Formalization of Shannon's Theorems Using the Coq Proof-Assistant

Reynald Affeldt

Research Institute for Secure Systems, National Institute of Advanced Industrial Science and Technology

概 要

Proofs in information theory can be very technical, to the point that the exposition of details must sometimes be sacrificed for the sake of clarity. The only way to provide complete proofs without impairing understanding is to formalize them with the help of dedicated computer software: proof-assistants. In this tutorial, we demonstrate formal proof for information theory. First, we explain the basics of formal proof within the Coq proof-assistant. Second, we introduce a library of formal definitions and lemmas to formalize information theory using Coq. Last, we explain how we formalized Shannon's theorems. Since the latter constitute the foundations of information theory, we believe that our work paves the way to further research in coding theory.

1 What are Formal Proofs Useful For?

A proof is a deductive argument that establishes a mathematical statement. It is usually articulated using natural language and conventional notations. A proof is only as detailed as it is necessary to be convincing. Yet, when proofs become large, they become uncheckable by human beings. This is for example the case of the proof of the Kepler conjecture by Hales [7].

Computers are now used to help mathematicians to construct proofs. It began in 1976 when Appel and Haken proved the four color theorem [4]. At first, some mathematicians disregarded this proof because it relied on an enumeration by a computer program that cannot be checked by hand.

Proof assistants are computer software for the construction of proofs. They not only help carrying out tedious enumerations, but they also provide a double-checking mechanism to ensure that no reasoning error has been introduced in the process. When Gonthier formalized the proof of Appel and Haken with the Coq proof-assistant [3] in 2005, it became clear that the four color theorem was proved at last [5].

In the case of the Coq proof-assistant, the double-checking mechanism is a proof-checking algorithm. It all comes from type theory, an alternative to set theory for the foundations of mathematics [8]. It was observed in 1968 that there is a correspondence between formal proofs and functional programs (this is the so-called Curry-Howard isomorphism). Since one knows how to check whether a program is well-typed, one also knows how to check that a proof is correctly constructed. This provides a small trusted base to check proofs mechanically and this is the basis for the construction of proof-assistants.

Application to Information Theory We claim that information theory and error-correcting codes can benefit from formal proofs. The main reason to believe so is that proofs in information theory are often very technical, so that the exposition of details is often sacrificed for the sake of clarity. For example, it is not rare to find in asymptotic proofs claims such as "this holds for n sufficiently large" without any justification for the existence of such a bound. Similarly, recent advances in coding theory (e.g., low-density parity-check codes) are supported by proofs lacking so much detail that their formalization is a subject of concern.

We have recently started working on formalization of information and coding theory using the Coq proof-assistant. In particular, we have recently completed the formalization of Shannon's theorems (a.k.a. the source and channel coding theorems) that define the basic notions of information theory, namely the entropy and the channel capacity. We are now working on formalization of error-correcting codes.

2 The Basics of the Coq Proof-Assistant

We now explain the basics behind the proof checking algorithm of the Coq proof-assistant. Let us consider the following mathematical statement, where A, B, and C are propositions:

$$(A \to B \to C) \to (A \to B) \to A \to C$$

It is certainly true, but how do we provide a checkable proof for it? A formal proof is written using a small set of reasoning rules. Here follow the two generic reasoning rules needed to prove statements about the implication:

$$\dfrac{\begin{array}{c}[F]_i \\ \vdots \\ G\end{array}}{F \to G} \to \mathsf{I}_i \qquad\qquad \dfrac{F \quad F \to G}{G} \to \mathsf{E}$$

Intuitively, the rule on the left says that to prove $F \to G$, it is sufficient to derive G from the assumption F. The rule on the right is the well-know *modus ponens*: if one knows F and $F \to G$, when one knows G. These two rules are all what is needed to prove the above statement. One solely needs to arrange them in a well-formed tree as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[A\to B\to C]_1 \quad [A]_3}{B\to C}\,{\to}\mathsf{E} \qquad \dfrac{[A\to B]_2 \quad [A]_3}{B}\,{\to}\mathsf{E}}{C}\,{\to}\mathsf{E}}{A\to C}\,{\to}\mathsf{I}_3}{(A\to B)\to A\to C}\,{\to}\mathsf{I}_2}{(A\to B\to C)\to(A\to B)\to A\to C}\,{\to}\mathsf{I}_1$$

That is a formal proof, and one can easily imagine a system that checks whether or not the above tree is well-formed. The way the Coq proof-assistant performs the above checking is a bit more convoluted: proofs are actually represented as terms of the so-called λ-calculus, an archetypal programming language, to which types are attached that can be read exactly as mathematical statements. This is this reading of types as statements and programs as proofs that is known as the Curry-Howard isomorphism.

