m}$ for some prime q that does not divide n, and `a` is expected to be a primitive nth root of unity (i.e., n is the smallest non-zero natural such that $\mathtt{a}^n = 1$):

```
Hypothesis qn : ¬ (q %| n). (* q does not divide n *)
Lemma RS_repair_is_correct : n.-primitive_root a →
  t.-BDD (RS.code a n d, RS_repair a n.-1 d).
```

The proof of the case where the received message $c + e$ is not a codeword relies essentially on the Lemma `solve_key_equation_coprimep` (of Sect. 5.2.3) and the error vector characterization lemma `erreval_vecE` (of Sect. 5.1.2). The facts that `q` and `n` are

[5] Indeed, σ and ω are coprime (see Sect. 5.1.2).

relatively prime and that `a` is a primitive nth root of unity are needed to handle the case where $c+e$ is a code word. In this case, we prove that e is in fact 0 by appealing to the minimum distance of the Reed-Solomon codes (see the forthcoming Sect. 6.5.1).

6.5 Minimum Distance of Reed-Solomon Codes and Other Properties

6.5.1 Minimum Distance of Reed-Solomon Codes

The minimum distance of a Reed-Solomon code is $d+1$ where d is the redundancy of the Reed-Solomon code:

```
Hypothesis dn : d < n.
Hypothesis qn : ¬ (q %| n). (* q does not divide n *)
Lemma RS_min_dist : n.-primitive_root a →
  min_dist (RS_not_trivial a dn) = d.+1.
```

The proof consists in two steps. First, we show that all the non-zero codewords have a weight greater than $d+1$:

```
Lemma RS_min_dist1 c : n.-primitive_root a → c ≠ 0 →
  c ∈ RS.code a n d → d.+1 ≤ wH c.
```

This proof relies on a technical but important result for code theory known as the "BCH argument" (or the "BCH bound"). Second, we exhibit one codeword of weight less than or equal to $d+1$. Concretely, we use the codeword corresponding to a generator (see Sect. 6.5.2 below).

6.5.2 Generator Polynomial for Reed-Solomon Codes

Let us define the polynomial `\gen_(a, d)` as follows:

```
Definition rs_gen := \prod_(1 ≤ i < d.+1) ('X - (a ^+ i)%:P).
```

We can prove that this polynomial is a codeword and that its Hamming weight is less than $d+1$:

```
Lemma mem_rs_gen_RS : poly_rV \gen_(a, d) ∈ RS.code a n d.
Lemma wH_rs_gen : wH (poly_rV \gen_(a, d) : 'rV[F]_n) ≤ d.+1.
```

We can therefore use `\gen_(a, d)` to conclude the proof of the minimum distance of Reed-Solomon codes of the previous section (Sect. 6.5.1).

However, the true purpose of `\gen_(a, d)` is to serve as an encoder for Reed-Solomon codes. Indeed, this is a generator in the sense of Sect. 5.3:

```
Lemma rs_gen_is_pgen :
  \gen_(a, d) ∈ 'pgen[RS.codebook a n' d]. (* n = n'.+1 *)
```

It can be used to formalize an encoder in systematic form (as we did in previous work [Affeldt et al, 2016]).

7 Use-case: Formalization of Bose-Chaudhuri-Hocquenghem Codes

Bose-Chaudhuri-Hocquenghem (hereafter, BCH) codes where discovered by R. C. Bose, D. K. Ray-Chaudhuri, and A. Hocquenghem [Hocquenghem, 1959; Bose and Ray-Chaudhuri, 1960]. In this section, we formalize BCH codes and apply the library for decoding with the Euclidean algorithm of Sect. 5 to produce a decoder. This application shares similarities with the application to Reed-Solomon codes seen in Sect. 6. In Sect. 7.1, we define BCH codes. In Sect. 7.2, we establish the key equation for BCH codes. In Sect. 7.3, we apply the Euclidean algorithm for decoding formalized in Sect. 5.2. In Sect. 7.4, we prove the correctness of the decoder using an additional property about the minimum distance of BCH codes.

7.1 Definition of BCH Codes and BCH Syndrome Polynomials

BCH Parity-check Matrix Although BCH codes are binary, their parity-check matrices are made up of elements from the finite field $\mathbb{F}_{2^m}$. Let a_i be n elements belonging to $\mathbb{F}_{2^m}$. BCH codes can be described by the $t \times n$ parity-check matrix

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \\ a_0^3 & a_1^3 & a_2^3 & \cdots & a_{n-1}^3 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{2t-1} & a_1^{2t-1} & a_2^{2t-1} & \cdots & a_{n-1}^{2t-1} \end{bmatrix}.$$

Using a row-vector `a` for the a_i's, this can be written formally:

```
(* Module BCH *)
Definition PCM : 'M_(t, n) := \matrix_(i < t, j < n) a_j ^+ i.*2.+1.
```

Alternate BCH Parity-check Matrix In fact, the proofs of the properties of BCH codes rely on an alternate parity-check matrix of size $2t \times n$:

$$\begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{n-1} \\ a_0^2 & a_1^2 & a_2^2 & \cdots & a_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{2t} & a_1^{2t} & a_2^{2t} & \cdots & a_{n-1}^{2t} \end{bmatrix}. \tag{1}$$

This matrix is built by adding rows to the parity-check matrix defined just above. Yet, it captures the same set of codewords, essentially because the finite field $\mathbb{F}_{2^m}$ has characteristic 2 and $x^2 = x$ for all $x \in \mathbb{F}_2$. The advantage of this alternate parity-check matrix is that it has a more regular shape that moreover happens to be the parity-check matrix of a Generalized Reed-Solomon code: in the definition seen in Sect. 5.1.3, instantiate the two vectors with the same vector a and the parameter r with $2t$.

BCH code BCH codes are defined as the kernel of the parity-check matrix above *restricted* to (binary) vectors of type `'rV['F_2]_n`. We define *restricted codes* as follows. Let `F0` and `F1` be finite fields with `f` a morphism of type `{rmorphism F0 → F1}`, and let `C` be a linear ECC of length `n` over `F1`. The restricted code `c` of length `n` over `F0` is the linear ECC that contains the vectors that are also in `C`:

```
(* Module Rcode *)
Inductive t (C : Lcode0.t F1 n) := mk {
  c : Lcode0.t F0 n ;
  P : ∀ x, x ∈ c ↔ map_mx f x ∈ C }.
```

Let a_i be n elements belonging to $\mathbb{F}_{2^m}$ and t be a bound. A BCH code of length n is the restriction to $\mathbb{F}_2$ of the linear ECC defined by the parity-check matrix `PCM a t`:

```
(* Module BCH *)
Definition code (a : 'rV_n) t :=
  Rcode.t (@GF2_of_F2_rmorphism m) (kernel (PCM a t)).
```

This definition uses implicitly the function `GF2_of_F2 m` that injects an elements of $\mathbb{F}_2$ into $\mathbb{F}_{2^m}$. It is an implicit parameter of the term `@GF2_of_F2_rmorphism m` that proves that it is indeed a morphism.

7.2 BCH Key Equation

Hereafter, we assume that the n elements a_i of the parity-check matrix are such that $a_i \stackrel{def}{=} a^i$ for some $a \in \mathbb{F}_{2^m}$.

7.2.1 Syndrome, Locator, and Evaluator Polynomials

We use the syndrome polynomials defined for Reed-Solomon codes in Sect. 6.2 to deal with BCH codes. Precisely, the syndrome polynomial of a vector `y` of type `'rV['F_2]_n` is formalized as (notation: `\BCHsynp_(a, y, t)`):

```
(* Module BCH *)
Definition syndromep a y t := syndromep a (F2_to_GF2 m y) t.*2.
```

The function `F2_to_GF2 m` injects a vector with elements in $\mathbb{F}_2$ into a vector with elements in $\mathbb{F}_{2^m}$, i.e., `map_mx (@GF2_of_F2 m)`. We can check that this indeed corresponds to the definition of a syndrome (as defined in Sect. 3.1) since a codeword `y` belongs to a BCH code if and only if its syndrome polynomial is 0:

```
Lemma BCH_syndrome_synp y : t.*2 < n →
  (syndrome (BCH.PCM (rVexp a n) t) (F2_to_GF2 m y) == 0) =
  (\BCHsynp_(rVexp a n, y, t) == 0).
```

BCH locator polynomials are defined directly using the generic definition of Sect. 5.1.1: `σ(rVexp a n, F2_to_GF2 m y)`.

