Formalisation and Proof

Robert Clarisó

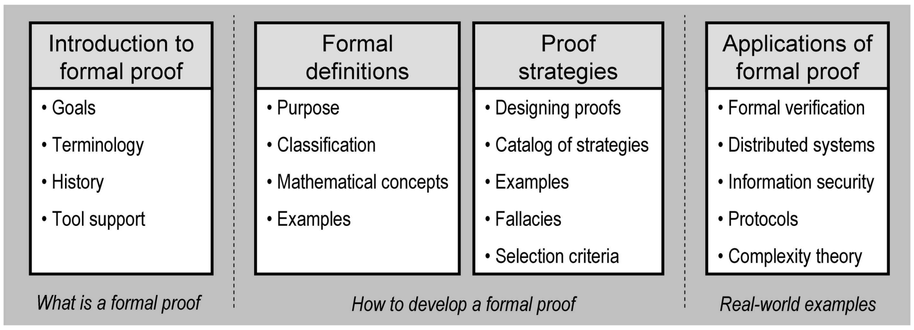
PID_00155542

Table of contents

Introduction

This document acts as a guide for the seminar “Formalisation and proof”. This seminar presents formal proof, an approach to research with application to many research problems in several disciplines such as information technology, information systems and computing. The contents of the course are divided into the four blocks depicted in Figure 1:

Figure 1. Course design.



- The first part of the course is an introduction to formal proof as a research strategy, presenting its goals, the vocabulary and structure of a proof, the historical development of formal proof in mathematics and computing and the use of software tools such as proof assistants.

- The second block introduces formal definitions as a mechanism to assign a precise meaning to concepts. Before starting an argument, it is necessary to understand the concepts being discussed. Mathematical notions such as sets or sequences can be used to describe concepts unambiguously, and notations like pseudocode can be used to formalise an algorithm in an abstract way.

- The third block constitutes the core of the course: an analysis of different strategies that can be used to develop proofs. In this section, each strategy is studied in detail, considering the situations in which its use is recommended, the directives for applying it correctly and the common pitfalls in its application. This knowledge will allow us to write our own proofs, to improve our understanding of proofs written by others and to identify errors within a proof.

- The final block is a study of several specific research fields where formal proof can be applied. The study is conducted through the review of real research papers which rely on formal proofs.

Prerequisites

All materials of the course (including this guide and all assessment activities) are written in English. A good understanding of written technical English is therefore a necessary requirement for all students.

Students of this course should also have a good understanding of **fundamental notions of propositional and first-order logic**. Concepts like truth tables, tautology and contradiction, logical connectives ($\wedge$, $\vee$,$\neg$, $\rightarrow$, $\leftrightarrow$), quantifiers and deduction rules will be used throughout the course. Although some of these concepts will be revised in the materials, it is recommended to refresh these concepts beforehand. A sample reference which reviews these topics is:

W. Hodges (1991) *Logic: an introduction to elementary logic*. Penguin books.

The example proofs revised in the paper use **basic notions of algebra and combinatorics**. Again, the necessary properties are introduced as needed but a previous background will be helpful to the reader.

Course materials

The course uses several additional materials besides this guide: the book *The Nuts and Bolts of Proofs*; the module *Formal Proof*; and several research papers where formal proof is used. The book is available in print while the remaining materials can be accessed from the classroom at the UOC virtual campus.

Although there are commercial proprietary programs for automating proofs, most available software is freely available for academic or research purposes. Activities that require the use of a specific software will provide pointers for downloading it. However, most activities within the course will consist of pen-and-paper proofs.

This guide references the required materials in each section and provides detailed reading guides and lists of exercises.

Methodology

At the start of the course, the instructor will propose a **schedule** for the study of the contents and the assessment activities. Each section provides a list of **reading materials** and **self-assessment exercises**, which should be performed by the student in the order suggested by this guide. Any questions about the reading materials or the exercises should be posted to the **forum of the virtual classroom**, where doubts and their answers can be shared with your peers.

Some proofs may require previous knowledge or reuse previous results proved elsewhere. If you do not have enough information to understand the proof, it is recommended to assume an **active attitude** and follow the thread: look up the references of the paper and/or locate papers which introduce these prerequisites. Before asking a question on the prerequisites of a proof, try to find an answer yourself and then, if you need to, ask for clarification or confirmation.

On the other hand, it is always a good idea to request clarifications on **proof development**. If you do not understand why a conclusion has been reached or whether an alternative is possible, feel free to ask your questions in the forum.

Regarding self-assessment exercises, the first block proposes general exercises on locating resources or understanding proofs. Meanwhile, the second and third blocks on definitions and proof strategies consider hands-on exercises that will require writing short proofs and definitions for sample problems. Finally, the exercises on the third block on the application of formal proof consist of the comprehension and critical review of real proofs.

Goals

1. Understand the goals and history of formal proof as a research strategy.
2. Learn the difference between computer-checked proofs and pen-and-paper proofs and discover the tool-support possibilities for computer checked proofs.
3. Become familiar with mathematical definitions, notation and concepts: pair, tuple, set, function, . . .
4. Know how to formalise a real-world research problem.
5. Learn different proof strategies, how to apply them in practice and how to choose the most suitable one for each problem.
6. Understand the value of formal proofs used in different research fields.
7. Be able to understand proofs, identify errors and correct them.
8. Write formal proofs in a variety of contexts.

1. Introduction to formal proof

1.1. Overview

A *formal proof* is an argument that establishes the validity of a statement using rigorous deduction. The purpose of a proof is to *certify* a result or property with absolute certainty. This degree of confidence is achieved through the use of (a) a precise notation based on mathematical concepts (*which avoids ambiguity*) and (b) logic inference rules which establish how to derive valid consequences from a set of premises (*which avoid errors in the deduction process*).

Before moving forward, let us consider a simple example of a formal proof:

Theorem 1. Let a and b be two natural numbers. If their sum $a+b$ is odd, then a and b must be different ($a \neq b$).
Proof (by contraposition). We will prove instead the contrapositive statement, which is equivalent to the original: if two numbers a and b are equal, then their sum is an even number. If $a = b$ then $a+b$ can be rewritten in the following way: $a+b = a+a = 2a$ which is clearly an even number. This proves the contrapositive. □

Contrapositive

The concept of contrapositive and its use in formal proof will be discussed in depth in Section 3 on proof strategies.

This proof considers a mathematical statement about the addition of natural numbers. It requires previous knowledge of mathematical concepts (legal manipulations of $a+b$, the definitions of “natural number”, “odd” and “even”) and logical concepts (the idea of “contrapositive”). However, any reader which this knowledge can use the proof as a tool to confirm the validity of the property and, therefore, use this property as the basis to discover new mathematical facts. Furthermore, the formal proof paradigm is not restricted to the context of mathematics and it has many applications in computing research.

In this section, we present an overview of formal proof. The goal is not showing *how to prove* X but answering questions like *“Is a proof for X required?”* or *“What is considered a valid proof of X?”*.

1.2. Required reading

Goals and fundamentals

[CLA09] **R. Clarisó (2009).** *Formal Proofs: Understanding, writing and evaluating proofs*. Editorial UOC.

[DAW06] **J. W. Dawson (2006).** *Why Do Mathematicians Re-prove Theorems?*. Philosophia Mathematica 14(3):269-286, Oxford Journals.

[HA08A] **T. C. Hales (2008).** *Formal Proof*. Notices of the AMS, 55(11):1370–1381, AMS.

[HA08B] **T. C. Hales (2008).** *Formal Proof: Theory and Practice*. Notices of the AMS, 55(11):1395–1406, AMS.

History of formal proofs

[KLE91] **I. Kleiner (1991).** *Rigor and Proof in Mathematics: A Historical Perspective*. Mathematics Magazine, 64(5):291–314, AMS.

[MAC95] **D. MacKenzie (1995).** *The Automation of Proof: A Historical and Sociological Exploration*. IEEE Annals of the History of Computing, 17(3):7–29, IEEE CS Press.

Tool support for formal proofs

[BUN99] **A. Bundy (1999).** *A Survey of Automated Reasoning*. Artificial Intelligence Today, LNCS 1600, pp.153–174, Springer.

[GEU09] **H. Geuvers (2009).** *Proof assistants: History, ideas and future*. Sadhana, 34(1):3–25, Indian Academy of Sciences.

- [CLA09], [DAW06], [HA08A] and [HA08B] are general references that provide an overview of formal proof and its goal: providing certainty about the validity of a statement.

- [KLE91] and [MAC95] illustrate the history and evolution of formal proofs, tracing its roots back to mathematics, logics and the origins of computing. This history is closely related to profound questions on the philosophy of science, like “what is knowledge?”, “what is truth’?” or “how can I be sure that something is true?”. Also, it reveals different trends in the construction of formal proofs, such as the acceptance of machine-checked proofs as valid proofs.

- [BUN99] and [GEU09] survey the tool-support for mechanized proofs. As this field is very active, the papers have been selected to provide a big picture of the area, rather than providing an up-to-date survey with the latest tools and developments.

1.3. Reading tips

- These papers constitute the introduction to the subject. At this point, it is better not to waste too much time trying to understand the specific details of each proof. Later sections provide additional contents that will improve our understanding of such proofs. It is a good idea to revisit any proof that you did not understand during the block on proof strategies.

- Some famous proofs referenced from the papers are extremely long, involved and require an extensive amount of previous knowledge to be understood. In case you are interested in checking a proof like that of Fermat's Last Theorem, be aware of its complexity! Some of these complex proofs provide a proof outline for the layman which is sufficient at this point. On the other hand, other famous proofs like the existence of an infinite number of primes are simple and constitute an interesting read.

1.4. Detailed exercise and reading plan

- Sections 1, 2 and 3 of the module [CLA09].
  Reading: Section 1 (*Defining formal proof*) presents the goals of formal proof and illustrates the types of problems where it can be applied. In Section 2 (*Anatomy of a formal proof*), the overall structure of a formal proof and the terminology used to designate its elements are presented. Finally, Section 3 (*Tool-supported proofs*) discusses the different types of software tools available that can automate all or part of the deduction process.
  Exercises:
  1. Using a web search engine, find three published research papers where the term "proof" is used in the title without meaning "formal proof", *e.g.* as in experimental or empirical proofs. State the research field of the paper, the intended meaning of the term "proof" and its relation to a formal proof.
  2. Collect a list of open problems in the area of research of your interest. For each problem, describe the problem statement and locate the reference where it was originally described.
  3. Using a web search engine, locate a formal proof for each of the following problems: (a) the Königsberg Bridge Problem, (b) the *N*-Queens problem and (c) Fermat's Little Theorem.
  4. Identify the name of three journals, three conferences and three workshops which have published research papers using formal proof during the last two years. Start the search within the research field of your interest.

- The paper [DAW06].
  Exercises:
  1. Using a web search engine, find a published research paper which reproves a previous result and is not referenced in [DAW06]. You can use a search term like "new proof" or "reproof". Identify the research field and the contributions of the new proof with respect to the old one.
  2. Using a web search engine, find the description of an error in a proof published in a conference or journal research paper. You can use search terms like "erratum", "errata" and "proof". Identify the research field and the type of problem being detected.

- The papers [HA08A] and [HA08B].
  Reading: These papers provide general overviews on the area and also a short introduction to an interactive proof assistant (HOL Light).

