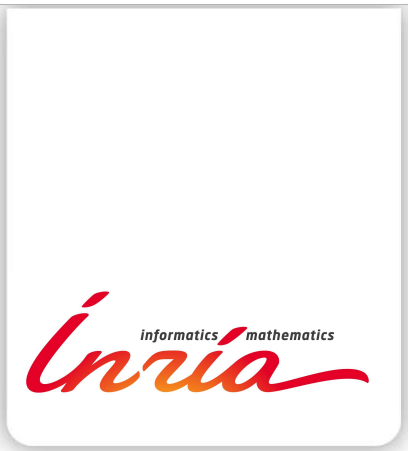


Implementation of Bourbaki's Elements of Mathematics in Coq: Part One Theory of Sets

José Grimm

RESEARCH REPORT
N° 6999
July 2009
Project-Team Marelle

ISSN 0249-6399 ISRN INRIA/RR--6999--FR+ENG



Implementation of Bourbaki's Elements of Mathematics in Coq: Part One Theory of Sets

José Grimm*

Project-Team Marelle

Research Report n° 6999 — version 6 — initial version July 2009 —
revised version October 2013 — 205 pages

Abstract: We believe that it is possible to put the whole work of Bourbaki into a computer. One of the objectives of the Gaia project concerns homological algebra (theory as well as algorithms); in a first step we want to implement all nine chapters of the book Algebra. But this requires a theory of sets (with axiom of choice, etc.) more powerful than what is provided by Ensembles; we have chosen the work of Carlos Simpson as basis. This reports lists and comments all definitions and theorems of the Chapter "Theory of Sets". The code (including almost all exercises) is available on the Web, under http://www-sop.inria.fr/marelle/gaia.
Version one was released in July 2009, version 2 in December 2009, version 3 in March 2010. Version 4 is based on the Coq ssreflect library. In version 5, released in December 2011, the "iff_eq" axiom has been withdraw, and the axiom of choice modified. Version 6 was relaesed in October 2013

Key-words: Gaia, Coq, Bourbaki, Formal Mathematics, Proofs, Sets

---

Work done in collaboration with Alban Quadrat, based on previous work of Carlos Simpson (CNRS, University of Nice-Sophia Antipolis), started when the author was in the Apics Team.

* Email: Jose.Grimm@inria.fr

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Implémentation des Éléments de mathématiques de Bourbaki en Coq, partie 1 Théorie des ensembles

Résumé : Nous pensons qu'il est possible de mettre dans un ordinateur l'ensemble de l'œuvre de Bourbaki. L'un des objectifs du projet Gaia concerne l'algèbre homologique (théorie et algorithmes); dans une première étape nous voulons implémenter les neuf chapitres du livre Algèbre. Au préalable, il faut implémenter la théorie des ensembles. Nous utilisons l'Assistant de Preuve Coq; les choix fondamentaux et axiomes sont ceux proposées par Carlos Simpson. Ce rapport liste et commente toutes les définitions et théorèmes du Chapitre théorie des ensembles. Presque tous les exercises ont été résolus. Le code est disponible sur le site Web http://www-sop.inria.fr/marelle/gaia.

Mots-clés : Gaia, Coq, Bourbaki, mathématiques formelles, preuves, ensembles

Chapter 1

Introduction

1.1 Objectives

Our objective (it will be called the *Bourbaki Project* in what follows) is to show that it is possible to implement the work of N. Bourbaki, "Éléments de Mathématiques"[3], into a computer, and we have chosen the Coq Proof Assistant, see [7, 1]. All references are given to the English version "Elements of Mathematics"[2], which is a translation of the French version (the only major difference is that Bourbaki uses an axiom for the ordered pair in the English version and a theorem in the French one). We start with the first book: theory of sets. It is divided into four chapters, the first one describes formal mathematics (logical connectors, quantifiers, axioms, theorems). Chapters II and III form the basis of the theory; they define sets, unions, intersections, functions, products, equivalences, orders, integers, cardinals, limits. The last chapter describes structures.

An example of structure is the notion of real vector space: it is defined on a set E, uses the set $\mathbb{R}$ of real numbers as auxiliary set, has some characterization (there are two laws on E, a zero, and a action of $\mathbb{R}$ over E), and has an axiom (the properties of the the laws, the action, the zero, etc.). A complete example of a structure is the *order*; given a set A, we have as characterization $s \in \mathfrak{P}(A \times A)$ and the axiom "$s \circ s = s$ and $s \cap s^{-1} = \Delta_A$". We shall see in the second part of this report that an ordering satisfies this axiom, but it is not clear if this kind of construction is adapted to more complicated structures (for instance a left module on a ring). Given two sets A and A′, with orderings s and s', we can define $\sigma(A, A', s, s')$, the set of increasing functions from A to A′. An element of this set is called a σ-morphism. In our implementation, the "set of functions f such that ..." does not exist[1]; we may consider the set of graphs of functions (this is well-defined), but we can also take another position: we really need σ to be a set if we try to do non-trivial set operations on it, for instance if we want to define a bijection between σ and σ′; these are non-obvious problems, dealt with by the theory of categories. There is however another practical problem; Bourbaki very often says: let E be an ordered set; this is a short-hand for a pair (A, s). Consider now a monoid $(A, +)$. Constructing an ordered monoid is trivial: the characterization is the product of the characterizations, and the axiom is the conjunction of the axioms. The ordered monoid could be $(A, (s, +))$. If f is a morphism for s, and $u \in A$, then the mapping $x \mapsto f(x + u)$ is a morphism for s, provided that + is compatible with s. If we want to convert this into a theorem in Coq, the easiest solution is to define an object X equivalent to $(A, (s, +))$, a way to extract $X' = (A, s)$ and $X'' = (A, +)$ from X, an operation s on A obtained from X or X′, and change the definition

[1] We changed the type of a function in V4, so that this set exists now

of σ: it should depend on X′ rather than on A and *s*. The compatibility condition is then a property of X, σ(X,Y) and σ(X′,Y) are essentially the same objects, if $f \in \sigma(X,Y)$ we can consider $f' = x \mapsto f(x+u)$, and show $f' \in \sigma(X,Y)$. From this we can deduce the mapping from σ(X′,Y) into σ(X′,Y) associated to $f \mapsto f'$.

1.2 Background

We started with the work of Carlos Simpson[2], who has implemented the Gabriel-Zisman localization of categories in a sequence of files: *set.v*, *func.v*, *ord.v*, *comb.v*, *cat.v*, and *gz.v*. Only the first three files in this list are useful for our project. The file *ord.z* contains a lot of interesting material, but if we want to closely follow Bourbaki, it is better to restart everything from scratch. The file *func.v* contains a lot of interesting constructions and theorems, that can be useful when dealing with categories. For instance, it allows us to define morphisms on the category of left modules over a ring. The previous discussion about structures and morphism explains why only half of this file is used.

This report is divided in two parts. The first part deals with implementation of Chapter II, "Theory of sets", and the second part with chapter III, "Ordered sets, cardinals; integers" of [2] Each of the six sections of Bourbaki gives a chapter in this report (we use the same titles as in Bourbaki) but we start with the description of the two files *set.v* and *func.v* by Carlos Simpson (it is a sequence of modules). Their content covers most of Sections 1 and 2 ("Collectivizing relations" and "Ordered pairs").

1.3 Introduction to Coq

The proof assistant Coq is a system in which you can define objects, assume some properties (axioms), and prove some other properties (theorems); there is an interpreter (that interprets sentences one after the other), and a compiler that checks a whole file and saves the definitions, axioms, theorems and proofs in a fast loadable binary file. Here is an example of a definition and a theorem.

```
Definition union2 (x y : Set) := union (doubleton x y).
Lemma union2_or : forall x y a, inc a (union2 x y) -> inc a x \/ inc a y.
Proof. ... Qed.
```

In Coq, every object has a type; for instance `doubleton` is of type `Set → Set → Set`, which means that it is a function of two arguments of type `Set` that returns an object of type `Set`, and `union` is a function of one argument of type `Set` that returns an object of type `Set`. Thus, the expression '`union (doubleton x y)`' is well-typed if and only if *x* and *y* are of type `Set`, and this object is of type `Set`. We define '`union2 x y`' to be this expression. In the definition we may indicate the type of arguments and return value, or omit them if Coq can deduce it (in most cases, type annotations are omitted).

The theorem says: for all *x*, *y* and *a* (of type `Set`) if $a \in x \cup y$ then $a \in x$ or $a \in y$. We give here different variants of the proof of the theorem:

```
ir. unfold union2 in H. pose (union_exists H).
nin e. xd. pose (doubleton_or H1).
```

[2] http://math.unice.fr/~carlos/themes/verif.html

```
nin o. rewrite H2 in H0. intuition.
rewrite H2 in H0; intuition.
```

The second proof is

```
ir. ufi union2 H. nin (union_exists H). nin H0.
nin (doubleton_or H1) ; [ left | right ] ; wrr H2.
```

The third proof is

```
rewrite/union2 => x y a ; rewrite union_rw.
by case => t [ aat td ]; case (doubleton_or td)=> <-; auto.
```

The current theorem is

```
Lemma setU2_hi x y a: inc a (x \cup y) -> inc a x \/ inc a y.
Proof. by case /setU_P => t aat /set2_P [] <-; auto. Qed.
```

Let's define a task as a list of expressions of the form $H_1 \dots H_n \vdash C$, where H_i is called the i-th assumption and C the conclusion. A task is said trivial if the conclusion is one of the assumptions. A proof script is a sequence of transformations that convert a task without assumption like $\vdash C$ into a list of trivial tasks. In this case, one can say `Qed`, and Coq considers C as a theorem.

The following transformations are legal. One may add an axiom or a theorem to the list of assumptions. If A and $A \to B$ are assumptions, then B can be added as an assumption. If C has the form $\forall x, C'$ or the form $A \to C'$, one may add the variable x or the proposition A to the list of hypotheses, and replace the conclusion by C'; the converse is possible. There are rules that govern the logical connectors `and` and `or`. For instance, one may replace the task $H \vdash A \wedge B$ by the two tasks $H \vdash A$ and $H \vdash B$, or replace $H \vdash A \vee B$ by any of the two tasks $H \vdash A$ or $H \vdash B$. If assumption H_i is $A \wedge B$ it can be replaced by the two assumptions H_A and H_B asserting A and B; if assumption H_i is $A \vee B$, the task can be replaced by the two tasks $H_A \vdash C$ and $H_B \vdash C$, where H_A means the list of assumptions H where H_i is replaced by A. The connectors `and` and `or` are inductive objects; this means that the rules for $\wedge$ and $\vee$ described above are not built-in in Coq, but are deduced from a more general scheme. In particular, there are infinite objects in Coq, but Bourbaki needs an axiom that says that there is an infinite set.

The mathematical proof is the following. By definition of a doubleton, $t \in \{x, y\}$ is equivalent to $t = x$ or $t = y$. We shall refer to this as theorem (D). On the other hand, a is in the union of b if and only if there is c such that $a \in c$ and $c \in b$. We shall refer to this as theorem (U). By definition $x \cup y$ is $\bigcup\{x, y\}$, so that $a \in x \cup y$ implies by (U) that there is t such that $a \in t$ and $t \in \{x, y\}$. By (D), $t = x$ or $t = y$, from which we deduce $a \in x$ or $a \in y$. In all four Coq proofs, you can see how definitions are unfolded, theorems (U) and (D) are introduced, the equality $t = x$ or $t = y$ is rewritten, and the logical `or` connector is handled (either by `auto`, `intuition` or specifying a branch). The first proof is that of Carlos Simpson, the second one is a slight simplification of it (it avoids introducing two local variables `e` and `o`).

The two other proofs use the SSReflect tactics. In particular, after the the tactical '`=>`', the arrow '`<-`' means 'rewrite from right to left', the notation '`rewrite/union2`' means 'unfold `union2`'. Both theorems (U) and (D) say that two quantities X and Y are equivalent. We sometimes provide a variant of the form $X \implies Y$ and $Y \implies X$ (these variants are used in the first two proofs). The theorem was once stated as X = Y (the third proof thus uses `rewrite`). In

the last version, we use the construction '`case/D`' meaning: use one of the two implications $X \Longrightarrow Y$ and $Y \Longrightarrow X$, then perform a case analysis (to introduce t or handle the disjunction). The brackets in '`case => t [ aat td ]`' mean: split the conjunction in two assumptions $a \in t$ and $t \in \{x, y\}$, named `aat` and `td`. In the last proof, there are no brackets, because we replaced `exists` by `exists2`. Moreover, the is no assumption `td`; we apply theorem (D) to $t \in \{x, y\}$, and follow it by case analysis (the empty brackets) and a rewrite. The final `auto` could be replace by '`[by left | by right]`'.

The last proof uses the SSREFLECT style of programming. It is characterized by the following three properties: each line of the proof is formed of a single sentence (a sequence of semi-colon separated statements that ends with a period); the code is indented according to the number of tasks. Finally all local names are given explicitly (x, y, t, `td`) instead of being computed by COQ (like o, e, `H2`, etc); this makes the proof more robust (note also that the last script uses much less names than the other ones). The third proof is slightly longer than the second one (on average, the size of the proofs increased by 7% after conversion to SSREFLECT style, but the possibility of chaining reduces this again).

1.4 Notations

Choosing tractable notations is a difficult task. We would like to follow the definitions of Bourbaki as closely as possible. For instance he defines the union of a family $(X_\iota)_{\iota\in I}(X_\iota \in \mathfrak{G})$. Classic French typography uses italic lower-case letters, and upright upper-case letters, but the current math tradition is to use italics for both upper- and lower-case letters for variables; constants like pr_1 and Card use upright font. The set of integers is sometimes noted $\mathbb{N}$; but Bourbaki uses only **N**. Some characters may have variants (for instance, the previous formula contains a Fraktur variant of the letter G). In the XML version of this document we do not use the Unicode character U+1D50A (because most browsers do not have the glyph), but a character with variant, so that there is little difference between G, **G**, G, G, $\mathbb{G}$, $\mathfrak{G}$. In this document we use only one variant of the Greek alphabet (Unicode provides normal, italic, bold, bold-italic, sans-serif and sans-serif bold italic; as a consequence, the XML version shows generally a slanted version of Greek characters, where the Pdf document uses an upright font).

We can easily replace lower Greek letters by their Latin equivalents (there is little difference between $(X_\iota)_{\iota\in I}$ and $(X_i)_{i\in I}$). We can replace these unreadable old German letters by more significant ones. In the original version, C. Simpson reserved the letters A, B and E. Thus, a phrase like: let A and B be two subsets of a set E, and $I = A \times B$, all four identifiers are reserved letters in Simpson's framework.

In the original version of C. Simpson, the following letters were defined: A B E I J L O P Q R S V W X Y Z. This means that, if we use such a letter as a local name, we must use a full qualified named in order to access the original meaning, for instance `Coq.Init.Logic.I` for I (this is a proof of `True` and is rarely used). COQ uses the letter O as the integer zero, but provides the notations `0` and `0%nat` for it (notations can be overloaded). In SSREFLECT, the successor of n is denoted `n.+1` instead of '`S n`'.

Quantities named R, B, X, Y, and Z by Simpson have been renamed to Ro, Bo, Xo, Yo and Zo (and Xo has been withdrawn). Quantities V and W have been renamed Vg and Vf. Quantity A has been removed (it was a prefix version of &). Quantity E has been renamed as `Bset` then as `Set`: this is the type of a Bourbaki set.

Given two objects x and y, one can construct a third object z, such that x and y can be obtained from z. If x is of type A, and y of type B, then z is of type $A * B$, the product of

the types (the name is `prod`). One can use the notation (x, y) instead of '`pair x y`'; the two quantities x and y can be obtained via '`fst z`' and '`snd z`' or '`z.1`' and '`z.2`' in SSREFLECT. In what follows, we shall define the cartesian product of two sets; here x, y and z will have the same type (namely `Set`). Our cartesian product will be called `product`, the notation for the pair constructor will be '`J x y`', while P and Q are notations for the two projectors, denoted by Bourbaki as pr_1 and pr_2.

Bourbaki has a section titled "definition of a function by means of a term". An example would be $x \mapsto (x, x)$ ($x \in \mathbb{N}$). This corresponds to the COQ expression `fun x:nat => (x,x)`. According to the COQ documentation, the expression "defines the abstraction of the variable `x`, of type `nat`, over the term `(x,x)`. It denotes a function of the variable `x` that evaluates to the expression `(x,x)`". Bourbaki says "a mapping of A into B is a function f whose source is equal to A and whose target is equal to B". The distinction between the terms function and mapping is subtle: there is a section called "sets of mappings of one set into another"; it could have been: "sets of functions whose source is equal to some given set and whose target is equal to some other given set". It is interesting to note that the term 'function' is used only once in the exercises to Chapter III, in a case where 'mapping' cannot be used because Bourbaki does not specify the set B.

In what follows, we shall use the term 'function' indifferently for S, or the mapping $n \mapsto n+1$, or the abstraction `n => S n`. Given a set A, we can consider the graph g of this mapping when n is restricted to A. This will be denoted by Lg. Given a set B, if our mapping sends A to B, we can consider the (formal) function f associated to the mapping with source A and target B. We shall denote this by `Lf`. These two objects f and g have the important property that, if n is in A, there is an m denoted by $f(n)$ or $g(n)$ such that $m = n+1$ (we have the additional property that $f(n)$ is in B). A short notation is required for the mapping $(g, n) \mapsto g(n)$ or $(f, n) \mapsto f(n)$. We shall use $\mathscr{V}$ or $\mathscr{W}$, in the documentation, `Vg` and `Vf` in the code.

There a possibility to change the COQ parser and pretty printer, and give meanings to `(x,y)` and `{ x : A | P }`. As mentioned above, notations can be overloaded, so that 0 may be the integer zero, in some cases, or the unit of a group in some other cases. We have seen that $A * B$ denotes the type of the pair (x, y), but it can denote the product of two integers, or two elements of a group. We shall not overload existing notations, but add notations similar to those existing in SSREFLECT. For instance '`\1c *c x = x`' means that the cardinal product of the cardinal 1 and x is equal to x. The notation '`{inc (domain f), f =1g g}`' means that the graphs f and g are functionally equal on the domain of f; i.e., whenever x is the domain of f, then $\mathscr{V}_f(x) = \mathscr{V}_g(x)$. The notation '`{inc X &, injective P}`' means that P is injective on X, i.e., for any x and y, if $x \in X$ and $y \in X$, then $P(x) = P(y)$ implies $x = y$.

1.5 Description of formal mathematics

Terms and relations. A mathematical theory $\mathscr{T}$ is a collection of words over a finite alphabet formed of letters, logical signs and specific signs. Logical signs are $\square$, τ, $\vee$, $\neg$ (the first two signs are specific to Bourbaki, the other ones, disjunction and negation, have their usual meaning). Specific signs are $=$, $\in$, letters are x, y, A, A′, A″, A‴, and "at any place in the text it is possible to introduce letters other than those which have appeared in previous arguments" [2, p. 15] (any number of prime signs is allowed; this is not in contradiction with the finiteness of the alphabet). An assembly is a sequence of signs and links. Some assemblies are well-formed according to some grammar rules. In Backus-Naur form they are:

Term := letter | τ_{letter} (Relation) | Ssign Term_1 ... Term_n

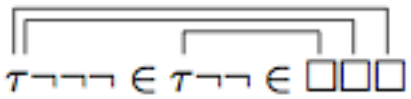
Relation := ¬ Relation | ∨ Relation Relation | Rsign Term$_1$ ... Term$_n$
Each sign has to be followed by the appropriate number of terms: □ takes none, ∈ and = are followed by two terms, and one can extend Bourbaki to non-standard analysis [6] by introducing a specific sign $^{\text{st}}$ of weight 1 qualifying the relation that follows to be standard. Each sign is substantific as □ (it yields a term) or relational as = (it yields a relation).

We shall see below that $\tau_x(\text{R})$ has to be interpreted as the expression where all occurrences of x in R are replaced by □ and linked to the τ. Parentheses are removed. This has one advantage: there is no x in $\tau_x(\text{R})$, hence substitution rules become trivial. For instance, the function $x \mapsto x+y$ is constructed by using τ, it is identical to the function $z \mapsto z+y$. If we want to replace y by z, we get $x \mapsto x+z$, but not $z \mapsto z+z$. In Coq, the variable y appears *free* in $x \mapsto x+y$, and the variable x appears *bound* in the same expression. Renaming bound variables is called α-*conversion*. Two α-convertible terms are considered equal in Coq.

The Appendix to Chapter I of [2] describes an algorithm that decides whether an assembly is a term, a relation, or is ill-formed. It works in two stages. In the first stage, links are ignored. A classical result in computer science is that there exists a program (called a *parser*) that recognizes all *significant words* (i.e., well-formed assemblies without links). We can associate a number to each sign (for instance 262 to 'a', 111 to '=') and thus to each assembly (for instance, 262111262 to 'a=a'). This will be called the Gödel number of the assembly, see [4] for an example. Two distinct assemblies have distinct Gödel numbers. The set of Gödel numbers is a recursively enumerable set. Given assemblies A_1, A_2, A_3, etc, one can form the concatenation $\text{A}_1\text{A}_2\text{A}_3\ldots$. If each assembly is a significant word, there is a unique way to recover A_i from the concatenation, hence from the Gödel number of the concatenation.

A *demonstrative text* for Bourbaki is a sequence of assemblies $\text{A}_1\text{A}_2\ldots\text{A}_n$, that contains a *proof*, which is a sub-sequence $\text{A}'_1\text{A}'_2\ldots\text{A}'_m$ of relations, where each A'_i can be shown to be true by application of a basic derivation rule that uses only A'_j for $j<i$. Each A'_i is a *theorem*. We shall use a variant: a *proof-pair* is a sequence of relations $\text{A}'_1\text{A}'_2\ldots\text{A}'_m$ satisfying the same conditions as above, and a *theorem* is the last relation A'_m in a proof-pair. If our basic rules are simple enough, the property of a number g to be the Gödel number of a proof-pair is primitive recursive. From this, one can deduce the existence of a true statement that has no proof (this is Gödel's Theorem).

An assembly A containing links is analyzed by using *antecedents*, which are assemblies of the form $\tau_x(\text{R})$ (where x is some variable) that are identical to A if x is substituted in R and links are added. The algorithm for deciding that an assembly with links is a term or a relation is rather complicated. Bourbaki gives three examples of assemblies with links; the antecedent of the first one is $\tau_x(x \in y)$ (there is a single link); the antecedent of the second one is $\tau_x(x \in \text{A}' \implies x \in \text{A}'')$ (there are two links); the third one is the empty set, see picture below. One can replace these links by the De Bruijn indices, so that the empty set would become $\tau\neg\neg\neg \in \tau\neg\neg \in 121$. This has two drawbacks: the first one is that 121 could be understood as one integer or a sequence of three integers, the second is that this notation assumes that integers are already defined. The remedy to the first problem would be to insert a separator (for instance a square) and a remedy to the second would be to use a base-one representation of integers; the empty set would be $\tau\neg\neg\neg \in \tau\neg\neg \in \square-\square--\square-$. The scope of the second τ is the scope of its operator, thus $\neg\neg \in \square\square$. This means that the two squares are in the scope of both τ, are are linked to the second and first τ respectively. The third square is in the scope of the first τ only, hence is linked to the first τ. Formal mathematics in Bourbaki is so complicated that the □ symbol is, in reality, never used.



Denote by (***B***|***x***) ***A*** the assembly obtained by replacing ***x***, wherever it occurs in ***A***, by the assembly ***B***. Bourbaki has some criteria of substitutions, CS1, CS2, etc, that are rules about substitutions. For instance CS3 says that $\tau_{\boldsymbol{x}}(\boldsymbol{A})$ and $\tau_{\boldsymbol{x}'}(\boldsymbol{A}')$ are identical if $\boldsymbol{A}'$ is $(\boldsymbol{x}'|\boldsymbol{x})$ ***A*** provided that $\boldsymbol{x}'$ does not appear in ***A*** (informally: since x does not appear in τ_x(A), the name of the variable x is irrelevant). Formative criteria CF1, CF2, etc., give rules about well-formedness of assemblies. For instance CF8 says that (***T***|***x***) ***A*** is a term or a relation whenever ***A*** is a term or a relation, ***T*** is a term, ***x*** is a letter.

Abbreviations are allowed, so that $\vee\neg$ can be replaced by $\Longrightarrow$, and $\neg\in$ can be replaced by $\notin$. Abbreviations may take arguments, for instance $\wedge$AB is the same as $\neg\vee\neg$A$\neg$B. A term may appear more than once, for instance $\Longleftrightarrow$ AB is the same as $\wedge\Longrightarrow$ AB $\Longrightarrow$ BA, and after expansion $\neg\vee\neg\vee\neg$AB$\neg\vee\neg$BA. The logical connectors $\neg$, $\vee$ and $\wedge$ are written ~, \/, and /\ in Coq[3]. Note that in Coq, A → B is the type of a function from A to B but also means A $\Longrightarrow$ B. There is no limit on the number of abbreviations (Bourbaki invented $\emptyset$ as a variant of Ø). Unicode provides a lot of symbols, but few of them are available in LaTeX or in Web browsers.

Starting with Section 2, Bourbaki switches to infix notation. For instance, whenever ***A*** and ***B*** are relations so is $\vee\neg\neg\vee\neg\boldsymbol{A}\neg\boldsymbol{B}\boldsymbol{A}$, by virtue of CF5 and CF9. Using abbreviations, this relation can be written as $\Longrightarrow\wedge\boldsymbol{A}\boldsymbol{B}\boldsymbol{A}$. The infix version is (***A*** and ***B***) $\Longrightarrow$ ***A***. In order to remove ambiguities, parentheses are required, but Bourbaki says: "Sometimes we shall leave out the brackets" [2, p. 24], in the example above three pairs of brackets are left out. In some cases Bourbaki writes A ∪ B ∪ C. This can be interpreted as (A ∪ B) ∪ C or A ∪ (B ∪ C). These are two distinct objects that happen to be equal: formally, the relation (A ∪ B) ∪ C = A ∪ (B ∪ C) is true. Similarly A ∨ B ∨ C is ambiguous, but it happens, according to C24, that (A ∨ B) ∨ C and A ∨ (B ∨ C) are equivalent (formally: related by $\Longleftrightarrow$). In Coq, we use `union2` as prefix notation for ∪, so we must chose between ∪(∪AB)C or ∪A(∪BC). Function calls are left-associative, and brackets are required where indicated. We use \/ or `\cup` as infix notation for ∨ or ∪, parentheses may be omitted, the operator is right associative.

Theorems and proofs. Each relation can be true or false. To say that P is false is the same as to say that ¬P is true. To say that P is either true or false is to say that P ∨ ¬P is true. A relation is true by assumption or deduction. A relation can be both true and false, case where the current theory is called contradictory (and useless, since every property is then true). There may be relations P for which it is impossible to deduce that P is true and it is also impossible to deduce that P is false (Gödel's theorem). A property can be independent of the assumptions. This means that it is impossible to deduce P or ¬P; in other words, adding P or ¬P does not make the theory contradictory. An example is the axiom of foundation (see below), or the continuum hypothesis (every uncountable set contains a subset which has the power of the continuum).

Some relations are true by assumption; these are called axioms. An axiom scheme is a rule that produces axioms. The list of axioms and schemes used by Bourbaki are given at the end of the document. A true relation is called a Theorem (or Proposition, Lemma, Remark, etc). A conjecture is a relation believed to be true, for which no proof is currently found. As said above, in Bourbaki, a theorem is a relation with a proof, which consists of a sequence of true statements, the theorem is one of them, and each statement R in the sequence is either

[3] We originally used & instead of ∧, since it is easier to type; but this forbids using some other notations

an axiom, follows by applications of rules (the axiom schemes) to previous statements, or there are two previous statements S and T before R, where T has the form S $\Longrightarrow$ R.

It is very easy for a computer to check that an annotated proof is correct (provided that we use a parsable syntax); but a formal proof is in general huge. Examples of formal proofs can be found in [4]; the theory used there is simpler than Bourbaki's, but contains arithmetics on integers. We give here a proof of 1 + 1 = 2:

(1)	∀a:∀b:(a+Sb)=S(a+b)	axiom 3
(2)	∀b:(S0+Sb)=S(S0+b)	specification (S0 for a)
(3)	(S0+S0)=S(S0+0)	specification (0 for b)
(4)	∀a:(a+0)=a	axiom 2
(5)	(S0+0)=S0	specification (S0 for a)
(6)	S(S0+0)=SS0	add S
(7)	(S0+S0)=SS0	transitivity (lines 3,6)

The proof is formed of the statements in the second column; the annotations of the third column are not part of the formal proof. The line numbers can be used in the annotations. In COQ, the annotations are part of the proof. The principle is: a theorem is a function and applying the theorem means applying the function. For instance, transitivity of equality is a function `eq_trans`; in line (7) we apply it to two arguments, the statements of lines 3 and 6. The statement of line 6 is obtained by applying `f_equal` with argument S to the statement that precedes (the `f_equal` theorem states that for every function f and equality $a = b$ we have $f(a) = f(b)$). In COQ, a proof is a tree, the advantage is that we do not need to worry about line numbers.

Bourbaki has over 60 criteria that help proving theorems. The first one says: if $\boldsymbol{A}$ and $\boldsymbol{A} \Longrightarrow \boldsymbol{B}$ are theorems, then $\boldsymbol{B}$ is a theorem. This is not a theorem, because it requires the fact that $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations. On the other hand $x = x$ is a theorem (the first in the book). The difference is the following: if A and B are letters then A $\Longrightarrow$ B is not well-formed. Until the end of E.II.5, Bourbaki uses a special font as in $\boldsymbol{A} \Longrightarrow \boldsymbol{B}$ to emphasize that $\boldsymbol{A}$ and $\boldsymbol{B}$ are to be replaced by something else.

Criterion C1 works as follows. If $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n$ and $\mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m$ are two proofs, if the first one contains A, if the second one contains A $\Longrightarrow$ B, then

$$\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n, \mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m, \mathrm{B}$$

is a proof that contains B. Assume that we have two annotated proofs R_i and S_j, where A is R_n and A $\Longrightarrow$ B is S_m. Each statement has a line number, and we can change these numbers so that they are all different (this is a kind of α-conversion). Let N and M be the line numbers of R_n and S_m. We get an annotated proof by choosing a line number for the last statement, and annotating it by: detachment N M (this is also known as syllogism, or Modus Ponens).

Criterion C6 says the following: assume $\boldsymbol{P} \Longrightarrow \boldsymbol{Q}$ and $\boldsymbol{Q} \Longrightarrow \boldsymbol{R}$. From axiom scheme S4, we get $(\boldsymbol{Q} \Longrightarrow \boldsymbol{R}) \Longrightarrow ((\boldsymbol{P} \Longrightarrow \boldsymbol{Q}) \Longrightarrow (\boldsymbol{P} \Longrightarrow \boldsymbol{R}))$. Applying Criterion C1 gives $(\boldsymbol{P} \Longrightarrow \boldsymbol{Q}) \Longrightarrow (\boldsymbol{P} \Longrightarrow \boldsymbol{R})$. Applying it again gives $\boldsymbol{P} \Longrightarrow \boldsymbol{R}$. If $\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n$ and $\mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m$ are proofs of $\boldsymbol{P} \Longrightarrow \boldsymbol{Q}$ and $\boldsymbol{Q} \Longrightarrow \boldsymbol{R}$ then a proof of $\boldsymbol{P} \Longrightarrow \boldsymbol{R}$ is

$$\mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n, \mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m, \mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n, \mathrm{S}_1, \mathrm{S}_2, \ldots, \mathrm{S}_m, \mathrm{A}_4, \mathrm{D}_y, \mathrm{D}_y.$$

Here A_4 and D_y are to be replaced by the appropriate relation, or in the case of an annotated proof, by the appropriate annotation (for instance in the case of A_4, we must give the values of three arguments of the axiom scheme S4, in the case of detachment D_y we must give the position of the arguments of the syllogism in the proof tree).

Criterion C8 says $\boldsymbol{A} \implies \boldsymbol{A}$. This is a trivial consequence from S2, $\boldsymbol{A} \implies (\boldsymbol{A} \vee \boldsymbol{A})$ and S1, $(\boldsymbol{A} \vee \boldsymbol{A}) \implies \boldsymbol{A}$. This is by definition $\neg \boldsymbol{A} \vee \boldsymbol{A}$, and is called the "Law of Excluded Middle".

There is a converse to C1. If we can deduce, from the statement that A is true, a proof of B, then A $\implies$ B is true. This is called the *method of the auxiliary hypothesis*. Almost all theorems we shall prove in Coq have this form.

Criterion C21 says that $\vee \neg \neg \vee \neg \boldsymbol{A} \neg \boldsymbol{B} \boldsymbol{A}$ is a theorem, whenever $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations. We have already seen this assembly and showed that it is a relation. If we could quantify relations, the criterion could be converted into a theorem that says "$(\forall A)(\forall B)((A \text{ and } B) \implies A)$".

If P and Q are propositions, one can show that $\neg\neg P \Rightarrow P$, $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$, $P \vee \neg P$, $\neg(\neg P \wedge \neg Q) \Rightarrow P \vee Q$ and $(P \Rightarrow Q) \Rightarrow (\neg P \vee Q)$ are equivalent. These statements are unprovable in Coq. They are true in Bourbaki since the last statement is a tautology. In [4], there is the Double-Tilde Rule that says that the string '~~' can be deleted from any theorem, and can be inserted into any theorem provided that the resulting string is itself well-formed. We solve this problem by adding the first statement as axiom. Then all theorems of Bourbaki can be proved in Coq. There are still two difficulties: the first one concerns the status of τ (see below); the second concerns sets. Bourbaki says *in the formalistic interpretation of what follows, the word "set" is to be considered as strictly synonymous with "term"* [2, p. 65]. Recall that there are only two kinds of valid assemblies, namely terms and relations. We shall see below how to implement sets in Coq.

In Coq, we can quantify everything so that criterion C21 becomes a conjunction of two theorems (`proj1` and `proj2` in the Coq library); the first of them can be proved as follows.

```
Lemma example: forall A B, A /\ B -> A. intros. induction H. exact H. Qed.
```

There are three steps in the proof. We start with a single task without assumption: $\vdash \forall A, B, A \wedge B \implies A$, then introduce some names and assumptions in order to get $A, B, A \wedge B \vdash A$, then destruct the logical connector: $A, B, H_A, H_B \vdash A$. This is a trivial task since H_A asserts the conclusion A. The last step could have been `trivial`, since Coq is able to find the assumption H_A. In the second step, we could use `destruct`, `case` or `elim` (the Coq library uses `destruct`).

Printing the theorem yields

```
example =
fun (A B : Prop) (H : A /\ B) => and_ind (fun (H0 : A) (_ : B) => H0) H
     : forall A B : Prop, A /\ B -> A
Arguments A, B are implicit
Argument scopes are [type_scope type_scope _]
```

This tells you that the arguments A and B are implicit (since they can be deduced from the third argument H), and gives information about the scope used by notations. The theorem has he form "name = proof : value". The last line is the value of the theorem. The second line is the proof. The proof could also be

```
fun A B : Prop => [eta and_ind (fun (H0 : A) (_ : B) => H0)]
```

Here '`[eta f]`' is a notation for '`fun x => f x`' so that '`[eta f z ]`' is a notation for '`fun x => f z x`'. Note that the notations hide the argument H and its type. In the case $A \wedge B \implies B$ you would see:

```
fun A B : Prop => [eta and_ind (fun _ : A => id)]
```

As you can see, this is not just a sequence of statements with their justification, but function calls. It applies `and_ind` to f_2 and H where f_2 is a function of two arguments that returns the first one, and ignores the second (the proof of B). In the case of $A \wedge B \implies B$, you see a function f_2' of two arguments that ignores the first argument, so that $f_2'(x)$ is the identity function (with argument of type B, x being of type A and ignored). We show here the function:

```
and_ind =
fun A B P : Prop => and_rect (A:=A) (B:=B) (P:=P)
     : forall A B P : Prop, (A -> B -> P) -> A /\ B -> P
Arguments A, B, P are implicit
```

Here `and_rect` is a function with five arguments, two propositions A and B, a type P, a function $f : A \to B \to P$ and an object c of type $A \wedge B$. It deduces two objects a and b of type A and B, and applies f to it, yielding an object of type P. The first three arguments are implicit. Now, `and_ind` is the same as `and_rect` (except for the type of P). This is a function that returns an object of type P, given f and c. Note that the arguments of `and_rect` must be given explicitly (they could be deduced from f, but f is not an argument). Assume that f returns its first (resp. second) argument and P is the type of this argument. We get: if there is an object of type $A \wedge B$, there is an object of type P.

You could also use `destruct` or `case`. In this case you see

```
fun (A B : Prop) (H : A /\ B) => match H with | conj H0 _ => H0 end
```

This has to be understood as follows. The object H is of type `and`, and we perform a case analysis on its constructors. There is only one, `conj`, that takes two arguments, says H_0 and H_1. The function returns H_0 (in this case, induction is the same as case analysis).

In SSREFLECT, you can say

```
Lemma example A B: A /\ B -> A. Proof. by case. Qed.
```

This yields the following proof

```
fun (A B : Prop) (_top_assumption_ : A /\ B) =>
(fun _evar_0_ : forall (a : A) (b : B), (fun _ : A /\ B => A) (conj a b) =>
 match _top_assumption_ as a return ((fun _ : A /\ B => A) a) with
 | conj x x0 => _evar_0_ x x0
 end) (fun (a : A) (_ : B) => a)
```

Let's write H and z for the two variables introduced by SSREFLECT, and let f be the third `fun`. This function returns A given any argument of type $A \wedge B$. This function is called twice. In the first case, the argument is '`conj a b`', where a and b are of type A and B, so that the argument has the right type. In the second case, its argument is a, where a is bound to H, so has the correct type. This means that we can replace $f(x)$ by A in both cases. Now, the argument of the second fun has type $\forall a : A, \forall b : B, A$, this is the type of f_2. Lines 3 and 4 become: "match H as c return T with C end", where T is A. Note that H has a single constructor; assume that its arguments are u and v. We apply f_2 and coerce this to type T; the coercion is trivial.

We show here the proof tree of the third variant of `union2_or`.

```
union2_or =
fun x y a : Set =>
eq_ind_r
```

```
  (fun _pattern_value_ : Prop => _pattern_value_ -> inc a x \/ inc a y)
  (fun _top_assumption_ : exists y0 : Set, inc a y0 & inc y0 (doubleton x y) =>
   match _top_assumption_ with
   | ex_intro t (conj aat td) =>
       match doubleton_or td with
       | or_introl _top_assumption_1 =>
           eq_ind t
             (fun _pattern_value_ : Set => inc a _pattern_value_ \/ inc a y)
             (or_introl (inc a y) aat) x _top_assumption_1
       | or_intror _top_assumption_1 =>
           eq_ind t
             (fun _pattern_value_ : Set => inc a x \/ inc a _pattern_value_)
             (or_intror (inc a x) aat) y _top_assumption_1
       end
   end) (union_rw a (doubleton x y))
     : forall x y a : Set, inc a (union2 x y) -> inc a x \/ inc a y
```

The current version is bit longer; we show here only a part of it. You can see the two `iffLR` and the `exists2`.

```
setU2_hi =
fun (x y a : Set) (_top_assumption_ : inc a (x \cup y)) =>
(fun
   _evar_0_ : forall (x0 : Set) (p : [eta inc a] x0)
                (q : (inc^~ (doubleton x y)) x0),
              (fun _ : exists2 z : Set, inc a z & inc z (doubleton x y) =>
               inc a x \/ inc a y)
                (ex_intro2 [eta inc a] (inc^~ (doubleton x y)) x0 p q) =>
 match
   iffLR (setU_P (doubleton x y) a) _top_assumption_ as e
   return
     ((fun _ : exists2 z : Set, inc a z & inc z (doubleton x y) =>
       inc a x \/ inc a y) e)
 with
 | ex_intro2 x0 x1 x2 => _evar_0_ x0 x1 x2
 end)
...
 (iffLR (set2_P t x y) _top_assumption_0))
```

Quantified theories. As mentioned above, Bourbaki defines $\tau_x(\mathrm{R})$ as the construction obtained by replacing all x in R by $\square$, adding τ in front, and drawing a line between τ and this square. An example is $\tau\neg\neg\neg \in \tau\neg\neg \in \square\square\square$. It corresponds to $\tau_x(\neg\neg\neg \in \tau_y(\neg\neg \in yx)x)$. The positions of the parentheses is fixed by the structure, but not the names (without the links the expression is ambiguous). If we admit that the double negation of P is P and use infix notation, the previous term is equivalent to $\tau_x(\tau_y(y \in x) \notin x)$. This is the empty set.

Denote by $(\mathrm{T}|x)$ R the expression R where all free occurrences of the letter x have been replaced by the term T. Paragraph 2.4.1 of [1] explains that this is a natural operation in Coq; the right amount of α-conversions are done so that free occurrences of variables in T are still free in all copies of T. For instance, if R is $(\exists z)(z = x)$, if we replace x by z, the result becomes $(\exists w)(w = z)$. These conversions are not needed in Bourbaki: there is no x in $\tau_x(\mathrm{R})$ and no z in $(\exists z)(z = x)$. Of course, if we want to simplify $(z|x)$ $(\exists z)(z = x)$, we can replace it by $(z|x)$ $(\exists w)(w = x)$ (thanks to rule CS8) then by $(\exists w)((z|x)$ $(w = x))$ (thanks to rule CS9), then simplify as $(\exists w)(w = z)$.

