Examples of Formal Proofs about Data Compression

Reynald Affeldt
National Institute of
Advanced Industrial Science and Technology

Jacques Garrigue
Grad. School of Mathematics
Nagoya University

Takafumi Saikawa
Grad. School of Mathematics
Nagoya University

***Abstract*— Because of the increasing complexity of mathematical proofs, there is a growing interest in formalization using proof-assistants. In this paper, we explain new formal proofs of standard lemmas in data compression (Jensen's and Kraft's inequalities) as well as concrete applications (to the analysis of compression methods and Shannon-Fano codes). We explain in particular how one turns the paper proof into formal terms and the relation between the informal proof and the formal one. These formalizations come as an extension to an existing formal library for information theory and error-correcting codes.**

I. Motivation and Contents

A proof-assistant is a piece of software to check mathematical proofs written in formal logic. A formal logic is essentially a minimum set of non-contradictory axioms. Most proof-assistants use (variants of) *type theory*, whose expressiveness is equivalent to set theory. In a proof-assistant, the only trusted part is the checking algorithm: all formal proofs need to be checked to integrate a formal library. However, it requires much expertise to turn a paper proof into formal logic and such experiments are mostly discussed in dedicated conferences.

In the past years, we have been working on the formalization of information theory and error-correcting codes for the Coq proof-assistant [10] using the MathComp library [11]. The result is another library (called InfoTheo [13]) that made it possible to verify Shannon's theorems [4], [5], as well as error-correcting codes [6], [8].

Our goal is now to turn the InfoTheo library into a practical tool. To achieve this goal, we are now involved in the following activities: enrich the library with new lemmas, make it easier to use already-formalized results to prove new ones, and publicize the whole library to potential users. These are the three goals of this paper. Concretely, in this paper, we provide new formal proofs of standard lemmas in data compression (namely, Jensen's and Kraft's inequalities), work out concrete applications (analysis of compression methods and Shannon-Fano codes), and explain our results so as to be understood by readers who may not be familiar with proof-assistants.

The only effort that we require from our readers is to cope with the syntax of formal logic (as implemented by the Coq proof-assistant). It is not difficult to decipher formal statements, especially when one already has the corresponding mathematical background. Indeed, modern parsing technologies allow for familiar LaTeX-like notations. In contrast, we do not expect the reader to read the details of proof scripts (the sequence of commands that symbolically manipulate formal statements to prove a goal) but we display them to show how (well) they match paper proofs.

II. Jensen's Inequality and Analysis of Compression Methods

A. Jensen's Inequality, Informally

Jensen's inequality is essentially a generalization of concavity for a finite set of points. Concavity of a real function f can be stated as:

$$\forall a, b > 0 \quad \frac{af(x) + bf(y)}{a+b} \le f\left(\frac{ax+by}{a+b}\right).$$

Jensen's inequality is a consequence of concavity, and can be stated as:

$$\forall a_1, \ldots, a_n > 0 \quad \frac{\sum a_i f(x_i)}{\sum a_i} \le f\left(\frac{\sum a_i x_i}{\sum a_i}\right).$$

B. Jensen's Inequality, Formally

While the above definition gives the gist of Jensen's inequality, when formalizing it is important both to use simple definitions (with few variables and conditions), and to make sure that the proven theorem can be applied in a wide range of situations. As we want it to apply to locally concave functions, we start by defining intervals, as convex subsets of $\mathbb{R}$.

```
Definition convex_interval (D:R -> Prop) := forall x y t,
  D x -> D y -> 0 <= t <= 1 -> D (t * x + (1-t) * y).
Record interval := mkInterval {
  mem_interval :> R -> Prop;
  interval_convex : convex_interval mem_interval }.
```

We then define concavity, using intervals as predicates.

```
Definition concave_leq (f : R -> R) (x y t : R) :=
  t * f x + (1 - t) * f y <=  f (t * x + (1 - t) * y).
Definition concave_in (D : interval) (f : R -> R) :=
  forall x y t : R, D x -> D y -> 0 <= t <= 1 ->
    concave_leq f x y t.
```

When stating Jensen's inequality, we replace the points and weights by a function `r` and a distribution `X`, both defined over a finite domain `A`.

```
Variables (f : R -> R) (D : interval).
Hypothesis concave_f : concave_in D f.
Variable A : finType.
Theorem jensen_dist_concave (r : A -> R) (X : dist A) :
  (forall x, D (r x)) ->
    \rsum_(a in A) f (r a) * X a
    <= f (\rsum_(a in A) r a * X a).
```

Using a distribution tells us that the weights are non-negative, and their sum is 1, in the same way as `interval` brings its

This is a preprint of a paper to be presented at ISITA 2018 (`http://www.isita.ieice.org/2018/home.html`).

invariant. This theorem is proved by induction on the size of the support of the distribution (i.e., the a's of `A` such that `X` $a \neq 0$) [2, Thm. 2.6.2]. Formally, we follow this approach by first defining such an induction principle, and using it to structure the proof, which as a result is only about 60 lines long. Note that the original proof used a binary distribution as base case, but it is simpler and more general to use a trivial unary distribution (i.e., `X` $a = 1$ for a single $a \in$ `A`).

