Formalization of Reed-Solomon codes and progress report on formalization of LDPC codes

Reynald Affeldt[1], Jacques Garrigue[2], and Takafumi Saikawa[2]

[1]National Institute of Advanced Industrial Science and Technology, Japan
[2]Graduate School of Mathematics, Nagoya University

***Abstract*—Error-correcting codes make possible reliable communication over noisy channels. One way to guarantee the correct implementation of error-correcting codes is to use formal verification. This requires in particular the formalization of the mathematical theory of error-correcting codes. This has been made possible by recent advances in the formalization of mathematics using proof-assistants. In this paper, we discuss formalization of linear error-correcting codes: we introduce a formalization of cyclic codes and Euclidean decoding that we apply to Reed-Solomon codes, and we discuss the advanced topic of LDPC codes.**

I. Towards a Formal Coding Theory

This paper is about formal verification of error-correcting codes. Formal verification consists in expressing pencil-and-paper proofs in terms of formal logic, so that they can be checked by a computer. The formal logic in question is the theory of types that the Coq proof-assistant implements (Section II-A). Formal verification is useful because formal proofs are very rigorous. Once a proof is formalized, it can be used together with a formal theory of programming languages to provide bug-free software. In other words, a formal theory of error-correcting codes is the first step towards verified implementation of encoders and decoders.

In this paper, we discuss two topics: formalization of cyclic codes and LDPC codes. Regarding cyclic codes, we do an extensive commentary about formalization, highlighting the main difficulties. In Section III, we define formally cyclic codes. In Section IV, we provide a library of formal definitions and lemmas for Euclidean decoding, primarily used for cyclic codes. In Section V, we apply our results to the formalization of Reed-Solomon codes, including encoding in systematic form and decoding based on the Euclidean algorithm. Regarding LDPC codes, we discuss in Section VI our approach to formalize an important theorem that relates tree and graph ensembles, a key step in proving that sum-product decoding applies even in the presence of cycles.

Our formalization is based on our previous work [1] and on the formalization of polynomials from the Mathematical Components library [6] (Section II-B). We follow pencil-and-paper proofs from standard literature. Formalization of cyclic codes is simplified by the fact that there exist many detailed proofs [7], [8]. We are less fortunate with LDPC codes, since the standard reference [10] does not provide such details.

This paper makes the following contribution. Our formalization of cyclic codes improves on related work: it contains the first formalization of Reed-Solomon codes, which required to extend our previous work [1] to deal with non-binary codes. We end up with very rigorous proofs (as a matter of fact, we were able to spot minor technical errors in our main reference material [7]) whose lemmas can be reused to formalize other cyclic codes. As for LDPC codes, we propose workable formal definitions for tree and graph ensembles.

This is a preprint of a paper to be presented at ISITA 2016 http://www.isita.ieice.org/2016/index.html.

II. Background: Formal Verification with Coq

A. Proof-assistant Based on Type Theory

We use the Coq proof-assistant [4]. It is an implementation of type theory. When we write `a : A`, it means that the object `a` has type `A`. For the purpose of this paper, a type can be understood as a set of objects. Objects are essentially data structures and programs from a foundational language known as the Calculus of Inductive Constructions. It has been shown that such objects can be used to represent proofs: this is the so-called Curry-Howard isomorphism. As a consequence, we also write `p : P` to say that `p` is a proof of the property `P`.

The Coq proof-assistant provides a small set of rules to construct objects and to check their type. This set of rules is trusted to be sound. Users of Coq builds mathematical theories on top on this small trusted base using tactics organized as scripts. A tactic corresponds to a small reasoning step such as lemma application or induction. The objects in Coq are usually written using ASCII characters in a way close to LaTeX commands. The language of Coq and tactics have a precise semantics that we do not detail here for lack of space.