Bourbaki defines $(\forall \boldsymbol{x})\boldsymbol{R}$ as "not $((\exists \boldsymbol{x})$ not $\boldsymbol{R})$", whereas '`forall x:T, R`' is a Coq primitive, whose meaning is (generally) obvious; instead of T, any type can be given, it may be omitted if it is deducible via type inference. The expression $(\forall_{\boldsymbol{T}} \boldsymbol{x})\boldsymbol{R}$ is defined in Bourbaki, similar to the Coq expression, but not used later on; we shall not use it here. The dual expression '`exists x:A, R`' is equivalent in Coq to '`ex(fun x:A=>R)`'. Note that the syntax rules allow '`forall x y, P x y`' or '`exists x y, P x y`'. There is a similar construction '`{x:A | P}`'. These are defined by

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> exists x, P x
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> {x | P x}
```

If y is of type '`{x:A | P}`', then there is x of type A satisfying P; it is '`sval y`'. If y is of type '`exists x, P`', then there is an x satisfying P, and it can be used in proof; however, there is no function $y \to x$. This can be restated as: if $f : \mathrm{A} \to \mathrm{B}$ is a surjective function, for any $y : \mathrm{B}$ there is $x : \mathrm{A}$ such that $f(x) = y$, but there is no function g such that $f(g(y)) = y$; a form of the axiom of choice is needed.

Bourbaki defines $(\exists \boldsymbol{x})\boldsymbol{R}$ as $(\tau_{\boldsymbol{x}}(\boldsymbol{R})|\boldsymbol{x})\boldsymbol{R}$. Write y instead of $\tau_{\boldsymbol{x}}(\boldsymbol{R})$. Our expression is $(y|\boldsymbol{x})\boldsymbol{R}$. It does not contain the variable $\boldsymbol{x}$, since $\boldsymbol{x}$ is not in y. If $(\exists \boldsymbol{x})\boldsymbol{R}$ is true, then $\boldsymbol{R}$ is true for at least one object, namely y. This object is explicit: we do not need to introduce a specific axiom of choice. Axiom scheme S5 states the converse: if for some $\boldsymbol{T}$, $(\boldsymbol{T}|\boldsymbol{x})\,\boldsymbol{R}$ is true, then $(\exists \boldsymbol{x})\boldsymbol{R}$ is true.

Let's give an example of a non-trivial rule. As noted in [4], it is possible to show, for each integer n, that $0 + n = n$ (where addition is defined by $n + 0 = n$ and $n + \mathrm{S}m = \mathrm{S}(n + m)$), but it is impossible to prove $\forall n, 0 + n = n$. The following induction principle is thus introduced: "Suppose u is a variable, and $\mathrm{X}\{u\}$ is a well-formed formula in which u occurs free. If both $\forall u : \langle \mathrm{X}\{u\} \supset \mathrm{X}\{\mathrm{S}u/u\}\rangle$ and $\mathrm{X}\{0/u\}$ are theorems, then $\forall u : \mathrm{X}\{u\}$ is also a theorem."

Criterion C61 [2, p. 168] is the following: Let $\mathrm{R}\{n\}$ be a relation in a theory $\mathscr{T}$ (where n is not a constant of $\mathscr{T}$). Suppose that the relation

$$\mathrm{R}\{0\} \text{ and } (\forall n)((n \text{ is an integer and } \mathrm{R}\{n\}) \implies \mathrm{R}\{n+1\})$$

is a theorem of $\mathscr{T}$. Under these conditions the relation

$$(\forall n)((n \text{ is an integer}) \implies \mathrm{R}\{n\})$$

is a theorem of $\mathscr{T}$.

The syntax is different, but the meaning is the same. This criterion is a consequence of the fact that a non-empty set of integers is well-ordered. In Coq a consequence of the definition of integers is the following induction principle (which follows trivially from the fact that one can define recursive functions):

```
nat_ind: forall P : nat -> Prop,
  P 0 -> (forall n : nat, P n -> P n.+1) -> forall n : nat, P n
```

Equality. In Bourbaki, equality is defined by the two axioms schemes S6 and S7, as well as axiom A1 (see section 8.1 for details). The first scheme says that if P is a property depending on a variable z, if $x = y$, then $\mathrm{P}(x)$ and $\mathrm{P}(y)$ are equivalent. The second scheme says that if Q and R are two properties depending on a variable z, if $\mathrm{Q}(z)$ and $\mathrm{R}(z)$ are equivalent for all z, then $\tau_z(\mathrm{Q}) = \tau_z(\mathrm{R})$. The axiom says that if $x \in \mathrm{A}$ is equivalent to $x \in \mathrm{B}$ then $\mathrm{A} = \mathrm{B}$ (the converse being true by S6).

Let $R(z)$ be a relation. If $R(x)$ and $R(y)$ implies $x = y$, then R is said *single-valued*. If moreover there exists x such that $R(x)$ is true, then R is said *functional*. In this case $R(x)$ is equivalent to $x = \tau_z(R)$. Proof. The relation $(\exists z)R$ is the same as $R(\tau_z(R))$. Since this is true, $R(x)$ implies $x = \tau_z(R)$, since R is single-valued. Conversely, if $x = \tau_z(R)$ then $R(x)$ is equivalent to $R(\tau_z(R))$ which is true.

Let $Q(z)$ and $R(z)$ be two functional relations, x and y denote $\tau_z(Q)$, and $\tau_z(R)$ respectively. By S7, if, for all z, $Q(z)$ and $R(z)$ are equivalent then $x = y$, and in this case, the converse is true (if for some z, $Q(z)$ is true, we have $z = x$, thus $z = \tau_z(R)$ and $R(z)$ is true). Example. If $y > 0$ is an integer, there exists a unique x such that $y = x + 1$. Denote by $p(y)$ the quantity $\tau_x(y = x+1)$. We have then the conclusion: $y = x+1$ if and only if $x = p(y)$. Thus $p(y_1) = p(y_2)$ if and only if $y_1 = y_2$. From now on, we can forget that p is defined via τ and equality is defined by S7.

Let $Q(x)$ and $R(x)$ be two relations, and $P(z)$ the following relation $(\forall x)(x \in z \iff Q(x))$. It is single-valued by the axiom of extent. Assume that it is functional[4]; applying τ_z to it gives a set denoted by $\{x, Q(x)\}$; assume that R shares the same property. The previous argument says $\{x, Q(x)\} = \{x, R(x)\}$ if and only if $(\forall x)(Q(x) \iff R(x))$. For instance $\{a, b\} = \{b, a\}$. Moreover $\{a\} = \{b\}$ is equivalent to $a = b$.

Consider now an equivalence relation $P(x, y)$: we assume that $P(x, y)$ implies $P(y, x)$, and that $P(x, y)$ and $P(y, z)$ imply $P(x, z)$. Let x be $\tau_z(P(a, z))$, and y be $\tau_z(P(b, z))$. We have that $P(a, z) \iff P(b, z)$ is equivalent to $P(a, b)$, so that S7 says that $P(a, b)$ implies $x = y$. Conversely, assume a and b in the domain of P; this means that for some z, $P(a, z)$ is true, it implies that $P(a, x)$ is true; we also assume $P(b, y)$ true. Then from S6 we get: if $x = y$ then $P(a, y)$ is true, thus $P(a, b)$. Thus: $x = y$ if and only if $P(a, b)$ is true. Example: Let P be the property that $a = (\alpha, \beta)$ and $b = (\alpha', \beta')$ are pairs of integers such that $\alpha + \beta' = \alpha' + \beta$. This is an equivalence relation, and the domain is the set of pairs of integers. Define $\beta - \alpha$ as $\tau_z P((\alpha, \beta), z)$. If α and β are integers, this is a pair of integers and $\beta - \alpha = \beta' - \alpha'$ if and only if $\alpha + \beta' = \alpha' + \beta$. We can from now on forget that this quantity is defined via τ.

Consider an equivalence relation whose domain is a set E. Let $C(a)$ be the equivalence class of a, namely the set of all $z \in E$ such that $P(a, z)$ is true. Then $P(a, z)$ is equivalent to $z \in C(a)$, and $x = \tau_z(z \in C(a))$. Denote by $r(X)$ the quantity $\tau_z(z \in X)$, so that $x = r(C(a))$. Now, $P(a, b)$ implies $C(a) = C(b)$ thus $x = y$ and scheme S7 is not required. Conversely, if $a \in E$ and $x = r(C(a))$ then $x \in C(a)$ and $P(a, x)$. It follows, as before, that if $b \in E$ and $y = r(C(b))$ and $x = y$ then $P(a, b)$ is true. The quantity $r(X)$ will be called the *representative* of the set X; it satisfies $r(X) \in X$ whenever X is non-empty. Whenever possible we shall use r rather than τ. There are two exceptions: for defining cardinals and ordinals (equivalence classes are not sets). Our current implementation of cardinals and ordinals differs from that of Bourbaki (see second part of this report), and this use of τ is not needed any more.

Finally, we may have $\tau_z(Q) = \tau_z(R)$ even when Q and R are non-equivalent. For instance consider two distinct elements a, b, the three sets $\{x\}$, $\{y\}$ and $\{x, y\}$, denoted by X, Y and Z. The quantities $r(X)$, $r(Y)$ and $r(Z)$ take the values a and b, thus cannot be distinct. We have $r(X) = a$ and $r(Y) = b$. Thus one of $r(X) = r(Z)$ and $r(Y) = r(Z)$ must be true, but which one is undecidable.

In our framework, few objects are defined via τ, and Axiom Scheme S7 is rarely used. For instance $\{1 + 1\} = \{2\}$ is a trivial consequence of $1 + 1 = 2$, and Criterion C44. Let's prove this criterion and the first three theorems of Bourbaki.

Theorem 1 is $x = x$. Bourbaki uses an auxiliary theory in which x is not a constant, so that

[4] For instance, if Q is $x \notin x$ then P is not functional

$(\forall x)(\boldsymbol{R} \iff \boldsymbol{R})$ is true, whatever the relation $\boldsymbol{R}$. Note that x is a letter, thus a term, and it could denote the set of real numbers **R**, case where quantification over x makes no sense, while $\boldsymbol{R}$ is just a notation. Scheme S7 gives $\tau_x(\boldsymbol{R}) = \tau_x(\boldsymbol{R})$. This can be rewritten as $(\tau_x\boldsymbol{R}|x)(x = x)$. Let $\boldsymbol{S}$ denote $x = x$ and $\boldsymbol{R}$ denote $\neg\boldsymbol{S}$. By definition of the universal quantifier, the previous relation is $\neg\neg(\forall x)(\boldsymbol{S})$, from which follows $(\forall x)(x = x)$. It follows that, whatever x (even if x is a constant) we have $x = x$.

Theorem 2 says $(x = y) \iff (y = x)$. Let's show one implication. Assume $x = y$. We apply S6 to $y = x$, considered as a function of y. It says $x = x \iff y = x$, the conclusion follows by Theorem 1.

Theorem 3 says $(x = y \wedge y = z) \implies x = z$. The same argument as above says that if $x = y$, then $x = z \iff y = z$, making the theorem obvious.

The same argument shows that if $\boldsymbol{T} = \boldsymbol{U}$, and if $\boldsymbol{V}$ is a term (depending on a parameter z), then $\boldsymbol{V}\{\boldsymbol{T}\} = \boldsymbol{V}\{\boldsymbol{U}\}$. This is Criterion C44, and is known in COQ as `eq_ind`, while Scheme S6 is `eq_rec`. Theorem 1 is the definition `eq_refl`, other theorems are `eq_sym` and `eq_trans`[5]

There is no equivalent of Scheme S7 in COQ. The Leibniz equality says that two objects x and y are equal iff every property which is true of x is also true of y. We shall later on define special terms called sets. They satisfy $x = y$ if $x \subset y$ and $y \subset x$. This makes equality weaker. In fact, if $x \neq y$, the middle excluded law implies that there exists some a such that either $a \in x$ and not $a \in y$ or $a \in y$ and not $a \in x$. Thus, assuming that 0 and 1 are sets, one of the following statements is true: there exists a such that $a \in 1$ and $a \notin 0$, or there exists b such that $b \in 0$ and $b \notin 1$, or $0 = 1$ (with the Bourbaki definition of integers as cardinals, the first assumption is true, but nothing can be said of a). In Bourbaki all terms are sets. In our work, we shall consider objects that are not sets. For instance, neither 1 nor 2 (considered as natural numbers) are sets. The relations $1 + 1 = 2$ and $1 \neq 2$ are the consequence of the fact that these objects have the same (or different) normal forms (modulo α-conversion).

Carlos Simpson introduced an axiom that says that two propositions are equal if they are equivalent. This is not possible in Bourbaki (since equality applies only to sets). Assume that we have shown a theorem H that says P $\iff$ Q. Let e be the equality P = Q; thanks to the axiom, the equality is true, and we case rewrite it. In fact we can rewrite H as well (thanks to the setoid rewrite rules). Better yet, we can use the constructions `move/H` and `apply/H`. This explains why the axiom has been removed in Version 5.

One can add an axiom that says if f and g are two functions, of type A → B, and if $f(x) = g(x)$ whenever x is of type A, then $f = g$. A stronger form is the following, introduced by Carlos Simpson, but not used anymore

```
(*
Axiom prod_extensionality :
  forall (x : Type) (y : x -> Type) (u v : forall a : x, y a),
    (forall a : x, u a = v a) -> u = v.
Axiom proof_irrelevance : forall (P : Prop) (q p : P), p = q.
*)
```

We are sometimes faced to the following problem: given a proposition P, two sets a and b, we want to select a if P is true, and b otherwise. Bourbaki uses $\tau_x((x = a \wedge \mathrm{P})$ or $(x = b \wedge \neg\mathrm{P}))$. Assume that can find two functions A(p) and B(q) whose values are a and b, whenever p is a proof of P and q a proof of ¬P. Consider the relation R: for any p and q as above, we have

[5]These lemmas have alternate names, and SSREFLECT redefines them, and provides alternatives `erefl`, `esym` and `etrans`.

$y = \mathrm{A}(p)$ and $y = \mathrm{B}(q)$, and let's apply τ_y. If P is true, it has proof a p and $y = \mathrm{A}(p) = a$; if P is false, then $\neg$P is true and has a proof q so that $y = \mathrm{B}(q) = b$. The trick is the following; the expression $\tau_y(\mathrm{R})$ is in general undefined, since there are undecidable propositions (there are also true propositions without proofs). However, we consider $\tau_y(\mathrm{R})$ only in the case where we know a proof of P or a proof of $\neg$P. The proof irrelevance axiom says that if p and p' are two proofs of P, then $p = p'$, which implies $\mathrm{A}(p) = \mathrm{A}(p')$, and makes some proofs easier. Example. Let I be a non-empty set, X_i a family of set indexed by $i \in \mathrm{X}$; we may define the intersection by $\{y \in \mathrm{X}_i, \forall j \in \mathrm{I}, y \in \mathrm{X}_j\}$. This definition depends on i, assumed to satisfy $i \in \mathrm{I}$; it exists because I is non-empty. Assume now that we have two proofs that I is non-empty; this give two possible indices i, and we must show that our definition is independent of i (which is obvious here).

In Coq, there is a data type `bool` that contains two values true and false (say T and F), and it is easy to define a function whose value is a if P = T and b otherwise (if P = F). Thus one can say one can say `(if 1<=2 then 3 else 4)`. (we assume here that we use the `ssrnat` library where $\leq$ is of type `bool` rather than `Prop`). We do not use the `bool` datatype, thus cannot use the if-then-else construction.

1.6 The theory of sets

According to Bourbaki, "the *theory of sets* is a theory which contains the relational sings =, $\in$ (of weight 2); in addition to the schemes S1 to S7 given in Chapter I, it contains the Scheme S8, and the explicit axioms A1, A2, A4, and A5. These explicit axioms contain no letters; in other words, the theory of sets is a theory *without constants*. Since the theory of sets is an equalitarian theory, the results of Chapter I are applicable."

The English version[2] is a bit different: there is a substantific sign $\supset$, of weight 2, and an axiom A3 that governs its use. If we write (x, y) instead of $\supset xy$, then for any sets x and y, the assembly (x, y) is well-formed and is a set. It is called an ordered pair; in the French version [3], a pair is defined in terms of symbols, and the axiom is replaced by a theorem (see discussion below, 2.9).

The last axiom of Bourbaki states that there exists an infinite set. It is equivalent to the existence of the set of natural numbers and will be discussed in the second part of this report. The other axioms, as well as axiom scheme S8, use the symbols $\in$, $\subset$ or $\mathrm{Coll}_x\mathrm{R}$, that are not defined in Coq. The notation $x \subset y$ is a short-hand for:

$$(\forall z)((z \in x) \implies (z \in y)).$$

If $\boldsymbol{x}$ are $\boldsymbol{y}$ are two distinct letters, and $\boldsymbol{R}$ a relation that does not depend on $\boldsymbol{y}$, the relation

$$(\exists \boldsymbol{y})(\forall \boldsymbol{x})((\boldsymbol{x} \in \boldsymbol{y}) \iff \boldsymbol{R})$$

is denoted by $\mathrm{Coll}_{\boldsymbol{x}}\boldsymbol{R}$, and read as: the relation $\boldsymbol{R}$ is collectivizing in $\boldsymbol{x}$. The first axiom (axiom of extent) in Bourbaki says:

$$(\forall x)(\forall y)((x \subset y) \text{ and } (y \subset x)) \implies (x = y).$$

We can restate it as: if x and y are two sets, then $x = y$ if and only if $z \in x$ is equivalent to $z \in y$. As a consequence, if $\mathrm{R}(x)$ is collectivizing in x, there exists a unique set y such that $x \in y$ if and only if $\mathrm{R}(x)$ is true. It is denoted by $\{x, \mathrm{R}(x)\}$, or $\{x \,|\, \mathrm{R}(x)\}$ or $\mathscr{E}_x(\mathrm{R}(x))$.

Some relations are not collectivizing, for instance $x \notin x$. In fact, if we assume that this is equivalent to $x \in y$, replacing x by y gives: $y \notin y$ is equivalent to $y \in y$, which is absurd. Almost

all sets defined by Bourbaki are obtained by application of Axiom A2 (the relation "$x = a$ or $x = b$" is collectivizing), Axiom A4 (the relation $x \subset y$ is collectivizing) or Scheme S8 (Scheme of Selection and Union); a notable exception is the set of integers, for which a special axiom is required. Scheme S8 is a bit complicated. In [5], it is replaced by the axiom

$$(\forall x)(\exists y)(\forall z)(z \in y) \iff ((\exists t)(t \in x \text{ and } z \in t))$$

that asserts the existence of the union of sets, and the following scheme (Scheme of Replacement):

> If E is a relation that depends on x, y, $a_1, \ldots, a_k$, then for all x_1, x_2, ..., x_k, if we denote by $\mathrm{R}(x, y)$ the relation $\mathrm{E}(x, y, x_1, \ldots x_n)$, the assumption $(\forall x)(\forall y)(\forall y')$ $\mathrm{R}(x, y) = \mathrm{R}(x, y') \implies y = y'$ implies that, for all t, the relation $(\exists u)(u \in t \text{ and } \mathrm{R}(u, v))$ is collectivizing in v.

The conclusion is the same as in S8. This scheme is more powerful than S8; for instance, it implies the axiom of the set of two elements A2. In fact we can deduce the existence of the empty set $\emptyset$ from this scheme (or from S8). Applying A4 to the empty set asserts the existence of a set that has a single element which is $\emptyset$, applying A4 again asserts the existence of a set t with two elements $\emptyset$ and $\{\emptyset\}$. If a and b are any elements, and $\mathrm{R}(u, v)$ is "$u = \emptyset$ and $v = a$ or $u = \{\emptyset\}$ and $v = b$", we get as conclusion: there exists a set formed solely of a and b. The assumption is clear: for fixed u, there is a unique v such that $\mathrm{R}(u, v)$. Question: can we apply S8 to this case? the answer is yes, provided that there exists a set X_a such that $a \in \mathrm{X}_a$ and a set X_b such that $b \in \mathrm{X}_b$. Such sets exist by virtue of Axiom A2. Hence A2 is required in Bourbaki, a conclusion of other axioms in [5]. The rules introduced below are closer to a Scheme of Replacement than to a Scheme of Selection and Union.

In the previous section, we have given a proof with seven lines that says $1 + 1 = 2$. The analogue proof is trivial in Coq (both objects have the same normal form `SSO`). We have also seen that the induction principle for integers in Bourbaki is the same as that of integers in Coq; as a consequence, if we can identify the Coqintegers with the integers of Bourbaki, then a lot of theorems will become trivial (i.e., are already proved by someone else). For this reason, all types, such as `nat`, will be a set. In the framework of C. Simpson, one can show that `False` is the empty set and `True` is $\{\emptyset\}$. In our framework, a set is any type whose sort is `Set`. Thus `nat` will be a set, but neither `False` nor `True` (whose sort is `Prop`).

Let $\mathbb{N}$ be the set of integers (i.e., the type `nat`) and $\mathbb{N}_2$ the set of even integers, defined as follows

```
Definition even n := ~odd n.
Lemma ed (n:nat): even(n.*2).
Lemma de n: even n -> n = (n./2).*2.

Definition N2 := { z : nat | even z}.
Definition to_N2 n :=  (exist (fun z => even z) n.*2 (@ed n)).
Lemma N2_a (z:N2): exists u, sval z = u.*2.
Lemma N2_b (z:nat): sval (to_N2 z) = z.*2.
```

We have already explained that an object y of type `N2` is the combination of an integer z (namely '`sval y`') and a proof that z is even. On the other hand, for any integer n, '`to_N2 n`' has type `N2`; its value is the double of n. We say that $\mathbb{N}_2$ is a subtype of $\mathbb{N}$ and write this as $\mathbb{N}_2 \subset \mathbb{N}$; this should be understood as: the function `sval` is an injection $\mathbb{N}_2 \to \mathbb{N}$. The Bourbaki interpretation will be: for all x, $x \in \mathbb{N}_2 \implies x \in \mathbb{N}$, where $x \in \mathbb{N}$ should be the same

as x is an integer, or, x has type `nat`. However, if x is of type $\mathbb{N}_2$, it is not a set (according to our definitions) and it is of course not of type $\mathbb{N}$. For this reason, we interpret $x \in y$ as: there is x' of type y, such that $x = \mathscr{R}_y(x')$, for some injective mapping $\mathscr{R}_y$. In this case, $\mathbb{N}_2 \subset \mathbb{N}$ is a consequence of $\mathscr{R}_{\mathbb{N}_2}(y) = \mathscr{R}_{\mathbb{N}}(\mathtt{val}(y))$. One could postulate that $\mathscr{R}_a(y) = \mathscr{R}_b(\mathtt{val}(y))$, whenever a is a sigma-type based on b; but this is not really needed. Without this postulate, the statement $\mathbb{N}_2 \subset \mathbb{N}$ might be wrong; but this causes no trouble: according to Bourbaki, there is a unique set N_2 such that $x \in \mathrm{N}_2$ if and only if n is an even integer, and we have $\mathrm{N}_2 \subset \mathrm{N}$. There is no reason why this should be equal (as a Coq object) to $\mathbb{N}_2$. One could postulate, as did Simpson, that $\mathscr{R}_{\mathbb{N}}(n)$ is the n-th von Neumann ordinal; this would imply that $\mathscr{R}2$ is `Prop`.

Notes. A reference of the form E.II.4.2 refers to [2], Theory of Sets, Chapter 2, section 4, subsection 2 (properties of union and intersection).

The document gives no proofs, except for the exercises. In order to show how difficult some theorems are, the numbers of lines of the proof is sometimes indicated in a comment.

Some statistics: there are 171 lemmas in jset, 98 in jfunc, 424 in set2 (correspondences), 364 in set3 and set31 (union; intersection, products) and 257 in set4 (equivalence relations).

In version 2, files jset and jfunc have been merged into set1, files set3 and set31 have also been merged. The number of theorems in these four files is now 279, 431, 375 and 257.

In Version 3, many trivial theorems have been removed, so that these numbers are respectively 202, 397, 338 and 242.

In Version 4, these numbers are respectively 241, 406, 322 and 241.

In Version 5, these numbers are respectively 221, 408, 318 and 241.

In Version 6, these numbers are respectively 326, 431, 297 and 227.

Chapter 2

Sets

This chapter describes the content of the file *sset1.v*, that is an adaptation of the work of C. Simpson. It is formed of several modules, that will be commented one after the other. It implements the basis of the theory of sets; this is a logical theory (as described in the previous chapter) that contains a specific sign $\in$ and some rules about its usage; we must define the COQ equivalent `inc` and the associated rules.

2.1 Module Axioms

Definitions. In our code, the term `property` denotes the type `Set→Prop`. For instance, if P is a property and x a set, then P(x) is a proposition. The term `relation` denotes the type `Set→Set→Prop`. For instance `inc` and `sub` are relations. Relations can be reflexive, symmetric, antisymmetric, transitive (in SSREFLECT, `reflexive` means $\forall x : \mathrm{T}, \mathrm{R}(x,x)$, where R has type $\mathrm{T} \to \mathrm{T} \to$ `bool`). We say that T is a functional term (in short `fterm`) if T(x) is a set, whenever x is a set. In some cases, we assume that T(x, y) is a set, whenever x and y are sets. The types `fterm`, `fterm2`, `property` and `relation` are often inferred by COQ, but we may give them explicitly for emphasis.

Let $p(x)$ be a property; if $p(x)$ and $p(y)$ imply $x = y$, then p is said *single-valued*; we consider also the case where p is a conjunction, i.e., the case where $p(x)$, $p(y)$, $q(x)$ and $q(y)$ imply $x = y$. If $p(x)$ and $p(y)$ imply $f(x) = f(y)$, then p is said *single-valued modulo* f; we sometimes say that f is constant on p.

```
Definition property := Set -> Prop.
Definition relation := Set -> Set -> Prop.
Definition reflexive_r (R: relation) := forall x, R x x.
Definition symmetric_r (R: relation) := forall x y, R x y -> R y x.
Definition transitive_r (R: relation) := forall y x z, R x y -> R y z -> R x z.
Definition antisymmetric_r (R: relation) :=  forall x y, R x y -> R y x -> x = y.

Definition fterm:= Set -> Set.
Definition fterm2:= Set -> Set -> Set.
Definition singl_val (p: property):=
   forall x y, p x -> p y -> x = y.
Definition singl_val2 (p q: property):=
   forall x y, p x -> q x -> p y -> q y -> x = y.
Definition singl_val_fp (p: property) (f: fterm) :=
  forall x y, p x -> p y -> f x = f y.
```

```
Definition exactly_one (P Q: Prop) := (P \/ Q) /\ ~(P /\ Q).
```

Some trivial lemmas. Given a conjunction A ∧ B, we get A or B via `proj1`, or `proj2`. We extend this to three-terms conjunctions. We also restate trivial properties of equivalence.

```
Lemma proj31 A B C: [/\ A, B & C] -> A.
Lemma proj32 A B C: [/\ A, B & C] -> B.
Lemma proj33 A B C: [/\ A, B & C] -> C.

Lemma iff_sym (P Q: Prop): (P <-> Q) -> (Q <-> P).
Lemma iff_trans (P Q R: Prop): (P <-> Q) -> (Q <-> R) -> (P <-> R).
Lemma iff_neg (P Q: Prop): (P <-> Q) -> ( ~ P <->  ~ Q).
```

Is element of. We assert existence of a function $\mathscr{R}$ such that, for any set x and any $y : x$, $\mathscr{R}_x(y)$ is a set. Moreover, for any set x, the function $\mathscr{R}_x$ is injective.

```
Parameter Ro : forall x : Set, x -> Set.
Axiom R_inj : forall (x : Set),  injective (@Ro x).
```

We define '$x \in y$' to be: there is an object a of type y such that $\mathscr{R}a = x$. Inclusion $x \subset y$ is defined as in Bourbaki. These two operations are called `inc` and `sub` in our framework. We also conider $x \subsetneq y$ (strict subset).

```
Definition inc (x y : Set) := exists a : y, Ro a = x.
Definition sub (a b : Set) := forall x : E, inc x a -> inc x b.
Definition ssub (a b : Set) := (sub a b) /\ (a <> b).
```

Extensionality. The axiom of extent is the same as in Bourbaki: if $x \subset y$ and $y \subset x$ then $x = y$. It could be restated as: $\subset$ is antisymmetric.

```
Axiom extensionality : antisymmetric_r sub.
```

Let X be a type; if x : X we say that x `inhabits` X and that X is `inhabited`. If X is a set, we say `nonempty` instead of `inhabited` (in fact, thanks to $\mathscr{R}$, the two properties are equivalent).

```
CoInductive nonempty (x : Set) : Prop :=
    nonempty_intro : forall y : Set, inc y x -> nonempty x.
```

The axiom of choice. The restricted form of the axiom of choice says that there is a function c, such that $c(y) \in y$, whenever y is a non-empty set. We can generalize it to any predicate $p(x)$, and any type t.

Assume that t is any type, p of type $t \to$ `Prop`, and q proves that t is inhabited. We assert the existence of a function $\mathscr{C}_\mathrm{T}$ such that $c = \mathscr{C}_\mathrm{T}(p, q)$ is of type t, and if there exists an object x of type t such that $p(x)$ is true, then $p(c)$ is true.

```
Section Choose.
Variable (t : Type).
Implicit Type (p : t -> Prop)  (q:inhabited t).

Parameter chooseT : forall p q, t.
Axiom chooseT_pr : forall p q, ex p -> p (chooseT p q).

End Choose.
```

Images. The scheme of selection and union is the following: Given a relation R(x, y); assume that for fixed y, we have a set E_y such that R(x, y) implies $x \in E_y$. Then, for every Y, there is a set Z_Y containing all x for which there is an $y \in Y$ such that R(x, y). A simple case is when R is independent of y. Another simple case is when R has the form $x = f(y)$. It is called the *axiom of replacement*. The axiom of the set of two elements (shown later) says that we can select $E_y = \{f(y)\}$. As a consequence the image of a set by a function is a set. We define here a parameter `IM`, and the corresponding axiom.

```
Parameter IM : forall x : Set, (x -> Set) -> Set.

Axiom IM_P :  forall (x : Set) (f : x -> Set) (y : Set),
          inc y (IM f) <-> exists a : x, f a = y.
```

Double negation axiom. The excluded middle axiom says that P or its negation is true. This is equivalent to say that the double negation of P implies P.

```
Axiom p_or_not_p: forall P:Prop,  P \/ ~ P.

Lemma equal_or_not (x y:Set): x = y \/ x <> y.
Lemma inc_or_not (x y:Set): inc x y \/ ~ (inc x y).
Lemma excluded_middle (p:Prop): ~ ~ p -> p.
```

2.2 Module constructions

These lemmas say that $x \subset x$, and if $x \subset y$ and $y \subset z$, then $x \subset z$; if one $\subset$ is replaced by $\subsetneq$ in the assumption, then the same holds in the conclusion.

```
Lemma sub_refl x: reflexive_r sub.
Lemma sub_trans b: transitive_r sub.
Lemma ssub_trans1 b a c: ssub a b -> sub b c -> ssub a c.
Lemma ssub_trans2 b a c: sub a b -> ssub b c -> ssub a c.
```

Empty sets. We say that a set is `empty` if it has no element; by extensionality, x is empty if and only if it is equal to $\emptyset$. Bourbaki proves existence of $\emptyset$ by considering the complement of x in itself. In Coq, the situation is simpler: we define $\emptyset$ as a type without constructor, hence there is no $a \in x$, since there is no $b : x$.

```
Definition empty (x : Set) := forall y : Set, ~ inc y x.
CoInductive emptyset : Set :=.
```

By excluded middle, a set is empty or nonempty. We deduce that non-empty is the same as not empty.

```
Lemma R_inc (x : Set) (a : x): inc (Ro a) x.

Lemma in_set_0 x: ~ inc x emptyset.
Lemma set0_P x: empty x <-> x = emptyset.
Lemma not_nonempty_empty: ~(nonempty emptyset).
Lemma emptyset_dichot x: x = emptyset \/ nonempty x.
Lemma nonemptyP x: nonempty x <-> (x <> emptyset).

Lemma sub_0set x: sub emptyset x.
Lemma sub_set0 x: sub x emptyset <-> (x = emptyset).
```

An inverse for $\mathscr{R}$. We define a function $\mathscr{B}$ that takes 3 arguments, x, y and H, two sets and a proof of $x \in y$. The first two arguments are implicit: they are deduced from the type of H. We shall sometimes write $\mathscr{B}(\mathrm{H} : x \in y)$. The function uses the axiom of choice $\mathscr{C}_{\mathrm{T}}(p,q)$ to select an object a of type y such that $p(a)$, namely $\mathscr{R}a = x$. Assumption H says that such an object exists, and as a consequence it implies q, a proof that the type y is inhabited. Thus $p(\mathscr{B})$ is true, i.e.,

$$\mathscr{R}(\mathscr{B}(\mathrm{H} : x \in y)) = x.$$

If we replace x by $\mathscr{R}z$, we get $\mathscr{R}(\mathscr{B}(\mathrm{H})) = \mathscr{R}z$, hence, by injectivity

$$\mathscr{B}(\mathrm{H} : \mathscr{R}z \in y) = z.$$

```
Definition Bo (x y : Set) (hyp : inc x y) :=
 chooseT (fun a : y => Ro a = x)
    (match hyp with | ex_intro w _ => inhabits w end)

Lemma B_eq x y (hyp : inc x y): Ro (Bo hyp) = x.
Lemma B_back (x:Set) (y:x) (hyp : inc (Ro y) x): Bo hyp = y.
```

Axiom of choice for sets. Let p be a property of sets. Since the empty set is a set, we get a function $\mathscr{C}(p)$, such that $p(\mathscr{C}(p))$ is true whenever there is a set x satisfying p. Note that if p and q are equivalent properties, we do not pretend that $\mathscr{C}(p) = \mathscr{C}(q)$. Thus $\mathscr{C}(p)$ is not equivalent to Bourbaki's τ.

```
Definition choose (p: property) :=  chooseT p (inhabits emptyset).
Lemma choose_pr p: ex p -> p (choose p).
```

Representatives of nonempty sets. If we apply the axiom of choice to $x \in y$ we get: there is a function $r(x)$ such that $r(x) \in x$, for every nonempty set x. It will be denoted by `rep z`.

```
Definition rep (x : Set) := choose (fun y : Set => inc y x).
Lemma rep_i x: nonempty x -> inc (rep x) x.
```

Set of elements such that P. In Bourbaki, the "Scheme of selection and union" is the following : we have four distinct variables x, y, X and Y, and a relation R that depends on x and y, but not on X, Y. The assumption is $\forall y, \exists \mathrm{X}, \forall x, \mathrm{R} \implies x \in \mathrm{X}$. The conclusion is that for every Y, the relation $\exists y, y \in \mathrm{Y} \wedge \mathrm{R}$ is collectivizing in x. Said otherwise, for every Y, there is a set Z such that $x \in \mathrm{Z}$ is equivalent to the existence of $y \in \mathrm{Y}$ such that R. A simple case is when R does not depend on y. Then, the assumption $\forall x, \mathrm{R}(x) \implies x \in \mathrm{X}$ implies the existence of Z such that $x \in \mathrm{Z}$ is equivalent to $\mathrm{R}(x)$. In particular, if $\mathrm{Q}(x)$ is any relation, there is a set Z such that $x \in \mathrm{Z}$ is equivalent to $x \in \mathrm{Y} \wedge \mathrm{Q}(x)$. Here is the Coq implementation.

```
CoInductive Zorec (x : Set) (f : x -> Prop) : Set :=
   Zorec_c : forall a: x, f a -> Zorec f.
Definition Zo (x:Set) (p:property) :=
  IM (fun  (z : Zorec (fun (a : x) => p (Ro a))) => let (a, _) := z in Ro a).
```

Consider a set X and a function f defined on the type X. We shall later on consider the case where f takes its values in `Set`. An object is of type '`Zorec X f`' if and only if it is a pair (a,b) where a is of type X, and b of type $f(a)$ and such a pair is created by `Zorec_c`. When we

construct a pair, we have to provide a proof of $f(a)$, and when we destruct the pair, we can extract the proof.

The construction '`let (a,_) := z in F`' means: replace in F all occurrences of a by the first field of the instance z. Using `IM`, we thus get the set of all a : X satisfying f. If $p(x)$ is a predicate defined for $x \in$ X, and $f(a) = p(\mathscr{R}a)$, we get the set of all $x \in$ X satisfying p.

The set is denoted in Bourbaki by $\mathscr{E}_x$(P and $x \in$ A). In the French version, it is denoted by $\{x \mid$ P and $x \in$ A$\}$; Bourbaki notes that this may be abbreviated as $\{x \in$ A $\mid$ P$\}$.

```
Lemma Zo_i x (p: property) y: inc y x -> p y -> inc y (Zo x p).
Lemma Zo_hi x (p: property) y: inc y (Zo x p) -> p y.
Lemma Zo_S x (p: property) : sub (Zo x p) x.
Lemma Zo_P x (p: property) y : inc y (Zo x p) <-> (inc y x /\ p y).
```

We have $\{x \in$ X$, p(x)\} = \{x \in$ X$, q(x)\}$, whenever p and q are equivalent in X.

```
Lemma Zo_exten1 (X : Set) (p q: property):
   (forall x, inc x X-> (p x <-> q x)) -> Zo X p = Zo X q.
Lemma Zo_exten2 (X Y: Set) (p q: property):
   (forall x, (inc x X /\ p x <-> inc x Y /\ q x)) -> Zo X p = Zo Y q.
```

2.3 Module Little

Given two sets x and y, we construct a set, a *doubleton*, denoted by $\{x, y\}$, satisfying $z \in \{x, y\} \iff z = x \vee z = y$, as the image of `bool` by the function f that associates x to true, and y to false. Bourbaki uses Axiom A2 to show existence of such a set.

```
Definition doubleton (x y : Set) :=
  IM (fun z => if z then x else y).

Lemma set2_1  x y: inc x (doubleton x y).
Lemma set2_2  x y: inc y (doubleton x y).
Lemma set2_hi z x y: inc z (doubleton x y) -> z = x \/ z = y.
Lemma set2_P z x y : inc z (doubleton x y) <-> (z = x \/ z = y).

Lemma doubleton_inj x y z w :
    doubleton x y = doubleton z w ->  (x = z /\ y = w) \/  (x = w /\ y = z).
Lemma set2_ne x y: nonempty (doubleton x y).
Lemma sub_set2 x y z: inc x z -> inc y z -> sub (doubleton x y) z.
Lemma set2_C : commutative doubleton.
Lemma set2_pr a b X:
  inc a X -> inc b X ->
  (forall z : Set, inc z X -> z = a \/ z = b)
  -> X = doubleton a b.
```

The set '`doubleton x x`' is called a *singleton* and denoted $\{x\}$. By construction $z \in \{x\} \iff z = x$. From this one can deduce that a singleton is nonempty, and we have an extensionality property.

```
Definition singleton x := doubleton x x.

Lemma set1_1 x: inc x (singleton x).
```

```
Lemma set1_eq x y: inc y (singleton x) -> y = x.
Lemma set1_inj: injective singleton.
Lemma set1P x y: inc y (singleton x) <-> (y = x).
Lemma set1_sub x X: inc x X -> sub (singleton x) X.
Lemma set1_ne x: nonempty (singleton x).
```

In the original version, we introduced a set TP with two elements TPa, TPb. We use here the elements $\emptyset$ and $\{\emptyset\}$, renamed as C0 and C1 and the set will be called C2.

```
Definition C0 := emptyset.
Definition C1 := singleton C0.
Definition C2 := doubleton C0 C1.

Lemma C1_p1 x:  inc x C1 <-> x = C0.
Lemma TP_ne: C0 <> C1.
Lemma TP_ne: C1 <> C0.
Lemma C2_P x:  inc C2 <-> (x = C0 \/ x = C1).
Lemma inc_C0_2: inc C0 C2.
Lemma inc_C1_2: inc C1 C2.
```

We say that x is a *small set* if $a \in x$ and $b \in x$ imply $a = b$. It is either empty or has a single element.

```
Definition singletonp x := exists u, x = singleton u.
Definition doubletonp x:= exists a b, a <> b /\ x = doubleton a b.

Definition small_set x :=  singl_val (inc ^~ x).

Lemma set1_pr x X: inc x X -> (forall z, inc z X -> z = x) ->
  X = singleton x.
Lemma set1_pr1 x X: nonempty X -> (forall z, inc z X -> z = x) ->
  X = singleton x.

Lemma small0: small_set emptyset.
Lemma small1 x: small_set (singleton x).
Lemma singletonP x: singletonp x <-> (nonempty x /\ small_set x).
Lemma small_set_pr x: small_set x -> x = emptyset \/ singletonp x.
Lemma subset1P A x: sub A (singleton x) <-> (A = emptyset \/ A = singleton x).
Lemma sub1setP A x : sub (singleton x) A <-> inc x A.
```

2.4 Module Image

If f is a mapping, x a set, we denote the image of x by f as $f\langle x\rangle$.

```
Definition fun_image (x : Set) (f : fterm) := IM (fun a : x => f (Ro a)).

Lemma funI_i x f a: inc a x -> inc (f a) (fun_image x f).
Lemma funI_P f x y:
  inc y (fun_image x f) <-> exists2 z, inc z x & y = f z.
```

2.5 Module Complement

If A and B are two subsets of E, the *complement* of A (in E) is the set of all x in E that are not in A; it is denoted by $\bar{A}$ or $\complement A$, or $\complement_E A$. The set $B \cap \bar{A}$ is the set of all x in B that are not in A; it is called the *set difference*. It is independent of E, and denoted by $B \setminus A$ or sometimes $B - A$. We shall not distinguish between these two notions.

By excluded middle, if $x \in B$ and $x \notin B - A$, then $x \in A$. It follows that if $B - A$ is empty, then $B \subset A$.

```
Definition complement (A B : Set) := Zo A (fun x : Set => ~ inc x B).

Lemma setC_P A B x: inc x (A -s B) <-> (inc x A /\ ~ inc x B).
Lemma setC_i x A B: inc x A -> ~ inc x B -> inc x (A -s B).
Lemma nin_setC x A B: inc x A -> ~ inc x (A -s B) -> inc x B.
Lemma empty_setC A B: A -s B = emptyset -> sub A B.
Lemma setC_T A B: sub A B -> A -s B = emptyset.
```

These lemmas are obvious. If $A \subset E$ then $E - (E - A) = A$. We have $E - E = \emptyset$ and $E - \emptyset = E$. If $A \subset X$ and $B \subset X$, then $X \setminus A \subset X \setminus B$ if and only if $B \subset A$.

```
Lemma sub_setC A B: sub (A -s B) A.
Lemma setC_ne A B: ssub A B  -> nonempty (B -s A).
Lemma setC_K A B: sub A B -> B -s (B -s A) = A.
Lemma setC_v A: A -s A = emptyset.
Lemma setC_0 A: A -s emptyset = A.
Lemma set_SC A B C : sub A B -> sub (A -s C) (B -s C).
Lemma set_CS A B C : sub A B -> sub (C -s B) (C -s A).
Lemma set_CSS A B C D : sub A C -> sub D B -> sub (A -s B)(C -s D).
Lemma set_CSm A B X: sub A X -> sub B X ->
  (sub A B <-> sub (X -s B) (X -s A)).
Lemma subsetC_P A B E : sub A E -> sub B E ->
 ( (sub A (E -s B)) <-> (sub B (E -s A))).
Lemma subCset_P A B E:sub A E -> sub B E ->
 ( (sub (E -s A) B) <-> (sub (E -s B) A)).
```

We study some properties of $X - \{a\}$.

```
Notation "a -s1 b" := (a -s (singleton b))  (at level 50).

Lemma setC1_P x A b: inc x (A -s1 b) <-> (inc x A /\ x <> b).
Lemma setC1_1 x A:  ~ (inc x (A -s1 x)).
Lemma setC1_proper A x : inc x A -> ssub (A -s1 x) A.
Lemma setC1_eq a X: ~(inc a X) -> X -s1 a = X.
```

2.6 Module Union

Bourbaki defines the *union* $\bigcup_{\iota \in I} X_\iota$ of a family of sets. This means that we have a set I and a mapping $\iota \mapsto X_\iota$ defined for $i \in I$. The union exists as a direct consequence of S8 (Scheme of Selection and Union). The assumption is $(\forall i)(\exists Z)(\forall x)(i \in I \text{ and } x \in X_i) \implies x \in Z$ (take $Z = X_i$). The conclusion is the existence of a set containing all elements satisfying $(\exists i)(i \in I \text{ and } x \in X_i)$. We use the same construction as in the case of Selection (compare with `Zorec`).

```
Section UnionDef.
Variable (I:Set)(f : I->Set).

CoInductive Uaux : Set :=
    Uaux_c : forall a:I, f a -> Uaux.
Definition uniont  :=
   IM (fun a : Uaux  => (let:  Uaux_c u v := a in @Ro (f u) v)).
End UnionDef.
```

Assume that I is a set, and f a function defined on the type I. We use a record, that holds (a,b) for all a : X, where b is of type $f(a)$. By definition of $\mathscr{R}$ we have $\mathscr{R}b \in f(a)$. Note that the implicit argument of $\mathscr{R}$ must be given explicitly here. The set of all these $\mathscr{R}b$ is a set U such that $x \in$ U is equivalent to $\exists a$, $x \in f(a)$. This set will be called `uniont`. Let X be a set and $f = \mathscr{R}_X$. Since a : X is the same as $\mathscr{R}a \in$ X, then $x \in$ U if and only if there is $b \in$ X such that $x \in b$. This set will be called `union`, and denoted by $\bigcup$X.

```
Definition union X := uniont (@Ro X).
Lemma setUt_P (I:Set) (f:I->Set) x:
  inc x (uniont f) <-> exists z,  inc x (f z).
Lemma setU_P X x:
  inc x (union X) <-> exists2 z,  inc x z & inc z X.
```

Some properties of union.