C. Application: Analysis of Compression Methods

A simple application of Jensen's inequality is comparing the zero-order entropy of strings to that of their concatenation.

Let N^s_S be the number of occurrences of the letter s in the string S. The zero-order empirical entropy of a string S of length $n = |S|$ over the alphabet $\Sigma = \{s_1, \ldots, s_\sigma\}$ is defined using the Shannon entropy of its observed probabilities [9, Sec. 2.3.2]:

$$\mathcal{H}_0(S) = \mathcal{H}\left(\left\langle \frac{\mathsf{N}^{s_1}_S}{n}, \ldots, \frac{\mathsf{N}^{s_\sigma}_S}{n} \right\rangle\right) = \sum_{s\in\Sigma} \frac{\mathsf{N}^s_S}{n} \log \frac{n}{\mathsf{N}^s_S}.$$

This leads to the simple formula: $n\mathcal{H}_0(S) = \displaystyle\sum_{s\in\Sigma} \mathsf{N}^s_S \log \frac{n}{\mathsf{N}^s_S}$.

If we consider two strings S_1 and S_2, of lengths n_1 and n_2 and their concatenation S of length $n = n_1 + n_2$, we have

$$\begin{aligned} n_1\mathcal{H}_0(S_1) + n_2\mathcal{H}_0(S_2) &= \sum_{s\in\Sigma} \mathsf{N}^s_{S_1} \log \frac{n_1}{\mathsf{N}^s_{S_1}} + \mathsf{N}^s_{S_2} \log \frac{n_2}{\mathsf{N}^s_{S_2}} \\ &\le \sum_{s\in\Sigma} (\mathsf{N}^s_{S_1} + \mathsf{N}^s_{S_2}) \log \frac{n_1 + n_2}{\mathsf{N}^s_{S_1} + \mathsf{N}^s_{S_2}} = n\mathcal{H}_0(S), \end{aligned}$$

where the inequality is obtained by applying Jensen at each s with parameters $a_i = \mathsf{N}^s_{S_i}, x_i = \frac{n_i}{\mathsf{N}^s_{S_i}}$, $f = \log$ [9, Sec. 2.8].

To make the example more convincing, we will apply it to the concatenation S of ℓ strings S_i (of length n_i). Namely, $\displaystyle\sum_{i=1}^{\ell} n_i\mathcal{H}_0(S_i) \le n\mathcal{H}_0(S)$, or in Coq's language:

```
Variable A : finType.
Definition nHs (s : seq A) :=
 \rsum_(a in A) if N(a|s) == 0%nat then 0 else
                 N(a|s) * log (size s / N(a|s)).
Theorem concats_entropy (ss : seq (seq A)) :
  \rsum_(s <- ss) nHs s <= nHs (flatten ss).
```

Here, `nHS` is the formalization of the function $S \mapsto |S|\mathcal{H}_0(S)$. We represent strings as sequences of symbols from an alphabet `A`. The length of a sequence `s` is `size s`. `N(a|s)` is a notation for $\mathsf{N}^{\mathsf{a}}_{\mathsf{s}}$. `\rsum_(a in A)` stands for the iterated sum[1] $\sum_{\mathsf{a}\in\mathsf{A}}$. In the formal statement, we represent the ℓ strings S_i as a sequence of sequences `ss`, so that we sum directly over `ss` instead of the indices $\{1, \ldots, \ell\}$, and concatenate the strings using the `flatten` operation on this sequence.

Like we saw in the informal definition above, `nHs` is related to the entropy of the symbol distribution as follows:

```
Lemma szHs_is_nHs (s : seq A) (H : size s != 0) :
  size s * `H (@num_occ_dist s H) = nHs s.
```

The formal proof of `concats_entropy` is shown in Fig. 1. Rather than just seeing it as an artefact, we will try to show how close it is to a paper proof. Some of the theorems used for transformations are given in Fig. 2. The proof goes through the following steps, which are marked in the script.