For BCH evaluator polynomials, we instantiate the generic definition of Sect. 5.1.2 as follows (notation: `\BCHomega_(rVexp a n, F2_to_GF2 m y)`):

```
Definition BCH_erreval := erreval (const_mx 1) (rVexp a n).
```

7.2.2 BCH Key Equation

In the case of BCH codes, the key equation takes the form $\sigma\mathscr{S} \equiv \omega \pmod{X^{2t}}$ where $2t$ is the length of codewords when one considers the alternate parity-check matrix (see Sect. 7.1). Recall that σ, $\mathscr{S}$, and ω are intended to be the locator, the syndrome, and the evaluator polynomials of the received message. For BCH codes, the evaluator polynomial of the received message is in fact the evaluator of the "twisted" received message [McEliece, 2002, Sect. 9.3, Equation 9.36]. A twisted vector is defined as follows:

```
Definition twisted (y : 'rV[F]_n) := \row_(i < n) (y_i * a ^+ i).
```

Using above definitions, we can prove the key equation for BCH codes:

```
Lemma BCH_key_equation y :
  σ(rVexp a n, F2_to_GF2 m y) * \BCHsynp_(rVexp a n, y, t) =
    \BCHomega_(rVexp a n, twisted a (F2_to_GF2 m y)) +
    BCH_mod (F2_to_GF2 m y) * 'X^t.*2.
```

In this equation, `BCH_mod` is some polynomial and `t.*2 < n`. The proof is a consequence of the key equation for GRS codes seen in Sect. 5.1.4.

7.3 BCH Decoding using the Euclidean Algorithm

To perform decoding, we initialize the Euclidean algorithm with $r_0 = X^{2t}$ and r_1 being the syndrome polynomial of the received message. Like for Reed-Solomon codes, the lemma of Sect. 5.2.3 gives us a scalar $k \neq 0$ such that $v_{stop} = k\sigma$, and, since $\sigma(0) = 1$, we know moreover that $k = \frac{1}{v_{stop}(0)}$. This gives us the locator polynomial. We can finally use the locator polynomial (`s` below) to compute the error vector. Note that in the case of BCH, we do not need the companion evaluator polynomial because, since the code is binary, error correction is just flipping the erroneous bit:

```
Definition BCH_err y : {poly 'F_2} :=
  let r_0 : {poly F} := 'X^t.*2 in
  let r_1 := \BCHsynp_(rVexp a n, y, t) in
  let v_stop := v r_0 r_1 (stop t r_0 r_1) in
  let s := v_stop.[0]^-1 *: v_stop in
  \poly_(i < n) (if s.[a^- i] == 0 then 1 else 0).
```

And we can use the error vector to repair a (possibly damaged) received message:

```
Definition BCH_repair : repairT [finType of 'F_2] [finType of 'F_2] n :=
  [ffun y ⇒ if \BCHsynp_(rVexp a n, y, t) == 0 then
      Some y
    else
      let ret := y + poly_rV (BCH_err y) in
      if \BCHsynp_(rVexp a n, ret, t) == 0 then Some ret else None].
```

The following lemma shows that `BCH_repair` implements Bounded Distance decoding (defined in Sect. 3.4). It can indeed repair any codeword with less than t errors (provided that $2t < n$ and that `a` is not a kth root on unity for $k < n$):

```
Lemma BCH_repair_is_correct (C : BCH.code (rVexp a n) t) : not_uroot_on a n →
  t.-BDD (C, BCH_repair).
```

The case where the received message $c+e$ is not a codeword is proved using the lemmas from Sect. 5. The case where $c+e$ is a codeword requires an additional result to prove that in this case $e=0$: this is a lower bound of the minimum distance of BCH codes whose proof is the matter of the forthcoming Sect. 7.4.1.

7.4 About the Minimum Distance and Cyclicity of BCH Codes

7.4.1 Lower Bound of the Minimum Distance of BCH Codes

In this section, we show that the parameter t of BCH codes is a lower bound of the minimum distance.

Let B be a matrix formed by the first r elements of a subset of r columns from the parity-check matrix of Equation (1) (see Sect. 7.1). The matrix B can be written as follows:

$$B = \begin{bmatrix} b_1 & b_2 & \cdots & b_r \\ b_1^2 & b_2^2 & \cdots & b_r^2 \\ \vdots & \vdots & \ddots & \vdots \\ b_1^r & b_2^r & \cdots & b_r^r \end{bmatrix}.$$

Let f be the function that returns for each column of B its corresponding column in the parity-check matrix of Equation (1). We can show that the determinant of B is:

$$\det B = \prod_{i \le r} b_{f(i)} \det(\mathtt{vander}\,[b_1, \ldots, b_r]). \tag{2}$$

Put formally:

```
Lemma BCH_det_mlinear t r' (f : 'I_r'.+1 → 'I_n) (rt : r' < t.*2) :
  let B := \matrix_(i, j) BCH.PCM_alt a t (widen_ord rt i) (f j) in
  let V := vander (row 0 B) in
  \det B = \prod_(i < r'.+1) BCH.PCM_alt a t (widen_ord rt 0) (f i) * \det V.
```

(`r'` is $r-1$, `widen_ord` is a type cast to inject indices strictly smaller than r into indices strictly smaller than $2t$.)

On the other hand, the determinant of (square) Vandermonde matrices (as we defined in Sect. 5.1.3) has the following property:

$$\det(\mathtt{vander}\,a) = \prod_{i<n} \prod_{i<j<n} (a_j - a_i).$$

Put formally (and assuming $n > 0$):

```
Lemma det_vander n (a : 'rV[R]_n.+1) :
  \det (vander a) = \prod_(i < n.+1) (\prod_(j < n.+1 | i < j) (a_j - a_i)).
```

We now see that Equation (2) indicates that when b_i's are pairwise distinct and f is injective, $\det B$ is not zero. This leads us to the conclusion that all (non-zero) codewords have a Hamming weight strictly greater than t:

```
Lemma BCH_min_dist1 t x (x0 : x ≠ 0) (t0 : 0 < t) (C : BCH.code a t) :
  x ∈ C → t < wH x.
```

Proof Suppose ab absurdo that `x` has a weight strictly smaller than $t+1$. We build an injective function f that associates to the ith non-zero bit of `x` its index strictly smaller than n. Using this function, we exhibit a linearly-dependent combination of `wH x` columns of the BCH parity-check matrix. We can then build a square matrix B from the columns of the BCH parity-check matrix such that $\det B = 0$, contradicting the property above.

7.4.2 Cyclic BCH Codes

For the sake of completeness, we provide a last result about BCH codes. In textbooks, they are often introduced as an example of cyclic code. Yet, we have not been relying on this property so far. Let us make it precise that BCH are cyclic under the additional hypothesis that a has order n:

```
Lemma rcsP_BCH_cyclic (C : BCH.code (rVexp a n) t) :
  a ^+ n = 1 (* a has order n *) → rcsP [set cw in C].
```

8 The Basics of Modern Coding Theory

In the previous sections, we have been dealing with codes whose properties could be formalized essentially using algebra and probabilities. The formalization of modern coding theory [Richardson and Urbanke, 2008] requires more, in particular graphs. In the following, we deal with low-density parity-check (hereafter, LDPC) codes, which are the representative instance of modern coding theory (as explained in Sect. 1).

The sum-product algorithm is the standard example of efficient decoder for LDPC codes. It computes for each bit its marginal aposteriori probability by propagating probabilities in a graph corresponding to the parity-check matrix of the LDPC code. Note however that, while practical LDPC codes use a large graph containing cycles, all proofs of the accuracy of sum-product decoding assume the graph to be acyclic [Kschischang et al, 2001]. While experimental results do show that sum-product decoding works still very well in presence of cycles, it is important to keep this in mind when assessing the applicability of proofs.

Below, we provide tools to formalize the sum-product algorithm. In Sect. 8.1, we explain how to formalize parity-check matrices as graphs. In Sect. 8.2, we explain a formalization of the *summary operator* used in modern coding theory. In Sect. 8.3, we evaluate the marginal aposteriori probability using the summary operator, so as to be able to verify concrete implementations of the sum-product algorithm.

8.1 Parity Check Matrices as Tanner Graphs

The graph-equivalent of a parity-check matrix with elements from the binary field $\mathbb{F}_2$ is called a *Tanner graph*. Let H be a parity-check matrix. Its Tanner graph is a (bipartite) graph with two types of nodes: one *function node* per row and one *variable node* per column. There is an edge between the variable node n_0 and the function node m_0

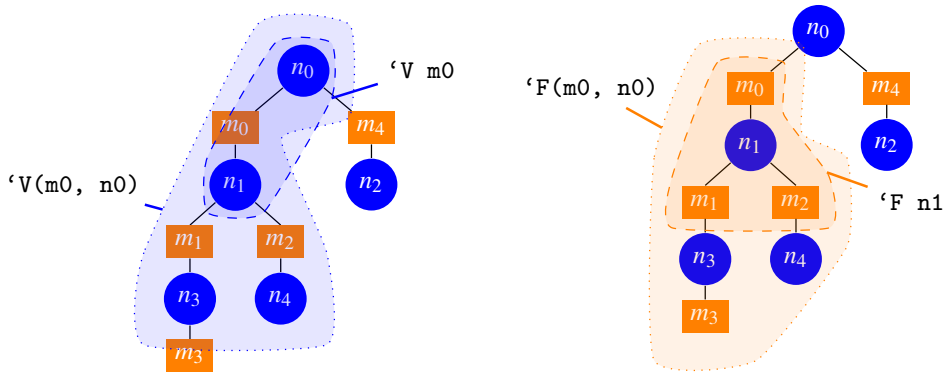

Fig. 2: Successors and subtrees in an acyclic Tanner graph

when $H_{m_0,n_0} = 1$ (Tanner graphs are undirected). See Fig. 2 for examples of Tanner graphs.

