UMinho | 2012

Diogo Araújo Carvalho Vilaça Moreira **Finite Probability Distributions in Coq**

Universidade do Minho
Escola de Engenharia

Diogo Araújo Carvalho Vilaça Moreira

Finite Probability Distributions in Coq

Janeiro de 2012

Universidade do Minho
Escola de Engenharia

Diogo Araújo Carvalho Vilaça Moreira

Finite Probability Distributions in Coq

Tese de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Carlos Bacelar Almeida

Janeiro de 2012

Knowing is not enough; we must apply. Willing is not enough; we must do.

Johann Wolfgang von Goethe

Acknowledgements

To my supervisor, Professor José Carlos Bacelar Almeida, for the support, expertise and careful guidance provided throughout this work, a special thanks for everything.

To all the people involved in the SMART project at Minho's University, in particular: Manuel Barbosa, José Barros, Manuel Alcino, Paulo Silva, Luís Miranda and Tiago Oliveira, for all the fruitful discussions and good working environment provided, a sincere thank you. During the execution of this work, I was given a research grant - Bolsa de Investigação Científica (BIC), funded by the ENIAC JU program (GA 120224).

To my home town friends, for understanding the countless times I did not show up for our weekly meetings, for the continuous support and friendship, thank you for never letting me down.

To my friends in Braga, but specially to my friends at Confraria do Matador, a huge thanks for this 5 years of great experiences.

To Sara and Sérgio, I am extremely grateful for all the help provided in the elaboration of this document.

To all my family, but specially to my parents, José and Maria, and my brother, Ricardo, words cannot describe how much you mean to me. A huge thank you from the bottom of my heart, for being my guiding light and source of inspiration, and for providing the conditions that allowed me to be where I am at this stage of my life.

Abstract

When building safety-critical systems, guaranteeing properties like correctness and security are one of the most important goals to achieve. Thus, from a scientific point of view, one of the hardest problems in cryptography is to build systems whose security properties can be formally demonstrated.

In the last few years we have assisted an exponential growth in the use of tools to formalize security proofs of primitives and cryptographic protocols, clearly showing the strong connection between cryptography and formal methods. This necessity comes from the great complexity and sometimes careless presentation of many security proofs, which often contain holes or rely on hidden assumptions that may reveal unknown weaknesses. In this context, interactive theorem provers appear as the perfect tool to aid in the formal certification of programs due to their capability of producing proofs without glitches and providing additional evidence that the proof process is correct.

Hence, it is the purpose of this thesis to document the development of a framework for reasoning over information theoretic concepts, which are particularly useful to derive results on the security properties of cryptographic systems. For this it is first necessary to understand, and formalize, the underlying probability theoretic notions. The framework is implemented on top of the *fintype* and *finfun* modules of SSREFLECT, which is a small scale reflection extension for the COQ proof assistant, in order to take advantage of the formalization of big operators and finite sets that are available.

Resumo

Na construção de sistemas críticos, a garantia de propriedades como a correção e segurança assume-se como um dos principais objetivos. Deste modo, e de um ponto de vista científico, um dos problemas criptográficos mais complicados é o de construir sistemas cujas propriedades possam ser demonstradas formalmente.

Nos últimos anos temos assistido a um crescimento enorme no uso de ferramentas para formalizar provas de segurança de primitivas e protocolos criptográficos, o que revela a forte ligação entre a criptografia e os métodos formais. Urge esta necessidade devido à grande complexidade, e apresentação por vezes descuidada, de algumas provas de segurança que muitas vezes contêm erros ou se baseiam em pressupostos escondidos que podem revelar falhas desconhecidas. Desta forma, os *provers* interativos revelam-se como a ferramenta ideal para certificar programas formalmente devido à sua capacidade de produzir provas sem erros e de conferir uma maior confiança na correção dos processos de prova.

Neste contexto, o propósito deste documento é o de documentar e apresentar o desenvolvimento de uma plataforma para raciocinar sobre conceitos da teoria de informação, que são particularmente úteis para derivar resultados sobre as propriedades de sistemas criptográficos. Para tal é necessário, em primeiro lugar, entender e formalizar os conceitos de teoria de probabilidades subjacentes. A plataforma é implementada sobre as bibliotecas *fintype* e *finfun* do SSREFLECT, que é uma extensão à ferramenta de provas assistas COQ, por forma a aproveitar a formalização dos somat' orios e conjuntos finitos disponíveis.

Contents

List of Figures

Chapter 1

Introduction

Number theory and algebra have been playing an increasingly significant role in computing and communications, as evidenced by the striking applications of these subjects in fields such as cryptography and coding theory [Sho05]. However, the commonly held opinion is that the formalization of mathematics is a long and difficult task for two reasons: first, standard mathematical proofs (usually) do not offer a high enough level of detail and second, formalized theory is often insufficient as a knowledge repository which implies that many lemmas have to be reproved in order to achieve relevant results.

In the last few years we have assisted an exponential growth in the use of tools to formalize security proofs of primitives and cryptographic protocols, clearly showing the strong connection between cryptography and formal methods. This necessity comes from the great complexity and sometimes careless presentation of many important scheme's proofs, which may lead to mistakes undetected until now. Moreover, and although provable cryptography provides high assurable cryptographic systems, security proofs often contain glitches or rely on hidden assumptions that may reveal unknown weaknesses. In this context, interactive theorem provers appear as the perfect tool to aid in the formal certification of programs due to their capability to produce proofs without the holes usually found in their paper versions and to provide additional evidence that the proof process is correct. This idea is reflected by the increasing use, despite some may think, of interactive theorem provers for industrial verification projects: for instance, NASA uses PVS[1] to verify software for airline control and Intel uses HOL light[2] to verify the design of new chips [Geu09].

Cryptographic protocols provide mechanisms to ensure security that are used in several application domains, including distributed systems and web services (and more recently also in the protection of intellectual property). However, designing secure cryptographic protocols is extremely difficult to achieve [BT05]. In this context there is an increasing trend to study such systems, where the specification of security requirements is provided in order to establish that the proposed system meets its requirements.

Thus, from a scientific point of view, one of the most challenging problems in cryptography is to build systems whose security properties can be formally demonstrated. Such properties only make sense in a context where a definition of security is given: furthermore, knowledge about the adversary's available information and computation power should also be provided. Both adversary and benign entity are probabilistic processes that communicate with each other, and so, it is possible to model this environment as a probability space. Equally important, we may use information (and probability) theoretic concepts (*e.g.*, entropy) to derive results on the security properties of cryptographic systems, which clearly reflects the fundamental role of these two areas in cryptography [Mau93].

1.1 Objectives

The central objective of this work is to build a framework that provides the user the possibility to reason over information theory concepts, particularly in the field of cryptography. Thus, it is first necessary to understand, and formalize, the underlying probability theory notions and also how to use them in order to build such framework. In this context we aim to develop, in COQ, a library that formalizes fundamental laws of probability theory and basic concepts of information theory.

[1]`http://pvs.csl.sri.com/`

[2]`http://www.cl.cam.ac.uk/~jrh13/hol-light/`

1.2 Contributions

This thesis proposes a formal framework to reason about probability and information theory concepts, possibly in a cryptographic environment, which is implemented using the COQ proof assistant, and particularly, the SSREFLECT plugin. The key to our implementation is to take advantage of the formalization of big operators and finite sets provided by the SSREFLECT libraries *finset* and *bigop*, since it allows to precisely capture the mathematical aspects of the concepts involved.

In detail, we make the following contributions:

- **Development of a probability oriented framework**
  A COQ framework is implemented that allows the user to reason about basic notions of probability and set theory.

- **Extension of the framework to include the formalization of relevant information theoretic concepts**
  On top of the probability oriented frame work, a formalization of information theory concepts is added that should serve as a basis to reason in cryptographic contexts.

1.3 Dissertation Outline

The remainder of the document is organized as follows: Chapter 2 briefly introduces the COQ proof assistant, its small scale reflection extension, SSREFLECT and the two most relevant libraries for this work; Chapter 3 explains basic concepts of probability theory; Chapter 4 discusses our approach to the formalization of discrete probabilities in COQ; Chapter 5 introduces important notions of information theory with a focus on the concept of entropy; Chapter 6 presents and discusses our formalization of such information theoretic concepts. The Chapter closes with a simple case study.

Chapter 2

Interactive Theorem Proving

"For our intention is to provide a medium for doing mathematics different from that provided by paper and blackboard. Eventually such a medium may support a variety of input devices and may provide communication with other users and systems; the essential point, however, is that this new medium is active, whereas paper, for example, is not."

Constable et al. [CAA$^+$86]

An interactive theorem prover, or proof assistant, is a software tool that aids in the development of formal proofs by interacting with the user (user-machine collaboration). Its purpose is to show that some statement, the conjecture, is a logical consequence of a set of statements, the axioms and hypotheses, given an appropriate formalization of the problem.

These statements are written using a logic (*e.g.*, first-order logic, high-order logic), which enables the user to formulate the problem in such a way that the machine may, unambiguously, understand it. The idea is to allow him to set up a mathematical context, define properties and logically reason about them. The user then guides the search for proofs in a proof editing mode where he should be relieved of trivial steps in order to concentrate on the decision points of the proofs. Therefore the system must provide a large, and trusted, set of mathematical theory which usually represents a measure of the success of the system and serves as a means of achieving complex results. In practice, for serious formalization of mathematics a good library is usually more important than a user friendly system.

The input language of a proof assistant can be declarative, where the user tells the system where to go, or procedural, where the user tells the system what to do [Geu09]. Usually the proofs of the latter are not readable by the common reader since they only have meaning for the proof assistant whereas the proofs of the former tend to be written in a more clear and natural way. This is exemplified by the fact that proof scripts[1] need not correspond to a proof in logic, as generally happens with procedural proofs but not with declarative proofs.

In mathematics, and in particular when dealing with proof assistants, a proof is absolute. It is the mathematical validation of a statement, whose correction can be determined by anyone, and is supposed to both convince the reader that the statement is correct and to explain why it is valid. Each proof can be divided in small steps that may be individually verified. This is important when handling the proof of a theorem that turns out to be false as it allows to pinpoint the exact step(s) that cannot be verified.

Nowadays proof assistants are mainly used by specialists who seek to formalize theories in it and prove theorems. In fact the community of people formalizing mathematics is relatively small and is spread across the existent proof assistants.

This chapter starts by giving a brief overview about the interactive theorem prover COQ, namely: its specification language, proof process, canonical structures mechanism and standard library. The chapter closes by introducing a small scale reflection extension to the COQ proof system called SSREFLECT and its most important libraries for the purposes of this work, *finset* and *bigop*.

2.1 The COQ Proof Assistant

The COQ system [Teaa] is an interactive theorem prover whose underlying formal language is based on an axiom-free type theory called the Calculus of Inductive Constructions[2]. Its core logic is intuitionistic but it can be extended to a classical logic by importing the correct module. Laws like the excluded middle, $A \vee \neg A$, or the double negation elimination, $\neg\neg A \Rightarrow A$, will then become available.

[1]proofs done using a proof assistant

[2]calculus of constructions with inductive definitions

As a programming language it is capable to express most of the programs allowed in standard functional languages (one of the exceptions being the inability to express non-terminating programs which enforces restrictions on the recursion patterns allowed in the definitions).

As a proof assistant, it is designed to allow the definition of mathematical and programming objects, writing formal proofs, programs and specifications that prove their correctness. Thus, it is a well suited tool for developing safe programs but also for developing proofs in a very expressive logic (*i.e.*, high order logic), which are built in an interactive manner with the aid of automatic search tools when possible. It may also be used as logical framework in order to implement reasoning systems for modal logics, temporal logics, resource-oriented logics, or reasoning systems on imperative programs [BC04].

2.1.1 The Language

COQ objects represent types, propositions and proofs which are defined in a specification language, called *Gallina*, that allows the user to define formulas, verify that they are well-formed and prove them. However, it comprises a fairly extensive syntax and so we only touch a few important aspects (see [The06] for a complete specification of the *Gallina* language).

Every object is associated to (at least) a type and each type is an object of the system. This means that every well-formed type has a type, which in turn has a type, and so on[3]. The type of a type is always a constant and is called a `sort`. For instance

```
true : bool
0 : nat
S : nat -> nat
```

where `nat` is the type for natural numbers and `bool` contains two constants associated with truth values (`true` and `false`). Hence all objects are sorted into two categories:

- `Prop` is the sort for logical propositions (which, if well-formed, have type

[3]this happens because all types are seen as terms of the language and thus should belong to another type

`Prop`). The logical propositions themselves are the types of their proofs;

- `Set` intends to be the type for specifications (*i.e.*, programs and the usual datatypes seen in programming languages such as booleans, naturals, *etc*).

Since `sorts` should be given a type, a `Type(0)` is added as the type of sorts `Set` and `Prop` (among others). This sort, in turn, has type `Type(1)`, and so on. This gives rise to a comulative hierarchy of type-sorts `Type(i)`, for all `i` $\in \mathbb{N}$. Such a hierarchy avoids the logical inconsistencies that result from considering an axiom such as `Type:Type` [The06]. Note however that, from the user perspective, the indices of the `Type(i)` hierarchy are hidden by the system, and treated as constraints that are inferred/imposed internally.

Useful Coq types are:

`nat`	type of natural numbers (*e.g.*, `6:nat`)
`bool`	type of boolean values (*e.g.*, `true:bool`)
`Prop`	type of propositions
`Type` (or `Set`)	type of types (*e.g.*, `nat:Set`)
`T1 -> T2`	type of functions from `T1` to `T2`
`T1 * T2`	type of pairs of type `T1` and `T2`

Coq syntax for logical propositions is summarized by (first row presents the mathematical notation and the second row presents the corresponding Coq notation):

$\perp$	$\top$	$x = y$	$x \neq y$	$\neg P$	$P \vee Q$	$P \wedge Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
`False`	`True`	`x = y`	`x <> y`	`~ P`	`P \/ Q`	`P /\ Q`	`P -> Q`	`P <-> Q`

Note that the arrow (->) associates to the right and so propositions such as `T1 -> T2 -> T3` are interpreted as `T1 -> (T2 -> T3)`.

Finally we show the syntax for universal and existential quantifiers:

$\forall x, P$	$\exists x, P$
`forall x, P`	`exists x, P`
`forall x:T, P`	`exists x:T, P`

The type annotation `:T` may be omitted if `T` can be inferred by the system. Universal quantification serves different purposes in the world of COQ as it can be used for first-order quantification like in `forall x:T, x = x` or for higher-order quantification like in `forall (A:Type)(x:A), x = x` or `forall A:Prop, A -> A`. Actually, both the functional arrow and the logical implication are special cases of universal quantifications with no dependencies (*e.g.*, `T1 -> T2` states the same as `forall _:T1, T2`).

2.1.2 Proof Process

Like in almost every programming language's compiler, the core of the COQ system is a type checker, *i.e.*, an algorithm which checks whether a formula is well-formed or not. Actually it is the way COQ checks proofs' correctness: proving a lemma may be seen as the same as proving that a specific type is inhabited. This is explained by the fact that COQ sees logical propositions as the type of their proofs, meaning that if we exhibit an inhabitant of `T:Prop`, we have given a proof of `T` that witnesses its validity [Ber06]. This is the essence of the well known Curry-Howard isomorphism.

Proof development is carried in the proof editing mode through a set of commands called tactics[4] that allow a collaborative (user-machine) proof process. The proof editing mode is activated when the user enunciates the theorem, using the `Theorem` or `Lemma` commands. These commands generate a top-level goal that the user will then try to demonstrate. In each step of the process there is a list of goals to prove that are generated by the tactics applied in the step before. When there are no goals remaining (*i.e.*, the proof is completed) the command `Qed` will build a proof term from the tactics that will be saved as the definition of the theorem for further reuse.