1) Expand definitions and inverse the order of the summations (`exchange_big`), so that the goal becomes:
$$\begin{aligned} &\sum_{s\in\Sigma}\sum_{i=1}^{\ell} \texttt{if } \mathsf{N}^s_{S_i} = 0 \texttt{ then } 0 \texttt{ else } \mathsf{N}^s_{S_i} \log \frac{|S_i|}{\mathsf{N}^s_{S_i}} \\ &\quad \le \sum_{s\in\Sigma} \texttt{if } \mathsf{N}^s_S = 0 \texttt{ then } 0 \texttt{ else } \mathsf{N}^s_S \log \frac{|S|}{\mathsf{N}^s_S} \end{aligned}$$
2) Show that it is sufficient to prove the inequalities for each s, removing strings which contain no occurrences of s (for simplicity of the indexing, let us pretend that the ℓ' first strings contain occurrences, the formal proof does a reordering).
$$\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i} \log \frac{|S_i|}{\mathsf{N}^s_{S_i}} \le \left(\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}\right) \log \frac{|S|}{\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}}$$
Note that $|S|$ on the right-hand side still contains the lengths of the ommitted strings. This proof makes repeated use of `bigID` and `big1` to separate and remove those strings in other sums.
3) Prove the inequality
$$\left(\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}\right) \log \frac{\sum_{i=1}^{\ell'} |S_i|}{\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}} \le \left(\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}\right) \log \frac{\sum_{i=1}^{\ell} |S_i|}{\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}}$$
using the monotonicity of log. Here the reasoning mostly involves real arithmetic, with the need to prove many side conditions on lemmas. In the formal proof, to conclude we need to work with the concrete indexing, which uses a filtered sequence rather than ℓ'.
4) Define the distribution d and the point function r as:
$$d(i) = \frac{\mathsf{N}^s_{S_i}}{\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}} \qquad r(i) = \frac{|S_i|}{\mathsf{N}^s_{S_i}}$$
and prove that $r(i)$ is strictly positive. In the script, `seq_nat_dist` builds a distribution from a non-zero summation of natural numbers, `in_tuple` turns a sequence into a tuple, which is just a fixed-length sequence, and `tnth` extracts its i^{th} element.
5) Use these proofs and the concavity of log to apply `jensen_dist_concave`. This gives:
$$\sum_{i=1}^{\ell'} \left(\log \frac{|S_i|}{\mathsf{N}^s_{S_i}}\right) \frac{\mathsf{N}^s_{S_i}}{\sum_{i=1}^{\ell'} |S_i|} \le \log \left(\sum_{i=1}^{\ell'} \frac{|S_i|}{\mathsf{N}^s_{S_i}} \frac{\mathsf{N}^s_{S_i}}{\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}}\right).$$
The formal proof uses `big_tnth` to convert from summing using numerical indices, as required by `jensen_dist_concave`, to summing directly on the elements of the sequence, which is the preferred approach throughout this proof. We choose not to distinguish both versions in the mathematical notation.
6) Multiply both sides by $\sum_{i=1}^{\ell'} \mathsf{N}^s_{S_i}$, and use distribution laws and algebraic laws to obtain:

[1] Note that definitions may contain implicit coercions: `size` or `N(a|s)` are natural numbers, but the arithmetic operators expect real numbers, so that a coercion from $\mathbb{N}$ to $\mathbb{R}$ is added. It can also be written explicitly as n`%:R`.

```
Theorem concats_entropy ss :
  \rsum_(s <- ss) nHs s <= nHs (flatten ss).
Proof.
rewrite exchange_big /nHs /=.
 Goal as in (1)
(* Move to per-symbol inequalities *)
apply ler_rsum=> a _.
(* Remove strings containing no occurrences *)
rewrite (bigID (fun s => N(a|s) == O)) /=.
rewrite big1; last by move=> i ->.
rewrite num_occ_flatten add0R.
rewrite [in X in _ <= X]
        (bigID (fun s => N(a|s) == O)).
rewrite [in X in _ <= X]big1 //= ?add0n;
  last by move=> s /eqP.
rewrite (eq_bigr
  (fun s => log (size s / N(a|s)) * N(a|s)));
  last by move=> s /negbTE ->; rewrite mulRC.
rewrite -big_filter -[in X in _ <= X]big_filter.
(* ss' contains only strings with occurrences *)
set ss' := [seq s <- ss | N(a|s) != O].
 Goal as in (2), using ss'
case/boolP: (ss' == [::]) => Hss'.
  by rewrite (eqP Hss') !big_nil eqxx.
have Hnum s : s \in ss' -> (N(a|s) > 0)%nat.
  by rewrite /ss' mem_filter lt0n => /andP [->].
have Hnum': 0 < N(a|flatten ss').
  apply /ltR0n; destruct ss' => //=.
  rewrite /num_occ count_cat ltn_addr //.
  by rewrite Hnum // in_cons eqxx.
have Hsz: 0 < size (flatten ss').
  apply (ltR_leR_trans Hnum').
  by apply /le_INR /leP /count_size.
apply (Rle_trans _ ((\sum_(i <- ss') N(a|i))%:R
    * log (size (flatten ss') /
         \sum_(i <- ss') N(a|i)))); last first.
(* Compensate for removed strings *)
case: ifP => Hsum.
  by rewrite (eqP Hsum) mul0R.
 Goal as in (3)
apply leR_wpmul2l => //.
apply Log_increasing_le => //.
  apply/mulR_gt0 => //.
  apply/invR_gt0/ltR0n.
  by rewrite lt0n Hsum.
apply leR_wpmul2r.
  apply /Rlt_le /invR_gt0 /ltR0n.
  by rewrite lt0n Hsum.
apply /le_INR /leP.
 size (flatten ss') <= size (flatten ss)
rewrite !size_flatten !sumn_big_addn.
rewrite !big_map big_filter.
rewrite [in X in (_ <= X)%nat]
  (bigID (fun s => N(a|s) == O)) /=.
by apply leq_addl.
(* (4) Prepare to use jensen_dist_concave *)
have Htotal := esym (num_occ_flatten a ss').
rewrite big_tnth in Htotal.
have Hnum2 : N(a|flatten ss') != O.
  rewrite -lt0n -ltR0n'; exact/ltRP.
set d := seq_nat_dist Htotal Hnum2.
set r := fun i =>
  (size (tnth (in_tuple ss') i))
  / N(a|tnth (in_tuple ss') i).
 Here d and r are as in (4)
have Hr: forall i, Rpos_interval (r i).
  rewrite /r /= => i.
  apply Rlt_mult_inv_pos; apply /ltR0n.
    apply (@leq_trans N(a|tnth (in_tuple ss') i)).
      by rewrite Hnum // mem_tnth.
    by apply count_size.
  by apply /Hnum /mem_tnth.
(* (5) Apply Jensen *)
move: (jensen_dist_concave log_concave d Hr).
rewrite /d /r /=.
rewrite -(big_tnth _ _ _ xpredT
  (fun s => log (size s / N(a|s))
          * (N(a|s) / N(a|flatten ss')))).
rewrite -(big_tnth _ _ _ xpredT
  (fun s => size s / N(a|s)
          * (N(a|s) / N(a|flatten ss')))).
(* (6) Transform the statement to match the goal *)
 Premise is as in (5), goal as in (6)
move/(@leR_wpmul2r N(a|flatten ss') _ _ (leR0n _)).
rewrite !big_distrl /=.
rewrite (eq_bigr
  (fun i => log (size i / N(a|i)) * N(a|i)));
  last first.
  move=> i _; rewrite !mulRA -mulRA mulVR ?mulR1 //.
  exact/eqP/gtR_eqF.
move/Rle_trans; apply. (* LHS matches *)
rewrite mulRC -num_occ_flatten big_filter.
rewrite (eq_bigr
  (fun i => size i * / N(a|flatten ss')));
  last first.
  move=> i Hi; rewrite !mulRA -(mulRA _ (/ _)).
  by rewrite mulVR ?mulR1 // INR_eq0'.
rewrite -big_filter -/ss' -big_distrl.
rewrite -big_morph_plus_INR /=.
by rewrite size_flatten sumn_big_addn big_map.
Qed.
```

Fig. 1. Proof for the zero-degree entropy of a concatenation of strings.

eq_bigr $g : (\forall i \in I,\ f(i) = g(i)) \to \sum_{i \in I} f(i) = \sum_{i \in I} g(i)$

exchange_big : $\sum_{i \in I} \sum_{j \in J} f(i,j) = \sum_{j \in J} \sum_{i \in I} f(i,j)$

bigID P : $\sum_{i \in I} f(i) = \sum_{i \in \{i \in I \mid P(i)\}} f(i) + \sum_{i \in \{i \in I \mid \neg P(i)\}} f(i)$

big1 : $(\forall i \in I,\ f(i) = 0) \to \sum_{i \in I} f(i) = 0$

big_distrl : $\left(\sum_{i \in I} f(i)\right) \cdot c = \sum_{i \in I} f(i) \cdot c$

leq_rsum : $(\forall i \in I,\ f(i) \leq g(i)) \to \sum_{i \in I} f(i) \leq \sum_{i \in I} g(i)$

Fig. 2. Some theorems concerning summations.

$$\sum_{i=1}^{\ell'} \mathsf{N}^{s}_{S_i} \log \frac{|S_i|}{\mathsf{N}^{s}_{S_i}} \leq \left(\sum_{i=1}^{\ell'} \mathsf{N}^{s}_{S_i} \right) \log \frac{\sum_{i=1}^{\ell'} |S_i|}{\sum_{i=1}^{\ell'} \mathsf{N}^{s}_{S_i}}$$

which lets us conclude combining (3) and (5). The machine proof contains some extra transformations at this stage, using morphisms to turn a sum on natural numbers into a sum of real numbers for instance.