```
Lemma setU_i x y a: inc x y -> inc y a -> inc x (union a).
Lemma setU_hi x a: inc x (union a) -> exists2 y,  inc x y & inc y a.
Lemma setU_s1 x y: inc x y -> sub x (union y).
Lemma setU_s2 x z: (forall y, inc y z -> sub y x)-> sub (union z) x.
Lemma setU_0: union emptyset = emptyset.
```

The union a family of two sets X and Y is denoted by X $\cup$ Y. An element is in the union if and only if it is in one of the sets. We have A $\subset$ A $\cup$ B and B $\subset$ A $\cup$ B, and other properties.

```
Definition union2 (x y : Set) := union (doubleton x y).

Notation "a \cup b" := (union2 a b) (at level 50).

Lemma setU2_hi x y a: inc a (x \cup y) -> inc a x \/ inc a y.
Lemma setU2_1 x y a: inc a x -> inc a (x \cup y).
Lemma setU2_2 x y a: inc a y -> inc a (x \cup y).
Lemma setU2_P a b x: inc x (a \cup b) <-> (inc x a \/ inc x b).
Lemma subsetU2l a b: sub a (a \cup b).
Lemma subsetU2r a b: sub b (a \cup b).

Lemma setU2_C: commutative union2.
Lemma setU2_id: idempotent union2.
Lemma setU2_A: associative union2.
Lemma setU2_CA : left_commutative union2.
Lemma setU2_AC : right_commutative union2.
Lemma set2_UUl : left_distributive union2 union2.
Lemma set2_UUr : right_distributive union2 union2.

Lemma setU2_S1 A B C : sub A B -> sub (C \cup A) (C \cup B).
Lemma setU2_S2 A B C : sub A B -> sub (A \cup C) (B \cup C).
```

```
Lemma setU2_SS A B C D : sub A C -> sub B D -> sub (A \cup B) (C \cup D).
Lemma setU2_12S A B C: sub A C -> sub B C -> sub (A \cup B) C.
Lemma subU2_setP A B C : (sub (B\cup C) A) <-> (sub B A /\ sub C A).
Lemma sub_setU2 A B C : (sub A B) \/ (sub A C) -> sub A (B \cup C).
Lemma setU2id_Pl A T: sub A T <-> A \cup T = T.
Lemma setU2id_Pr A T: sub A T <-> T \cup A = T.
Lemma setU2_0 A : A \cup emptyset  = A.
Lemma set0_U2 A : emptyset \cup A = A.

Lemma setU_1 x: union (singleton x) = x.
Lemma setU2_11 x y: (singleton x) \cup (singleton y) = doubleton x y.
Lemma setU2_eq0P A B : (A \cup B = emptyset) <-> (A = emptyset /\ B = emptyset).
Lemma subCset_P2 A B C : (sub (A -s B) C) <-> (sub A (B \cup C)).

Lemma setU2Cr1 A B:  A \cup (A -s B) = A.
Lemma setU2Cr2 A B:  A \cup (B -s A) = A \cup B.
Lemma setU2_Cr A T: sub A T -> A \cup (T -s A) = T.
```

In some cases (induction on finite sets), one needs to consider the union of a set and a singleton.

```
Notation "a +s1 b" := (a \cup (singleton b)) (at level 50).

Lemma setU1_P x y z:  (inc z (x +s1 y) ) <-> (inc z x \/ z = y).
Lemma setU1_1 a b: inc a (b +s1 a).
Lemma sub_setU1 a b: sub b (b +s1 a).
Lemma setU1_r a b y: inc y b -> inc y (b +s1 a).
Lemma setU1_eq x y: inc y x -> x +s1 y = x.
Lemma setU1_sub x y z: sub x z -> inc y z -> sub (x +s1 y) z.
Lemma setCU_K x y: sub y x <-> (x -s y) \cup y = x.
Lemma setC1_K x y: inc y x -> (x -s1 y) +s1 y = x.
Lemma setU1_K a x: ~(inc a x) ->  (x +s1 a) -s1 a = x.
Lemma setU1_inj x A B: ~(inc x A) -> ~(inc x B) -> A +s1 x = B +s1 x -> A = B.
Lemma setC1_inj x A B: inc x A -> inc x B -> A -s1 x = B -s1 x -> A = B.
```

If we add an element to a doubleton we get a tripleton.

```
Definition tripleton a b c := (doubleton a b) +s1 c.
Lemma set3_P a b c x:
   inc x (tripleton a b c) <-> [\/ x = a , x = b | x = c].
```

The union of x and $\{x\}$ will be denoted later on by x^+, and called the successor of x. The successor of `C2` will be `C3` and successor of `C3` will be `C4`. The sets `C3` and `C4` have exactly three and four distinct elements.

```
Definition C3 := C2 +s1 C2.
Definition C4 := C3 +s1 C3.

Lemma C3_P x: inc x C3 <-> [\/ x = C0, x = C1 | x = C2].
Lemma C4_P x: inc x C4 <-> [\/ x = C0, x = C1, x = C2 | x = C3].
Lemma C2_neC01: C2 <> C0 /\ C2 <> C1.
Lemma C3_neC012: [/\ C3 <> C0, C3 <> C1 & C3 <> C2].
```

The direct image of a set by a function is compatible with union.

```
Lemma funI_set0 f: fun_image emptyset f = emptyset.
Lemma funI_setne f x: nonempty x ->  nonempty (fun_image x f).
Lemma funI_setne1 f x: fun_image x f = emptyset ->  x = emptyset.
Lemma funI_set2 f a b:
  fun_image (doubleton a b) f = doubleton (f a) (f b).
Lemma funI_set1 f x: fun_image (singleton x) f = singleton (f x).

Lemma funI_setU f X:
   fun_image (union X) f = union (fun_image X (fun z => (fun_image z f))).
Lemma funI_setU2 f:  {morph (fun_image ^~ f): x y / x \cup y}.
Lemma funI_setU1 g X a:
   fun_image (X +s1 a) g = fun_image X g  +s1  (g a).
Lemma funI_S f a b: sub a b -> sub (fun_image a f) (fun_image b f).
```

Variant of the axiom of choice. Let E be any set, $p(x)$ be a property, $\mathrm{F} = \{x \in \mathrm{E}, p(x)\}$ and $z = \bigcup \mathrm{F}$. If there is a unique x in E that satisfies p, then $\mathrm{F} = \{x\}$, and $z = x$ so that $p(z)$ holds. This quantity z is denoted by '`select p E`'. Assume that p depends on a parameter y and $p(x)$ implies $x \in f(y)$. Then '`select (p y) (f y)`' is the same as '`choose p y`'. Whenever possible, we use `select` rather then `choose`.

```
Definition select (p: property) (E: Set) := union (Zo E p).

Lemma select_uniq (p: property) E:
  (singl_val2 (inc ^~ E) p) ->
  forall x, inc x E -> p x -> x = (select p E).
Lemma select_pr (p : property) E:
  (exists2 x, inc x E & p x) ->
(singl_val2 (inc ^~ E) p) ->
  (p (select p E) /\ inc (select p E) E).
```

Conditional definition. Let x and y be two sets, $\mathrm{E} = \{x, y\}$, P any property and $p(z)$ be

$$(z = x \text{ and } \mathrm{P}) \text{ or } (z = y \text{ and } \neg \mathrm{P})$$

If $p(z)$ holds then z is is either x or y, thus is in E. Moreover, there is at most one element in E satisfying p. More precisely, assume P true. Then $p(z)$ implies $z = x$ and $p(x)$ holds; assume P false; then $p(z)$ implies $z = y$ and $p(y)$ holds.

This allows us to construct a function $\mathscr{Y}(\mathrm{P}, x, y)$ which is x if P holds and y otherwise (the `Ytac` tactic uses the fact that either P or $\neg$P hold).

```
Definition Yo (P : Prop) (x y : Set) :=
  select (fun z => (z = x /\ P) \/ (z = y /\  ~P)) (doubleton x y).
Lemma Y_true  (P:Prop) (hyp :P) x y: Yo P x y = x.
Lemma Y_false  (P:Prop) (hyp : ~P) x y: Yo P x y = y.
Lemma Y_same (P: Prop) x : Yo P x x = x.
```

2.7 Module Powerset

Bourbaki introduces an axiom that says that for every set x, there is a set y, the *powerset* of x, denoted $\mathfrak{P}(x)$ containing the subsets of x. This set is canonically isomorphic to the set of functions $x \to \mathrm{X}$, where X is a set with two elements A and B (to each function f, associate

$f^{-1}\langle A\rangle$). The set of (graphs of) functions $x \to X$ is a subset of $\mathfrak{P}(x \times X)$. Thus existence of the powerset is equivalent to existence of sets of functions. We consider here the *type of functions* rather than the set of functions.

We shall denote here by 0, 1 and 2 the sets $\emptyset$, $\{\emptyset\}$ and $\{0,1\}$. Let q be a function $x \to 2$. Thus, whenever $z \in x$ we have $q(z) \in 2$. We construct a function p of type $x \to 2$ as follows: if $t : x$, then $\mathscr{R}(t) \in x$, so that $q(\mathscr{R}(t)) \in 2$, and applying $\mathscr{B}$ to this gives an object of type 2. We define $q^{-1}(0)$ as the set of all $z \in x$ such that $q(z) = 0$. This set is also denoted by $p^{-1}(0)$. If H says $z \in x$, then $z \in p^{-1}(0)$ is equivalent to $\mathscr{R}(p(\mathscr{B}H)) = 0$. Let X be the set of all $p^{-1}(0)$. An element of X is a subset of x.

Let y be a subset of x, and q the function that maps z to 0 if $z \in y$ and to 1 otherwise (we use the function $\mathscr{Y}$ defined above). Then $y = p^{-1}(0)$, so that $y \in X$. In other words, X is the power set of x.

```
Definition powerset (x : Set) :=
 IM (fun p : x -> C2 =>
    Zo x (fun y : Set => forall hyp : inc y x, Ro (p (Bo hyp)) = C0)).
```

All properties but the first are trivial. Note that the canonical doubleton is just $\mathfrak{P}(\mathfrak{P}(\emptyset))$.

```
Lemma setP_i x y: sub x y -> inc x (powerset y).
Lemma setP_hi x y: inc x (powerset y) -> sub x y.
Lemma setP_P x y: inc x (powerset y) <-> sub x y.
Lemma setP_Ti x: inc x (powerset x).
Lemma setP_0i x: inc emptyset (powerset x).
Lemma setP_S a b:  sub a b <-> sub (powerset a) (powerset b).
Lemma setP_0: powerset emptyset = singleton emptyset.
Lemma setP_1 x: powerset (singleton x) = doubleton emptyset (singleton x).
Lemma setP_00 : powerset (powerset emptyset) = C2.
```

2.8 Module Intersection

Bourbaki defines the *intersection* of a family of sets $(X_\iota)_{\iota\in I}$ as the dual of union. We have $x \in \bigcap_{\iota\in I} X_\iota$ if and only if x is in every element of the family. We consider here the case (denoted by $\bigcap X$) where the mapping $\iota \mapsto X_\iota$ is the identity of X, the general case will be studied in a future Chapter. We have then

$$\bigcap X = \{x \in E, \forall a, a \in X \implies x \in a\}$$

where E is any adequate set. If the family is empty, then Bourbaki defines the intersection to be E. We do not like this definition, since it depends on the context. Taking for E the union of the family solves the problem, it defines the intersection of an empty family to be empty.

```
Definition intersection (x : Set) :=
  Zo (union x) (fun y : Set => forall z : Set, inc z x -> inc y z).

Lemma setI_0: intersection emptyset = emptyset.
Lemma setI_i x a:
  nonempty x -> (forall y, inc y x -> inc a y) -> inc a (intersection x).
Lemma setI_hi x a y: inc a (intersection x) -> inc y x -> inc a y.
Lemma setI_s1 x y: inc y x -> sub (intersection x) y.
```

The intersection of two sets is denoted $X \cap Y$, the properties listed here are obvious.

```
Definition intersection2 (x y : Set) := intersection (doubleton x y).

Notation "a \cap b" := (intersection2 a b) (at level 50).

Lemma setI2_i x y a:  inc a x -> inc a y -> inc a (x \cap y).
Lemma setI2_1 x y a: inc a (x \cap y) -> inc a x.
Lemma setI2_2  x y a: inc a (x \cap y) -> inc a y.
Lemma setI2_P x y a: inc a (x \cap y) <-> (inc a x /\ inc a y).
Lemma subsetI2l a b: sub (a \cap b) a.
Lemma subsetI2r a b: sub (a \cap b) b.
Lemma setI2_id: idempotent intersection2.
Lemma setI_1 x:  intersection (singleton x) = x.
Lemma setI2_C: commutative intersection2.
Lemma setI2_A: associative intersection2.
Lemma setI2_CA : left_commutative intersection2.
Lemma setI2_AC : right_commutative intersection2.
Lemma set2_IIl : left_distributive intersection2 intersection2.
Lemma set2_IIr : right_distributive intersection2 intersection2.

Lemma setI2_S1 A B C : sub A B -> sub (C \cap A) (C \cap B).
Lemma setI2_S2 A B C : sub A B -> sub (A \cap C) (B \cap C).
Lemma setI2_SS A B C D : sub A C -> sub B D -> sub (A \cap B) (C \cap D).
Lemma setI2_12S A B C: sub C A -> sub C B -> sub C (A \cap B).
Lemma subsetI2_P A B C : sub C (A\cap B)  <-> (sub C A /\ sub C B).
Lemma subI2_set A B C : (sub A C) \/ (sub B C) -> sub (A \cap B) C.
Lemma subsetI2_P A B C : sub C (A\cap B)  <-> (sub C A /\ sub C B).
Lemma setI2id_Pl A T: sub A T <-> A \cap T = A.
Lemma setI2id_Pr A T: sub A T <-> T \cap A = A.
```

We state here some distributivity properties.

```
Lemma set_UI2l: left_distributive union2 intersection2.
Lemma set_UI2r: right_distributive union2 intersection2.
Lemma set_IU2l: left_distributive intersection2 union2.
Lemma set_IU2r: right_distributive intersection2 union2.
Lemma setPI : morphism_2 powerset intersection2 intersection2.

Lemma set_U2K A B: (A \cup B) \cap A = A.
Lemma set_K2U A B: A \cap (B \cup A) = A.
Lemma set_I2K A B: (A \cap B) \cup A = A.
Lemma set_K2I A B: A \cup (B \cap A) = A.
Lemma setU2_ni x A B: ~inc x (A\cup B) -> (~ inc x A  /\ ~ inc x B).
Lemma setI2_ni x A B: ~inc x (A\cap B) -> (~ inc x A \/ ~ inc x B).
Lemma setC_ni x A B: ~inc x (A -s B) -> (~ inc x A \/ inc x B).

Lemma set_CU2 A B X: X -s (A\cup B) = (X -s A) \cap (X -s B).
Lemma set_CI2 A B X: X -s (A\cap B) = (X -s A) \cup (X -s B).
Lemma setCI2_pr1 A B E: sub A E -> A -s  B = A \cap (E -s B).
Lemma set_CC A B E: sub B E -> (E -s (A -s B)) = (E -s A) \cup B.
Lemma setI2_Cr A B : (A \cap B) \cup (A -s B) = A.

Lemma setCU2_l A B C : (A \cup B) -s C = (A -s C) \cup (B -s C).
Lemma setCU2_r A B C : A -s (B \cup C) = (A -s B) \cap (A -s C).
Lemma setCI2_l A B C : (A \cap B) -s C = (A -s C) \cap (B -s C).
```

```
Lemma setCI2_r A B C : A -s (B \cap C) = (A -s B) \cup (A -s C).
Lemma setCC_l A B C : (A -s B) -s C = A -s (B \cup C).
Lemma setCC_r A B C : A -s (B -s C) = (A -s B) \cup (A \cap C).
```

We say that two sets are *disjoint* if the intersection is empty. Here are some properties.

```
Definition disjoint (x y: Set) := x \cap y = emptyset.
Definition disjointVeq (x y: Set) := x = y \/ disjoint x y.
Lemma disjoint_pr a b:
  (forall u, inc u a -> inc u b -> False) -> disjoint a b.
Lemma nondisjoint a b c: inc a b -> inc a c -> ~ disjoint b c.
Lemma disjointVeq_pr x y z: disjointVeq x y -> inc z x -> inc z y -> x = y.

Lemma setI2_0 A : disjoint A emptyset.
Lemma set0_I2 A : disjoint emptyset A.
Lemma set_IC1r A B:  A \cap (A -s B) =  A -s B.
Lemma set_I2Cr A B: disjoint B (A -s B).
Lemma disjoint_S: symmetric_r disjoint.
```

Lemmas using disjoint and complement.

```
Lemma subsets_disjoint_P A B E: sub A E ->
  (sub A B <-> disjoint A (E -s B)).
Lemma disjoint_subsets_P A E: sub A E -> forall B,
  (disjoint A B <-> sub A (E -s B)).
Lemma setCId_Pl A B: A -s B = A <->  disjoint A B.
Lemma subCset_P3 A B C : sub A (B -s C) <-> ((sub A B) /\ (disjoint A C)).
Lemma subsetC1_P A B x: (sub A (B -s1 x)) <-> (sub A B /\ ~inc x A).
Lemma properI2_r A B : ~(sub B A) -> ssub (A \cap B) B.
Lemma properI2_l A B : ~(sub A B) -> ssub (A \cap B) A.
Lemma properU2_r A B : ~(sub A B) -> ssub B (A \cup B).
Lemma properU2_l A B : ~(sub B A) -> ssub A (A \cup B).
Lemma properI2_set A B C : (ssub B A) \/ (ssub C A) -> (ssub (B \cap C) A).
Lemma properI2 A B C : (ssub A (B \cap C)) -> (ssub A B /\ ssub A C).
Lemma properU2 A B C : (ssub (B \cup C) A) -> (ssub B A /\ ssub C A).
```

2.9 Module Pair

We define here three functions, `kpair`, `kpr1`, and `kpr2`, and a hack that makes them completely opaque. Some comments can be found in section 8.3.

```
Definition kpair x y := doubleton (singleton x) (doubleton x y).
Definition kpr1 x := union (intersection x).
Definition kpr2 x :=
   union (Zo (union x) (fun z => (doubleton (kpr1 x) z) = (union x))).
```

This is the hack.

```
Module Type PairSig.
Parameter first_proj second_proj : Set -> Set.
Parameter pair_ctor : Set -> Set -> Set.
Axiom kpr1E: first_proj = kpr1.
Axiom kpr2E: second_proj = kpr2.
```

```
Axiom kprE: pair_ctor = kpair.
End PairSig.

Module Pair : PairSig.
Definition pair_ctor := kpair.
Definition first_proj := kpr1.
Definition second_proj := kpr2.
Lemma kprE: pair_ctor = kpair. Proof. by []. Qed.
Lemma kpr1E: first_proj = kpr1. Proof. by []. Qed.
Lemma kpr2E: second_proj = kpr2. Proof. by []. Qed.
End Pair.
```

The three functions defined above will be renamed as J, P and Q. The usual notations are (x, y), $\mathrm{pr}_1 z$ and $\mathrm{pr}_2 z$.

```
Notation J := Pair.pair_ctor.
Notation P := Pair.first_proj.
Notation Q := Pair.second_proj.
```

The important properties are: if $z = (x, y)$, then $\mathrm{pr}_1 z = x$ and $\mathrm{pr}_2 z = y$.

```
Lemma pr1_pair x y: P (J x y) = x.
Lemma pr2_pair x y: Q (J x y) = y.
```

It follows: if $(x, y) = (x', y')$ then $x = x'$ and $y = y'$. We say that z is a *pair* is z has the form (x, y), for some x and y, in particular, if $z = (\mathrm{pr}_1 z, \mathrm{pr}_2 z)$.

The properties of projectors say that $(a, b) = (c, d)$ implies $a = c$ and $b = d$. We say that x is a pair if $(\mathrm{pr}_1 x, \mathrm{pr}_2 x) = x$. Any (a, b) is a pair. Two pairs x and y are equal if and only if $\mathrm{pr}_1 x = \mathrm{pr}_1 y$ and $\mathrm{pr}_2 x = \mathrm{pr}_2 y$.

```
Definition pairp x  := J (P x) (Q x) = x.

Lemma pr1_def a b c d: J a b = J c d -> a = c.
Lemma pr2_def a b c d: J a b = J c d -> b = d.
Lemma pair_is_pair x y : pairp (J x y).
Lemma pair_exten a b:
  pairp a -> pairp b -> P a = P b -> Q a = Q b -> a = b.
```

J, P and Q

In the English Edition, Bourbaki assumes that there is way to create ordered pairs. This means that, given any two sets x and y, there is a third set z, denoted here '`J x y`', and traditionally (x, y). Sets of these form are called "ordered pairs". The "axiom of the ordered pair" states that if $(x, y) = (x', y')$, then $x = x'$ and $y = y'$. The unique quantity x (defined by the Axiom of Choice) such that $z = (x, y)$ is called the "first projection", and denoted here as '`P z`' and by Bourbaki as $\mathrm{pr}_1 z$. The unique quantity y such that $z = (x, y)$ is called the "second projection", and denoted here as '`Q z`' and by Bourbaki as $\mathrm{pr}_2 z$. Thus $z = (\mathrm{pr}_1 z, \mathrm{pr}_2 z)$ is equivalent to "z is an ordered pair".

Note that $\mathrm{pr}_1 \emptyset$ and $\mathrm{pr}_2 \emptyset$, are two well-defined sets, but whether $\emptyset$ is an ordered set or not is undecidable. Thus $\emptyset = (\mathrm{pr}_1 \emptyset, \mathrm{pr}_2 \emptyset)$ could be true or false.

In the French Version, Bourbaki shows that $(x, y) = \{\{x\}, \{x, y\}\}$ satisfies the axiom of the ordered pair, and uses this to define a pair. It follows that a pair is a set with one or two elements, so that the empty set is not a pair, and the relation given above is false. The definition

of pr_1 is the same as above, so that $\mathrm{pr}_1\emptyset$ is some well-defined set that satisfies no particular relation, since the Axiom of Choice does apply here.

In the definition that follows, we avoid using the Axiom of Choice. Then $\mathrm{pr}_1\emptyset = \mathrm{pr}_2\emptyset = \emptyset$. We use a hack that forbids COQ to unfold the definitions.The definition of a pair was introduced by Kuratowski in 1923, and used in [3]. Consider two sets x and y, and the pair $z = (x, y)$. If $a = \{x\}$ and $b = \{x, y\}$, then $z = \{a, b\}$. If $\mathrm{U} = \bigcup z$ and $\mathrm{I} = \bigcap z$, then $\mathrm{U} = \{x, y\}$ and $\mathrm{I} = \{x\}$. It follows $x = \bigcup \mathrm{I}$, and this gives a definition for pr_1. Given I and U, one can deduce y. In effect, either U = I, and y is the single element of U, and otherwise y is the single element of U not in I. Wikipedia defines pr_2 as the single element of $\{t \in \mathrm{U}, \mathrm{U} \neq \mathrm{I} \implies t \notin \mathrm{I}\}$ (this strange definition avoids the if-then-else construction). Note that y is the single element t of U such that $\{x, t\} = \mathrm{U}$, and this is the definition we shall use.

2.10 Module Cartesian

The *cartesian product* $\mathrm{A} \times \mathrm{B}$ of two sets A and B is the set of all pairs z such that $\mathrm{pr}_1 z \in \mathrm{A}$ and $\mathrm{pr}_2 z \in \mathrm{B}$. It is the union (for $x \in \mathrm{A}$) of the sets B_x of all (x, y) for $y \in \mathrm{B}$.

```
Definition product (A B : Set) :=
  union (fun_image A (fun x =>  (fun_image B (J x)))).
Definition coarse A := product A A.

Notation "A \times B" := (product A B) (at level 40).

Lemma setX_P x A B:
  inc x (A \times B) <-> [/\ pairp x, inc (P x) A & inc (Q x) B].
Lemma setX_pair x A B: inc x (A \times B) -> pairp x.
Lemma setX_i x A B:
  pairp x -> inc (P x) A -> inc (Q x) B -> inc x (A \times B).
Lemma setXp_i x y A B:
  inc x A -> inc y B -> inc (J x y) (A \times B).
Lemma setXp_P x y A B:
  inc (J x y) (A \times B) <-> (inc x A /\ inc y B).
```

A product is empty if and only one factor is empty. This is Proposition 2 [2, p. 75].

```
Lemma setX_0l B: emptyset \times B = emptyset.
Lemma setX_0r A: A \times emptyset = emptyset.
Lemma setX_0 A B:
   A \times B = emptyset -> (A = emptyset \/ B = emptyset).
```

The product $\mathrm{A} \times \mathrm{B}$ is increasing in A and B, strictly if the other argument is non empty. This is Proposition 1 [2, p. 74].

```
Lemma setX_Sl x x' y: sub x x' -> sub (x \times y) (x' \times y).
Lemma setX_Sr x y y': sub y y' -> sub (x \times y) (x \times y').
Lemma setX_Slr x x' y y':
  sub x x' -> sub y y' -> sub (x \times y) (x' \times y').
Lemma setX_lS x x' y: nonempty y ->
  sub (x \times y) (x' \times y) -> sub x x'.
Lemma setX_rS x y y': nonempty x ->
  sub (x \times y) (x \times y') -> sub y y'.
```

We sometimes write X_i instead of $(X, \{i\})$.

```
Definition indexed (x i: Set) := x \times singleton i.
Definition indexedr (i x: Set) := singleton i \times x.
Notation "a *s1 b" := (indexed a b) (at level 50).

Lemma indexed_pi x i y: inc y x -> inc (J y i) (x *s1 i).
Lemma indexed_P x i y:
  inc y (x *s1 i) <-> [/\ pairp y, inc (P y) x & Q y = i].
Lemma indexedrP a b c:
  inc a (indexedr b c) <-> [/\ pairp a , P a = b & inc (Q a) c].
```

2.11 Module Function

We introduce here some notations. Assume that P means $\forall x, p(x)$. Then `{inc X, P}` means $\forall x \in X, p(x)$. Assume that Q means $\forall x \forall y, q(x,y)$. Then `{inc X & Y, Q}` means $\forall x \in X, \forall y \in Y, q(x,y)$ and `{inc X &, Q}` means $\forall x \in X, \forall y \in X, q(x,y)$. (note that `prop_inc1` takes 3 arguments; a set X, a property p and third argument, that is not used, but whose type is some phantom built from P. Since p can be deduced from P, thus from the type of the third argument, the second argument is implicit; with this trick the definition can use p, although P is given).

```
Definition prop_inc1 (X : Set) (P: property)
   & (phantom Prop (forall x0 : Set, P x0)) :=
   forall x, inc x X -> P x.
Definition prop_inc11 X Y (P: relation)
  & (phantom Prop (forall x y: Set, P x y)) :=
   forall x y, inc x X -> inc y Y -> P x y.
Definition prop_inc2 X (P: Set -> Set -> Prop)
  & (phantom Prop (forall x y: Set, P x y)) :=
   forall x y, inc x X -> inc y X -> P x y.

Notation "{ 'inc' d , P }" :=
  (prop_inc1 d (inPhantom P))
  (at level 0, format "{ 'inc'  d ,  P }") : type_scope.

Notation "{ 'inc' d1 & d2 , P }" :=
  (prop_inc11 d1 d2 (inPhantom P))
  (at level 0, format "{ 'inc'  d1  &  d2 ,  P }") : type_scope.

Notation "{ 'inc' d & , P }" :=
  (prop_inc2 d (inPhantom P))
  (at level 0, format "{ 'inc'  d  & ,  P }") : type_scope.
```

Assume that P is as above; then `{when r, P}` means $\forall x, r(x) \implies p(x)$. We consider also variants where a property q depends on two variables. In particular `{when: r, Q}` means $\forall x, y, r(x,y) \implies q(x,y)$.

```
Definition prop_when1 (X : property) (P: property)
   & (phantom Prop (forall x0 : Set, P x0)) :=
   forall x, X x -> P x.
```

```
Definition prop_when11 (X Y:  property) (P: Set -> Set -> Prop)
  & (phantom Prop (forall x y: Set, P x y)) :=
   forall x y, X x -> Y y -> P x y.
Definition prop_when2 (X:  property) (P: relation)
  & (phantom Prop (forall x y: Set, P x y)) :=
   forall x y, X x -> X y -> P x y.
Definition prop_when22 (X: relation) (P: relation)
  & (phantom Prop (forall x y: Set, P x y)) :=
   forall x y, X x y -> P x y.
Notation "{ 'when' d , P }" :=
  (prop_when1 d (inPhantom P))
  (at level 0, format "{ 'when'  d ,  P }") : type_scope.


Notation "{ 'when' d1 & d2 , P }" :=
  (prop_when11 d1 d2 (inPhantom P))
  (at level 0, format "{ 'when'  d1  &  d2 ,  P }") : type_scope.

Notation "{ 'when' d & , P }" :=
  (prop_when2 d (inPhantom P))
  (at level 0, format "{ 'when'  d  & ,  P }") : type_scope.

Notation "{ 'when' : d , P }" :=
  (prop_when22 d (inPhantom P))
  (at level 0, format "{ 'when' : d , P }") : type_scope.
```

We say that f and g *commute* at x if $f(g(x)) = g(f(x))$. We say that f and g *commute* if they commute everywhere.

```
Definition commutes_at (f g: Set -> Set) x:= f (g x) = g (f x).
Definition commutes f g := forall x, commutes_at f g x.
```

More notations. We say that an operation f is compatible with p and q if $p(x)$ implies $q(f(x))$. If f takes two arguments, this means that $p(x,y)$ implies $q(f(x),f(y))$.

```
Definition compatible_1 f (p q: property) :=
    forall x, (p x) -> q (f x).
Definition compatible_2 f (p q:relation)  :=
    forall x y, (p x y) -> q (f x) (f y).
Definition compatible_3 f (p q:property)  :=
    forall x y, (p x) -> (p y) -> q (f x y).

Notation "{ 'compat' f : x / p >-> q }" :=
  (compatible_1 f  (fun x => p) (fun x => q))
  (at level 0,  f at level 99, x ident,
     format "{ 'compat'  f  : x  /  p  >->  q }") : type_scope.

Notation "{ 'compat' f : x / p }" :=
  (compatible_1 f  (fun x => p) (fun x => p))
  (at level 0,  f at level 99, x ident,
    format "{ 'compat'  f  : x  /  p }") : type_scope.

Notation "{ 'compat' f : x y / p >-> q }" :=
  (compatible_2 f  (fun x y => p) (fun x y => q))
  (at level 0,  f at level 99, x ident, y ident,
```

```
    format "{ 'compat'  f  : x  y  /  p  >->  q }") : type_scope.
Notation "{ 'compat' f : x y / p }" :=
  (compatible_2 f  (fun x y => p) (fun x y => p))
  (at level 0,  f at level 99, x ident, y ident,
    format "{ 'compat'  f  : x  y  /  p }") : type_scope.

Notation "{ 'compat' f : x & / p >-> q }" :=
  (compatible_3 f  (fun x => p) (fun x => q))
  (at level 0,  f at level 99, x ident,
     format "{ 'compat'  f  : x  &  /  p  >->  q }") : type_scope.

Notation "{ 'compat' f : x & / p }" :=
  (compatible_3 f  (fun x => p) (fun x => p))
  (at level 0,  f at level 99, x ident,
     format "{ 'compat'  f  : x  &  /  p }") : type_scope.
```

We give here examples

```
(*
Lemma setU2_2 a y:  {compat (union2 y): x / inc a x >-> inc a x}.
Lemma setU2_2 a y:  {compat (union2 y): x / inc a x}.
Lemma funI_setne f: {compat (fun_image ^~ f): x / nonempty x }.
Lemma setI2_S1 C: {compat (intersection2 C) : x y / sub x y }.
Lemma set_SC C:  {compat (fun z => z -s C) : x y / sub x y}.
Lemma set_CS C:  {compat (fun z => C -s z) : x y / sub x y >-> sub y x}.
Lemma setI2_12S y: {compat intersection2 : x & / sub y x}.
Lemma setU2_12S y: {compat union2 : x & / sub x y}.
*)
```

A simple *graph* is a set of pairs. If (x, y) is in the graph, we say that x and y are related. A functional graph is one for which the first projection is injective. We use `sgraph` and `fgraph` in the definitions that follow. The `domain` and `range` are the images of the first and second projection.

```
Definition alls (X: Set)(P: property) := forall a, inc a X -> P a.
Definition sgraph r := alls r pairp.
Definition fgraph f := sgraph f /\ {inc f &, injective P}.
Definition domain f := fun_image f P.
Definition range f := fun_image f Q.
Definition related r x y := inc (J x y) r.
```

Some properties of a functional graph.

```
Lemma fgraph_sg f: fgraph f -> sgraph f.
Lemma fgraph_pr f x y y': fgraph f -> inc (J x y) f -> inc (J x y') f -> y = y'.
```

The domain and range of a graph are characterized by the following two lemmas.

```
Lemma domainP r: sgraph r ->  forall x,
  (inc x (domain r) <-> (exists y, inc (J x y) r)).
Lemma rangeP r:  sgraph r ->  forall y,
  (inc y (range r) <->  (exists x, inc (J x y) r)).
```

These lemmas are obvious from the definitions.

```
Lemma domain_i1 f x: inc x f -> inc (P x) (domain f).
Lemma range_i2 f x: inc x f -> inc (Q x) (range f).
Lemma domain_i f x y: inc (J x y) f -> inc x (domain f).
Lemma range_i f x y: inc (J x y) f -> inc y (range f).
```

Some properties of a simple graph.

```
Lemma sgraph_exten r r':
  sgraph r -> sgraph r' ->
  (forall u v, related r u v <-> related r' u v) -> r = r'.
Lemma setI2_graph1 x y: sgraph x -> sgraph (x \cap y).
Lemma setI2_graph2 x y: sgraph y -> sgraph (x \cap y).
Lemma setU2_graph x y:  {compat union2 : x & / sgraph x}.
```

We consider the domain and range of various sets. In particular, domain and range are morphisms for union.

```
Lemma range_set0: range emptyset = emptyset.
Lemma domain_set0: domain emptyset = emptyset.
Lemma domain_set0P x: nonempty (domain x) <-> nonempty x.
Lemma domain_set0_P r: (domain r = emptyset <-> r = emptyset).
Lemma range_set0_P r:  (range r = emptyset <-> r = emptyset).
Lemma domain_set1 x y: domain (singleton (J x y)) = singleton x.
Lemma range_set1 x y: range (singleton (J x y)) = singleton y.
Lemma range_setU2: {morph range: x y /  x \cup y}.
Lemma domain_setU2: {morph domain: x y /  x \cup y}.
Lemma domain_setU z: domain (union z) = union (fun_image z domain).
Lemma range_setU z: range (union z) = union (fun_image z range).
Lemma domain_setU1 f x y:  domain (f +s1 (J x y)) = (domain f) +s1 x.
Lemma range_setU1 f x y:  range (f +s1 (J x y)) = (range f) +s1 y.
```

We consider here some special graphs. The union of functional graphs is a functional graph, when the domains are mutually disjoint; we consider here the union of two graphs, the general case will be considered later on.

```
Lemma sgraph_set0: sgraph emptyset.
Lemma fgraph_set0: fgraph emptyset.
Lemma fgraph_setU1 f x y:
  fgraph f -> ~inc x (domain f) ->
  fgraph (f +s1 (J x y)).
Lemma fgraph_setU2 a b: fgraph a -> fgraph b ->
  disjoint (domain a) (domain b) ->
  fgraph (a \cup b).
```

Assume that g is a functional graph, and x is in the domain. Then there is y such that $(x, y) \in g$ (since g is a graph), and this y is unique (since the graph is functional). Moreover, y is in the range of g. Thus, we can define a function '`Vg g x`' or $\mathcal{V}(x, g)$, or $\mathcal{V}_g(x)$, that maps x to y under these circumstances.

We say '`same_Vg f g`' or '`f =1g g`' when $\mathcal{V}_f(x) = \mathcal{V}_g(x)$ for every x, so that '`{inc a, f =1g g}`' means $\mathcal{V}_f(x) = \mathcal{V}_g(x)$ for every $x \in a$. We say '`allf g p`' when $p(\mathcal{V}_g(x))$ holds whenever x is in the domain of g.

```
Definition action_prop (f g: Set -> Set -> Set) :=
```

```
   forall a b x, f (g a b) x = (f a (f b x)).

Definition Vg (f x: Set) := select (fun y : Set => inc (J x y) f) (range f).
Definition allf (g: Set) (p: property) :=
   forall x, inc x (domain g) -> (p (Vg g x)).

Definition same_Vg f g: Vg f =1 Vg g.
Notation "f1 =1g f2" := (same_Vg f1 f2)
  (at level 70, no associativity) : fun_scope.
```

Assume that f is a functional graph. If x is in the domain, then $(x, \mathcal{V}_f(x)) \in f$. If $z \in f$, then z is a pair, say $z = (x, y)$, and $y = \mathcal{V}_f(x)$. It follows that y is the range of f if and only if there is x in the domain of f such that $y = \mathcal{V}_f(x)$. Finally, if f and g are two functional graphs with the same domain, such that $\mathcal{V}_f(x) = \mathcal{V}_g(x)$ on the domain, it follows $f = g$.

```
Section Vprops.
Variable f: Set.
Hypothesis fgf: fgraph f.

Lemma fdomain_pr1 x: inc x (domain f) -> inc (J x (Vg f x)) f.
Lemma in_graph_V x: inc x f -> x = J (P x) (Vg f (P x)).
Lemma pr2_V x: inc x f -> Q x = Vg f (P x).
Lemma range_gP y:
  (inc y (range f) <-> (exists2 x, inc x (domain f) & y = Vg f x)).
Lemma inc_V_range x: inc x (domain f) -> inc (Vg f x) (range f).

End Vprops.

Lemma fgraph_exten f g:
  fgraph f -> fgraph g -> domain f = domain g ->
   {inc (domain f), f =1g g} -> f = g.
```

Consider now a function f and a set x. The set of all pairs $(a, f(a))$ for $a \in x$ will be denoted by $\mathscr{L}_x f$. This is a functional graph; its domain is x, and its evaluation function is f. If f and g are equal on a, then $\mathscr{L}_a f = \mathscr{L}_a g$.

```
Definition Lg (x : Set) (p : fterm) :=
  fun_image x (fun y => J y (p y)).

Lemma Lg_i x y p : inc x y ->  inc (J x (p x)) (Lg y p).
Lemma Lg_fgraph p x: fgraph (Lg x p).
Lemma Lg_domain l x p: domain (Lg x p) = x.
Lemma LVg_E  x p y: inc y x -> Vg (Lg x p) y = p y.
Lemma Lg_exten a f g: {inc a, f =1 g} -> Lg a f = Lg a g.
```

The range of $\mathscr{L}_x f$ is the image $f\langle x\rangle$ (according to Section 2.4; on page 45 we shall define $g\langle x\rangle$ where g is a graph). There are some other useful properties.

If v is a graph with domain x and evaluation function f, then $v = \mathscr{L}_x f$. We have $\mathscr{L}_x f = \mathscr{L}_y g$ if $x = y$, and f and g agree on x.

```
Lemma Lg_range p x: range (Lg x p) = fun_image x p.
Lemma Lg_create a f: Lg a (fun x => Vg (Lg a f) x) = Lg a f.
Lemma Lg_range_P sf f a:
  inc a (range (Lg sf f)) <-> exists2 b, inc b sf & a = f b.
```

```
Lemma Lg_recovers f:
  fgraph f -> Lg (domain f) (Vg f) = f.
Lemma Lg_exten a f g: {inc a, f =1 g} -> Lg a f = Lg a g.
```

An interesting function is the *identity* function: it maps everything on itself. We consider here the graph of this function. More properties will be given later.

```
Definition identity_g (x : Set) := Lg x id.

Lemma identity_fgraph x: fgraph (identity_g x).
Lemma identity_sgraph x: sgraph (identity_g x).
Lemma identity_d x: domain (identity_g x) = x.
Lemma identity_r x: range (identity_g x) = x.
Lemma identity_ev x a: inc x a -> Vg (identity_g a) x = x.
```

Another interesting function is the constant function that maps any element of X onto y. Its graph is $X \times \{y\}$. This set will be denoted X_y (for instance when we consider disjoint unions, and cardinal sums).

```
Definition cst_graph x y:= Lg x (fun _ => y).

Lemma cst_graph_pr x y:  cst_graph x y = x *s1 y
Lemma cst_graph_ev x y t : inc t x -> Vg (cst_graph x y) t = y.
Lemma cst_graph_d x y  : domain (cst_graph x y) = x.
Lemma cst_graph_fgraph a b: fgraph (cst_graph a b).
```

We give here the evaluation function for the graph f, extended by $x \mapsto y$.

```
Lemma setU1_V_out f x y:
  fgraph f -> ~ (inc x (domain f)) ->  Vg x (f +s1 (J x y)) = y.
Lemma setU1_V_in f x y u:
  fgraph f -> ~ (inc x (domain f)) -> inc u (domain f) ->
  Vg (f +s1 (J x y)) u = Vg f u.
```

Assume that g is a functional graph, and $f \subset g$. Then f is a functional graph, its domain and range are subsets of the domain and range of g; its evaluation function is the same. There is a converse: if we have two functional graphs, if the domain of f is a part of the domain of g, and if the evaluation function is the same on the domain of f, then f is a subset of g. From this we deduce an extensionality property.

```
Lemma sub_graph_fgraph f g: fgraph g -> sub f g -> fgraph f.
Lemma domain_S f g: sub f g -> sub (domain f) (domain g).
Lemma range_S f g: sub f g -> sub (range f) (range g).
Lemma sub_graph_ev f g:
  fgraph g -> sub f g -> {inc (domain f), f =1g g}.
```

The *restriction* of a graph f to a set x is the graph of $t \to \mathcal{V}_f(t)$ for $t \in x$.

```
Definition restr f x := Lg x (Vg f).
Lemma restr_d f x: domain (restr f x) = x.
Lemma restr_fgraph f x: fgraph (restr f x).
Lemma restr_ev f x i: inc i x -> Vg (restr f x) i = Vg f i.
Lemma double_restr f a b:  sub a b -> (restr (restr f b) a) = (restr f a).
```

```
Lemma restr_Lg a b f: sub b a -> restr (Lg a f) b  = Lg b f.
Lemma restr_to_domain f g: fgraph f -> sub g f -> restr f (domain g) = g.
Lemma restr_range1 f x: fgraph f -> sub x (domain f) ->
  sub (range (restr f x)) (range f).
```

¶ We denote by $g \circ f$ the *composition* of the two functions. It maps x to $g(f(x))$. We shall give three definitions, one adapted for functional graphs, one for simple graphs, and one for functions. We consider here functional graphs. Assume that the range of f is a subset of the domain of g. In this case, for any x in the domain of f, $f(x)$ is in the domain of g, so that $g(f(x))$ is in the domain of g.

```
Definition composablef (f g : Set) :=
  [/\ fgraph f, fgraph g & sub (range g) (domain f)].
Definition composef f g :=  Lg (domain g)  (fun y => Vg f (Vg g y)).


Notation "x \cf y" := (composef x y) (at level 50).
Notation "x \cfP y" := (composablef x y) (at level 50).

Lemma composef_ev x f g:
  inc x (domain g) -> Vg (f \cf g) x = Vg f (Vg g x).
Lemma composef_fgraph f g: fgraph (f \cf g).
Lemma composef_domain f g: domain (f \cf g) = domain g.
Lemma composef_range f g: f \cfP g ->
  sub (range (f \cf g)) (range f).
```

Chapter 3

Correspondences

From now on, we follow Bourbaki as closely as possible. The series "Elements of mathematics" is divided in 9 books, the first one is called "Theory of sets". This book is divided into four chapters, the second one is "Theory of sets". This chapter is divided into 6 sections; we implement here section 3 "Correspondences". When we talk about Proposition 1, this is to be understood as Proposition 1 of [2] of the current section (i.e., the current Chapter of this report).

3.1 Graphs and correspondences

The next theorem is Proposition 1 in [2, p. 76]; it claims existence and uniqueness of two sets denoted by $\mathrm{pr}_1\langle r\rangle$ and $\mathrm{pr}_2\langle r\rangle$. The notation $\mathrm{pr}_1\langle r\rangle$ is defined in section 2.4; it is the domain of r.