B. The Mathematical Components Library

The Mathematical Components library is a library of formal definitions and lemmas built on top of the Coq proof-assistant and developed to formalize the odd order theorem [5]. 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 Mathematical Components library. It is not our purpose to explain them in details. For quick reference, most notations are summarized in Table I.

TABLE I
MATHEMATICAL COMPONENTS [5] NOTATIONS USED IN THIS PAPER

Type of mathematical objects

`'I_n`	the natural numbers strictly smaller than `n`
`{set A}`	sets with elements of type `A`
`'M[R]_(m, n)`	`m`×`n` matrix with elements of type `R`
`'rV[R]_n`	row-vector of length `n` with elements of type `R`
`{poly F}`	polynomials with coefficients of type `F`

Construction of mathematical objects

`[set a | P a ]`	the set of elements `a` that satisfy `P`
`a%:P`	the constant polynomial `a`
`'X, 'X^d`	the monomial X, the monomial X^d
`\matrix_(i, j) E i j`	the matrix $[\mathtt{E}_{ij}]_{i,j}$
`\poly_(i <n) E i`	the polynomial $\mathtt{E}_0 + \cdots + \mathtt{E}_{n-1}X^{n-1}$
`[ffun x ⇒ E x]`	the finite-domain function $x \mapsto \mathtt{E}\,x$

Operations on mathematical objects

`n.+1`	successor of `n`
`n.-1`	predecessor of `n` (0 if $n = 0$)
`n.*2`	`n + n`
`a ^+ n`	the `n`th power of `a`
`a ^- n`	the inverse of `a ^+ n`
`*m`	matrix multiplication
`v ``_ i`	the `i`th component of the row-vector `v`
`p.[x]`	the evaluation of a polynomial `p` at point `x`
`size p`	$\deg(\mathtt{p}(X)) + 1$
`p %/ q`	quotient of the polynomial division of `p` by `q`
`p %% q`	remainder of the polynomial division of `p` by `q`
`k *: p`	the polynomial `p` scaled by a factor `k`
`rVpoly c`	the polynomial corresponding to the row-vector `c`
`poly_rV p`	the (possibly truncated) vector corresponding to `p`

III. FORMALIZATION OF CYCLIC CODES

A. Linear Error-correcting Codes, Formally

A linear code is essentially a vector space of codewords. Let `F` be a finite field and `n` a natural number. `'rV[F]_n` is the type of row-vectors of length `n` over `F` (Table I). A linear code of length `n` over `F` is formally defined as follows:

```
(* Module Lcode0 *)
Record t F n := mk {space :> {vspace 'rV[F]_n}}.
```

`{vspace _}` is the type of vector spaces provided by the Mathematical Components library. When `F` is $\mathbb{F}_2$, we talk about binary codes.

In practice, we define a linear code as the kernel of a parity-check matrix. Let H be a (parity-check) matrix. $(Hy^T)^T$ is the syndrome of vector y:

```
Definition syndrome y := (H *m y^T)^T.
```

The syndrome is a linear application whose kernel is a vector space. Let `lin_syndrome` be the corresponding linear application. The sought vector space is defined as follows:

```
Definition kernel := lker (linfun lin_syndrome).
```

When dealing with linear codes, it is often useful to regard a codeword $(c_0, c_1, \ldots, c_{n-1})$ as the polynomial $c_0 + c_1X + \cdots + c_{n-1}X^{n-1}$ (formally defined by `rVpoly`, Table I). A polynomial such that the codewords consist of all the polynomials divisible by it is called a *generator*:

```
Definition is_pgen F n (C : {set 'rV[F]_n}) g
  := ∀ x, x ∈ C ↔ g %| rVpoly x.
```

Linear codes for which there exists a generator are called *polynomial codes*.

We make use of above definitions to define Reed-Solomon codes in Section V-A.