COQ's proof development process is intended to be user guided but it still provides some kind of automatisation through a set of advanced tactics to solve complex goals. For instance:

- `auto`: *Prolog* style inference, solves trivial goals. Uses the hints from a

[4]a tactic is a program which transforms the proposition to be proved (goal) in a set of (possibly empty) new subgoals that are sufficient to establish the validity of the previous goal

database, which can be extended by the user;

- `tauto`: complete for (intuitionistic) propositional logic;
- `ring`: solves equations in a ring or semi-ring structure by normalizing both hand sides of the equation and comparing the results;
- `fourier`: solves linear inequalities on real numbers;
- `omega`: solves linear arithmetic goals.

2.1.3 Canonical Structures

COQ's canonical structures mechanism are introduced due to their importance throughout the rest of this document.

A `Canonical Structure` is an instance of a record/structure[5] type that can be used to solve equations involving implicit arguments [The06]. Its use is subtle in the SSREFLECT libraries, but provides a mechanism of proof inference (by type inference), which is used as a Prolog-like proof inference engine as often as possible [GMT08].

For the purposes of this work, canonical structures are mostly used as a subtyping mechanism. In order to demonstrate this idea, we give an example (taken from [BGOBP08]). It is possible to describe the equality of comparable types as:

```
Structure eqType : Type := EqType {
   sort :> Type;
   eqd : sort -> sort -> bool;
   _ : forall x y, (x == y) <-> (x = y)
} where "x == y" := (eqd x y).
```

This structure allows to define a unified notation for `eqd`. Note that every `eqType` contains an axiom stating that `eqd` is equivalent to the Leibnitz equality, and hence it is valid to rewrite `x` into `y` given `x == y`. Now, we can extend the notion of comparable types to the naturals. If we can prove

```
Lemma eqnP : forall m n, eqn m n <-> m = n.
```

[5]a record, or labelled product/tuple, is a macro allowing the definition of records as is done in many programming languages

for a given specific function `eqnP : nat -> nat -> bool` then we can make `nat`, a `Type`, behave as an `eqType` by declaring

```
Canonical Structure nat_eqType := EqType eqnP.
```

This creates a new `eqType` with `sort` ≡ `nat` and `eqd` ≡ `eqn` (both are inferred from the type of `eqnP`), which allows `nat` to behave as `sort nat_eqType` during type inference. Therefore Coq can now make interpretations like `n == 6`, as `eqn n 6`, or `2 == 2`, as `eqn 2 2`.

2.1.4 Standard Library

Usually, proof development is carried with the help of a trusted knowledge repository. Coq automatically loads a library that constitutes the basic state of the system. Additionally, it also includes a standard library that provides a large base of definitions and facts. It comprises definitions of standard (intuitionistic) logical connectives and properties, data types (*e.g.*, `bool` or `nat`), operations (*e.g.*, `+`, `*` or `-`) and much more, which are directly accessible through the `Require` command. Useful libraries are:

- `Logic`: classical logic and dependent equality;
- `Reals`: axiomatization of real numbers;
- `Lists`: monomorphic and polymorphic lists.

2.2 A Small Scale Reflection Extension to Coq

SSReflect[6] [Teab] is an extension to the Coq proof system which emerged as part of the formalization of the Four Colour Theorem by Georges Gonthier in 2004 [Gon05]. Therefore it uses, and extends, Coq's logical specification language, *Gallina*. It is a script language that strongly focuses on the readability and maintainability of proof scripts by giving special importance to good bookkeeping. This is essential in the context of an interactive proof development because

[6]which stands for "small-scale reflection"

it facilitates navigating the proof, thus allowing to immediately jump to its erroneous steps. This aspect gains even more importance if we think that a great part of proof scripts consist of steps that do not prove anything new, but instead are concerned with tasks like assigning names to assumptions or clearing irrelevant constants from the context.

SSREFLECT introduces a set of changes that are designed to support the use of reflection in formal proofs by reliably and efficiently automating the trivial operations that tend to delay them. In fact it only introduces three new tactics, renames three others and extends the functionality of more than a dozen of the basic COQ tactics. Several important features include:

- support for better script layout, bookkeeping and subterm selection in all tactics;
- an improved set tactic with more powerful matching. In SSREFLECT some tactics can be combined with tactic modifiers in order to deal with several similar situations:
  - a prime example is the `rewrite` tactic. The extended version allows to perform multiple rewriting operations, simplifications, folding/unfolding of definitions, closing of goals and *etc.*;
- a "view" mechanism that allows to do bookkeeping and apply lemmas at the same time. It relies on the combination of the / view switch with bookkeeping tactics (and tactics modifiers);
- better support for proofs by reflection.

Note that only the last feature is specific to small-scale reflection. This means that most of the others are of general purpose, and thus are intended for normal users. Such features aim to improve the functionality of COQ in areas like proof management, script layout and rewriting.

SSREFLECT tries to divide the workload of the proof between the prover engine and the user, by providing computation power and functions that the user may use in his proof scripts. These scripts comprise three kinds of steps [GMT08]:

- deduction steps: to specify part of the construction of the proof;

- bookkeeping steps: to manage the proof context by introducing, renaming, discharging or splitting constants and assumptions;
- rewriting steps: to locally change parts of the goal or assumptions.

SSREFLECT is fully backward compatible, *i.e.*, any COQ proof that does not use any new feature should give no problems when compiling with the extended system [GMT08].

2.3 Important SSREFLECT Libraries

The SSREFLECT framework includes a vast repository of algebraic and number theoretic definitions and results (see `http://ssr.msr-inria.inria.fr/~hudson/current/`), as a result of the contribution of a large number of people. This library is divided into files according to a specific area of reasoning. For example, *ssralg* provides definitions for the type, packer and canonical properties of algebraic structures such as fields or rings [GGMR09] whereas *finfun* implements a type for functions with a finite domain.

Using this framework will almost certainly take the user to work with more than one of these theory files due to the hierarchical way they are organized. Depending on what one intends to do, some will have higher importance and play a more active role whereas others will merely be needed as the backbone of the former. While developing this framework, two libraries proved to be of extreme importance as their results had a direct impact in most definitions and proofs processes. Next we give a slight overview about *finset*[7] and *bigop*[8], two SSREFLECT libraries.

2.3.1 finset

finset is an SSREFLECT library that defines a type for sets over a finite type, which is based on the type of functions over a finite type defined in *finfun*[9]. A finite type,

[7]`http://ssr.msr-inria.inria.fr/~hudson/current/finset.html`
[8]`http://ssr.msr-inria.inria.fr/~hudson/current/bigop.html`
[9]`http://ssr.msr-inria.inria.fr/~hudson/current/finfun.html`

or `finType`, is defined as a type with decidable equality (`eqType`) together with a sequence enumerating its elements (together with a property stating that the sequence is duplicate-free and includes every member of the given type) [GMT08]. The family of functions defined in *finfun* is implemented with a `finType` domain and an arbitrary codomain as a tuple[10] of values, and has a signature of the type `{ffun aT -> rT}`, where `aT` must have a `finType` structure and is a `#|aT|.-tuple rT` (*i.e.*, a tuple with `#|aT|`[11] elements of type `rT`).

Boolean functions are a special case of these functions, as they allow to define a mask on the `finType` domain, and therefore inherit important properties of the latter, such as (Leibniz) intentional and extensional equalities. They are denoted by `{set T}` and defined as:

```
Inductive set_type (T : finType) := FinSet of {ffun pred T}.
```

where `T` must have a `finType` structure and `pred` is a boolean predicate `(T -> bool)`.

The library is extensive and contains many important results of set theory, such as De Morgan's laws.

Notations

For types `A, B: {set T}` we highlight the following notations:

`x \in A`	`x` belongs to `A`
`set0`	the empty set
`setT` or `[set: T]`	the full set
`A :|: B`	the union of `A` and `B`
`A :&: B`	the intersection of `A` and `B`
`A :\: B`	the difference `A` minus `B`
`~: A`	the complement of `A`
`\bigcup_<range> A`	iterated union over `A`, for all `i` in `<range>`. `i` is bound in `A`
`\bigcap_<range> A`	iterated intersection over `A`, for all `i` in `<range>`. `i` is bound in `A`

[10]SSREFLECT treats tuples as sequences with a fixed (known) length
[11]# denotes the cardinality of aT

Most of these notations regard basic set operations but there are also results that allow the user to work with partitions or powersets, for example. Moreover, it is also possible to form sets of sets: since `{set T}` has itself a `finType` structure one can build elements of type `{set {set T}}`. This leads to two additional important notations: for `P: {set {set T}}`:

`cover P`	the union of the set of sets `P`
`trivIset P`	the elements of `P` are pairwise disjoint

Main Lemmas

We are mainly interested in lemmas regarding set operations (*i.e.*, union, intersection, complement and De Morgan's laws), as they play a crucial role in probability theory (see Chapter 3).

```
Lemma setIC A B : A :&: B = B :&: A.
Lemma setIA A B C : A :&: (B :&: C) = A :&: B :&: C.
Lemma setIUr A B C : A :&: (B :|: C) = (A :&: B) :|: (A :&: C).
```

lemmas `setIC`, `setIA` and `setIUr` represent the commutativity, associativity and distributivity of intersection, respectively. On the other hand:

```
Lemma setUC A B : A :|: B = B :|: A.
Lemma setUA A B C : A :|: (B :|: C) = A :|: B :|: C.
Lemma setUIr A B C : A :|: (B :&: C) = (A :|: B) :&: (A :|: C).
```

`setUC`, `setUA` and `setUIr` represent the same properties, but now regarding the union of sets.

Since `~:` denotes the complement, we can state, in COQ, the De Morgan's laws as follows (see section 3.1) as follows:

```
Lemma setCU A B : ~: (A :|: B) = ~: A :&: ~: B.
Lemma setCI A B : ~: (A :&: B) = ~: A :|: ~: B.
```

Finally, the next lemmas state properties regarding the interaction between an arbitrary set and the empty/full set.

```
Lemma setU0 A : A :|: set0 = A.
Lemma setI0 A : A :&: set0 = set0.
Lemma setUT A : A :|: setT = setT.
Lemma setIT A : A :&: setT = A.
```

Note that the names of all lemmas follow a specific pattern: they start by the word "set" and end with a specific suffix, which is associated to the operation in scope: I for intersection, U for union, D for difference, C for complement or commutativity, A for associativity, 0 for empty set and T for full set.

2.3.2 bigop

bigop is an SSReflect library that contains a generic theory of big operators (*i.e.*, sums, products or the maximum of a sequence of terms), including unique lemmas that perform complex operations such as reindexing and dependent commutation, for all operators, with minimal user input and under minimal assumptions [BGOBP08]. It relies on Coq's canonical structures to express relevant properties of the two main components of big operators, indexes and operations, thus enabling the system to infer such properties automatically.

To compute a big operator it is necessary to enumerate the indices in its range. If the range is an explicit sequence of `type T`, where `T` has an `eqType` structure (*i.e.*, it is a type with decidable Leibniz equality) this computation is trivial. However, it is also possible to specify the range as a predicate, in which case it must be possible to enumerate the entire index type (*i.e.*, work with a `finType`).

Notations

This library provides a generic notation that is independent from the operator being used. It receives the operator and the value for empty range as parameters, and has the following general form:

```
<bigop>_<range> <general_term>
```

- `<bigop>` is one of `\big[op/idx]` (where `op` is the operator and `idx` the value for empty range): `\sum`, `\prod` or `\max` for sums, products or maximums, respectively;
- `<general_term>` can take the form of any expression;

- `<range>` binds an index variable `i` in `<general_term>` and states the set over which it iterates; `<range>` is one of:

`(i <- s)`	`i` ranges over the sequence `s`
`(m <= i < n)`	`i` ranges over the natural interval `[m.. n-1]`
`(i < n)`	`i` ranges over the (finite) type `'I_n` (*i.e.*, ordinal `n`)
`(i : T)`	`i` ranges over the finite type `T`
`i` or `(i)`	`i` ranges over its inferred finite type
`(i \in A)`	`i` ranges over the elements that satisfy the predicate `A`, which must have a finite type domain
`(i <- s | C)`	limits the range to those `i` for which `C` holds

There are three ways to give the range: via a sequence of values, via an integer interval or via the entire type of the bound variable, which must then be a `finType` . In all three cases, the variable is bound to `<general_term>`. Additionally, it is also possible to filter the range with a predicate - in this case the big operator will only take the values from the range that satisfy the predicate. This definition should not be used directly but through the notations provided for each specific operator, which will allow a more natural use of big operators.

The computation of any big operator is implemented by the following code:

```
Definition reducebig R I op idx r (P : pred I) (F : I -> R) : R :=
       foldr (fun i x => if P i then op (F i) x else x) idx r.

Notation "\big [ op / nil ]_ ( i <- r | P ) F" :=
       (reducebig op nil r (fun i => P%B) (fun i => F)) : big_scope.
```

Therefore all big operators reduce to a *foldr*, which is a recursive high-order function that iterates a specific function over a sequence of values `r`, combining them in order to compute a single final value. For each element in `r`, *foldr* tests if it satisfies the predicate `P`: if it does so, the value of `F` on that element is computed and combined with the rest. The value `idx` is used at the end of the computation.

Main Lemmas

bigop offers around 80 lemmas to deal with big operators. They are organized in two categories:

- lemmas which are independent of the operator being iterated:
  - extensionality with respect to the range, to the filtering predicate or to the expression being iterated;
  - reindexing, widening or narrowing of the range of indices;
- lemmas which are dependent on the properties of the operator. In particular, operators that respect:
  - a plain monoid structure, with only associativity and an identity element (*e.g.*, splitting);
  - an abelian monoid structure, whose operation is commutativity (*e.g.*, permuting);
  - a semi-ring structure (*e.g.*, exchanging big operators).

Despite the wide variety of results available, some lemmas proved to be more important than others. For example, we may use:

```
Lemma eq_bigl r (P1 P2 : pred I) F :
   P1 =1 P2 ->
 \big[op/idx]_(i <- r | P1 i) F i = \big[op/idx]_(i <- r | P2 i) F i.
```

to rewrite a big operation's range, or

```
Lemma eq_bigr r (P : pred I) F1 F2 :
   (forall i, P i -> F1 i = F2 i) ->
 \big[op/idx]_(i <- r | P i) F1 i = \big[op/idx]_(i <- r | P i) F2 i.
```

to rewrite a big operation's formula. Similarly, we may use the general rule,

```
Lemma eq_big : forall (r : seq I) (P1 P2 : pred I) F1 F2 :
    P1 =1 P2 -> (forall i, P1 i -> F1 i = F2 i) ->
  \big[op/idx]_(i <- r | P1 i) F1 i
         = \big[op/idx]_(i <- r | P2 i) F2 i,
```

which allows to rewrite a big operation in the predicate or the expression parts. It states that two big operations can be proved equal given two premises: `P1 =1 P2`, which expresses that it suffices that both predicates should be extensionally equal, and `(forall i, P1 i -> F1 i = F2 i)`, which states that both expressions should be extensionally equal on the subset of the type determined by the predicate `P1` [BGOBP08].

Another interesting lemma is the one that allows to permute nested big operators:

```
Lemma exchange_big (I J : finType) (P : pred I) (Q : pred J) F :
  \big[*%M/1]_(i | P i) \big[*%M/1]_(j | Q j) F i j =
    \big[*%M/1]_(j | Q j) \big[*%M/1]_(i | P i) F i j.
```

This is achieved by first showing that two nested big operations can be seen as one big operation that iterates over pairs of indices. It is then applied a re-indexing operation on the pairs in order to obtain this commutation lemma. The notation `*%M` indicates that the operator associated to the big operation has an abelian monoid structure.

To finish, we emphasize yet another important property, distributivity among operators:

```
Lemma big_distrl I r a (P : pred I) F :
  \big[+%M/0]_(i <- r | P i) F i * a
        = \big[+%M/0]_(i <- r | P i) (F i * a).
```

where both operators have a semi-ring structure. The `+%M` notation denotes addition in such structure and `*%M` its multiplication.