When dealing with Tanner graphs, one needs to be able to talk about sets of successors and about some specific subgraphs.

First, we distinguish between sets of successors of variable nodes and of function nodes. We denote the successors of the function (resp. variable) node `m0` (resp. `n0`) by `'V m0` (resp. `'F n0`). See Fig. 2 for examples.

Second, we define "subgraphs rooted at an edge". Let `g` be a graph (formalized by a binary relation) and `m` and `n` be two connected vertices. The subgraph rooted at the edge `m–n` is the set of vertices reachable from `m` without passing through `n`:

```
Variables (V : finType) (g : rel V).
Definition except n : rel V := [rel x y | g x y && (x ≠ n) && (y ≠ n)].
Definition subgraph m n := [set v | g n m && connect (except n) m v].
```

In MATHCOMP, `rel` is the type of binary relations and `connect` is the transitive closure of a relation. We distinguish subgraphs of variable nodes and of function nodes by denoting the function nodes of the subgraph rooted at edge `m0–n0` by `'F(m0, n0)`. Similarly, we denote the variable nodes of the subgraph rooted at edge `m0–n0` *(to which we add `n0`)* by `'V(m0, n0)`. See Fig. 2 for examples and our implementation for complete definitions [Infotheo, 2019].

Last, it is important to distinguish *acyclic Tanner graphs*. We formalize acyclic Tanner graphs by requiring that any path with 3 or more nodes is not a cycle:

```
Definition acyclic g := ∀ p, 2 < size p → ~ path.ucycle g p.
```

A path is essentially a list of connected vertices and the predicate `path.ucycle` comes from the MATHCOMP library.

8.2 The Summary Operator

Pencil-and-paper proofs in modern coding theory make use of a special summation called the *summary operator* [Kschischang et al, 2001]. It is denoted by $\sum_{\sim s} e$ and indicates a summation over all variables contained in the expression e *except* the

variables in the set s. This operator saves the practitioner "from a flood of notation" [Richardson and Urbanke, 2008, p. 49], for example by writing steps such as

$$\prod_{m_0 \in F(n_0)} \sum_{\sim\{n_0\}} \cdots = \sum_{\sim\{n_0\}} \prod_{m_0 \in F(n_0)} \cdots, \tag{3}$$

the reader being trusted to understand that the two occurrences of the summary operator may sum over two different sets (see Sections 8.3.1 and 8.3.2 for concrete examples).

For the sake of formalization, we regard $\sum_{\sim s} e$ as a sum over a vector of variables $[x_0; x_1; \ldots; x_{n-1}]$ such that x_i is fixed using a default vector d when $i \notin r$, where r is the set of freely enumerated indices, i.e., we expect $\{x_i \mid i \in r\} = fv(e) \setminus s$. Instead of $\sum_{\sim s} e$, we write $\sum_{x=d\,[\sim r]} e$ because it is a summation over the vectors x equal to d on all components except the freely enumerated indices in r. The formal Coq notation for $\sum_{x=d\,[\sim r]} e$ is `\sum_(x = d [~ r]) e`, defined as follows:

```
Definition freeon (r : {set 'I_n}) (x d : 'rV[A]_n) : bool :=
  [∀ j, (j \notin r) ⟹ (x_j == d_j)].
Notation "\sum_ ( x '=' d [~ r ] ) F" := (\sum_( x | freeon r d x ) F)
```

We found it difficult to recover formally the terseness of the pencil-and-paper summary operator. First, the precise set of varying x_j's is implicit; it can be inferred by looking at the x_j's appearing below the summation sign but this is difficult to achieve in Coq unless one reflects the syntax. Second, the pencil-and-paper summary operator alternatively suggests to work with vectors x of varying sizes, but this would be technically involved because the size of vectors appears in dependent types (tuples or row-vectors in MathComp). In contrast, our formalization makes clear, for example, that in Equation (3) the first summary operator sums over x_i's with $i \in V(m_0, n_0) \setminus \{n_0\}$ while the second one sums over x_i's with $i \in [1, \ldots, n] \setminus \{n_0\}$. More importantly, thanks to our encoding, we can benefit from the MathComp lemmas about big operators to prove various properties about the summary operator (see Sect. 8.3) or equivalent definitions of the summary operator. For example, our summary operator $\sum_{x=d\,[\sim r]} e(x)$ can be alternatively thought as

$$\sum_{x_1 \in \mathbb{F}_2} \cdots \sum_{x_{|r|} \in \mathbb{F}_2} e(d[r_1 := x_1] \cdots [r_{|r|} := x_{|r|}])$$

where $d[i := b]$ represents the vector d where index i is updated with b. Put formally:

```
Definition summary_fold (r : {set 'I_n}) d e :=
  foldr (fun n0 F t ⇒ \sum_(b in 'F_2) F (t `[n0 := b])) e (enum r) d.
```

(`enum r` is the list $[r_1; r_2; \cdots; r_{|r|}]$.) This definition is equivalent to our summary operator in the sense that `\sum_(x = d [~ r]) e x` *is* equal to `summary_fold r d e`, but `summary_fold` was easier to use to verify our implementation of the sum-product algorithm in Sect. 9.2.

8.3 Evaluation of the Marginal Aposteriori Probability

8.3.1 Correctness of the Estimation

Let us consider a channel W and a channel output y. With sum-product decoding, we are concerned with evaluating $P_{n_0}^W(b|y)$ where P is a uniform distribution and b is the value of the n_0th bit of the input codeword (see Sect. 2.2.2). In the following, we show that $P_{n_0}^W(b|y)$ satisfies the proportionality relation

$$P_{n_0}^W(b|y) \propto W\left(y_{n_0}\middle|b\right) \prod_{m_0 \in F(n_0)} \alpha_{m_0,n_0}(d) \tag{4}$$

where $\alpha_{m_0,n_0}(d)$ is defined below (using the summary operator and Tanner graphs) and d is a vector such that $d_{n_0} = b$.

The expression $\alpha_{m_0,n_0}(d)$ represents the contribution of the n_0th bit to the marginal aposteriori probability coming from a subtree of the Tanner graph (we assume that the Tanner graph is acyclic). More precisely, $\alpha_{m_0,n_0}(d)$ is the marginal aposteriori probability of the n_0th bit of the input codeword in the modified Tanner graph that includes only function nodes from the subgraph rooted at edge m_0–n_0 and in which the received bit y_{n_0} has been erased:

$$\alpha_{m_0,n_0}(d) \stackrel{def}{=} \sum_{x = d\,[\sim V(m_0,n_0)\setminus n_0]} W\left(y_{|V(m_0,n_0)\setminus n_0}\middle|x_{|V(m_0,n_0)\setminus n_0}\right) \prod_{m_1 \in F(m_0,n_0)} \delta^x_{V(m_1)}.$$

In the definition above, δ_V^x is an indicator function that performs checksum checks: $\delta_V^x \stackrel{def}{=} \left(\sum_{i\in V} x_i\right) = 0$. Observe that $\alpha_{m_0,n_0}(d)$ only depends on $d_{n_0} = b$.

Here follows a detailed account of the pencil-and-paper proof of Equation (4). Let b be a bit and d be a vector such that $d_{n_0} = b$. The proof goes as follows. The first step (5a)–(5b) uses the fact that the input distribution is uniform (see Sect. 2.2.2). We will comment about the steps (5c)–(5d) and (5e)–(5f) below in the light of the formal

statement.