- The papers [KLE91] and [MAC95].
  Reading: These papers present a historical review and should be easy to read. Again, do not worry about the specific details of all proofs (some of them require previous knowledge).
  Exercises:
  1. Using the texts [KLE91] and [MAC95], collect a list of people who have contributed to the field of formal proof: their names, their research fields and the type of contribution.
  2. Using the texts [KLE91] and [MAC95], build a timeline of the most relevant events (famous problem statements, landmark proofs) related to formal proof. Use a web search engine to complete this timeline up to the present day.

- The papers [BUN99] and [GEU99].
  Reading: Make sure to remember the major tools and the most relevant techniques (term rewriting, unification, logic programming, abstract interpretation, . . .) discussed in these surveys. They will be referenced later when we study the application of formal proof using real-world research papers.
  Exercises:
  1. Using a web search engine, find the two most recent papers which perform a proof using the theorem provers Isabelle, HOL Light or Coq. Describe the research field of the paper and the problem being proved.

2. Find a proof that $\sqrt{2}$ is an irrational number in an interactive theorem prover like HOL light (`http://www.cl.cam.ac.uk/~jrh13/hol-light/`), Isabelle (`http://www.cl.cam.ac.uk/research/hvg/Isabelle/`) or Coq (`http://coq.inria.fr`). Document the sequence of steps required to download the prover, install it in your system and verify the proof.

3. Using a web search engine, find information on automated theorem proving and constraint solver competitions. You can use search terms like "prover", "solver" and "competition". Describe their purpose and create a list of competitions indicating for each one (a) the competition name and its associated conference, (b) its URL, (c) the problem being studied, (d) the year of its last edition, and (e) the last winner.

4. Using a web search engine, collect a list of collaborative repositories of formal proofs, *e.g.* wikis.

1.5. Further reading

The bibliography of the material used in this block provides an extensive list of recommended readings in the field or formal proof. The following books can also be used as a reference:

D. J. Velleman (1994). *How to Prove It: A Structured Approach*. Cambridge University Press, 2nd edition.

D. Solow (2004). *How to Read and Do Proofs: An Introduction to Mathematical Thought Processes*. Wiley, 4th edition.

T. Sundstrom (2006). *Mathematical Reasoning: Writing and Proof*. Prentice Hall, 2nd edition.

K. J. Devlin (2002). *The Millenium Problems: The seven greatest unsolved mathematical puzzles of our time*. Basic Books.

W. Dunham (1991). *Journey through Genius: The great theorems of Mathematics*. Penguin.

2. Formal definitions

2.1. Overview

In order to reason about a problem, it is necessary to understand it correctly. Natural language is a good means to present a problem or to argue about the problem intuitively. However, natural language has an important shortcoming: *ambiguity*. Terms and concepts may have several meanings, and usually the language is not precise enough to provide an accurate description of a problem.

In contrast, mathematical language offers a plethora of concepts and notations with well-defined meanings. Logic provides formal languages to describe properties and deduction rules to infer consequences or detect contradictions. The process of describing a problem in terms of mathematical and logical concepts is called *formalisation*.

In a formal context, a *definition* is a precise description of an entity or concept, which is referred from this point onward using a given *term* or *notation*. The purpose of a definition is to provide a clear and unambiguous understanding about the concept being described.

For example, the following is an example of definition from a branch of theoretical computer science, *formal languages*. This discipline studies properties on words and collections of words called languages. One of the first concepts that needs to be defined is the alphabet, the collection of symbols that can be used to build a valid word:

Definition 1 (Alphabet). An *alphabet* is a finite and non-empty set of elements called *symbols*. Alphabets are usually denoted using the greek upper-case letter Σ. For example, the alphabet for writing binary numbers is the set $\{0,1\}$.

Let us examine this definition in detail to identify several good practices:

- It provides a term to refer to this concept, *alphabet*.
- It defines the notation that will be used to refer to alphabets, Σ.
- It provides an example of the concept being defined to facilitate the comprehension of the definition.
- It provides a precise, unambiguous and brief definition of the concept.

As it is based on mathematical notions of sets, it is conveying a lot of information compactly:

Set An alphabet is a collection of elements, with no order among elements and no duplicates. Therefore, {1,1} is not an alphabet and {0,1} is the same alphabet as {1,0}.

Finite The number of elements in the set must be finite. Therefore, the set of integer numbers $\mathbb{Z}$ is not an alphabet.

Non-empty There must be at least one element in the alphabet. Thus, the empty set $\emptyset$ is not an alphabet.

Once such a definition is provided, it is possible to reason about alphabets without misunderstandings: the reader will have a clear picture of what is an alphabet and what are the properties of an alphabet. Consequently, this definition will make our arguments more concise: if we are talking about an entity Σ and we say it is an alphabet, we automatically know that it is a finite and non-empty set.

Another advantage of definitions is their use as an abstraction mechanism: the effort required to formalise a concept usually leads us to identify the relevant aspects of the concepts and discard the rest. In this case, our only concern is the set that all symbols have the same role within the same alphabet and that the size of the alphabet cannot be infinite nor zero.

There are some fundamental 'rules-of-thumb' that should be considered when creating a definition:

- Use terms and notations which are meaningful to the audience. Specifically, avoid using counter-intuitive definitions such as "$a + b$ denotes the product of a and b": if a term has a well-established meaning in the research field, use a new term to avoid misunderstandings.

- If a concept has been previously defined in the literature, it is (almost) always better to reuse that previous definition verbatim, citing the original source. The only reason not to reuse a previous definition is when a new notation will clarify our exposition. Even then, it is a good idea to clarify the differences between the old definition and the new one.

- Make sure that the audience understands the notions on which the definition is based. For example, if we define a Petri Net as a "directed bipartite graph", the readers should know what those terms mean beforehand: they should either be part of the folklore of the research field or they should be introduced prior to the definition.

- If the concept being defined is very complex, it is better to define it incrementally, *i.e.* start creating auxiliary definitions of its components before describing the concept as a whole.

- Definitions may refer to other definitions, but it is necessary to avoid circular references. For example, two definitions like "a wolf is a member of a pack" and "a pack is a group of wolves" are unacceptable because of their circularity.

- Examples can clarify a definition, but they do not replace it. Saying that "11 and 13 are twin primes" does not constitute a proper definition of "twin prime": it does not provide information about whether other numbers like 17 and 19 are twin primes.

Twin primes

Two natural numbers x and y are twin primes if both x and y are primes and $x = y + 2$. Some examples of twin primes are: 5 and 7, or 11 and 13.

In this section, we provide a classification of definitions, recall several mathematical concepts that can be used in a definition (the families of numbers, collections like tuples or sets, functions, . . .) and describe several strategies to formalise a computation. Finally, we also study examples of more complex formal definitions.

2.2. Classification of definitions

There are several strategies to define a concept:

- **Extensional definition:** A concept can be defined by exhaustively enumerating the instances of this concept.

  For example, if we need to define "week-days", we can simply do it by listing them: {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday}. Obviously, this approach is more adequate to describe concepts with a finite number of instances, as they are amenable to an exhaustive enumeration. For example, if we need to define "even numbers" we can enumerate them as {0, 2, 4, 6, . . . }. However, unless the list of remaining terms is obvious or already known by the reader, using ". . . " or *etcetera* might create ambiguity and it is not recommended.

- **Intensional definition:** A concept can be also be defined by stating the properties that identify all the instances of this concept, *i.e.* the necessary and sufficient conditions to be classified as an instance of the concept.

  Usually, intensional definitions build upon a previously defined concept. One way to do this is to add an additional constraint to a concept, establishing a subclass within that concept. For example, even numbers are a subclass of natural numbers, those that are divisible by two. As another example, an isosceles triangle is a class of triangle in which two sides have the same length.

Necessary and sufficient

Necessary and sufficient are relationships among two properties A and B. A is *necessary* for B if B implies A, *i.e.* B can only be true when A is true. A is *sufficient* for B if A implies B, *i.e.* B is always true when A is true. For example, an atmosphere is a necessary condition for the habitability of a planet, but it is not a sufficient condition.

Another way to build upon previous definitions is to combine different concepts or several instances of the same concept. For instance, complex numbers are defined as a pair of real numbers where the first number is the *real* component and the second is the *imaginary* component. As another example, in two-dimensional geometry a circle can defined using a point (the centre of the circle) plus a natural number (its radius).

- **Recursive definition:** A concept can also be defined in terms of itself, although special care needs to be taken to avoid a circular definition.

  This definition usually starts with a set of *base cases*, which are instances of this concept that can be identified directly. Other instances of the concept are built incrementally from previously existing instances, *i.e.* each instance should lead back to some base case in a finite number of steps.

  For example, let us consider the following definition of the set of natural numbers (ℕ). Let us define a constant, *zero* (0), and one operation, next(ℕ):ℕ, which takes one natural number as an argument and returns a natural number. The set of natural numbers ℕ is defined as follows:

  1. Zero is a natural number.
  2. If *n* is a natural number, next(*n*) is a natural number.
  3. Nothing else is a natural number.

  The notation used by this definition is a bit cumbersome, as 2 will be expressed using next(next(zero)). However, it captures the notion of natural numbers: the infinite set of positive whole numbers that start from zero. Also, it is interesting to highlight the condition 3 of this definition, which is sometimes found explicitly in recursive definitions and sometimes left implicit.

2.3. Numbers

Numbers are the mathematical abstraction that captures quantity and they can be used to express magnitude, duration, position, length, cost, . . . The following is an incomplete list of families of numbers:

- **Natural numbers (ℕ):** A natural is a positive whole number, *e.g.* 2, 35 or 2350. Depending on the context, this set can be defined to start from 0 or from 1.

- **Integer numbers (ℤ):** An integer is a whole number which is either positive, negative or zero. Some examples of integers are 0, 13 or -976. Notice that all natural numbers are also integers.

- **Rational numbers ($\mathbb{Q}$):** A rational is a number that can be expressed as the division of two integers a/b, with $b \neq 0$. For example, 2/3 or -18/7 are examples of rational numbers. Rational numbers also include integers, as b can be equal to 1, *e.g.* 13/1 = 13.

- **Real numbers ($\mathbb{R}$):** A real is any number which can be expressed as a finite sequence of digits (the whole part) and a possibly infinite sequence of digits (the decimal part). This definition includes all rational numbers and other *irrational* numbers that cannot be expressed as a fraction, like $\sqrt{2}$ or $\pi = 3.141592\ldots$

- **Complex numbers ($\mathbb{C}$):** A complex number is a pair of two reals, the *real* component and the *imaginary* component. The imaginary component is a multiple of the imaginary unit i, defined as $\sqrt{-1}$. Real numbers are specific cases of complex numbers where the imaginary component is zero.

Regarding the use of numbers in definitions, natural and integer numbers can be used to describe magnitudes that are measured in multiples of a *discrete* unit like bits, instructions, cycles, ... Some examples are: the length of a message or document (in terms of bits, bytes, characters, ...); the maximum capacity or current size of a container (in terms of elements, bytes, pages,...); the duration of an event (in terms of number of periods or cycles); or the location of an object within a discrete grid. On the other hand, real and rational numbers are used to capture *continuous* magnitudes with an arbitrary precision, *e.g.* the duration of an event in seconds (with decimals).

2.4. Mathematical collections

Some complex concepts can be described as an aggregation or *collection* of simpler concepts. There are several ways to describe this collection, depending on whether the order of elements matters, whether the number of elements is fixed a priori and whether there can be duplicate elements. These different views originate the concepts of *tuple*, *set* and *sequence*, that are detailed in Table 1.

For collections with a variable number of elements, it is possible to describe specific qualities of the size of this aggregation. For example, it is possible to say it is empty (no elements) or non-empty (at least one element); that is is finite or infinite; that it is bounded (the number of elements is fixed a priori) or unbounded (the number of elements is arbitrary). Also, we can use the term “possibly” to express that one of the previous properties may occur, but is not required. For example, a collection may be described as “a finite non-empty set” or as “a possibly infinite sequence”.