A Proof in Coq In practice, it is not manageable to manipulate graphical objects such as the above proof tree. Instead, the user of the Coq proof-assistant writes down *proof scripts* using characters (often in ASCII to ease input) to construct proofs interactively. The user first inputs the statement he wants to prove using standard notations from type theory: one writes `P : T` where `P` is a proof and `T` is a type; propositions `Prop` are basic types; types can be arranged with the implication `->`:

```
$ Variables A B C : Prop.
$ Lemma goal : (A->B->C)->(A->B)->A->C.
```

Coq answers by displaying the bottom of the proof tree to be constructed:

```
>  ===========
>  (A->B->C)->(A->B)->A->C
```

The user goes on by giving names to the hypotheses (`H1`, `H2`, `H2`):

```
$ intro H1. intro H2. intro H3.
```

Coq responds by updating the proof object accordingly, adding named hypotheses above the conclusion:

```
>  H1 : A->B->C
>  H2 : A -> B
>  H3 : A
>  ===========
>  C
```

The rule for modus ponens is also called `cut`. It results in splitting the proof tree into two sub-goals:

```
$ cut B.
```

```
>  H1 : A->B->C
>  H2 : A -> B
>  H3 : A
>  ===========
>  B -> C
>
>  subgoal 2 is:
>   B
```

The first subgoal can be solved by applying the hypothesis `H1` to the hypothesis `H2`:

```
$ apply (H1 H3).
```

Similarly `apply (H2 H3)` solves the second subgoal and this concludes the proof.

Coq can therefore been seen as the language to write proof scripts (`intro`, `cut`, `apply`, etc.) and to manage the resulting lemmas (the lemma `goal` here) into reusable libraries.

3 Shannon's Theorems

In this tutorial, we will focus on formal definitions to state Shannon's theorems; much work actually goes into the construction of proof scripts but this is too technical to fit this abstract.

3.1 Finite Probabilities

First of all, we will need a formalization of probability distributions to be able to formalize probability of error or success of decoding. A probability distribution over a set `A` can be formalized as a function from `A` to `R+` (the type of positive reals) such that the sum over `A` of all the probabilities is equal to 1 (i.e., $\sum_{a\in A} P(a) = 1$). Already much needs to be formalized in order to write down such a definition but luckily Coq comes with several libraries that provide the most basic mathematical objects: the type `R` of reals comes from the standard library of Coq, and one can find summation operators in the SSReflect [9] extension of Coq. Let us note `pmf` the function that defines a probability distribution (function from `A` to `R+`) and `pmf1` the proof that $\sum_{a\in A} P(a) = 1$. Since `pmf` and `pmf1` need to be put together for the formalization of a probability distribution to make sense, we arrange them as a `Record`:

```
Record dist := mkDist {
  pmf :> A -> R+ ;
  pmf1 : \rsum_(a in A) pmf a = 1 }.
```

Since we are implicitly considering sample spaces to be the powerset of some set `A`, an event can be formalized as a boolean predicate over `A` (`pred A`, which is nothing more than a function from `A` to booleans). The probability that the event `E` holds is defined as $\sum_{\substack{a\in A\\ E\,a}} P(a)$:

```
Definition Pr P (E : pred A) :=
  \rsum_(a in A | E a) P a.
```

Now, let us assume that we are given some distribution `P` over `A` (i.e., some object `P` with the type `dist A`). Then one can easily define the entropy of `P` (i.e., $-\sum_{a\in A} P(a)\log P(a)$):

```
Definition entropy P :=
  - \rsum_(a in A) P a * log (P a).
```

In the following, `H P denotes the entropy of P.