$$P_{n_0}^W(b|y) \stackrel{def}{=} K' \sum_{x \in \mathbb{F}_2^n \, x_{n_0}=b} P^W(x|y) \tag{5a}$$

$$= K' \sum_{x \in \mathbb{F}_2^n \, x_{n_0}=b} K W(y|x) \, [x \in C] \tag{5b}$$

$$\propto \sum_{x=d\left[\sim\{n_0\}^{\complement}\right]} W(y|x) \prod_{m_0<m} \delta^x_{V(m_0)}$$

$$= W\left(y_{n_0}\middle|b\right) \sum_{x=d\left[\sim\{n_0\}^{\complement}\right]} W\left(y_{|\{n_0\}^{\complement}}\middle|x_{|\{n_0\}^{\complement}}\right) \prod_{m_0<m} \delta^x_{V(m_0)} \tag{5c}$$

$$= W\left(y_{n_0}\middle|b\right) \sum_{x=d\left[\sim\{n_0\}^{\complement}\right]} W\left(y_{|\{n_0\}^{\complement}}\middle|x_{|\{n_0\}^{\complement}}\right) \prod_{m_0 \in F(n_0)} \prod_{\substack{m_1 \in \\ F(m_0,n_0)}} \delta^x_{V(m_1)} \tag{5d}$$

$$= W\left(y_{n_0}\middle|b\right) \sum_{x=d\left[\sim\{n_0\}^{\complement}\right]} \prod_{m_0 \in F(n_0)} W\left(y_{|V(m_0,n_0)\setminus n_0}\middle|x_{|V(m_0,n_0)\setminus n_0}\right) \prod_{\substack{m_1 \in \\ F(m_0,n_0)}} \delta^x_{V(m_1)} \tag{5e}$$

$$= W\left(y_{n_0}\middle|b\right) \prod_{m_0 \in F(n_0)} \sum_{x=d\,[\sim V(m_0,n_0)\setminus n_0]} W\left(y_{|V(m_0,n_0)\setminus n_0}\middle|x_{|V(m_0,n_0)\setminus n_0}\right) \prod_{\substack{m_1 \in \\ F(m_0,n_0)}} \delta^x_{V(m_1)} \tag{5f}$$

$$\stackrel{def}{=} W\left(y_{n_0}\middle|b\right) \prod_{m_0 \in F(n_0)} \alpha_{m_0,n_0}(d)$$

We now provide the formal statement. Let `W` be a channel. Let `H` be a $m \times n$ parity-check matrix such that the corresponding Tanner graph is acyclic (the formal hypothesis is `Tanner.acyclic_graph (tanner_rel H)`, where `tanner_rel` turns a parity-check matrix into the corresponding Tanner graph). Let `y` be the channel output to decode; we assume that it is receivable (see Sect. 2.2.2). Finally, let `d` be the vector used in the summary operator. Then the aposteriori probability $P_{n_0}^W(b|y)$ can be evaluated by a closed formula:

```
Lemma estimation_correctness (d : 'rV_n) n0 :
  let b := d_n0 in let P := `U C_not_empty in
  P '_ n0 `^^ W (b | y) = Kmpp y * Kppu [set cw in C] y *
    W ``(y_n0 | b) * \prod_(m0 in `F n0) alpha m0 n0 d.
```

The distribution `` `U C_not_empty `` of codewords has the following probability mass function: $c \mapsto 1/|C|$ if $c \in C$ and 0 otherwise. The terms headed by `Kmpp` and `Kppu` correspond to K' and K defined in Sect. 2.2.2. The formal definition of `alpha` relies on the summary operator and follows the pencil-and-paper definition given at the beginning of this section:

```
Definition alpha m0 n0 d := \sum_(x = d [~`V(m0, n0) :\ n0])
  W ``(y # `V(m0, n0) :\ n0 | x # `V(m0, n0) :\ n0) *
    \prod_(m1 in `F(m0, n0)) INR (δ (`V m1) x).
```

In this formal definition, `δ (`V m1) x` corresponds to the indicator function $\delta^x_{V(m_1)}$ explained above. It is implemented using the Boolean equality (symbol `==`, see Table 3) by the following function:

```
Definition checksubsum n (V : {set 'I_n}) (x : 'rV['F_2]_n) :=
  (\sum_(n0 in V) x_n0) == 0.
```

Technical Aspects of the Proof Let us comment about two technical aspects of the proof of `estimation_correctness`.

The first technical aspect is the need to instrument Tanner graphs with partition lemmas to be able to decompose big prods. This is what happens in the step (5c)–(5d). We will come back to this aspect when detailing another example in Sect. 8.3.2 (about the Lemma `recursive_computation`).

The second technical aspect is the main motivation for using the summary operator. We need to make big sums commute with big prods like in the step (5e)–(5f):

$$\sum_{x=d\left[\sim\{n_0\}^{\complement}\right]} \prod_{m_0\in F(n_0)} F_{m_0,x} = \prod_{m_0\in F(n_0)} \sum_{x=d\,[\sim V(m_0,n_0)\setminus n_0]} F_{m_0,x}.$$

By the way, the Coq formalization of this step looks like this:

```
\sum_(x = d [~ setT :\ n0]) \prod_(m0 in 'F n0) ... =
  \prod_(m0 in 'F n0) \sum_(x = d [~'V(m0, n0) :\ n0]) ...
```

To perform such commutations of big operators, we start by applying the MATHCOMP Lemma `big_distr_big_dep` (from the `bigop` theory). Using pencil-and-paper notations, it can be written as

$$\prod_{i\in P}\sum_{j\in Q(i)} F_{i,j} = \sum_{f\in \underset{\lambda x.Q(x)}{\overset{P}{\Downarrow d}}} \prod_{i\in P} F_{i,f(i)}$$

where $\underset{\lambda x.Q(x)}{\overset{P}{\Downarrow d}}$ is a finite set of functions f that map each element $x \in P$ to an element $f(x) \in Q(x)$, d being the default value returned when $x \notin P$. The following example should convey the intuition for this lemma:

$$\begin{aligned}
\prod_{i\in\{1,2\}}\sum_{j\in\{1,2,3\}} F_{i,j} &= (F_{1,1}+F_{1,2}+F_{1,3})(F_{2,1}+F_{2,2}+F_{2,3}) \\
&= F_{1,1}F_{2,1}+F_{1,1}F_{2,2}+F_{1,1}F_{2,3}+\cdots \\
&= \prod_{i\in\{1,2\}} F_{i,\substack{1\mapsto 1\\2\mapsto 1}(j)} + \prod_{i\in\{1,2\}} F_{i,\substack{1\mapsto 1\\2\mapsto 2}(j)} + \prod_{i\in\{1,2\}} F_{i,\substack{1\mapsto 1\\2\mapsto 3}(j)} + \cdots \\
&= \sum_{f\in \underset{\lambda x.\{1,2,3\}}{\overset{\{1,2\}}{\Downarrow}}} \prod_{i\in\{1,2\}} F_{i,f(j)}.
\end{aligned}$$

This is a simple example where there is no dependency between the predicates that characterize the big prod and the big sum on the left-hand side; see Equation (9a) for an example with a dependency.

Second, to make big sums commute with big prods, we perform reindexing. This is where the technical difficult lies. For example, in the case of step (5e)–(5f), it amounts to show that

$$\sum_{\substack{f\in \overset{F(n_0)}{\underset{\lambda xy.y=d\,[\sim V(x,n_0)\setminus\{n_0\}]}{\Downarrow d}}}} \prod_{m_0\in F(n_0)} F_{m_0,f(m_0)} = \sum_{x=d\left[\sim\{n_0\}^{\complement}\right]} \prod_{m_0\in F(n_0)} F_{m_0,x}.$$

The notation $\overset{F(n_0)}{\underset{\lambda xy.y=d\,[\sim V(x,n_0)\setminus\{n_0\}]}{\Downarrow d}}$ means a set of functions that return for $x\in F(n_0)$ all the vectors with index-range $V(x,n_0)\setminus\{n_0\}$ (the contents at other indices being fixed accordingly to the vector d). To perform such a reindexing, we need to provide functions that transform a vector of bits into a family of functions such that these functions can be recombined to recover the vector of bits. See our implementation for the concrete definitions of such functions [Infotheo, 2019, functions `dprojs` and `comb`, file `summary_tanner.v`]. Equipped with these functions, we can use MATHCOMP lemmas such as `reindex_onto` (from the `bigop` theory) to actually perform the reindexing. Informally, applying `reindex_onto` corresponds to the rewriting rule $\sum_{i\in P} F_i = \sum_{h(j)\in P, h'(h(j))=j} F_{h(j)}$ under the hypothesis that the function h cancels the function h' on P.

8.3.2 Recursive Computation of α's

The property proved in the previous section provides a way to evaluate $P_{n_0}^W(b|y)$ but not an efficient algorithm because the computation of $\alpha_{m_0,n_0}(d)$ is *global*: it is about the whole subgraph rooted at the edge m_0–n_0. The second property that we formalize introduces β quantities such that α's (resp. β's) can be computed *locally* from neighboring β's (resp. α's). More precisely, we prove that α's can be computed using β's by the formula

$$\alpha_{m_0,n_0}(d) = \sum_{x=d\,[\sim V(m_0)\setminus n_0]} \delta^x_{V(m_0)} \prod_{n_1\in V(m_0)\setminus n_0} \beta_{n_1,m_0}(x) \tag{6}$$

where β is defined using α:

$$\beta_{n_0,m_0}(d) \overset{def}{=} W\left(y_{n_0}\middle|d_{n_0}\right) \prod_{m_1\in F(n_0)\setminus m_0} \alpha_{m_1,n_0}(d).$$

(We assume the same setting as for the Lemma `estimation_correctness`.)

We now provide a detailed account of the pencil-and-paper proof of Equation (6). This proof relies on ideas similar to the ones already explained in Sect. 8.3.1 but is a little bit more involved.