Table 1: Mathematical collections.

Concept	Size?	Ordered?	Duplicates?	Example
Ordered pair	Fixed (2)	Yes	Allowed	Coordinates of a point in 2D
Triple	Fixed (3)	Yes	Allowed	Coordinates of a point in 3D
Quadruple	Fixed (4)	Yes	Allowed	Parcel measures: Length + Height + Width + Weight
k-Tuple	Fixed (k)	Yes	Allowed	Results of k blood analysis
Sequence	Unbounded	Yes	Allowed	Symbols of a word
Set	Unbounded	No	Disallowed	Integer numbers $\mathbb{Z}$
Multiset	Unbounded	No	Allowed	Votes in a ballot box

2.4.1. Tuples

Tuples are ordered collections of elements with a fixed size and where duplicates are allowed. Tuples may receive different names according to the number of elements they hold: pair, triple, quadruple, quintuple, . . . For larger tuples, they may be called k-tuples, where k is the number of elements, *e.g.* a chess board can be defined as a 64-tuple.

The elements of the tuple are comma-separated and enclosed between parentheses (), brackets [] or angle parentheses ⟨ ⟩, *e.g.*⟨12,3,4.5⟩. Usually, when the tuple is defined, we should assign a name to each element in the tuple to allow direct references. For example, a graph G can be defined as a tuple $G = (V,E)$ where V is called the set of vertices and E is a set of edges. Here, the names V and E are used to directly identify each element in the tuple.

2.4.2. Sets

Sets are unbounded and unordered collections without duplicates. For example, all families of numbers ($\mathbb{N}$,$\mathbb{Z}$,$\mathbb{Q}$,$\mathbb{R}$,$\mathbb{C}$) are sets. It is also possible to define special flavours of sets where duplicates are allowed (multisets).

Sets are denoted using capital letters (A, B, X) and, to perform an extensional definition, their elements are listed within keys, *e.g.* {1,2,3}. It is also possible to provide an intensional definition of a set, by defining the property satisfied by its elements. The format used for this definition is { x ∈ domain | property(x) }, where x denotes an element of the set. For example, the set of odd natural numbers can be defined as:

$$\text{ODD} = \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} : x = 2y + 1\}$$

This definition should be read as "the set of odd numbers is formed by natural numbers which are equal to $2y$+1, for some other natural number y".

The fundamental operation on a set is *membership*: deciding whether an element belongs to the set. The symbols ∈ and ∉ are used to communicate that an element belongs or does not belong to a set, respectively. For example, 3 is a natural number ($3 \in \mathbb{N}$) but the square root of 2 is not a natural number ($\sqrt{2} \notin \mathbb{N}$). The set with no members is called the *empty set* ($\emptyset$). The number of elements in a set A is called the *cardinality* of a set and it is denoted as $|A|$, *e.g.* $|\emptyset| = 0$.

If all elements of a set A belong to another set B, we say that A is *included in B* or that A is *subset* of B. The notation used is $A \subseteq B$ (if A and B may be equal) or $A \subset B$ (if B has some elements that are not in A).

Defining sets

The subset symbol can be used to define a set implicitly. For example, $A \subseteq Z$ states that A is a subset of the set of integers: implicitly, it is defining A as a set.

The *power set* of a set A is the set formed by all subsets of A. The power set can be denoted as $P(A)$ or 2^A, as the number of subsets of a set with n elements is precisely 2^n. For example, if $A = \{a,b,c\}$ then

$$P(A) = \{\emptyset,\{a\},\{b\},\{c\},\{a,b\},\{b,c\},\{a,c\},\{a,b,c\}\}$$

Defining sets of sets

If an aggregation is a set of sets, its type can be defined using the power set. For example, $A \subseteq P(\mathbb{R})$ means that A is a set whose elements are sets of real numbers.

Sets can be combined using the *union* ($\cup$) and *intersection* ($\cap$) operators. The result of these operators is a new set with the elements that belong to at least one set (union) or both sets (intersection). For example, if $A = \{x,z\}$ and $B = \{y,z,w\}$, then $A \cap B = \{z\}$ and $A \cup B = \{x,y,z,w\}$. If two sets have an empty intersection, *i.e.* $A \cap B = \emptyset$, they are called *disjoint*.

The difference of two sets A and B, $A \setminus B$, is the set having all members of A that do not belong to B. For example, natural numbers are either even or odd: $\text{EVEN} = \mathbb{N} \setminus \text{ODD}$. It is also possible to define the *complement* of a set A, denoted as $\overline{A}$, which is a new set formed by all elements that do not belong to A. In order to define this complement, it is necessary to define what we consider the *universe*, *i.e.* the complete set of elements. For example, if we consider the set of natural numbers as our universe, then $\overline{\emptyset} = \mathbb{N}$ and $\overline{\text{ODD}} = \text{EVEN}$.

Finally, the *cartesian product* of two sets A and B, denoted as $A \times B$, is a set formed by all pairs where the first element belongs to A and the second element belongs to B. For example, if $A = \{x,y\}$ and $B = \{1,2,3\}$ then

$$A \times B = \{(x,1),(x,2),(x,3),(y,1),(y,2),(y,3)\}$$

Defining tuples

The cartesian product can be used to describe the type of each element in a tuple. For example, an ordered pair x of natural numbers can be defined as $x \in \mathbb{N} \times \mathbb{N}$ and a triple y of two reals and one integer can be defined as $y \in \mathbb{R} \times \mathbb{R} \times \mathbb{Z}$.

2.4.3. Sequences

Sequences are unbounded and ordered collections that allow duplicates. Elements of a sequence are typically presented as a comma-separated list, ended in "..." if the sequence is infinite. For example, the numbers in the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, ... Also, a word is formally defined as a finite sequence of symbols chosen from a finite alphabet, *e.g.* "abcd" is a sequence of four symbols taken from the Latin alphabet.

Fibonacci sequence

The Fibonacci sequence is a sequence of numbers which begins with 0 and 1 and where each subsequent number is the sum of the two previous numbers. This sequence exhibits many interesting mathematical properties and it occurs naturally in different biological settings.

The symbol used to denote an empty sequence depends on the context. For example, the Greek letters lambda (λ) and epsilon (ε) are both used to describe an empty word in theoretical computer science.

Furthermore, there are different notations to refer to a sequence. In general, the entire sequence is referenced using a variable name plus an index like i,n or N as a subscript, *e.g.* the sequence of Fibonacci numbers is F_i. A specific element within the sequence is identified by the value of the index, *e.g.* F_2 is the element in position 2 in the Fibonacci sequence. Depending on the context, the first element of a sequence can be 0 or 1. Thus, we should make sure that we establish which of these conventions is used in our case.

Regarding the operations on a sequence, there is no generally accepted notation to describe that an element appears (or does not appear) in a sequence or whether it appears before another one in the sequence. Also, it is possible to define concepts like *subsequence*, but the exact meaning of these concepts and the notation used to describe them depends on the problem.

2.5. Relationships among concepts

A concept does not exist in isolation and it is often *related* to other concepts. For example, a person is related to his family and friends, a node in a network is connected to other nodes, each item in an auction has a current price, ... There are different types of relationships and there are mathematical concepts to formalise relationships: *relations* and *functions*.

A *binary relation R* among two sets A and B is a set of pairs (a,b) where $a \in A$ and $b \in B$, *i.e.* $R \subseteq A \times B$. Set A is called the *domain* and set B is the *codomain*. Intuitively, each pair (a,b) in the relation denotes that a is related to b. For example, given two sets $People = \{Ann,Peter\}$ and $Desserts = \{ice\text{-}cream,cake,cookie\}$, the relation Likes = $\{(Ann,cookie),(Peter,ice\text{-}cream),(Peter,cookie)\}$ could be used to state the fact that Ann likes cookies and Peter likes both ice-cream and cookies. Obviously, it is possible to extend binary relations into n-ary relations, where each element is a n-tuple.

> There are many examples of binary relations in mathematics, such as ordering relations. For example, the less-or-equal ($\leq$) ordering among the reals is a binary relation among $\mathbb{Z}$ and $\mathbb{Z}$ where, in all pairs, the first element is less-or-equal than the second, *i.e.* LessOrEqualZ = $(0,0),(1,1),(0,2),(4,7),\ldots$

Relations can describe many different types of relationships as, for example, they do not impose constraints on the number of occurrences of a specific element from the domain or codomain in the relation. This allows the description of different types of mappings, *e.g.* one-to-one, one-to-many, many-to-one and many-to-many mappings. For instance, equality (=) among integers is a one-to-one mapping $\{\ldots, (-1,-1), (0,0), (1,1), \ldots\}$ while disequality ($\neq$) is a many-to-many mapping.

However, sometimes the relation to be described is simpler: each element of the domain is mapped to exactly one element of the codomain. This specific type of relation is called a *function*, and it is formally represented as $f : A \to B$, where f is the function name, A is the domain and B is the codomain. Given an element x in the domain, the corresponding element y in the codomain is called the *image* of x, as it denoted as $f(x)$.

For example, let us formalise the execution time of a program in a machine language by assigning a duration (in terms of processor cycles) to each instruction. First, we need to define our domain: I will be the finite set of instructions labels in our machine language. The codomain will be the set of natural numbers $\mathbb{N}$. Let us choose the name d (from Duration) for our function. Finally, our function can be formally defined as $d : I \to \mathbb{N}$. For example, the fact that the `goto` instruction requires 2 processor cycles can be stated as $d(\texttt{goto}) = 2$.

A function is called *injective* if $f(a) = f(b)$ implies $a = b$, *i.e.* two elements from the domain have the same image only if they are equal. A function is called *surjective* if every element in the codomain is the image of some element in the domain. Finally, a *bijection* or *bijective* function is a function which is both injective and surjective.

It is also possible to define a function with several inputs or outputs. From the point of view of the definition, this is the same as considering that the input or output of the function is a tuple, so its type can be defined using the cartesian product operator. For example, the addition among reals can be defined as the function $add : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, which receives two reals as input and produces one real as output, e.g. $add(1.3,2) = (3.3)$.

To specify a function, it is necessary to describe the mapping between the input and the output. For example, $add(x,y) = x + y$ is a sufficient description for the addition of integers. Sometimes it is easier to describe the mapping in terms of cases. For example, let us consider the *sign* function which, given an integer, returns -1,0 or +1 depending on the sign of this integer. This function can be formally defined as $sgn : \mathbb{Z} \to \{-1,0,+1\}$, where the result of sgn is as follows:

$$sgn(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ +1 & \text{if } x > 0 \end{cases}$$

2.6. Defining computations

In computing, the research contribution may be a process, computation or algorithm. In almost every context, it is not possible nor desirable to exhaustively document the full source code or hardware architecture performing this computation: the implementation might be too long and it might contain many irrelevant details which are dependent on the specific platform. Instead, we should aim to provide an abstract description which is *language and platform-independent.*

A first approximation is defining the *result* of the algorithm, rather than the computation that leads to this result. In this way, the problem consists in characterising the result as a function or mathematical formula on the inputs. For example, let us consider the primality test, a function that checks whether a given natural number is prime or not:

$$prime(x) = \begin{cases} 0 & \text{if } \exists y \in \mathbb{Z} : x \bmod y = 0 \ \wedge \ y \neq 1 \ \wedge \ y \neq x) \\ 1 & \text{otherwise} \end{cases}$$

A number is prime if it can only be divided by and itself, *i.e.* the division by any other number has a remainder or modulo different from zero. The definition of the primality test as a function simply states this property, but it does not describe the process required to conclude whether there are divisors or not.