While a paper proof would usually ignore many of these steps, we think that they are important, as there might be hidden subtleties. The need of special handling for $\mathsf{N}^{s}_{S_i} = 0$, which in turn caused the addition of step (3), overlooked in [9, Sec. 2.8], was actually discovered doing the formal proof.

III. Kraft's Inequality and Shannon-Fano Codes

A. Formal Definition of Codes and Prefix Codes

A code is essentially a set of codewords that we formalize as lists of elements (type `seq T` below). The code itself becomes a sequence of codewords (type `seq (seq T)`) without duplicates (i.e., it is `uniq`):

```
Record code_set := CodeSet {
  codeset :> seq (seq T) ; _ : uniq codeset }.
```

This formal definition is fine because it will let us recover the expected properties. Yet, it distinguishes codes with a different ordering of codewords. Of course, properties can be established modulo ordering, but it could be more practical to define codes using *sets* of codewords. This enhancement is not immediate for technical reasons; we defer it to future work.

A prefix code is a code s.t. no codeword is a prefix of any other. We define the prefix relation by comparing the head symbols of codewords (the function `take` n below extracts the n leading elements of a list). The property of being a prefix code follows (`~~` is the logical negation):

```
Definition prefix a b := a == take (size a) b.
Definition prefix_code (C : code_set T) :=
  forall c c', c \in C -> c' \in C -> c != c' ->
  ~~ prefix c c'.
```

Example of Code: Let us consider a sorted list ℓ of n natural numbers and a finite alphabet T. For all $j \in [0, n-1]$, we define the following natural numbers: $w_j = \sum_{i<j} |T|^{\ell_j - \ell_i}$. Let $(x)_b$ be the representation of the natural number x in base b. For each w_j, we construct the list of symbols σ_j by prepending $\ell_j - |(w_j)_{|T|}|$ 0's to $(w_j)_{|T|}$. The σ_j's form a code because the function $j \mapsto \sigma_j$ is injective. Let us call `ACode` the resulting code. We omit its formal definition in this paper (see [13]) but we will use it in the proof script of Kraft's inequality (Sec. III-C3).

B. Kraft's Inequality, Informally

Kraft's inequality is a necessary and sufficient condition for the existence of a prefix code. It says that, given the lengths $\ell_0, \ldots, \ell_{n-1}$, there exists a prefix code with n codewords C_i s.t. $|C_i| = \ell_i$ iff $\sum_{i<n} |T|^{-\ell_i} \leq 1$, where T is the alphabet for the codewords.

To prove the direct part, let $\ell_{\max}$ be the largest length. Then:

$$\sum_{i<n} |T|^{\ell_{\max}-\ell_i} = \sum_{i<n} |\{x | \text{prefix } C_i \ x\}| \tag{III.1}$$

$$= \left| \bigcup_{i<n} \{x | \text{prefix } C_i \ x\} \right| \tag{III.2}$$

$$\leq |T|^{\ell_{\max}}. \tag{III.3}$$

In particular, the equality (III.1)–(III.2) holds thanks to the prefix property. See Sec. III-C2 for a complete formal proof.

Conversely, suppose that we are given a sorted list ℓ of n lengths that satisfies the Kraft condition (for some alphabet T). Then the code of Sec. III-A can be proved to be prefix. The proof is by contradiction [1]. Let us assume *ab absurdo* that this code is not prefix. Then one can find j and k s.t. $j < k$ and σ_j is a prefix of σ_k. Let $r = \frac{w_k}{|T|^{\ell_k-\ell_j}}$. We can establish a contradiction by showing that both $r \geq w_j+1$ and $r-1 < w_j$ hold. See Sec. III-C3 for the (formal) details.

C. Kraft's Inequality, Formally

1) The Kraft Condition: The formalization of the Kraft condition is direct using elements of the MathComp library:

```
Definition kraft_cond (T : finType) (l : seq nat) :=
  let n := size l in
  (\sum_(i < n) #|T|%:R ^- l``_i <= (1 : R))%R.
```

`#|T|` is a notation for the cardinal of `T`. `^-` is the inverse of the exponent. ``l``_i`` is a notation for ℓ_i.