Chapter 3

Elements of Probability Theory

The word probability derives from the Latin *probare* which means to prove or to test. Although the scientific study of probabilities is a relatively modern development, the notion of this concept has its most remote origins traced back to the Middle Ages[1] in attempts to analyse games of chance.

Informally, *probable* is one of many words used to express knowledge about a known or uncertain event. Through some manipulation rules, this concept has been given a precise mathematical meaning in probability theory, which is the branch of mathematics concerned with the analysis of random events. It is widely used in areas such as gambling, statistics, finance and others.

In probability theory one studies models of random phenomena. These models are intended to describe random events, that is, experiments where future outcomes cannot be predicted, even if every aspects involved are fully controlled, and where there is some randomness and uncertainty associated. A trivial example is the tossing of a coin: it is impossible to predict the outcome of future tosses even if we have full knowledge of the characteristics of the coin.

To understand the algorithmic aspects of number theory and algebra, and to know how they are connected to areas like cryptography, one needs an overview of the basic notions of probability theory. This chapter introduces fundamental concepts from probability theory (with the same notation and similar presentation as in [Sho05]), starting with some basic notions of probability distributions on finite sample spaces and then closing with a brief overview about random

[1]period of European history from the 5th century to the 15th century

variables and distributions based on them.

3.1 Basic Notions

Let Ω be a finite, non-empty set. A probability distribution on Ω is a function $P : \Omega \to [0,1]$ that satisfies the following property:

$$\sum_{\omega \in \Omega} P(\omega) = 1. \tag{3.1}$$

The elements ω are the possible outcomes of the set Ω, known as the sample space of P, and $P(\omega)$ is the probability of occurrence of that outcome. Also, one can define an event as a subset $\mathcal{A}$ of Ω with probability defined as follows:

$$P[\mathcal{A}] := \sum_{\omega \in \mathcal{A}} P(\omega). \tag{3.2}$$

Clearly, for any probability distribution P, and every event A,

$$P[\mathcal{A}] \geq 0 \tag{3.3}$$

stating that every event has a non-negative probability of occurring.

Additionally, if $\{\mathcal{A}_1, \mathcal{A}_2, ...\} \in \Omega$ are pairwise disjoint events (i.e., $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for every $i \neq j$) then,

$$P[\cup_i \mathcal{A}_i] = \sum_i P[\mathcal{A}_i], \tag{3.4}$$

which just states that, for any sequence of mutually exclusive events, the probability of at least one of these events occurring is just the sum of their respective probabilities [Ros09b]. Formulas (3.1), (3.3) and (3.4) are called the Axioms of Probability[2].

In addition to working with probability distributions over finite sample spaces, one can also work with distributions over infinite sample spaces. However, for

[2]or Kolmogorov's axioms

the purpose of this work we will only consider the former case.

It is possible to logically reason over sets using rules such as De Morgan's laws, which relate to the three basic set operations: union, intersection and complement. For events $\mathcal{A}$ and $\mathcal{B}$ these operations are graphically represented in Figure 3.1 and can be defined as:

(*i*) $\mathcal{A} \cup \mathcal{B}$: denotes the logical union between $\mathcal{A}$ and $\mathcal{B}$, that is, it represents the event where *either* the event $\mathcal{A}$ *or* the event $\mathcal{B}$ occurs (or both);

(*ii*) $\mathcal{A} \cap \mathcal{B}$: denotes the intersection between $\mathcal{A}$ and $\mathcal{B}$ (logically represents the event where both occur);

(*iii*) $\overline{\mathcal{A}} := \Omega \setminus \mathcal{A}$: denotes the complement of $\mathcal{A}$, that is, it represents the event where $\mathcal{A}$ does not occur.

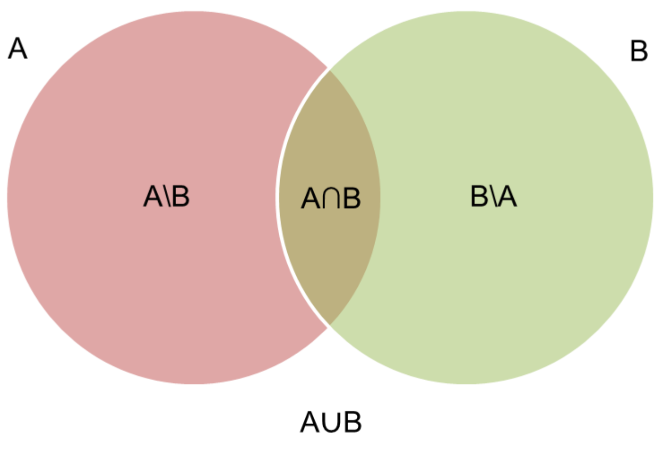

Figure 3.1: Venn diagram for set operations: union, intersection and complement.

From (*i*), (*ii*), (*iii*) and using the usual Boolean logic follows a formal definition of De Morgan's laws:

$$\overline{\mathcal{A} \cup \mathcal{B}} = \overline{\mathcal{A}} \cap \overline{\mathcal{B}} \tag{3.5}$$

$$\overline{\mathcal{A} \cap \mathcal{B}} = \overline{\mathcal{A}} \cup \overline{\mathcal{B}} \tag{3.6}$$

Moreover, Boolean laws such as commutativity, associativity and distributivity also play an important role. For all events $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$:

- Commutativity: $\mathcal{A} \cup \mathcal{B} = \mathcal{B} \cup \mathcal{A}$ and $\mathcal{A} \cap \mathcal{B} = \mathcal{B} \cap \mathcal{A}$;
- Associativity: $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C} = \mathcal{A} \cup (\mathcal{B} \cup \mathcal{C})$ and $\mathcal{A} \cap \mathcal{B} \cap \mathcal{C} = \mathcal{A} \cap (\mathcal{B} \cap \mathcal{C})$;
- Distributivity: $\mathcal{A} \cup (\mathcal{B} \cap \mathcal{C}) = (\mathcal{A} \cup \mathcal{B}) \cap (\mathcal{A} \cup \mathcal{C})$ and $\mathcal{A} \cap (\mathcal{B} \cup \mathcal{C}) = (\mathcal{A} \cap \mathcal{B}) \cup (\mathcal{A} \cap \mathcal{C})$.

One can also derive some basic facts about probabilities. The probability of an event $\mathcal{A}$ is 0 if it is impossible to happen or 1 if it is certain. Therefore, the probability of an event ranges between 0 and 1. Additionally,

$$P[\overline{\mathcal{A}}] = 1 - P[\mathcal{A}]. \tag{3.7}$$

Considering the union of events $\mathcal{A}$ and $\mathcal{B}$, we have:

$$P[\mathcal{A} \cup \mathcal{B}] = P[\mathcal{A}] + P[\mathcal{B}] - P[\mathcal{A} \cap \mathcal{B}], \tag{3.8a}$$

$$P[\mathcal{A} \cup \mathcal{B}] = P[\mathcal{A}] + P[\mathcal{B}],\ if\ A \cap B = \emptyset. \tag{3.8b}$$

Equation (3.8b) addresses the case where $\mathcal{A}$ and $\mathcal{B}$ are mutually exclusive, that is, they cannot occur at the same time (no common outcomes).

Regarding intersection of events there are also theorems worth mentioning:

$$P[\mathcal{A} \cap \mathcal{B}] = P[\mathcal{A}|\mathcal{B}] \cdot P[\mathcal{B}], \tag{3.9a}$$

$$P[\mathcal{A} \cap \mathcal{B}] = P[\mathcal{A}] \cdot P[\mathcal{B}],\ if\ A\ and\ B\ are\ independent\ events. \tag{3.9b}$$

Once again the second equation, (3.9b), addresses a special case where $\mathcal{A}$ and $\mathcal{B}$ are independent events (i.e. the occurrence of one event makes it neither more nor less probable that the other occurs). (3.9a) introduces a new concept, conditional probability, which we explain next.

3.1.1 Conditional Probability

Informally, one can simply define conditional probability as the probability measure of an event after observing the occurrence of another one. Otherwise, it can be formally characterized as: let $\mathcal{A}$ and $\mathcal{B}$ be events and $P[\mathcal{B}] > 0$ (i.e., $\mathcal{B}$ has positive probability). The conditional probability of event $\mathcal{A}$ happening given that $\mathcal{B}$ already occurred is defined as:

$$P[\mathcal{A}|\mathcal{B}] = \frac{P[\mathcal{A} \cap \mathcal{B}]}{P[\mathcal{B}]}, \tag{3.10}$$

note that if $P[\mathcal{B}] = 0$ (i.e., it is impossible), then, $P[\mathcal{A}|\mathcal{B}]$ is undefined[3]. If $\mathcal{A}$ and $\mathcal{B}$ are independent it is clear that they are not conditionalized on each other, so:

$$P[\mathcal{A}|\mathcal{B}] = P[\mathcal{A}], \tag{3.11a}$$

$$P[\mathcal{B}|\mathcal{A}] = P[\mathcal{B}]. \tag{3.11b}$$

From (3.11a) and (3.11b) it is possible to deduce that independence means that observing an event has no impact on the probability of the other to occur. As a side note it is important to emphasize that some of the most relevant results in probability theory, such as the law of total probability or the Bayes' theorem, were built on top of the conditional probability. These results are discussed next.

3.1.2 Law of Total Probability

Let $\{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_i\}_{i \in I}$ be a finite and pairwise disjoint family of events, indexed by some set I, whose union comprises the entire sample space. Then, for any event $\mathcal{A}$ of the same probability space:

[3]Borel-Kolmogorov paradox

$$P[\mathcal{A}] = \sum_{i \in I} P[\mathcal{A} \cap \mathcal{B}_i]. \tag{3.12}$$

Moreover, if $P[\mathcal{B}_i] > 0$ for all i (i.e., every $\mathcal{B}_i$ is a partition of Ω), we have:

$$P[\mathcal{A}] = \sum_{i \in I} P[\mathcal{A}|\mathcal{B}_i] \cdot P[\mathcal{B}_i]. \tag{3.13}$$

Equations (3.12) and (3.13) are known as the law of total probability. This is an important result since, by relating marginal probabilities ($P[\mathcal{B}_i]$) to the conditional probabilities of $\mathcal{A}$ given $\mathcal{B}_i$, it allows to measure the probability of a given event to occur.

3.1.3 Bayes' Theorem

Recall the family of events $\{\mathcal{B}_1, \mathcal{B}_2, ..., \mathcal{B}_i\}_{i \in I}$ and the event $\mathcal{A}$ from section 3.1.2. Suppose that $P[\mathcal{A}] > 0$, then, for all $j \in I$ we have:

$$P[\mathcal{B}_j|\mathcal{A}] = \frac{P[\mathcal{A} \cap \mathcal{B}_j]}{P[\mathcal{A}]} = \frac{P[\mathcal{A}|\mathcal{B}_j] \cdot P[\mathcal{B}_j]}{\sum_{i \in I} P[\mathcal{A}|\mathcal{B}_i] \cdot P[\mathcal{B}_i]}, \tag{3.14}$$

Equation (3.14) is known as the Bayes' theorem, or the principle of inverse probability [Jay58], and it can be seen as a way to perceive how the probability of an event to occur is affected by the probability of another one. It has been mostly used in a wide variety of areas such as science and engineering, or even in the development of "Bayesian" spam blockers for email systems[4] [SDHH98].

Following is a real life example, from [Sho05], regarding the application of the Bayes' theorem. It is one of many versions of the same problem, famous for the counter intuitive manner that it presents the solution. The reader can check the Monty Hall problem [Ros09a] for another famous problem involving similar reasoning.

Suppose that the rate of incidence of disease X in the overall population is 1% and there is a test for this disease; however, the test is not perfect: it has a 5% false

[4]which work by observing the use of tokens, typically words, in e-mails and then using Bayesian inference to measure the probability that an e-mail is or is not spam

positive rate, that is, 5% of healthy patients test positive for the disease, and a 2% false negative rate (2% of sick patients test negative for the disease). Advised by his doctor, a patient does the test and it comes out positive. What should the doctor say to his patient? In particular, what is the probability that the patient actually has disease X, given the test result turned out to be positive?

Surprisingly, the majority of people, including doctors and area expertises, will say the probability is 95%, since the test has a false positive rate of 5%. However, this conclusion could not be far from truth.

Let $\mathcal{A}$ be the event that the test is positive and $\mathcal{B}$ be the event that the patient has disease X. The relevant quantity that one needs to estimate is $P[\mathcal{A}|\mathcal{B}]$, that is, the probability that the patient has disease X given a positive test result. Using Bayes' theorem to do this leads to:

$$P[\mathcal{B}|\mathcal{A}] = \frac{P[\mathcal{A}|\mathcal{B}] \cdot P[\mathcal{B}]}{P[\mathcal{A}|\mathcal{B}] \cdot P[\mathcal{B}] + P[\mathcal{A}|\overline{\mathcal{B}}] \cdot P[\overline{\mathcal{B}}]} = \frac{0.98 \cdot 0.01}{0.98 \cdot 0.01 + 0.05 \cdot 0.99} \approx 0.17.$$

Thus, a patient whose test gave a positive result only have 17% chance to really have disease X. So, the correct intuition here is that it is much more likely to get a false positive than it is to actually have the disease.

3.2 Random Variables and Distributions

Generally, we are not interested (only) in events within the sample space, but rather in some function on them. In most cases it is necessary to associate a real number, or other mathematical object[5], to each of the outcomes of a given event. For example, suppose one plays a game that consists on rolling a dice and counting the number of dots faced up; furthermore, suppose one receives 1 euro if the total number of dots equals 1 or 2, 2 euros if the total number of dots is 3 or 4 and that one has to pay 5 euros otherwise. So, as far as "prizes" is concerned, we have three groups of dots: $\{1,2\}$, $\{3,4\}$ and $\{5,6\}$. This means that our "prize" is a function of the total number of dots faced up after rolling the dice. Now, to calculate the probability we have to win the the 2 euros prize (or any other), we

[5]such as boolean values, functions, complex numbers, *etc.*

just have to compute the probability that the total number of dots faced up after rolling the dice falls into the class $\{3, 4\}$, which corresponds to the 2 euros prize. The notion of a random variable formalizes this idea [Sho05].

Intuitively, random variable is a measure of the outcome of a given event. It is formally characterized by a function from the probability space to an abstract set:

$$X : \Omega \to S. \tag{3.15}$$

Despite existing several types of random variables, the two most used are the discrete and the continuous [Fri97]. Suppose that a coin is tossed into the air 5 times and that X represents the number of heads occurred in the sequence of tosses. Because the coin was only tossed a finite number of times, X can only take a finite number of values, so it is known as a discrete random variable. Similarly, suppose that X is now a random variable that indicates the time until a given ice cream melts. In this case X can take an infinite number of values and therefore it is known as a continuous random variable.

In order to specify the probabilities associated with each possible value of a random variable, it is often necessary to resort to alternative functions from which the probability measurement immediately follows. These functions will depend on the type of the random variable: because we are only interested in discrete random variables, suppose X takes on a finite set of possible outcomes (*i.e*, X is a discrete random variable), in this case the easiest way to specify the probabilities associated with each possible outcome is to directly assign to each one a probability. From (3.15) we can now determine a new function:

$$p_X(x) : S \to [0, 1], \tag{3.16}$$

where $p_X(x) := P[X = x]$, for each $x \in S$. As we are dealing with discrete random variables p_X (or simply p if the random variable is understood from context) is known as a probability mass function (pmf), the distribution of X. Similarly, there is a specific family of functions that characterizes the distribution of a continuous random variable, the probability density function.

A pmf is a probability distribution, and hence satisfies their characterizing properties, namely:

$$0 \leq P[X = x] \leq 1, \tag{3.17}$$

$$\sum_{x \in S} P[X = x] = 1, \tag{3.18}$$

which are similar to the properties that we have introduced at the beginning of section 3.1.

Throughout the rest of this document the term random variable will be used when referring discrete random $\mathbb{R}$-valued variables[6] (unless told otherwise). These variables will be denoted by upper case letters.