Another option to formally describe computations is to characterise the computation *recursively*. For example, the *greatest common divisor* (*gcd*) of two non-negative integers is another integer that can be computed recursively in the following way:

$$gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, x - y \left\lfloor \frac{x}{y} \right\rfloor) & \text{otherwise} \end{cases}$$

A critical aspect of a recursive definition is ensuring the termination of the recursion. For example, in the previous definition, it is necessary to ensure that the case $y = 0$ will eventually be reached for any arbitrary x and y after a finite number of iterations. Termination may be non-obvious as in this example and a complex proof may be required to ensure it. Another drawback of recursive definitions is that they are not adequate for all problems, only for those that can be addressed using a divide-and-conquer approach.

For more complex computations, the use of *pseudocode* is recommended. Pseudocode is a generic high-level algorithmic notation. Pseudocode provides basic control-flow primitives typically used in structured programming, such as if-then-else conditionals or for/while-do/repeat-until loops. For example, the following is a pseudocode for a sorting algorithm called *insertion sort*:

Algorithm 1 InsertionSort(A): Array[0..*N*].
Precondition: The elements of *A* can be compared using an order relation <.
Postcondition: *A* is sorted in increasing order according to the relation <.

```
1: for $i = 1$ to $N - 1$ do
2:   $v \leftarrow A[i]$                  {Find a location for $v$ in A[0..$i$].}
3:   $j \leftarrow i - 1$
          {Decrease $j$ until we reach the start of the array ($j = -1$) or
           we find a value which is smaller or equal to $v$ ($A[j] \leq v$).}
4:   while $j \geq 0$ and $v < A[j]$ do
5:     $A[j+1] \leftarrow A[j]$        {Advance $A[j]$ as $v$ needs be stored before it.}
6:     $j \leftarrow j - 1$
7:   end while
8:   $A[j+1] \leftarrow v$
9: end for
```

Let us take a closer look at this example of pseudocode:

- We describe the *signature* of the algorithm, *i.e.* the number of input and output parameters and their types.
- The algorithm and each input and output parameter is given a name that identifies it.
- The algorithm describes the requirements on the inputs (*precondition*) and the expected outcome of the execution (*postcondition*) in natural language.
- The code is commented to improve its understanding while trying to avoid being verbose, *i.e.* lines like $j \leftarrow j - 1$ are self-explanatory.
- When they are not necessary, variable declarations are skipped: their type is obvious from the context and variable names are not reused to avoid confusion. Other syntactical constructs like the end-of-statement semicolon (;) are also skipped when they do not contribute to the clarity of the pseudocode. On the other hand, keywords like “end” can be preserved to clarify the nesting of control-flow constructs (when indentation is not sufficient).
- All the instructions in the algorithm are numbered to allow references from the text.

A disadvantage of pseudocode is that it is oriented towards an *informal* description of an algorithm. Usually, the description of the process abstracts details to improve clarity or for brevity. This does not mean that a description in pseudocode cannot become formal: if the algorithm is simple enough that each of its steps has a well-defined meaning, a pseudocode can be used to reason about the correctness or efficiency of an algorithm. This is the case, for example, for the insertion sort algorithm used as our previous example.

2.7. Putting everything together

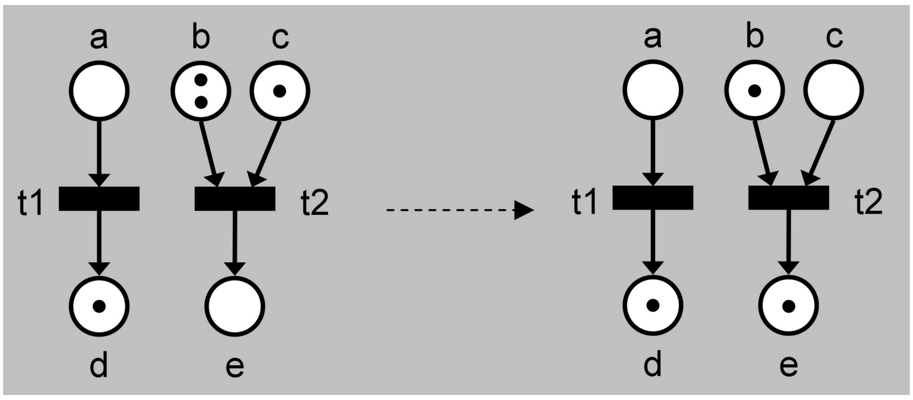
To conclude, we will analyse several complex formal definitions which use the mathematical concepts described in this section. The concept being defined is a modelling notation for concurrent systems called *Petri Nets*.

Intuition: A Petri Net is a directed graph with two types of nodes, *places* (modelling variables or resources) and *transitions* (modelling computations or actions). Places can be connected to and from transitions through directed arcs. Each place can be marked with *tokens*. Graphically, places are drawn as circles, tokens as black dots within the place, transitions as rectangles or bars and arcs as arrows. For example, Figure 2(left) depicts a Petri Net with fives places (a-e), two transitions (t1-t2) and four tokens in total.

Additional reading

[Mur89] **T. Murata (1989).** *Petri Nets: Properties, Analysis and Applications*. Proceedings of the IEEE, 77(4):541–580, IEEE Press.

Figure 2. Example of a Petri Net.



The state of a Petri Net evolves by *firing* transitions. Firing a transition removes tokens from its *input places* (places with an arc towards the transition) and puts tokens on its *output places* (places with an arc from the transition). The number of tokens being removed from an input place or created in an output place is called *weight* and it is denoted as a number labelling the corresponding arc. By convention, if the weight of an arc is one (the transition removes one token or puts one token in that place) then it is not drawn in the arcs of the figure.

For example, Figure 2(right) shows the result of firing transition t2 on the Petri Net from Figure 2(left). In order to fire a transition, there must be a sufficient number of tokens in the input places, *i.e.* the transition must be *enabled*. For example, in Figure 2(left), transition t2 is enabled while transition t1 is disabled as there are no tokens in its input place A.

Definitions: The following is a formal definition of a Petri Net extracted from [MUR89]:

Definition 2 (Petri Net): A Petri Net is a 5-tuple $PN = (P,T,F,W,M_0)$ where:
$P = \{p_1,p_2,\ldots,p_m\}$ is a finite set of places
$T = \{t_1,p_2,\ldots,t_n\}$ is a finite set of transitions
$F \subseteq (T \times P) \cup (P \times T)$ is a set of arcs (flow relation)
$W : F \rightarrow \{1,2,3,4,\ldots\}$ is a weight function
$M_0 : P \rightarrow \{0,1,2,3,\ldots\}$ is the initial marking
$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$
A Petri Net structure without any particular marking is denoted as $N = (P,T,F,W)$.

Again, let us study this definition in detail:

- The definitions of places and transitions state that they are finite sets. Therefore, they cannot be infinite and there are no duplicates. Furthermore, we have that $P \cap T = \emptyset$ (nothing can be simultaneously a place and a transition) and $P \cup T \neq \emptyset$ (a Petri Net must have at least one place or transition).

- The flow relation is stating succinctly that arcs can only connect places with transitions and transitions with places. Hence, there can be no edges among pairs of places or transitions or from a place or transition to itself.

  Here, an arc is being defined as a pair (*source*,*target*) where the first element is the source of the arc and the second is the target. Therefore, $T \times P$ is denoting all the potential arcs from transitions to places and $P \times T$ the set of arcs from places to transitions. As F is a subset of both, we have that some places and transitions may not be connected. Furthermore, there is no requirement on F to be non-empty: therefore, it is possible to have a Petri Net with no arcs among their elements. Moreover, there is no restriction on a place being both input and output for the same transition, *i.e.* the transition can remove tokens and then add tokens to the same place.

- W and M_0 describe the assignment of a weight to each arc and a number of initial tokens to each place, respectively. Notice that this assignment is defined as a function and that the domains and codomains of the function provide relevant information. For example, the weight of a node is a natural number greater or equal to 1: it is not possible to have weight 0 on an edge. On the contrary, it is possible to have 0 tokens in the initial marking.

Now let us consider how the concept of firing can be formally defined:

Definition 3 (Transition Enabling): A transition t of a Petri Net $N = (P,T,F,W)$ is called enabled in a given marking $M : P \rightarrow \{0,1,2,3,\ldots\}$ if for every place $p \in P$ we have that $(p,t) \in F \rightarrow M(p) \geq w(p,t)$. A non-enabled transition is also called "disabled".

Definition 4 (Transition Firing): The firing of a transition t of a Petri Net $N = (P,T,F,W)$ on a marking M where it is enabled produces a new marking M' such that:

$$M'(p) = \begin{cases} M(p) - w(p,t) & (p,t) \in F \ \wedge \ (t,p) \notin F \ \text{(input place)} \\ M(p) + w(t,p) & (p,t) \notin F \ \wedge \ (t,p) \in F \ \text{(output place)} \\ M(p) - w(p,t) + w(t,p) & (p,t) \in F \ \wedge \ (t,p) \in F \ \text{(in/out place)} \\ M(p) & \text{otherwise (unaffected by the firing)} \end{cases}$$

Let us review this definition one more time:

- Also, notice that we have split the definition of "enabling" and "firing", both to make them simpler and also to allow reference to each concept individually.

- In order to define whether a transition is enabled or not, we need to know which are its input and output places. This can be done using the flow relation F: if (p,t) belongs to F it means that p is an input place and conversely, if (t,p) is in F it means p is an output place of t.

- Therefore, the property on enabling can be re-read as: the number of tokens $M(p)$ in each input place p should be equal or larger to the number of tokens consumed by the transition, *i.e.* the weight of the arc.

- Regarding the firing of a transition, we define the new marking after the firing in terms of the marking before the firing. The relationship between the new and old markings is also made explicit in the choice of names for the markings: M' vs M. The only difference is the deletion of tokens in the input places and the creation of tokens in the output places, while the remaining places are unchanged. Notice that we need to distinguish a special case in our definition: those places that are both input and output places!

- Finally, firing is only defined on enabled transitions. This avoids creating markings where a place has a negative number of tokens.

Petri Nets inspire many other concepts that require a formal definition: deadlocks, invariants, boundedness, . . . [MUR89] provides a starting point to practise these formal definitions.

2.8. Exercises

1. Using a web search engine, find a **formal** definition for the following concepts:

   a) Finite State Machine (FSM).

   b) Deadlock.

   c) Linear optimization problem.

   d) Constraint Satisfaction Problem (CSP).

   e) Public-key cryptosystem.

   f) Genetic algorithm.

   g) Serialization of a set of transactions.

   h) Longest common subsequence of two strings.

   i) Inverse of a matrix.

   For each concept, identify the most significant terms and notation in the definition and analyse its meaning by creating specific instances of the concept.

2. Write a formal definition for the following concepts:

   a) The list of tasks in a TODO list, each consisting on a deadline, a description and a leader.

   b) The contents of a `/etc/passwd` file in the Unix operating system.

   c) A single move in a game of chess. Refer to the *algebraic chess notation* (`http://en.wikipedia.org/wiki/Algebraic_chess_notation`) for a description of the information provided by a chess move.

   d) A complete game of chess, including its conclusion as a win or draw. You do not need to check whether each move is correct according to the rules of chess.

   e) The contents of a five-card hand in a game of poker.

   f) The rock-paper-scissors game and the process for deciding the winner.