3.2 The Source Coding Theorem

The source coding theorem is a theorem about information compression over a noiseless channel. A (source) code is a pair of (1) an encoding function that turns tuples of k elements of some input alphabet A into n bits (i.e., booleans):

```
Definition encT := k.-tuple A -> n.-tuple bool.
```

and (2) a decoding function that recovers the tuples of A elements from the tuples of bits:

```
Definition decT := n.-tuple bool -> k.-tuple A.
```

Formally, a source code is a pair of such encoding and decoding functions:

```
Record scode := mkScode { enc : encT; dec : decT }.
```

The goal of a source code is to minimize the number of bits needed for encoding, in other words to lower the *rate* of source codes:

```
Definition SrcRate (sc : scode) := n / k.
```

The probability of error for decoding with a source code sc is defined as the probability that decoding (i.e., function dec sc) of encoding (enc sc) fails:

```
Definition SrcErrRate P sc := Pr (P `^ k)
  [pred t | dec sc (enc sc t) != t].
```

The probability here is taken over the distribution of tuples of k elements emitted with probability distribution P. We note P `^ k this *tuple distribution*, defined as $t \mapsto \prod_{1 \leq i \leq k} P(t_i)$ (formalization omitted here).

The direct part of the source coding theorem says that for a source emitting symbols with probability distribution P there exists a source code of rate greater than `H P such that the probability of decoding error is negligible. Above formal definitions are enough to formally paraphrase this statement as follows:

```
Theorem source_coding_direct :
  forall lambda, 0 < lambda < 1 ->
  forall r, `H P < r ->
    exists k n (sc : scode A k n),
      r = SrcRate sc /\
      SrcErrRate P sc <= lambda.
```

Conversely, codes with rate smaller than the entropy of the source have non-negligible probability of decoding error:

```
Theorem source_coding_converse :
  forall lambda, 0 < lambda < 1 ->
  forall r : Qplus, 0 < r < `H P ->
    forall n k (sc : scode A k.+1 n),
      r = SrcRate sc ->
      SrcConverseBound P (num r) (den r)
        n lambda <= k.+1 ->
      SrcErrRate P sc >= lambda.
```

Here, SrcConverseBound (formal definition omitted here) is a function that defines a bound above which k is big enough for the source code to have non-negligible error rate, a bound that is never defined in pencil-and-paper proofs but that needs to be made explicit to complete the formal proof.

3.3 The Channel Coding Theorem

The channel coding theorem is a theorem for reliable information transmission over a noisy channel. The basic idea is to encode messages from some set M by a longer message. Formally, such an encoding function is a function that turns messages into tuples of, say n, elements of A where A is the input alphabet of the channel:

```
Definition encT := {ffun M -> n.-tuple A}.
```

A decoding function conversely turns the output of the channel (tuples of n elements of B, the output alphabet) back into messages. To represent the fact that decoding can fail, the codomain of the decoding function is formalized by an option type:

```
Definition decT := {ffun n.-tuple B -> option M}.
```

A (channel) code is then simply a pair of an encoding function and a decoding function:

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

Of course, we are looking for n to be as small as possible. In other words, we are interested in maximizing the *rate* of the code defined as follows:

```
Definition CodeRate (c : code) := log #| M | / n.
```

Channels are represented by stochastic matrices, that are best formalized by functions returning distributions (the row of the stochastic matrix). Here follows the type of a channel with input alphabet A and output alphabet B:

```
Definition channel1 := A -> dist B
```

Hereafter, we denote channel A B by `C1 A B. The main characteristic of a channel is its *capacity*. The latter is the least upper bound of the mutual information (between the input distribution and the output distribution) taken over all possible input distributions. We formalize the capacity by a relation capacity between a channel and a real number. Beforehand, let us formalize the mutual information. The output distribution is the distribution of the outputs:

```
Definition out_dist (P : dist A)
  (W : `Ch1 A B) : dist B.
apply makeDist with (fun b =>
  \rsum_(a in A) W a b * P a).
...
```

(This definition needs to be completed by the proof that this function is really a distribution, we omit it for lack of space.) We also need the joint distribution of the inputs and the outputs:

```
Definition joint_dist (P : dist A)
  (W : `Ch1 A B) : dist [finType of A * B].
apply makeDist with (fun ab =>
  W ab.1 ab.2 * P ab.1).
...
```

Let us note `` `H(P o W) `` the entropy of the output distribution and `` `H(P , W) `` the entropy of the joint distribution. The mutual information is then formalized as follows:

```
Definition mut_info P (W : `Ch1 A B) :=
  `H P + `H(P `o W) - `H(P , W).
```

We are now in a position to formalize the relation between a channel `W` and its capacity `cap`:

```
Definition capacity (W : `Ch1 A B) cap :=
  lubound (fun P => `I(P ; W)) cap.
```

Here, `lubound` is a relation such that `lubound f lub` holds when `lub` is the least upper bound of `f`.