```
Theorem range_domain_exists r: sgraph r ->
  ((exists! a, forall x, inc x a <-> (exists y, inc (J x y) r)) /\
    (exists! b, forall y, inc y b <-> (exists x, inc (J x y) r))).
```

A product $x \times y$ is a graph. The domain is x, the range is y, whenever the sets are nonempty. It is a functional graph only if the domain is a singleton.

```
Lemma setX_graph x y: sgraph (x \times y).
Lemma sub_setX_graph u x y: sub u (x \times y) -> sgraph u.
Lemma sub_graph_setX r: sgraph r -> sub r ((domain r) \times(range r)).
Lemma setX_relP x y a b:
  related (x \times y) a b <-> (inc a x /\ inc b y).
Lemma setX_domain x y:  nonempty y -> domain (x \times y) = x.
Lemma setX_range x y:   nonempty x -> range (x \times y) = y.
```

The *diagonal* of x, denoted Δ_x, is the set of all pairs (a,a), with $a \in x$. This is the graph of the identity function on x.

```
Definition diagonal x := Zo (coarse x)(fun y=> P y = Q y).

Lemma diagonal_i_P x u:
  inc u (diagonal x) <-> [/\ pairp u, inc (P u) x & P u = Q u].
Lemma diagonal_pi_P x u v:
```

```
  inc (J u v) (diagonal x)  <-> (inc u x /\ u = v).
Lemma diagonal_is_identity x: diagonal x = identity_g x.
```

For Bourbaki, "a *correspondence* between a set A and a set B is a triple[1] $\Gamma = (G, A, B)$ where G is a graph such that $\mathrm{pr}_1\langle G\rangle \subset A$ and $\mathrm{pr}_2\langle G\rangle \subset B$". The quantities G, A, and B are respectively called the graph, source and target of Γ. We get an equivalent definition by using $G \subset A \times B$, and this is the same as $G \in \mathfrak{P}(A \times B)$.

```
Definition triplep f := pairp f /\ pairp (Q f).
Definition triple s t g  := J g (J s t).
Definition source x := P (Q x).
Definition target x := Q (Q x).
Definition graph x := P x.

Definition correspondence f  :=
  triplep f i /\ sub (graph f) ((source f) \times (target f)).
```

Denote by s, t and g, the source, target and graph of a correspondence. If $\Gamma = (a, b, c)$ then $s(\Gamma) = a$, $t(\Gamma) = b$ and $g(\Gamma) = c$. If Γ is a triple, then $\Gamma = (s(\Gamma), t(\Gamma), g(\Gamma))$.

```
Lemma triple_corr s t g: triplep (triple s t g).
Lemma corresp_s s t g: source (triple s t g) = s.
Lemma corresp_t s t g: target (triple s t g) = t.
Lemma corresp_g s t g: graph (triple s t g) = g.
Lemma corresp_recov f: triplep f ->
  triple (source f) (target f) (graph f) = f.
Lemma corresp_recov1 f: correspondence f ->
  triple (source f) (target f) (graph f) = f.
```

If Γ is a correspondence defined by s, t, and g, then $g \subset s \times t$. This is equivalent to: g is a graph whose domain is a subset of s and whose range is a subset of t.

```
Lemma corr_propcc s t g:
  sub g (s \times t) <-> [/\ sgraph g, sub (domain g) s & sub (range g) t].
Lemma corr_propc f (g := graph f):
  correspondence f ->
  [/\ sgraph g, sub (domain g) (source f) & sub (range g) (target f)].

Lemma corresp_create s t g:
  sub g (s \times t) -> correspondence (triple s t g).
Lemma corresp_is_graph g:
  correspondence g -> sgraph (graph g).
Lemma corresp_sub_range g:
  correspondence g -> sub (range (graph g)) (target g).
Lemma corresp_sub_domain g:
  correspondence g -> sub (domain (graph g)) (source g).
```

A triple (G, A, B) is a correspondence if and only if $G \in \mathfrak{P}(A \times B)$, but Bourbaki defines the powerset only later. From this, we deduce that the set of all correspondences between A and B is $\mathfrak{P}(A \times B) \times \{A\} \times \{B\}$.

```
Definition correspondences x y :=
```

[1] Bourbaki interprets (G, A, B) as ((G, A), B). We prefer (G, (A, B)).

```
  (powerset (x \times y)) \times( (singleton x) \times (singleton y)).
Lemma correspondencesP x y z:
  inc z (correspondences x y) <->
  [/\ correspondence z, source z = x & target z = y].
```

¶ Direct image of a set X by a functional object f. This will be denoted by $f\langle X\rangle$. In the first definition f is a graph, and we consider all elements y for which there is a $z \in X$ such that $(z,y) \in f$. In the other definitions, f is a correspondence, and we consider the image by its graph of either X or the source of f.

```
Definition direct_image f X:=
  Zo (range f) (fun y=>exists2 x, inc x X & inc (J x y) f).
Definition image_by_fun f := direct_image (graph f).
Definition image_of_fun f := image_by_fun f (source f).
```

We give now some basic properties. The image is a part of the range; it is the full range if we consider the full domain. The image of a subset X of the domain is empty if and only if X is empty. Proposition 2 in [2, p. 77] says that the $X \mapsto f\langle X\rangle$ is increasing

```
Lemma dirim_P f X y:
  inc y (direct_image f X) <-> exists2 x, inc x X & inc (J x y) f.
Lemma dirimE f X: fgraph f -> sub X (domain f) ->
  direct_image f X = fun_image X (Vg f).
Lemma dirim_Sr f X: sub (direct_image f X) (range f).
Lemma dirim_domain f: sgraph f -> direct_image f (domain f) = range f.
Lemma dirim_set0 f: direct_image f emptyset =  emptyset.
Lemma dirim_setn0 f u: sgraph f -> nonempty u -> sub u (domain f)
  -> nonempty (direct_image f u).
Theorem dir_im_S f: {compat (direct_image f): u v / sub u v}.
```

A special case is when X is a singleton $\{x\}$. If f is a correspondence, the notation $G(f)\langle\{x\}\rangle$ is sometimes simplified to $f\langle\{x\}\rangle$ or $f(x)$ (this last notation is ambiguous, since it denotes also the value of f at x).

```
Definition im_of_singleton f x := direct_image f (singleton x).

Lemma dirim_set1_P f x y:
  inc y (im_of_singleton f x) <-> inc (J x y) f.
Lemma dirim_set1_S f f': sgraph f -> sgraph f' ->
  ((forall x, sub (im_of_singleton f x) (im_of_singleton f' x)) <-> sub f f').
```

3.2 Inverse of a correspondence

The inverse graph of G, denoted by $\overset{-1}{G}$, or G^{-1} is the set of all pairs (x,y) such that $(y,x) \in G$. This is also the set of all $(\mathrm{pr}_2 z, \mathrm{pr}_1 z)$ for $z \in G$ (these two sets may be different if G is not a graph).

```
Definition inverse_graph r :=
  Zo ((range r) \times (domain r)) (fun y=> inc (J (Q y)(P y)) r).

Lemma igraph_alt r: sgraph r ->
  inverse_graph r = fun_image r (fun z => J(Q z) (P z)).
```

Some trivialities to start with.

```
Lemma igraph_graph r: sgraph (inverse_graph r).
Lemma igraphP r y:
  inc y (inverse_graph r) <-> (pairp y /\  inc (J (Q y)(P y)) r).
Lemma igraph_pP r x y:
  inc (J x y) (inverse_graph r) <-> inc (J y x) r.
```

Taking the inverse swaps range and domain. Taking twice the inverse gives the same graph. The inverse of a product is the product in reverse order. The inverse of the empty set or identity is itself.

```
Lemma igraph_involutive : {when sgraph, involutive inverse_graph}.
Lemma igraph_range r: sgraph r ->  range (inverse_graph r) = domain r.
Lemma igraph_domain  r: sgraph r ->  domain (inverse_graph r) = range r.
Lemma igraph0: inverse_graph (emptyset) = emptyset.
Lemma igraphX x y: inverse_graph (x \times y) = y \times x.
Lemma igraph_identity_g x: inverse_graph (identity_g x) = identity_g x.
```

The inverse of the correspondence $\Gamma = (G, A, B)$ is $(\overset{-1}{G}, B, A)$. It is denoted by $\overset{-1}{\Gamma}$. It satisfies some trivial properties.

```
Definition inverse_fun m :=
  corresp(target m) (source m)(inverse_graph (graph m)).

Lemma ifun_s f: source (inverse_fun f) = target f.
Lemma ifun_t f: target (inverse_fun f) = source f.
Lemma ifun_g f: graph (inverse_fun f) = inverse_graph (graph f).
Lemma icor_correspondence m:
  correspondence m -> correspondence (inverse_fun m).
Lemma icor_involutive: {when correspondence, involutive inverse_fun}.
```

The inverse image by a graph (or correspondence or a function) is the direct image of its inverse. It is denoted by $g^{-1}\langle x\rangle$.

```
Definition inverse_image r := direct_image (inverse_graph r).
Definition inv_image_by_fun r:=  inverse_image (graph r).

Lemma iim_fun_pr r :
  inv_image_by_fun r  = image_by_fun (inverse_fun r) x.
Lemma iim_graph_P x r y:
  (inc y (inverse_image r x)) <-> (exists2 u, inc u x & inc (J y u) r)).
Lemma iim_fun_P x r y:
  (inc y (inv_image_by_fun r x))
    <-> (exists2 u, inc u x & inc (J y u) (graph r)).
```

3.3 Composition of two correspondences

The *composition* of two graphs $G_2 \circ G_1$ is the set of all (x, z) for which there is a y such that (x, y) is in G_1 and and (y, z) is in G_2. It is a subset of the product of the domain of the first graph and the range of the second. If both arguments are functional graphs and are composable, then the result is a functional graph such that $G_2 \circ G_1(x) = G_2(G_1(x))$, for any

x in the domain of G_1; in other terms, this notion coincides with the previous definition of composition.

```
Definition composeg r' r :=
 Zo((domain r)\times (range r'))
   (fun w => exists2 y, inc (J (P w) y) r & inc (J y (Q w)) r').
Notation "x \cg y" := (composeg x y) (at level 50).

Lemma compg_graph r r':  sgraph (r \cg r').
Lemma compg_P r r' x:
  inc x (r' \cg r) <->
  (pairp x /\ ( exists2 y, inc (J (P x) y) r & inc (J y (Q x)) r')).
Lemma compg_pP r r' x y:
  inc (J x y) (r' \cg r) <-> (exists2 z, inc (J x z) r & inc (J z y) r').
Lemma compg_domain_S r r':  sub (domain (r' \cg r)) (domain r).
Lemma compg_range_S  r r':  sub (range (r' \cg r)) (range r').
Lemma compg_composef f g: f \cfP g -> f \cf g =  f \cg g.
```

Proposition 3 in [2, p. 79] says $(G' \circ G)^{-1} = G^{-1} \circ (G')^{-1}$.

```
Theorem compg_inverse: {morph inverse_graph : a b / a \cg b >-> b \cg a}.
```

Proposition 4 [2, p. 79] says that graph composition is associative.

```
Theorem compgA: associative composeg.
```

Proposition 5 [2, p. 79] says $(G' \circ G)\langle A\rangle = G'\langle G\langle A\rangle\rangle$. We have a characterization of the domain and range of the composition as direct or inverse image of the domain or range. We have an interesting formula $A \subset G^{-1}\langle G\langle A\rangle\rangle$.

```
Theorem compg_image: action_prop direct_image composeg.

Lemma compg_domain r r':
  sgraph r' -> domain (r' \cg r) = inverse_image r (domain r').
Lemma compg_range r r':
  sgraph r -> range (r' \cg r) = direct_image r' (range r).
Lemma inverse_direct_imageg r x:
  sgraph r -> sub x (domain r) ->
  sub x (inverse_image r (direct_image r x)).
Lemma compg_S r r' s s':
  sub r s -> sub r' s' -> sub (r' \cg r) (s' \cg s).
```

We say that f and f' are composable if they are correspondences where the target of f is the source of f'. We may assume $f = (G, A, B)$ and $f' = (G', B, C)$. Let $G' \circ G$ be the composition of the two graphs (both previous definitions agree in this case). We define the *composition* $f' \circ f = (G' \circ G, A, C)$; this is a correspondence, with source A, target C, and graph $G' \circ G$. Proposition 5 implies $(f' \circ f)\langle A\rangle = f'\langle f\langle A\rangle\rangle$, and Proposition 3 gives $(f' \circ f)^{-1} = f^{-1} \circ f'^{-1}$, provided both correspondences are composable.

```
Definition composableC r' r :=
  [/\ correspondence r, correspondence r' & source r' = target r].
Definition compose r' r :=
  triple (source r)(target r') ((graph r') \cg (graph r)).
```

```
Notation "f1 \co f2" := (compose f1 f2) (at level 50).

Lemma compf_correspondence r' r:
  correspondence r -> correspondence r' ->
  correspondence (r' \co r).
Lemma compf_image: action_prop image_by_fun compose.
Lemma compf_inverse:  {morph inverse_fun : a b / a \co b >-> b \co a}.
```

The identity I_A of a set A is the correspondence $(\Delta_\mathrm{A}, \mathrm{A}, \mathrm{A})$, where Δ_A the diagonal of A.

```
Definition identity x := triple x x (identity_g x).

Lemma identity_triple x: correspondence (identity_fun x).
```

If f is a correspondence between A and B then $f \circ \mathrm{I}_\mathrm{A}$ and $\mathrm{I}_\mathrm{B} \circ f$ are equal to f. In particular $\mathrm{I}_\mathrm{A} \circ \mathrm{I}_\mathrm{A} = \mathrm{I}_\mathrm{A}$. We can restate this statement as: the identity of B is a left inverse for composition, whenever f is a correspondence with target B.

```
Lemma identity_s x: source (identity x) = x.
Lemma identity_t x: target (identity x) = x.
Lemma identity_graph0 x: graph (identity x) = identity_g x.
Lemma compf_id_left m:
   correspondence m -> (identity (target m)) \co m = m.
Lemma compf_id_right m:
  correspondence m -> m \co (identity (source m)) = m.
Lemma compf_id_id x:
  (identity x) \co (identity x) = (identity x).
Lemma identity_self_inverse x:
  inverse_fun (identity x) =  (identity x).

Corollary compose_identity_left E:
  {when (fun x => correspondence x /\ (target x) = E),
  left_id (identity E) compose}.
```

3.4 Functions

We say that a graph r is *functional* if each x is related to at most one y. We show that this definition is equivalent to the one given in Section 2.11, that says that if z and z' are in r, then $\mathrm{pr}_1 z = \mathrm{pr}_1 z'$ implies $z = z'$.

```
Definition functional_graph r :=
  forall x, singl_val (related r x).

Lemma functionalP r:
  (sgraph r /\ functional_graph r) <-> (fgraph r).
```

A *function* is a correspondence $f = (\mathrm{G}, \mathrm{A}, \mathrm{B})$ with a functional graph G, where A is the domain of G. This means that every x in A is related to unique y. This is denoted in Bourbaki by $f(x)$ or $\mathrm{G}(x)$. Here we use either $\mathcal{V}_\mathrm{G} x$ or $\mathcal{W}_f x$. Note: Bourbaki says [2, p. 82] "we shall often use the word 'function' in place of 'functional graph' ".

```
Definition function f :=
  [/\ correspondence f, fgraph (graph f) & source f = domain (graph f)].

Lemma function_pr s t g:
  fgraph g -> sub (range g) t -> s = domain g ->
  function (triple s t g).
Lemma function_fgraph f: function f -> fgraph (graph f).
Lemma function_sgraph f:  function f -> sgraph (graph f).
Lemma f_domain_graph f: function f -> domain (graph f) = source f.
Lemma f_range_graph f: function f -> sub (range (graph f))(target f).
Lemma image_by_fun_source f: function f ->
  image_of_fun f =  range (graph f).
Lemma function_functional f: correspondence f ->
  function f <-> (forall x, inc x (source f) ->
    exists! y,  related (graph f) x y).
```

Each property of $\mathcal{V}$ gives a corresponding one for $\mathcal{W}$. All lemmas listed here are trivial. Let $f = (\mathrm{G},\mathrm{A},\mathrm{B})$ be a function. If $x \in \mathrm{A}$ then $(x,\mathcal{W}_f x) \in \mathrm{G}$, $\mathcal{W}_f x \in \mathrm{range(G)}$ and $\mathcal{W}_f x \in \mathrm{B}$. If $y \in \mathrm{range(G)}$, there exists x such that $y = \mathcal{W}_f x$. If $z \in \mathrm{G}$ then $z = (\mathrm{pr}_1 z, \mathcal{W}_f \mathrm{pr}_1 z)$, $\mathrm{pr}_2 z = \mathcal{W}_f \mathrm{pr}_1 z$, and $\mathrm{pr}_1 z \in \mathrm{A}$. If $(x,y) \in \mathrm{G}$ then $y = \mathcal{W}_f x$, $x \in \mathrm{A}$ and $y \in \mathrm{B}$. Finally, if $\mathrm{X} \subset \mathrm{A}$ then $y \in f\langle \mathrm{X} \rangle$ if and only if there is $x \in \mathrm{X}$ such that $y = \mathcal{W}_f x$.

```
Definition Vf f x := Vg (graph f) x.

Section W_pr.
Variable  f: Set.
Hypothesis ff: function f.

Lemma Vf_pr3 x: inc x (source f) -> inc (J x (Vf f x)) (graph f).
Lemma in_graph_Vf x: inc x (graph f) -> x = (J (P x) (Vf f (P x))).
Lemma Vf_pr2 x:  inc x (graph f) -> Q x = Vf f (P x).
Lemma Vf_pr x y: inc (J x y) (graph f) -> y = Vf f x.
Lemma range_fP y:
   inc y (range (graph f)) <-> exists2 x, inc x (source f) & y = Vf f x.
Lemma Vf_range_g f x:inc x (source f) -> inc (Vf x tf) (range (graph f)).
Lemma Vf_target x: inc x (source f) -> inc (Vf f x) (target f).
Lemma p1graph_source x y: inc (J x y) (graph f) ->  inc x (source f).
Lemma p2graph_target x y: inc (J x y) (graph f) ->  inc y (target f).
Lemma p1graph_source1 x: inc x (graph f) ->  inc (P x) (source f).
Lemma p2graph_target1 x: inc x (graph f) ->  inc (Q x) (target f).

Lemma Vf_image_P x: sub x (source f) -> forall y,
  (inc y (image_by_fun f x) <-> exists2 u, inc u x & y = Vf f u).
Lemma Vf_imageP1: forall y,
  inc y (image_by_fun f (source f))
    <-> (exists u, inc u (source f) & y = Vf f u).
Lemma fun_image_Starget: sub (image_of_fun f) (target f).
Lemma fun_image_Starget1 x: sub (image_by_fun f x) (target f).
End W_pr.
```

Two functions having same source, same target and same evaluation function are the same. Two functions having same graph and target are the same.

```
Definition same_Vf  f g:= Vf f =1 Vf g.
Definition cstfp f (E: Set) := singl_val_fp (inc ^~E) (Vf f).
```

```
Definition cstgp (f E: Set) := singl_val_fp (inc ^~E) (Vg f).

Notation "f1 =1f f2" := (same_Vf f1 f2)
  (at level 70, no associativity) : fun_scope.

Lemma function_exten3 f g:
  function f -> function g ->
  graph f = graph g -> target f = target g -> source f = source g ->
  f = g.
Lemma function_exten1 f g:
  function f -> function g ->
  graph f = graph g -> target f = target g ->
  f = g.
Lemma function_exten f g:
  function f -> function g ->
  source f = source g ->  target f = target g -> {inc (source f), f =1f g}
  -> f = g.
```

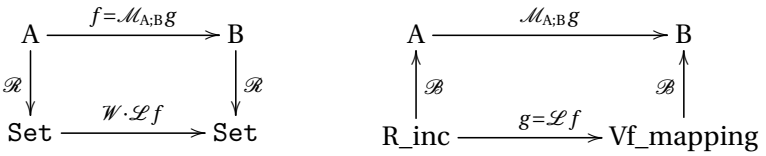
The first lemma says $f\langle\{x\}\rangle = \{f(x)\}$. Remember that the LHS is the set of all y related to x by the function; we claim that there is exactly one such element, and is chosen by the $\mathscr{W}$ function. We have $f^{-1}\langle \mathrm{B} \setminus \mathrm{X}\rangle = \mathrm{A} \setminus f^{-1}\langle \mathrm{X}\rangle$ if it is a function from A to B.

```
Lemma fun_image_set1 f x:
  function f -> inc x (source f) ->
  image_by_fun f (singleton x) = singleton (Vf f x).

Lemma iim_fun_C g x:
  function g ->
  inv_image_by_fun g ((target g) -s x) = (source g) -s (inv_image_by_fun g x).

Lemma iim_fun_set1_hi f x y: function f ->
   inc x (inv_image_by_fun f (singleton y)) -> y = Vf f x.
Lemma iim_fun_set1_i f x y: function f -> inc x (source f) ->
  Vf f x = y -> inc x (inv_image_by_fun f (singleton y)).
Lemma iim_fun_set1_P f y: function f -> forall x,
   inc x (inv_image_by_fun f (singleton y)) <->
   (inc x (source f) /\ y = x Vf f).
Lemma iim_fun_set1_E f y: function f ->
   (inv_image_by_fun f (singleton y)) = Zo (source f) (fun x => y = Vf f x).
```

¶ Let h be a mapping (for instance $x \mapsto x+1$) and A a set (for instance the set of odd integers). We can associate a graph, namely $\mathscr{L}_{\mathrm{A}}h$. If B is another set (for instance the set of the even integers), such that $x \in \mathrm{A}$ implies $h(x) \in \mathrm{B}$ we can consider the function $\mathscr{L}_{\mathrm{A};\mathrm{B}}h$ from A to B whose graph is $\mathscr{L}_{\mathrm{A}}h$ (see Section 3.6). Assume now that f maps type A into type B, its composition h with $\mathscr{R}$ is a mapping that satisfies: $x \in \mathrm{A}$ implies $h(x) \in \mathrm{B}$. The quantity $\mathscr{L}_{\mathrm{A};\mathrm{B}}h$ will be denoted by $\mathscr{L}f$. The graph of $\mathscr{L}f$ is the set of all $(\mathscr{R}i, \mathscr{R}f(i))$, for all $i : \mathrm{A}$. We shall see in a moment that f can be obtained from $g = \mathscr{L}f$ by the formula $f = \mathscr{M}_{\mathrm{A};\mathrm{B}}g$. Lemma `acreate_V` says that the following diagram (left part) commutes.



(a/b create)

Given a function $f : a \to b$, where a and b are two sets, we consider the set G of pairs $(x, f(x))$. This is a functional graph, a subset of $a \times b$. The correspondence (G, a, b) is denoted by $\mathscr{L} f$.

```
Definition gacreate (a b:Set) (f:a->b) := IM (fun y:a => J (Ro y) (Ro (f y))).
Definition acreate (a b:Set) (f:a->b)  := triple a b (gacreate f).

Lemma acreate_triple (a b:Set) (f:a->b): correspondence (acreate f).

Lemma acreateP (A B:Set) (f:A->B) x:
  inc x (graph (acreate f)) <-> exists u:A, J(Ro u)(Ro (f u)) = x.
Lemma acreate_function (A B:Set) (f:A->B): function(acreate f).
Lemma acreate_V (A B:Set) (f:A->B) (x:A):
  Vf (acreate f) (Ro x) = Ro (f x).
```

Given a function g, with source A and target B, we can use the inverse function $\mathscr{B}$ of $\mathscr{R}$ to get a map f from type A to type B. We shall denote it by $\mathscr{M} g$ or $\mathscr{M}_{\mathrm{A;B}} g$. We have $\mathscr{L} f = g$. The notation $\mathscr{M} g$ is a shorthand for $\mathscr{M}_{\mathrm{source}(g);\mathrm{target}(g)} g$. If A = source($g$) and B = target($g$) but if equality is not identity then $\mathscr{M} g$ and $\mathscr{M}_{\mathrm{A;B}} g$ are objects of different type, and are not equal in Coq. In particular, if h is a mapping of type A → B, and if $g = \mathscr{L} h$, then $\mathscr{M} g$ is a function A′ → B′, where A′ is source(g) and not A, so that $\mathscr{M}\mathscr{L} h$ is not equal to h.

We create here $\mathscr{M} f$. The expression '`R_inc x`' is a proof of $x \in \mathrm{source}(f)$. The expression `(inc_Vf_target _ _)` shows $w \in$ B, where B is the target of f and w the value of f. Evaluating $\mathscr{B}$ yields an object of type B, whose evaluation by $\mathscr{R}$ is w. This is summarized by the first lemma. The second one says $\mathscr{L}\mathscr{M} f = f$. Remember that in order to use $\mathscr{M} f$ one needs a proof H that f is a function, and f is implicit, since it can be deduced from H.

```
Definition bcreate1 f (H:function f) :=
  fun x:source f => Bo (inc_Vf_target H (R_inc x)).

Lemma prop_bcreate1 f (H:function f) (x:source f):
  Ro(bcreate1 H x) = Vf f (Ro x).
Lemma bcreate_inv1 f (H:function f):
  acreate (bcreate1 H) = f.
```

We create here $\mathscr{M}_{a;b} g$. It depends on three assumptions, g is a function, a is the source and b is the target. See diagram (a/b create) above, right part. If $x : a$, and $y = \mathscr{R} x$, the assertion '`R_inc x`' says $y \in a$, and applying $\mathscr{B}$ to the assertion gives y. Let $w = \mathscr{W}_g x$. The `Vf_mapping` lemma says (because of our three assumptions) that $w \in b$. If we apply $\mathscr{B}$, we get some element of type b, which is $\mathscr{M}_{a;b} g(x)$.

We have $\mathscr{L}\mathscr{M}_{\mathrm{A;B}} g = g$ and $\mathscr{M}_{\mathrm{A;B}}\mathscr{L} f = f$. In fact, if f' is $\mathscr{M}_{\mathrm{A;B}}\mathscr{L} f$ we have $f'(a) = f(a)$ for all a of type A. This follows from injectivity of $\mathscr{R}$.

```
Lemma Vf_mapping f A B (Ha:source f = A)(Hb:target f = B) x:
  function f -> inc x A -> inc (Vf f x) B.
Lemma acreate_source (A B:Set) (f:A->B): source (acreate f)= A.
Lemma acreate_target (A B:Set) (f:A->B): target (acreate f)= B.

Definition bcreate f A B
  (H:function f)(Ha:source f = A)(Hb:target f = B):=
  fun x:A => Bo (Vf_mapping Ha Hb H (R_inc x)).
```

```
Lemma prop_bcreate2 f A B
  (H:function f) (Ha:source f = A)(Hb:target f= B)(x:A):
  Ro(bcreate H Ha Hb x) = Vf f (Ro x).

Lemma bcreate_inv2 f A B
  (H:function f) (Ha:source f = A)(Hb:target f = B):
  acreate (bcreate H Ha Hb) = f.

Lemma bcreate_inv3 (A B:Set) (f:A->B):
  bcreate (acreate_function f) (acreate_source f)(acreate_target f) =1 f.

Lemma bcreate_eq f (H:function f):
  bcreate1 H =1 bcreate H (refl_equal (source f)) (refl_equal (target f)).
```

Let's consider some examples of functions. A function whose graph or target is empty has empty source. Thus, if the target is empty, it is the identity of the empty set. For any set x, there is a unique function with empty graph and target x.

```
Definition function_prop f s t:=
  [/\ function f, source f = s & target f = t].

Definition empty_functionCt x := fun t:emptyset => match t return x with end.
Definition empty_functionC := empty_functionCt emptyset.
Definition empty_function_tg (x: Set) := acreate (empty_functionCt x).
Definition empty_function:= empty_function_tg emptyset.

Lemma empty_function_tg_function x:
   function_prop (empty_function_tg x) emptyset x.
Lemma empty_function_function: function_prop empty_function emptyset emptyset.

Lemma empty_function_graph x: graph (empty_function_tg x) = emptyset.
Lemma empty_function_p1 f: function f ->
  graph f = emptyset -> source f = emptyset.
Lemma empty_function_p2 f: function f ->
  target f = emptyset -> source f = emptyset.
Lemma empty_source_graph f:
  function f -> source f = emptyset -> graph f = emptyset.
Lemma empty_target_graph f:
  function f -> target f = emptyset -> graph f = emptyset.
Lemma empty_function_tg_function x:
   function_prop (empty_function_tg x) emptyset x.
Lemma empty_source_graph2 f:
  function f -> source f = emptyset ->
  f =  empty_function_tg (target f).
```

¶ We have already met the identity function. The properties shown here are trivial.

```
Lemma identity_prop x: function_prop (identity x) x x.
Lemma identity_f x: function (identity x).
Lemma identity_V x y: inc y x -> Vf (identity x) = y.
```

We define '`identityC a`' to be the identity on a as a Coq function.

```
Definition identityC (a:Set): a->a := @id a.
```

```
Lemma identity_prop1 (a: Set): acreate (identityC a) = identity a.
Lemma identity_prop2 a:
  bcreate (identity_f a) (identity_s a) (identity_t a) =1 @id a.
```

¶ We say that a function or graph f is `constant` if $f(a) = f(b)$ whenever both terms are defined. It follows that f has a small range.

```
Definition constantp x y (f:x->y) :=  forall a b, f a = f b.
Definition constantfp f := function f /\ cstfp f (source f).
Definition constantgp f := fgraph f /\ cstgp f (domain f).

Lemma constant_prop1 f: constantgp f -> small_set (range f).
Lemma constant_prop2 f: constantfp f -> constantgp (graph f).
Lemma constant_prop3 x y: constantgp (cst_graph x y).
Lemma constant_prop4 f: function  f ->
  (constantfp f <-> small_set (range (graph f))).
```

Given two sets A and B, and $y \in$ B, one can consider the constant function with value y. Its graph is '`cst_graph A y`' or $A \times \{y\}$.

```
Section ConstantFunction.
Variables (A B y: Set).
Hypothesis (yB: inc y B).

Definition constant_function :=  acreate ([fun: A => Bo yB]).

Lemma constant_s: source (constant_function) = A.
Lemma constant_t: target (constant_function) = B.
Lemma constant_g: graph (constant_function) =  A *s1 y.
Lemma constant_f: function (constant_function).
Lemma constant_prop: function_prop (constant_function A yB) A B.
Lemma constant_V x: inc x A ->  Vf (constant_function) x = y.
Lemma constant_constant_fun: constantfp (constant_function).

End ConstantFunction.
```

We show here that every constant function is constant or empty.

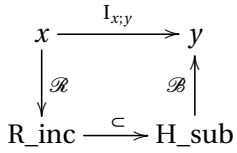
```
Lemma constant_fun_constantC x y (a:y)
  constantp ([fun:x => a]).
Lemma constant_prop5 x y (f:x->y)(b:x): constantp f ->
  exists a:y, f =1 [fun :x => a].
Lemma constant_prop6 f:
  constantfp f ->
  f = empty_function_tg (target f) \/
  exists a: target f, f = constant_function (source f)  (R_inc a).
```

3.5 Restrictions and extensions of functions

The *restriction* of a function f to a set x can be defined in different ways, for instance as the composition with the inclusion map from x to the source of f. This is the definition we shall use for Coq functions. In Bourbaki, composition is defined for correspondences, and the case of functions is studied later, in Section 3.7.

We define here the composition of two COQ functions; associativity is trivial, it suffices to unfold the definitions. Identity is a unit; this relies on the fact that f is equal to the function that maps u to $f(u)$.

```
Lemma composeC_ev a b c (g:b->c) (f: a->b) x:
   (g \o f) x = g (f x).
Lemma compositionC_A a b c d (f: c->d)(g:b->c)(h:a->b):
  (f \o g) \o h = f \o (g \o h).
Lemma compose_id_leftC (a b:Set) (f:a->b):
  (@id b) \o f =1 f.
Lemma compose_id_rightC (a b:Set) (f:a->b):
  (@id b) \o f =1 f.
```



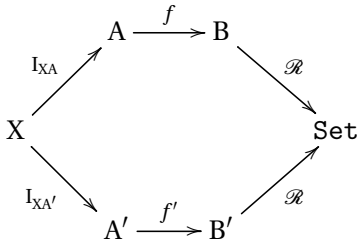
(inclusion)

We now define the inclusion I_{xy}. See diagram (inclusion) which is an instance of (a/b create). If x and y are two sets, H is the assumption $x \subset y$, if u is of type x, then '`R_inc u`' says that $\mathscr{R}u \in x$. Applying H gives $\mathscr{R}u \in y$, denoted by `H_sub` on the diagram, and using $\mathscr{B}$ yields an object of type y. The important property is $\mathscr{R}a = \mathscr{R}(\mathrm{I}_{xy}(a))$. From the injectivity of $\mathscr{R}$ we deduce $\mathrm{I}_{xx} = \mathrm{I}_x$ and $\mathrm{I}_{yz} \circ \mathrm{I}_{xy} = \mathrm{I}_{xz}$ (where `sub_refl` says $x \subset x$ and `sub_trans` expresses the transitivity of inclusion, in other words, it says that if $\mathrm{I}_{yz} \circ \mathrm{I}_{xy}$ is defined so is I_{xz}).

```
Definition inclusionC x y (H: sub x y):=
  [fun u:x => Bo (H (Ro u) (R_inc u))].

Lemma inclusionC_pr  x y (H: sub x y) (a:x):
  Ro(inclusionC H a) = Ro a.
Lemma inclusionC_identity x:
  inclusionC (sub_refl (x:=x)) =1 @id x.
Lemma inclusionC_compose x y z (Ha:sub x y)(Hb: sub y z):
  (inclusionC Hb) \o (inclusionC Ha) =1 inclusionC (sub_trans Ha Hb).
```

We say that two functions agree on a set x if this set is a subset of the sources, and if the functions take the same value on x. Consider two functions (G, A, B) and (G′, A′, B′). If A = A′ and G = G′, the functions agree on A. Conversely, the property "A ⊂ A′ and the functions agree on A" is the same as G ⊂ G′. Thus if A = A′ we have G = G′. If moreover B = B′, the functions are the same.



(restriction/agree)

In the case of COQ functions, $f : \mathrm{A} \to \mathrm{B}$ and $f' : \mathrm{A}' \to \mathrm{B}'$ agree on X if the diagram (restriction/agree) commutes.

```
Definition agrees_on x f f' :
  [/\ sub x (source f), sub x (source f') &  {inc x, f =1f f'} ].

Definition restrictionC (x a b:Set) (f:a->b)(H: sub x a) :=
   f \o (inclusionC H).
Definition agreeC (x a a' b b':Set) (f:a->b) (f':a'-> b')
  (Ha: sub x a)(Hb: sub x a') :=
  forall u:x, Ro(restrictionC  f Ha u) = Ro(restrictionC f' Hb u).

Lemma same_graph_agrees f f':
  function f -> function f' -> graph f = graph f' ->
  agrees_on (source f) f f'.

Lemma function_exten2 f f':
  function f -> function f' ->
  (f = f' <->
    [/\ source f = source f', target f = target f' & agrees_on (source f) f f']).

Lemma sub_function f g:
  function f -> function g ->
  (sub (graph f) (graph g) <-> agrees_on (source f) f g).
```

¶ We consider here the restriction of a function $f : x \to y$ as a function $z \to t$. We assume $z \subset x$, and $f\langle z\rangle \subset t \subset y$. In the first definition we consider $t = y$, and in the second definition $t = f\langle z\rangle$; in the last one, t is a parameter.

```
Definition restriction f x :=
  triple x (target f) (restr (graph f) x).
Definition restriction1 f x :=
  triple x (image_by_fun f x) (restr (graph f) x).
Definition restriction2 f x y :=
  triple x y  (graph f) \cap (x \times (target f))).
Definition restriction2_axioms f x y :=
  [/\ function f,
  sub x (source f), sub y (target f) & sub (image_by_fun f x) y].
```

Here are some properties.

```
Lemma restr_range f x:
  function f-> sub x (source f) ->
  sub (range (restr (graph f) x)) (target f).
Lemma restriction2_props f x y:
  restriction2_axioms f x y ->
  domain ((graph f) \cap (x \times (target f))) = x.

Lemma restriction1_prop f x:
  function f -> sub x (source f) ->
  function_prop (restriction1 f x) x (image_by_fun f x).
Lemma restriction_prop f x:
  function f -> sub x (source f) ->
  function_prop (restriction f x) x (target f).
Lemma restriction2_prop f x y:
  restriction2_axioms f x y ->  function_prop (restriction2 f x y) x y.

Lemma restriction_function f x:
```

```
  function f -> sub x (source f) ->
  function (restriction f x).

Lemma restriction1_function f x,:
  function f -> sub x (source f) ->
  function (restriction1 f x).
Lemma restriction_V f x:
  function f -> sub x (source f) ->
  {inc x, (restriction f x) =1f f}.
Lemma restriction1_V f x:
  function f -> sub x (source f) ->
  {inc x,  (restriction1 f x) =1f f}.
Lemma restriction2_V f x y:
  restriction2_axioms f x y ->
  {inc x,  (restriction2 f x y) =1f f}.
Lemma restriction1_pr f:
  function f -> restriction2 f (source f) (image_by_fun f (source f)) =
  restriction1 f (source f).
```

We say that $g = (G, C, D)$ *extends* $f = (F, A, B)$ if $F \subset G$ and $B \subset D$. This implies $A \subset C$. Both functions agree on A. In the case of COQ functions, there is no notion of graph, hence: for every f and g such that the source of f is a subset of the source of g, we say that g extends f if the target of f is a subset of the target of g, and if the functions agree on the source of f.

```
Definition extends g f :=
 [/\ function f, function g,  sub (graph f) (graph g)
    & sub (target f)(target g) ].
Definition extendsC (a b a' b':Set) (g:a'->b')(f:a->b)(H: sub a a') :=
  sub b b' /\ agreeC g f H (sub_refl (x:=a)).

Lemma extends_Ssource f g:
  extends g f -> sub (source f) (source g).
Lemma extends_sV f g:
  extends g f -> {inc (source f), f =1f g}.

Lemma extendsC_pr  (a b a' b':Set) (g:a'->b')(f:a->b)(H: sub a a'):
  extendsC g f H -> forall x:a, Ro (f x) = Ro(g (inclusionC H x)).
```

If f is a function, X a subset of its source, then f extends its restriction to X. If f and g are two functions with the same target, that agree on X, their restrictions to X are equal. The same is true for COQ functions. Bourbaki notes that the graph of the restriction is the intersection with the product of X and the target (but he cannot prove this statement, since intersection is not yet defined).

```
Lemma function_extends_restr f x:
  function f -> sub x (source f) ->
  extends f (restriction f x).
Lemma function_extends_restC  (x a b:Set) (f:a->b)(H:sub x a):
  extendsC f (restrictionC f H) H.
Lemma agrees_same1 f g x: agrees_on x f g -> sub x (source f).
Lemma agrees_same2 f g x: agrees_on x f g -> sub x (source g).
Lemma agrees_same_restriction f g x:
  function f -> function g -> agrees_on x f g ->
  target f = target g ->
  restriction f x = restriction g x.
```

```
Lemma agrees_same_restrictionC (a a' b x:Set) (f:a->b)(g:a'->b)
  (Ha: sub x a)(Hb: sub x a'):
  agreeC f g Ha Hb -> restrictionC f Ha =1 restrictionC g Hb.
Lemma restriction_graph1 f x:
  function f -> sub x (source f) ->
  graph (restriction_function f x) = (graph f) \cap (x \times (target f)).
Lemma restriction_recovers f x:
  function f -> sub x (source f) ->
  restriction_function f x = triple x (target f)
  ((graph f) \cap (x \times (target f))).
```

¶ The restriction agrees with f on X. If f is the extension of some function g, then g is the restriction of f to its source and target.

```
Lemma function_rest_of_prolongation f g:
  extends g f -> f = restriction2 g (source f) (target f).
```

$$\begin{array}{ccc} a & \xrightarrow{f} & b \\ {\scriptstyle \mathrm{I}_{ac}}\downarrow & & \downarrow{\scriptstyle \mathrm{I}_{bd}} \\ c & \xrightarrow[\mathrm{R}_{cd}f]{} & d \end{array} \qquad \text{(restriction2C)}$$

In the case of Coq functions, we start with a function $f : a \to b$, with the assumptions $c \subset a$ and $d \subset b$. The restriction $\mathrm{R}_{cd}f$ is the one that makes diagram (restriction2C) commute. In order for it to exist, each y in the image of the LHS must be convertible to type d, i.e. $\mathscr{R}y \in d$.

```
Definition restriction2C (a a' b b':Set) (f:a->b)(Ha:sub a' a)
  (H: forall u:a', inc (Ro (f (inclusionC Ha u))) b') :=
  fun u=> Bo (H u).
Lemma restriction2C_pr(a a' b b':Set) (f:a->b)(Ha:sub a' a)
  (H: forall u:a', inc (Ro (f (inclusionC Ha u))) b') (x:a'):
  Ro (restriction2C f Ha H x) = Vf (acreate f) (Ro x).
Lemma restriction2C_pr1 (a a' b b':Set)  (f:a->b)
  (Ha:sub a' a)(Hb:sub b' b)
  (H: forall u:a', inc (Ro (f (inclusionC Ha u))) b'):
  f \o (inclusionC Ha) =1  (inclusionC Hb) \o (restriction2C f Ha H).
```

3.6 Definition of a function by means of a term

In Bourbaki [2, p. 83], Criterion C54 says that if $\boldsymbol{A}$ and $\boldsymbol{T}$ are two terms, $\boldsymbol{x}$ and $\boldsymbol{y}$ are two distinct letters, $\boldsymbol{x}$ is not in $\boldsymbol{A}$, $\boldsymbol{y}$ is neither in $\boldsymbol{T}$ nor in $\boldsymbol{A}$, then the relation $\boldsymbol{x} \in \boldsymbol{A}$ and $\boldsymbol{y} = \boldsymbol{T}$ admits a graph $\boldsymbol{F}$, which is functional, and $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{T}$. If $\boldsymbol{C}$ is a set which contains the set $\boldsymbol{B}$ of objects of the form $\boldsymbol{T}$ for $\boldsymbol{x} \in \boldsymbol{A}$ (where $\boldsymbol{y}$ does not appear in $\boldsymbol{C}$), the function $(\boldsymbol{F}, \boldsymbol{A}, \boldsymbol{C})$ is also denoted by the notation $\boldsymbol{x} \to \boldsymbol{T}$ $(\boldsymbol{x} \in \boldsymbol{A}, \boldsymbol{T} \in \boldsymbol{C})$, where the terms in parentheses may be omitted. It can also be written as $(\boldsymbol{T})_{\boldsymbol{x} \in \boldsymbol{A}}$. In what follows, we shall use $x \mapsto \mathrm{T}$ to denote the function that associates T to x, and $x \to \mathrm{T}$ to mean a function from the set (or type) x to the set (or type) T.

The non-trivial point is the existence of the set $\boldsymbol{B}$, since $\boldsymbol{F}$ is then a subset of $\boldsymbol{A} \times \boldsymbol{B}$. The range of $\boldsymbol{F}$ is $\boldsymbol{B}$, so that $(\boldsymbol{F}, \boldsymbol{A}, \boldsymbol{C})$ is a function when $\boldsymbol{B} \subset \boldsymbol{C}$. In these definitions, $\boldsymbol{y}$ is just an

auxiliary letter (because it neither appears in ***A***, ***B***, ***T*** nor ***F***). On the other hand, ***x*** may appear in ***T***, it does not appear in ***A***, ***B***, nor ***F***.

If we have an object $f : \mathrm{E} \to \mathrm{E}$, and consider $\mathrm{T} = f(x)$ then $\mathrm{F} = \mathscr{L}_{\mathrm{A}} f$. The second claim of the criterion, namely $\mathrm{F}(x) = \mathrm{T}$, is just $\mathscr{V}(x, \mathscr{L}_{\mathrm{A}} f) = f(x)$ (see Section 2.11). The function (F, A, C) will be denoted by `BL f A C`, or $\mathscr{L}_{\mathrm{A;C}} f$. The following lemmas are obvious from the definitions of $\mathscr{L}_{\mathrm{A}}$ and $\mathscr{L}_{\mathrm{A;C}}$.