2.9. Further reading

In this section, we have considered basic mathematical concepts like sets or functions and their use in definitions. Each research field uses its own family of concepts that are used to establish formal definitions: words, languages, graphs, trees, groups, lattices, . . . Defining all these concepts is out of the scope of this section. Our goal has been providing the basic techniques and several examples on how to study formal definitions and extract their meaning.

The final section on applications of formal proof provides plenty of examples of formal definitions in research papers. It is recommended to revise this section if you are having trouble with a specific definition. Remember that before understanding a definition, it is necessary to understand the terms and notation used inside, going back to the source where they defined if it is necessary.

3. Proof strategies

3.1. Overview

This part of the course focuses on the techniques that can be used to develop a formal proof. Each property is different and its proof may require using a different approach. In this block, we will study the different approaches to tackle a formal proof, learn how to apply them to specific problems and understand which techniques are more suitable for each type of problem. To sum up, our goals are three-fold:

- Learn to understand proofs by comprehending the overall approach used in the proof.

- Learn to identify incorrect proofs and situations where a proof strategy has not been applied correctly. This knowledge will be useful to avoid making those mistakes in our proofs.

- Learn how to set up and write our own formal proofs for many different types of properties.

3.2. Required reading

[CLA09] **R. Clarisó (2009).** *Formal Proofs: Understanding, writing and evaluating proofs*. Editorial UOC.

[CUP05] **A. Cupillari (2005).** *The Nuts and Bolts of Proofs*. Elsevier.

[CAS00] **B. Casselman (2000).** *Pictures as Proofs*. Notices of the AMS, 47(10):1257–1266, 2000, AMS.

- Section 4 of [CLA09] presents the overall flow in the development of a formal proof, focusing on the basic steps that need to be completed and the common pitfalls in each of these steps. For example, at some point we should assume a critical attitude towards the property that is being proved and try to actually *disprove* it. This will either reveal flaws in the property or provide an intuition on why the property holds.

- Section 5 of [CLA09] introduces the concept of *proof strategy*, a generic approach to complete proofs which can be applied to a variety of problems. An example of a strategy is *reduction to an absurd*, where we assume that the property to be proved is actually false and try to reach a contradiction (which leads us to conclude the impossibility of the property being false). Each proof strategy is briefly presented, together with an example of a formal proof using that approach. Finally, some suggestions on which strategies might be more adequate for each specific type of problems are presented.

- The core of the book [Cup05] is devoted to the study of proof strategies. Each proof strategy is presented in detail individually, together with motivating examples and exercises. Finally, the book concludes with several indications on selecting the most suitable strategy.

- Finally, the paper [Cas00] presents another proof strategy: using graphical depictions as a formal proof. The paper shows several examples of these proofs and discusses the conditions under which such a proof can be considered valid. This completes our catalogue of proof strategies.

3.3. Reading tips

- First and foremost, do not be discouraged by the apparent complexity of any proof. Reading a proof related to any field outside of one's area of expertise requires an additional effort, and many examples deal with mathematical properties that may be unfamiliar. Reading should be targeted at comprehending the overall strategy rather than understanding the rationale behind every single step of the proof.

- Do not be fooled by papers and textbooks which always seem to make the right choice directly and without effort: proofs are the result of an iterative process and the final version of the proof may not reflect all the approaches that have been explored. Furthermore, selecting the most suitable proof strategy is a matter of experience. The saying "practice makes perfect" applies to this context: exercise this skill to improve it. Choosing the proper strategy will seem obvious in hindsight but it is not trivial, and making a wrong choice is a good way to gather insights to select a better alternative.

- In addition to learning proof strategies, proof examples are a good source of tips on terminology, writing style and even formatting. Make sure to notice these details in the examples as you read them.

- Notice that a property may be proved using different strategies. As an example, consider the property "the sum of the first n natural numbers is equal to $\frac{n\cdot(n+1)}{2}$". This property is proved both in Example 3 in Section *Direct Proof* of [Cup05] and in Example 1 in Section *Mathematical Induction* of [Cup05].

- These chapters show the most common mistakes that appear when performing a proof. Make sure that you avoid those errors in your proofs.

3.4. Detailed exercise and reading plan

1) [CUP05], Section *Introduction and Basic Terminology*.

2) [CUP05], Section *General suggestions*.

3) [CLA09], Section 4 (*Planning formal proofs*).
Reading: Understand that a proof is the result of an iterative process and that proof strategies define the way the problem is approached. Learn about other specifics of proof development: language, style, ...
Exercises: None so far. The following readings will study each proof strategy one by one and provide additional examples and exercises.

4) [CUP05], Section *Basic Techniques to Prove If/Then Statements*
Reading: This section focuses on the concepts of *negation* and *implication*, which appear in many properties, and the concept of *truth table* as formal means to analyse the veracity of a statement. It also introduces two basic proof strategies: direct proof and proof by contrapositive.
Exercises: From this section of [CUP05], 1-10 (negation of statements), 11-24 (contrapositive, inverse and converse) and 25-28 (comprehension of proofs). It is recommended to solve at least 4 from the first two blocks and at least one from the latter.

5) [CLA09], Section 5 (*Proof strategies*).
Reading: This section provides a quick overview of several proof strategies that will be reviewed in more detail in the following chapters of [CUP05].

6) [CUP05], Section *"If and Only If" or "Equivalence Theorems"*
Reading: The concept of "A if and only if B" (abbreviated as "A iff B" or using the notation $A \leftrightarrow B$) denotes a double implication: A implies B and B implies A. This means that A is true whenever B is true and false whenever B is false, *i.e.* A and B can be used interchangeably. Equivalence properties are very common!
Exercises: From this section of [CUP05], exercises 2, 3, 5 and 6.

7) [CUP05], Section *Use of Counterexamples*
Reading: If we want to prove that a property is false, it is sufficient to find a case where it is not satisfied. This approach can sometimes be a useful proof strategy. In order to fully take advantage of this chapter, it is necessary to recall the strategies for building the negation of a statement, from the previous section *How to construct the negation of a statement* of [CUP05].
Exercises: From this section of [CUP05], exercises 1, 3 and 5 (find a counterexample for statements that are known false) and 7, 8, 10 and 13 (decide whether a statement is true or false, and if it is false, find a counterexample).

8) [CUP05], Section *Mathematical induction*
Reading: Induction is a powerful proof strategy used when the property being proved involves an infinite collection of elements. As proving the property element by element is impossible, induction offers a general mechanism to complete the proof for *all* its elements at once.
Exercises: From this section of [CUP05], exercises 1, 3, 4, 6, 7 and 9.

9) [CUP05], Section *Existence theorems*
Reading: The remaining sections of [CUP05] consider specific families of properties and focus on specific strategies targeted at that kind of properties.

This section discusses existence theorems, *i.e.* theorems that assert the existence of at least one element satisfying a specific property. Then, it is necessary to prove the existence of such element or to provide a systematic procedure that constructs or selects such element.
Exercises: From this section of [CUP05], exercises 2, 5, 7 and 8.

10) [CUP05], Section *Uniqueness theorems*
Reading: Other theorems assert that there is at most one element satisfying a specific property. Uniqueness properties are often combined with existence: "there exists X such that $p(X)$ and it is unique".
Exercises: From this section of [CUP05], exercises 3 and 5.

11) [CUP05], Section *Equality of sets*
Reading: In this Section, properties on the equality of sets are considered. These properties state that two sets have exactly the same collection of elements. For example, this situation arises when we need to prove that two definitions of the same set are equivalent.
Exercises: From this section of [CUP05], exercises 1, 5, 7 and 9.

12) [CUP05], Section *Equality of numbers*
Reading: Counting proofs and many proofs in arithmetic and analysis deal with the equality of two numbers defined in two different ways. This section shows several examples of such proofs of equality.
Exercises: From this section of [CUP05], exercises 1, 3 and 5.

13) [CUP05], Section *Composite statements*
Reading: Finally, the statement to be proved could be a combination of several statements using logical operators, where each individual statement conforms to one of the patterns discussed in previous sections. This Section considers how to analyse these complex statements in order to use the proof strategies discussed so far.
Exercises: From this section of [CUP05], exercises 3, 4 and 5.

14) The paper [CAS00].
Reading: Theorems can also be proved with the use of graphical depictions, and not only in geometrical problems, but also in other fields. This paper shows several examples of these proofs and discusses when such a proof can be considered valid.
Exercises: Prove graphically that for any pair of natural numbers a and b, $a^2 + b^2 \leq (a + b)^2$. Also, prove graphically that there exists an integer k such that for all $x \geq k$, $x \geq \log_2(x)$, where $\log_2$ is the logarithm for base 2.

15) [CUP05], Section *Collection of Proofs*.
Reading: This section shows several examples of incorrect proofs, *i.e.* proofs that allegedly certify the validity of a statement but have errors like ignoring special cases or incurring in logical fallacies. Learning to detect errors in a proof is as important as being able to create a correct proof. Section 5 of

[CLA09] also showed several common pitfalls that may appear when using several proof strategies.
Exercises: Find the errors in the alleged proofs.

16) [CUP05], Diagram in the last page and [CLA09], Closing table of Section 5.
Reading: The final step is a set of guidelines to decide which proof strategy is most suitable for our theorems. In the real world, there is no silver bullet that will tell us which is the best proof strategy. However, there are several intuitions that can reveal suitable proof strategies, based on the structure of the property to be proved. The diagram in [CUP05] and the table in [CLA09] summarize those rules of thumb which have been discussed in the previous chapters.
Exercises: From [CLA09], self-assessment exercises 1-4. From [CUP05], review exercises 1, 2, 4, 5, 6, 10, 18, 21, 22, 23, 25, 33, 34, 37, 38 and 39. Finally, Section *Exercises without solutions* in [CUP05] are good candidates for discussion in the forum.

3.5. Further reading

- [CUP05] provides an additional section on proofs related to limits of functions and sequences. Also, each Section provides more exercises in addition to those discussed in this reading guide.

- There are many available resources that study *logical fallacies* in detail. For example, the following web resources provide a starting point to identify incorrect arguments.

Wikipedia. *List of fallacies* `http://en.wikipedia.org/wiki/List_of_fallacies` Last retrieved on 20010/01/01.
Fallacy files portal. `http://www.fallacyfiles.org/` Last retrieved on 20010/01/01.
Michael C. Labossiere (1996) *Tutorial on fallacies* `http://www.nizkor.org/features/fallacies/index.html` Last retrieved on 20010/01/01.
Stephen Downe (2001). *Stephen's Guide to the Logical Fallacies.* `http://onegoodmove.org/fallacy/toc.htm` Last retrieved on 20010/01/01.
Logical fallacies portal. `http://www.logicalfallacies.info` Last retrieved on 20010/01/01.

Fallacies in everyday life

Advertising, political statements and articles of opinion, specially in controversial topics, can contain fallacies that seem to support the author's point of view.

4. Applications of formal proof

4.1. Overview

This section describes several research fields in computing where formal proof can be useful. The goal of this section is three-fold. First, it is important to understand the type of problems where formal proof can be applied. Second, we will get to analyse several examples of proofs which are more complex than textbook examples. Finally, we can learn different conventions for formal proofs from different fields, *e.g.* notation, terminology, proof style, . . . The list of research fields that will be considered is the following:

- Concurrent and distributed systems.
- Communication and cryptographic protocols.
- Formal verification of safety-critical systems.
- Computational complexity theory.
- Security and cryptography.

4.2. Required reading

[CLA09] **R. Clarisó (2009).** *Formal Proofs: Understanding, writing and evaluating proofs*. Editorial UOC.