Proof of Equation (6) (part 1)

$$\alpha_{m_0,n_0}(d) \stackrel{def}{=} \sum_{x=d\,[\sim V(m_0,n_0)\setminus n_0]} W\left(y_{|V(m_0,n_0)\setminus n_0} \middle| x_{|V(m_0,n_0)\setminus n_0}\right) \prod_{m_1 \in F(m_0,n_0)} \delta^x_{V(m_1)} \tag{7a}$$

$$= \sum_{x=d\,[\sim V(m_0,n_0)\setminus n_0]} W\left(y_{|V(m_0,n_0)\setminus n_0} \middle| x_{|V(m_0,n_0)\setminus n_0}\right) \delta^x_{V(m_0)} \prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} \prod_{\substack{m_1 \in \\ F(n_1)\setminus m_0}} \prod_{\substack{m_2 \in \\ F(m_1,n_1)}} \delta^x_{V(m_2)} \tag{7b}$$

$$= \sum_{x=d\,[\sim V(m_0,n_0)\setminus n_0]} \delta^x_{V(m_0)} \underbrace{\prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} W(y_{n_1}|x_{n_1}) \prod_{\substack{m_1 \in \\ F(n_1)\setminus m_0}} W\left(y_{|V(m_1,n_1)\setminus n_1} \middle| x_{|V(m_1,n_1)\setminus n_1}\right) \prod_{\substack{m_2 \in \\ F(m_1,n_1)}} \delta^x_{V(m_2)}}_{A_0(x)}$$

$$= \sum_{x=d\,[\sim V(m_0)\setminus n_0]} \sum_{x'=x\,[\sim V(m_0,n_0)\setminus V(m_0)]} \delta^{x'}_{V(m_0)} A_0(x')$$

$$= \sum_{x=d\,[\sim V(m_0)\setminus n_0]} \delta^x_{V(m_0)} \underbrace{\sum_{x'=x\,[\sim V(m_0,n_0)\setminus V(m_0)]} A_0(x')}_{A(x)}$$

Let us comment on one technical aspect of this proof. The step (7a)–(7b) performs a splitting of the inner product in α messages. Formally, it turns

```
\prod_(m1 in ‘F(m0, n0) :\ m0) INR (δ (‘V m1) x)
```

into

```
\prod_(n1 in ‘V m0 :\ n0) \prod_(m1 in ‘F n1 :\ m0) \prod_(m2 in ‘F(m1, n1))
  INR (δ (‘V m2) x)
```

This requires to show that `‘F(m0, n0) :\ m0` can be partitioned (when `H` is acyclic) into smaller `‘F(m1, n1)` where `n1` is a successor of `m0` and `m1` is a successor of `n1`, i.e., according to the following partition:

```
Definition Fgraph_part_Fgraph m0 n0 : {set {set 'I_m}} :=
  (fun n1 ⇒ \bigcup_(m1 in ‘F n1 :\ m0) ‘F(m1, n1)) @: ((‘V m0) :\ n0).
```

Once `Fgraph_part_Fgraph m0 n0` has been shown to cover `‘F(m0, n0) :\ n0` with pairwise disjoint sets, this step essentially amounts to use the lemmas `big_trivIset` and `big_imset` of the MATHCOMP library (from the `finset` theory).

Proof of Equation (6) (part 2) To conclude the proof, one needs to show that $A(x) = \prod_{n_1 \in V(m_0)\setminus n_0} \beta_{n_1,m_0}(x)$. Here follows a detailed account of this part of the proof. Similarly to the proof of the Lemma `estimation_correctness` (Sect. 8.3.1), it involves commutations of big sums and big prods using the summary operator. See the two steps (8a)–(9a) and (10a)–(10b) below.

$$\prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} \beta_{n_1,m_0}(x) \overset{def}{=} \prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} W(y_{n_1}|x_{n_1}) \underbrace{\prod_{\substack{m_1 \in \\ F(n_1)\setminus m_0}} \sum_{z=x[\sim V(m_1,n_1)\setminus n_1]} W\left(y_{|V(m_1,n_1)\setminus n_1} \middle| z_{|V(m_1,n_1)\setminus n_1}\right) \prod_{\substack{m_2 \in \\ F(m_1,n_1)}} \delta^z_{V(m_2)}}_{B} \tag{8a}$$

The first step is a commutation between the leading big prod and big sum of B:

$$B = \sum_{\substack{z\in\mathbb{F}_2^n \\ F(n_1)\setminus m_0 \\ \Downarrow x \\ f_z \in \lambda m_1 y.\, y=x[\sim V(m_1,n_1)\setminus n_1]}} \underbrace{\prod_{\substack{m_1 \in \\ F(n_1)\setminus m_0}} W\left(y_{|V(m_1,n_1)\setminus n_1} \middle| f_z(m_1)_{|V(m_1,n_1)\setminus n_1}\right) \prod_{\substack{m_2 \in \\ F(m_1,n_1)}} \delta^{f_z(m_1)}_{V(m_2)}}_{C(z)} \tag{9a}$$

Observe that we do not sum over functions directly but over vectors z on which the functions f_z depend because we use the vectors z explicitly in the next steps:

$$\begin{aligned}
&\prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} \beta_{n_1,m_0}(x) \\
&= \prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} \sum_{\substack{z\in\mathbb{F}_2^n \\ F(n_1)\setminus m_0 \\ \Downarrow x \\ f_z \in \lambda m_1 y.\, y=x[\sim V(m_1,n_1)\setminus n_1]}} W(y_{n_1}|z_{n_1})\, C(z) && \text{(10a)} \\
&= \sum_{\substack{V(m_0)\setminus n_0 \\ \Downarrow x \\ g \in \lambda n_1 z.\, f_z \in \begin{pmatrix} F(n_1)\setminus m_0 \\ \Downarrow x \\ \lambda m_1 y.\, y=x[\sim V(m_1,n_1)\setminus n_1] \end{pmatrix}}} \prod_{\substack{n_1 \in \\ V(m_0)\setminus n_0}} W\left(y_{n_1} \middle| g(n_1)_{n_1}\right) C(g(n_1)) && \text{(10b)} \\
&= A(x).
\end{aligned}$$

The last step (10a)–(10b) is again a commutation between big operators. □

9 Use-case: Sum-Product Decoding

9.1 Implementation of Sum-Product Decoding over the Binary Erasure Channel

We formalize a concrete implementation [Hagiwara, 2012, Chapter 9] of sum-product decoding in the case of communication over the *binary erasure channel*, i.e., a channel such that the output message is a codeword in which some bits have been erased (but none has been flipped). This is a simple setting that demonstrates how Tanner graphs are used, but it is also an important setting to study the theoretical properties of LDPC codes.

To model communication over the binary erasure channel, we introduce an alphabet with erasures `Star` (`Blank` is for unknown letters):

```
Inductive letter := Bit of 'F_2 | Star | Blank.
```

We also introduce a comparison operator for letters (notation: $\leq_l$) that we lift to a comparison operator for matrices of letters (notation $\leq_m$):

```
Definition lel (a b : letter) :=
  (a == b) || if a is Bit _ then b == Star else false.
Definition mxlel m n (A B : 'M[letter]_(m, n)) := [∀ m, [∀ n, A m n ≤_l B m n]].
```

Let us assume a binary code with parity-check matrix `H`. The codeword `c` has been received as the message `y` by communication over the binary erasure channel when the following relation holds:

```
Definition BEC_IO m n (H : 'M['F_2]_(m, n)) c y :=
  syndrome H c = 0 ∧ map_mx Bit c ≤_m y.
```

The Sum-Product algorithm performs iteratively a Sum and a Product operation until a fixed point is reached. The Sum operation consists in inferring the bit that hides behind an erasure by using the parity-check equations of the parity-check matrix (or returning an erasure when this is not possible):

```
Definition Sum (s : seq letter) :=
  if has starblank s then Star
  else Bit (sum_letter (filter is_Bit s)).
```

The Prod operation consists in performing a majority vote (`N(b | s)` counts the number of `b`s in the sequence `s`):

```
Definition Prod (s : seq letter) :=
  if N(Bit 0 | s) > N(Bit 1 | s) then Bit 0
  else if N(Bit 0 | s) < N(Bit 1 | s) then Bit 1
  else Star.
```

The input of the Sum-Product algorithm is the set of the parity-check equations instantiated using the received message; the erasures are seen as the unknowns. These equations are conveniently represented using an $m \times n$-matrix. The Sum-Product algorithm consists in transforming the input matrix by applying the `Sum` operations to each matrix entry, and then the `Prod` operations likewise:

```
Definition mxSum M := \matrix_(i < m, j < n)
  if i ∈ 'F j then Sum (rowVnextD1 (row i M) i j) else M i j.
Definition mxProd (y : 'rV[letter]_n) M := \matrix_(i < m, j < n)
  if i ∈ 'F j then Prod (colFnextD1 y_j (col j M) j i) else M i j.
Definition mxSumProd (y : 'rV[letter]_n) := mxSum \o mxProd y.
```

The sequence `rowVnextD1 r i j` contains the r_k's such that `k ∈ 'V i :\ j`. The sequence `colFnextD1 l c j i` contains the c_k's such that `k = l` or `k ∈ 'F j :\ i`. The MATHCOMP notation `\o` is for function composition.