3.2.1 Joint Distribution

Probability distributions can also be applied to a group of random variables, in situations where one may want to know several quantities in a given random experiment [MD00]. In such cases, these probability distributions are called joint distributions: they define the probability of an event happening in terms of all random variables involved.

However, for the purposes of this work, we will only consider cases that just concern two random variables. Thus, given X and Y, defined on the same probability space and with codomains S^x and S^y, we can characterize their joint probability mass function as:

$$p_{XY}(x,y) : S^x x\, S^y \rightarrow [0,1], \tag{3.19}$$

where $p_{XY}(x,y) := P[X = x, Y = y]$, for each $x \in S^x$ and $y \in S^y$. This joint pmf is again a probability distribution and hence,

$$0 \leq p_{XY}(x,y) \leq 1, \tag{3.20}$$

that is, the joint probability of two random variables ranges between 0 and 1 and,

[6]random variables with a distribution that is characterized by a pmf and whose image is the real numbers

$$\sum_{x \in S^x} \sum_{y \in S^y} P[X = x, Y = y] = 1, \tag{3.21}$$

which states that the sum of all joint probabilities equals to 1 with respect to a specific sample space.

From the joint pmf of X and Y we can obtain the marginal distribution of X by:

$$p_X(x) = \sum_{y \in S^y} p_{XY}(x, y), \tag{3.22}$$

and similarly for Y:

$$p_Y(y) = \sum_{x \in S^x} p_{XY}(x, y). \tag{3.23}$$

Moreover, if X and Y are independent, their joint pmf may be defined by the multiplication of their independent probabilities, that is:

$$p_{XY}(x, y) = p_X(x) \cdot p_Y(y). \tag{3.24}$$

3.2.2 Conditional Distribution

Recall equation (3.10), which defines the conditional probability of an event given the occurrence of another. If X and Y are random variables it is thus natural to define their conditional pmf as:

$$p_{X|Y}(x, y) = \frac{p_{XY}(x, y)}{p_Y(y)}, \tag{3.25}$$

where $p_{X|Y}(x, y) := P[X|Y = y]$, for each $x \in S^x$ and $y \in S^y$. This means that conditional distributions seek to answer the question: which is the probability distribution of a random variable given that another takes a specific value?

If X and Y are independent random variables, then:

$$p_{X|Y}(x, y) = p_X(x), \tag{3.26}$$

and,

$$p_{Y|X}(y, x) = p_Y(y). \tag{3.27}$$

Joint distributions relate to conditional distributions through the following two important properties:

$$p_{XY}(x, y) = p_{(X|Y=y)}(x) \cdot p_Y(y), \tag{3.28}$$

$$p_{XY}(x, y) = p_{(Y|X=x)}(y) \cdot p_X(x), \tag{3.29}$$

Chapter 4

Finite Probability Distributions in Coq

A probability lays on a hierarchy of numerous other concepts, without which it cannot be assembled. Recall, from chapter 3, that an event can be described by a random variable and that a random variable can be characterized by a distribution. These concepts, in turn, are defined by functions and therefore enjoy a number of specific mathematical properties that one needs to ensure in order to guarantee the correct specification of a probability.

Suppose you are building a house: it makes sense to start by building the foundations and then to continue all the way up until the roof, so the house does not collapse upon itself, right? Similarly, in this case the most important aspect is to ensure the proper specification and implementation of all concepts involved, in a bottom up way. So, if we want to be able to work with probabilities it is advisable to first specify the basic components that are a part of them. Although not being at the bottom of the hierarchy, the functions that underlie the definition of probability, necessary to formalize both distributions and random variables, can be classified as the key components for the construction of a framework focused on reasoning over probabilities.

Fortunately, SSREFLECT is equipped with tools that are ideal to work with such objects. For instance, *finset* module offers the possibility to work with sets by defining a type for sets over a finite type. Also, *bigop* provides a generic definition for iterating an operator over a set of indexes. Once knowing how to work with finite sets and how to iterate over their possible values one can start defining the basic concepts that underlie the probability of an event, like its sample space, and from there continue developing the framework. Of course there are still many properties one needs to ensure, but more on that later.

The development of such a framework was not always steady and needed a lot of tweaking along the way. This chapter presents and discusses some of the design decisions made through the developing stage of this work. It closes by presenting a possible application for the formalized concepts.

4.1 An Approach to Finite Probability Distributions in COQ

Section 3.1 introduced basic notions about probabilities. Those concepts represented the basis for the construction of a probability-oriented framework and naturally appeared as the first ones in need to be formally characterized. However there were important aspects to take care first before approaching such matters.

We knew that eventually conditional probabilities would appear along the way, which meant that we were destined to deal with fractions at a certain point. For this reason we chose to define probabilities over the rational numbers (since COQ defines them as fractions), although they are generally defined over real numbers, using a library developed under [Mah06]. This library provided indispensable theory, necessary to work with rationals - addition, multiplication, subtraction, *etc.* - along with proofs of relevant properties regarding their arithmetic - associativity, commutativity, distributivity, etc.

Next we present the initial approach to the implementation of the framework with a special focus on the basic concepts introduced in sections 3.1 and 3.1.1.

4.1.1 Specification of a Finite Probability Distribution

Recall definition (3.1) from section 3.1: an event is a set of outcomes, and a subset of the sample space, to which a probability is assigned. Each probability is given by its probability function and can be specified in COQ as follows:

```
Structure pmf := Pmf {
  pmf_val :> {ffun aT -> VType};
  _ : (forallb x, (0 <= pmf_val x)) && \sum_(x:aT) pmf_val x == 1
}.
```

Since we are only interested in discrete distributions, this definition only concerns pmf's. It contains:

- a function `pmf_val` that maps each of the event's outcome to a rational number `VType`;
- two properties that every pmf must satisfy:
  - each outcome has a non-negative probability of occurring;
  - the sum of all probabilities, regarding the same sample space, equals 1.

The `:>` symbol makes `pmf_val` into a coercion, which means we can use a `P : pmf` as if it were a `{ffun aT -> VType}` - COQ's type inference system will insert the missing `pmf_val` projection. The rest are important aspects, stated before, we had to guarantee in order to characterize a discrete distribution. Both properties are well known axioms of probability and hence we can just impose them to hold. From here, the probability of an event just follows as:

```
Variable aT: finType.
Variable d: pmf aT.

Definition prob (E: {set aT}) := \sum_(x \in E) (d x).
```

Thus, the probability of an event is given by the sum of all atomic events that compose it. In the definition above, `d` denotes the pmf that characterizes the probability distribution of the event `E` whereas `d x` maps a specific outcome `x` to its probability. An event is defined as a set over a finite type mainly because of two things:

- it allows handling the sums within the distributions in an efficient and easy way;

- we can use the tens of lemmas *finset* had to offer directly. This greatly smooths the work to be done since those lemmas include results like De Morgan's laws or others related to set theory.

In mathematical languages, definitions can often get complex or big enough to make it hard for the user to use or understand them. In COQ we can redefine the way those definitions look, by assigning new notations, in order to improve their cleanliness. Most users are used to a notation similar to the one presented in chapter 3 so we decided to follow the same path:

```
Notation "\Pr_ d '[' E ']'" := (prob d E)
  (at level 41, d at level 0, E at level 53,
          format "'[' \Pr_ d '/' [ E ] ']'") : prob_scope.
```

This notation allows to express the probability of an event as `\Pr_d[A]`, where `A` represents an event or a set operation. The idea is to enable the user to write lemmas or other definitions in a more natural and comfortable way.

4.1.2 Main Lemmas

After defining the concept of probability, it makes sense to move into the formalization of properties that will actually allow to handle it. At this point, the most logical thing to do is to start with the ones that we talked about in section 3.1. Union, intersection and complement are extremely important results and thus are often used in other proofs. But first, take a glimpse at an important property many times disregarded:

```
Lemma prob_decomp : forall A B,
        \Pr_d[A] = \Pr_d[A :&: B] + \Pr_d[A :&: ~:B].
```

To prove this lemma it suffices to use results available in *finset*. First start by showing that $A = A \cap S$, where S represents the full set, and then that $A \cap S = A \cap (B \cup \overline{B})$ since the union of a set with its complement always equals the full set. Using intersection's distributivity leads to $A \cap (B \cup \overline{B}) = (A \cap B) \cup (A \cap \overline{B})$ and to a goal state of the proof that looks like `\Pr_d[(A :&: B):|: (A :&: ~:B)] = \Pr_d[(A :&: B)] + \Pr\d[(A :&: ~:B)]`, where `:|:` denotes the union between sets and is defined in *finset*. Recall from equation (3.8b) that we can expand the union of two events into their sum if they are mutually independent. Since the probability of an event is defined over big operators (as a summation), we will have to manage the proof at that level in order to complete it:

```
Lemma big_setU: forall (R : Type) (idx : R) (op : Monoid.com_law idx)
  (I : finType) (A B : {set I}) (F : I -> R),
     [disjoint A & B] ->
  \big[op/idx]_(i \in A :|: B) F i =
      op (\big[op/idx]_(i \in A) F i) (\big[op/idx]_(i \in B) F i).
```

This lemma allows to do just that. It expresses that a big operation, whose range is a predicate that looks like $i \in A \cup B$, can be unfolded into an operation between two big operators, where each one's range contains a predicate regarding one of the sets[1], given that the premise `[disjoint A & B]` holds. This is exactly what we need to finish the proof - applying this lemma will lead to a last goal to be proven, `[disjoint A :&: B & A :&: ~:B]`, which is true. Note that `[disjoint A & B]`[2] is a boolean expression that evaluates to true if, and only if, the boolean predicates `A, B:pred T` (with `T:finType`) are extensionally equal to `pred0`[3], that is, they are disjoint.

Mostly, we intended to prove properties introduced in section 3.1 but along the way many others emerged. They turned out to be necessary in order to achieve such initial goals. This happened with `prob_decomp`, which despite not being one of the main properties we aimed to prove, later emerged as a relevant result (*e.g.*, it was vital to prove equation (3.8a)).

There is no perfect recipe for theorem proving and sometimes it can get really troublesome. The important thing is to know how to approach the problem and never give up, because certainly many obstacles will appear. In this case, we soon realized three important aspects:

- almost all proofs will involve set operations properties manipulation;
- sometimes it will probably be necessary to manage the proof at a lower level (*i.e.*, handle the big operators *per se*);

[1] $i \in A$ or $i \in B$ since $i \in (A \cup B) = (i \in A) \cup (i \in B)$

[2] and similarly, `[disjoint A :&: B & A :&: ~:B]`

[3] `pred0` is defined in the *ssrbool* library and denotes the predicate that always returns false

- some proofs will rely on additional results (*i.e.*, other properties we are also aiming to prove).

This is reflected in the next example. Consider equation (3.8a), another important result of probability theory regarding intersection of events: if one had to prove its validity in an informal way, like by pencil-and-paper, it could be done like this:

$$
\begin{aligned}
P[\mathcal{A} \cup \mathcal{B}] &= P[\mathcal{A}] + P[\mathcal{B}] - P[\mathcal{A} \cap \mathcal{B}] \\
&\overset{1}{\Longleftrightarrow} \\
&= P[\mathcal{A} \cap \mathcal{B}] + P[\mathcal{A} \cap \overline{\mathcal{B}}] + P[\mathcal{B}] - P[\mathcal{A} \cap \mathcal{B}] \\
&\overset{2}{\Longleftrightarrow} \\
&= (1 - P[\overline{\mathcal{B}}]) + P[\mathcal{A} \cap \overline{\mathcal{B}}] \\
&\overset{3}{\Longleftrightarrow} \\
&= 1 - (P[\mathcal{A} \cap \overline{\mathcal{B}}] + P[\overline{\mathcal{A}} \cap \overline{\mathcal{B}}]) + P[\mathcal{A} \cap \overline{\mathcal{B}}] \\
&\overset{4}{\Longleftrightarrow} \\
&= 1 - P[\mathcal{A} \cap \overline{\mathcal{B}}] - P[\overline{\mathcal{A}} \cap \overline{\mathcal{B}}] + P[\mathcal{A} \cap \overline{\mathcal{B}}] \\
&\overset{5}{\Longleftrightarrow} \\
&= 1 - P[\overline{\mathcal{A}} \cap \overline{\mathcal{B}}] \\
&\overset{6}{\Longleftrightarrow} \\
&= P[\mathcal{A} \cup \mathcal{B}]
\end{aligned}
$$

Which is explained as follows: in 1 we just decompose $P[\mathcal{A}]$ into $P[\mathcal{A} \cap \mathcal{B}] + P[\mathcal{A} \cap \overline{\mathcal{B}}]$ through a result we have already talked about, `prob_decomp`. Then, in 2, we cut both $P[\mathcal{A} \cap \mathcal{B}]$ since one cancels the other, and apply equation (3.7) to $P[\mathcal{B}]$ to get its complement. After that, in 3, we once again use `prob_decomp` to replace $P[\overline{\mathcal{B}}]$ with $P[\mathcal{A} \cap \overline{\mathcal{B}}] + P[\overline{\mathcal{A}} \cap \overline{\mathcal{B}}]$. 4 and 5 resumes to the use of simple arithmetic and finally in 6 we only need to resort to the application of De Morgan's laws and to the definition of complement (equations (3.5) and (3.7), respectively). It is interesting to see how this proof unfolds in COQ because we can clearly see the relation between it and the informal proof given previously:

```
Lemma prob_union : forall A B,
     \Pr_d[A :|: B] = \Pr_d[A] + \Pr_d[B] - \Pr_d[A :&: B].
Proof.
move=> A B.
rewrite (prob_decomp A B).
have Heq: forall (a b c:Qcb_fieldType), a+b+c-a = b+c.
 by move => ? ? ?; field.
rewrite Heq{Heq}.
rewrite setIC -{3}[B]setCK prob_compl.
rewrite [prob _ (~: _)](prob_decomp (~:B) A).
rewrite -setCU prob_compl.
have Heq: forall (a b c:Qcb_fieldType), a+(b-(a+(b-c))) = c.
 by move => ? ? ?; field.
rewrite Heq{Heq}.
by rewrite setUC.
Qed.
```

Step 1 corresponds to the first `rewrite` tactic, as you have probably realized by now. The two following Coq code lines and the second `rewrite` represent the arithmetic arrangement in 2, while the third one gets the complement in 4. The remaining code lines correspond to the last two steps. Lemmas `addrC`, `addrA`, `addrKr`, `setIC`, `setCl`, `setCK`, and `setUC` are already defined in SSReflect and represent some of the basic set properties, like associativity and commutativity, and some of De Morgan's laws as well.

4.1.3 Conditional Probability

If we introduce the definition:

$$x/y = z \textbf{ if and only if } x = y \cdot z,$$

we soon realize it contains ambiguities. For $y = 0$ we cannot prove there is a unique z such that $x = y \cdot z$. Suppose that $x = 1$ and $y = 0$: there is no z such that $1 = 0 \cdot z$ and any number z has the property $0 = 0 \cdot 1$. Since there is no unique solution, the result should be left undefined.

This is the mathematical explanation to why equation (3.10) is undefined when $P[\mathcal{B}] = 0$. COQ does not provide a way to handle these exceptions (there are no types for infinity or undefined) and so, we had to find an alternative solution. There were several ways to approach the problem, but only two are highlighted:

- include the property $P[\mathcal{B}] > 0$ as an argument to the definition, thus only considering cases of conditional probability where both events were defined;
- define conditional probability for all values, with $P[\mathcal{B}] = 0$. This means we would be able to evaluate all cases thus leading to an easier statement of properties and proof management;
- explicitly handle partiality with the aid of the "option" type (analogous to HASKELL's *Maybe*).

We deduced that the second option offered more advantages and decided to move on with it, which led to the following definition of conditional probability:

```
Definition condProb A :=
                  if \Pr_d[B] == 0 then
                        0
                  else \Pr_d[A :&: B] / \Pr_d[B].
```

and the respective notation:

```
Notation "\Pr_ d [ A | B ]" := (condProb d A B)
  (at level 41, d at level 0, A, B at level 53,
          format "'[' \Pr_ d '/' [ A | B ] ']'") : prob_scope.
```

We are now able to express the conditional probability of an event `A` given the occurrence of an event `B` as `\Pr_d[A | B]`.