B. Cyclic Error-correcting Codes

Let `rcs` be a function that performs a right-cyclic shift, i.e., that turns a vector $(c_0, c_1, \cdots, c_{n-2}, c_{n-1})$ into $(c_{n-1}, c_0, \cdots, c_{n-3}, c_{n-2})$. A set of vectors is stable by right-cyclic shift when it satisfies the predicate `rcsP`:

```
Definition rcsP C := ∀ x, x ∈ C → rcs x ∈ C.
```

A linear cyclic error-correcting code is a linear error-correcting code whose codebook is stable by right-cyclic shift:

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

For cyclic codes, a codeword that is a (non-zero) polynomial of lowest degree is called a *generator*. Let `'cgen[C]` be the set of generators. This definition of generator for cyclic codes coincides with the definition of generator of the previous section (Section III-A):

```
Lemma mem_cgen_dvdp g :
  g \is 'cgen[C] → is_pgen C (rVpoly g).
Lemma cgen_dvd_mem g (C0 : 0%VS ≠ C) :
  is_pgen C (rVpoly g) → g \is 'cgen[C].
```

We will see a concrete example of generator in Section V-C when dealing with Reed-Solomon codes.

IV. FORMAL LIBRARY FOR EUCLIDEAN DECODING

In this section, we provide generic formal definitions and lemmas used to perform Euclidean decoding. We use this library to decode Reed-Solomon codes (Sections V-D and V-E). In the following, `a` is an element from a field `F` such that $a^k \neq 1$ for all $k \in (0, n)$.

A. Error-locator Polynomials

The support set of a vector `e` is defined as follows:

```
Definition supp e := [set i | e ``_ i ≠ 0].
```

Given a vector `e`, the locator polynomial σ`_(a, e)` is the polynomial `errloc a (supp e)`, where `errloc` is defined as follows:

```
Definition errloc a E :=
  ∏_(i in E) (1 - a ^+ i *: 'X).
```

Note that `size` σ`_(a, e)` $\leq$`#|supp e|.+1` (or equivalently $\deg(\sigma$`_(a, e)`$) \leq$`#|supp e|`). The `i`th punctured locator polynomial [8, p. 239] σ`_(a, e, i)` is `errloc a (supp e :\ i)`.

B. Error-evaluator Polynomials

The evaluator polynomial $\omega\_(a, e)$ is defined using the ith punctured locator polynomial:

```
Definition erreval a e := ∑_(i in supp e)
  e ``_ i * a ^+ i *: σ_(a, e, i).
```

Note that size $\omega\_(a, e) \leq t$ when $\#|\text{supp } e| \leq t$.

Error vectors are characterized by the following lemma:

```
Lemma err_vecE a e i : i ∈ supp e →
  e ``_ i = - ω_(a, e).[a ^- i] /
            σ_(a, e)^`().[a ^- i].
```

where ^`() is a notation for the formal derivative. This lemma means that to decode we only need to compute $\omega\_(a, e)$ and $\sigma\_(a, e)$. In the following, we discuss a method to compute the locator and evaluator polynomials. They actually come from the so-called "key equation".

C. Key Equation

1) Syndrome Polynomials: The *syndrome polynomial* is a polynomial equivalent to the syndrome we saw in Section III-A. To define it, it is practical to introduce the so-called "frequency-domain coordinates" [8, p. 237] of a vector y:

```
Definition fdcoor y i := (rVpoly y).[a ^+ i].
```

The syndrome polynomial of y is denoted by \synp_(a, y, t) and is defined as follows:

```
Definition syndromep y :=
  ∑_(k in 'I_t.*2) ((fdcoor y k.+1) *: 'X^k).
```

2) The Key Equation: The key equation is an important relation between the locator polynomial, the syndrome polynomial, and the evaluator polynomial:

```
Lemma equation : σ_(a, y) * \synp_(a, y, t)
  = ω_(a, y) + eqn_mod
```

where eqn_mod %% 'X^(t0 + t) = 0 when $t0 \leq t$.