2) Kraft's Inequality (direct part): We display the script corresponding to the (first part of the) informal proof of Sec. III-B. The intermediate steps appear in the form of comments that match the informal proof:

```
Lemma prefix_implies_kraft_cond : prefix_code C ->
  0 < #|T| -> kraft_cond R T (map size C).
Proof.
move=> prefixC T_gt0; rewrite /kraft_cond size_map -/n.
at this point, the goal is $\sum_{i<n} |T|^{-\ell_i} \leq 1$
have /ler_pmul2l <- : ((0 : R) < #|T|%:R ^+ lmax)%R.
  by rewrite exprn_gt0 // ltr0n.
rewrite mulr1 big_distrr /=.  the goal is $\sum_{i<n} \frac{|T|^{\ell_{\max}}}{|T|^{\ell_i}} \leq |T|^{\ell_{\max}}$
rewrite (eq_bigr (fun i : 'I_n => #|suffixes C``_i|%:R)%R); last first.
  move=> i _; rewrite card_suffixes; last by apply/nthP; exists i.
  rewrite natrX exprB // ?(nth_map [::]) //.
  by apply/leq_lmax/nthP; exists i.
  by rewrite unitfE pnatr_eq0 -lt0n.
the goal is now $\sum_{i<n} |\{x|\text{prefix } C_i \ x\}| \leq |T|^{\ell_{\max}}$, Eqn (III.1)
apply (@ler_trans _ (#|\bigcup_(i < n) suffixes (C ``_ i)|%:R)%R).
  rewrite -sum1_card.
  rewrite partition_disjoint_bigcup /=.
    rewrite natr_sum ler_sum // => i _.
    by rewrite sum1_card.
  move=> i j ij.
  rewrite -setI_eq0 disjoint_suffixes //.
  by apply/nthP; exists i.
  by apply/nthP; exists j.
  by rewrite nth_uniq //; case: C.
the goal is $\left|\bigcup_{i<n}\{x|\text{prefix } C_i \ x\}\right| \leq |T|^{\ell_{\max}}$, step (III.2)-(III.3)
by rewrite -natrX -card_tuple ler_nat max_card.
Qed.
```

3) Kraft's Inequality (converse part): We now complete the informal proof of the converse of Kraft's inequality (started in Sec. III-B) and provide a complete proof script. We were left with two subgoals. $r = \frac{w_k}{|T|^{\ell_k-\ell_j}} \geq w_j+1$ is proved as follows:

$$r = \sum_{i<k} |T|^{\ell_j} |T|^{-\ell_i} \tag{III.4}$$

$$= w_j + \sum_{j \leq i < k} |T|^{\ell_j} |T|^{-\ell_i} \tag{III.5}$$

$$\geq w_j + 1. \tag{III.6}$$

Here follows the proof for $r - 1 < w_j$:

$$r - 1 = \frac{w_k}{|T|^{\ell_k-\ell_j}} - 1 \qquad \text{(by definition)} \tag{III.7}$$

$$= \frac{w_j |T|^{\ell_k-\ell_j} + w_k \bmod |T|^{\ell_k-\ell_j}}{|T|^{\ell_k-\ell_j}} - 1 \tag{III.8}$$