- Section 1 of [CLA09] presents a short overview of several research problems within computer science and information systems research that involve formal proof. This list can be used as a starting point before moving forward. Notice that formal proof is not the best fit for every discipline: some problems should be evaluated empirically rather than being formally proved. Formal proof is more adequate for problems where there is a need for *certainty* and it is possible to eliminate *ambiguity*, *e.g.* mathematical properties, solutions to abstract problems or safety-critical systems.

- Section 8 –*Evaluating formal proofs*– of [CLA09] provides an evaluation framework to critically evaluate a formal proof. This checklist will be the reference guide when studying the formal proofs used in real research papers.

- Section 7 –*Examples of proofs in IS and computing*– of [CLA09] provides a small list of research papers that use formal proofs as part of their research.

The remainder of the section is devoted to each of the research fields in the previous list. For each field, we provide a general overview of the field and the type of problems that are studied. This introduction attempts to identify the basic concepts and notation used in the field, in order to facilitate the comprehension of research papers. Whenever possible, we also provide a broad survey of the research fields that presents the relevant problems and results. Finally, we provide some examples of real research papers from those fields that base some or all of their contributions on formal proof. For each paper, a short summary is provided to provide context for the proof being used in the paper and to highlight the relevant aspects.

4.3. Reading tips

- The goal of this section of the course is not to cover all the research fields: you should select *only two* of the research fields covered in this section. You should choose the one which is closest to your research interest and any other from the list. In case you cannot decide which two to focus on, the recommended topics are "Concurrent and distributed systems" and "Formal verification of safety-critical systems".

- Keep the checklist in Section 8 of [CLA09] close while you are reading research papers. Evaluate each aspect of the checklist in every formal proof: this constitutes the exercises of this part of the course.

- Before reading proofs from a specific field, make sure that you understand the basic concepts and notation that appear in the proof. Refer to the introduction to each field provided in this guide and the surveys for further information.

4.4. Formal Verification of Safety-Critical Systems

4.4.1. Overview

The process of deciding whether a program or hardware system (circuit, processor, ...) satisfies its specification is called *formal verification*. The term "formal" is used because, in this process, both the implementation and specification are expressed in a formal notation that assigns them a precise meaning and, then, they can be compared using sound reasoning techniques. Usually, this type of formal process is used in systems whose correct operation is *critical* due to the potential impact of an error in the lives of their users, *e.g.* systems in the medicine, automotive, aerospace or military industries.

Formal Verification vs Validation

Verification checks whether a system satisfies a specification, *i.e.* is the system *right*? Meanwhile, validation checks whether the system being built matches the intent of the designer, *i.e.* is this the *right* system?

A typical paper on formal verification needs to describe (1) the specification, (2) the implementation, (3) the semantics of the selected formalism, (4) the translation of the specification and implementation into that formalism, (5) the algorithm used to check the specification and (6) the results of this analysis.

The specification can take several forms: a contract in the form of a precondition and postcondition, the description of the protocol that is being implemented, a mathematical formula describing the expected result, a state diagram illustrating the expected changes in the behaviour of the system,... Usually the specification describes the dynamic evolution of the system and, in that case, the property being described can be classified into two categories:

- *Safety* properties describe something bad that should never happen in our system, *i.e.* a set of error configurations that should not be reached by our system. For instance, a typical safety property is the lack of runtime errors: "the program should not execute a division by zero, access a null pointer, access an array outside its bounds or throw a runtime exception". Another example of a safety property in the case of a traffic light controller would be "the car and pedestrian lights should never be both green simultaneously".

- *Liveness* properties describe something good that should happen in our system, *i.e.* a set of correct configurations that should eventually be reached, or that should be reached infinitely often. For example, some examples of liveness properties are: "the program should terminate after a finite number of steps" or "both the car and pedestrian light should become green infinitely often".

The sheer size of current software and hardware systems makes manual formal verification impractical in almost all scenarios. It would be possible, for instance, to verify by hand very high-level abstractions of a real system. However, for larger systems, software tools are required to automate the deduction process.

The selection of a tool is dictated by the type of property to be checked and the type of system being checked: checking that a concurrent program terminates is not the same as checking that the clock cycle time of circuit is sufficiently long. In this sense, the characteristics of the specification and implementation influence the choice of a formalism, and each formalism supports a range of tools for reasoning purposes:

- When the formalism is a logic of some sort, the tool is called a *theorem prover*.
- If the formalism is a *temporal logic*, *i.e.* a logic notation used to describe properties about time, sequences of events and dynamic behaviour, then the tool is called a *model checker*.
- If the formalism is an optimization problem or a system of constraints, the tool is called a *constraint solver*. Some widely used families of constraints problems are SAT (satisfiability of a logical formula in conjunctive normal

form) and linear programming (maximising a linear expression that satisfies a system of linear inequalities such as $x + 2y \geq 7$).

- *Static analysers* infer information about the dynamic behaviour of a system from its structure, that is, without trying to simulate its execution. Static analysis may be able to prove simple dynamic properties but, in general, it is unable to prove every dynamic property. However, as static analysers tend to be much more efficient, the potential lack of a conclusive answer is considered a reasonable trade-off.

In any case, verification is a computationally complex problem: the time required to verify a property is usually at least exponential in terms of the size of the system. Furthermore, several properties like termination are *undecidable*: it is impossible to write a procedure able to check them in finite time for some system. Even with approximated algorithms and accepting that the tool may not terminate in some cases, most methods have scalability problems and become unusable when the size of the implementation to be checked grows.

Finally, hardware and software systems have peculiarities that may affect the way in which their verification is addressed. For example, in software systems the flow of execution is sequential (except for concurrent/distributed systems) and the control flow plays an important role (conditionals, loops, jumps, . . .). On the other hand, hardware is inherently concurrent (logic gates operate in parallel) while the computation is data-flow oriented. Also, the properties of interest are different in software and hardware systems, *e.g.* termination does not make sense in many hardware contexts. Consequently, although theorem provers and solvers are general purpose, the verification tools built upon them may be oriented towards hardware, software or *embedded* systems (which combine software and hardware components).

4.4.2. Required reading

Surveys and introductory papers

[CWI96] **E. C. Clarke, M. Wing (1996).** *Formal Methods: State of the Art and Future Directions*. ACM Computing Surveys, 28(4):626–643, ACM.

[HJ+08] **M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, T. Margaria, T (2008).** *Software engineering and formal methods*. Communications of the ACM, 51(9):54–59, ACM.

[CG+05] **E. C. Clarke, A. Gupta, H. Jain, H. Veith (2005).** *Model Checking: Back and Forth between Hardware and Software*. VSTTE 2005, LNCS 4171, pp. 251-255, Springer.

[WAN04] **F. Wang (2004).** *Formal Verification of Timed Systems: A Survey and Perspective*. Proceedings of the IEEE, 92(8):1283–1305, IEEE Press.

Papers using formal proofs

[KH+00] **S. King, J. Hammond, R. Chapman, A. Pryor (2000).** *Is Proof More Cost-Effective Than Testing?*. IEEE Transactions on Software Engineering, 26(8):675–686, IEEE CS Press.

[LER09] **X. Leroy (2009).** *Formal Verification of a Realistic Compiler*. Communications of the ACM, 52(7):107–115,ACM.

[ALE95] **M. Aagaard, M. Leeser (1995).** *Verifying a Logic-Synthesis Algorithm and Implementation: A Case Study in Software Verification*. IEEE Transactions on Software Engineering, 21(10):822–834, IEEE CS Press.

[DCB04] **S. Dajani-Brown, D. Cofer, A. Bouali (2004).** *Formal Verification of an Avionics Sensor Voter Using SCADE*. FORMATS 2004, LNCS 3253, pp. 5-20, Springer.

4.4.3. Detailed reading guide

- [CWI96] provides a general view of formal verification approaches in several disciplines, collectively described as formal methods. Several tools, approaches, and examples of successful verification are introduced.

- [HJ+08] discusses the role of formal verification in the software development process, arguing that the use of formal methods can improve reliability of software. Several promising trends in software verification are discussed.

- [CG+05] provides a brief high-level overview of model checking, *i.e.* the verification of dynamic properties of a system. The survey [WAN04] provides a more detailed discussion of this topic. However, due to its length and density, it is better used as a reference material.

- [KH+00] combines formal proof with a quantitative analysis of the advantages achieved by formal methods-based development over conventional development strategies. The system under study is a safety-critical system advising on the safety of flying helicopters from a naval vessel.

- [LER09] considers the verification of a compiler for a subset of the C language. The goal is ensuring that the code generated by the compiler preserves the semantics of the original code. The paper defines the notions of "correctness" that will be checked in the compiler, outlines the overall structure of the compiler and provides an overview of the verification process using the interactive theorem prover Coq.

- [ALE95] addresses the verification of a logic synthesis algorithm, a procedure which reduces the size of a boolean function to improve the performance of its physical implementation. Obviously, the optimised circuit should compute the same output as the original circuit. The verification of this property uses the theorem prove NuPRL.

- [DCB04] addresses the verification of a sensor voter algorithm, a safety redundancy mechanism where multiple sensors are used to measure each sample in order to detect potential sensor defects. The correct operation of the protocol is analysed used the SCADE tool.

4.5. Concurrent and distributed systems

4.5.1. Overview

A *concurrent* system is any system containing multiple tasks that are executed simultaneously. These tasks are not executed independently and they may interact freely among them, *e.g.* exchanging intermediate results, coordinating among them, sharing resources or even competing for mutual resources. The set of tasks may be executing in a single-core processor (time-shared among all tasks), a multi-core processor or multiple processors that may communicate using a shared memory or a network that allows communication through messages. In the latter case, we talk about *distributed systems* to emphasize that there is communication taking place and that tasks may not be physically on the same computer. Usually, tasks are described using terms like “thread” or “process” for concurrent systems, and with the terms “node” or “processor” for distributed systems.

In a similar way, a concurrent/distributed algorithm is the description of a set of *N* tasks such that their concurrent/distributed execution produces a desired result. The description is often provided in terms of pseudocode and usually the code for all tasks is equal, except for a unique identifier given to each task. A distributed algorithm may be replicating the computation of an equivalent sequential algorithm, for example (a) to provide tolerance to failures, (b) to improve performance or split workload by dividing a computation among multiple nodes or (c) to allow concurrent access to a shared resource. In these scenarios, the correctness of the distributed algorithm will demand that its execution provides identical results to a sequential execution. That is, the concurrent system should avoid having intermediate results from one task disrupt another. One common mechanism to achieve this goal is the definition of *critical sections*, *e.g.* fragments of the task where its execution should occur in *mutual exclusion*, that is, as if the task was executed individually in a sequential system.

Nevertheless, a distributed algorithm may also attempt to achieve communication and interaction among the set of nodes. An example of such algorithms is *leader election*, where a set of identical peers must designate a leader among them. This algorithm belongs to a more general family of *consensus* algorithms where different nodes must reach a common agreement.

The correctness of concurrent/distributed systems considers some problems that do not appear in sequential systems. For example, the system as a whole should not *starve*, *i.e.* reach a configuration where each task is waiting for resources allocated to other tasks. This type of configuration is called *deadlock* and it can be ensured through careful policies for allocating and releasing resources. Furthermore, the system should *terminate* or *converge* by producing a result after a finite sequence of steps, *e.g.* if the goal is reaching an agreement, such agreement must be reached after a finite sequence of steps.