For the purpose of this paper, let us call the following polynomial r a *key remainder* of p:

```
Definition r a y t t' p :=
  (p * \synp_(a, y, t)) %% 'X^t'.
```

Let $\omega 2\_(a, y, t, t')$ be the key remainder of $\sigma\_(a , y)$. We can derive from the key equation that $\omega 2\_(a, y, t, t')$ is the evaluator polynomial when the cardinality of the support of y is less than t:

```
Lemma errevalE : #| supp y | ≤ t →
 ∀ t0, t0 ≤ t → ω_(a, y)=ω2_(a, y, t, t0 + t).
```

We saw in Sect IV-B that we need a pair of a locator polynomial and an evaluator polynomial to perform decoding. Lemma errevalE indicates moreover that a locator polynomial can give rise to an evaluator polynomial. We therefore just need to focus on finding a locator polynomial.

3) Generalization of the Key Equation: We can generalize the key equation to deal, instead of the locator polynomial, with any polynomial p satisfying the weaker *key condition*:

```
Definition keycond a y t p := ∀ k,
  t ≤ k < t.*2 → (p * \synp_(a, y, t))`_k = 0.
```

The generalized key equation says that p * \synp_(a, y, t) is equal to the key remainder modulo X^{2t} as long as p meets the key condition:

```
Variable p : {poly F}.
Hypothesis keycond_p : keycond a y t p.
Lemma gen_key_equation : p * \synp_(a, y, t)
  = q a y t p * 'X^t.*2 + r a y t t p.
```

q is what we call in this paper the *key quotient* of p; it is defined similarly to the key remainder r.

One can find a p satisfying the generalized key equation with the Euclidean algorithm. Before discussing the Euclidean algorithm, we introduce a specification for locator polynomials.

D. Key Equation and Locator Polynomials

It turns out that we can provide a complete specification of the locator polynomial in terms of the key equation. Concretely, consider a polynomial p such that
(*loc 1*) it is "normalized":

```
Definition normalized := p.[0] = 1.
```

(*loc 2*) its size is less then t + 1 (i.e., its degree is less than t):

```
Definition deg_ub := size p ≤ t.+1.
```

(*loc 3*) it satisfies the key condition and
(*loc 4*) it is relatively prime with the key remainder:

```
Definition key_coprime :=
  coprimep p (Key.r a y t t p).
```

We prove formally that such a polynomial can only be the locator polynomial of Section IV-A (when the cardinality of the support of y is less than t):

```
Lemma errlocP p :
  Errloc.spec a e t p ↔ p = σ_(a, e).
```

This is a consequence of the generalized key equation. The lemma errlocP plays an important role to establish the correctness of Euclidean decoding (Section V-E).

E. The Euclidean Algorithm for Decoding

1) The Setting: The Euclidean algorithm involves four sequences of polynomials r_i, q_i, u_i, and v_i. r_i is defined by iterated modulo:

```
Fixpoint r k :=
  if k is k1.+1 then
    if k1 is k0.+1 then r k0 %% r k1
    else r1
  else r0.
```

q_i is defined such that $r_i = q_{i+2}r_{i+1} + r_{i+2}$. 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. $(v_0, v_1) = (0, 1)$). Given these definitions, it follows in particular that:

```
Lemma vu k :
  v k.+1 * u k - v k * u k.+1 = (- 1)^+k.
Lemma ruv k : r k = u k * r 0 + v k * r 1.
```

2) Stopping Condition: We can show that the size (or equivalently degree) of r_i is strictly decreasing (provided `size r1 <size r0`). Let us assume some `t` such that `t <size r0`. The Euclidean algorithm stops when $\deg(r_i)$ is strictly smaller that `t`. The formalization of such a stopping condition is not immediate because the termination is not *structural*, i.e., it cannot be decided by a simple syntactic criterion. (In comparison, the definition of the sequence `r` just above obviously terminates because `k` is strictly decreasing.) The `ex_maxn` construct of the Mathematical Components library provides a solution.