In COQ, we formalize the Sum-Product algorithm as a recursive function that returns a fixpoint of `mxSumProd`. We use the number of stars in the matrix to show that the function terminates (measure `num_stars` below):

```
Program Fixpoint SP_BEC0_rec y M
  (_ : approx_BEC_input H y M) (_ : mxSumProd H y M ≤_m M)
  { measure (num_stars M) } :
  {M' | mxSumProd H y M' = M' ∧ M' ≤_m M ∧
        ∃ k, M' = iter k (mxSumProd H y) M} :=
```

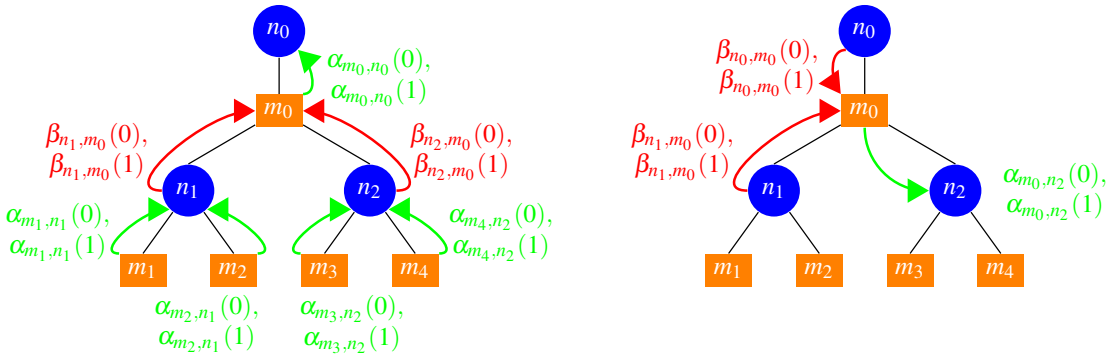

Fig. 3: Illustrations for `sumprod_up` and `sumprod_down`. Left: `sumprod_up` computes the `up` links from the leaves to the root. Right: `sumprod_down` computes the `down` link of edge m_0–n_2 using the β's of edges m_0–n_i ($i \neq 2$).

```
  let M0 := mxSumProd H y M in
  match Bool.bool_dec (M == M0) true with
  | left H ⇒ M
  | right H ⇒ @SP_BEC0_rec y M0 _ _ _
  end.
...
```

The other arguments of this function are invariants: (1) the input matrix is an "BEC-approximation" (where BEC stands for "binary erasure channel") of an input codeword (predicate `approx_BEC_input` above), and (2) the matrix of a Sum-Product step is always smaller than the input according to $\leq_m$ (hypothesis `mxSumProd H y M` $\leq_m$ `M` above).

The formalization above started as an effort to formalize a combinatorial result about the performance of iterative decoding ([Di et al, 2002, Lemma 1.1] formalized in [Obata, 2015; Infotheo, 2019]).

9.2 Extraction of an Implementation of Sum-Product Decoding over the Binary Symmetric Channel

An implementation of sum-product decoding over a binary symmetric channel takes as input a Tanner graph and an output y, and computes for all variable nodes, each representing a bit of the decoded codeword, its marginal aposteriori probability. One chooses to decode the n_0th bit either as 0 if $P^W_{n_0}(0|y) \geq P^W_{n_0}(1|y)$ or as 1 otherwise, so as to perform Maximum Posterior Marginal decoding (defined in Sect. 3.4).

The algorithm we implement is known in the literature as the forward/backward algorithm and has many applications [Kschischang et al, 2001]. It uses the tree view of an acyclic Tanner graph to structure recursive computations. In a first phase it computes α's and β's (see Sect. 8.3) from the leaves towards the root of the tree, and then computes α's and β's in the opposite direction (starting from the root that time). Figure 3 illustrates this.

Concretely, we provide Coq functions to build the tree, compute α's and β's, and extract the estimations, and prove formally that the results indeed agree with the definitions from Sect. 8.3.

Definition of the tree Function nodes and variable nodes share the same data structure, and are just distinguished by their kind. This allows to factorize traversals and parts of the associated proofs that do not depend on the kind.

```
Definition R2 := (R * R)%type.
Inductive kind : Set := kf | kv.
Definition negk k := match k with kf ⇒ kv | kv ⇒ kf end.
Inductive tag : kind → Set := Func : tag kf | Var : R2 → tag kv.
Inductive tn_tree (k : kind) (U D : Type) : Type :=
  Node { node_id : id;
         node_tag : tag k;
         children : seq (tn_tree (negk k) U D);
         up : U; down : D }.
```

This tree is bipartite by construction, thanks to the type system, as a child is required to have a kind opposite to its parent. Additionally, in each variable node, `node_tag` is expected to contain the channel probabilities for this bit to be 0 or 1, i.e., the pair $(W(y_{n_0}|0), W(y_{n_0}|1))$. The `up` and `down` fields are to be filled with the values of α and β (according to the kind), going to the parent node for `up`, and coming from it for `down`. Here again we will use pairs of the 0 and 1 cases. Note that the values of α's and β's need not be normalized.

Computation of α and β The function `alpha_beta` takes as input the tag of the source node, and the α's and β's from neighboring nodes, excluding the destination, and computes either α, if the source is a function node, or β, if it is a variable node. Thanks to this function, the remainder of the algorithm keeps a perfect symmetry between variable and function nodes.

```
Definition alpha_op (out inp : R2) :=
  let (o, o') := out in let (i, i') := inp in
  (o * i + o' * i', o * i' + o' * i).
Definition beta_op (out inp : R2) :=
  let (o, o') := out in let (i, i') := inp in (o * i, o' * i').
Definition alpha_beta k (t : tag k) : seq R2 → R2 :=
  match t with
  | Func ⇒ foldr alpha_op (1, 0)
  | Var v ⇒ foldl beta_op v
  end.
```

The definition for β is clear enough: assuming that `v` contains the channel probabilities for the corresponding bit, it suffices to compute the product of these probabilities with the incoming α's, which denote the marginal aposteriori probabilities for this bit computed from their respective subgraphs. For α, starting from the `recursive_computation` lemma, we remark that assuming a bit to be 0 leaves the parity unchanged, while assuming it to be 1 switches the parities. This way, the sum-of-products can be computed as an iterated product, using `alpha_op`. This optimization is described in [Kschischang et al, 2001, Sect. 5-E]. We will of course need to prove that these definitions compute the same α's and β's as in Sect. 8.3.

Propagation of α and β Functions `sumprod_up` and `sumprod_down` compute respectively the contents of the `up` and `down` fields. They exploit the symmetry of our node representation, as they do not distinguish between variable and function nodes, only passing on the tag to the combining function `alpha_beta` which selects the right computation.

```
Fixpoint sumprod_up {k} (n : tn_tree k unit unit) : tn_tree k R2 unit :=
  let children' := map sumprod_up (children n) in
  let up' := alpha_beta (node_tag n) (map up children') in
  Node (node_id n) (node_tag n) children' up' tt.
Fixpoint seqs_but1 (a b : seq R2) :=
  if b is b1 :: b2 then (a ++ b2) :: seqs_but1 (rcons a b1) b2 else [::].
Fixpoint sumprod_down {k} (n : tn_tree k R2 unit) (from_above : option R2)
  : tn_tree k R2 R2 :=
  let (arg0, down') :=
    if from_above is Some p then ([:: p], p) else ([::], (1, 1)) in
  let args := seqs_but1 arg0 (map up (children n)) in
  let funs := map
      (fun n' l ⇒ sumprod_down n' (Some (alpha_beta (node_tag n) l)))
      (children n) in
  let children' := apply_seq funs args in
  Node (node_id n) (node_tag n) children' (up n) down'.
Definition sumprod {k} n := sumprod_down (@sumprod_up k n) None.
```

The `from_above` argument is `None` for the root of the tree, or the message coming from the parent node otherwise. The function `seqs_but1 a b` returns the concatenations of `a` and subsequences of `b` omitting just one element. The function `apply_seq` applies a list of functions to a list of arguments. This is a workaround to allow defining `sumprod_down` as a `Fixpoint` recursing on children of `n`.