4.2 Improving the Previous Specification

As a consequence of the previous specification of probability distribution, several problems emerged. As we will see in Chapter 6, our formalization of the logarithm relies on COQ's formalization of the natural logarithm, which is defined over the its axiomatized real numbers. For this reason, it is necessary to change the definition of probability distribution so that it may map values on the real numbers. Fortunately, the change is direct since both rational and real numbers are defined over a field (hence we are able to use the same kind of tactics to reason over the real numbers).

Furthermore, the formalization of the conditional probability appears to be a bit limited, as does the rest of the theory. Intuitively, the notion of conditional probability should indicate that the possible values that an experiment can take are restricted by the occurrence of another experiment. In such cases, it is possible that the distribution may not be defined over the entire sample space. This idea motivates the introduction of a relaxed notion of distribution: the sub-distribution, *i.e.*, a distribution just like the one we have talked in the previous section but with a single difference, the sum of all probabilities regarding the same sample space is a value between 0 and 1[4] (instead of always being equal to 1). Although this seems a good idea, that could help solve our problems, it raises a number of new issue: having two distinct definitions to characterize a distribution is not efficient as it implies the formalization of theory for both (distributions and sub-distributions) without the possibility to use the results of one in the proofs of the other, which leads to a far more complex framework than it is desirable, and so, it is necessary to find an alternative solution.

Next, we introduce and discuss the (final) changes made in order to achieve a uniform formalization of basic probability theory results.

4.2.1 Proper Distributions

Keeping it simple, and not overcomplicating things, is almost always the right thing do. Building a uniform platform to reason over probabilities is the best option to achieve an efficient framework that will actually be capable of doing something useful.

The complexity that would arise have we adopted the idea of formalizing the notions of distribution and sub-distribution would be excessive and would not provide any real advantage since the results obtained would be almost two en-

[4]to reflect the cases where the distribution may not be defined for the entire sample space

tirely different, and unrelated, sets of theory. Hence, the best scenario should only include one definition that allows to work in both a distribution and a sub-distribution context.

This clearly raises a problem: how can we work with two kinds of measures when they are not defined in comparable scales? The solution is to normalize every distribution that is conditioned by an event, thus making it possible to see them as normal distributions (*i.e.*, distributions whose sum of all probabilities equals 1). By doing so, we eliminate the need to have different definitions for different types of distributions as all of them will be understood as being of the same kind.

While developing the theory based on the new definition it became clear that it was also necessary to handle the cases where a distribution is a zero function, or in other words, a distribution whose pmf is always 0. This means that a distribution is whether a zero function or a normal distribution, which is reflected by the following structure:

```
Structure Distr := mk_Distr {
  Distr_val :> {ffun sT -> R};
  _ : ffun_ge0 Distr_val &&
        (is_ffun0 Distr_val || (ffun_sum setT Distr_val == 1))
}.
```

Comparing to the structure introduced in section 4.1.1, there are some changes to be noted. `Distr_val` now coerces to a function `{ffun sT -> R}`, which maps an event `sT` to its probability (here represented as a real number `R` instead of a rational). A few functions are also introduced:

```
Variable fT: finType.

Definition ffun_ge0 (f:{ffun fT->R}) := forallb a, 0 <= f a.
Definition ffun_sum (X: {set fT}) (f: {ffun fT->R})
                         := \sum_(x \in X) f x
Definition ffun0 : {ffun fT -> R} := finfun (fun _ => 0).
Definition is_ffun0 (f: {ffun fT -> R}) := (f == ffun0).
```

Handling all the properties/proofs related to distributions would sometimes end up making the code complex and hardly readable if we always had to write the definition of sum, distribution, *etc*. For this reason, it is necessary to refine the code in order to avoid further complications. Therefore `ffun_ge0` denotes the predicate that says whether or not the outcome of a function `f` is greater or equal than 0 and `ffun_sum` denotes the summation over a set, that we explicitly give, of the possible values of the function `f`. For instance, in the structure above we are stating that the summation iterates over the full set `setT` (*i.e.*, the entire sample space). This will be particularly useful when handling restricted distributions. The remaining two functions allow to precisely capture the notion of a zero function. `is_ffun0` has type `{ffun aT -> R} -> bool`: it returns true if a specific outcome of the function `f` is 0 or false otherwise. Note that these functions are defined at the level of the functions that underlie the definition of a distribution, which means that it is possible to prove several important properties that are necessary to handle finite functions (and sums and others) without needing to care about the distributions *per se*.

Due to the way we define a distribution, for some of the properties to be proven it is necessary to give a premise that states if the distribution in scope is a zero function or not, which may be a necessary fact that one needs to know at the time of the proof. For instance:

```
Lemma prob_compl: forall E,
      ~~ is_ffun0 d -> \Pr_d[~: E] = 1 - \Pr_d[E].


Lemma prob_decomp: forall A B,
  \Pr_d[A] = \Pr_d[A :&: B] + \Pr_d[A :&: ~:B].


Lemma prob_eq0: forall A, is_ffun0 d -> \Pr_d[A] = 0.
```

While `prob_compl` only holds when `d` is a normal distribution and `prob_eq0` when `d` is a zero function, `prob_decomp` needs no premise as it holds for all cases. For simplicity, we always state the premise as `is_ffun0` or its boolean negation `~~ is_ffun0`, which is equivalent to `ffun_sum setT d == 1` (if a distribution is not a zero function, then it must be a normal distribution). This transition between properties will be useful later, and so, a reflection lemma[5] is available:

```
Lemma Distr_sumP: forall (d: Distr),
```

[5]a reflection lemma is an equivalence property between the boolean and the `Prop` worlds

```
    reflect (~~ is_ffun0 d) (ffun_sum setT d == 1).
```

The new definition of conditional probability is somewhat different. The idea is to define it just as an ordinary probability, but with a small difference: the distribution is conditioned/restricted by an event:

```
Definition probCond (d: Distr aT) (A B: Event) :=
    prob (condDistr d B) A.
```

Further explanation on conditional distributions is available in section 6.2.2.

4.3 Application: Semantics of Security Games

"Typically, the definition of security is tied to some particular event $\mathcal{S}$. Security means that for every efficient adversary, the probability that event $\mathcal{S}$ occurs is very close to some specified target probability: typically, either 0, 1/2, or the probability of some event $\mathcal{T}$ in some other game in which the same adversary is interacting with a different challenger."

V. Shoup [Sho04]

When developing probability theory, one of our interests is to formalize security proofs of cryptographic techniques. In cryptography, a security proof can be organized as a sequence of games, which is typically defined as an "attack game" played between an adversary and a benign entity. It is not applicable to all proofs and even when it is, its only purpose is to serve as an organization tool in order to reduce their complexity. The key idea here is to allow one to transform a problem into a sequence of games, each one increasingly similar with a well known cryptographic problem, to a point where the last game is negligibly[6] close to the problem that we want to prove.

Since both adversary and benign entity are probabilistic processes that communicate with each other, it is possible to model the game as a probability space. In order to develop a Coq framework to reason about this kind of security proofs one can now understand the importance of the specifications given until now.

[6]with a difference close to 0

However, there are some answers left to be answered: what about this sequence of games, how can one make the transition between them? And how can one know when to stop? For the latter question we come up with a COQ specification that allows to capture the concept of negligibility:

```
Definition prEq epsilon := | \Pr_d1[E] - \Pr_d2[E] | <= epsilon,
```

prEq defines what negligibility intends to be in the context of cryptographic security proofs (the difference between two events is negligible if it is smaller than a certain quantity epsilon). For the former, the concept of monad has to be introduced.

Originally, the term monad came from category theory. It can be used as a way of chaining operations together to build a pipeline, allowing programmers to construct computations that simulate sequenced procedures. In functional programming languages, like COQ or HASKELL, a monad is an abstract data type constructor used to represent computations, that is defined by a type constructor and two operations: *return* and *bind*. The former is just a way to produce a value encapsulated in the monad type and the latter is the operation that actually allows to construct sequences of computations. They can be specified in COQ as follows:

```
Definition singlff a : {ffun aT -> R} :=
  [ffun x => if x==a then 1 else 0].

Definition bindff {bT} (d:Distr aT) (f:aT -> Distr bT)
  : {ffun bT -> R} := [ffun b => \sum_a (d a) * (f a b)].

Canonical Structure ret x :=
  Distr_build (singlff_ppos x) (singlff_psum x).

Canonical Structure bind {bT} x f :=
  Distr_build (@bindff_ppos bT x f) (bindff_psum x f).
```

singlff and bindff are necessary to guarantee that return and bind hold some properties after being executed (*e.g.*, the probability of a distribution ranges between 0 and 1).

At this point, we have at our disposal the tools for managing a security proof organized as a sequence of games: a COQ formalization of basic concepts from probability theory much needed to model the games as probability spaces, a way to construct the sequence of games and a way to know when the proof is completed. But note that this is not enough to actually construct the proof: one must resort to other means regarding the aspects of security analysis and cryptographic construction.

Chapter 5

Elements of Information Theory

Curiously, information theory is one of the few scientific fields to have an identifiable beginning. In [Sha48], C. Shannon presented the world with a theoretical paper that would eventually reshape the perception of all scientific community about communications. The idea that it would be possible to quantify, and even reduce to a mathematical formula, an abstract concept like information attracted enormous attention [ACK$^+$01]. This led to the possibility to encode, in bits, essentially every kind of communication which represented the starting point of the Digital Age [ACK$^+$01].

Despite acknowledging the work done until then, it was Shannon's vision and ideas that originated an infinity of possibilities and fostered an exponential growth of communication research, which led to the field of information theory we nowadays know. His huge contribution allowed to discover fundamental limits on signal processing operations and to answer two extremely important questions in communication theory: what is the ultimate data compression (the entropy), and what is the ultimate transmission rate of communication (the channel capacity). Since then, Shannon's work has blossomed to make a direct impact in a wide variety of fields, as illustrated in Figure 5.1, such as computer science (Kolmogorov complexity), communication theory (limits of communication theory), probability theory (limit theorems and large deviations) and many others [CT06].

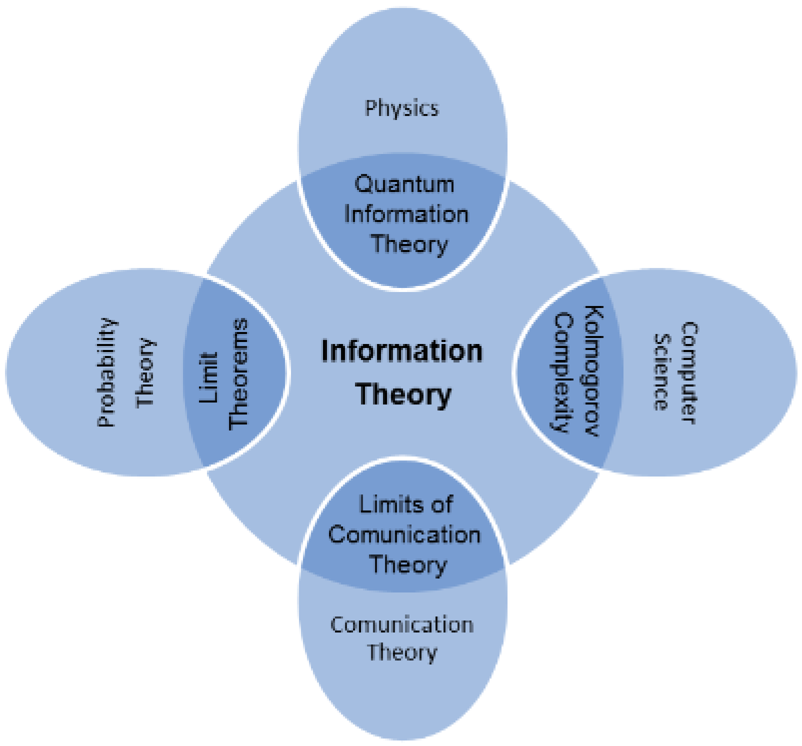

Figure 5.1: Relationship of information theory to other fields.

"Bell Labs were working on secrecy systems. I'd work on communication systems and I was appointed to some of the committees studying cryptanalytic techniques. The work on both the mathematical theory of communications and the cryptography went forward concurrently from about 1941. I worked on both of them together and I had some of the ideas while working on the other. I wouldn't say that one came before the other - they were so close together you couldn't separate them"

C. Shannon [Kah96]

Information theory and cryptography share a special bound. In fact, the work from the former is very much presented in terms of the work of the latter [CLaPS00]. For a long time information theory has been used to provide lower bounds on the size of the secret key required to attain desired levels of security in specific systems [Mau93]. Therefore, information theory plays a central role on deriving results on the provable security properties of a cryptographic system.

This chapter introduces the important concept of entropy, which is then extended to define mutual information. As we will see, these quantities are functions of the probability distributions that underlie the process of communication and maintain a close relation by sharing a number of simple properties, some of which are presented in this chapter.

5.1 Entropy

Entropy was first defined by C. Shannon, in his 1948 paper [Sha48], as a measure of the uncertainty associated with a random variable (or in other words, as a quantification of the expected value of a random variable) and it is perhaps the most important concept in information theory.

Suppose X is a random variable that describes the experiment where one tosses a coin and checks the result: if the coin is fair (*i.e.*, both sides of the coin have equal probability of $\frac{1}{2}$) then the entropy of the result reaches its maximum value, since it is the situation with higher uncertainty; otherwise, if both faces of the coin have different probabilities, we know that each coin toss will bring less uncertainty, since one side will have a higher probability to show up, and therefore the entropy value will drop. There will be no entropy if the experiment does not offer any uncertainty (*e.g.*, if the coin is double-headed[1] the result will always be heads) thus making the result trivially predictable.

Information is a complex concept and thus hard to define, but entropy does a pretty good job. It comprises a set of properties that resembles the general idea of what a measure of information should be [Sch95]. Formally, for any probability distribution, the amount of information in X is given by:

$$H(X) = -\sum_{x \in X} p(x) \cdot \log p(x), \tag{5.1}$$

where $H(X)$ denotes the entropy of X with pmf $p(x) = P[X = x]$, for all $x \in X$. This is not the only way to define entropy but it is probably the best for our purposes. Note that the entropy of a random variable only depends on its probability

[1] both sides of the coin are heads

distribution (and not on the actual values that the random variable takes). We use the binary logarithm (to the base 2) in order to measure the entropy in bits ($H(X)$ then becomes the average number of bits necessary to describe X). Throughout the rest of this document the base is omitted but every time a logarithm appears it should be understood as the binary logarithm. Any change toward the equalization of the probabilities increases the entropy (as it increases uncertainty). Since the value of $p(x)$ ranges between 0 and 1, we can immediately derive the following property:

$$H(X) \geq 0, \tag{5.2}$$

stating that the entropy of a random variable is always greater than 0 (with equality when the random variable is deterministic,*i.e.*, it is certain to be predicted). Actually, entropy is a concave function, which makes sense since, as stated before, it takes its minimum value when $p(x) = 0$ and $p(x) = 1$ (there is no uncertainty associated) and its maximum when $p(x) = \frac{1}{2}$ ($p(x)$ and $p(\overline{x})$ are equiprobable). Additionally, if A_X is the alphabet of X (*i.e.*, the set of all possible outcomes of X), then:

$$H(X) \leq \log |A_X| \tag{5.3}$$

represents the upper bound on the entropy of X, with equality if, and only if, X is distributed uniformly over A_X.

It is possible to change the base of the logarithm in the definition using the rule:

$$H_b(X) = (\log_b a) \cdot H_a(X). \tag{5.4}$$

Note that changing the base of the logarithm is not important because it only results in information measures which are just constant multiples of each other.