Consider the following predicate, which holds for indices `k` such that `t <size (‘r i)` (i.e., $t \leq \deg(r_i)$) for all `i : 'I_k.+1` (i.e., $i \leq k$):

```
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:

```
Definition stop' :=
  ex_maxn ex_euclid_cont euclid_cont_size_r.
```

Thus, `stop := stop'.+1` is the first index at which `euclid_cont` does not hold anymore.

3) Relation with the Key Equation: Let us use the Euclidean algorithm with r_0 `='X^t.*2` (i.e., X^{2t}) and r_1 `=\synp_(a, y, t)`. Let ℓ be the `stop` index. We can show that ‘`v_`ℓ satisfies the key condition (using in particular the lemma `ruv` from Section IV-E1):

```
Lemma keycond_vstop : keycond a y t (‘v_ℓ).
```

We also observe that the degree of ‘`v_`ℓ is less than t:

```
Lemma deg_ub_vstop : Errloc.deg_ub t (‘v_ℓ).
```

In other words, ‘`v_`ℓ satisfies conditions (*loc 2)* and (*loc 3)* of locator polynomials. In Section V-E, we use ‘`v_`ℓ to build a polynomial that satisfies in addition (*loc 1)* and (*loc 4).*

V. Formalization of Reed-Solomon Codes

A. Formal Definition of Reed-Solomon Codes

$\left[(a^{i+1})^j\right]_{i\in[0,d-1),j\in[0,n)}$ is a Reed-Solomon parity-check matrix. Formally, this matrix has type `'M[F]_(d.-1, n)` and is defined as follows:

```
Definition PCM := \matrix_(i, j) (a ^+ i.+1) ^+ j.
```

The Reed-Solomon `codebook` consists of the vectors whose syndrome is 0. The Reed-Solomon code is the kernel of the matrix `PCM` (we use the `kernel` function of Section III-A):

```
Definition code := mkLcode0 (kernel PCM).
```

B. Reed-Solomon Generator Polynomial

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

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

It is a generator in the sense of Section III-A, i.e., the codewords are exactly the polynomials that are divisible by `\gen_(a, d)`:

```
Lemma rs_gen_is_pgen :
  is_pgen (codebook a n d) \gen_(a, d).
```

We can also show that `\gen_(a, d)` is a generator in the sense of Section III-B. We first show that Reed-Solomon codes can be viewed as cyclic. The additional hypothesis that `a` is primitive is a sufficient condition:

```
Lemma RS_cyclic : a ^+ n = 1 →
  rcsP [set cw in code a n d].
```

Then `\gen_(a, d)` can be shown to belong to the set of generators (of the cyclic Reed-Solomon code):

```
Lemma rs_gen_is_cgen (a1 : a ^+ n = 1) :
  let C := Ccode.mk (RS_cyclic a1) in
  poly_rV \gen_(a, d) \is 'cgen[C].
```

C. Systematic Reed-Solomon Encoding

The generator polynomial provides a first encoding technique. Given a message $m = m_0 + \cdots + m_{n-d}X^{n-d}$, the polynomial $m \cdot g$ is a codeword.

Systematic-form encoding is a preferred way of encoding because the message appears in clear in the codeword. $mX^{d-1} - (mX^{d-1})\%g$ provides systematic encoding of message m (the term mX^{d-1} makes room for check symbols):

```
Definition encoder := let g := \gen_(a, d) in
  [ffun m ⇒ let mxd := rVpoly m * 'X^d.-1 in
    poly_rV (mxd - mxd %% g)].
```

The function `encoder` indeed provides systematic encoding. We show more precisely that the bits of higher weight of a codeword are actually the message it encodes:

```
Lemma RS_enc_surjective c :
  c ∈ codebook a n d → encoder (high c) = c.
```

where the `high` part of a codeword `c` is defined as `poly_rV (rVpoly c %/ 'X^d.-1)`.

D. Decoding using the Euclidean Algorithm

The input of the decoder is a possibly corrupted “codeword” $y = c + e$, where c is the transmitted codeword. It outputs the error vector e, whose hamming weight is smaller than $t \leq (d-1)/2$. Decoding is performed as follows:
(1) Compute the sequences ‘`r` and ‘`v` like in Section IV-E with r_0 `='X^t.*2` and r_1 `=\synp_(a, y, t)`. Let ℓ b the `stop` index.
(2) Define the polynomials `nvstop` to be $\sigma = v_\ell/v_\ell(0)$ and `rstop` to be $\omega = r_\ell/v_\ell(0)$:

```
Definition nvstop := ‘v_ℓ * ((‘v_ℓ).[0]^-1)%:P.
Definition rstop := ‘r_ℓ * ((‘v_ℓ).[0]^-1)%:P.
```

(3) Define the polynomial $\sum_{i=0}^{n-1}[\sigma(a^{-i}) = 0]\frac{-\omega(a^{-i})}{\sigma'(a^{-i})}X^i$ ($[\cdots]$ are Iverson brackets and σ' is the formal derivative of σ):

```
Definition euclid_err :=
  let sigma' := deriv nvstop in
  \poly_(i < n)
    if nvstop.[a^- i] = 0 then
      - rstop.[a ^- i] / sigma'.[a ^- i]
    else 0.
```

E. Correctness of the Decoder of Section V-D

In Section IV-E3, we already proved that the output ‘v_ℓ of the Euclidean algorithm meets the specification (*loc 2)* and (*loc 3)* of locator polynomials. These results are also true for nvstop. We now show that nvstop is normalized (*loc 1)*:

```
Lemma normalized_nvstop :
  normalized (nvstop a y t).
```

(which requires to show (‘v_ℓ).[0] $\neq 0$) and that it meets the specification (*loc 4)* of locator polynomials:

```
Lemma key_coprime_nvstop :
  key_coprime a e t (nvstop a y t).
```

We can therefore use the lemma errlocP (Section IV-D) to prove that nvstop is the locator polynomial:

```
Lemma nvstop_errloc : nvstop a y t = σ_(a, e).
```

In Section IV-C2, we saw that the lemma errevalE turns rstop into the evaluator polynomial ω_(a, e). It follows that Reed-Solomon decoding (Section V-D) computes the polynomials of the error vector characterization lemma (Section IV-B). We can therefore use the latter to prove the correctness of Reed-Solomon decoding:

```
Lemma RS_err : euclid_err a y t = rVpoly e.
```

VI. Progress on LDPC codes

In another branch of our work on the formalization of linear codes, we have been working on LDPC codes and sum-product decoding [1]. In the already published part of this work, we have proved properties of the algorithm under the classical assumption that the graph has no cycle (i.e., this is actually a tree). As this assumption does not hold in practice, we have since then started the analysis of decoding for a *computation graph ensemble*, as described in [10]. For simplicity, we are now working on the binary erasure channel case, but we can already see the daunting difficulties of the task.

Our first target has been to prove Theorem 3.49 of [10], the so-called *Convergence to Tree Channel*.

Theorem 1: For a given degree distribution pair (λ, ρ) consider the associated ensembles $\mathrm{LDPC}(n, \lambda, \rho)$ for increasing block-length n under l rounds of BP (sum-product) decoding. Then

$$\lim_{n\to\infty} \mathbf{E}_{G\in\mathrm{LDPC}(n,\lambda,\rho)}[\mathrm{P}_{\mathrm{b}}^{\mathrm{BP}}(G,\epsilon,l)] = \mathrm{P}_{\mathring{\mathcal{T}}_l(\lambda,\rho)}^{\mathrm{BP}}(\epsilon).$$

This theorem is interesting, as it relates an ensemble of complete graphs, of fixed global degree distribution, and which may contain cycles, on one side, with an idealized ensemble of infinite trees, whose degree distribution is probabilistic and independent for each node, on the other side.

Our formalization starts by defining the notions of ensemble for both trees and computation graphs. This is already technical, as the notion of distribution we use only applies to finite sets. We had to prove that the set of trees of depth l that we consider is indeed finite, and use this fact in the ensuing definitions.

At this point we have only been able to prove a small part of the above theorem, saying that the probability that a partial graph of radius l, whose degree distribution for each node follows (λ, ρ), has no cycle, tends to 1 as the size of the graph grows. Note that our proof covers both regular and irregular cases, whereas the only proof we could find in the literature [9, Appendix A] only covers the regular case. This step is actually only a sub-part of the last equation in the proof of [10], but it already required several thousands lines of code (cf. degree_profile.v in Table II).

TABLE II
Coq files discussed in this paper (see also [2])

File name	*Relevant sections*	*l.o.c.*
linearcode.v	Section III-A (and [1])	705
cyclic_code.v	Section III-B	559
euclid.v	Section IV-E	404
cyclic_decoding.v	Section IV (excl. Section IV-E)	1010
reed_solomon.v	Section V	814
degree_profile.v	Section VI	5061

VII. Conclusion and Related Work

In this paper, we reported on an original formalization of cyclic codes, Euclidean decoding, and Reed-Solomon codes, as well as on our progress regarding formalization of LDPC codes (Coq scripts summarized in Table II).

There is not much related work on the topic of formalization of error-correcting codes. This can be explained by the fact that this is only recently that substantial theories of formal mathematics have been made available. We have already commented on our previous work [1]. There is also a piece of work done in the Isabelle proof-assistant, in which 2-error-correcting BCH codes were formalized as part of a case study for an interface between computer algebra and theorem prover [3]. Our formalization is not restricted to the binary case, this is why we can apply it to Reed-Solomon codes.

As for future work, we plan to apply our formal library to the formalization of BCH codes and to further investigate Goppa codes, whose decoding involves techniques similar to the ones discussed in this paper.

Acknowledgment

This work was partially supported by a JSPS Grant-in-Aid for Scientific Research (Project Number: 25289118) and has benefited from the advice of the project members. The authors are grateful to the reviewers for their comments.

REFERENCES

[1] R. Affeldt and J. Garrigue, "Formalization of Error-correcting Codes: from Hamming to Modern Coding Theory," in *6th Int. Conf. Interactive Theorem Proving*, Nanjing, China, 2015, LNCS vol. 9236, pp. 17–33.

[2] R. Affeldt, J. Garrigue, and T. Saikawa. *Coq scripts for this paper* (2016, July 31) [Online]. Available: https://staff.aist.go.jp/reynald.affeldt/ecc/.

[3] C. Ballarin and L. C. Paulson, "A Pragmatic Approach to Extending Provers by Computer Algebra—with Applications to Coding Theory," *Fundam. Inform.,* vol. 34, no. 1–2, pp. 1–20, 1999.

[4] The Coq Development Team, "Reference Manual," INRIA, 1999–2016. Ver. 8.5pl2. Available at http://coq.inria.fr.

[5] G. Gonthier et al., "A Machine-Checked Proof of the Odd Order Theorem," in *4th Int. Conf. Interactive Theorem Proving*, Rennes, France, 2013, LNCS vol. 7998, pp. 163–179.

[6] G. Gonthier, A. Mahboubi, and E. Tassi, "A small scale reflection extension for the Coq system," INRIA, Rep. RR-6455, 2015. Ver. 16.

[7] M. Hagiwara, *Coding Theory: Mathematics for Digital Communication.* Nippon Hyoron Sha, 2012. In Japanese.

[8] R. J. McEliece, *The Theory of Information and Coding.* Encyclopedia of Mathematics and its Applications 86. Cambridge University Press, 2002. Second Edition.

[9] T. Richardson, R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inf. Theory*, vol. 47, pp. 599–618, Feb. 2001.

[10] T. Richardson, R. Urbanke, *Modern Coding Theory.* Cambridge University Press, 2008.