Regarding distributed systems, another correctness concept which is introduced is that of *fault-tolerance*: the resilience of the system as a whole to the incorrect operation of one or more of its nodes. Errors may consist, for example, in the failure to send/receive a message, sending unsolicited messages or computing an incorrect result in one step of the algorithm. Specific patterns of error models may receive a name, *e.g.* Byzantine errors usually refer to unreliable communications among nodes. Usually, each algorithm is classified according to the type of errors and the number of erroneous nodes it can withstand.

Byzantine generals problem

The Byzantine generals problem is described in detail in one of the required readings of this section, [TSA92].

Notice that distributed algorithms deal with errors caused by crashes, malfunctions or faulty communication channels, but they *usually* do not consider malicious nodes in their analysis. This type of problem is studied by *cryptographic protocols*, that are studied in detail in Section 4.7.

Finally, another notion which often appears in the context of distributed algorithms is their *cost*: as communication among nodes may be slow, algorithms are usually oriented towards exchanging the minimum amount of information among nodes. It is common to provide formal proofs of correctness followed by formal proofs of the lower/upper bounds on the number of messages required to reach the result.

4.5.2. Required reading

Surveys and introductory papers

[FME03] **M. J. Fischer, M. Merritt (2003).)** *Appraising two decades of distributed computing theory research*. Distributed Computing, 16: 239-247, Springer.

[TSA92] **J. Turek, D. Sasha (1992).** *The Many Faces of Consensus in Distributed Systems*. Computer, 25(6):8–17, IEEE Computer Society Press.

[THO98] **A. Thomasian (1998).** *Concurrency control: methods, performance, and analysis*. ACM Computing Surveys, 30(1):70–119, ACM.

Papers using formal proofs

[LAM74] **L. Lamport (1974).** *A new solution to Dijkstra's concurrent programming problem*. Communications of the ACM, 17(8):453–455, ACM.

[LAM78] **L. Lamport (1978).** *Time, Clocks, and the Ordering of Events in a Distributed System*. Communications of the ACM, 21(7):558–565, ACM.

[FLP85] **M. J. Fischer, N. A. Lynch, M. S. Paterson (1985).** *Impossibility of Distributed Consensus with One Faulty Process*. Journal of the ACM, 32(2):374–382, ACM.

[GLY02] **S. Gilbert, N. Lynch (2002).** *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News, 33(2):51–59, ACM.

[SM+03] **I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, H. Balakrishnan. (2003)** *Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications*. IEEE/ACM Transactions on Networking, 11(1):17–32, IEEE Press.

4.5.3. Detailed reading guide

- [FME03] provides a general description of the state of the art and theoretical problems in the field of distributed computing.
- [TSA92] focuses on a specific family of distributed algorithms, consensus algorithms, and discusses different solutions to the problem in different settings. For each algorithm, we need to ensure termination (eventually, each node chooses a value), agreement (upon termination, all values chosen by nodes are equal) and validity (if all nodes entered the consensus with the same value, then consensus will be reached on that value).
- [THO98] discusses algorithms for ensuring that the concurrent execution of a set of tasks produces an equivalent result to their sequential execution. Again, this survey is very extensive and it is useful mainly as a reference. In any case, it is recommended to read at least sections 1 and 2 to understand the basic problems related to concurrency control.
- [LAM74] originally introduced the well-known *bakery* algorithm to achieve mutual exclusion among a set of concurrent tasks. The paper describes the algorithm using pseudocode and follows up with a formal proof of its correctness.
- [LAM78] presents a method that allows the definition of an order among all the events that occur in a distributed system. Its purpose is being able to decide, for any pair of events *a* and *b* which may happen in different nodes, whether *a* happens before *b* or vice versa.

- [FLP85] reaches a theoretical result: consensus algorithms cannot ensure termination in distributed systems where communication among nodes is *asynchronous* (message delivery may take an arbitrary amount of time) *and* there is at least one faulty node that can crash at any time.

- [GLY02] discusses a theoretical result about web services, which is somehow related to [FLP85]: a web service cannot offer three notions which are desirable for web services (consistency, availability and partition-tolerance). The paper defines these notions and formally proves the impossibility of ensuring all of them simultaneously.

- [SM+03] presents a distributed look-up system, Chord, addressed at answering queries about which peer in the network has a desired data item. The paper provides several proofs to ensure that the number of lookup operations required to reach the right peer is proportional to log(n) where n is the number of nodes in the network. As some aspects of the algorithm involve using random number generators, the proofs use probability to reach conclusions "with high probability". In these proofs, it is generally assumed that random number generators are "perfect" in the sense that they achieve perfect randomness.

4.6. Security and Cryptographic Algorithms

4.6.1. Overview

Information security is a broad research field encompassing the access, transmission, modification and storage of information. The underlying concept is preventing malicious actors (*attackers*) from breaching the *confidentiality* (secrecy of designated information), *integrity* (avoiding or detecting unauthorised data manipulation) or *availability* (allowing access to authorised users) of the information. This idea can be applied to many different areas in computing that deal with information and communication, *e.g.* networks, databases, operating systems, . . .

Security is achieved through the use of mathematical algorithms devised for hiding information, known as *cryptographic* algorithms. The classical examples of cryptographic algorithms are encryption algorithms: a procedure that given a message (*plaintext*) and a *key*, generates a *ciphertext* from which it is not possible to extract the original plaintext without the proper key. Another typical cryptographic device is the *one-way function* or *hash* function, which can be used to generate message digests that act as a proof of the integrity of a message.

One-way function

A one-way function is a function where (a) the output can be computed from the input efficiently and (b) inverting the function is computationally complex: given an output, it is hard to compute an input that produces that output.

Research on information security is very dependent on proofs to demonstrate the validity of its claims. A new encryption algorithm cannot be considered secure unless it is possible to formally prove its immunity to the attacks relevant to its operation. Also, contrary to fields like formal verification, it is not uncommon to find pen-and-paper proofs for security related problems.

A typical paper on (theoretical) information security needs to describe the setting, the cryptographic algorithms being used and the attacker model, *i.e.* the assumed capabilities of a potential attacker. Then, either the paper proves that the system is secure to the attacks under consideration or it describes an unknown vulnerability that can be abused by an attacker.

4.6.2. Required reading

Surveys and introductory papers

[DHE79] **W. Diffie, M. E. Hellman (1979).** *Privacy and Authentication: An Introduction to Cryptography*. Proceedings of the IEEE, 67(2):397–428, IEEE Press.

[GOL05] **O. Goldreich (2005).** *Foundations of Cryptography - A primer*. Foundations and Trends in Theoretical Computer Science, 1(1):1-116, Now Publishers.

[PAK99] **F. A. P. Petitcolas, R. J. Anderson, M. G. Kuhn (1999).** *Information hiding - a survey*. Proceedings of the IEEE, 87(7):1062–1078, 1999.

Papers using formal proofs

[CAM00] **J. Camenisch (2000).** *Efficient Anonymous Fingerprinting with Group Signatures*. ASIACRYPT2000, LNCS 1976, pp. 415-428, Springer.

[ACP05] **J. Andronik, B. Chetali, C. Paulin-Mohring (2005).** *Formal Verification of Security Properties of Smart Card Embedded Source Code*. FM 2005, LNCS 3582, pp. 302–317,Springer.

[GWO00] **D. Wagner, I. Goldberg (2000).** *Proofs of Security for the Unix Password Hashing Algorithm*. ASIACRYPT 2000, LNCS 1976, pp. 560–572, Springer.

[BMO84] **R. S. Boyer, J. S. Moore (1984).** *Proof Checking the RSA Public Key Encryption Algorithm*. American Mathematical Monthly, 91(3):181–189, AMS.

4.6.3. Detailed reading guide

- [DHE79] presents fundamental concepts in cryptography and information security which have become part of the folklore of the field: encryption, public and private key cryptography, . . . However, as a survey, this paper is extremely outdated: many new cryptographic algorithms, protocols, attacks and problems have been proposed since its publication.

- [PAK99] introduces a specific subfield within information security: information hiding, the embedding of a secret within a message. The paper discusses the basic concepts and applications of information hiding. Again, the paper is too dated to provide a clear picture of the state-of-the-art in the area but it is a useful summary of basic concepts.

- [GOL05] is a very extensive introduction to cryptography in general. It is recommended to use it merely as a reference rather than reading it in depth from start-to-end.

- [CAM00] discusses a method for computing fingerprints to identify products sold by a merchant. In that way, if unauthorised copies of a product are discovered, it is possible to identify the buyer who created the copies. Moreover, the paper characterises the security requirements of this signature method: it should enable the sellers to identify malicious buyers, while protecting buyers from being falsely accused. The proof of these properties is developed using the pen-and-paper style.

- [ACP05] considers the verification of security properties in an operating system module embedded in a smartcard. More precisely, the smartcard can be powered down at any time and this feature could be used by an attacker to leave memory in an inconsistent state. The presented approach annotates the model with correctness conditions and then performs an analysis of the source code to ensure that those conditions are met. The Coq interactive theorem prover is used to reason on correctness conditions and source code behavior.

- [GWO00] analyses the security of the hashing algorithm used to store passwords in the Unix operating system, *i.e.* it should be hard to recover a password from a hash, even if the attacker is aware of the hashing algorithm. The paper provides a formal definition of this security notion and proves it in the pen-and-paper style.

- [BMO84] studies the correctness of the RSA encryption algorithm, a widely used *public key* encryption algorithm. More precisely, it uses a theorem to verify the invertibility of the algorithm, *i.e.* that it is possible to recover information encrypted with the public key by using the private key.

4.7. Communication and Cryptographic Protocols

4.7.1. Overview

A *protocol* is a collection of rules that regulates the interaction among several parties. A communication protocol governs the exchange of information among several computer systems. For example, a protocol may dictate which party can start the communication, the appropriate sequence of messages and responses, how to acknowledge the correct reception of a message or how to deal with faulty error channels.

On the other hand, a *cryptographic protocol* is a collection of rules and cryptographic procedures to achieve a desired security property, even in the presence of an *attacker*, *i.e.* malicious individuals or systems interested in breaking the security property for their own advantage. This goal is achieved through the use of cryptographic methods: using keys and encryption algorithms to keep the contents of a message secret, using timestamps to identify when each message is exchanged, using one-way digest functions to detect data tampering, using random numbers (*nonces*) to detect the reuse of old messages, . . .

Some possible goals of a cryptographic protocol may be *authentication* (establishing the identity of one of the parties), *key agreement* (choosing a key that will be used in further communications) or *non-repudiation* (ensuring that no party can claim that a message was not sent or received). Depending on the property, the attacker or attackers will have a different goal: successfully repudiate a message, become authenticated as somebody else, learn the keys used between two parties, . . . Also, the assumed capabilities of the attacker may vary: he may be able to resend old messages (*replay attack*), intercept and alter all communications among the parties (*man-in-the-middle*), . . .

A formal proof is required to ensure that the protocol has no holes that can be abused by an attacker. The proof requires a precise description of the protocol and usually a careful case-by-case analysis of all possible scenarios. As protocols have become more complex over time, it is more common to use automated tools like theorem provers or model checkers to establish that a protocol is secure.

4.7.2. Required reading

Surveys and introductory papers

[KIN91] **P. W. King (1991).** *Formalization of protocol engineering concepts*. IEEE Transactions on Computers, 40(4):387–403, IEEE Press.

[LIE93] **A. Liebel**. *Authentication in Distributed Systems: A Bibliography*. ACM SIGOPS Operating Systems Review, 27(4):31–41, ACM.

[MEA03] **C. A. Meadows (2003).** *Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends*. IEEE journal on Selected Areas in Communication, 21(1):44-54, IEEE Press.

Papers using formal proofs

[DLM82] **R. A. DeMillo, N. A. Lynch, M. J. Merritt (1982).** *Cryptographic protocols*. STOC'82, pp.383–400, ACM.