5.1.1 Joint Entropy

Equation (5.1) may be extended to define joint entropy: the entropy of a group of random variables. Once again, we will only consider the cases that just concern two random variables:

$$H(X,Y) = -\sum_{x \in X} \sum_{y \in Y} p(x,y) \cdot \log p(x,y). \tag{5.5}$$

It is obvious that not much needs to be changed. Since (X,Y) can be considered a single vector-valued random variable [CT06], we define joint entropy, from the definition of regular entropy, just by swapping the regular distribution for the joint distribution concerning both random variables in scope. As a measure of uncertainty, joint entropy always takes a non-negative value (similar reasoning as in the entropy case) and its content is at most the sum of the individual entropies of both random variables:

$$H(X,Y) \leq H(X) + H(Y), \tag{5.6}$$

with equality if, and only if, X and Y are independent (*i.e.*, $p(x,y) = p(x) \cdot p(y)$). Intuitively, if we observe two experiments the information that we learn is the sum of the information of each one. However, joint entropy is always greater or equal than each individual entropy:

$$H(X) \leq H(X,Y), \tag{5.7a}$$
$$H(Y) \leq H(X,Y). \tag{5.7b}$$

5.1.2 Conditional Entropy

Similarly to what happened with joint entropy, we can extend the definition of entropy to measure the uncertainty regarding the conditional distribution of two random variables:

$$H(X|Y) = -\sum_{x\in X}\sum_{y\in Y} p(x,y)\cdot logp(x|y). \tag{5.8}$$

Note that (5.8) refers to the average entropy of X conditional on the value of Y, averaged over all possible outcomes of Y. This is different than conditioning on Y taking one particular value (which is what happens in conditional distributions).

From (5.2) and since conditioning always reduce entropy (it makes sense that adding information will most probably reduce the uncertainty) it follows that:

$$0 \leq H(X|Y) \leq H(X), \tag{5.9}$$

with equality on the left side if, and only if, Y uniquely determines X and equality on the right side if, and only if, X and Y are independent.

Finally, we show a fundamental rule for transforming uncertainties:

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y), \tag{5.10}$$

which is known as the chain rule. It relates joint to conditional entropies by stating that the total uncertainty about the value of X and Y is equal to the uncertainty about X plus the (average) uncertainty about Y once we know X. Intuitively, if we have a message that is described in terms of both X and Y, then it has $H(X,Y)$ bits of information. If we learn X, we get $H(X)$ bits of information thus only remaining $H(Y|X)$ bits of uncertainty about the message.

5.2 Mutual Information

Mutual information is the reduction in the uncertainty of one random variable due to the knowledge of the other, or in other words, it measures the amount of information that two random variables share. For instance, suppose X represents the experiment where one flips a coin, and Y represents whether the result was heads or tails. Clearly, X and Y are related as each outcome provides information about the other, *i.e.*, they share mutual information. However, if X represents the experiment where one flips a coin and Y represents the flip of another coin, then

X and Y do not share mutual information.

Like entropy, mutual information is always a non-negative value and represents the relative entropy between the joint distribution and the product distribution [CT06]. Formally, it may be defined as:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \cdot log \frac{p(x,y)}{p(x) \cdot p(y)}, \tag{5.11}$$

If both random variables are independent (5.11) will be 0 (this is intuitive because if they are independent there will be no shared information whatsoever).

Mutual information shares a lot of important properties with entropy. It is symmetric[2] (*i.e.*, X tells as much about Y as Y about X) and it may be seen as the reduction in the uncertainty of X due to the knowledge of Y. Therefore, we can equivalently express mutual information as:

$$I(X;Y) = H(Y) - H(Y|X) = H(X) - H(X|Y) = I(Y;X), \tag{5.12}$$

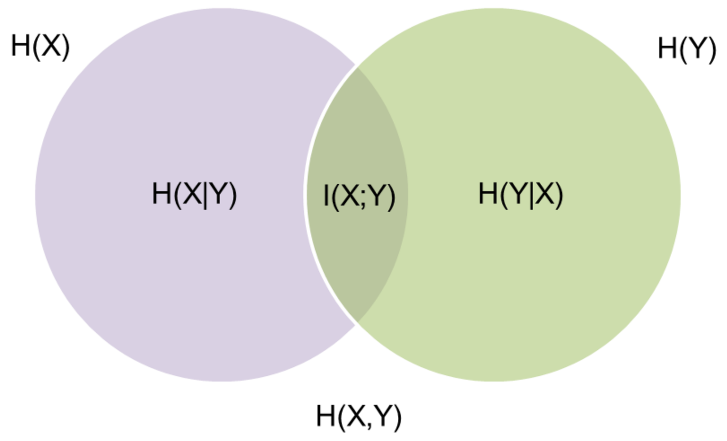

Figure 5.2: Graphical representation of the relation between entropy and mutual information.

In some sense mutual information is the intersection between the entropies of

[2]which exhibited by the fact that joint distributions are symmetric

both random variables, since it represents their statistical dependence (see Figure 5.2). From equations (5.12) and (5.10) it follows that:

$$I(X;Y) = H(X) + H(Y) - H(X,Y). \tag{5.13}$$

Sometimes entropy is referred to as self-information. This happens because the mutual information of a random variable with itself is the entropy of the random variable (since both random variables are perfectly correlated, they share exactly the same information):

$$I(X;X) = H(X) - H(X|X) = H(X). \tag{5.14}$$

Chapter 6

Entropy in Coq

Chapter 4 gave insight on how we implemented finite discrete probability distributions in COQ. At the time, the better option seemed to be approaching the problem in a simplistic way and model the probability of an event as a function that would map a specific event in the sample space to its probability of occurring. If in one hand this allowed to keep things clean and "simple", as that was all we needed to handle probabilities, in the other it turned out to be a limiting factor to the development of the framework.

To go forward into the formalization of entropy it was missing some theory regarding random variables. It then became necessary to extend the framework to include the missing theory: more specifically, the framework was upgraded in such a way that it could reflect the relation between the sample space, a specific random variable and the distribution that characterized it towards the sample space. This implied the formalization of theory introduced in section 3.2 (regular, joint and conditional distributions regarding discrete random variables), but also the formalization of other important concepts that were needed to define entropy and mutual information (*e.g.*, logarithm to the base 2).

As you shall see, it was possible to reuse much of the work done previously to define a proper kind of distribution (now based on random variables). In this context, the most important SSREFLECT libraries continued to be the *finset* and the *bigop*, but also the standard COQ library *Rpower*[1], which offered a formalization of the natural logarithm (sometimes denoted as ln and others as $\log_e$) over the reals

[1]`http://coq.inria.fr/stdlib/Coq.Reals.Rpower.html`

and served as the basis for the development of the theory related to logarithms.

This Chapter begins by introducing and explaining the main steps taken to capture, in COQ, the notion of random variable. It is afterwards used to redefine the previous idea of a discrete distribution in such a way that it may reflect the relation between the sample space, a random variable and a distribution. The formalization of joint and conditional distributions is also highlighted. We then focus on the logarithm to the base 2 in order to define the important notion of entropy (joint and conditional entropy as well) and mutual information. The Chapter closes with a simple case study.

6.1 Random Variables

Obviously, without the formalization of random variables we would not be able to proceed into the formalization of distributions over random variables, and so, that is the first thing to do. Roughly speaking, a random variable is the value of measurement associated with a random experiment. Formally it is characterized by a measurable function from a probability space to some kind of set. In COQ it could be something like:

```
Definition RVar {sT} (d: Distr sT) (aT: finType) := sT -> aT.
```

A random variable is thus coded as function and takes 3 arguments: an implicit[2] finite type `sT` representing the sample space, a distribution `d` on `sT` that characterizes the random variable `RVar` and a finite type `aT` for the set of outcomes. It is necessary to make the second and third arguments (the distribution and the outcome set, respectively) explicit so we may be able to define different random variables on the same probability space, which is later needed in order to work with joint and conditional distributions.

The structure defined in section 4.2 did not represent a "true" discrete probability distribution since it lacked any reference to random variables, but we are now in a good position to correct things and do it the proper way. The next section gives insight on the work done toward that goal: more precisely, the formalization of the concepts of regular, joint and conditional entropies.

[2]in COQ the user can make an argument implicit by putting its name inside braces

6.2 Discrete Random Variable Based Distributions

Recall that we defined a distribution as a structure that included a function mapping an event to its probability and one property stating that every probability is a non-negative value and that the distribution is one of two things: a zero function or a normal distribution, *i.e.*, the sum of all probabilities, regarding the same sample space, equals 1.

We now give the definition of a distribution over a random variable (note that the actual definition of the distribution remains the same but it is now used to define a new kind of distribution that maps values of the random variable, instead of the sample space, into their probabilities). The first step is to define the pmf:

```
Definition pmf_rvar {sT aT} {d: Distr sT} (X: RVar d aT)
   : {ffun aT -> R} := [ffun x : aT => \Pr_d[X @^-1: pred1 x]].
```

pmf_rvar is a finite function from a type aT (recall that X is a random variable with type {sT -> aT}) to a real number R, and so, it maps a particular value of the random variable to its probability. By now, the notation used should be familiar, although the set over which the sum iterates may seem a bit strange. That notation is defined in the *finset* library as: f @^-1: R, the preimage of the collective predicate R under f. In this case, X @^-1: pred1 x allows to define a predicate that checks if any outcome of the random variable X equals a given x, which is exactly what we want. Expanding \Pr_d[X @^-1: pred1 x] reveals a sum over the resulting set that computes the probability of that outcome to occur in the distribution d.

If we want the new structure to be a distribution it is also necessary to demonstrate the properties we talked about previously, but now regarding the pmf defined in pmf_rvar. First, we show that it is always a non-negative value:

```
Lemma pmf_rvar_ge0 : forall aT (X: RVar d aT),
  ffun_ge0 (pmf_rvar X).
```

which is a fairly straightforward lemma to prove, using induction. Then, it remains to demonstrate that all distributions are zero functions or normal distributions:

```
Lemma pmf_rvar_0 : forall aT (X: RVar d aT),
  is_ffun0 d -> is_ffun0 (pmf_rvar X).

Lemma pmf_rvar_sum1: forall aT (X: RVar d aT),
  ~~ is_ffun0 d -> ffun_sum setT (pmf_rvar X) == 1.
```

Note that the method introduced in section 4.2 to handle certain properties is also used in this situation (we are still managing distributions, so we need to know if they are zero functions or normal distributions). `pmf_rvar_0` is easily provable by induction but for `pmf_rvar_sum1` it gets a little trickier. `ffun_sum setT (pmf_rvar X)` is actually a summation, over the sample space, of all probabilities related to the random variable X. In fact, `pmf_rvar X` is just filtering one value (which is the probability of the event), and so, we just need to show that the summation is filtering the value that we want (since it iterates over the entire sample space, this must happen.)

Finally, it is important to make sure that Coq interprets `pmf_rvar` as a `Distr` as it will allow the use of properties related to distributions (besides being an important aspect regarding the framework's uniformity). This is achieved through the use of the canonical structures mechanism:

```
Canonical Structure pmf {aT} (X: RVar d aT) :=
  mk_Distr (pmf_rvarP X).
```

where `pmf_rvarP X` provides proof of the properties that demonstrate that `pmf_rvar` is a distribution:

```
Lemma pmf_rvarP : forall aT (X: RVar d aT),
        ffun_ge0 (pmf_rvar X) &&
 (is_ffun0 (pmf_rvar X) || (ffun_sum setT (pmf_rvar X) == 1)).
```

The proof of `pmf_rvarP` is carried by doing case analysis on `is_ffun0 d` (*i.e.*, if it is true or if it is false) and invoking the lemmas `pmf_rvar_0` and `pmf_rvar_sum1` to close the resulting goals, after splitting the boolean conjunction (`ffun_ge0 ( pmf_rvar X)` is proved by invoking `pmf_rvar_ge0`).

To check if all went well, we run the command:

```
Check (pmf X).
```

which should display the following result:

```
pmf X
     : Distr aT
```

indicating that `pmf X` is indeed a distribution (the `Check` command allows the user to verify if an expression is well-formed and learn what is its type).

6.2.1 Joint Distribution

Naturally, we want to extend the definition of distribution to include two random variables, as talked in section 3.2.1, since it will be necessary for defining several concepts related to entropy. However, we can not use `pmf_rvar` directly since `prob` (the underlying way to compute the probability of an event) is not explicitly defined to handle events as pairs. Therefore, it is first necessary to define the product of random variables:

```
Definition RVarX {sT aT bT} (d: Distr sT) (X:RVar d aT) (Y:RVar d bT)
   : RVar d (prod_finType aT bT) := fun w => (X w, Y w).
```

`RVarX` then packs the image of each element of the sample space along with each of the given variables. Note that it has type `RVar d aT -> RVar d bT -> RVar d (prod_finType aT bT)`. In COQ `prod_finType` enjoys a `finType` structure (*i.e.*, the product of two `fintype` variables is also a `fintype` variable): hence, we can make use of `pmf` and define the joint distribution of two random variables as:

```
Definition pmfProd {aT bT} (X: RVar d aT) (Y: RVar d bT) :=
   pmf (RVarX X Y).
```

Since `pmf` is expecting a random variable as a parameter and `RVarX X Y` returns something with that type, it follows that `pmfProd` is well typed. COQ already interprets `pmfProd` as a distribution (note that it is defined as a `pmf`), and so, all the basic properties needed are automatically ensured.

Once more, verifying the type of `pmfProd X Y` displays the expected result:

```
Check pmfProd X Y.
==============
pmf_prod X Y
     : Distr (prod_finType aT bT)
```

Most of the times, definitions by themselves do not allow to achieve anything relevant since one also needs the tools to handle them. Hence, there are a few properties worth mention due to their usefulness in future proofs. For instance,

```
Lemma pmfProd_sym: forall {aT bT} (X: RVar d aT) (Y: RVar d bT) x y,
   pmfProd X Y (x,y) = pmfProd Y X (y,x).
```

states the symmetry of the joint distribution. Its proof boils down to showing that the ranges of both summations are the same (using the lemma `eq_bigl`). Another example, which relates joint to marginal distributions (see equations (3.22) and (3.23)), is:

```
Lemma pmfProd_indx: forall {aT bT} (X: RVar d aT) (Y:RVar d bT) y,
   \sum_(x:aT) pmfProd X Y (x,y) = pmf Y y.

Lemma pmfProd_indy: forall {aT bT} (X: RVar d aT) (Y:RVar d bT) x,
   \sum_(y:bT) pmfProd X Y (x,y) = pmf X x.
```

where the key idea is the same as in `pmf_rvar_sum1`: the summation is only filtering one value, thus allowing to "eliminate" one of the components of the pair, which should suffice to get the desired marginal distribution.

6.2.2 Conditional Distribution

We can see conditional distributions as a special case of joint distributions, where one wants to compute the probability distribution of a random variable given that another takes a specific value. To exemplify this idea we introduce the following definition:

```
Definition RVarR {sT aT} (d:Distr sT) (X:RVar d aT) R
        : RVar (condDistr d R) aT := X.
```

`RVarR` may seem a bit strange as it appears that it is simply returning a random variable that taken as parameter. However it is doing more than that. Note that the returning type is `RVar (condDistr d R)aT`: in fact, `RVarR` returns the random variable with a slight difference, its distribution is conditioned by the set `R`. In this context, `CondDistr` is extremely important as it is the function that actually computes de conditional distribution. In its essence, it is defined as:

```
Definition cond_ffun {aT} (d: Distr aT) (E: {set aT}) :=
   norm_ffun (restr_ffun E d).
```

where `restr_ffun` is a function that conditions the values of the distribution `d` regarding the event `E` (*i.e.*, `d` is only defined for the values of `E`):

```
Definition restr_ffun (E: {set aT}) (f: {ffun aT->R}) :=
   [ffun x => if x \in E then f x else 0].
```

Conditioning distributions implies that the sum of all of their values may no longer be 1. This means that they will not be normal distributions anymore nor suitable for use when handling any distribution's theory. We may surpass this issue by normalizing such distributions in order to establish a point where all distributions are defined over the same scale:

```
Definition norm_ffun (f: {ffun aT->R}) :=
   scale_ffun (ffun_sum setT f)^-1 f.

Definition scale_ffun (s: R) (f: {ffun aT->R}) :=
   [ffun x => s * (f x)].
```

Not much needs to be said as the definitions are pretty self explanatory. The actual normalization is performed by `scale_ffun`, which is 0 if `f` is a zero function.