Building the tree A parity-check matrix `H` and the probability distribution `rW` for each bit (computed from the output `y` and the channel `W`) is turned into a `tn_tree`, using the function `build_tree`, and fed to the above `sumprod` algorithm:

```
Variables (W : 'Ch('F_2, B)) (y : ('U C_not_empty).-receivable W).
Let rW n0 := (W '(y_n0 | 0), W '(y_n0 | 1)).
Let computed_tree := sumprod (build_tree H rW (k := kv) 0).
```

Extraction of estimations We finally recover normalized estimations from the tree:

```
Definition normalize (p : R2) :=
  let (p0, p1) := p in (p0 / (p0 + p1), p1 / (p0 + p1)).
Fixpoint estimation {k} (n : tn_tree k R2 R2) :=
  let l := flatten (map estimation (children n)) in
  if node_tag n is Var _ then
    (node_id n, normalize (beta_op (up n) (down n))) :: l
  else l (* node_tag n is Func *).
```

Correctness The correctness of the algorithm above consists in showing that the estimations computed are the intended aposteriori probabilities:

```
Let estimations := estimation computed_tree.
Definition esti_spec n0 b := 'U C_not_empty '_ n0 '^^ W (b | y).
Definition estimation_spec := uniq (unzip1 estimations) ∧
  ∀ n0, (inr n0, (esti_spec n0 0, esti_spec n0 1)) ∈ estimations.
```

where `esti_spec n0 b` defines the same aposteriori probability $P_{n_0}^W(b|y)$ as appears in lemma `estimation_correctness` from Sect. 8.3.

A formal proof of `estimation_spec` appears as theorem `estimation_ok` in our implementation [Infotheo, 2019, file `ldpc_algo_proof.v`]. As key steps, it uses lemmas `estimation_correctness` and `recursive_computation`.

Applying the algorithm As we mentioned at the beginning of Sect. 8, while proofs for the marginal aposteriori probability require the graph to be acyclic, concrete LDPC codes are usually based on graphs containing cycles. Codes based on acyclic graphs are rare and not very efficient [Etzion et al, 1999]. We still could test our implementation with one of them [Infotheo, 2019, file `sumprod_test.ml`].

While this voids the proofs, it is of course possible to apply our algorithm to graphs containing cycles. A practical approach is to build a tree approximating the graph, by unfolding it to a finite depth, and give it as input to our algorithm. This is essentially equivalent to the more classical iterative approach, where α's and β's for the whole graph are computed repeatedly, propagating them in the graph until the corrected word satisfies the parity checks, and failing if the result is not reached within a fixed number of iterations [Kschischang et al, 2001, Sect. 5].

Such approaches are known to give good results in practice. Recently, some stochastic properties on sets of codes have been proved, telling us that by using codes of growing size we have a growing probability to find a code for which sum-product decoding will approximate the marginal aposteriori probability [Richardson and Urbanke, 2008]. This unfortunately does not remove the need to test extensively every LDPC code used in industrial applications, to ensure that it really behaves in the expected way.

10 Related Work

Coding theory has been considered as an application of the interface between the Isabelle proof-assistant and the Sumit computer algebra system [Ballarin and Paulson, 1999]. In order to take advantage of the computer algebra system, proofs are restricted to a certain code length. Only binary codes are considered (Hamming and BCH). Though the mathematical background about polynomials has been formally verified, results about coding theory are only asserted. In comparison, we formally verify much more (generic) lemmas. Yet, for example when proving that certain bitstrings are codewords, we found ourselves performing formal proofs close to symbolic computation.

We are also aware of a formalization effort for coding theory in the Lean proof-assistant. It includes a proof of the number of errors that can be corrected by repetition codes and a proof that the $(7,4)$ instance of Hamming codes can correct one error [Hagiwara et al, 2016]. It has furthermore been extended with algebraic properties of insertion/deletion codes [Kong et al, 2018].

11 Work in Progress and Future Work

11.1 Toward a Complete Formalization of BCH Codes

There are several ways to pursue our formalization of BCH codes. In this paper, we have formalized *narrow-sense, binary BCH codes* but they can be generalized in several ways [MacWilliams and Sloane, 1977, Chapter 9]. We have also postponed the formalization of a generator polynomial for BCH codes because we expect to be able to have a uniform treatment of encoders for polynomial codes in general.

11.2 Formalization of Goppa Codes

We are now tackling the formalization of binary Goppa codes. They are well-known for their use in the (post-quantum) McEliece cryptosystem. Like Reed-Solomon and BCH codes, Goppa codes can be decoded using the Euclidean algorithm [Sugiyama et al, 1975]. They form a class of *alternant codes*, which can themselves be defined as restricted codes (in the sense of Sect. 7.1) using Generalized Reed-Solomon codes (see Sect. 5.1). They can also be seen as a generalization of a class of BCH codes called *primitive, narrow-sense BCH codes*. We can see that there exist many classes of codes with non-trivial relations between them, whose definition in a single, unified framework raises new formalization challenges.

11.3 Progress on LDPC codes

As explained in Sect. 9.2, to get closer to practical applications, one needs to consider Tanner graphs that contain cycles. We are now working on formalizing results on sum-product decoding for such graphs [Affeldt et al, 2016]. One goal is to show that, under some conditions, one can expect a large Tanner graph to only contain cycles long enough that a localized sum-product algorithm, seeing locally the graph as a tree, will be able to correctly approximate the marginal aposteriori probability [Richardson and Urbanke, 2008, Sect. 3.8]. Though our effort focuses on the restricted case of the binary erasure channel, it gets us closer to formalizing recent research about LDPC codes.

12 Conclusion

In this paper, we have presented a formalization of linear ECCs in the Coq proof-assistant. Table 4 provides an overview of the implementation. This formalization also relies on updated previous work [Affeldt et al, 2014].

Our formalization is complete in the sense that it covers the basic contents of a university-level class on coding theory. For a concrete example, together with our previous work [Affeldt et al, 2014], this paper covers most of the material in a recent textbook [Hagiwara, 2012] used in Japan universities. Like most formalizations, our work has the advantage of sorting out the implicit hypotheses that are made in

textbooks (as discussed for example in Sect. 3.4). We believe that our formalization can support further effort of formalization of coding theory. We tried to isolate reusable libraries. For example, we isolated the formalization of the Euclidean algorithm for decoding (in Sect. 5) that we applied to Reed-Solomon codes (Sect. 6) and to BCH codes (Sect. 7). The formalization of sum-product decoding required us to overcome technical difficulties (see Sect. 8.3) but we are now in a position to consider new formalization problems. For example, the formalization of sum-product decoding over the binary erasure channel (in Sect. 9.1) was also used to prove an original lemma about the performance of iterative decoding [Obata, 2015]. We could also use our formalization to extract a verified implementation of sum-product decoding (see Sect. 9.2). Last, Sect. 11 highlights other challenges whose formalization is made possible by our library.

Acknowledgements T. Asai, N. Obata, K. Sakaguchi, and Y. Takahashi contributed to the formalization. The formalization of modern coding theory was a collaboration with M. Hagiwara, K. Kasai, S. Kuzuoka, and R. Obi. The authors are grateful to M. Hagiwara for his help. We acknowledge the support of the JSPS KAKENHI Grant Number 18H03204.

References

Affeldt R, Garrigue J (2015) Formalization of error-correcting codes: from Hamming to modern coding theory. In: 6th Conference on Interactive Theorem Proving (ITP 2015), Nanjing, China, August 24–27, 2015, Springer, Lecture Notes in Computer Science, vol 9236, pp 17–33

Affeldt R, Hagiwara M, Sénizergues J (2014) Formalization of Shannon's theorems. Journal of Automated Reasoning 53(1):63–103

Affeldt R, Garrigue J, Saikawa T (2016) Formalization of Reed-Solomon codes and progress report on formalization of LDPC codes. In: International Symposium on Information Theory and Its Applications (ISITA 2016), Monterey, California, USA, October 30–November 2, 2016, IEICE, pp 537–541, IEEE Xplore

Affeldt R, Garrigue J, Saikawa T (2019) Reasoning with conditional probabilities and joint distributions in Coq. In: 21st Workshop on Programming and Programming Languages (PPL2019), Iwate-ken, Hanamaki-shi, March 6–8, 2019, Japan Society for Software Science and Technology

Asai T (2014) The SSReflect extension of the Coq proof-assistant and its application. Master's thesis, Graduate School of Mathematics, Nagoya University, in Japanese.

Ballarin C, Paulson LC (1999) A pragmatic approach to extending provers by computer algebra—with applications to coding theory. Fundamenta Informaticae 39(1-2):1–20

Bose RC, Ray-Chaudhuri DK (1960) On a class of error correcting binary group codes. Information and Control 3(1):68–79

Di C, Proietti D, Telatar IE, Richardson TJ, Urbanke RL (2002) Finite-length analysis of low-density parity-check codes on the binary erasure channel. IEEE Transactions on Information Theory 48(6):1570–1579

Etzion T, Trachtenberg A, Vardy A (1999) Which codes have cycle-free tanner graphs? IEEE Trans Inf Theory 45(6):2173–2181

Gallager RG (1962) Low-density parity-check codes. IRE Trans Information Theory 8(1):21–28

Gonthier G, Asperti A, Avigad J, Bertot Y, Cohen C, Garillot F, Roux SL, Mahboubi A, O'Connor R, Biha SO, Pasca I, Rideau L, Solovyev A, Tassi E, Théry L (2013) A machine-checked proof of the odd order theorem. In: 4th International Conference on Interactive Theorem Proving (ITP 2013), Rennes, France, July 22–26, 2013, pp 163–179

Gowers T (ed) (2008) The Princeton companion to Mathematics. Princeton University Press

Hagiwara M (2012) Coding Theory: Mathematics for Digital Communication. Nippon Hyoron Sha, (in Japanese)

Hagiwara M, Nakano K, Kong J (2016) Formalization of coding theory using Lean. In: International Symposium on Information Theory and Its Applications (ISITA 2016), Monterey, California, USA, October 30–November 2, 2016, IEICE, pp 527–531, IEEE Xplore

Hamming RW (1950) Error detecting and error correcting codes. The Bell System Technical Journal XXVI(2):147–160

Hocquenghem A (1959) Codes correcteurs d'erreurs. Chiffres (Paris) 2:147–156

Infotheo (2019) A Coq formalization of information theory and linear error-correcting codes. https://github.com/affeldt-aist/infotheo/, Coq scripts. Version 0.0.2.