```
Definition Lf f a b :=  triple a b (Lg a f).

Lemma lf_source f a b: source (Lf f a b) = a.
Lemma lf_target f a b: target (Lf f a b) = b.
Lemma lf_graph1 f a b c:
  inc c (graph (Lc f a b)) -> c = J (P c) (f (P c)).
Lemma lf_graph2 f a b c:
  inc c a -> inc (J c (f c)) (graph (Lf f a b)).
Lemma lf_graph3 f a b c:
  inc c (graph (Lf f a b)) -> inc (P c) a.
Lemma lf_graph4 f a b c:
  inc c (graph (Lf f a b)) -> f (P c) = (Q c).
```

The expression $\mathscr{L}_{\mathrm{A;B}} f$ is a function if f maps A into B. If $x \in \mathrm{A}$, the value at x is $f(x)$. By extensionality, if f is a function with source A, target B, and evaluation function $\mathscr{W}_f$, then $\mathscr{L}_{\mathrm{A;B}} \mathscr{W}_f = f$.

```
Definition lf_axiom f a b :=
  forall c, inc c a -> inc (f c) b.

Lemma lf_function f a b:
  lf_axiom f a b -> function (Lf f a b).

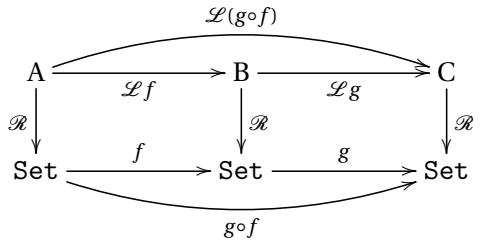
Lemma lf_V f a b c:
  lf_axiom f a b ->  inc c a -> Vf c (Lf f a b) = f c.
Lemma lf_recovers f:
  function f -> Lf (Vf f)(source f)(target f) = f.
Lemma identity_Lf x: identity x = Lf id x x.
Lemma resrtriction_Lf f x: function f -> sub x (source f) ->
 restriction f x = Lf (Vf f) x (target f).
```

We consider here an example of a function defined by a term, the first and second projection, denoted pr_1 and pr_2, on the range and target.

```
Definition first_proj g := Lf P g (domain g).
Definition second_proj g := Lf Q g (range g).

Lemma first_proj_V g: {inc g, Vf (first_proj g) =1 P}.
Lemma second_proj_V g: {inc g, Vf (second_proj g) =1 Q}.
Lemma first_proj_f g:  function (first_proj g).
Lemma second_proj_f g:  function (second_proj g).
```

3.7 Composition of two functions. Inverse function



(composition)

We say that f = (F, A, B) and g = (G, B, C) are `composable` if they are functions and if they are composable as correspondences. Their graphs are `composablef`. Proposition 6 [2, p. 84] says that the composition is a function. The evaluation function is the composition of the evaluation functions. We have $\mathscr{L}(g\circ f) = (\mathscr{L}g)\circ(\mathscr{L}f)$. In other words, the two definitions of composition (for Bourbaki and Coq functions) are really the same.

```
Definition composable g f :=
  [/\ function g, function f,  source g = target f].
Notation "f1 \coP f2" := (composable f1 f2) (at level 50).

Lemma composable_pr f g: g \coP f -> (graph g) \cfP (graph f).
Lemma compf_graph:
  {when: composable, {morph graph: f g / f \co g >-> f \cf g}}.
  (* g \coP f -> graph (g \co f) = (graph g) \cf (graph f) *)
Lemma compf_domg g f: g  \coP f->
  domain (graph (g \co f)) = domain (graph f).

Lemma compf_s f g: source (g \co f) = source f.
Lemma compf_t f g: target (g \co f) = target g.
Theorem compf_f  g f: g \coP f  -> function (g \co f).

Lemma compf_V g f x: g \coP f ->
  inc x (source f) ->   inc x (source f) -> Vf (g \co f) x = Vf g (Vf f x).

Lemma composable_acreate (a b c:Set) (f: a-> b)(g: b->c):
  (acreate g) \coP (acreate f).
Lemma compose_acreate (a b c:Set) (g: b->c)(f: a-> b):
  (acreate g) \co (acreate f) = acreate(g \o f).
```

Composition is associative, and identity is a unit. One could write the first lemma as `{when ??, associative compose}`, but this is horrible.

```
Lemma compfA f g h: f \coP g -> g \coP h ->
   f \co (g \co h) = (f \co g) \co h.
Lemma compg_id_l m:
  function m -> (identity (target m)) \co m = m.
Lemma compf_id_r m:
  function m -> m \co (identity (source m)) = m.
Corollary fcomp_identity_left E:
  {when (fun x => function x /\ (target x) = E),
  left_id (identity E) compose}.
```

We say that f is *injective* if it is a function such that $f(x) = f(y)$ implies $x = y$. We say that f is *surjective* if the range of its graph is the target. The phrase "f is a mapping of A onto B" is

sometimes used by Bourbaki as a shorthand of "f is surjective, its source is A, and its target is B". We say that f is *bijective* if it satisfies both properties. We list here some trivial properties. Note that two surjective functions with the same source and evaluation functions have the same graph, thus the same target, thus are equal.

```
Definition injection f:=
  function f /\ {inc source f &, injective (Vf f)}.

Definition surjection f :=
  function f /\
  (forall y, inc y (target f) -> exists2 x, inc x (source f) & Vf f x = y).
Definition bijection f :=
  injection f /\ surjection f.

Lemma inj_function f: injection f -> function f.
Lemma surj_function f: surjection f-> function f.
Lemma bij_function f: bijection f-> function f.
Lemma bij_inj f: bijection f -> {inc source f &, injective (Vf f)}.
Lemma bij_surj f: bijection f ->
  (forall y, inc y (target f) -> exists2 x, inc x (source f) & Vf f x = y).

Lemma injective_pr f y: injection f ->
   singl_val (fun x => related (graph f) x y).
Lemma injective_pr3 f y: injection f ->
   singl_val (fun x => inc (J x y) (graph f)).
Lemma injective_pr_bis f:
  function f -> (forall y,  singl_val (fun x => related (graph f) x y)) ->
  injection f.

Lemma surjective_pr0 f: surjection f -> image_of_fun f = target f.
Lemma surjective_pr1 f: function f -> image_of_fun f = target f ->
  surjection f.
Lemma surjective_pr f y:
  surjection f -> inc y (target f) ->
  exists2 x, inc x (source f) & related (graph f) x y .
Lemma surjective_pr5 f:
  function f -> (forall y, inc y (target f) ->
  exists2 x, inc x (source f) & related (graph f) x y) -> surjection f.
Lemma surjective_pr3 f:
  surjection f -> range (graph f) = target f.
Lemma surjective_pr4 f:
  function f-> range (graph f) = target f -> surjection f.

Lemma lf_injective f a b: lf_axiom f a b ->
  (forall u v, inc u a-> inc v a -> f u = f v -> u = v) ->
  injection (Lf f a b).
Lemma lf_surjective f a b: lf_axiom f a b ->
  (forall y, inc y b -> exists2 x, inc x a & y = f x) ->
  surjection (Lf f a b).
Lemma lf_bijective f a b: lf_axiom f a b ->
  (forall u v, inc u a-> inc v a -> f u = f v -> u = v) ->
  (forall y, inc y b -> exists2 x, inc x a & y = f x) ->
  bijection (Lf f a b).
Lemma bijective_pr f y:
  bijection f -> inc y (target f) ->
  exists! x, (inc x (source f) /\ Lf f x = y).
```

```
Lemma function_exten4 f g: source f = source g ->
  surjection f -> surjection g -> {inc source f, f =1f g} ->
   f = g.
```

Let's consider the case of Coq functions. Assume $f : A \to B$ surjective. Whenever $b : B$, there is $a : A$ such that $f(a) = b$. The axiom of choice gives a function g, such that $f(g(b)) = b$. It is called a "right inverse" of f. If f is moreover injective, we get $g(f(a)) = a$, and g is the inverse of f. In particular, our definition of bijective agrees with that of Coq.

```
Definition injectiveC (a b:Set) (f:a->b) := forall u v, f u = f v -> u =v.
Definition surjectiveC (a b:Set) (f:a->b) := forall u, exists v, f v = u.
Definition bijectiveC (a b:Set) (f:a->b) := injectiveC f /\ surjectiveC f.
Definition right_inverseC (a b:Set) (f: a->b)  (H:surjectiveC f) (v:b) :=
   (chooseT (fun k:a => f k = v)
     match H v with | ex_intro x _ => inhabits x end).
Definition inverseC  (a b:Set) (f: a->b) (H:bijectiveC f)
   := right_inverseC (proj2 H).

Lemma bijectiveC_pr (a b:Set) (f:a->b) (y:b):
  bijectiveC f -> exists! x:a, f x = y.
Lemma composeC_inj (a b c:Set) (f:a->b)(f':b->c):
  injectiveC f-> injectiveC f' -> injectiveC (f' \o f).
Lemma composeC_surj (a b c:Set) (f:a->b)(f':b->c):
  surjectiveC f-> surjectiveC f' -> surjectiveC (f' \o f).
Lemma composeC_bij (a b c:Set) (f:a->b)(f':b->c):
  bijectiveC f-> bijectiveC f' -> bijectiveC (f' \o f).
Lemma identityC_fb (x: Set):  bijectiveC (@id x).

Section InverseProps.
Variables (A B: Set) (f: A -> B).
Hypothesis (H:bijectiveC f).

Lemma inverseC_prb (x: B): f ((inverseC H) x) = x.
Lemma inverseC_pra (x: A):  (inverseC H) (f x) = x.
Lemma bij_left_inverseC: (inverseC H) \o f =1 @id A.
Lemma bij_right_inverseC: f \o (inverseC H) =1 @id B.
Lemma bijective_inverseC:  bijectiveC (inverseC H).
End InverseProps.

Lemma bijection_coq (a b: Set) (f:a->b):
  bijective f <-> bijectiveC f.
Lemma inverse_fun_involutiveC (a b:Set) (f:a->b) (H: bijectiveC f):
  f =1 inverseC(bijective_inverseC H).
```

The acreate/bcreate mappings are morphisms for the notion of injectivity, surjectivity and bijectivity.

```
Lemma bcreate_fi f a b
  (H:function f)(Ha:source f = a)(Hb:target f = b):
  injection f -> injectiveC (bcreate H Ha Hb).
Lemma bcreate_fs f a b
  (H:function f)(Ha:source f = a)(Hb:target f = b):
  surjection f -> surjectiveC (bcreate H Ha Hb).
Lemma bcreate_fb f a b
```

```
  (H:function f)(Ha:source f = a)(Hb:target f = b):
  bijection f -> bijectiveC (bcreate H Ha Hb).

Lemma bcreate1_fi f (H:function f),
  injection f -> injectiveC (bcreate1 H).
Lemma bcreate1_fs f (H:function f):
  surjection f -> surjectiveC (bcreate1 H).
Lemma bcreate1_fb f (H:function f):
  bijection f -> bijectiveC (bcreate1 H).

Lemma acreate_fi (a b:Set) (f:a->b):
  injectiveC f -> injective (acreate f).
Lemma acreate_fs (a b:Set) (f:a->b):
  surjectiveC f -> surjective (acreate f).
Lemma acreate_fb (a b:Set) (f:a->b):
  bijectiveC f -> bijective (acreate f).
```

We say that two sets are *equipotent* if there is a bijection between them. Which definition of bijection used is irrelevant. Note that Bourbaki uses a prefix notation Eq(X,Y), while we use here an infix notation `X \Eq Y`. This is an equivalence relation, since identity is a bijection, composition of two bijections is a bijection and the inverse of a bijection is a bijection.

```
Definition bijection_prop f s t :=
  [/\ bijection f, source f = s & target f = t].
Definition surjection_prop f x y:=
  [/\ surjection f, source f = x & target f = y].
Definition injection_prop f x y:=
  [/\ injection f, source f = x & target f = y].

Definition equipotent x y :=
 Notation "x \Eq y" := (equipotent x y) (at level 50).

Lemma equipotent_aux f a b:
  bijection (Lf f a b) -> a \Eq b.
Lemma equipotentC x y: x \Eq y <-> exists f:x->y, bijectiveC f.
Lemma equipotent_aux f a b:
  bijection (Lf f a b) -> a \Eq b.
Lemma equipotentR: reflexive_r equipotent.
Lemma equipotentT: transitive_r equipotent.
Lemma equipotentS: symmetric_r equipotent.
```

The identity function is bijective; the restriction of a function f to X and Y is injective if f is injective; it is surjective if for instance X is the source and Y the range. It is surjective if $Y = f(X)$.

```
Lemma identity_fb x:  bijection (identity x).
Lemma restriction2_fi f x y:
  injection f -> restriction2_axioms f x y
  -> injection (restriction2 f x y).
Lemma restriction2_fs f x y:
  restriction2_axioms f x y ->
  source f = x -> image_of_fun f = y ->
  surjection (restriction2 f x y).
Lemma restriction1_fs f x:
  function f -> sub x (source f) ->
```

```
  surjection (restriction1 f x).
Lemma restriction1_fb f x:
  injection f -> sub x (source f) ->
  bijection (restriction1 f x).
```

We deduce: if f is injective, its source is equipotent to its range.

```
Definition restriction_to_image f :=
  restriction2  f (source f) (image_of_fun f).

Lemma restriction_to_imageE f: function f ->
  restriction_to_image f = restriction1 f (source f).

Lemma restriction_to_image_axioms f: function f ->
  restriction2_axioms f (source f) (image_of_fun f).
Lemma restriction_to_image_fs f: function f ->
  surjection (restriction_to_image f).
Lemma restriction_to_image_fb f: injection f ->
  bijection (restriction_to_image f).
Lemma iim_fun_r f (h:=restriction_to_image f): function f ->
   forall a, image_by_fun f a = inv_image_by_fun (inverse_fun h) a.

Lemma equipotent_restriction1  f x:
  sub x (source f) -> injection f ->
  x \Eq (image_by_fun f x).
Lemma equipotent_range f: injection f ->
  (source f) \Eq (range (graph f)).
```

¶ Given a correspondence f and a pair (x, y), we can extend f as f' by imposing $f'(x) = y$; this is a correspondence, it is a function if x is not in the source of f. This extension is unique if we merely add x to the source, y to the target and (x, y) to the graph. The extension is a surjective function if f is surjective.

```
Definition extension f x y:=
  triple ((source f) +s1 x) ((target f) +s1 y)  ((graph f) +s1 (J x y)).

Lemma extension_injective x f g a b:
  domain f = domain g -> ~ (inc x (domain f)) ->
  (f +s1 (J x a) = g +s1 (J x b)) -> f = g.
Lemma restr_setU1 f x a:
  fgraph f -> ~ (inc x (domain f)) ->
  restr (f +s1 (J x a)) (domain f) = f.
Lemma setU1_restr f x E:
   fgraph f -> ~ (inc x E) -> domain f = E +s1 x->
   (restr f E) +s1 (J x (Vg f x)) = f.

Section Extension.
Variables (f x y: Set).
Hypothesis (ff: function f) (xsf:  ~ (inc x (source f))).

Lemma extension_f: function (extension f x y).
Lemma extension_Vf_in: {inc (source f), (extension f x y) =1f f}.
Lemma extension_Vf_out: Vf (extension f x y) x = y.
Lemma extension_fs: surjection f -> surjection (extension f x y).
Lemma extension_restr:
    restriction2 (extension f x y) (source f) (target f) = f.
```

```
End Extension.


Lemma extension_f_injective x f g a b:
  function f -> function g -> target f = target g ->
  source f = source g -> ~ (inc x (source f)) ->
  (tack_on_f f x a = tack_on_f g x b) -> f = g.
```

¶ The canonical injection of A into B is the identity of B restricted to A. In other terms, if $A \subset B$ it is the function with source A, target B, whose evaluation function is $x \mapsto x$. It is injective with range A. Its COQ equivalent has been introduced page 54.

```
Definition canonical_injection a b :=
  triple a b (identity_g a).

Lemma canonical_injection_p1 a b (H:sub a b):
   (canonical_injection a b) = acreate  (inclusionC H).
Lemma inclusionC_fi a b (H: sub a b):  injectiveC (inclusionC H).
Lemma ci_fi a b: sub a b -> injection (canonical_injection a b).
Lemma ci_f a b: sub a b ->  function  (canonical_injection a b).
Lemma ci_V a b x:
  sub a b -> inc x a -> Vf (canonical_injection a b) x = x.
Lemma ci_range a b: sub a b ->
  range (graph (canonical_injection a b)) = a.
```

A constant function h can be written as $h = g \circ f$ where the image of f is a singleton and f is surjective surjective (let x be in the source of h, $y = h(x)$, so that $E = \{y\}$ is the image of h; let g be the canonical injection of E into the target of h, and f the constant function with value y).

```
Lemma constant_function_pr2 x h:
  inc x (source h) -> constantfp h ->
  exists g f,
  [/\ g \coP f, h = g \co f, surjection f & singletonp (target f)].
Lemma constant_function_pr3 (a:Set) (h:a->Set) (x:a):
  qconstantp h ->
  exists f: a -> singleton (Ro x),
   exists g:singleton (Ro x) -> Set,
     (forall u:a, h u = g (f u)) /\ (g (Bo (set1_1 (Ro x))) = h x).
```

¶ The diagonal application is the function from X to $X \times X$ that maps x to (x,x). It is an injection into the diagonal of X.

```
Definition diagonal_application a :=
  Lf (fun x=> J x x) a (coarse a).

Lemma diag_app_f a: function (diagonal_application a).
Lemma diag_app_fi a: injection (diagonal_application a).
Lemma diag_app_V a x:
  inc x a -> Vf (diagonal_application a) x  = J x x.
Lemma diag_app_r  a:
  range (graph (diagonal_application a)) = diagonal a.
```

¶ Both projections pr_1 and pr_2 are surjective by construction. The first projection on G is injective if only if G is a functional graph.

```
Lemma second_proj_fs g: surjection (second_proj g).
Lemma first_proj_fs g:  surjection (first_proj g).
Lemma first_proj_fi g:
  sgraph g -> (injection (first_proj g) <-> functional_graph g).
```

¶ If G is a graph, the map $(x, y) \mapsto (y, x)$ is a bijection $G \to G^{-1}$. It follows that $A \times B$ and $B \times A$ are equipotent. The three sets A, $A \times \{b\}$ and $\{b\} \times A$ are equipotent.

```
Lemma inv_graph_canon_fb g: sgraph g ->
  bijection ( Lf (fun x=> J (Q x) (P x)) g (inverse_graph g)).
Lemma equipotent_product_sym  a b:
  (a \times b) \Eq (b \times a).
Lemma equipotent_indexed a b:  a \Eq (a *s1 b).
Lemma equipotent_rindexed a b:  a \Eq (indexedr b a).
Lemma equipotent_source_graph f: function f ->
  (graph f) \Eq  (source f).
```

Proposition 7 [2, p. 85] states that if f is a bijection, then the inverse correspondence f^{-1} is a function. It also says that if f and f^{-1} are functions then f is a bijection.

```
Theorem bijective_inv_f f:
  bijection f -> function (inverse_fun f).
Theorem inv_function_fb f:
  function f -> function (inverse_fun f) -> bijection f.
```

The case of Coq functions is a bit more tricky: if $f : A \to B$ is a function, its inverse is $f^{-1} = \mathscr{M}_{B;A}((\mathscr{L} f)^{-1})$. However $\mathscr{M}$ can be used only if some conditions hold (in particular, if B is non-empty, the set A has to be non-empty too). For this reason, we consider the inverse of f only when f is bijective.

```
Lemma inverseC_prc (a b:Set) (f:a-> b) (H:bijectiveC f):
  inverse_fun(acreate f) = acreate(inverseC H).
```

If a function has a left and right inverse, the function is bijective, and its inverse is equal to these inverses. In fact, if $f(g(x)) = x$ for all x, then f is surjective, since every x is the image of $g(x)$. If $g'(f(y)) = y$, applying g' to $f(y) = f(y')$ gives $y = y'$, hence proves injectivity. Now, $g'(f(g(x))) = g'(x) = g(x)$, this shows that $g = g'$. We have $g'(x) = f^{-1}(x)$, since $x = f(g(x))$, and, by definition, the RHS is $g(x)$. We have already seen that the LHS is this quantity.

We deduce from this that the inverse function of a bijection is a bijection.

```
Lemma bijective_double_inverseC (a b:Set) (f:a->b) g g':
  g \o f =1 @id a -> f \o g' =1 @id b ->
  bijectiveC f.
Lemma bijective_double_inverseC1 (a b:Set) (f:a->b) g g'
  (Ha: g \o f =1 @id a)(Hb: f \o g' =1 @id b):
  g =1 inverseC (bijective_double_inverseC Ha Hb)
  /\ g' =1 inverseC (bijective_double_inverseC Ha Hb).
```

If f is a bijective, then f^{-1} is also a bijective. The composition in any order is the identity function. The proofs of these three lemmas are similar: let $g = \mathscr{M}_{a;b} f$; then $f = \mathscr{L} g$, $f^{-1} = \mathscr{L}(g^{-1})$, $f \circ f^{-1} = \mathscr{L}(g \circ g^{-1})$.

```
Lemma acreate_exten (a b: Set) (f g: a-> b):
  f =1 g -> acreate f = acreate g.
Lemma ifun_involutive:  {when function, involutive inverse_fun}.

Lemma inverse_bij_fb f:
  bijection f ->  bijection (inverse_fun f).
Lemma composable_f_inv  f:
  bijection f ->  f \coP (inverse_fun f).
Lemma composable_inv_f f:
  bijection f ->  (inverse_fun f) \coP f.
Lemma bij_right_inverse f:
  bijection f ->  f \co (inverse_fun f) = identity (target f).
Lemma bij_left_inverse f:
  bijection f ->  (inverse_fun f) \co f = identity (source f).
Lemma compf_lK f g : bijection g -> g \coP f ->
  (inverse_fun g) \co (g \co f) = f.
Lemma compf_rK f g : bijection g -> f \coP g ->
  (f \co g) \co (inverse_fun g) = f.
Lemma compf_regr f f' g : bijection g ->
  g \coP f -> g \coP f' -> g \co f = g \co f' -> f = f'.
Lemma compf_regl f f' g : bijection g ->
  f \coP g -> f' \coP g -> f \co g = f' \co g -> f = f'.

Lemma inverse_V f x:
  bijection f -> inc x (target f) ->
  Vf f (Vf (inverse_fun f) x) = x.
Lemma inverse_V2 f y:
  bijection f -> inc y (source f) ->
  Vf (inverse_fun f) (Vf f y) = y.
Lemma inverse_Vis f x:
  bijection f -> inc x (target f) -> inc (Vf (inverse_fun f) x) (source f).
```

We apply the results of COQ functions to Bourbaki functions. Note that Bourbaki shows that the inverse $h = f^{-1}$ is a bijection by noting that its inverse is f, hence is a function and Proposition 7 [2, p. 85] applies. The relation $x = \mathscr{W}_f y$ is equivalent to $y = \mathscr{W}_{f^{-1}} x$ if either x is in the target of f or y in the source.

```
Lemma bijective_inv_aux a b (f:a->b):
  bijectiveC f -> function (inverse_fun (acreate f)).
Lemma bijective_source_aux a b (f:a->b):
  source (inverse_fun (acreate f)) = b.
Lemma bijective_target_aux a b (f:a->b):
  target (inverse_fun (acreate f)) = a.
```

Let f be a function from A to B. We have shown before that $x \subset f^{-1}\langle f\langle x\rangle\rangle$ if $x \subset$ A (this is true for any correspondence). Equality holds if f is injective. We have $f\langle f^{-1}\langle y\rangle\rangle \subset y$ if $y \subset$ B. Equality holds if f is surjective.

```
Lemma sub_inv_im_source f y:
  function f -> sub y (target f) ->
  sub (inv_image_by_fun f y) (source f).
Lemma nonempty_image f x:
  function f -> nonempty x -> sub x (source f) ->
  nonempty (image_by_fun f x).
Lemma direct_inv_im f y:
```

```
  function f ->  sub y (target f) ->
  sub (image_by_fun f (image_by_fun (inverse_fun f) y)) y.

Lemma direct_inv_im_surjective f y:
  surjection  f -> sub y (target f) ->
  (image_by_fun f (image_by_fun (inverse_fun f) y)) = y.

Lemma inverse_direct_image f x:
  function f -> sub x (source f) ->
  sub x (image_by_fun (inverse_fun f) (image_by_fun f x)).

Lemma inverse_direct_image_inj f x:
  injection f -> sub x (source f) ->
   x = (image_by_fun (inverse_fun f) (image_by_fun f x)).
```

3.8 Retractions and sections



(retraction/section)

A *retraction* r of f is a right inverse; a *section* s is a left inverse. This means that $r \circ f$ and $f \circ s$ are the identity functions. Assume f is a function from A to B. The definition of r implies the existence of $r \circ f$, i.e. the source of r is B. A consequence is that the target is A. In the same way, the definition of s implies the existence of $f \circ s$, i.e. the target of s is A. A consequence is that the source is B. In the case of Coq functions, if f has type $a \to b$, its inverse r or s has type $b \to a$ (there is a unique type for r compatible with the relation $r \circ f = \mathrm{I_A}$).

```
Definition is_left_inverse f r :=
  r \coP f /\ r \co f = identity (source f).

Definition is_right_inverse f s :=
  f \coP s /\ f \co s = identity (target f).

Definition is_left_inverseC (a b:Set)  (f:a->b) r:= r \o f =1 @id a.
Definition is_right_inverseC (a b:Set) (f:a->b) s:= f \o s =1 @id b.

Lemma left_i_target f r:  is_left_inverse f r ->  target r = source f.
Lemma left_i_source f r: is_left_inverse f r ->  source r = target f.
Lemma right_i_source f s: is_right_inverse f s ->  source s = target f.
Lemma right_i_target f s: is_right_inverse f s ->  target s = source f.

Lemma right_i_V f s x:
  is_right_inverse f s -> inc x (target f) -> Vf f (Vf s x) = x.
Lemma left_i_V f r x:
  is_left_inverse f r -> inc x (source f) -> Vf r (Vf f x) = x.
Lemma right_i_v (a b:Set) (f:a->b) s (x:b):
  is_right_inverseC f s ->  f (s x) = x.
Lemma left_i_v (a b:Set)  (f:a->b) r (x:a):
  is_left_inverseC f r -> r (f x) = x.
```

Proposition 8 [2, p. 86] expresses the next four theorems. Assume that f is a function from A to B. If for some function s, $f \circ s = \mathrm{I_B}$ then f is surjective; if for some function r, $r \circ f = \mathrm{I_A}$

then f is injective. The converse holds; one has to take care that if $A = \emptyset$, every function is injective, and there is in general no function from B to A (unless B is empty). Hence for the retraction r to exist, we assume $A \neq \emptyset$. We start with the easy case.

```
Lemma inj_if_exists_left_invC (a b:Set) (f:a-> b):
  (exists r, is_left_inverseC f r) -> injectiveC f.
Lemma surj_if_exists_right_invC (a b:Set) (f:a->b):
  (exists s, is_right_inverseC f s) -> surjectiveC f.

Theorem inj_if_exists_left_inv f:
  (exists r, is_left_inverse f r) -> injection f.
Theorem surj_if_exists_right_inv f:
  (exists s, is_right_inverse f s) -> surjection f.
```

¶ Consider a function $f : a \to b$. For $x : b$ we consider "$f(y) = x$ or x is not in the image of f", and apply the axiom of choice to select an element y, call it $g(x)$. If $x = f(z)$, such a y exists, hence $f(g(f(z))) = f(z)$. If f is injective, we have $g(f(z)) = z$, and g is a left inverse of f. In the second case, y exists also as we assume a non-empty.

```
Definition left_inverseC (a b:Set) (f: a->b)(H:inhabited a)
  (v:b) := (chooseT (fun u:a => (~ (exists x:a, f x = v)) \/ (f u = v)) H).
Lemma left_inverseC_pr (a b:Set) (f: a->b) (H:inhabited a) (u:a):
  f(left_inverseC f H (f u)) = f u.
Lemma left_inverse_comp_id (a b:Set) (f:a->b) (H:inhabited a):
  injectiveC f ->   (left_inverseC f H) \o f =1 @id a.
Lemma exists_left_inv_from_injC (a b:Set) (f:a->b): inhabited a ->
  injectiveC f -> exists r, is_left_inverseC r f.

Definition right_inverseC (a b:Set) (f: a->b)  (H:surjectiveC f) (v:b) :=
   (chooseT (fun k:a => f k = v)
     match H v with | ex_intro x _ => inhabits x end).
Lemma right_inverse_pr (a b:Set) (f: a->b) (H:surjectiveC f) (x:b):
  f(right_inverseC H x) = x.
Lemma right_inverse_pr  (a b:Set) (f: a->b) (H:surjectiveC f) (x:b):
  f(right_inverseC H x) = x.
Lemma right_inverse_comp_id (a b:Set) (f:a-> b) (H:surjectiveC f):
  f \o (right_inverseC H)  =1 @id b.
Lemma exists_right_inv_from_surjC (a b:Set) (f:a-> b)(H:surjectiveC f):
  exists s, is_right_inverseC f s.
```

Bourbaki shows existence of a left inverse of the function $f : A \to B$ by considering the subset of $B \times A$ formed of all pairs (x, y) such that $y \in A$ and $y = f(x)$ or $y = e$ and $x \in B \setminus f\langle A\rangle$, where $e \in A$ (such an element exists when A is nonempty). This set is a functional graph, and the function with this graph is an answer to the question.

```
Theorem exists_left_inv_from_inj f:
  injection f -> nonempty (source f) -> exists r, is_left_inverse f r.
Theorem exists_right_inv_from_surj f:
  surjection  f -> exists s, is_right_inverse f s.
```

¶ Some consequences. If r is a left inverse of f, then f is a right inverse of r, and vice versa. A left inverse is surjective, a right inverse is injective. If g is both a left inverse and a right inverse of f, then g is bijective as well as f.

```
Lemma bijective_from_compose g f:
   g \coP f -> f \coP g -> g \co f = identity (source f)
  -> f \co g = identity (source g)
  -> [/\ bijection f, bijection g & g = inverse_fun f].

Lemma right_inverse_from_leftC (a b:Set) (r:b->a)(f:a->b):
  is_left_inverseC f r -> is_right_inverseC r f.
Lemma left_inverse_from_rightC (a b:Set) (s:b->a)(f:a->b):
  is_right_inverseC f s -> is_left_inverseC s f.
Lemma left_inverse_surjectiveC (a b:Set) (r:b->a)(f:a->b):
  is_left_inverseC f r -> surjectiveC r.
Lemma right_inverse_injectiveC (a b:Set) (s:b->a)(f:a->b):
  is_right_inverseC f s -> injectiveC s.
Lemma section_uniqueC (a b:Set) (f:a->b)(s:b->a)(s':b->a):
  is_right_inverseC f s -> is_right_inverseC f s' ->
  (forall x:a, (exists u:b, x = s u) = (exists u':b, x = s' u')) ->
  s =1 s'.

Lemma right_inverse_from_left f r:
  is_left_inverse f r -> is_right_inverse r f.
Lemma left_inverse_from_right s f:
  is_right_inverse f s -> is_left_inverse s f.
Lemma left_inverse_fs f r:
  is_left_inverse f r -> surjection r.
Lemma right_inverse_fi f s:
  is_right_inverse f s -> injection s.
Lemma section_unique f:
 {when  (is_right_inverse f) &, injective (fun s => range (graph s)) }.
```

Theorem 1 in Bourbaki [2, p. 87] comes next. We assume that f and f' are two composable functions and $f'' = f' \circ f$. If f and f' are injective so is f'', if f and f' are surjective, so is f''. Hence, if f and f' are bijections, so is f''. If f and f' have a left inverse, so has f'' (it is the composition of the inverses in reverse order). The same holds for right inverses.

If f'' has a left inverse r'' then $r'' \circ f'$ is a right inverse of f, and $f \circ r''$ is a left inverse of f' provided that f is surjective (in which case f is invertible). If f'' has a right inverse s'' then $f \circ s''$ is a right inverse of f' and $s'' \circ f'$ is a left inverse of f, provided that f' is injective, in which case f' is a bijection.

If f'' is injective then f is injective, and for f' to be injective it suffices that f is surjective; if f'' is surjective then f' is surjective; and for f to be surjective it suffices that f' is injective.

```
Lemma left_inverse_composeC (a b c:Set)
  (f:a->b) (f':b->c)(r:b->a)(r':c->b):
  is_left_inverseC  f' r' -> is_left_inverseC f r ->
  is_left_inverseC  (f' \o f) (r \o r').
Lemma right_inverse_composeC  (a b c:Set)
  (f:a->b) (f':b->c)(s:b->a)(s':c->b):
  is_right_inverseC f' s' -> is_right_inverseC f s ->
  is_right_inverseC (f' \o f) (s \o s') .
Lemma inj_right_composeC (a b c:Set) (f:a->b) (f':b->c):
  injectiveC (f' \o f) -> injectiveC f.
Lemma surj_left_compose (a b c:Set) (f:a->b) (f':b->c):
  surjectiveC (f' \o f) -> surjectiveC f'.
Lemma surj_left_compose2C (a b c:Set) (f:a->b) (f':b->c):
  surjectiveC (f' \o f) -> injectiveC f' -> surjectiveC f.
```

```
Lemma inj_left_compose2C (a b c :Set)(f:a->b) (f':b->c):
  injectiveC (f' \o f) -> surjectiveC f -> injectiveC f'.
Lemma left_inv_compose_rfC (a b c:Set) (f:a->b) (f':b->c)(r'': c->a):
  is_left_inverseC  (f' \o f) r'' ->
  is_left_inverseC f (r'' \o f').
Lemma right_inv_compose_rfC (a b c:Set) (f:a->b) (f':b->c)(s'': c->a):
  is_right_inverseC (f' \o f) s'' ->
  is_right_inverseC f' (f \o s'').
Lemma left_inv_compose_rf2C (a b c:Set) (f:a->b) (f':b->c)(r'': c->a):
  is_left_inverseC (f' \o f) r'' -> surjectiveC f ->
  is_left_inverseC f' (f \o r'').
Lemma right_inv_compose_rf2C (a b c:Set) (f:a->b) (f':b->c)(s'': c->a):
  is_right_inverseC (f' \o f) s'' -> injectiveC f'->
  is_right_inverseC f (s'' \o f').
```

Now the same results, in Bourbaki notations.

```
Theorem compose_fi  f f':
  injection f -> injection f' -> f' \coP f -> injection (f' \co f).
Lemma inj_compose1 f f':
  injection f -> injection f' -> source f' = target f ->  injection (f'\co f).
Theorem compose_fs f f':
  surjection f -> surjection f' -> f' \coP f -> surjection (f' \co f).
Lemma compose_fb f f':
  bijection f -> bijection f' -> f' \coP f -> bijection (f' \co f).
Lemma left_inverse_composable f f' r r':  f' \coP  f ->
  is_left_inverse f' r' -> is_left_inverse f r ->  r \coP r'.
Lemma right_inverse_composable f f' s s':  f' \coP f ->
  is_right_inverse f' s' -> is_right_inverse f s -> s \coP s'.
Theorem left_inverse_compose  f f' r r': f' \coP f ->
  is_left_inverse f' r' -> is_left_inverse f r ->
  is_left_inverse  (f' \co f) (r \co r').
Theorem right_inverse_compose f f' s s': f' \coP f ->
  is_right_inverse f' s' -> is_right_inverse f s ->
  is_right_inverse (f' \co f) (s \co s').
Theorem right_compose_fi f f':
  f' \coP f -> injection (f' \co f) -> injection f.
Theorem left_compose_fs f f':
  f' \coP f -> surjection (f' \co f) -> surjection f'.
Theorem left_inv_compose_rf f f' r'':
  f' \coP f -> is_left_inverse (f' \co f) r'' ->
  is_left_inverse f (r'' \co f').
Theorem right_inv_compose_rf f f' s'':
  f' \coP f -> is_right_inverse (f' \co f) s'' ->
  is_right_inverse f' (f \co s'').
Theorem left_compose_fs2 f f':
  f' \coP f-> surjection (f' \co f) -> injection f' -> surjection f.
Theorem left_compose_fi2 f f':
  f' \coP f -> injection (f' \co f) -> surjection f -> injection f'.
Theorem left_inv_compose_rf2 f f' r'':
  f' \coP f -> is_left_inverse (f' \co f) r'' -> surjection f ->
  is_left_inverse f' (f \co r'').
Theorem right_inv_compose_rf2  f f' s'':
  f' \coP f -> is_right_inverse (f' \co f) s'' -> injection f'->
  is_right_inverse f (s'' \co f').
```

If $f \circ g$ is a bijection, one of f or g is a bijection, so is the other.

```
Lemma right_compose_fb f f':
  f' \coP f -> bijection (f' \co f) -> bijection f' -> bijection f.
Lemma left_compose_fb f f':
  f' \coP f -> bijection (f' \co f) -> bijection f -> bijection f'.
```



(decomposition, Prop 9)

Proposition 9 [2, p. 88] is implemented in the next lemmas. If f and g have the same source and if g is surjective, then the condition $g(x) = g(y) \implies f(x) = f(y)$ is a necessary and sufficient condition for the existence of h with $f = h \circ g$. Such a mapping is then unique and is $f \circ s$, for any right inverse of g.

```
Lemma exists_left_composableC (a b c:Set) (f:a->b)(g:a->c):
  surjectiveC g ->
  ((exists h, h \o g =1 f) <->
  (forall (x y:a), g x = g y -> f x = f y)).
Theorem exists_left_composable f g:
  function f -> surjection g -> source f = source g ->
  ((exists h:E, h \coP g /\ h \co g = f) <->
   (forall (x y:E), inc x (source g) -> inc y (source g) ->
     Vf g x = Vf g y -> Vf f x = Vf f y)).

Lemma exists_left_composable_auxC (a b c:Set) (f:a->b) (g:a-> c) s h:
  is_right_inverseC g s ->
  h \o g =1 f -> h =1 f \o s.
Theorem exists_left_composable_aux f g s h:
  function f ->  source f = source g ->
  is_right_inverse g s -> h \coP g ->
  h \co g = f -> h = f \co s.

Lemma exists_unique_left_composableC (a b c:Set) (f:a->b)(g:a->c) h h':
  surjectiveC g -> h \o g =1 f -> h' \o g =1 f ->
  h =1 h'.
Theorem exists_unique_left_composable f g h h':
  function f -> surjection g -> source f = source g ->
  h \coP g -> h' \coP g ->
  h \co g = f ->  h' \co g = f -> h = h'.

Lemma left_composable_valueC (a b c:Set) (f:a->b)(g:a->c) s h:
  surjectiveC g ->  (forall (x y:a), g x = g y -> f x = f y) ->
  is_right_inverseC g s  ->  h =1 f \o s ->
  h \o g =1 f.
Theorem left_composable_value f g s h:
  function f -> surjection g -> source f = source g ->
  (forall x y, inc x (source g) -> inc y (source g) ->
    Vf g x = Vf g y -> Vf f x = Vf f y) ->
  is_right_inverse g s  ->  h = f \co s -> h \co g = f.
```

Second part of Proposition 9. We assume that f and g have the same target, g is injective; the condition range$(f) \subset$ range(g) is a necessary and sufficient condition for the existence of h with $f = g \circ h$, such a mapping is then unique and is $r \circ f$, for any left inverse of g.

```
Lemma exists_right_composable_auxC (a b c:Set) (f:a->b) (g:c->b) h r:
   is_left_inverseC g r  -> g \o  h =1 f
  -> h =1  r \o f.
Theorem exists_right_composable_aux f g h r:
  function f -> target f = target g ->
  is_left_inverse g r  -> g \coP h -> g \co h = f
  -> h = r \co f.

Lemma exists_right_composable_uniqueC (a b c:Set) (f:a->b)(g:c->b) h h':
  injectiveC g -> g \o h =1 f -> g \o h' =1 f -> h =1 h'.
Theorem exists_right_composable_unique f g h h':
  function f -> injection g -> target f = target g ->
  g \coP h ->  g \coP h' ->
  g \co h = f -> g \co h' = f -> h = h'.
Lemma exists_right_composableC (a b c:Set) (f:a->b) (g:c->b):
  injectiveC g ->
  ((exists h, g \o h =1 f) <->  (forall u, exists v, g v = f u)).
Theorem exists_right_composable f g:
  function f -> injection g -> target f = target g ->
  ((exists h, g \coP h /\ g \co h = f) <->
   (sub (range (graph f)) (range (graph g)))).

Lemma right_composable_valueC (a b c:Set) (f:a->b) (g:c->b) r h:
  injectiveC g -> is_left_inverseC r g -> (forall u, exists v, g v = f u) ->
  h =1 r \o f ->  g \o h =1 f.
Theorem right_composable_value f g r h:
  function f -> injection g -> target g = target f ->
  is_left_inverse g r ->
  (sub (range (graph f)) (range (graph g))) ->
  h = r \co f -> g \co  = f.
```

3.9 Functions of two arguments

For Bourbaki, a *function of two arguments* is a function f whose domain is a set of pairs. Let D be its domain, and assume $D \subset A \times B$. If $z \in D$, there exist $x \in A$ and $y \in B$ such that $z = (x, y)$. Instead of $f((x, y))$, one writes $f(x, y)$. For any y, we may consider the set A_y of all $x \in A$, such that $(x, y) \in D$, and the mapping $x \mapsto f((x, y))$ with source A_y, that has the same target as f. This is called the *partial mapping defined by f, with respect to the value y of the second argument.* It is sometimes denoted f_y of $f(\cdot, y)$. Similarly, f_x is defined by $f_x(y) = f((x, y))$ (whenever $(x, y) \in D$).

In a future chapter, we shall define the set of functions $X \to Y$ and denote it by $\mathscr{F}(X;Y)$. We shall show that $\mathscr{F}(A \times B, C)$ is canonically isomorphic to $\mathscr{F}(A, \mathscr{F}(B;C))$; in Coq, the two types $A \times B \to C$ and $A \to (B \to C)$ are also isomorphic. In this section, we just assume that the source of f is a subset of a product (so that A_y may depend on y).

```
Definition partial_fun2 f y :=
  Lf(fun x=> Vf f (J x y)) (im_of_singleton(inverse_graph (source f)) y)
   (target f).
Definition partial_fun1 f x :=
  Lf(fun y=> Vf f (J x y))(im_of_singleton (source f) x) (target f).

Section Funtion_two_args.
Variable f:Set.
```

```
Hypothesis ff: function f.
Hypothesis sgf: sgraph (source f).