$$< w_j. \tag{III.9}$$

The fact that σ_j is a prefix of σ_k is used in the step (III.7)–(III.8). These two subgoals respectively correspond to the subgoals `have H1` and `have H2` below:

```
Lemma kraft_implies_prefix : kraft_cond R T l ->
  exists C : code_set T, prefix_code C.
Proof.
move=> H; exists (ACode _ l_n sorted_l).
apply nnpp_prefix.
move=> /(if_not_prefix l_neq0 H) /existsP[j /existsP[ k /andP[jk pre]]].
at this point, the goal is $\forall j, k. i < k \to \neg\text{prefix } \sigma_j \ \sigma_k$
pose r := ((w k)%:R / #|T|%:R^+(l``_k - l``_j) : R)%R.
let $r = w_k/|T|^{\ell_k-\ell_j}$
have H1 : (r >= (w j)%:R + (1 : R))%R.  here we prove $r \geq w_j + 1$
  pose r' := (\sum_(i < k) #|T|%:R ^+ l``_j * #|T|%:R ^- l``_i : R)%R.
  let $r' = \sum_{i<k} |T|^{\ell_j} |T|^{-\ell_i}$
  have -> : r = r'.  here we prove $r = r'$, see Eqn (III.4)
    rewrite /r /r' natr_sum big_distrl /=; apply/eq_bigr => i _.
    have ? : (#|T|%:R ^+ (l ``_ k - l ``_ j) : R)%R \is a GRing.unit.
      by rewrite unitfE expf_eq0 card_ord pnatr_eq0 andbF.
    apply: (@mulIr _ (#|T|%:R ^+ (l``_k - l``_j))%R) => //.
    rewrite natrX -mulrA mulVr // mulr1 exprB; last 2 first.
      by rewrite nth_of_sorted // ltnW //= l_n.
      by rewrite unitfE pnatr_eq0 card_ord.
    rewrite exprB; last 2 first.
      by rewrite nth_of_sorted // ltnW //= l_n.
      by rewrite unitfE pnatr_eq0 card_ord.
    rewrite mulrCA mulrAC mulrV // ?mul1r //.
    by rewrite unitfE -natrX pnatr_eq0 expn_eq0 card_ord.
  pose u := (\sum_(j<=i<k) #|T|%:R ^+ l``_j * #|T|%:R ^- l``_i : R)%R.
  let $u = \sum_{j \leq i<k} |T|^{\ell_j} |T|^{-\ell_i}$
  have -> : (r' = (w j)%:R + u :> R)%R.  $r' = w_j + u$, Eqn (III.5)
    pose f := (fun i : nat => #|T|%:R^+l``_j * #|T|%:R^-l``_i : R)%R.
    case/boolP : (j == ord0) => j0.
      rewrite /u (eqP j0) wE0 add0r big_mkord /r'.
      apply/eq_bigr => i _; by rewrite (eqP j0).
    rewrite /r' /u -(big_mkord xpredT f)%R natr_sum.
    rewrite (eq_bigr (fun i : 'I__ => f i)); last first.
      move=> i _; rewrite natrX exprB //.
      by rewrite nth_of_sorted // ltnW //= l_n.
      by rewrite unitfE pnatr_eq0 card_ord.
    by rewrite -(big_mkord xpredT f)%R -big_cat_nat //= ltnW.
  rewrite ler_add //.
   at this point, the subgoal is $1 \leq u$, for the step (III.5)-(III.6)
  rewrite /u -(@prednK k); last by rewrite (leq_ltn_trans _ jk).
  rewrite big_nat_recl; last by move/(leq_sub2r 1) : jk; rewrite !subn1.
  rewrite divrr ?unitfE -?natrX ?pnatr_eq0 ?expn_eq0 ?card_ord //.
  rewrite ler_addl sumr_ge0 // => i _.
  by rewrite natrX divr_ge0 // exprn_ge0 // ?card_ord ?ler0n.
have H2 : (r - 1 < (w j)%:R)%R.  here we prove $r - 1 < w_j$
  have /(congr1 (fun x => x%:R : R)%R) : w k =
    w j * #|T| ^ (l``_k - l``_j) + w k %% #|T| ^ (l``_k - l``_j).
```

we prove $w_k = w_j |T|^{\ell_k - \ell_j} + w_k \bmod |T|^{\ell_k - \ell_j}$, leading to (III.8)

```
    have := prefix_modn pre.
    do 2 rewrite nat_of_ary_cat nat_of_ary_nseq0 mul0n add0n ary_of_natK.
    by rewrite !size_cat !size_nseq !subnK // size_ary_of_nat // -/t
     -(card_ord t) (w_sub l_n sorted_l H).
  rewrite natrD => /(congr1 (fun x => x / #|T|%:R^+(l``_k - l``_j)))%R.
  rewrite -/r mulrDl natrM natrX mulrK; last first.
    by rewrite unitfE expf_eq0 card_ord pnatr_eq0 andbF.
  move=> wkE.
  have : ((w k %% #|T| ^ (l``_k - l``_j))%:R /
          #|T|%:R ^+ (l``_k - l``_j) < (1 : R))%R.
```

here we prove $(w_k \bmod |T|^{\ell_k - \ell_j})/|T|^{\ell_k - \ell_j} < 1$, leading to (III.9)

```
    rewrite ltr_pdivr_mulr; [|by rewrite -natrX ltr0n expn_gt0 card_ord].
    by rewrite mul1r -natrX ltr_nat ltn_mod expn_gt0 card_ord.
  by rewrite {}wkE ltr_sub_addl addrC ltr_add2r.
by rewrite ltr_subl_addl addrC ltrNge H1 in H2.
Qed.
```

D. Application: Shannon-Fano Codes

Let us assume a source that emits symbols from an alphabet A with probability Pr. A Shannon-Fano code is such that each symbol a is mapped to a codeword of length $\left\lceil \log \frac{1}{\Pr[a]} \right\rceil$. This definition requires to make explicit the encoding function from A to the codewords as a type `Encoding.t`:

```
(* Module Encoding *)
Record t (A T : finType) := mk {
  f :> {ffun A -> seq T}; f_inj : injective f }.
```

Let `P` be a distribution over `A`. Formally, a Shannon-Fano code is an encoding function that satisfies the following predicate:

```
Definition is_shannon_fano (f : Encoding.t A T) :=
  forall a, size (f a) =
    Zabs_nat (ceil (Log #|T|%:R (1 / P a))).
```

(`Zabs_nat` takes the absolute value of an integer.)