Kong J, Webb DJ, Hagiwara M (2018) Formalization of insertion/deletion codes and the levenshtein metric in lean. In: International Symposium on Information Theory and Its Applications (ISITA 2018), Singapore, October 18–31, 2018, IEICE, pp 11–15, IEEE Xplore

Kschischang FR, Frey BJ, Loeliger H (2001) Factor graphs and the sum-product algorithm. IEEE Trans Information Theory 47(2):498–519

MacWilliams FJ, Sloane NJA (1977) The Theory of Error-correcting Codes. Elsevier, seventh impression 1992

Mahboubi A, Tassi E (2016) Mathematical Components. Available at: https://math-comp.github.io/mcb/, with contributions by Yves Bertot and Georges Gonthier.

McEliece RJ (2002) The theory of information and coding, Encyclopedia of Mathematics and its Applications, vol 86. Cambridge University Press

Obata N (2015) Formalization of Theorems about Stopping Sets and the Decoding Performance of LDPC Codes. Department of Communications and Computer Engineering, Graduate School of Science and Engineering, Tokyo Institute of Technology, master's thesis (in Japanese)

Reed IS, Solomon G (1960) Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics (SIAM) 8(2):300–304

Richardson T, Urbanke R (2008) Modern coding theory. Cambridge University Press

Richardson TJ, Urbanke RL (2001) The capacity of low-density parity-check codes under message-passing decoding. IEEE transactions on information theory 47(2):599–618

Sugiyama Y, Kasahara M, Hirasawa S, Namekawa T (1975) A method for solving key equation for decoding Goppa codes. Information and Control 27(1):87–99

The Coq Development Team (2018) The Coq Proof Assistant Reference Manual. INRIA, version 8.9.0

Table 1: Types of mathematical objects

`seq T`	lists of items of type `T`
`'I_n`	integers less than `n`, called "ordinals"
`{ffun A → B}`	functions with a finite domain
`{set A}`	finite sets with elements of type `A`
`'M[R]_(m, n)`	`m`×`n` matrix with elements of type `R`
`'rV[R]_n`, `'cV[R]_n`	row/col-vectors of size `n` with elements of type `R`
`'rV_n`, `'cV_n`	row/col-vectors of size `n` (type of elements automatically inferred)
`{poly A}`	polynomials with coefficients of type `A`
`F_p`	the finite field $\mathbb{F}_{\mathtt{p}}$ where `p` is a prime power
`{vspace vT}`	subspaces of `vT`

Table 2: Construction of mathematical objects

`inord i`	integer `i` seen as an "ordinal" of type `'I_n` determined by the context
`[::]`, `[:: e]`, `::`	empty list, singleton list with the element `e`, list constructor
`[pred x | E x]`	the boolean predicate `E`
`[ffun x ⇒ e]`	the function $x \mapsto e$ with a finite domain
`[set a | P a]`	the finite set of elements `a` that satisfy `P`
`\matrix_(i, j) E i j`	the matrix $[\mathtt{E}_{ij}]_{i,j}$
`1%:M`	the identity matrix
`\row_(i < n) E i`	the row-vector $[E_0, \ldots, E_{n-1}]$
`\poly_(i < n) E i`	the polynomial $E_0 + E_1X + \cdots + E_{n-1}X^{n-1}$
`a%:P`	the constant polynomial `a`
`'X`	the polynomial indeterminate
`a *: v`	`v` scaled by `a`
`0%VS`	the trivial vector space

Table 3: Operations on mathematical objects

`==`, $\neq$, `||`, `&&`, $\Longrightarrow$, $\neg$	boolean operators
`x ^ n`	the `n`th power of `x`, where `x` is a natural or a real number
`n.+1`, `n.-1`, `n.*2`, `./2`	$n+1$, $n-1$, $n\times 2$, $n/2$
`a ^+ n`	the `n`th power of `a`
`a ^- n`	the inverse of `a ^+ n`
`#|E|`	the cardinality of `E`
`S0 \subset S1`	`S0` $\subseteq$ `S1`
`A :\ x`	`A` $\setminus \{$ `x` $\}$
`f @: X`	the image set of `X` under `f`
`\sum_(i in E) e i`	$\sum_{i\in E} e_i$
`\prod_(i in E) e i`	$\prod_{i\in E} e_i$
`*m`	matrix multiplication
`^T`	matrix transpose
`const_mx a`	the constant matrix whose entries are all `a`
`row_mx A B`	the matrix `A||B`
`lsubmx M`	the left sub-matrix with n_1 columns if `M` has n_1+n_2 columns
`map_mx f A`	the pointwise image of `A` by `f`
`\det M`	the determinant of `M`
`e_i`	the `i`th element of row-vector `e`
`p^'()`	formal derivative of the polynomial `p`
`rVpoly c`	the polynomial corresponding to the row-vector `c`
`poly_rV p`	the partial inverse of `rVpoly`
`size p`	$1+\deg(p)$, or 0 if the polynomial `p` is 0 (as an instance of the size of lists)
`p.[x]`	the evaluation of polynomial `p` at point `x`
`p %% q`	remainder of the pseudo-division of `p` by `q`
`p %| q`	tests the nullity of the remainder of the pseudo-division of `p` by `q`

file	contents	l.o.c.*
General-purpose library (`lib` *directory)*		
`hamming.v`	Hamming distance (Sect. 3.1, [Affeldt et al, 2014])	795
`euclid.v`	Euclidean algorithm for decoding (Sect. 5.2)	743
`dft.v`	Discrete Fourier transform (Sect. 6.2) and BCH argument (Sect. 6.5.1)	398
`vandermonde.v`	Vandermonde matrices (Sections 5.1.3 and 7.4.1)	403
Information-theoretic extension (`information_theory` *directory)*		
`pproba.v`	Aposteriori probability (Sect. 2.2.2)	199
Classic linear error-correcting codes (`ecc_classic` *directory)*		
`linearcode.v`	Formalization of linear codes (Sect. 3.1)	846
`repcode.v`	Repetition codes (Sect. 3.3)	251
`decoding.v`	Specifications of decoders (Sect. 3.4)	307
`hamming_code.v`	Hamming codes (Sect. 4)	914
`poly_decoding.v`	Locator, evaluator, and syndrome polynomials (Sect. 5.1)	396
`grs.v`	Generalized Reed-Solomon codes (Sect. 5.1.3)	205
`cyclic_code.v`	Cyclic codes (Sect. 5.3)	503
`reed_solomon.v`	Reed-Solomon codes (Sect. 6)	785
`bch.v`	BCH codes (Sect. 7)	572
Modern coding theory (`ecc_modern` *directory)*		
`subgraph_partition.v`	Bipartite/acyclic graphs, cover/partition properties (Sect. 8.1)	1139
`tanner.v`	Tanner graphs (Sect. 8.1)	195
`tanner_partition.v`	Cover/partition properties of Tanner graphs (used in Sect. 8.3)	1062
`summary.v`	Summary operator (Sect. 8.2)	294
`summary_tanner.v`	Lemmas about the summary operator (e.g., reindexing, see Sect. 8.3.1)	697
`checksum.v`	Indicator function seen in Sect. 8.3.1	159
`ldpc.v`	Properties of sum-product decoding (Sections 8.3.1 and 8.3.2)	801
`ldpc_erasure.v`	Sum-product decoding for the binary erasure channel (Sect. 9.1)	730
`ldpc_algo.v`	Sum-Product decoder (Sect. 9.2)	235
`ldpc_algo_proof.v`	Verification of the Sum-Product decoder (Sect. 9.2)	2028
Extraction of sum-product decoding (`extraction` *directory)*		
`sumprod.{ml,mli}`	OCaml code extracted from the sum-product algorithm (Sect. 9.2)	339
	Total (Coq proof scripts only)	14636

∗ *lines of code computed with* `coqwc`*, comments excluded*

Table 4: Overview of the formalization discussed in this paper. It is a subset of a larger formalization of information theory [Infotheo, 2019].