Lemma partial_fun1_axioms x:
  lf_axiom (fun y=> Vf f (J x y))(im_of_singleton (source f) x) (target f).
Lemma partial_fun1_f f x: function (partial_fun1 f x).
Lemma partial_fun1_V x y:
  inc (J x y) (source f) -> Vf (partial_fun1 f x) y = Vf f (J x y).
Lemma artial_fun2_axioms y:
 lf_axiom (fun x=> Vf f (J x y))(im_of_singleton(inverse_graph (source f)) y)
  (target f).
Lemma partial_fun2_f  y:
  function (partial_fun2 f y).
Lemma partial_fun2_V x y:
  inc (J x y) (source f) -> Vf (partial_fun2 f y) x = Vf f (J x y).
End Funtion_two_args.
```

An example of function of two arguments is the function obtained from two functions u and v by associating to (x, y) the pair $(u(x), v(y))$.

```
Definition ext_to_prod u v :=
  Lf(fun z=> J (Vf (P z))(Vf v (Q z)))
  ((source u) \times (source v))
  ((target u)\times (target v)).

Section Ext_to_prod.
Variables u v: Set.
Hypothesis (fu: function u) (fv: function v).

Lemma ext_to_prod_f:  function (ext_to_prod u v).
Lemma ext_to_prod_s: source (ext_to_prod u v) = source u \times source v.
Lemma ext_to_prod_V a b:
  inc a (source u) -> inc b (source v)->
  Vf (ext_to_prod u v) (J a b) =  J (Vf u a) (Vf v b).
Lemma ext_to_prod_V2 u v c:
  inc c ((source u) \times (source v)) ->
  Vf (ext_to_prod u v) c = J (Vf u (P c)) (Vf v (Q c)).
Lemma ext_to_prod_r u v:
  range (graph (ext_to_prod u v)) =
    (range (graph u)) \times (range (graph v)).
End Ext_to_prod.
```

$$\begin{array}{ccccc} a & \xleftarrow{\text{pr1C}} & a\times b & \xrightarrow{\text{pr2C}} & b \\ \downarrow{\scriptstyle u} & & \downarrow{\scriptstyle u\times v} & & \downarrow{\scriptstyle v} \\ a' & \xrightarrow[\text{J}]{} & a'\times b' & \xleftarrow[\text{J}]{} & b' \end{array} \qquad \text{(prod extension)}$$

We can consider the product of two COQ functions. We first define the projections from $a \times b$ to a and b and the inverse function. In the diagram above, this inverse function corresponds to the two arrows named J. In other words, if z is the pair (x, y), we have $\mathrm{P}(z) = x$ and $\mathrm{Q}(z) = y$. To say that J is the inverse means that J applied to x and y gives z. This function takes two arguments (its type is `Set` → `Set` → `Set`) but is not a function of two arguments (its type is not `Set` × `Set` → `Set`).

```
Lemma ext_to_prod_propP: forall a a' (x: a \times a'),  inc (P (Ro x)) a.
Lemma ext_to_prod_propQ: forall a a' (x: a \times a'),  inc (Q (Ro x)) a'.
Lemma ext_to_prod_propJ: forall (b b':Set) (x:b)(x':b'),
  inc (J (Ro x)(Ro x'))  (b \times b').


Definition pr1C a b:= fun x:a \times b => Bo(ext_to_prod_propP x).
Definition pr2C a b:= fun x:a \times b => Bo(ext_to_prod_propQ x).
Definition pairC a b:= fun (x:a)(y:b) => Bo(ext_to_prod_propJ x y).

Definition ext_to_prodC (a b a' b':Set) (u:a->a')(v:b->b') :=
  fun x => pairC (u (pr1C x)) (v (pr2C x)).

Lemma prC_prop(a b:Set) (x: a \times b):
  Ro x = J (Ro (pr1C x)) (Ro (pr2C x)).
Lemma pr1C_prop (a b:Set) (x:a \times b):  Ro (pr1C x) =  P (Ro x).
Lemma pr2C_prop (a b:Set) (x: a \times b):  Ro (pr2C x) =  Q (Ro x).
Lemma prJ_prop (a b:Set) (x:a)(y:b):   Ro(pairC x y) = J (Ro x) (Ro y).
Lemma prJ_recov (a b:Set) (x:a \times b): pairC (pr1C x) (pr2C x) = x.
Lemma ext_to_prod_prop:
   (a b a' b':Set) (u:a->a')(v:b->b') (x:a)(x':b):
  J(Ro (u x)) (Ro (v x')) = Ro(ext_to_prodC u v (pairC x x')).
```

If both functions are injective, surjective or bijective, so is the product. The inverse is the product of the inverses. It is compatible with composition.

```
Lemma ext_to_prod_injective u v:
  injection u -> injection  v -> injection (ext_to_prod u v).
Lemma ext_to_prod_surjective u v:
  surjection u -> surjection  v-> surjection (ext_to_prod u v).
Lemma ext_to_prod_bijective u v:
  bijection u -> bijection  v-> bijection (ext_to_prod u v).
Lemma ext_to_prod_inverse u v:
  bijection u -> bijection  v->
  inverse_fun (ext_to_prod u v) =
  ext_to_prod (inverse_fun u)(inverse_fun v).
Lemma composable_ext_to_prod2 u v u' v':
  u' \coP u -> v' \coP v -> (ext_to_prod u' v') \coP (ext_to_prod u v).
Lemma compose_ext_to_prod2 u v u' v':
  u' \coP u -> v' \coP v ->
  (ext_to_prod u' v') \co (ext_to_prod u v) =
  ext_to_prod (u' \co  u)(v' \co v).
```

Same lemmas for COQ functions.

```
Lemma injective_ext_to_prod2C (a b a' b':Set) (u:a->a')(v:b->b'):
  injectiveC u -> injectiveC v -> injectiveC (ext_to_prodC u v).
Lemma surjective_ext_to_prod2C (a b a' b':Set) (u:a->a')(v:b->b'):
  surjectiveC u -> surjectiveC v -> surjectiveC (ext_to_prodC u v).
Lemma bijective_ext_to_prod2C (a b a' b':Set) (u:a->a')(v:b->b'):
  bijectiveC u -> bijectiveC v -> bijectiveC (ext_to_prodC u v).
Lemma compose_ext_to_prod2C (a b c a' b' c':Set) (u:b-> c)(v:a->b)
  (u':b'-> c')(v':a'->b'):
  (ext_to_prodC u u') \o (ext_to_prodC v v') =1
  ext_to_prodC (u \o v)(u' \o v').
Lemma inverse_ext_to_prod2C (a b a' b':Set) (u:a->a')(v:b->b')
```

```
  (Hu: bijectiveC u)(Hv:bijectiveC  v):
  inverseC (bijective_ext_to_prod2C Hu Hv) =1
  ext_to_prodC (inverseC Hu)(inverseC Hv).
```

¶ Canonical decomposition of a function, version one. Let f be a function from A to B, and C its range. Then f is the composition of the restriction of f to its range, and the canonical injection from the range to the target. The first function satisfies $g(x) = f(x)$; the second satisfies $i(x) = x$.

```
Lemma image_of_fun_range f: function f ->
   image_of_fun f = range (graph f).
Lemma canonical_decomposition1 f
  (g:= restriction_to_image f)
  (i := canonical_injection (range (graph f)) (target f)):
  function f ->
  [/\ i \coP g, f = i \co g, injection i, surjection g
  (& injection f -> bijection g )].
```

In the case of COQ functions, we replace the range of the graph by the image.

```
Definition imageC (a b:Set) (f:a->b) := IM (fun u:a => Ro (f u)).
Lemma imageC_inc (a b:Set) (f:a->b) (x:a): inc (Ro (f x)) (imageC f).
Lemma imageC_exists (a b:Set) (f:a->b) x:
  inc x (imageC f) -> exists y:a,  x = Ro (f y).
Lemma sub_image_targetC (a b:Set) (f:a->b): sub (imageC f) b.

Definition restriction_to_imageC a b (f:a->b) :=
  fun x:a => Bo (imageC_inc f x).

Lemma restriction_to_imageC_pr (a b:Set) (f:a->b) (x:a):
  Ro(restriction_to_imageC f x) = Ro (f x).
Lemma canonical_decomposition1C (a b:Set) (f:a->b)
  (g:a-> imageC f)(i:imageC f ->b):
  g = restriction_to_imageC f ->
  i = inclusionC (sub_image_target (f:=f))  ->
  [/\ injectiveC i , surjectiveC g &
  (injectiveC f -> bijectiveC g)].
```

Chapter 4

Union and intersection of a family of sets

Bourbaki gives the following definitions:

Definition 1. *Let* $(X_\iota)_{\iota\in I}$ *be a family of sets (*resp. *a family of subsets of a set* E*). The set* D_1*, that is to say [...], is called the union of the family and denoted by* $\bigcup_{\iota\in I} X_\iota$.

Definition 2. *Let* $(X_\iota)_{\iota\in I}$ *be a family of sets whose index set* I *is not empty. The set* D_2*, that is to say [...], is called the intersection of the family and denoted by* $\bigcap_{\iota\in I} X_\iota$.

Definition 3. *Let* $(X_\iota)_{\iota\in I}$ *be a family of subsets of a set* E*. The set* D_3*, in other words [...], is called the intersection of the family and denoted by* $\bigcap_{\iota\in I} X_\iota$.

The definitions use the following sets

$$D_1 := \{x \mid (\exists\iota)(\iota \in I \text{ and } x \in X_\iota)\},$$

$$D_2 := \{x \mid (\forall\iota)((\iota \in I) \implies (x \in X_\iota))\},$$

$$D_3 := \{x \mid x \in E \text{ and } (\forall\iota)((\iota \in I) \implies (x \in X_\iota))\},$$

that Bourbaki explicits in plain English as, for instance for D_1: "the set of all x which belong to at least one set of the family $(X_\iota)_{\iota\in I}$". Let $P_1(x)$ and $P_2(x)$ be the predicates $(\exists\iota)(\iota \in I \text{ and } x \in X_\iota)$ and $(\forall\iota)((\iota \in I) \implies (x \in X_\iota))$, so that D_3 is the set of all $x \in E$ such that $P_2(x)$ holds. The two quantities D_1 and D_2 are sets provided that P_1 and P_2 are collectivizing, which is a non-trivial property.

Let's recall that a "family" is a functional graph. If G is such a graph, and x is in its domain, there is a unique y such that $(x, y) \in G$. This is called the value of G at x, and generally denoted by $G(x)$. The domain of the family is also called the "index set". The notation $(X_\iota)_{\iota\in I}$ has to be understood as: I is the domain of the family, and ι is just a dummy variable. Similarly in $\bigcup_{\iota\in I} X_\iota$, the variable ι is a dummy one. By abuse of language, the union may also be written as $\bigcup X_\iota$. By another abuse of language, $\bigcup_{\iota\in J} X_\iota$ denote the union of the restriction of X to J.

In order to "help the intuitive interpretation", Bourbaki uses the phrase "family of sets" (this is an obvious pleonasm), and he writes G_x instead of $G(x)$ (for instance in D_1). A "family of subsets of a set E", is a function $\Gamma = (G, A, B)$ such that any element of B is a subset of E. Thus G is a functional graph, A its index set, $G(x) \in B$ whenever $x \in A$, so that $G(x) \subset E$ whenever $x \in A$. In Definition 3, (as well as in Definition 1, "resp." part), the notation $(X_\iota)_{\iota\in I}$ has to be

understood as "we have some function with graph X, source I, target $\mathfrak{G}$" and ι is a dummy variable. In D_1 and D_3, the quantity X_ι is the value of the function at ι.

Existence of union follows from Axiom Scheme S8 that reads: "Let $\boldsymbol{R}$ be a relation, let $\boldsymbol{x}$ and $\boldsymbol{y}$ be distinct letters, and let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be letters distinct from $\boldsymbol{x}$ and $\boldsymbol{y}$ which do not appear in $\boldsymbol{R}$. Then the relation

$$(\forall \boldsymbol{y})(\exists \boldsymbol{X})(\forall \boldsymbol{x})(\boldsymbol{R} \implies (\boldsymbol{x} \in \boldsymbol{X})) \implies (\forall \boldsymbol{Y})\,\mathrm{Coll}_{\boldsymbol{x}}((\exists \boldsymbol{y})((\boldsymbol{y} \in \boldsymbol{Y}) \text{ and } \boldsymbol{R}))$$

is an axiom."

Take $\boldsymbol{R} := x \in X_\iota$, $\boldsymbol{x} := x$, $\boldsymbol{y} := \iota$, $\boldsymbol{X} := Z$, and $\boldsymbol{Y} := I$. We get

$$(\forall \iota)(\exists Z)(\forall x)((x \in X_\iota) \implies (x \in Z)) \implies (\forall I)\,\mathrm{Coll}_x((\exists \iota)((\iota \in I) \text{ and } (x \in X_\iota)).$$

The assumption is true, since it suffices to take $Z = X_\iota$. Thus the conclusion holds, and we get $(\forall I)\,\mathrm{Coll}_x(P_1(x))$. This implies $\mathrm{Coll}_x(P_1(x))$, thus the existence of the set D_1.

Note that Bourbaki proves the following

$$(\forall \iota)(\exists Z)(\forall x)((\iota \in I \text{ and } x \in X_\iota) \implies (x \in Z)).$$

and applies S8. There is a subtlety here. If $\boldsymbol{R}$ is relation that appears here, it contains I, so that we cannot use $\boldsymbol{Y} := I$ anymore. Taking Y instead gives

$$(\forall Y)\mathrm{Coll}_x(\exists \iota)((\iota \in Y) \text{ and } (\iota \in I \text{ and } x \in X_\iota))$$

We can now take $Y = I$ and simplify.

Consider now a non-empty family. Fix some $\alpha \in I$. Then $P_2(x)$ implies $x \in X_\alpha$, so that D_2 is just the set of all x in X_α that satisfy P_2, so that Definition 2 makes sense.

Consider now a family of subsets of E; in the case of intersection, we shall assume the index set non-empty. Then both the union and intersection are subsets of E. They are independent of E and of the target of the function. In particular $D_2 = D_3$.

Assume now I empty. In this case $P_2(x)$ holds for every x and $D_3 = E$. However, P_2 is not collectivizing, and D_2 is not a set, so that Definition 2 cannot be applied. In summary: in the case of union, there is no difference between the two definitions, and in the case of intersection, there is a difference only if the index set is empty. In this case, we do not follow Bourbaki. In the case where the index set is empty, we shall define the intersection to be empty.

4.1 Definition of the union and intersection of a family of sets

We give four definitions of union and intersection. We have already defined '`uniont f`', where f is of type $I \to$ `Set`, and I is a set; it is the set of all x such that $x \in f(z)$ for some z of type I. If g is of type `Set` $\to$ `Set` and I a set, composing g with $\mathscr{R}_I$ yields a function of type $I \to$ `Set`. The union (in the previous sense) is '`unionf I g`'; this is the set of all x such that there exists $i \in I$ such that $x \in g(i)$. If G is any functional graph, '`unionb G`' is the union (in the previous sense) of the evaluation function of G on its domain; this is the Bourbaki definition. Finally '`union X`' has already been defined to be union (in the previous sense) of the identity graph of X.

We define similarly '`intersectiont f`', '`intersectionf I g`', '`intersectionb G`' and '`intersection X`', using D_3, where E is the union. Note that intersection is defined for

empty families, but some lemmas apply only when the family is non-empty. Note also that few theorems assume that G is functional graph. This is because of the following: let G any set, I the set of all $\mathrm{pr}_1 x$, where $x \in \mathrm{G}$, and f the function that associates to each $y \in \mathrm{I}$ the quantity $\mathscr{V}_{\mathrm{G}}(x)$ where $\mathrm{pr}_1 x = y$. Then the union of G is the union of f over I. If G is a functional graph, then for any $y \in \mathrm{I}$, there is a unique z such that $(y,z) \in \mathrm{G}$ and $z = f(y)$.

```
Definition intersectiont (I:Set) (f : I->Set):=
  Zo (uniont f) (fun y => forall z : I, inc y (f z)).

Definition unionf (x:Set)(f: fterm) := uniont (fun a:x => f (Ro a)).
Definition unionb g := unionf (domain g)(Vg g).
Definition intersectionf (x:Set)(f: fterm):= intersectiont(fun a:x => f (Ro a)).
Definition intersectionb g :=  intersectionf (domain g) (Vg g).
```

We have now a bunch of lemmas that show how to use these definitions.

```
Lemma setUf_P x i f:
  inc x (unionf i f) <->  exists2 y, inc y i & inc x (f y).
Lemma setUb_P x f:
  inc x (unionb f) <-> exists2 y, inc y (domain f) & inc x (V f y).
Lemma setUb_P1 x a f:
   inc x (unionb (Lg a f)) <-> exists2 y, inc y a & inc x (f y).
Lemma setUt_i (In:Set) (f : In->Set) y x:
  inc x (f y) -> inc x (uniont f).
Lemma setUf_i x y i f:
  inc y i -> inc x (f y) -> inc x (unionf i f).
Lemma setUb_i x y f:
  inc y (domain f) -> inc x (Vg f y) ->  inc x (unionb f).
Lemma setUf_hi x i f:
  inc x (unionf i f) ->  exists2 y, inc y i & inc x (f y).
Lemma setUb_hi x f:
  inc x (unionb f) -> exists2 y, inc y (domain f) & inc x (Vg f y).
```

Trivial cases where the domain is empty.

```
Lemma setUt_0 (I:Set) (f:I-> Set): I = emptyset ->  uniont f = emptyset.
Lemma setUf_0 f: unionf emptyset f = emptyset.
Lemma setUb_0: unionb emptyset = emptyset.
Lemma setIt_0 (I:Set) (f:I-> Set):  I = emptyset -> intersectiont f = emptyset.
Lemma setIf_0 f: intersectionf emptyset f = emptyset.
Lemma setIb_0: intersectionb emptyset = emptyset.
```

Some lemmas for the intersection. All these lemmas are obvious from the definitions and the link between $\mathscr{R}$ and $\mathscr{B}$.

```
Lemma setIt_P (I:Set) (f:I-> Set): nonempty I -> forall x,
  (inc x (intersectiont f) <-> (forall j, inc x (f j))).
Lemma setIf_P I f:  nonempty I -> forall x,
  (inc x (intersectionf I f) <-> (forall j, inc j I -> inc x (f j))).
Lemma setIb_P g: nonempty g -> forall x,
  (inc x (intersectionb g) <-> (forall i, inc i (domain g) -> inc x (Vg g i))).
Lemma setI_P y: nonempty y -> forall x,
  (inc x (intersection y) <-> (forall i, inc i y -> inc x i)).
```

```
Lemma setIt_i (I:Set) (f:I-> Set) x: nonempty I ->
  (forall j, inc x (f j)) -> inc x (intersectiont f).
Lemma setIt_hi (I:Set) (f:I-> Set) x j:
  inc x (intersectiont f) -> inc x (f j).
Lemma setIf_i I f x: nonempty I ->
  (forall j, inc j I -> inc x (f j)) -> inc x (intersectionf I f).
Lemma setIf_hi I f x j:
  inc x (intersectionf I f) -> inc j I -> inc x (f j).
Lemma setIb_i g x:  nonempty g ->
  (forall i, inc i (domain g) -> inc x (Vg g i)) -> inc x (intersectionb g).
Lemma setIb_hi g x i:
  inc x (intersectionb g) -> inc i (domain g) -> inc x (Vg g i).
```

These lemmas are trivial consequences of the previous ones. They explain when two unions or intersections are equal.

```
Lemma setUt_exten (I:Set) (f: I-> Set) (f':I->Set):
 f =1 f' -> uniont f = uniont f'.
Lemma setUf_exten sf f f':
  {inc sf,  f =1 f'} -> unionf sf f = unionf sf f'.
Lemma setIt_exten (I:Set) (f f':I-> Set):
   f =1 f' - -> (intersectiont f) = (intersectiont f').
Lemma setIf_exten I f f':  {inc I,  f =1 f'} ->
  intersectionf I f  = intersectionf I f'.
```

These trivial lemmas say that for all j, $X_j \subset \bigcup X_i$ and $\bigcap X_i \subset X_j$. On the other hand, if for all i, we have $A \subset X_i \subset B$, then $A \subset \bigcup X_i \subset B$ and $A \subset \bigcap X_i \subset B$. Note that for two of these inclusions, the index set must be nonempty.

```
Lemma setUt_s1 (In:Set) (f: I-> Set) i:
  sub (f i) (uniont f).
Lemma setIt_s1 (I:Set) (f: I-> Set) i:
  sub (intersectiont f) (f i).
Lemma setUt_s1 (I:set) (f: I-> Set) x:
  (forall i, sub (f i) x) ->  sub (uniont f) x.
Lemma setI2_s2 (I:Set)(f: I-> Set) x: nonempty I ->
  (forall i, sub x (f i)) ->  sub x (intersectiont f).
Lemma setI_sub2 (I:Set) (f: In-> Set) x:
   (forall i, sub (f i) x) ->  sub (intersectiont f) x.
Lemma setUt_sub2 (I:Set) (f: I-> Set) x:
  nonempty I -> (forall i, sub x (f i)) -> sub x (uniont f).
```

Bourbaki says in Proposition 1 [2, p. 92]: Let f be a function from K onto I, X_ι a family of sets indexed by I. Then the union and the intersection of the family is the union and the intersection of $X_{f(\kappa)}$ over K. Note that I and K are both empty or non-empty.

```
Theorem setUt_rewrite (I K:Set) (f: K->I) (g:I ->Set):
  surjectiveC f ->
  uniont g = uniont (g \o f).
Theorem setIt_rewrite (I K:Set) (f: K->I) (g:I ->Set):
  surjectiveC f ->
  intersectiont g  = intersectiont (g \o f).
```

The Bourbaki statement about union is `setUb_rewrite1`. In the second lemma we just assume that f is a functional graph. In the case of intersection, if g is empty, so is $g \circ f$, and conversely.

```
Lemma setUb_rewrite1 f g:
  function f -> fgraph g -> range (graph f) = domain g ->
  unionb g = unionb (g \cf (graph f)).
Lemma setUb_rewrite f g:
  fgraph f -> range f = domain g ->
  unionb g = unionb (g \cf f).
Lemma setIb_rewrite f g:
  fgraph f -> range f = domain g ->
  intersectionb g = intersectionb (g \cf f).
```

Let f be a constant function and $x \in$ I. Then the intersection and union of f on I is $f(x)$. (Bourbaki uses a strange method: he writes $f = g \circ h$, where h is a surjective function whose target is a singleton; by surjectivity, the union of f is that of g. Now the domain of g is a singleton, and the range of g contains $f(x)$).

```
Lemma seUt_constant (I:Set) (f:I ->Set) (x:I):
  constantp f -> uniont f = f x.
Lemma setIt_constant (I:Set) (f:I ->Set) (x:In):
  constantp f -> intersectiont f = f x.
Lemma setUg_constant f x: constantgp f -> inc x (domain f) ->
  unionb f = Vg f x.
Lemma setIg_constant f x: constantgp f -> inc x (domain f) ->
  intersectionb f = Vg f x.

Lemma setUt_1 (a:Set) (x:a) (f: singleton (Ro x) -> Set):
  uniont f = f (Bo (set1_1 (Ro x))).
Lemma setIt_1: (a:Set)(x:a) (f: singleton (Ro x) -> Set):
  intersectiont f = f (Bo (set1_1 (Ro x))).
Lemma setUf_1 f x: unionf (singleton x) f = f x.
Lemma setIf_1 f x: intersectionf (singleton x) f = f x.
```

¶ Link between these unions and intersections and the old ones: the union of a set of sets X is the union of the identity function on X. If f is a functional graph, its union is also the union of the range.

```
Lemma setU_prop x: union x = unionf x id.
Lemma setUb_alt f: fgraph f -> unionb f = union (range f).
Lemma setUb_identity x: unionb (identity_g x) = union x.
Lemma setI_prop x: intersection x = intersectionf x id.
Lemma setIb_alt f: fgraph f -> intersectionb f = intersection (range f).
Lemma setIb_identity x: intersectionb (identity_g x) = intersection x.
```

4.2 Properties of union and intersection

We first show that the union and intersection of F over I are monotone with respect to the function and index. In the last theorem, one index set must be non-empty.

```
Lemma setUt_S (In:Set) (f g:In->Set):
  (forall i, sub (f i) (g i)) -> sub (uniont f) (uniont g).
Lemma setIt_S (I:Set)(f g:I->Set):
  (forall i, sub (f i) (g i)) -> sub (intersectiont f)(intersectiont g).
Lemma setUf_S2 f: f: {compat (unionf ^~ f) : x y / sub x y }.
Lemma setIf_S f:
  {compat (intersectionf ^~ f) : x y / sub x y /\ nonempty x >-> sub y x}.
```

Proposition 2 [2, p. 93] states associativity of union and intersection. It says:

$$\bigcup_{\iota\in I} X_\iota = \bigcup_{\lambda\in L}\left(\bigcup_{\iota\in J_\lambda} X_\iota\right), \qquad \bigcap_{\iota\in I} X_\iota = \bigcap_{\lambda\in L}\left(\bigcap_{\iota\in J_\lambda} X_\iota\right) \qquad I = \bigcup J_\lambda .$$

In the case of intersection, we require J_λ to be non-empty, since these sets are not taken into account in the LHS, while the corresponding intersection is replaced by the empty set in the RHS.

```
Theorem setUf_A sg f g:
  unionf (unionf sg g) f = unionf sg (fun l => unionf (g l) f).
Theorem setIf_A sg f g:
 (alls sg (fun i => (nonempty (g i)))) ->
  intersectionf (unionf sg g) f
  = intersectionf sg (fun l => intersectionf (g l) f).
```

Proposition 3 [2, p. 94] says that if Γ is a correspondence, then $\Gamma\langle\bigcup X_\iota\rangle = \bigcup\Gamma\langle X_\iota\rangle$ and $\Gamma\langle\bigcap X_\iota\rangle \subset \bigcap\Gamma\langle X_\iota\rangle$. (note: let G be the graph of Γ, so that $\Gamma\langle X\rangle = G\langle X\rangle$; the two formulas are true whatever G). Proposition 4 [2, p. 95] says that we have equality if Γ is the inverse of a function, and, as a consequence, if Γ is an injective function. In fact, we use the canonical decomposition $\Gamma = i\circ g$, where g is the restriction of Γ on its image (hence is bijective), and i is the inclusion map from the image of Γ to its target. Then $\Gamma\langle x\rangle = g^{-1}\langle x\rangle$ for every set x.

```
Theorem dirim_setUt (I:Set) (f:I->Set) g:
  direct_image g (uniont f) = uniont (fun i => direct_image g (f i)).
Theorem dirim_setIt (I:Set) (f:I->Set) g:
  sub (direct_image g (intersectiont f))
  (intersectiont (fun i => direct_image g (f i))).
Theorem iim_fun_setIt (I:Set) (f:I->Set) g:
  function g ->
  (inv_image_by_fun g (intersectiont f)) =
  (intersectiont (fun i => inv_image_by_fun g (f i))).
Lemma inj_image_setIt (I:Set) (f:I->Set) g:
  injection g ->
  (image_by_fun g (intersectiont f))
  = (intersectiont (fun i => image_by_fun g (f i))).
```

4.3 Complements of unions and intersections

Let X_ι be a non-empty family of sets; define $Y_\iota = X\setminus X_\iota$. Then the intersection (resp. union) of the X_ι is the complement in X of the union (resp. intersection) of the Y_ι. This is Proposition 5 [2, p. 96] (Bourbaki assumes, $X_\iota \subset X$, which is not needed).

```
Theorem setCUt2 (I:Set) (f:I-> Set) x: nonempty I ->
  x -s (uniont f) = intersectiont (fun i=> x -s (f i)).
Theorem setCIt2 (I:Set) (f:I-> Set) x:  nonempty I ->
  x -s (intersectiont f) = uniont (fun i=> x -s (f i)).
Lemma setCUf2 sf f x: nonempty sf ->
  x -s (unionf sf f) = intersectionf sf (fun i=> x -s (f i)).
Lemma setCIf2 sf f x: nonempty sf ->
  x -s (intersectionf sf f) = unionf sf (fun i=> x -s (f i)).
```

4.4 Union and intersection of two sets

Bourbaki defines the union and intersection of two sets A and B as the union and intersection of the identity function on the doubleton $\{A, B\}$. This was defined as `union2` and `intersection2`. All results shown here are easy.

```
Lemma setUf2f x y f: unionf (doubleton x y) f = (f x)\cup (f y).
Lemma setIf2f x y f: intersectionf (doubleton x y) f =  (f x) \cap (f y).
Lemma setUf2 x y: unionf (doubleton x y) id = x \cup y.
Lemma setIf2 x y: intersectionf (doubleton x y) id = x \cap y.
```

We have (these results have been proved in a previous chapter).

$$\{x\} \cup \{y\} = \{x, y\}, \qquad x \cup x = x, \qquad x \cap x = x, \qquad x \cap y = y \cap x, \qquad x \cup y = y \cup x.$$

We have:

$$x \cup (y \cup z) = (x \cup y) \cup z, \qquad x \cap (y \cap z) = (x \cap y) \cap z,$$

$$x \cup (y \cap z) = (x \cup y) \cap (x \cup z), \qquad x \cap (y \cup z) = (x \cap y) \cup (x \cap z).$$

We have $x \subset y$ if and only if $x \cup y = y$. We have $x \subset y$ if and only if $x \cap y = x$. We have:

$$z \setminus (x \cup y) = (z \setminus x) \cap (z \setminus y), \qquad z \setminus (x \cap y) = (z \setminus x) \cup (z \setminus y).$$

We have $x \cup (z \setminus x) = z$ and $x \cap (z \setminus x) = \emptyset$. If g is a correspondence, we have $g\langle x \cup y\rangle = g\langle x\rangle \cup g\langle y\rangle$ and $g\langle x \cap y\rangle \subset g\langle x\rangle \cap g\langle y\rangle$. Equality holds if g is an injective function or $g = f^{-1}$ where f is a function.

```
Lemma dirim_setU2 g: {morph (direct_image g): x y / x \cup y}.
Lemma dirim_setI2 g x y:
  sub (direct_image g (x \cap y))
      ((direct_image g x) \cap (direct_image g y)).
Lemma iim_fun_setI2 g: function g ->
  {morph (inv_image_by_fun g): x y / x \cap y}.
Lemma inj_image_setI2 g : injection g ->
  {morph (image_by_fun g): x y / x \cap y}.
```

If f is a function from A into B, then we have $f^{-1}\langle B \setminus x\rangle = A \setminus f^{-1}\langle x\rangle$ and $f\langle A \setminus x\rangle = B \setminus f\langle x\rangle$ if f is a injective with range B (Proposition 6, [2, p. 98]).

```
Lemma iim_fun_C1 f: function f ->
 {when  eq^~ (target f) & sub^~ (target f),
   {morph inv_image_by_fun f : a b / a -s b}}.
Lemma inj_image_C f: injection f ->
 {when eq^~ (source f) & sub^~ (source f),
   {morph image_by_fun f : a b / a -s b}}.
```

4.5 Coverings

A *covering* of a set X is a family X_ι whose union contains X. By extension, a set whose union contains X is also called a covering.

```
Definition covering f x :=  fgraph f /\ sub x (unionb f).
Definition covering_s f x :=  sub x (union f).

Lemma covering_P f x:  fgraph f ->
  (covering f x <-> covering_s (range f) x).
```

We say that a covering $(Y_\kappa)_{\kappa\in K}$ refines $(X_\iota)_{\iota\in I}$ if for all κ there is ι such that $Y_\kappa \subset X_\iota$. We sometimes say that Y is finer than X, or that X is coarser than Y. This definition will be extended to set coverings: the definition '`coarser_cs y y'`' says that the set of sets y' refines y. In other words, for all $a \in y'$ there is $b \in y$ such that $a \subset b$. We will show that this is an order on the set of all partitions.

```
Definition coarser_cg f g :=
  [/\ fgraph f, fgraph g &
  forall j, inc j (domain g)
    -> exists2 i, inc i (domain f) &  sub (Vg g j) (Vg f i)].
Definition coarser_cs y y' :=
  forall a, inc a y' -> exists2 b, inc b y & sub a b.

Lemma coarser_cP f g: fgraph f -> fgraph g ->
  (coarser_cg f g <-> coarser_cs (range f) (range g)).
Lemma sub_covering f I x (g := restr f I):
  (sub I (domain f)) -> covering f x -> covering g x ->
  coarser_cg f g.
```

Given two families X_ι and Y_κ, we can consider the family $X_\iota \cap Y_\kappa$. Given two sets of sets X and Y, we can consider the set of elements of the form $a \cap b$ for $a \in X$ and $b \in Y$. Hence, given two coverings X_ι and Y_κ of Z we find a covering $i(X_\iota, Y_\kappa)$ of Z that refines X_ι and Y_κ, this is the supremum for the coarser ordering (ordering are defined in the second part of this report).

```
Definition intersection_covering f g :=
  Lg ((domain f) \times (domain g))
    (fun z => (Vg f (P z)) \cap (Vg g (Q z))).
Definition intersection_covering2 x y:=
  range (intersection_covering (identity_g x) (identity_g y)).

Lemma setI_covering2_P x y z:
  inc z (intersection_covering2 x y) <->
  exists a b, [/\ inc a x, inc b y,  a \cap b = z ].
Lemma setI_covering E: {compat intersection_covering : x & / covering x E}.
Lemma setI_coarser_cl f g x:
  covering f x -> covering g  x ->
  coarser_cg f (intersection_covering f g).
Lemma setI_coarser_cr f g x:
  covering f x -> covering g  x ->
  coarser_cg g (intersection_covering f g).
Lemma setI_coarser_clr h x:
  covering h x -> {when covering ^~ x &,
      {compat intersection_covering : f & / coarser_cg f h}}.
```

We show here the equivalent properties for sets of sets. Essentially, we prove that $i(x, y)$ is the least upper bound for the order defined by `coarser_cs` (which is defined on the set of partitions, as will be seen later).

```
Lemma setI_covering2 E:
  {compat intersection_covering2 : x & / covering_s x E}.
Lemma setI_coarser2_cl f g x:
  covering_s f x -> covering_s g x ->
  coarser_cs f (intersection_covering2 f g).
Lemma setI_coarser2_cr f g x:
  covering_s f x -> covering_s g x ->
  coarser_cs g (intersection_covering2 f g).
Lemma setI_coarser2_clr h x:
  covering_s h x -> {when covering_s ^~ x &,
      {compat intersection_covering2 : f & / coarser_cs f h}}.
```

If X_ι is a covering and g a function, then the family of sets $g^{-1}\langle X_\iota\rangle$ is a covering; if g is surjective, then $g\langle X_\iota\rangle$ is a covering.

```
Lemma image_of_covering f g:
  surjection g -> covering  f (source g)
  -> covering (Lg (domain f) (fun w => image_by_fun g (Vg f w))) (target g).
Lemma inv_image_of_covering f g:
  function g -> covering f (target g)
  -> covering (Lg (domain f) (fun w => inv_image_by_fun g (Vg f w))) (source g).
Lemma product_of_covering f g x y:
  covering f x -> covering g y ->
  covering (Lg ((domain f) \times (domain g))
        (fun z => (V f (P z)) \times (Vg g (Q z))))
  (x \times y).
```

Proposition 7 [2, p. 99] says that if X_ι is a covering of E, then two functions that agree on each X_ι agree on E. Moreover, assume that f_ι is a function defined on X_ι (with target T_ι). Assume that f_ι and f_κ agree on $X_\iota \cap X_\kappa$. There is a unique function f defined on E, that agrees with f_ι on X_ι, whose target is the union of the T_ι. We prove uniqueness only in the case where all the T_ι are equal to a same T.

```
Definition function_prop_sub f s t:=
  [/\ function f, source f = s, sub (target f) t].
Definition common_ext f h t:=
  triple (unionb f) t (unionb (L (domain f) (fun i => (graph (h i))))).

Lemma agrees_on_covering f x g g':
  covering f x -> function g -> function g' ->
  source g = x -> source g' = x ->
  (forall i, inc i (domain f) ->  agrees_on (x \cap (Vg f i)) g g') ->
  agrees_on x g g'.
Lemma extension_covering f t h
  (d:= domain f) (g := common_ext f h t) :
  (forall i, inc i d -> function_prop (h i) (Vg f i) t) ->
  (forall i j, inc i d -> inc j d ->
    agrees_on ((Vg f i) \cap (Vg f j)) (h i) (h j)) ->
  [/\ function_prop g (unionb f) t /\
    graph g = (unionb (Lg d (fun i => (graph (h i))))),
    range (graph g) = unionb  (Lg d (fun i => (range (graph (h i))))),
    (forall i, inc i d -> agrees_on (Vg f i) g (h i))].
Lemma extension_covering_thm f t h (d:= domain f):
  (forall i, inc i d -> function_prop (h i) (V f i) t) ->
  (forall i j, inc i d -> inc j d ->
```

```
    agrees_on ((Vg f i) \cap (Vg f j)) (h i) (h j)) ->
  fgraph f ->
  exists! g,(function_prop g (unionb f) t /\
    (forall i, inc i d -> agrees_on (Vg f i) g (h i))).
```

4.6 Partitions

Definition 7 in Bourbaki [2, p. 100] is: a *partition* of a set E is a family of *non-empty* mutually disjoints subsets of E which covers E; the phrase non-empty is missing in the French version. We consider the strong and weak versions, as well as version where a family is replaced by a set of sets.

```
Definition nonempty_fam f := allf f nonempty.

Definition mutually_disjoint f :=
  (forall i j, inc i (domain f) -> inc j (domain f) ->
    i = j \/ (disjoint (Vg f i) (Vg f j))).
Definition partition_w y x:=
  (union y = x) /\
  (forall a b, inc a y -> inc b y -> disjointVeq a b).
Definition partition_s y x:=
  partition_w y x /\ (alls y nonempty).
Definition partition_w_fam f x:=
  [/\ fgraph f, mutually_disjoint f & unionb f = x].
Definition partition_s_fam_s f x:=
  partition_w_fam f x /\ nonempty_fam f.
```

We list below some properties of partitions.

```
Lemma mutually_disjoint_prop f:
  (forall i j y, inc i (domain f) -> inc j (domain f)  ->
    inc y (Vg f i) -> inc y (Vg f j) -> i = j) ->
  mutually_disjoint f.
Lemma mutually_disjoint_prop2 x f:
  (forall i j y, inc i x -> inc j x  ->
    inc y (f i) -> inc y (f j) -> i = j) ->
  mutually_disjoint (Lg x f).
Lemma mutually_disjoint_prop1 f: function f ->
  (forall i j y, inc i (source f) -> inc j (source f)  ->
    inc y (Vf f i) -> inc y (Vf f j) -> i = j) ->
  mutually_disjoint (graph f).

Lemma partition_same y x:
  partition_w y x -> partition_w_fam (identity_g y) x.
Lemma partition_same2 y x:
  partition_fam y x -> partition_s (range y) x.
Lemma partitions_is_covering y x:
  partition_w y x -> covering_s y x.
Lemma partition_fam_is_covering y x:
  partition_w_fam y x -> covering y x.
```

If $(X_\iota)_{\iota\in I}$ is a partition of E, each element of E is in a unique X_ι. Thus, we have a function $f : E \to I$, such that $x \in X_{f(x)}$.

```
Definition cover_at f y := select (fun i =>  inc y (Vg f i)) (domain f).

Lemma cover_at_in f x y (i := cover_at f y):
  partition_w_fam f x -> inc y x ->
  (inc y (Vg f i) /\ inc i (domain f)).
Lemma cover_at_pr f x y i:
   partition_w_fam f x -> inc i (domain f) -> inc y (Vg f i) ->
   cover_at f y = i.
Lemma same_cover_at f x y z (i := cover_at f y):
  partition_w_fam f x -> inc y x -> inc z (Vg f i) ->  cover_at f z = i.
```

We show here that "coarser" is an ordering on the set of partitions of a set E.

```
Lemma coarserR: reflexive_r coarser_cs.
Lemma coarserT: transitive_r coarser_cs.
Lemma coarserA x: {when partition_s ^~ x &, antisymmetric_r coarser_cs}.
```

¶ We construct here a function that maps a to x and and b to y. This function is well-defined if a and b are distinct element, for instance in the case of `C0` and `C1`.

```
Definition variant a x y := (fun z:Set => Yo (z = a) x y).
Definition variantL a b x y := Lg (doubleton a b) (variant a x y).
Definition variantLc f g:=  Lvariant C0 C1 f g.
Definition varianti x a b := fun z => Yo (inc z x) a b.

Lemma variant_true a x y z: z = a -> variant a x y z = x.
Lemma variant_false a x y z: z <> a -> variant a x y z = y.
Lemma varianti_in z x a b: inc z x -> (varianti x a b z) = a.
Lemma varianti_out z x a b: ~ inc z x -> (varianti x a b z) = b.
Lemma variant_V_a a b x y: Vg (variantL a b x y) a = x.
Lemma variant_V_b a b x y: b <> a ->  Vg (variantL a b x y) b = y.
Lemma variant_fgraph a b x y: fgraph (variantL a b x y).
Lemma variant_d a b x y: domain (variantL a b x y) = doubleton a b.
Lemma variantLc_fgraph x y: fgraph (variantLc x y).
Lemma variantLc_dxs f g:  domain (variantLc f g) = C2.
Lemma variantLc_domain_ne: forall f g, nonempty (domain (variantLc f g)).
Lemma variant_V_ca x y: Vg (variantLc x y) C0 = x.
Lemma variant_V_cb x y: Vg (variantLc x y) C1 = y.
Lemma variant_true1 x y: variant C0 x y C0 = x.
Lemma variant_false1 x y: variant C0 x y C1 = y.
Lemma variantLc_comp a b f:
  variantLc (f a) (f b) =
  Lg (domain (variantLc a b)) (fun z => f (Vg (variantLc a b) z)).
```

If X is a subset of E then X and E \ X form a partition of X (it is a non-empty partition only if X is neither empty nor E).

```
Definition partition_with_complement x j :=
  variantLc j (x -s j).

Lemma is_partition_with_complement x j:
  sub j x -> partition_w_fam (partition_with_complement x j) x.
```

The set of non-empty partitions on X can be ordered by the finer ordering on coverings; we give here the smallest and largest element of the set. If X_ι is a partition family, then the

mapping $\iota \mapsto X_\iota$ is injective (we use the fact that X_ι is not empty). Inverse images of disjoint sets by a function are disjoint.

```
Definition greatest_partition x := fun_image x singleton.
Definition least_partition x := (singleton x).
Definition injective_graph f:=
  fgraph f /\  {inc domain f &, injective (Vg f)}.

Lemma least_is_partition x:
   nonempty x -> partition_s (least_partition x) x.
Lemma greatest_partition_P x z:
  inc z (greatest_partition x) <-> exists2 w, inc w x & z = singleton w.
Lemma greater_is_partition x: partition_s (greatest_partition x) x.
Lemma injective_partition f x:
  partition_s_fam f x -> injective_graph f.
Lemma partition_fam_partition f x:
  partition_s_fam f x -> partition_s (range f) x.
Lemma inv_image_disjoint g: function g ->
  {compat (inv_image_by_fun g) : x y / disjoint x y}.
```

Proposition 8 [2, p. 100] is an immediate consequence of Proposition 7. If $(X_\iota)_\iota$ is a partition of X and $f_\iota \in \mathscr{F}(X_\iota, T)$, then there exists a unique $f \in \mathscr{F}(X, T)$ that extends every f_ι. The assumption is that f_ι is a function defined on X_ι, with target T. We give a variant (without uniqueness) where the target of f_ι is a subset of T. The set of functions $\mathscr{F}$ will be defined later.