[BOL96] **D. Bolignano (1996).** *An Approach to the Formal Verification of Cryptographic Protocols*. CCS'96, pp. 106–118, ACM.

[BPA97] **G. Bella, L. C. Paulson (1997).** *Using Isabelle to Prove Properties of the Kerberos Authentication System*. DIMACS Workshop on Design and Formal Verification of Security Protocols.

[DMT00] **G. Denker, J. Meseguer, C. Talcot (2000).** *Formal Specification and Analysis of Active Networks and Communication Protocols: The Maude Experience*. DARPA Information Survivability Conference & Exposition, 1:251–265.

4.7.3. Detailed reading guide

- [KIN91] focuses on the definition of communication protocols using different formal notations. The paper does not focus on security concerns.

- [LIE93] and [MEA03] are surveys on the formal verification of cryptographic protocols, discussing the relevant problems in the field and the different approximations that have been used in the literature. It is recommended to review them in chronological order, as they illustrate the progress of the state of the art and the techniques being used increase in complexity over time.

- [DLM82] and [BOL96] offer two examples of a manual proof of the security of a protocol. Both papers first discuss their model of the protocol and the attackers and, then, they proceed to verify specific cryptographic protocols in detail.

- On the other hand, [BPA97] and [DMT00] present mechanized proofs of cryptographic protocols, using two different tools: the theorem prover Isabelle and the rewriting system Maude. First, they discuss how to encode their model of the protocol and the attackers in the formalism used by the tool. After that, the tools automate part of the reasoning process. Notice that creating an abstract but complete formalisation of the protocol which is faithful to the original is the core of the problem.

4.8. Complexity theory

4.8.1. Overview

Complexity theory is a discipline within theoretical computer science that studies the inherent difficulty of problems, measured in terms of the computational resources required to solve them. The resources being considered are the *memory usage* and *execution time* of the most efficient program that solves the problem. An algorithm may work better for some inputs than others, so when measuring its resource usage we will have to consider its best case, average case or worst case. We usually focus only on the latter, the cost in the worst case.

Execution time is abstracted as the "number of operations to be executed" to avoid conclusions that are dependent on a specific technology. Depending on the context, our understanding of what operations are counted may vary. For example, database algorithms may consider the number of disk read/write operations and distributed algorithms may evaluate the number of messages being exchanged. Usually, if no context is provided, we are implicitly counting the number of assignments and operators (logic, arithmetic and relational).

On the other hand, memory usage counts the storage space used by the intermediate data required to compute the result. Storage is measured in terms of the "number of bits", again to be independent from technological choices.

Resource usage grows with the size of the input problem. For example, it will take longer to sort 1.000.000 words in lexicographic order than to sort 10. For this reason, the complexity of a problem is measured as the growth rate of resources with respect to the input. We are not interested in the specific formula (*e.g.* $3n^3 + 17n - 1$), but on the order of magnitude of the growth (*e.g.* proportional to n^3 or *cubic*). Some other examples of typical growth functions are *logarithmic* ($\log n$), *linear* (n), *quasilinear* ($n \log n$) and *quadratic* (n^2).

Big O notation

This mathematical notation describes the growth rate of a function and it is often used in complexity theory and algorithmics. For example, a program with execution time $O(n^3)$ requires a number of operations that is, at most, proportional to n^3.

4.8.2. Complexity classes

In addition to identifying bounds on resource usage, a second goal of complexity theory is comparing the relative difficulty of different problems. Thus, complexity theory describes *families* of problems with the same difficulty and the relationships between these families. Each family corresponds to a different category of growth rates, for example *polynomial* growth, *i.e.* $O(n^k)$ for some constant k, or *exponential* growth, *i.e.* $O(k^n)$ for some constant $k > 1$ or $O(n^n)$ or $O(n!)$.

Polynomial vs Exponential

Deciding whether an unsorted list contains a given element can be done in polynomial time (in fact, linear time). However, deciding whether checkmate is inevitable from a given chess position requires exponential time: the number of possible moves is exponential with respect to the size of the board.

Exponential growth can rapidly lead to prohibitive costs in terms of memory usage or execution time. For instance, for a large input, the execution time of the program could take longer than the life of the universe! Problems

Table 2: Complexity classes

Cost	Complexity class
Polynomial execution time	PTIME or simply P
Polynomial memory usage	PSPACE
Exponential execution time	EXPTIME or simply EXP
Exponential memory usage	EXPSPACE

where the most efficient procedure to find the solution has an exponential cost are called *intractable*: they cannot be solved in practice beyond small inputs. Therefore, it is important to discern whether a problem is tractable or intractable.

Remarkably, there is a large family of problems whose tractability is still unknown. First, there is no known polynomial solution for any of these problems. On the other hand, there is no proof that it cannot exist. Furthermore, those problems have an additional trait: for each solution it is possible to generate a *certificate* with a polynomial size in terms of the input, from which it is possible to check whether the solution is correct in polynomial time or space. These families of problems are called NP (non-deterministic polynomial time) and NP-SPACE (non-deterministic polynomial space) respectively.

Let us consider, for instance, the PARTITION problem: given a set of integers, decide whether it can be partitioned into two disjoint subsets with an equal total sum. For example, the set {3, - 1,2,7,1} can be partitioned into {3,2,1} and {7, - 1}, which both have a total sum of 6. A set of n integers can be partitioned in 2^n different ways and there is no known way of computing the correct partition efficiently. However, checking if a partition is correct can be done in polynomial time, by simply adding the numbers in both partitions: a solution to PARTITION is also its certificate. Therefore, PARTITION belongs to NP.

4.8.3. Decision problems

Problems can be classified into three categories according to the expected solution. A *decision* problem attempts to answer a "yes or no" question. On the other hand, a *computation* problem asks for any solution under some constraints. Finally, an *optimisation* problem asks for the optimal solution according to some criteria. Usually, it is possible to define a decision, computation and optimisation version for the same problem, each of which is harder than the previous one.

Complexity theory usually studies decision problems because of two reasons. First, it is the easiest version of a problem, *i.e.* if a decision problem is EXP, the optimisation version will be at least EXP. Moreover, this simplicity allows a convenient representation of the problem: the inputs can be divided into those whose answer is "yes" and those whose answer is "no".

Table 3: Example of problem types.

Problem type	Statement
Decision	Starting from the initial chess board, can the white player win in five moves or less?
Computation	Starting from the initial chess board, compute a sequence of four moves where the white player wins.
Optimisation	Starting from the initial chess board, compute the shortest sequence of moves where the white player wins.

4.8.4. Reductions

In order to classify the complexity of a problem A, it is useful to identify an upper bound for its complexity, *i.e.* a problem B which is as hard as A or harder. For example, if we consider the complexity of "computing the area of a triangle", the problem of "computing the area of a regular polygon" may be used as an upper bound, as our original problem is just a special case. Therefore, if we know that our hard problem B can be solved using a specific amount of memory or a given execution time, these are upper bounds for any solution to our simple problem A.

Sometimes, the relationship between the two problems is not so direct. It might happen that the solution of one problem leads to a solution of the other. For example, consider the problem PARTITION and SUBSET-SUM, we can see that PARTITION is at most as hard as SUBSET-SUM:

PARTITION: Given a set of integers, decide if it can be divided into two disjoint subsets with an equal total sum.
Sample input: {1,2,3}.
Answer: Yes, it can be partitioned as {1,2} and {3}.

SUBSET-SUM: Given a set of integers and a value k, decide if there is a subset whose total sum is equal to k.
Sample input: {1,2,3,4} and $k = 8$.
Answer: Yes, the subset {1,3,4} has a sum of 8.

If we have a program P that can decide SUBSET-SUM, it is very simple to build a program that solves PARTITION using P. If the total sum of the elements in the set is N, we simply call P providing the same set and the value $\frac{N}{2}$. If P answers "yes", it means that a subset of the original set has a sum of $\frac{N}{2}$, but obviously the sum of the remaining elements will have to be $\frac{N}{2}$ as we chose N to be the total sum. Therefore, the answer to PARTITION is also "yes". Similarly, we can see that if P answers "no", it means that the original set cannot be partitioned.

This relationship among two decision problems A and B in which A is at most as hard as B is called a *reduction* and it is noted as $A \leq_p B$, *e.g.* PARTITION $\leq_p$ SUBSET-SUM. To conclude that $A \leq_p B$, it is necessary to define a mapping from the inputs of a problem A to the inputs of a problem B, just as we have done. This mapping can be defined as an algorithm that, given an input of problem *A*, generates an input of problem *B*. The program computing this mapping should satisfy two conditions:

- The answer for the input of *A* will be the same as the answer for the input of *B*, *i.e.* if the answer for *A* is "yes", the answer for *B* will be "yes" and, similarly, if the answer for *A* is "no", the answer for *B* will be "no".

- Computing the input for *B* should need resources which are at most polynomial in terms of the size of the input for *A*.

4.8.5. Required reading

Surveys and introductory papers

[Coo93] **S. A. Cook (1993).** *An Overview of Computational Complexity*. Communications of the ACM, 26(6):400–408, ACM.

[For09] **L. Fortnow (2009).** *The Status of the P vs NP Problem*. Communications of the ACM, 52(9):78–86, ACM.

Papers using formal proofs

[Kar72] **R. M. Karp (1972).** *Reducibility among combinatorial problems*. Complexity of Computer Computations, pp. 85-103, Plenum.

[AKS04] **M. Agrawal, N. Kayal, N. Saxena (2004)**. *PRIMES is in P*. Annals of Mathematics, 160(2):781–793.

[DGP09] **C. Daskalakis, P. W. Goldberg, C. H. Papadimitriou (2009).** *The Complexity of Computing a Nash Equilibrium*. Communications of the ACM, 52(2):89–97, ACM.

4.8.6. Detailed reading guide

- [Coo93] provides a general description of the field of computational complexity, describing relevant open problems.

- [For09] describes the current status of one of the fundamental problems in the field of computational complexity: the relationship between complexity classes P and NP. Through this discussion, the paper provides a clear definition of the key concepts in the field of complexity theory and the consequences of a possible result regarding the equality of P and NP.

- [Kar72] is a landmark paper in the field which describes reduction relationships among several key NP problems and introduces the concept of *completeness*. A problem *p* is called complete within a complexity class *X* if any other problem within *X* can be reduced to *p*. Intuitively, complete problems are the hardest among their class, as solving them efficiently provides an efficient solution for all other problems in the same class.

- [AKS04] proves that the problem PRIMES (deciding whether an input number is a prime or not) can be solved in polynomial time. The solution consists on an algorithm that performs this test and a proof setting a polynomial upper bound to its execution time. Interestingly, this problem was considered open in [Coo93].

- The paper [DGP09] is focused on the complexity of a problem arising in game theory: the computation of a Nash equilibrium. This concept is introduced in the paper, and a manual proof of the complexity of its computation is provided.

Summary

This course has introduced the use formal proof and provided strategies for developing proofs which are applicable to a variety of disciplines. These techniques are useful in the comprehension of proofs made by others and also in the development of new proofs.

It is important to remember that the goal of a proof is the *communication* of a result, which should be reproducible by others. Therefore, it is important to ensure that our proofs are easy to understand by choosing the right vocabulary and notation, as well as exercising care in the style and writing of the proof.

The application of these strategies in real-world problems has also been studied in several research fields. As we have seen, each field has its own rules and preferences when it comes to proofs, for example in the use of mathematical notation versus natural language. Before preparing a proof, it is highly recommended to take into account the audience and the context.