Now, we may define the conditional distribution simply as:

```
Definition pmfCond {sT aT bT} {d:Distr sT}
   (X: RVar d aT) (Y: RVar d bT) (y:bT)
      := pmf (RVarR X (Y @^-1: [set y])).
```

In `pmfCond` we impose `d` to be conditioned by a specific event `y` of the random variable `Y`, which precisely reflects what we were saying at the beginning of this section. Note that we define `pmfCond` as a `pmf` and thus all distributions' related properties are automatically ensured.

Finally, equations (3.28) and (3.29), which relate joint to conditional distributions, are extremely important results that we also formalize:

```
Lemma pmfProd_l1: forall {aT bT} (X: RVar d aT) (Y: RVar d bT) x y,
   pmfProd X Y (x,y) = pmfCond X Y y x * pmf Y y.
```

```
Lemma pmfProd_12: forall {aT bT} (X: RVar d aT) (Y: RVar d bT) x y,
   pmfProd X Y (x,y) = pmfCond Y X x y * pmf X x.
```

6.3 Logarithm

As you have already seen, in section 5.1, entropy is defined at the expense of the binary logarithm. The COQ standard library provides a set of theory[3] regarding the formalization of the natural logarithm, which we use in order to define the binary logarithm.

This is done by using the next rule, which allows to obtain the logarithm with any base:

$$\log_b x = \frac{\log_a x}{\log_a b}$$

for any a and b positive real numbers. Therefore, we extend the definition in *Rpower* and get the following COQ formalization of the binary logarithm:

```
Definition log2 (x: R_oFieldType) := ln x / ln 2.
```

Naturally, the library also provides a set of properties to manage the logarithm. We use them to prove the same properties for the binary logarithm, which in turn are used when handling entropy and mutual information related theory. These properties hold no matter the base we are considering. For instance, the logarithm of 1 is 0

```
Lemma log2_1 : log2 1 = 0.
```

the binary logarithm of 2 is 1

```
Lemma log2_2 : log2 2 = 1.
```

the logarithm of any value between 0 and 1 is negative

```
Lemma log2_lt0 : forall x, 0 < x < 1 -> log2 x < 0.
```

[3]included in the *Rpower* library, which is available at `http://coq.inria.fr/stdlib/Coq.Reals.Rpower.html`

the logarithm of a product is the sum of the logarithms of the numbers being multiplied

```
Lemma log2_mul : forall x y,
   (0 < x) -> (0 < y) -> log2 (x*y) = log2 x + log2 y.
```

and finally, the logarithm of a division is the difference of the logarithms of the numbers being divided

```
Lemma log2_div : forall x y,
   (0 < x) -> (0 < y) -> log2 (x/y) = log2 x - log2 y.
```

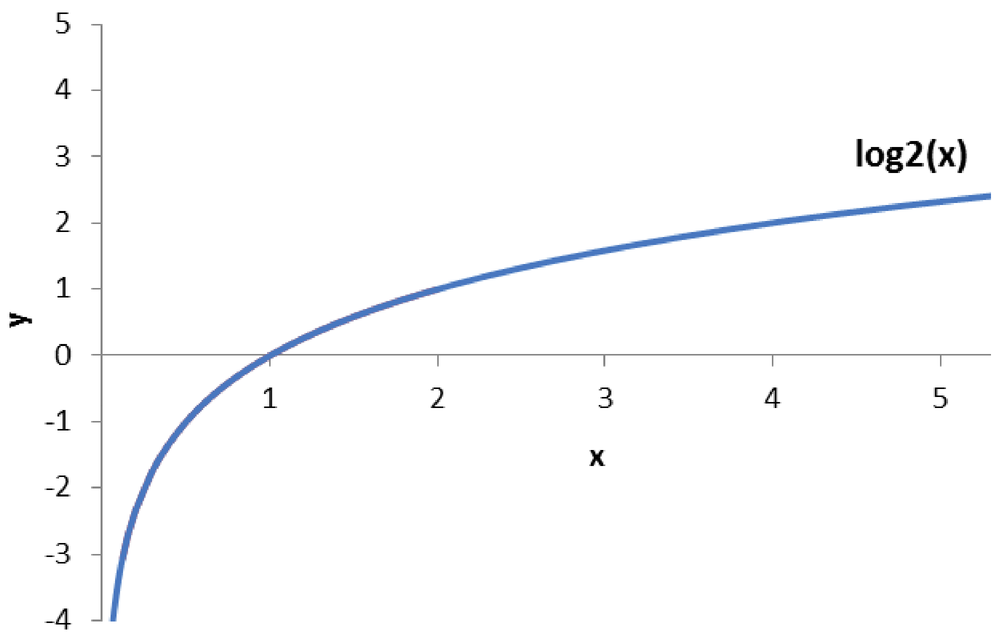

Figure 6.1: Graph of the logarithm to the base 2.

The logarithm of 0 does not exist, *i.e.*, it is undefined (and the same goes for negative numbers). This is a problematic case, mathematically explained by the fact that there is no a such that $b^a = 0$ (where b is the base of the logarithm). In fact, if we look at the graph of $\log_2 x$ (see Figure 6.1), at $x = 0$, we observe the existence of a vertical asymptote, meaning that $\log_b x \to \infty$ as we get closer to 0. The graph also helps to understand some of the above mentioned properties.

As you shall see, this will be a problem when handling entropy and mutual information related theory. The solution is introduced next, alongside with the formalization of entropy.

6.4 Entropy

The formalization of concepts such as the random variable, probability distribution and logarithm provides the needed tools to try the same with one of the most important notions in information theory. In the field of computer science, entropy may be used to quantify the amount of information associated to a random variable, which, by itself, is the perfect example of its importance, particularly in areas like cryptography.

The entropy of a random variable may take several different formal definitions. The one that is proposed next is the most suitable for being handled in the context of this framework, mainly because of all the theory related to sums, which is available via the *bigop* library (other definitions would also imply the formalization of additional concepts, *e.g.*, expectation). It follows from equation (5.1):

```
Definition entropy {aT} (X: RVar d aT) :=
  - \sum_x (pmf X x) * log2 (pmf X x).
```

It is also provided a notation

```
Notation "\H '(' X ')'" := (entropy X)
 (at level 41, X at level 53,
         format "'[' \H '/' ( X ) ']'") : prob_scope.
```

so it may be possible to manage entropy with the kind of notation that usually appears in the literature.

This definition introduces only one setback: how to deal with the cases where we have to handle the logarithm of 0? The solution is to use the convention that $0 \cdot \log 0 = 0$, which is easily justified by continuity since $x \log x \to 0$ as $x \to 0$ [CT06]:

```
Axiom log2L0 : 0 * log2 0 = 0.
```

Regarding this work, we may understand this convention as the interpretation of an event of measure zero, and consequently, no entropy.

It is possible to surpass this problem without axiomatizing $0 \cdot \log 0 = 0$ if we define, and use, the support of a function (which is the set of points where the function is not zero). In such case, the summation that computes the probability of an event shall exclude the problematic instances where the distribution is 0. Another possibility is to use the `mul0r` tactic of the *ssralg* library whenever we want to rewrite something like $0 \cdot x$ into 0, for all $x \in \mathbb{R}$. This is possible because COQ's natural logarithm is defined for all the real numbers, although it is only possible to reason over the positive ones, which enables the use of `mul0r` for every logarithm.

`log2L0` will be assumed when handling entropy and mutual information. However, we still provide a definition for the support of a function due to its possible implementation in the context of this framework:

```
Definition supp_ffun (f: {ffun aT -> R}) : {set aT} :=
     [set x | f x != 0].
```

6.4.1 Joint Entropy

Joint entropy is the entropy of a joint probability distribution. Therefore, it is very much alike regular entropy as shown in section 5.1.1 and demonstrated by the following definition:

```
Definition joint_entropy {aT bT} (X:RVar d aT) (Y:RVar d bT) :=
   - \sum_x \sum_y pmfProd X Y (x,y) * log2 (pmfProd X Y (x,y)).
```

A notation is once more provided

```
Notation "\H '(' X ',' Y ')'" := (joint_entropy X Y)
  (at level 41, X at level 53, Y at level 53,
          format "'[' \H '/' ( X , Y ) ']'") : prob_scope.
```

6.4.2 Conditional Entropy

The formalization of conditional entropy follows from the definition given in section 5.1.2 as:

```
Definition cond_entropy {aT bT} (X: RVar d aT) (Y: RVar d bT) :=
  - \sum_x \sum_y pmfProd X Y (x,y) * log2 (pmfCond X Y y x).
```

The corresponding notation is also provided

```
Notation "\H '(' X '|' Y ')'" := (cond_entropy X Y)
  (at level 41, X at level 53, Y at level 53,
          format "'[' \H '/' ( X | Y ) ']'") : prob_scope.
```

There are several relevant properties related to conditional entropy that are worth to be proven but we only highlight one. Its importance is huge as it allows to relate joint to conditional entropies. Additionally, it also plays an important role in the demonstration of several other properties related to mutual information. It is the rule stated by equation (5.10), which in COQ looks like:

```
Lemma chainR_jEnt: forall aT bT (X:RVar d aT) (Y:RVar d bT),
  \H(X,Y) = \H(X) + \H(Y|X).
```

Since the proof of `chainR_jEnt` is representative of the importance that some lemmas previously stated have, we give insight on how it may be carried. It can be mathematically proven as follows:

$$
\begin{aligned}
H(X,Y) &= -\sum_{x\in X}\sum_{y\in Y} p(x,y)\cdot\log\ p(x,y)\\
&\overset{1}{\Longleftrightarrow}\\
&= -\sum_{x\in X}\sum_{y\in Y} p(x,y)\cdot\log\ (p(x)\cdot p(y|x))\\
&\overset{2}{\Longleftrightarrow}\\
&= -\sum_{x\in X}\sum_{y\in Y} p(x,y)\cdot\log\ p(x) - \sum_{x\in X}\sum_{y\in Y} p(x,y)\cdot\log\ p(y|x)\\
&\overset{3}{\Longleftrightarrow}\\
&= -\sum_{x\in X} p(x)\cdot\log\ p(x) - \sum_{x\in X}\sum_{y\in Y} p(x,y)\cdot\log\ p(y|x)\\
&\overset{4}{\Longleftrightarrow}\\
&= H(X) + H(Y|X)
\end{aligned}
$$

In Step 1 we just need to show that $p(x,y) = p(x) \cdot p(y|x)$. Unfortunately, we can not do the substitution directly because of the way summations are defined. So, we first need to rewrite the entire body of the summation, which in COQ is done using the lemma `eq_bigr` (see section 2.3.2):

```
rewrite (eq_bigr (fun x=> \sum_y pmfProd X Y (x,y) *
                  log2 (pmfCond Y X x y * pmf X x))).
```

using this tactic will perform the transformation and originate a side goal[4] that asks to prove

```
  ============================
   \sum_y pmfProd X Y (x, y) * log2 (pmfProd X Y (x, y)) =
   \sum_y pmfProd X Y (x, y) * log2 (pmfCond Y X x y * pmf X x)
```

in order to do so, it is necessary to rewrite the body of the summation again

```
rewrite (eq_bigr (fun y=> pmfProd X Y (x, y) *
              log2 (pmfCond Y X x y * pmf X x))) //.
 by move=> y _; rewrite pmfProd_l2.
```

we are now able to prove the equality $p(x,y) = p(x) \cdot p(y|x)$ using the lemma `pmfProd_l2` (see section 6.2.1). The side goal generated by the `rewrite` tactic is trivially solved by //.

Step 2 comprises a set of operations to perform. Note that it is supposed to split the summation, which cannot be done directly using `eq_bigr`. The solution is to use `match goal`, a tactic that replaces expressions using pattern matching. We then start by doing so, thus leading to the following equality to be proven

```
  ============================
   - (\sum_x \sum_y pmfProd X Y (x, y) * log2 (pmf X x)) -
   \sum_x \sum_y pmfProd X Y (x, y) * log2 (pmfCond Y X x y) =
   - (\sum_i \sum_y pmfProd X Y (i, y) * log2 (pmfCond Y X i y * pmf
      X i))
```

in which suffices to show that $\log(p(x) \cdot p(y|x)) = \log p(x) + \log p(y|x)$ and then that $p(x,y) \cdot (\log p(x) + \log p(y|x)) = (p(x,y) \cdot \log p(x)) + (p(x,y) \cdot \log p(y|x))$,

[4]which is the equality between the previous body of the summation and the new, that we are trying to rewrite to

before splitting the summation. The first two operations are performed at the same time

```
rewrite (eq_bigr (fun x => \sum_y (pmfProd X Y (x,y) * log2 (pmf X x)
    +
                  pmfProd X Y (x,y) * log2 (pmfCond Y X x y)))); last
                      first.
```

but, like before, we need to prove the transformation *per se*. Again, we rewrite the body of the summation in order to prove the side goal that is generated by the previous `rewrite` tactic

```
rewrite (eq_bigr (fun y=> (pmfProd X Y (x, y) * log2 (pmf X x) +
              pmfProd X Y (x, y) * log2 (pmfCond Y X x y)))) //.
move=> y _; rewrite log2_mul.
 by rewrite GRing.mulr_addr GRing.addrC.
```

and end by using the lemma `log2_mul` (see section 6.3) to expand the logarithm, and also one lemma regarding multiplication's distributivity, `GRing.mulr_addr`, and another regarding addition's commutativity, `GRing.addrC`. We complete step 2 by splitting the summation (first the inner, and then the outer)

```
rewrite (eq_bigr (fun i=> ( (\sum_y (pmfProd X Y (i, y) * log2 (pmf X
    i)) +
     (\sum_y pmfProd X Y (i, y) * log2 (pmfCond Y X i y))))));
               last by move=> x _; rewrite big_split.
rewrite big_split.
```

In step 3, we make use of yet another already mentioned property. Note that if we take $\log p(x)$ out of the (first) inner summation we obtain $\sum_{y \in Y} p(x,y)$, which is the marginal probability $p(x)$. So, we rewrite the summation to reallocate $\log p(x)$ using the `big_distrl` tactic (see section 2.3.2)

```
rewrite (eq_bigr (fun x => log2 (pmf X x) * \sum_y pmfProd X Y (x,y))
   );
        last by move=> i _; rewrite GRing.mulrC -big_distrl.
```

and do another rewrite to show that $\sum_{y \in Y} p(x,y) = p(x)$, which is proved using the lemma `pmfProd_indy` (see section 6.2.1)

```
rewrite (eq_bigr (fun x => pmf X x * log2 (pmf X x)));
        last by move=> x _; rewrite pmfProd_indy GRing.mulrC.
```

Finally, step 4 involves minor proof tweaking since it is practically terminated, *i.e.*, exchanging summations using `exchange_big` and using the joint distribution symmetry property via the `pmfProd_sym` lemma (see section 2.3.2 and section 6.2.1, respectively).

6.5 Mutual Information

As you have probably noticed, we defined the three kinds of entropy similarly. It is important to do so as it shall simplify the process of managing them in order to reach the definition of mutual information. From equation (5.11), the Coq definition of mutual information follows from the definitions of regular/joint entropy and binary logarithm as:

```
Definition mutI {aT} {bT} (X: RVar d aT) (Y: RVar d bT) :=
  \sum_x \sum_y
    pmfProd X Y (x,y) * log2 (pmfProd X Y (x,y) / (pmf X x * pmf Y
       y)).
```

As usual, a specific notation is provided

```
Notation "\I '(' X ';' Y ')'" := (mutI X Y)
 (at level 41, X at level 53, Y at level 53,
        format "'[' \I '/' ( X ; Y ) ']'") : prob_scope.
```

The non-negativity of all kinds of entropy naturally follows from their definitions. However, the same does not happen with mutual information: usually, it follows from a result known as Gibbs' inequality, which states the non-negativity of relative entropy (mutual information is the relative entropy between a joint distribution and the product of its marginals).