```
Lemma extension_partition_aux f x t h:
  partition_w_fam f x ->
  (forall i, inc i (domain f) -> function_prop (h i) (Vg f i) t) ->
  (forall i j, inc i (domain f) -> inc j (domain f) ->
    agrees_on ( (Vg f i) \cap (Vg f j)) (h i) (h j)).
Lemma extension_partition1 f x t h (g := common_ext f h t):
  partition_w_fam f x ->
  (forall i, inc i (domain f) -> function_prop (h i) (Vg f i) t) ->
  (function_prop g x t /\
    (forall i, inc i (domain f) -> agrees_on (Vg f i) g (h i))).
Lemma extension_partition2 f x t h
  (g:= common_ext t (fun i => (triple (Vg f i) t (graph (h i)))) t):
  partition_w_fam f x ->
  (forall i, inc i (domain f) -> function_prop_sub (h i) (Vg f i) t) ->
  ( function_prop g x t /\
    forall i, inc i (domain f) -> agrees_on (Vg f i) g (h i)).

Theorem extension_partition_thm f x t h:
  partition_w_fam f x ->
  (forall i, inc i (domain f) -> function_prop (h i) (Vg f i) t) ->
  exists ! g, (function_prop g x t /\
    (forall i, inc i (domain f) -> agrees_on (Vg f i) g (h i))).
```

4.7 Sum of a family of sets

Proposition 9 [2, p. 100] says that, for any family X_ι, there exists a family X'_ι of sets equipotent to X_ι, that are mutually disjoint, and a set X that is the union of these sets. After that, Bourbaki defines the *sum* of a family as the union of the family $X_\iota \times \{\iota\}$. These sets form a

partition of the sum. Proposition 10 [2, p. 101] says that that if X_t is a family with union A and sum S, there is a bijection between A and S if the family is disjoint. A comment says that there always exists a surjection. This will be used later on to prove that the cardinal of a union is not greater that the sum of the cardinals of the members of the family.

```
Definition disjointU_fam f := Lg (domain f)(fun i => (Vg f i) *s1 i).
Definition disjointU f := unionb (disjointU_fam f).

Lemma disjointU_disjoint f:
  mutually_disjoint(disjointU_fam f).
Lemma disjointU_fgraph f: fgraph (disjointU_fam f).
Lemma disjointU_d f: domain (disjointU_fam f) = domain f.

Theorem disjoint_union_lemma f:
  exists g x,
    [/\ fgraph g, x = unionb g,
    (forall i, inc i (domain f) -> (Vg f i) \Eq (Vg g i))
    & mutually_disjoint g].

Lemma disjointU_hi  f x: inc x (disjointU f) ->
  [/\ inc (Q x) (domain f), inc (P x) (Vg f (Q x)) & pairp x].
Lemma disjointU_P f x: inc x (disjointU f) <->
  [/\ inc (Q x) (domain f), inc (P x) (Vg f (Q x)) & pairp x].
Lemma disjointU_pi f x y:
  inc y (domain f) -> inc x (Vg f y) ->
  inc (J x y) (disjointU f).

Lemma disjointU2_rw a b x y:  y <> x ->
  disjointU (variantL x y a b) = (a *s1 x) \cup (b *s1 y).
Lemma disjoint_union2_rw1 a b:
  disjointU (variantLc a b) = (a *s1 C0) \cup (b *s1 C1).

Lemma partition_disjointU f:
  partition_w_fam (disjointU_fam f)  (disjointU f).

Theorem disjointU_pr f
  (h := fun i => Lf P  ((Vg f i) *s1 i) (unionb f))
  (g := common_ext (disjointU_fam f) h (unionb f)):
  [/\ source g = disjointU f,
    target g = unionb f,
    surjection g,
    (mutually_disjoint f -> bijection g)].
```

We sometimes consider the disjoint union of two sets A and B. This is the set $A \times \{0\} \cup B \times \{1\}$.

```
Definition canonical_du2 a b := disjointU (variantLc a b).

Lemma candu2_rw a b:
  canonical_du2 a b =  (a *s1 C0) \cup (b *s1 C1).
Lemma candu2P a b x:
  inc x (canonical_du2 a b) <-> (pairp x /\
  ((inc (P x) a /\ Q x = C0) \/ (inc (P x) b /\ Q x = C1))).
Lemma candu2_pr2 a b x:
  inc x (canonical_du2 a b) -> (Q x = C0 \/ Q x = C1).
Lemma candu2_pra a b x:
```

```
  inc x a -> inc (J x C0) (canonical_du2 a b).
Lemma candu2_prb a b x:
  inc x b -> inc (J x C1) (canonical_du2 a b).
```

Chapter 5

Product of a family of sets

5.1 The axiom of the set of subsets

Bourbaki has an axiom (Axiom 4 in the English edition) that asserts the existence, for every set X, of the set of subsets of X. This is sometimes called the powerset of X, it is denoted by $\mathfrak{P}(X)$. C. Simpson has defined it in Section 2.7.

If f is a correspondence from A to B, then $f\langle X\rangle \subset B$ whenever $X \subset A$. This gives a function from $\mathfrak{P}(A)$ to $\mathfrak{P}(B)$, called *extension to sets of subsets*. If we denote it by $\hat{f}$, then the extension of $f \circ g$ is $\hat{f} \circ \hat{g}$. The extension of the identity is the identity. The extension of an inverse is an inverse of the extension; this can be more formally restated in Proposition 1 [2, p. 101] as: if f is surjective (resp. injective), then its restriction to the set of sets is surjective (resp. injective).

```
Definition extension_to_parts f :=
  Lf (image_by_fun f) (powerset (source f)) (powerset (target f)).
Lemma etp_axiom f:  correspondence f ->
  lf_axiom (image_by_fun f) (powerset (source f))  (powerset (target f)).
Lemma etp_f f:
  correspondence f -> function (extension_to_parts f).
Lemma etp_V f x:
  correspondence f -> sub x (source f)
  -> Vf (extension_to_parts f) x = image_by_fun f x.
Lemma etp_composable f g:
  composableC g f ->
  (extension_to_parts g) \coP (extension_to_parts f).
(*
  Lemma etp_compose f g:
    composableC g f ->
    (extension_to_parts g) \co (extension_to_parts f)
    = extension_to_parts (g \co f).
*)
Lemma etp_compose:
  {when: composableC , {morph extension_to_parts: x y / x \co y }}.

Lemma etp_identity x:
  extension_to_parts (identity x) = identity (powerset x).
Lemma composable_for_function f g: g \coP f -> composableC g f.

Theorem etp_fs f: surjection f -> surjection (extension_to_parts f).
Theorem etp_fi f: injection f -> injection (extension_to_parts f).
```

5.2 Set of mappings of one set into another

The set of all graphs of functions from E to F is denoted by F^E: this is a subset of the powerset of $E \times F$. The set of all functions, namely the set of triples (G, E, F) where $G \in F^E$, is denoted by $\mathscr{F}(E;F)$. A bijection from E to itself is called a *permutation* of E.

```
Definition functions x y :=
  Zo (correspondences x y)
     (fun z => fgraph (graph z)/\ x = domain (graph z)).
Definition permutations E :=
  Zo (functions E E) bijection.
Lemma fun_set_P x y f:
  inc f (functions x y) <-> (function_prop f x y).
```

We introduce now F^E. It is canonically isomorphic to $\mathscr{F}(E;F)$; this means that using one set or the other does not change the size of a proof.

```
Definition gfunctions x y :=
  Zo (powerset (x \times y))(fun z => fgraph z /\ x = domain z).

Lemma gfun_set_i f:
  function f -> inc(graph f) (gfunctions (source f) (target f)).
Lemma gfun_set_hi x y z:
  inc z (gfunctions x y) -> exists f,
    [/\ function  f,  source f = x, target f = y & graph f = z].
```

The set of partial functions from x to y will be used later on. It is the union of the sets of functions from x' to y, where $x' \subset x$. We give the characteristic property of this set.

```
Definition sub_functions x y:=
  unionf(powerset x)(functions ^~ y).
Lemma sfun_set_P x y f:
  inc f (sub_functions x y) <->
  [/\ function  f,  sub (source f) x & target f = y].
```

The set of functions $E \to F$ is small (has at most one element) if E or F is empty. It is non-empty if E is empty, or if F is non-empty. We could restate this as: if x and y are two cardinals, if one of them is zero, then x^y is zero or one; if x is non-zero, or y is zero, then x^y is non-zero. We then show that there is an obvious bijection between $\mathscr{F}(E;F)$ and F^E.

```
Lemma function_exten5 x y a b:
  inc a (functions x y) -> inc b (functions x y) ->
  graph a = graph b -> a = b.
Lemma fun_set_small_source y: small_set (functions emptyset y).
Lemma fun_set_small_target x: small_set (functions x emptyset).
Lemma fun_set_ne x y: (x = emptyset \/ nonempty y) -> nonempty (functions x y).
Lemma graph_lf_axiom x y: lf_axiom graph (functions x y)  (gfunctions x y).
Lemma graph_fb x y: bijection (Lf graph (functions x y) (gfunctions x y)).
Lemma fun_set_equipotent x y:  (functions x y) \Eq (gfunctions x y).
```

$$\begin{array}{ccc} E & \xrightarrow{f} & F \\ {\scriptstyle u}\uparrow & & \downarrow{\scriptstyle v} \\ E' & \xrightarrow[f']{} & F' \end{array}$$

(compose3function)

Given $f \in \mathscr{F}(\mathrm{E};\mathrm{F})$, we construct $f' \in \mathscr{F}(\mathrm{E}';\mathrm{F}')$ via $f' = v \circ f \circ u$, provided that u is a function from E' to E and v is a function from F into F'. Proposition 2 [2, p. 102] says that if u is surjective and v is injective, then this mapping is injective; if u is injective and v is surjective, then this mapping is surjective. The situation is a bit more tricky when some sets are empty.

```
Definition compose3function u v :=
  Lf (fun f =>  (v \co f) \co u)
  (functions (target u) (source v))
  (functions (source u) (target v)).

Lemma c3f_axiom u v:
  function u -> function v ->
  lf_axiom (fun f => ((v \co f) \co u))
  (functions (target u) (source v))
  (functions (source u) (target v)).
Lemma c3f_f u v:
  function u -> function v -> function (compose3function u v).
Lemma c3f_V u v f:
  function u -> function v ->
  function f -> source f = target u -> target f = source v ->
  Vf (compose3function u v) f = (v \co f) \co u.
Theorem c3f_fi u v:
  surjection u -> injection v -> injection (compose3function u v).
Theorem c3f_fs u v:
  (nonempty (source u) \/ (nonempty (source v)) \/ (nonempty (target v))
    \/ target u = emptyset) ->
  injection u -> surjection v -> surjection (compose3function u v).
Lemma c3f_fb u v:
  bijection u -> bijection v -> bijection (compose3function u v).
```

We now define the canonical bijections from $\mathscr{F}(\mathrm{B}\times\mathrm{C};\mathrm{A})$ into $\mathscr{F}(\mathrm{B};\mathscr{F}(\mathrm{C};\mathrm{A})$ or $\mathscr{F}(\mathrm{C};\mathscr{F}(\mathrm{B};\mathrm{A}))$. For any function $f(x,y)$ we can fix one of the variables to get a function.

```
Definition partial_fun_axiom f :=
  function f /\ exists a b, source f = a \times b.
Definition first_partial_fun f y:=
  Lf(fun x => Vf f (J x y)) (domain (source f)) (target f).
Definition second_partial_fun f x:=
  Lf(fun y => Vf f (J x y)) (range (source f)) (target f).
Definition first_partial_function f:=
  Lf(fun y => first_partial_fun f y) (range (source f))
  (functions (domain (source f)) (target f)).
Definition second_partial_function f:=
  Lf(fun x => second_partial_fun f x) (domain (source f))
  (functions (range (source f)) (target f)).
Definition first_partial_map b c a:=
  Lf (fun f=> first_partial_function f)
  (functions (b \times c) a)
  (functions c (functions b a)).
Definition second_partial_map b c a:=
  Lf (fun f=>  second_partial_function f)
  (functions (b \times c) a)
  (functions b (functions c a)).
```

The next lemmas show that for fixed x, the partial application f_x that maps y to $f(x,y)$ is a function. Similarly for f_y.

```
Lemma partial_fun_axiom_pr f:
  partial_fun_axiom f ->
  source f = (domain (source f)) \times (range (source f)).
Lemma fpf_axiom f y:
  partial_fun_axiom f -> inc y (range (source f)) ->
  lf_axiom (fun x => Vf f (J x y)) (domain (source f)) (target f).
Lemma spf_axiom f x:
  partial_fun_axiom f -> inc x (domain (source f)) ->
  lf_axiom (fun y => Vf f (J x y)) (range (source f)) (target f).
Lemma fpf_f f y:
  partial_fun_axiom f -> inc y (range (source f)) ->
  function (first_partial_fun f y).
Lemma spf_f f x:
  partial_fun_axiom f -> inc x (domain (source f)) ->
  function (second_partial_fun f x).
Lemma fpf_V  x y:
  partial_fun_axiom f -> inc x (domain (source f)) ->
  inc y (range (source f)) ->
  Vf (first_partial_fun f y) x = Vf f (J x y).
Lemma spf_V f x y:
  partial_fun_axiom f -> inc x (domain (source f)) ->
  inc y (range (source f)) ->
  Vf (second_partial_fun f x) y = Vf f (J x y).
```

The next lemmas show that both $x \mapsto f_x$ and $y \mapsto f_y$ are functions.

```
Lemma fpfa_axiom f:
  partial_fun_axiom f ->
  lf_axiom (fun y => first_partial_fun f y)(range (source f))
  (functions (domain (source f)) (target f)).
Lemma spfa_axiom f :
  partial_fun_axiom f ->
  lf_axiom (fun x => second_partial_fun f x)(domain (source f))
  (functions (range (source f)) (target f)).
Lemma fpfa_f f:
  partial_fun_axiom f -> function (first_partial_function f).
Lemma spfa_f f:
  partial_fun_axiom f -> function (second_partial_function f).
Lemma fpfa_V f y:
  partial_fun_axiom f -> inc y  (range (source f)) ->
  Vf (first_partial_function f) y = first_partial_fun f y.
Lemma spfa_V f x:
  partial_fun_axiom f -> inc x (domain (source f)) ->
  Vf (second_partial_function f) x = second_partial_fun f x.
```

Denote the mapping $x \mapsto f_x$ by $\tilde{f}$. We show here that the mapping $f \mapsto \tilde{f}$ is a function. We assume that the source is nonempty.

```
Lemma fpfb_axiom a b c:
  nonempty b -> nonempty c ->
  lf_axiom (fun f=> first_partial_function f)
  (functions (b \times c) a)
  (functions c (functions b a)).
Lemma spfb_axiom a b c:
  nonempty b -> nonempty c ->
  lf_axiomf (fun f=> (fun f=> second_partial_function f)
```

```
  (functions (b \times c) a)
  (functions b (functions c a)).
Lemma fpfb_f a b c:
  nonempty a -> nonempty b -> function (first_partial_map a b c).
Lemma spfb_f a b c:
  nonempty a -> nonempty b -> function (second_partial_map a b c).
Lemma fpfb_V a b c f:
  nonempty a -> nonempty b -> inc f (functions (a \times b) c) ->
  Vf (first_partial_map a b c) f = first_partial_function f.
Lemma spfb_V a b c f:
  nonempty a -> nonempty b -> inc f (functions (a \times b) c) ->
  Vf (second_partial_map a b c) f = second_partial_function f.
Lemma fpfb_VV a b c f x:
  nonempty a -> nonempty b -> inc f (functions (a \times b) c) ->
  inc x (a \times b) ->
  Vf (Vf (Vf (first_partial_map a b c) f) (Q x)) (P x) = Vf f x.
Lemma spfb_VV a b c f x:
  nonempty a -> nonempty b -> inc f (functions (a \times b) c) ->
  inc x (a \times b) ->
  Vf (Vf (Vf (second_partial_map a b c) f) (P x)) (Q x) = Vf f x.
```

We now prove the main result, Proposition 3 of [2, p. 103].

```
Theorem fpfa_fb a b c:
  nonempty a -> nonempty b -> bijection (first_partial_map a b c).
Theorem spfa_fb a b c:
  nonempty a -> nonempty b -> bijection (second_partial_map a b c).
```

5.3 Definition of the product of a family of sets

An element of the product of two sets X_1 and X_2 is a pair of elements of X_1 and X_2, an element of the product of n sets $X_1,\ldots,X_n$ is a tuple $(x_1,\ldots,x_n)$, and thus an element of the product of a family $(X_\iota)_{\iota\in I}$ is a family $(x_\iota)_{\iota\in I}$ such that $x_\iota \in X_\iota$. We give two definitions of the product, in the same way as we gave four definitions for the union or the intersection (the variant `productt` could be defined but it not used, the last variant `product` corresponds to the notion of an unordered product; for an example, see annex of the second part of this report).

Given a family $(X_\iota)_{\iota\in I}$ of sets defined on I, we may consider functions f such that $f(\iota) \in X_\iota$. The target of $f(\iota)$ is in the union $A = \bigcup X_\iota$ and the graph is an element of $\mathfrak{P}(I \times A)$. Thus, we define the product $\prod X_\iota$ as the set of all elements of $\mathfrak{P}(I \times A)$ that are graphs of functions with this property. If all X_ι are the same set E, then $A = E$, this justifies the notation E^I for the set of functional graphs from I to E since it is the product of a constant family.

```
Definition productb f:=
  Zo (powerset ((domain f) \times (unionb f)))
  (fun z => [/\ fgraph z, domain z = domain f
    & forall i, inc i (domain f) -> inc (Vg z i) (Vg f i)]).

Definition productf sf f :=  productb (Lg sf f).
```

We list below some basic properties of products.

```
Lemma setXb_P f x,
  inc x (productb f) <->
   [/\ fgraph x,  domain x = domain f /\
     forall i, inc i (domain f) -> inc (Vg x i) (Vg f i)].
Lemma setXf_P sf f x:
  inc x (productf sf f) <->
  [/\ fgraph x, domain x = sf & forall i, inc i sf -> inc (Vg x i) (f i)].
```

We give here an extensionality properties for elements of a product.

```
Lemma productb_gr x: productb (Lg (domain x) (Vg x)) = productb x.
Lemma unionb_gr X : unionb (Lg (domain X) (Vg X)) = unionb X.
Lemma setXb_exten f x x':
  inc x (productb f) -> inc x' (productb f) -> {inc (domain f), x =1g x'} ->
  x = x'.
Lemma setXf_exten sf f x x':
  inc x (productf sf f) -> inc x' (productf sf f) -> {inc sf, x =1g x'} ->
  x = x'.
```

We define now pr_ι, the ι-th projection of a product; it is like pr_1 and pr_2 for the product of two sets. Let f be an element of the product and ι an index; we have $\mathrm{pr}_\iota f = f_\iota$, where f_ι denotes $f(\iota)$. Thus this mapping is nothing else than $\mathcal{V}$. Here we define a function, whose source is the product $\prod \mathrm{X}_\kappa$ and whose target is X_ι.

```
Definition pr_i f i:=  Lf (Vg ^~ i) (productb f) (Vg f i).

Lemma pri_axiom f i:
  inc i (domain f) ->
  lf_axiom (Vg ^~ i)(productb f)(Vg f i).
Lemma pri_f f i:  inc i (domain f) -> function (pr_i f i).
Lemma pri_V f i x:
  inc i (domain f) -> inc x (productb f) ->
  Vf (pr_i f i) x = Vg x i.
```

If the sets X_ι are non empty, so is the product, and conversely. The idea is that we have $x_\iota \in \mathrm{X}_\iota$ for some x_ι, and we can construct a function $\iota \mapsto x_\iota$.

```
Lemma setXb_0 : productb emptyset = singleton emptyset.
Lemma setXb_0' f: productb (Lg emptyset f) = singleton emptyset.
Lemma setXb_ne f:  nonempty_fam f -> nonempty (productb f).
Lemma setXb_ne2 f: nonempty (productb f) -> nonempty_fam f.
```

Assume $\mathrm{X}_\iota \subset \mathrm{E}$. An element of the product $\prod_{\iota\in\mathrm{I}} \mathrm{X}_\iota$ is the graph of a function from I to E. The converse is true if $\mathrm{X}_\iota = \mathrm{E}$ for all ι. Then $\prod_{\iota\in\mathrm{I}} \mathrm{E} = \mathrm{E}^{\mathrm{I}}$.

```
Lemma graphset_P1 a b z:
  inc z (gfunctions a b) <->
   [/\ fgraph z, domain z = a & sub z (a \times b)].
Lemma graphset_P2 a b z:
  inc z (gfunctions a b) <->
  [/\  fgraph z, domain z = a & sub (range z) b].
Lemma setXb_sub_graphset f x:
  sub (unionb f) x ->
  sub (productb f) (gfunctions (domain f) x).
```

```
Lemma setXb_eq_graphset f x:
  (forall i, inc i(domain f) -> Vg i f = x) ->
  productb f = gfunctions (domain f) x.
```

¶ Special cases of products. We have already seen that if the index set I is empty, then the product has a single element: the empty graph. We consider here the case where the index set has one element α. The product is then canonically isomorphic to X_α.

```
Definition product1 x a := productb (cst_graph(singleton a) x).
Definition product1_canon x a :=
  Lf (fun i => cst_graph (singleton a) i)  x (product1 x a).

Lemma cst_graph_pr x y:  productb (cst_graph x y) = gfunctions x y.
Lemma setX1_pr x a:  product1 x a = gfunctions (singleton a) x.

Lemma setX1_P x a y:
  inc y (product1 x a) <->
  [/\ fgraph y, domain y = singleton a & inc (Vg y a) x].
Lemma setX1_pr2 f x: domain f = singleton x ->
  product1 (Vg f x) x =  productb f.

Lemma setX1_canon_axiom x a:
  lf_axiom (fun i => cst_graph (singleton a) i)  x (product1 x a).
Lemma setX1_canon_f x a: function (product1_canon x a).
Lemma setX1_canon_V x a i:
  inc i x -> Vf (product1_canon x a) i = cst_graph (singleton a) i.
Lemma setX1_canon_fb x a:  bijection (product1_canon x a).
```

We now consider the case of two sets. For each index set $I = \{\alpha,\beta\}$, if x and y are two sets, we can consider the family $(X_\iota)_{\iota\in I}$ such that $x = X_\alpha$, $y = X_\beta$. We can define a bijection between $x \times y$ and the the product $\prod X_\iota$. For simplicity, we consider only the case where I is the canonical doubleton.

```
Definition product2 x y := productf C2 (variant C0 x y).
Definition product2_canon x y :=
  Lf (fun z => (variantLc (P z) (Q z)))  (x \times y) (product2 x y).

Lemma setX2_P x y z:
  inc z (product2 x y) <->
  [/\ fgraph z, domain z = C2P , inc (Vg z C0) x & inc (Vg z C1) y].
Lemma setX2_canon_axiom x y:
  lf_axiom (fun z  => Lvariantc (P z) (Q z))
  (x \times y) (product2 x y).
Lemma setX2_canon_f x y:
  function (product2_canon x y).
Lemma setX2_canon_V x y z:
  inc z (x \times y) -> Vf (product2_canon x y) z = variantLc (P z) (Q z).
Lemma setX2_canon_fb x y:
  bijection (product2_canon x y).
```

If each X_ι is a singleton, so is the product $\prod X_\iota$.

```
Lemma setX_set1 f: (allf f singletonp) -> singletonp (productb f).
```

The set of graphs of constant functions $I \to E$ is called the diagonal of E^I. The application that associates to x the constant function with value x is an injection from E to E^I.

```
Definition diagonal_graphp e i :=
  Zo (gfunctions i e) constantgp.
Definition constant_functor i e:=
  Lf (fun x => cst_graph i x) e (gfunctions i e).

Lemma diagonal_graph_P e i x:
  inc x (diagonal_graphp e i) <->
  [/\ constantgp x, domain x = i & sub (range x) e].
Lemma cf_injective i e:
  nonempty i -> injection (constant_functor i e).
```

Proposition 4 [2, p. 104] says: Given a family X_ι and a bijection f, the product $\prod X_\iota$ is isomorphic to the product $\prod X_{f(\iota)}$. Note that in the case of union or intersection, we have equality if f is surjective. The idea is that, if $x_\iota \in X_\iota$ and $\iota = f(\kappa)$ then $(x \circ f)_\kappa \in (X \circ f)_\kappa$. Some machinery is needed because x is a graph and f a function. These objects are not composable (we must compose x and the graph of f).

```
Definition product_compose f u :=
  Lf (fun x => x \cg (graph u))
  (productb f) (productf (source u) (fun k => Vg f (Vf u k))).

Section ProductCompose.
Variables (f u: Set).
Hypotheses (bu: bijection u) (tudf: target u = domain f).

Lemma pc_axiom0 c
  (g:= (triple (domain c) (range c) c) \co u):
  inc c (productb f)  ->
  [/\ function g, c \cg (graph u) = graph g &
    (forall i, inc i (source u) ->
    Vg (graph g) i = Vg c (Vg (graph u) i))].
Lemma pc_axiom:
  lf_axiom (fun x => x \cg (graph u))
     (productb f) (productf (source u) (fun k  => Vg f (Vf u k))).
Lemma pc:  function (product_compose f u).
Lemma pc_V x:
  inc x (productb f) -> Vf (product_compose f u) x  = x \cg (graph u).
Lemma pc_VV f u x i:
  inc x (productb f) -> inc i (source u) ->
   Vg (Vf (product_compose f u) x) i = Vg x (Vf u i).
Lemma pc_fb: bijection (product_compose f u).
End ProductCompose.
```

5.4 Partial products

Given a family X_i with index I and a subset $J \subset I$, we can restrict the family to J; we have $\bigcup_J \subset \bigcup_I$ and $\bigcap_J \supset \bigcap_I$. The case of a product is more complicated. If $x \in \prod_I$, the restriction of x to J is in $\prod_J$. The converse is not clear: given an element of $\prod_J$, is there an extension? Is it unique? We start with some lemmas concerning restrictions.

```
Lemma restriction_graph2 f j:
```

```
  fgraph f -> sub j (domain f) ->
  Lg j (Vg f) = f \cg (diagonal j).
```

We now define the restriction product and the function that associates to each x of the product its restriction to J. This function will be denoted by pr_J.

```
Definition restriction_product f j := productb (Lg j (Vg f)).

Definition pr_j f j :=
  Lf (fun z => restr z j) (productb f)(restriction_product f j).

Section RestrictionProduct.
Variables (f j: Set).
Hypotheses (jdf: sub j (domain f)).

Lemma restriction_productE :
  restriction_product f j = productb (restr f j).
Lemma prj_axiom:
  lf_axiom  (fun z => restr z j)
  (productb f)(restriction_product f j).
Lemma prj_f: function (pr_j f j).
Lemma prj_V x: inc x (productb f)  -> Vf (pr_j f j) x = (restr x j).
Lemma prj_VV  x i:
 inc x (productb f) -> inc i j
  -> Vg (Vf (pr_j f j) x) i =  V x i.
End RestrictionProduct.
```

Propositions 6 and 5 [2, p. 105] state that if X_ι is nonempty for $\iota \notin J$, then we can extend a function defined on J to the whole of I (using the axiom of choice or the fact that a nonempty product is nonempty). Then pr_J is surjective. A special case is when J has a single element α. Then pr_J is the composition of pr_α and the canonical function that identifies a product of a single set with this set. Thus pr_α is surjective.

```
Theorem extension_psetX f j g:
  nonempty_fam f ->
  fgraph g -> domain g = j -> sub j (domain f) ->
  (forall i, inc i j -> inc (Vg g i) (Vg f i)) ->
  exists h, [/\ domain h = domain f, fgraph h,
    (forall i, inc i (domain f) -> inc (Vg h i) (Vg f i)) &
    {inc j, h =1g g} ].
Theorem prj_fs f j:  nonempty_fam f -> sub j (domain f) ->
  surjection (pr_j f j).
Lemma pri_fs f k: nonempty_fam f ->
  inc k (domain f) -> surjection (pr_i f k).
```

A consequence is that if $X_\iota \subset Y_\iota$ then $\prod X_\iota \subset \prod Y_\iota$ (the converse is true if no X_ι is empty).

```
Lemma setXb_monotone1 f g:
  domain f = domain g ->
  (forall i, inc i (domain f) -> sub (Vg f i) (Vg g i))
  -> sub (productb f) (productb g).
Lemma setXb_monotone2 f g:
  domain f = domain g ->
  nonempty_fam f ->
  sub (productb f) (productb g)  ->
  (forall i, inc i (domain f) -> sub (Vg f i) (Vg g i)).
```

5.5 Associativity of products of sets

Consider a family X_ι. Assume that the index set I is the union of sets J_λ. For each λ, we can consider the function pr_{J_λ}. If $f \in \prod X_\iota$, then $\mathrm{pr}_{J_\lambda} f \in \prod_{\iota\in J_\lambda}$. We can consider this as a function of λ and write it as $(\mathrm{pr}_{J_\lambda} f)_\lambda$. Thus we get a function

$$f \mapsto (\mathrm{pr}_{J_\lambda} f)_{\lambda\in L}, \qquad \prod_{\iota\in I} X_\iota \to \prod_{\lambda\in L}\ (\prod_{\iota\in J_\lambda} X_\iota).$$

It is a bijection if the sets J_λ are mutually disjoint, in other words if they form a partition of I. This is Proposition 7 [2, p. 106].

```
Definition prod_assoc_axioms f g :=
  fgraph f  /\ partition_w_fam g (domain f).

Definition prod_assoc_map f g :=
  Lf (fun z => (Lg domain g) (fun l => Vf (pr_j f (Vg g l)) z)))
  (productb f)
  (productf (domain g) (fun l => (restriction_product f (Vg g l)))).
Lemma pam_axiom f g:
  prod_assoc_axioms f g ->
  lf_axioms (fun z => (Lg (domain g) (fun l => Vf (pr_j f (Vg g l)) z)))
  (productb f)
  (productf (domain g) (fun l => (restriction_product f (Vg g l)))).
Lemma pam_f f g:
  prod_assoc_axioms f g ->
  function (prod_assoc_map f g).
Lemma pam_V f g x:
  prod_assoc_axioms f g -> inc x (productb f) ->
  Vf (prod_assoc_map f g) x = (Lg (domain g) (fun l => Vf (pr_j f (Vg g l)) x)).
Lemma pam_fi f g:
  prod_assoc_axioms f g ->
  injection (prod_assoc_map f g).
Theorem pam_fb f g:
  prod_assoc_axioms f g ->
  bijection (prod_assoc_map f g).
```

Assume that the domain I is the disjoint union of two set I_1 and I_2. Let Y, Y_1 and Y_2 be the products of the family X_i over I, I_1 and I_2. There is a bijection between Y and $Y_1 \times Y_2$, because this set is equipotent to the product of the family with two elements. Assume now that each X_i is a singleton when $i \in I_2$. Then Y_2 is a singleton. The first projection from $Y_1 \times Y_2$ onto Y_1 is then a bijection. This gives a bijection between Y and Y_1. The last lemma here says that this bijection is pr_{I_1}.

```
Lemma variantLc_prop x y:
  variantLc x y = Lg C2 (variant C0 x y).
Lemma prod_assoc_map2 f g:
  prod_assoc_axioms f g -> domain g = C2
  -> (productb f) \Eq
  ((restriction_product f (Vg g C0)) \times (restriction_product f (Vg g C1))).

Lemma first_proj_fb x y:
  singletonp y -> bijection (first_proj (x \times y)).
Lemma prj_fb f j:
```

```
  sub j (domain f) ->
  (forall i, inc i ((domain f) -s j) -> singletonp (Vg f i)) ->
  bijection (pr_j f j).
```

5.6 Distributivity formulae

Let $((X_{\lambda,\iota})_{\iota\in J_\lambda})_{\lambda\in L}$ be a family of families of sets. Let $I = \prod J_\lambda$. (For the first formula, we assume J_λ non-empty). We have (Proposition 8 [2, p. 107])

$$\bigcup_{\lambda\in L}\Big(\bigcap_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcap_{f\in I}\Big(\bigcup_{\lambda\in L} X_{\lambda,f(\lambda)}\Big),$$

$$\bigcap_{\lambda\in L}\Big(\bigcup_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcup_{f\in I}\Big(\bigcap_{\lambda\in L} X_{\lambda,f(\lambda)}\Big).$$

The first result can be shown as follows. If x is in the LHS, there is a λ such x is in the intersection over J_λ, hence $x \in X_{\lambda,\mu}$, whatever μ; in particular it could be $f(\lambda)$. Conversely, Bourbaki assumes that x is not in the LHS; he considers the set $\{\iota \in J_\lambda \mid x \notin X_{\lambda,\iota}\}$. This set is not empty so that there is a function $f \in I$ whose value is in the set, so that x cannot be in the union of $X_{\lambda,f(\lambda)}$. The second result is shown by taking complements in a big set, namely the union of all sets involved. This gives a large proof (90 lines). The direct proof is shorter (ten lines).

```
Theorem distrib_union_inter f:
  (forall l, inc l (domain f) -> nonempty (domain (Vg f l))) ->
  unionf (domain f) (fun l => intersectionb (Vg f l)) =
  intersectionf (productf (domain f) (fun l => (domain (Vg f l))))
  (fun g => (unionf (domain f) (fun l => Vg (Vg f l) (Vg g l)))).
Lemma distrib_inter_union f:
  intersectionf (domain f) (fun l => unionb (Vg f l)) =
  unionf (productf (domain f) (fun l => (domain (Vg f l))))
  (fun g => (intersectionf (domain f)(fun l => Vg (Vg f l) (Vg g l)))).
```

The result is now the following: the union of $\bigcap_{\iota\in I} F_\iota$ and $\bigcap_{\kappa\in K} G_\kappa$ is the intersection on L of all $F_\iota \cup G_\kappa$; there is a similar formula if we exchange union and intersection. The general distributivity formula says that L is some complicated product, but it can be replaced by an equivalent set; we use the fact that the product of the family of two sets is equipotent to a normal product, so that $L = I \times K$ (this gives a proof who size is 50 lines long; direct proof requires only 14 lines for union, and 5 for intersection).

```
Lemma distrib_union2_inter:
  (intersectionb f)\cup (intersectionb g) =
  intersectionf((domain f) \times (domain g))
    (fun z => ((Vg f (P z)) \cup (Vg g (Q z)))).
Lemma distrib_inter2_union f g:
  (unionb f) \cap (unionb g) =
  unionf((domain f) \times (domain g))
   (fun z => ((Vg f (P z)) \cap (Vg g (Q z)))).
```

Let $((X_{\lambda,\iota})_{\iota\in J_\lambda})_{\lambda\in L}$ be a family of families of sets. Let $I = \prod J_\lambda$. We assume L and I not empty in the case of intersection. Proposition 9 [2, p. 109] says

$$\prod_{\lambda\in L}\Big(\bigcup_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcup_{f\in I}\Big(\prod_{\lambda\in L} X_{\lambda,f(\lambda)}\Big)$$

$$\prod_{\lambda\in L}\left(\bigcap_{\iota\in J_\lambda} X_{\lambda,\iota}\right) = \bigcap_{f\in I}\left(\prod_{\lambda\in L} X_{\lambda,f(\lambda)}\right).$$

In the case of union, we have to consider the special cases where L and I could be empty. Otherwise the proofs are similar. We must sometimes find an element f in the product I such that $f(\lambda)$ satisfies a given property $P(\lambda)$. We do this by considering the representative of the non-empty set $\{\lambda \in L, P(\lambda)\}$.

```
Theorem distrib_prod_union f:
  productf (domain f) (fun l => unionb (Vg f l)) =
  unionf (productf (domain f) (fun l => (domain (Vg f l))))
  (fun g => (productf (domain f) (fun l => Vg (Vg f l) (Vg g l)))).
Theorem distrib_prod_intersection f:
  (forall l, inc l (domain f) -> nonempty (domain (V f l))) ->
  productf (domain f) (fun l => intersectionb (Vg f l)) =
  intersectionf (productf (domain f) (fun l => (domain (V f l))))
  (fun g => (productf (domain f) (fun l => Vg (Vg f l) (Vg g l)))).
```

Let X_λ be the union of $X_{\lambda,\iota}$. The distributivity formula says that the product $\prod X_\lambda$ is a union; this union is a partition of the product, provided that the sets are mutually disjoint, i.e., if the $X_{\lambda,\iota}$ form a partition of X_λ.

```
Lemma partition_product f:
  (forall l, inc l (domain f) -> (partition_w_fam (Vg f l) (unionb (Vg f l)))) ->
  partition_w_fam(Lg(productf (domain f) (fun l => domain (Vg f l)) )
    (fun g => (productf (domain f) (fun l => Vg (Vg f l) (Vg g l)))))
  ( productf (domain f) (fun l => unionb (Vg f l))).
```

We apply the distributivity formulas to the case of two families of sets. In a first variant, we consider the product of a family of two sets, after that, we convert it to a normal product.

```
Lemma distrib_prod2_union f g:
  product2 (unionb f)(unionb g) =
  unionf((domain f) \times (domain g))
      (fun z =>  (product2 (V f (P z)) (V g (Q z)))).
Lemma distrib_prod2_inter f g:
  product2 (intersectionb f)(intersectionb g)=
  intersectionf((domain f) \times (domain g)) (fun z =>
    (product2 (Vg f (P z)) (Vg g (Q z)))).

Lemma distrib_product2_union f g:
  (unionb f) \times (unionb g) =
  unionf(product (domain f)(domain g)) (fun z =>
    ((V f (P z)) \times (V g (Q z)))).
Lemma distrib_product2_inter f g:
  (intersectionb f) \times (intersectionb g) =
  intersectionf(product (domain f)(domain g)) (fun z =>
    ((Vg f (P z)) \times (Vg g (Q z)))).
```

Proposition 10 [2, p. 110] says that the intersection of a product is the product of the intersection.

$$\prod_{\iota\in I}\left(\bigcap_{\kappa\in K} X_{\iota,\kappa}\right) = \bigcap_{\kappa\in K}\left(\prod_{\iota\in I} X_{\iota,\kappa}\right).$$

This is a special case of the general distributivity formula where the set J_λ is independent of λ. In the case of two families of two sets, we get $(a\times b)\cap(c\times d) = (a\times c)\cap(b\times d)$. In the case

of union, the general theorem says $(a \times b) \cup (c \times d) = (a \times c) \cup (b \times d) \cup (a \times d) \cup (b \times c)$ and there is no simpler formula. Note that if I and K are empty, the intersection is empty and the product is a singleton; this is the only case of failure.

```
Theorem distrib_inter_prod f sa sb:
  (nonempty sa \/ nonempty sb) ->
  intersectionf sb (fun k => productf sa (fun i=> Vg f (J i k))) =
  productf sa (fun i => intersectionf sb (fun k=> Vg f (J i k))).
```

If one of the sets I or K is a doubleton then we get

$$\Big(\prod_{\iota\in I} X_\iota\Big) \cap \Big(\prod_{\iota\in I} Y_\iota\Big) = \prod_{\iota\in I} (X_\iota \cap Y_\iota), \qquad \Big(\bigcap_{\iota\in I} X_\iota\Big) \times \Big(\bigcap_{\iota\in I} Y_\iota\Big) = \bigcap_{\iota\in I} (X_\iota \times Y_\iota).$$

```
Lemma distrib_prod_inter2_prod f g:
  domain f = domain g ->
  (productb f) \cap (productb g) =
  productf (domain f) (fun i => (Vg f i) \cap (Vg g i)).

Lemma distrib_inter_prod_inter f g:
    domain f = domain g ->
    product2 (intersectionb f) (intersectionb g) =
    intersectionf (domain f) (fun i => product2 (Vg f i) (Vg g i)).
Lemma distrib_prod2_sum A f:
  A \times (unionb f) = unionb (Lg (domain f) (fun x =>  A \times (Vg f x))).
```

¶ Given two functional graphs f and f' with the same domain I, we define the product to be the graph that associates $(f(x), f'(x))$ to x. Let $f'' = (f, f')$ be a pair of graphs; we can consider it as a function that associates $(f(x), f'(x))$ to x. Thus we have a mapping from $\prod \mathrm{F}_\iota \times \prod \mathrm{F}'_\iota$ into $\prod(\mathrm{F}_\iota \times \mathrm{F}'_\iota)$. We need a bunch of lemmas in order to prove that this mapping is a bijection.

```
Definition prod_of_fgraph x x':=
  Lg (domain x)(fun i => J (Vg x i) (Vg x' i)).

Definition prod_of_products_canon f f':=
  Lf (fun w => prod_of_fgraph (P w) (Q w))
  ((productb f)\times  (productb f'))
  (productf (domain f)(fun i => (Vg f i) \times (Vg f' i))).

Definition prod_of_product_aux f f' :=
  fun i =>  ((Vf f i) \times (Vf f' i)).

Definition prod_of_prod_target f f' :=
  fun_image(source f)(prod_of_product_aux f f').

Definition prod_of_products f f' :=
  Lf (prod_of_product_aux f f')(source f)(prod_of_prod_target f f').


Lemma prod_of_products_f f f':
  function (prod_of_products f f').
Lemma prod_of_products_V f f' i:
  inc i (source f) ->
  Vf (prod_of_products f f') i =  (Vf f i) \times (Vf f' i).

Section ProdProdCanon.
```

```
Variables (f f': Set).
Hypotheses (ff:function f) (ff': function f').
Hypothesis (sfsf:source f = source f').

Lemma prod_of_function_axioms x x':
  inc (graph x) (productb (graph f)) -> inc (graph x') (productb (graph f')) ->
  lf_axiom (fun i => J (Vf x i) (Vf x' i))
     (source f) (union (prod_of_prod_target f f')).
Lemma prod_of_function_V x x' i:
  inc x (productb (graph f)) ->  inc x' (productb (graph f')) ->
  inc i (source f) ->
  Vg (prod_of_fgraph x x') i =  J (Vg x i) (Vg x' i).
Lemma prod_of_function_f x x':
  inc x (productb (graph f)) ->  inc x' (productb (graph f')) ->
  inc (prod_of_fgraph x x')
  (productb (graph (prod_of_products f f'))).
Lemma popc_target_aux:
  productb(Lg (domain (graph f))
   (fun i =>  (Vg (graph f) i) \times (Vg (graph f') i))) =
  productb(graph (prod_of_products f f')).
Lemma popc_axioms :
  lf_axiom(fun w => prod_of_fgraph (P w) (Q w))
  ((productb (graph f)) \times (productb (graph f')))
  (productb (graph (prod_of_products f f'))).
Lemma popc_V w:
  inc w ((productb (graph f)) \times (productb (graph f'))) ->
  Vf (prod_of_products_canon (graph f) (graph f')) w=
  prod_of_fgraph (P w) (Q w).
Lemma popc_fb:
  bijection (prod_of_products_canon (graph f) (graph f')).
End ProdProdCanon.
```

5.7 Extensions of mappings to products

$$\begin{array}{ccc} X_\iota & \xrightarrow{f_\iota} & Y_\iota \\ {\scriptstyle \mathrm{pr}_\iota}\uparrow & & \uparrow{\scriptstyle \mathrm{pr}_\iota} \\ \prod X_\iota & \xrightarrow[(f_\iota)_{\iota\in I}]{} & \prod Y_\iota \end{array} \qquad \text{(extension)}$$

Assume that X_ι, Y_ι and f_ι are families with the same index I. We assume that f_ι is a functional graph with source X_ι and target Y_ι. If $x \in \prod X_\iota$ then $x_\iota \in X_\iota$, $f_\iota(x_\iota) \in Y_\iota$ and the mapping $\iota \mapsto f_\iota(x_\iota)$ is in $\prod Y_\iota$. This induces a function $\prod X_\iota \to \prod Y_\iota$ called the *extension* of the functions f_i.

```
Definition ext_map_prod_aux x  f := fun i=> Vg (f i) (Vg x i).
Definition ext_map_prod I src trg f :=
  Lf (fun x => Lg I (ext_map_prod_aux x f))
    (productf I src ) (productf I trg).

Definition ext_map_prod_axioms I src trg f :=
  forall i, inc i I ->
    [/\ fgraph (f i), domain (f i) = src i & sub (range (f i)) (trg i)].
```

```
Section ExtMapProd.
Variables (I: Set) (src trg f: Set-> Set).
Hypothesis ax:  ext_map_prod_axioms I src trg f.