We can show that the list of codewords generated by a Shannon-Fano code satisfies the Kraft condition[2]:

```
Variable (f : Encoding.t A T).
Let sizes := [seq (size \o f) a| a in A].
Lemma shannon_fano_is_kraft :
  is_shannon_fano P f -> kraft_condR T sizes.
```

In consequence, the construction of Sec. III-A provides a way to construct a (prefix) Shannon-Fano code.

For the sake of completeness, we establish formally the fact that Shannon-Fano codes are sub-optimal. It suffices to show that the average code length is less than $\mathcal{H}(\Pr) + 1$:

```
Lemma shannon_fano_suboptimal :
  is_shannon_fano P f -> average P f < `H P  + 1.
```

Here, the average code length of `f` is defined as follows:

```
Definition average :=
  \rsum_(x in A) P x * (size (f x))%:R.
```

IV. Related Work

There already exist several examples of formal proofs about data compression. Shannon's source coding theorems have been formalized in the Coq proof-assistant [4], [5], the algorithmic aspects (but not the information-theoretic aspects) of Huffman codes have been formalized in Isabelle/HOL [3]. These examples are larger than the ones discussed in this paper but our work still improves on previous work: our proof of Kraft's inequality covers alphabets of any size (the proof of Kraft's inequality in [7] is for the binary case), our proof of Jensen's inequality covers partial functions (the proof in [12] is for total functions). One could argue that these are minor improvements but they are moreover integrated within the same library for formal verification of information theory and error-correcting codes [13], allowing for further applications.

V. Conclusion and Future Work

We provided new formalizations about data compression (Jensen's and Kraft's inequalities) as well as two concrete applications (analysis of string compression and Shannon-Fano codes). We tailored our technical explanations for a reader who is not proficient neither in formal logic nor with proof-assistants, by explaining in particular how one gets from the paper proof to a formal proof. Our hope is to improve the usability of our library for formalization of information theory and error-correcting codes.

Proof scripts are available online [13]. The files relevant to this paper are: `jensen.v` (Sec. II-B), `string_entropy.v` (Sec. II-C), `kraft.v` (Sec. III-C), and `shannon_fano.v` (Sec. III-D). There were also several technical improvements to the InfoTheo library (mostly about the interface with Coq's standard library for real analysis).

We plan a number of technical improvements: support for a version of the MathComp library with well-integrated real numbers, better formal definitions of codes and encoding functions. We will then generalize results (for example from Kraft's inequality to Kraft-McMillan's) to tackle other codes. The resulting formal theory of data compression should be large enough to help us verify software implementations of compact data structures [9, Chap. 2].

References

[1] R. J. McEliece. *The theory of information and coding*, Encyclopedia of Mathematics and its Applications 86. Cambridge University Press, 2002

[2] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Second edition, John Wiley & Sons, 2006

[3] J. C. Blanchette. Proof Pearl: Mechanizing the Textbook Proof of Huffman's Algorithm. J. Autom. Reasoning 43(1): 1-18 (2009)

[4] R. Affeldt, M. Hagiwara, and J. Sénizergues. Formalization of Shannon's theorems. J. Autom. Reasoning 53(1):63–103 (2014)

[5] R. Obi, M. Hagiwara, and R. Affeldt. Formalization of variable-length source coding theorem: Direct part. In International Symposium on Information Theory and Its Applications (ISITA 2014), pages 201–205

[6] R. Affeldt and J. Garrigue. Formalization of error-correcting codes: from Hamming to modern coding theory. In 6th Conference on Interactive Theorem Proving (ITP 2015), LNCS 9236:17–33, Springer

[7] C.-S. Senjak and M. Hofmann. An implementation of Deflate in Coq. In International Symposium on Formal Methods (FM 2016), LNCS 9995:612–627, Springer

[8] R. Affeldt, J. Garrigue, and T. Saikawa. Formalization of Reed-Solomon codes and progress report on formalization of LDPC codes. In International Symposium on Information Theory and Its Applications (ISITA 2016), pages 537–541

[9] G. Navarro. *Compact Data Structures*. Cambridge University Press, 2016

[10] The Coq proof-assistant. https://coq.inria.fr/

[11] Mathematical Components: Libraries of formalized mathematics. http://math-comp.github.io/math-comp/

[12] P.-Y. Strub. https://github.com/math-comp/analysis/blob/master/altreals/distr.v. Part of https://github.com/math-comp/analysis/

[13] The infotheo library. https://github.com/affeldt-aist/infotheo

[2] `kraft_condR` is an ad-hoc instantiation of `kraft_cond` from Sec. III-C1 with the type of real numbers. It is ad-hoc because, even though the definition from Sec. III-C1 is generic, Coq reals have not yet been integrated into the MathComp library.