Despite the formalization of relative entropy not being one of the purposes of this work, it is still possible to demonstrate the non-negativity of mutual information without it. The proof of Gibbs' inequality relies on a fundamental result of the natural logarithm, which we use as an axiom in our framework:

```
Axiom lnine: forall x, ln x <= x - 1.
```

We are fairly confident about `lnine` since it only concerns COQ's own definition of natural logarithm. Naturally, we extend, and prove, the result to include our definition of logarithm to the base 2:

```
Lemma log2ine: forall a, log2 a <= (a - 1) / ln 2.
```

The non-negativity of mutual information is an important theorem not only by itself, but also because it allows to derive other results such as the right hand side equation of equation (5.9): note that $0 \leq I(X;Y)$ and $I(X;Y) = H(X) - H(X|Y)$, so $0 \leq H(X) - H(X|Y) \Leftrightarrow H(X|Y) \leq H(X)$.

6.6 A Simple Case Study

We now introduce a simple case study, whose aim is to review the relation between information theory and cryptography. Therefore, and using the aforementioned concepts, we prove that for any one-time pad cipher, the entropy of the key must be greater or equal than the entropy of the message to be encrypted.

6.6.1 The One-Time Pad

"As a practical person, I've observed that one-time pads are theoretically unbreakable, but practically very weak. By contrast, conventional ciphers are theoretically breakable, but practically strong.'

S. Bellovin

The one-time pad (OTP) is a type of cipher which has been mathematically proven to be unbreakable[5] if used correctly [Sha49], and thus, there is nothing that can be used to attack the encryption using statistical analysis or pattern matching. At first, a secret key is randomly generated and used to encrypt a message (*i.e.*, ciphertext) that is then decrypted by the receiver using a matching cipher and the key. It must only be used once to encrypt messages, hence the "

[5]this holds even if the attacker possesses infinite time and computational power

one-time" in the cipher's name, and kept secret at all times of the process. The OTP provides absolute security in theory but is impracticable in real situations due to several reasons:

- the key has to be as long as the message and used only once;
- obtaining truly random sequences is computationally expensive. However, failing to ensure this property means that it is possible to get information about the message by analyzing successive ciphertexts;
- the key distribution and management is often a big problem. Moreover, using other ciphers to perform key distribution implies that the system is only as secure as the cipher that was used to transmit the keys.

6.6.2 Proving a Security Property of the OTP's Key

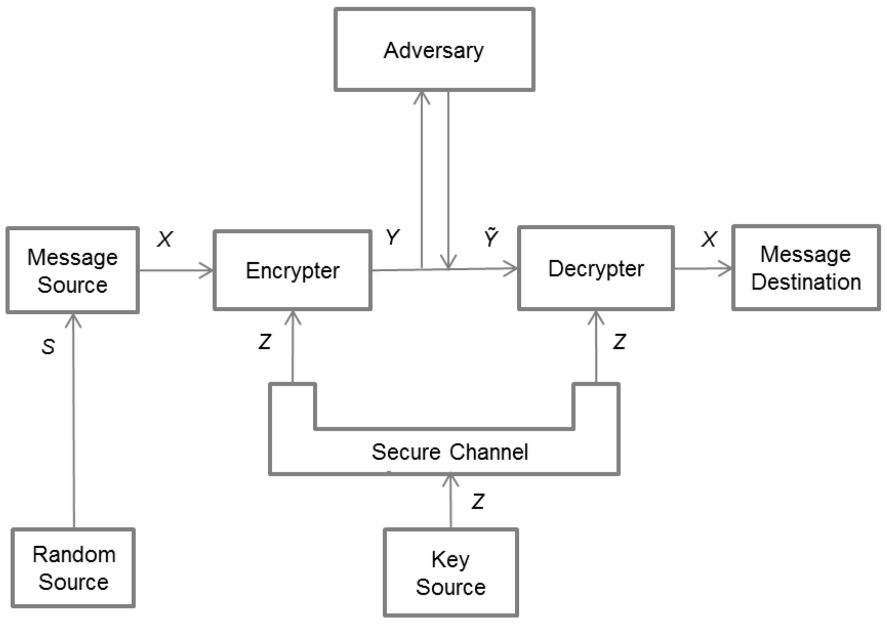

Figure 6.2: Generalization of Shannon's model of a symmetric cipher.

There are two dual and complementary security goals in communication: confidentiality (or secrecy) and authenticity[6] [Mau93]. OTP's are information theoretically secure, in the sense that even if the adversary intercepts the ciphertext, he will not be able to get any information about the message (in cryptography, this is a very strong notion of security which is sometimes called perfect secrecy). However, we do not propose to demonstrate such properties but instead, to show that, for every OTP cipher, the size of the key must be at least as large as the message to be encrypted.

Suppose that two entities communicate in the environment illustrated by Figure 6.2: it contains a secret randomizer S (which we ignore for simplicity purposes) known only to the sender of a message X, a secure channel that handles key distribution and an adversary that watches the communication channel.

Using information theoretic concepts, we may formulate the problem as follows: every cipher must be uniquely decipherable, that is, there is only one combination of cyphertext/key that gives the appropriate message:

$$H(X|Y,Z) = 0, \tag{6.1}$$

Furthermore we define perfect secrecy as the statistical independence between the message X and the ciphertext Y:

$$I(X;Y) = 0, \tag{6.2}$$

or alternatively,

$$H(X|Y) = H(X), \tag{6.3}$$

Now, showing that $H(X) \leq H(Z)$ resumes to:

[6]authenticity means that an active adversary cannot successfully insert a fraudulent message $\tilde{Y}$ that will be accepted by the receiver

$$\begin{aligned}
H(X) = H(X|Y) &\leq H(X,Z|Y) \\
&\overset{1}{\Longleftrightarrow} \\
&= H(Z|Y) + H(X|Y,Z) \\
&\overset{2}{\Longleftrightarrow} \\
&= H(Z|Y) \\
&\overset{3}{\Longleftrightarrow} \\
&\leq H(Z).
\end{aligned}$$

At the beginning, on the left hand side of the equation we use the definition of perfect secrecy (equation (6.3)) and the right hand side is explained by the fact that joint entropy has at least as much information as the individual ones (see equation (5.7a)). By using the basic expansion rule for conditional entropy we reach step 1. In step 3 we know that $H(Z|Y) + H(X|Y,Z) = H(Z|Y)$ because of equation (6.1). Finally, we know that removing knowledge can only increase uncertainty, so $H(Z|Y) \leq H(Z)$.

Chapter 7

Conclusions

We have proposed a framework for reasoning over probability and information theoretic concepts that was implemented as a library of definitions and theorems for the Coq proof assistant. This library consists of a set of files to handle the underlying theory, namely: reals and logarithm (to the base 2), finite functions, finite discrete distributions, probability theory notions, random variables based distributions and information theory notions. Furthermore we have used SSReflect, a small scale extension for the Coq system, to demonstrate all the results. Its contribute was vast, mainly due to its improved rewrite tactics, reflection features and specifications of finite functions and finite sets, which turned out to be the base for the development of this library. We have illustrated the framework's use by proving, for a one-time pad cipher, that the size (entropy) of the key must be greater than or equal to the size (entropy) of the message to be encrypted. The main advantages of using a proof assistant are that proof holes are not possible and that all the assumptions must be stated. However, we cannot simply claim that something is obvious: all aspects of the proof must be handled (even the tedious ones), which is why it is necessary to develop extensive and trustful knowledge repositories.

The framework relies on Coq's canonical structures for expressing structural and specific properties related to the probability distributions. In this perspective, the canonical structures mechanism was indeed very useful as it allowed rewriting and resolution to infer such properties automatically whenever needed.

7.1 Related Work

ALEA: a library for reasoning on randomized algorithms in COQ

The ALEA [PM10] library proposes a method for modelling probabilistic programs in the COQ proof assistant. It interprets programs as probabilistic measures, and more precisely as monadic transformations, which allows to derive important results that are necessary to estimate the probability for randomised algorithms to satisfy certain properties. ALEA has been used in larger developments, such as the *CertiCrypt*[1] framework, for the interactive development of formal proofs for computational cryptography.

The probability theory part of the library is divided in two files:

- *Probas*: includes the definition of a dependent type A for distributions on a given type. Such distributions are records that contain a function with type $(A \rightarrow [0,1]) \xrightarrow{m} [0,1]$ and proofs that this function enjoys the stability properties of measures. In this library, the interval $[0,1]$ corresponds to a pre-determined type that is axiomatized (operations on this type are also axiomatized);
- *SProbas*: includes the definition of a relaxed notion of distributions, called sub-distributions, suitable to model programs using both non-deterministic and random choice.

This work is limited to discrete distributions to ensure monadic transformation to interpret properly programs as mathematical measures [APM09]. As the main focus is put on the computation of programs, the monadic approach comes in handy. However such approach increases the library's complexity and hampers its use.

David Nowak's toolbox

In [Now07], David Nowak proposes a certification toolbox[2] for cryptographic algorithms, which is built on top of the COQ proof assistant. The toolbox is divided

[1] `http://certicrypt.gforge.inria.fr/`

[2] source code is available at `http://staff.aist.go.jp/david.nowak/toolbox/coqindex.html`

in two layers: the first includes various extensions of the standard library of COQ (*e.g.*, the formalization of probability distributions) and the second includes theory to handle security proofs.

The proofs of cryptographic protocols are seen as a sequence of games which in turn are interpreted as probabilistic algorithms. The probabilistic nature of the framework is formalized with recourse to the definition of a distribution monad while games are defined as functions returning a distribution. Moreover, probabilities are computed by deciding whether a predicate is true or false for each value of the distribution.

Unlike what happens in the ALEA library, the definition of a distribution is kept as simple as possible: it is implemented as a list of pairs that associate a value to its corresponding weight, a real number, in the distribution. However, Nowak's toolbox lacks the rewriting features that SSREFLECT adds to COQ, which implies a number of difficulties when rewriting equalities between games.

Information theory formalizations in HOL

If we consider formalizations outside COQ's context, the range of options greatly increase. In this context, we highlight [MHeT11] and [Cob08] due to their focus on the formalization of some entropy notions. The former presents a formalization in HOL of measures, Lebesgue integration and probability theories (based on the set of extended real numbers $\overline{\mathbb{R}}$[3]), which then are used to formalize the notions of entropy and relative entropy. The latter provides an information theoretic technique for proving information leakage of programs formalized in the HOL4 theorem-prover, which is based on the use of information theoretic notions such as regular/conditional entropy and regular/conditional mutual information.

7.2 Future Work

Currently, the framework includes the formalization of basic concepts of probability and information theory. The next step should be to continue along the

[3]the set of real numbers $\mathbb{R}$ extended with two additional elements, namely, the positive infinity $+\infty$ and the negative infinity $-\infty$

same line and add more results to the library, such as the Bayes' theorem, the total probability law or more properties regarding entropy. To smooth the proof process, we may redefine summations to only iterate over their range's support, which will dramatically decrease the cases that we need to evaluate at proof time.

SSREFLECT's finite functions seem to be a bit restrictive as they do not allow to formalize potentially interesting notions (*e.g.*, $\mathbb{R}$-valued random variables and distributions over lists of any type) even though our theory is perfectly legit to comprise such cases. With a finite support it is still possible to use summations: in this context, the only thing that needs to change is the starting point regarding our finite sets, instead of relying on SSREFLECT's *finset* library, we can use one that provides the formalization of finite sets over (potentially) infinite types.

Bibliography

[ACK$^+$01] O. Aftab, P. Cheung, A. Kim, S. Thakkar, and N. Yeddanapudi. Information theory and the digital age. *Final paper of MIT Course "The Structure of Engineering Revolutions"*, 2001.

[APM09] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in coq. *Sci. Comput. Program.*, 74:568–589, June 2009.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[Ber06] Yves Bertot. Coq in a hurry. *CoRR*, abs/cs/0603118, 2006.

[BGOBP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 86–101, Berlin, Heidelberg, 2008. Springer-Verlag.

[BT05] Gilles Barthe and Sabrina Tarento. A machine-checked formalization of the random oracle model. In *in Proceedings of TYPES?04, Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[CAA$^+$86] Robert L. Constable, Stuart F. Allen, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, Scott F. Smith, James T. Sasaki, and S. F. Smith. Implementing mathematics with the nuprl proof development system, 1986.

[CLaPS00] Eugene Chiu, Jocelyn Lin, Brok Mcferron andNoshirwan Petigara, and Satwiksai Seshas. Mathematical theory of claude shannon. *Final paper of MIT Course "The Structure of Engineering Revolutions"*, 2000.

[Cob08] Aaron R. Coble. Formalized information-theoretic proofs of privacy using the hol4 theorem-prover. In *Privacy Enhancing Technologies*, pages 77–98, 2008.

[CT06] T. M. Cover and Joy A. Thomas. *Elements of Information Theory 2nd Edition*. Wiley-Interscience, 2006.

[Fri97] Bert Fristedt. *A Modern Approach to Probability Theory (Systems & Control)*. Birkhauser, 1997.

[Geu09] H Geuvers. Proof assistants: History, ideas and future, 2009.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Allemagne, 2009. Springer.

[GMT08] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[Gon05] Georges Gonthier. A computer-checked proof of the four colour theorem. 2005.

[Jay58] E. T. Jaynes. Probability theory in science and engineering. 1958.

[Kah96] David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996.

[Mah06] Assia Mahboubi. Programming and certifying a cad algorithm in the coq system. In Thierry Coquand, Henri Lombardi, and Marie-Franccoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum f"ur Informatik (IBFI), Schloss Dagstuhl, Germany.

[Mau93] Ueli M. Maurer. The role of information theory in cryptography. In *IN FOURTH IMA CONFERENCE ON CRYPTOGRAPHY AND CODING*, pages 49–71, 1993.

[MD00] Arian Maleki and Tom Do. Review of probability theory. *CS 229*, 2(1):1–12, 2000.

[MHeT11] Tarek Mhamdi, Osman Hasan, and Sofiène Tahar. Formalization of entropy measures in hol. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, volume 6898 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011.

[Now07] David Nowak. A framework for game-based security proofs. Cryptology ePrint Archive, Report 2007/199, 2007. `http://eprint.iacr.org/`.

[PM10] Christine Paulin-Mohring. A library for reasoning on randomized algorithms in Coq - version 5. Description of a Coq contribution, Université Paris Sud, November 2010. `http://www.lri.fr/~paulin/ALEA/alea-v5.pdf`.

[Ros09a] Jason Rosenhouse. *The Monty Hall Problem: The Remarkable Story of Math's Most Contentious Brain Teaser*. Oxford University Press, USA, 2009.

[Ros09b] Sheldon Ross. *First Course in Probability, A (8th Edition)*. Prentice Hall, 2009.

[Sch95] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd Edition*. Wiley, 1995.

[SDHH98] Mehran Sahami, Susan Dumais, David Heckerman, and Eric Horvitz. A bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*, Madison, Wisconsin, 1998. AAAI Technical Report WS-98-05.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27:379–423, July 1948.

[Sha49] Claude E. Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal*, 28:656–715, 1949.

[Sho04] V. Shoup. Sequences of games: a tool for taming complexity in security proofs. *Manuscript*, 2004.

[Sho05] Victor Shoup. *A computational introduction to number theory and algebra*. Cambridge University Press, New York, NY, USA, 2005.

[Teaa] Development Team. The coq proof assistant. Contact: `http://coq.inria.fr/`.

[Teab] Development Team. Ssreflect. `http://www.msr-inria.inria.fr/Projects/math-components`.

[The06] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.1*, October 2006.