The direct part of the channel coding theorem says that for some rate strictly smaller that the channel capacity, there exists a code with this rate such that the probability of error is negligible. Put formally:

```
Theorem channel_coding : r < cap ->
  forall epsilon, 0 < epsilon ->
    exists n M (c : code A B M n),
      sval r = CodeRate c /\
      CodeErrRate W c < epsilon.
```

`CodeErrRate` is the probability that decoding of encoded messages fails. It is formalized as follows. First we define the probability of decoding error knowing that some message `m` was sent:

```
Definition ErrRateCond (W : `Ch1 A B) c m :=
  Pr (W ``^ n (| enc c m) )
    [pred tb | dec c tb != Some m].
```

``W ``^ n (| enc c m)`` is the distribution of outputs knowing that that `enc c m` was sent (formalization omitted here). The probability of decoding error is defined as the average probability for all messages in `M`:

```
Definition CodeErrRate (W : `Ch1 A B) c :=
  1 / INR #| M | *
    \rsum_(m in M) ErrRateCond W c m.
```

The most difficult Shannon's theorem is the converse part of the channel coding theorem, which shows that codes with rate beyond channel capacity have negligible success rate. Let `W` be a channel with input alphabet `A` and output alphabet `B`, and capacity `cap` (i.e., `capacity W cap`). Let `epsilon` be some strictly positive real number and let `minRate` be some real strictly bigger than the capacity (`minRate > cap`). The converse of the channel coding theorem can be formalized as follows:

```
Theorem channel_coding_converse : exists n0,
  forall n M (c : code A B M n),
  M != set0 -> n0 < n -> minRate <= CodeRate c ->
  CodeSuccRate W c < epsilon.
```

The success rate is the complement probability of the error rate seen above:

```
Definition CodeSuccRate W c := 1 - CodeErrRate W c.
```

Conclusion We gave an overview of the formalization of Shannon's theorems using the Coq proof-assistant. We provided most formal definitions necessary to understand the formal statements of the theorems (documentation for the complete formal development can be found online [2], see also [1] for more details and related work). Compared to pencil-and-paper proofs, our formalization has the benefit of providing explicit proofs of existence for all the bounds that appear in the proofs and is guaranteed to be free of typos, implicit assumptions, and, of course, errors. This effort helped in particular in debugging the pencil-and-paper proofs in the textbook by Hagiwara [6].

Acknowledgments This tutorial is based on joint work with Manabu Hagiwara and Jonas Sénizergues.

参考文献

[1] R. Affeldt, M. Hagiwara. Formalization of Shannon's Theorems in SSReflect-Coq. In 3rd Conference on Interactive Theorem Proving, LNCS 7406:233–249. Springer, 2012

[2] Formalization of Shannon's Theorems. Coq documentation. `http://staff.aist.go.jp/reynald.affeldt/shannon`

[3] The Coq Proof Assistant. `http://coq.inria.fr` INRIA, 1984–2013

[4] K. Appel, W. Haken. Every map is four colourable. Bulletin of the American Mathematical Society 82:711–712 (1976)

[5] G. Gonthier. Formal Proof—The Four-Color Theorem. Notices of the American Mathematical Society 55(11): 1382–1393 (2008)

[6] 萩原 学. 符号理論：デジタルコミュニケーションにおける数学. 日本評論社，2012

[7] T. C. Hales. A proof of the Kepler conjecture. Annals of Mathematics 162(3):1065–1185 (2005)

[8] J. van Heijenoort. From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931. Harvard University Press (1967)

[9] G. Gonthier, A. Mahboubi, E. Tassi. A Small Scale Reflection Extension for the Coq System. Technical Report 6455. Version 11. INRIA, 2012