Lemma ext_map_prod_taxioms:
  lf_axiom (fun x => Lg I (ext_map_prod_aux x f))
    (productf I src) (productf I trg).
Lemma ext_map_prod_f: function (ext_map_prod In src trg f).
Lemma ext_map_prod_V x:  inc x (productf I src) ->
  Vf (ext_map_prod I src trg f) x= L In (ext_map_prod_aux x f).
Lemma ext_map_prod_VV x i:
  inc x (productf I src) -> inc i I ->
  Vg (Vf (ext_map_prod I src trg f) x) i = Vg (f i) (Vg x i).
End ExtMapProd.
```

Proposition 11 [2, p. 111] says that composition of extensions is extension of compositions. Bourbaki uses this property to show that if all f_ι are injective, so is the extension, by exhibiting a left inverse. We use a direct proof because it is easier (note that f_ι is not a function, just the graph of a function).

```
Lemma ext_map_prod_composable I p1 p2 p3  g f h:
  ext_map_prod_axioms I p1 p2 f ->
  ext_map_prod_axioms I p2 p3 g ->
  (forall i, inc i I -> h i = (g i) \cf (f i)) ->
  (forall i, inc i I -> (g i) \cfP (f i)) ->
  ext_map_prod_axioms I p1 p3 h.

Lemma ext_map_prod_compose  I p1 p2 p3  g f h:
  ext_map_prod_axioms I p1 p2 f ->
  ext_map_prod_axioms I p2 p3 g ->
  (forall i, inc i I -> h i = (g i) \cf (f i)) ->
  (forall i, inc i I -> (g i) \cfP (f i)) ->
  (ext_map_prod I p2 p3 g) \co (ext_map_prod I p1 p2 f) =
  (ext_map_prod I p1 p3 h).

Lemma ext_map_prod_fi I p1 p2 f:
  ext_map_prod_axioms I p1 p2 f ->
  (forall i, inc i I -> injective_graph (f i)) ->
  injection (ext_map_prod I p1 p2 f).
Lemma ext_map_prod_fs I p1 p2 f:
  ext_map_prod_axioms I p1 p2 f ->
  (forall i, inc i I -> range (f i) = p2 i) ->
  surjection (ext_map_prod I p1 p2 f).
```

Let f be a function from E to A, where A is a product X_ι over I. Consider the function $\mathrm{pr}_\iota \circ f$ from E to X_ι. Its extension to products is some function $\overline{f}$ from E^I to $\prod X_\iota$. Let d be the diagonal mapping from E to E^I. We have $f = \overline{f} \circ d$. If f_ι is a family of functions from E to X_ι, and $\overline{f}$ is its extension to the products, then $\mathrm{pr}_\iota \circ (\overline{f} \circ d) = f_\iota$. The mapping from f to $\overline{f}$ is a bijection between $(\prod X_\iota)^E$ and $\prod X_\iota^E$.

```
Definition fun_set_to_prod src F :=
  Lf (fun f =>
       Lg(domain F)( fun i=> (graph ( (pr_i F i) \co
          (triple src (productb F) f)))))
```

```
     (gfunctions src (productb F))
    (productb (Lg (domain F) (fun i=> gfunctions src (Vg F i)))).

Lemma fun_set_to_prod1 F f i:
  fgraph F -> inc i (domain F) ->
  function f -> target f = productb F ->
  function_prop (pr_i F i \co f) (source f) (Vg F i) /\
   (forall x, inc x (source f) ->  Vf (pr_i F i \co f) x = Vg (Vf f x) i).

Section FunSetToProd.
Variables  (src F: Set).
Hypothesis (fF: fgraph F).


Lemma fun_set_to_prod2 f gf:
  inc gf (gfunctions src (productb F)) ->
  f = (triple src (productb F) gf) -> function_prop f src (productb F).
Lemma fun_set_to_prod3 :
  lf_axiom(fun f =>
       Lg(domain F)( fun i=> (graph (compose (pr_i F i)
          (triple src (productb F) f)))))
     (gfunctions src (productb F))
    (productb (Lg (domain F) (fun i=> gfunctions src (Vg F i)))).
Lemma fun_set_to_prod4:
  function_prop (fun_set_to_prod src F) (gfunctions src (productb F))
  (productb (Lg (domain F) (fun i=> gfunctions src (Vg i F)))).
Definition fun_set_to_prod5 F f :=
  ext_map_prod (domain F) (fun i=> source f)(fun i=> Vg F i)
  (fun i => (graph (compose (pr_i F i) f))).
Lemma fun_set_to_prod6 f:
  function f -> target f = productb F ->
  (function (fun_set_to_prod5 F f) /\
    (fun_set_to_prod5 F f) \coP (constant_functor (domain F)(source f) )/\
    (fun_set_to_prod5 F f) \co (constant_functor (domain F)(source f)) =f).
Lemma fun_set_to_prod7 f g:       (* 56 *)
  (forall i, inc i (domain F) -> inc (f i) (gfunctions src (Vg F i))) ->
  g = ext_map_prod (domain F) (fun i=> src)(Vg F) f ->
  (forall i, inc i (domain F) ->
    f i = graph ((pr_i F i) \co (g \co (constant_functor (domain F) src) ))).
Lemma fun_set_to_prod8: (* 60 *)
  bijection (fun_set_to_prod src F).

End FunSetToProd.
```

Chapter 6

Equivalence relations

The code of the first two sections of this chapter was originally written by Carlos Simpson. A *relation* between two objects x and y, often denoted by $x \sim y$, is a function of type `Set` → `Set` → Prop. An *equivalence relation* will be a relation with some properties; an *equivalence* will be a graph with similar properties; this differs from the Bourbaki's definition, where an equivalence is a correspondence.

6.1 Definition of an equivalence relation

We say that x is related to y by the graph r and denote it by $x \overset{r}{\sim} y$ whenever the pair (x, y) is in the graph. The set of related objects is called the *substrate* of the graph.

```
Definition substrate r :=  (domain r) \cup (range r).
```

We have some characterizations of the substrate. Only the last one requires that r be a graph.

```
Lemma pr1_sr r y:  inc y r -> inc (P y) (substrate r).
Lemma pr2_sr r y:  inc y r -> inc (Q y) (substrate r).
Lemma arg1_sr r x y:  related r x y -> inc x (substrate r).
Lemma arg2_sr: forall r x y,  related r x y -> inc y (substrate r).
Lemma substrate_smallest r s:
  (forall y, inc y r -> inc (P y) s) ->
  (forall y, inc y r -> inc (Q y) s) ->
  sub (substrate r) s.
Lemma substrate_P r: sgraph r ->  forall x,
  inc x (substrate r) <->
   ((exists y, inc (J x y) r) \/  (exists y, inc (J y x) r)).
Lemma substr_i r x: inc (J x x) r -> inc x (substrate r).
```

We say that a relation $\sim$ is *symmetric* if $x \sim y$ implies $y \sim x$, *antisymmetric* if $x \sim y$ and $y \sim x$ imply $x = y$, *transitive* if $x \sim y$ and $y \sim z$ implies $x \sim z$. We say that it is *reflexive* on E if $x \in$ E is equivalent to $x \sim x$; we say that it is *reflexive* if $x \sim y$ implies $x \sim x$ and $y \sim y$.

We say that $\sim$ is an *equivalence relation* if it is symmetric and transitive (it is then reflexive). We say that it is a *preorder relation* if it is reflexive and transitive; we say that it is an *order relation* if it is reflexive, antisymmetric and transitive. We say that a relation is an *equivalence relation on* E or an *order on E* if it is an equivalence or an order, and moreover is reflexive on E.

```
Section Definitions.
Implicit Type  (r: relation).

Definition reflexive_re r E :=  forall x, inc x E <-> r x x.
Definition reflexive_rr r :=  forall x y, r x y -> (r x x /\  r y y).

Definition equivalence_r r :=  symmetric_r r /\ transitive_r r.
Definition equivalence_re r E :=  equivalence_r r /\ reflexive_re r E.
Definition order_r r := [/\ transitive_r r, antisymmetric_r r & reflexive_rr r].

Definition preorder_r r :=  transitive_r r /\ reflexive_rr r.
Definition order_re r E :=  order_r r /\ reflexive_re r E.
End Definitions.
```

The definitions for a graph are similar. We say that a graph is reflexive if its associated relation is reflexive on the substrate, i.e., if $x \overset{r}{\sim} y$ implies $x \overset{r}{\sim} x$ and $y \overset{r}{\sim} y$. An *equivalence* is a set that is reflexive, symmetric, and transitive.

```
Definition reflexivep r := forall y, inc y (substrate r) -> related r y y.
Definition symmetricp r := symmetric_r (related r).
Definition antisymmetricp r := antisymmetric_r (related r).
Definition transitivep r :=  transitive_r (related r).

Definition equivalence r :=
  [/\ sgraph r, reflexivep r, transitivep r & symmetricp r].
Definition order r :=
  [/\ sgraph r, reflexivep r, transitivep r & antisymmetricp r].
Definition preorder r :=
  [/\ sgraph r, reflexivep r & transitivep r].
Definition order_on r E := order r /\ substrate r = E.
```

Let r be a set, R, D and S be the range, domain and substrate of r; by definition $S = D \cup R$. If $(x, x) \in r$, then $x \in D$, thus $x \in S$. The relation r is reflexive when $x \in S$ implies $(x, x) \in r$. If this is the case, we have S = D. In particular S = D whenever r is an equivalence or an order. Note that if r is symmetric, transitive and a graph, it is reflexive, thus an equivalence. Since both D and R are increasing functions of r (for inclusion), so is S.

```
Lemma equivalence_sgraph  r: equivalence r -> sgraph r.
Lemma order_sgraph r: order r -> sgraph r.
Lemma preorder_sgraph r: preorder r -> sgraph r.

Lemma reflexive_domain g: reflexivep g -> domain g = substrate g.
Lemma domain_sr g: equivalence g -> domain g = substrate g.
Lemma domain_sr1 r: order r -> domain r = substrate r.
Lemma symmetric_transitive_equivalence r:
  sgraph r -> symmetricp r -> transitivep r -> equivalence r.
Lemma equivalence_relation_pr1 g:
  sgraph g -> equivalence_r (related g)  -> equivalence g.
Lemma substrate_sub: {compat substrate : x y / sub x y}.
```

Some trivial properties of an equivalence.

```
Lemma reflexivity_e r u:
  equivalence r -> inc u (substrate r) -> related r u u.
```

```
Lemma symmetricity_e r u v:
  equivalence r -> related r u v -> related r v u.
Lemma transitivity_e r v u w:
  equivalence r ->  related r u v -> related r v w -> related r u w.
Lemma equivalence_equivalence r:
  equivalence r -> equivalence_re (related r)(substrate r).
```

For every set E and relation $\sim$ we can define a set $r = g_{\mathrm{E}}(\sim)$, the graph of $\sim$ on E, which is the set of all pairs $(x, y) \in \mathrm{E} \times \mathrm{E}$ such that $x \sim y$. We deduce a relation $\overset{r}{\sim}$ whose substrate is a subset of E. By definition, $x \overset{r}{\sim} y$ is equivalent to $x \in \mathrm{E}$, and $y \in \mathrm{E}$ and $x \sim y$. Assume that $\sim$ is an equivalence or order relation on E. Then $x \overset{r}{\sim} y$ simplifies to $x \sim y$. In fact, in the case of an ordering, we assume that $x \sim y$ implies $x \sim x$ and $y \sim y$. In the case of an equivalence this follows from symmetry and transitivity. Now $x \sim x$ implies $x \in \mathrm{E}$ by reflexivity.

```
Definition graph_on (r:relation) x
  := Zo(coarse x)(fun w => r (P w)(Q w)).

Lemma graph_on_graph r x:  sgraph(graph_on r x).
Lemma graph_on_P0 r x a b:
  inc (J a b) (graph_on r x) <->  [/\ inc a x, inc b x & r a b].
Lemma graph_on_P1 r x a b:
  related  (graph_on r x) a b <->  [/\ inc a x, inc b x & r a b].
Lemma graph_on_P2 r x : equivalence_re r x -> forall u v,
  (related (graph_on r x) u v <-> r u v).
Lemma graph_on_P3 r x:  order_re r x -> forall u v,
  (related (graph_on r x) u v <-> r u v).
Lemma graph_on_sr1 r x:
  sub (substrate (graph_on r x)) x.
```

If the relation $\sim$ is a preorder, an order, or an equivalence relation, then its graph $g_{\mathrm{E}}(\sim)$ is a preorder, an order, or an equivalence. If $x \sim x$ holds for any $x \in \mathrm{E}$, then the substrate of this graph is E.

```
Lemma order_preorder r: order r -> preorder r.
Lemma preorder_from_rel r x:
  preorder_r r -> preorder (graph_on r x).
Lemma order_from_rel r x:
  order_r r -> order (graph_on r x).
Lemma equivalence_from_rel r x:
  equivalence_r r -> equivalence (graph_on r x).
Lemma graph_on_sr (r: relation) x:
  (forall a, inc a x -> r a a) ->
  substrate (graph_on r x) = x.
```

If E is a set, and $x \sim y$ is the equality relation restricted to E, namely, "$x \in \mathrm{E}$ and $y \in \mathrm{E}$ and $x = y$", its graph is the diagonal of E. This is both an equivalence relation and an order (note that, if r is an equivalence and an order, it is symmetric and antisymmetric, so that $x \overset{r}{\sim} y$ gives $x = y$).

```
Definition restricted_eq x := fun u v => inc u x /\ u = v.

Lemma diagonal_graph_on x: graph_on (restricted_eq x) x = diagonal x.
Lemma diagonal_equivalence x: equivalence (diagonal x).
Lemma diagonal_osr x:  order_on (diagonal x) x.
```

We have already shown that to be equipotent is reflexive, symmetric and transitive. Thus, it is an equivalence relation.

```
Lemma equipotent_equivalence: equivalence_r equipotent.
```

We can consider the relation on E for which all elements are related. Its graph is E × E.

```
Lemma coarse_sr u: substrate (coarse u) = u.
Lemma coarse_graph x: sgraph (coarse x).

Lemma coarse_related u x y:
  related (coarse u) x y <-> (inc x u /\ inc y u).
Lemma coarse_equivalence u:
  equivalence (coarse u).

Lemma sub_graph_coarse_substrate r:
  sgraph r -> sub r (coarse (substrate r)).
```

The set of all elements related to x is its class. We have seen two examples where (a) classes are singleton, and (b) there is a unique class, the whose set. We give here an example, with $A \subset E$, where the classes are A, and all singletons $\{x\}$, for $x \notin A$.

```
Lemma equivalence_relation_bourbaki_ex5 A E
  (r := (fun x y => (inc x (E -s A) /\ (x = y) \/ (inc x A /\ inc y A)))):
  sub A E ->
  (equivalence (graph_on r E) /\ substrate  (graph_on r E) = E).
```

Consider a family of relations $(\sim_i)_{i\in I}$. This intersection is reflexive, symmetric, transitive, an equivalence, and order, provided that each member of the family has the property.

```
Lemma setIrel_graph z:
  (all z sgraph) ->  sgraph (intersection z).
Lemma setIrel_P z: nonempty z -> forall x y,
  (related (intersection z) x y <->
   (forall r, inc r z -> related r x y)).
Lemma setIrelR z:
  (alls z reflexivep) -> reflexivep (intersection z).
Lemma setIrel_sr z e:
  nonempty z -> (alls z reflexivep) ->
  (forall r, inc r z -> substrate r = e) ->
  substrate (intersection z) = e.
Lemma setIrelT z:
  (alls z transitivep) -> transitivep (intersection z).
Lemma setIrelS z:
  (all s z -> symmetricp) -> symmetricp (intersection z).
Lemma setIrel_equivalence z:
  (alls z equivalence) -> equivalence (intersection z).
Lemma setIrel_or z: (alls z order) -> order (intersection z).
```

We can consider the set of all equivalences on E. It is not empty.

```
Definition equivalences x :=
  Zo (powerset (coarse x)) (fun r => (equivalence r)
    /\ (substrate r = x)).
Lemma equivalencesP r x:
```

```
  inc r (equivalences x) <->(equivalence r /\ (substrate r = x)).
Lemma inc_coarse_all_equivalence_relations u:
  inc (coarse u) (equivalences u).
```

Proposition 1 [2, p. 114] says that a correspondence Γ between X and X is an equivalence on X if and only if X is the domain of Γ, $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$. We prove this property for graphs rather than correspondences.

```
Lemma selfinverse_graph_symmetric r: sgraph r ->
  (symmetricp r <-> (r = inverse_graph r)).
Lemma idempotent_graph_transitive r:
  sgraph r -> (transitivep r <-> sub (r \cg r) r).
Theorem equivalence_pr r:
  equivalence r  <-> ((r \cg r) = r /\  r = inverse_graph r).
```

6.2 Equivalence classes; quotient set

Let f be a function on E; the relation $f(x) = f(y)$ is an equivalence relation on E. We shall denote it by $\sim_f$. It has a graph, namely $\mathrm{F}^{-1} \circ \mathrm{F}$, where F is the graph of f.

```
Definition eq_rel_associated f x y :=
  [/\ inc x (source f), (inc y (source f)) & (Vf f x = Vf f y)].
Definition equivalence_associated f :=
  (inverse_graph (graph f)) \cg (graph f).

Section EquivalenceAssociated.
Variable (f: Set).
Hypothesis (ff : function f).

Lemma ea_graph_on:
 graph_on (eq_rel_associated f) (source f) = equivalence_associated f.
Lemma graph_ea_equivalence:
  equivalence (equivalence_associated f).
Lemma graph_ea_substrate:
  substrate (equivalence_associated f) = source f.
Lemma ea_relatedP x y:
  related (equivalence_associated f) x y <->
  [/\ inc x (source f), (inc y source f) & (Vf f x = Vf f y)].
End EquivalenceAssociated.
```

Bourbaki says that for every equivalence relation $\sim$ on E, there is a function f such that the equivalence associated with f is $\sim$ such that $\sim = \sim_f$. Let G be the graph of the equivalence relation and $x \in \mathrm{E}$. For $x \in \mathrm{E}$, the set $\mathrm{G}(x)$ of all y such that $(x, y) \in \mathrm{G}$ will be called the *equivalence class* of x, and the set of all equivalence classes will be called the *quotient set*, and denoted by $\mathrm{E}/\sim$ (or E/R if the relation is R). Let's denote by $\bar{x}$ the class of x modulo R, this is also $\mathrm{pr}_2 \mathrm{G}_x$, where G_x denotes the set all elements $z \in \mathrm{G}$ with $\mathrm{pr}_1 z = x$. We shall use the four characteristic properties of classes: (a) $y \in \bar{x}$ if and only if $x \overset{\mathrm{R}}{\sim} y$, (b) $\bar{x} \subset \mathrm{E}$, (c) $x \in \mathrm{E}$ if and only if $\bar{x} \neq \emptyset$ and (d) if $x \in \mathrm{E}$, then $x \overset{\mathrm{R}}{\sim} y$ and $\bar{x} = \bar{y}$ are equivalent. This last property says that $\sim$ is the equivalence associated to the function $x \mapsto \bar{x}$.

```
Definition class r x := fun_image (Zo r (fun z => P z = x)) Q.
Definition quotient r := fun_image (substrate r) (class r).
Definition classp r x := inc (rep x) (substrate r) /\ x = class r (rep x).
```

```
Section Class.
Variable (r:Set).
Hypothesis (er: equivalence r).

Lemma class_P x y:   (inc y (class r x) <-> related r x y).
Lemma class_is_im_of_singleton x:
  class r x = im_of_singleton r x.
Lemma sub_class_substrate x: sub (class r x) (substrate r).
Lemma class_eq1 u v: related r u v -> class r u = class r v.
Lemma class_eq2 u v: inc u (class r v) -> class r u = class r v.
```

We denote by $\hat{x} = \tau_z(z \in x)$ some element of x (if x is non-empty of course), so that $x \mapsto \hat{x}$ is is a mapping from E/R into E (it is a retraction of the canonical projection, meaning $x = \bar{\hat{x}}$). We say that x is *a class for* r if r is an equivalence (with substrate s_r), $\hat{x} \in s_r$ and $x = \bar{\hat{x}}$. Clearly, if $x \in$ E, then $\bar{x}$ is a class. Thus $a \overset{\mathrm{R}}{\sim} b$ if and only if there is a class x such that $a \in x$ and $b \in x$. If $x \in$ E/R and $y \in x$ then $\hat{x} \overset{\mathrm{R}}{\sim} y$.

```
Lemma setQ_ne x: inc x (quotient r) -> nonempty x.
Lemma setQ_repi x: inc x (quotient r) -> inc (rep x) x.
Lemma class_class x: inc x (substrate r) -> classp r (class r x).
Lemma inc_class_setQ  x:
  inc x (substrate r) -> inc (class r x) (quotient r).
Lemma setQ_P x:  inc x (quotient r) <-> classp r x.
Lemma class_rep x: inc x (quotient r) -> class r (rep x) = x.
Lemma in_class_relatedP y z:
  related r y z <-> (exists x, [/\ classp r x, inc y x & inc z x]).
Lemma related_rep_in_class  x y:
 inc x (quotient r) -> inc y x  -> related r (rep x) y.
```

A class x is a nonempty subset of E such that for all $y \in x$, properties $z \in x$ and $y \overset{\mathrm{R}}{\sim} z$ are equivalent. Two classes are equal or disjoint. If $x \in$ E then $\bar{x} \in$ E/R. If $x \in y$ and $y \in$ E/R then $x \in$ E. As a consequence, the union of E/R is E. If $x \in$ E/R then $\hat{x} \in$ E, and $\bar{\hat{x}} = x$. If $x \in$ E then $x \in \bar{x}$ and $x \overset{\mathrm{R}}{\sim} \hat{\bar{x}}$. If u and v are in E/R, then $\hat{u} \overset{\mathrm{R}}{\sim} \hat{v}$ if and only if $u = v$. The relation $u \overset{\mathrm{R}}{\sim} v$ is equivalent to $u \in$ E and $v \in$ E and $\bar{u} = \bar{v}$. If $x \in y$ and $y \in$ E/R then $y = \bar{x}$.

```
Lemma rep_in_class x: classp r x -> inc (rep x) x.
Lemma rel_in_class x y: classp r x -> inc y x -> related r (rep x) y.
Lemma sub_class_sr x: classp r x -> sub x (substrate r).
Lemma rel_in_class2 x y: classp r x -> related r (rep x) y -> inc y x.
Lemma class_dichot x y:
  classp r x -> classp r y -> disjointVeq x y.

Lemma inc_in_setQ_sr x y:
  inc x y -> inc y (quotient r) -> inc x (substrate r).
Lemma setU_setQ: union (quotient r) = substrate r.
Lemma rep_i_sr x:  inc x (quotient r) -> inc (rep x) (substrate r).
Lemma inc_itself_class x: inc x (substrate r) ->  inc x (class r x).
Lemma related_rep_class r x:
  inc x (substrate r) -> related r x (rep (class r x)).
Lemma related_rr_P u v:
  inc u (quotient r) -> inc v (quotient r) ->
  (related r (rep u) (rep v) <-> (u = v)).
Lemma related_equiv_P r u v:
  related r u v <->
```

```
   [/\ inc u (substrate r), inc v (substrate r) & class r u = class r v].
Lemma is_class_pr x y:
  inc x y -> inc y (quotient r) -> y = class r x.
End Class.
```

The *canonical projection* is the mapping $x \mapsto \bar{x}$ from E onto E/R. An important property is that this function is surjective.

```
Definition canon_proj r := Lf(class r) (substrate r) (quotient r).

Section CanonProj.
Variable (r:Set).
Hypothesis (er: equivalence r).

Lemma canon_proj_s: source (canon_proj r) = substrate r.
Lemma canon_proj_t: target (canon_proj r) = quotient r.
Lemma canon_proj_f: function (canon_proj r).
Lemma canon_proj_V x:
  inc x (substrate r) -> Vf (canon_proj r) x = class r x.
Lemma canon_proj_setQ_i  x:
  inc x (substrate r) -> inc (Vf (canon_proj r) x) (quotient r).
Lemma rel_gcp_P x y:
  inc x (substrate r) -> inc y (quotient r) ->
  (inc (J x y) (graph (canon_proj r)) <-> inc x y).
Lemma canon_proj_fs: surjection (canon_proj r).
```

The next lemma says that if $A \subset E$ and $x \subset \bar{A}$ (where $\bar{A}$ is the set of all $\bar{a}$ for $a \in A$) then $x \in E/R$. We then state Criterion 55 [2, p. 115]: $u \overset{R}{\sim} v$ if and only if $\bar{u} = \bar{v}$. The exact Bourbaki statement is "Let R be an equivalence relation on a set E, and let p be the canonical mapping of E onto E/R. Then $R\{x, y\} \iff (p(x) = p(y))$". The correct statement would be: $R\{x, y\}$ if and only if $x \in E$ and $y \in E$ and $p(x) = p(y)$. The proof is a bit strange. It starts with: "let x and y be elements of E such that $(x, y) \in G$. Then $x \in E$ and $y \in E$; let us show..."

```
Lemma sub_im_canon_proj_quotient a x:
  sub a (substrate r) ->
  inc x (image_by_fun (canon_proj r) a) ->
  inc x (quotient r).
Lemma related_e_P u v:
 related r u v <->
   [/\ inc u (source (canon_proj r)),
     inc v (source (canon_proj r)) &
     Vf (canon_proj r) u = Vf (canon_proj r) v].
End CanonProj.
```

The canonical projection $E \mapsto E/R$ is a bijection if and only if each class is a singleton. In other terms, iff R is the equality relation on E.

```
Lemma diagonal_class x u:
  inc u x -> class (diagonal x) u = singleton u.
Lemma canon_proj_diagonal_fb x:
  bijection (canon_proj (diagonal x)).
Lemma canon_proj_diagonal_fb_contra r:
  equivalence r ->  bijection (canon_proj r) ->
  r = diagonal (substrate r).
```

¶ Equivalence associated with projectors. In the product $E \times F$, one can consider the equivalence associated to the projectors pr_1 and pr_2. Since $E \times F$ is canonically isomorphic to $F \times E$, it suffices to consider the first projection. The equivalence associated to this function is such that $(x, y) \sim (x, z)$ holds for every x, y and z. Classes are objects of the form $\{x\} \times F$. The function $x \mapsto \{x\} \times F$ is a bijection of E onto $(E \times F)/R$.

```
Definition first_proj_eq x y  :=
  equivalence_associated (first_proj (x \times y)).

Lemma first_proj_equivalence x y:
  equivalence (first_proj_eq x y).
Lemma first_proj_eq_related_P x y a b:
  related (first_proj_eq x y) a b <->
  [/\ inc a (x \times y), inc b (x \times y) & P a = P b].
Lemma first_proj_sr x y:
  substrate(first_proj_eq x y) = x \times y.
Lemma first_proj_classP x y :  nonempty y -> forall z,
  (classp (first_proj_eq x y) z <->
  exists2 u, inc u x & z = (singleton u) \times y).
Lemma first_proj_equiv_proj x y:
  nonempty y ->
  bijection (Lf (fun u => (singleton u) \times  y)
    x (quotient (first_proj_eq x y))).
```

For any equivalence, the quotient is a partition of the substrate.

```
Lemma sub_quotient_powerset r:
  equivalence r -> sub (quotient r) (powerset (substrate r)).
Lemma partition_from_equivalence r:
  equivalence r ->
  partition(quotient r)(substrate r).
```

We consider now the converse. Let f be a function defined on E and F its graph. Assume that F is a partition of X. We shall write X_i instead of $f(i)$. There is a function g defined on X with values in E such that $x \in X_{g(x)}$. We can consider the relation $x \sim y$ defined by: there is an i such that $x \in X_i$ and $y \in X_i$. This relation is also $g(x) = g(y)$. This relation has a graph on X, say r. Then r is an equivalence on X. Each class of r is some X_i. Conversely, if X_i is non-empty, it is a class. If no X_i is empty, then $i \mapsto f(i)$ is a bijection from $E \to X/r$.

```
Definition in_same_coset f x y:=
   exists i, [/\ inc i (source f) , inc x (Vf f i) & inc y (Vf f i)].

Definition partition_relation f x :=
  graph_on (in_same_coset f) x.

Section InSameCoset.
Variables (f x: Set).
Hypothesis (ff: function f).
Hypothesis fpa:  partition_w_fam (graph f) x.

Lemma partition_inc_unique1 i j y:
  inc i (source f) -> inc y (Vf f i) ->
  inc j (source f) -> inc y (Vf f j) -> i = j.
Lemma isc_hi a b: (in_same_coset f a b) ->  (inc a x /\ inc b x).
```

```
Lemma isc_rel_P a b:
  (related (partition_relation f x) a b <-> in_same_coset f a b).
Lemma isc_rel1P a b: inc a x -> inc b x ->
   ((in_same_coset f a b) <-> (cover_at (graph f) a = cover_at (graph f) b)).
Lemma isc_rel_sr: substrate (partition_relation f x)= x.
Lemma isc_rel_equivalence :
  equivalence (partition_relation f x).

Lemma isc_rel_class a:
  classp (partition_relation f x) a
  -> exists2 u, inc u (source f) & a = Vf f u.
Lemma isc_rel_class2 u:
  inc u (source f) -> nonempty (Vf f u)
  -> classp (partition_relation f x)  (Vf f u).
Lemma partition_fun_fb:
  (allf (graph f) nonempty)
  -> bijection (Lf (Vf f)
    (source f) (quotient (partition_relation f x))).
End InSameCoset.
```

With the same notations, a system of representatives is a set S such that $X_i \cap S$ is a singleton. The same name is given to an injective function g whose image is a system of representatives. In this case, for every i there is a unique j such that $g(j) \in X_i$. Conversely if this condition holds and g is injective, it is a system of representatives. As a consequence, every right inverse of the canonical projection of X on the quotient set defined by the partition X_i of X is a system of representatives.

```
Definition representative_system s f x :=
  [/\ function f, partition_w_fam (graph f) x, sub s x
  & forall i, inc i (source f) -> singletonp ((Vf f i) \cap s)].

Definition representative_system_function g f x :=
  injection g /\ (representative_system (range (graph g)) f x).

Lemma rep_sys_function_pr g f x i:
  representative_system_function g f x -> inc i (source f)
  -> exists! a, (inc a (source g) /\ inc (Vf g a) (Vf f i)).
Lemma rep_sys_function_pr2 g f x:
  injection g -> function f -> partition_w_fam (graph f) x
  -> sub (target g) x
  -> (forall i, inc i (source f)
    -> exists! a, (inc a (source g) /\ inc (Vf g a) (Vf f i)))
  -> representative_system_function g f x.
Lemma section_canon_proj_pr g f x y r:
  r = partition_relation f x ->  function f -> partition_w_fam (graph f) x
  -> is_right_inverse (canon_proj r) g ->
  inc y x ->
  related r y (Vf g (class r y)).
Lemma section_is_representative_system_function g f x:
  function f -> partition_w_fam (graph f) x
  -> is_right_inverse (canon_proj (partition_relation f x)) g ->
  (forall u, inc u (source f) -> nonempty (Vf f u)) ->
  representative_system_function g f x.
```

6.3 Relations compatible with an equivalence relation

We say that P(x) is *compatible* with $\sim$ if P(x) and $x \sim y$ imply P(y). Every property is compatible with the equality.

```
Definition compatible_with_equiv_p (p: property)(r:Set) :=
  forall x x', p x -> related r x x' -> p x'.

Lemma trivial_equiv p x:  compatible_with_equiv_p p (diagonal x).
```

If p is compatible with $\sim$, we can define P(t) on the quotient E/R of $\sim$ by:

$$t \in \mathrm{E/R} \text{ and } (\exists x)(x \in t \text{ and } p\{\!\{x\}\!\}).$$

Criterion C56 says that this is equivalent to

$$t \in \mathrm{E/R} \text{ and } (\forall x)(x \in t \implies p\{\!\{x\}\!\}).$$

It is said to be *induced* by $p\{\!\{x\}\!\}$ on passing to the quotient (with respect to x) with respect to R. If there is $x \in t$ with $p(x)$, then for all $x \in t$ we have $p(x)$. If x is in the substrate, then $p(x)$ is equivalent to P($\bar{x}$) where $\bar{x}$ is the class of x.

```
Definition relation_on_quotient p r :=
  fun t => inc t (quotient r) /\ exists2 x, inc x t & p x.

Lemma rel_on_quoP p r:
  equivalence r -> compatible_with_equiv_p p r -> forall t,
  (relation_on_quotient p r t
    <-> (inc t (quotient r) /\  forall x, inc x t -> p x)).
Lemma rel_on_quoP2 p r:
  equivalence r -> compatible_with_equiv_p p r -> forall y,
  ( (inc y (substrate r) /\ relation_on_quotient p r (Vf (canon_proj r) y))
   <-> (inc y (substrate r) /\ p y)).
```

6.4 Saturated subsets

A subset A of the substrate of a relation r is said *saturated* if $x \in$ A is compatible with r. This is the same as saying that for every $y \in$ A the class of y is a subset of A, or that there exists a set B formed by classes modulo r whose union is A.

```
Definition saturated r x := compatible_with_equiv_p (fun y=> inc y x) r.

Lemma saturatedP r x:
  equivalence r -> sub x (substrate r) ->
  ((saturated r x) <-> (forall y, inc y x -> sub (class r y) x)).
Lemma saturated2P r x:
  equivalence r -> sub x (substrate r) ->
  ((saturated r x) <->
      exists2 y, (forall z, inc z y -> classp r z) & x = union y).
```

Given a function f and a set X, we consider X_f to be $f^{-1}\langle f\langle \mathrm{X}\rangle\rangle$. We have $y \in \mathrm{X}_f$ if and only if there is a $z \in$ X such that $f(y) = f(z)$. If we have an equivalence relation r and f is the

canonical projection onto the quotient set, then $f(y) = f(z)$ is the same as $y \overset{r}{\sim} z$. If X is the singleton $\{x\}$, then X_f is the class of x modulo r. As a consequence X is saturated if and only if $X = X_f$. If X is part of the substrate, it is saturated if and only if it is the inverse image (of some set) by the canonical projection on the quotient set.

```
Definition inverse_direct_value f x :=
  image_by_fun (inverse_fun f) (image_by_fun f x).
Lemma idvalue_P f x: function f -> sub x (source f) ->forall y,
  inc y (inverse_direct_value f x) <->
    (inc y (source f) /\  (exists2 z, inc z x & Vf f y = Vf f z)).
Lemma idvalue_cprojP r x:
   equivalence r -> sub x (substrate r) ->forall y,
  inc y (inverse_direct_value (canon_proj r) x) <->
  (inc y (substrate r) /\ (exists2 z, inc z x & class r y = class r z)).
Lemma class_is_inv_direct_value r x:
  equivalence r -> inc x (substrate r) ->
  class r x = inverse_direct_value (canon_proj r) (singleton x).
Lemma saturated_P3 r x:
  equivalence r -> sub x (substrate r) ->
  (saturated r x <-> (x= inverse_direct_value (canon_proj r) x)).
Lemma saturated_P4 r x:
  equivalence r -> sub x (substrate r) ->
  (saturated r x <->  (exists2 b, sub b (quotient r)
    & x = image_by_fun (inverse_fun (canon_proj r)) b)).
```

The following lemmas show that `saturated` behaves friendly with union, intersection and complement.

```
Lemma saturated_setU r x:
  equivalence r ->
  (alls x (sub^~ (substrate r))) -> (alls x (saturated r)) ->
  (sub (union x) (substrate r) /\  saturated r (union x)).
Lemma saturated_setI r x:
  equivalence r -> nonempty x ->
  (alls x (sub^~ (substrate r))) -> (alls x (saturated r)) ->
  (sub (intersection x) (substrate r) /\  saturated r (intersection x)).
Lemma saturated_setC r a:
  equivalence r -> sub a (substrate r) -> saturated r a ->
  saturated r ((substrate r) -s a).
```

The set X_f is called the saturation of X by r if f is the canonical projection associated to r. It is the union of classes of elements of X. It is the smallest saturated set that contains X. If X_i is a family of sets, A_i their saturations, then the saturation of $\bigcup X_i$ is $\bigcup A_i$.

```
Definition saturation_of r x :=
  inverse_direct_value (canon_proj r) x.
Lemma saturation_of_pr r x:
  equivalence r -> sub x (substrate r) ->
  saturation_of r x =
  union (Zo (quotient r)(fun z=> exists2 i, inc i x & z = class r i)).
Lemma saturation_of_smallest r x:
  equivalence r -> sub x (substrate r) ->
  [/\ saturated r (saturation_of r x),
    sub x (saturation_of r x)
    & (forall y, sub y (substrate r) -> saturated r y -> sub x y
```

```
      -> sub (saturation_of r x) y)].

Definition union_image x f:=
  union (Zo x (fun z=> exists2 i, inc i (source f) & z = Vf f i)).

Lemma saturation_of_union r f g:
  equivalence r -> function f -> function g ->
  (forall i, inc i (source f) -> sub (Vf f i) (substrate r)) ->
  source f = source g ->
  (forall i, inc i (source f) -> saturation_of r (Vf f i) = Vf g i)
  -> saturation_of r (union_image (powerset(substrate r)) f) =
  union_image (powerset(substrate r)) g.
```

6.5 Mappings compatible with equivalence relations

We start with some properties of the function s that maps a non-empty set x to a representative: If R is an equivalence relation, this is a function from E/R to E; it is a section (right inverse) of the canonical projection.

```
Definition section_canon_proj r :=
  Lf rep (quotient r) (substrate r).
Lemma section_canon_proj_axioms r:
  equivalence r ->
  lf_axiom rep (quotient r) (substrate r).
Lemma section_canon_proj_V r x:
  equivalence r ->
  inc x (quotient r) -> Vf (section_canon_proj r)x  = (rep x).
Lemma section_canon_proj_f r:
  equivalence r -> function (section_canon_proj r).
Lemma right_inv_canon_proj r:
  equivalence r ->
  is_right_inverse (canon_proj r) (section_canon_proj r).
```

We say that a function f is *compatible* with R if the relation $f(x) = y$ is compatible; by definition this is: if $x \overset{\mathrm{R}}{\sim} x'$ then $f(x) = y$ implies $f(x') = y$. By symmetry, these two relations are equivalent, and we can eliminate y. We first prove that our definition is the same as the original one, then show that this means that the function is constant on equivalence classes. This means that f can be factored through the canonical projection g (see below; we show here $g(x) = g(y)$ implies $f(x) = f(y)$). (see diagram (retraction/section) on page 67).

```
Definition compatible_with_equiv f r :=
  [/\ function f, source f = substrate r  &
  forall x x', related r x x' -> Vf f x = Vf  f x'].

Lemma compatible_with_equiv_pr f r:
  function f -> source f = substrate r  ->
  compatible_with_equiv f r <->
  (forall y, compatible_with_equiv_p (fun x => y = Vf f x) r)).
Lemma compatible_constant_on_classes f r x y:
  equivalence r ->
  compatible_with_equiv f r -> inc y (class r x) -> Vf f x = Vf f y.
Lemma compatible_constant_on_classes2 f r x:
  equivalence r -> compatible_with_equiv f r ->
  constantfp (restriction f (class r x)).
```

```
Lemma compatible_with_proj f r x y:
  equivalence r -> compatible_with_equiv f r ->
  inc x (substrate r) -> inc y (substrate r) ->
  Vf (canon_proj r) x = Vf (canon_proj r) y -> Vf f x = Vf f y.
```

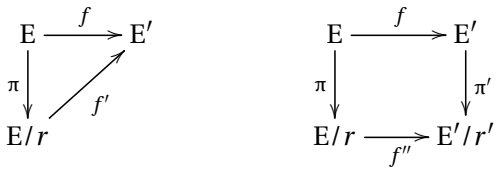
Given two relations r and s, we say that the function f is *compatible* with r and s if $g \circ f$ is compatible with r, when g is the canonical projection of E/s. We can restate this as: $x \overset{r}{\sim} y$ implies $f(x) \overset{s}{\sim} f(y)$. If h is the canonical projection of E/r, then $h(x) = h(y)$ implies that $f(x)$ and $f(y)$ have the same class modulo s.

```
Definition compatible_with_equivs f r r' :=
  [/\ function f, target f =  substrate r' &
  compatible_with_equiv ((canon_proj r') \co f) r].
Lemma compatible_with_pr r r' f x y:
  equivalence r ->  equivalence r' ->
  compatible_with_equivs f r r' ->
  related r x y -> related r' (Vf f x) (Vf f y).
Lemma compatible_with_pr2 r r' f:
  equivalence r ->  equivalence r' ->
  function f ->
  target f = substrate r'-> source f = substrate r->
  (forall x y,  related r x y -> related r' (Vf f x) (Vf f y)) ->
  compatible_with_equivs f r r'.
Lemma compatible_with_proj3 r r' f x y:
  equivalence r -> equivalence r' ->
  compatible_with_equivs f r r'->
  inc x (substrate r) -> inc y (substrate r) ->
  Vf (canon_proj r) x = Vf (canon_proj r) y ->
  class r' (Vf f x) = class r' (Vf f y).
```

Assume that f is compatible with an equivalence r on E, let g be the canonical projection onto E/r and s a section of g. If f is compatible with r, there exists a unique function h such that $h \circ g = f$ and $h = f \circ s$. This mapping is said to be *induced by f on passing to the quotient*. This is criterion C57 (for details, see page 191).

```
Definition fun_on_quotient r f :=
 f \co (section_canon_proj r).

Lemma exists_fun_on_quotient f r:
  equivalence r -> function f -> source f = substrate r ->
  (compatible_with_equiv f r <->
  (exists h, h \coP (canon_proj r) /\ h \co (canon_proj r) = f)).
Lemma exists_unique_fun_on_quotient f r h:
  equivalence r -> compatible_with_equiv f r ->
  h \coP (canon_proj r) -> h \co (canon_proj r) = f ->
  h = fun_on_quotient r f.
Lemma compose_foq_proj f r:
  equivalence r ->  compatible_with_equiv f r ->
  (fun_on_quotient r f) \co (canon_proj r) = f.
```

(fun on quotient)

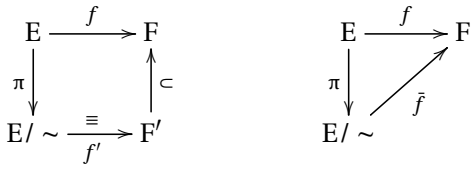
Assume that f is a function from E into E$'$ on which we have equivalence relations r and r'. Let π and π' be the canonical projections onto E$/r$ and E$'/r'$, s and s' associated sections. We can consider $f = f \circ s'$, the mapping induced by f on passing on the quotient, or $f'' = \pi \circ f \circ s$, the mapping induced by f on passing to the quotients with respect to r and s. We consider two cases: f is a mapping, and f is a graph. In order to simplify the statements, we write X and X$'$ instead of `is_equivalence r` or `is_equivalence r'`.

```
Definition fun_on_rep f: Set -> Set := fun x=> f(rep x).
Definition fun_on_reps r' f := fun x=> Vf (canon_proj r')(f(rep x)).
Definition function_on_quotient r f b  :=
  Lf (fun_on_rep f)(quotient r)(b).
Definition function_on_quotients r r' f :=
  Lf (fun_on_reps r' f)(quotient r)(quotient r').
Definition fun_on_quotients r r' f :=
  ((canon_proj r') \co f) \co (section_canon_proj r).
Lemma foq_axioms r f b: X->
  lf_axiom f (substrate r) b ->
  lf_axiom (fun_on_rep f) (quotient r) b.
Lemma foqs_axioms r r' f: X -> X' ->
  lf_axiom f (substrate r)(substrate r') ->
  lf_axiom (fun_on_reps r' f) (quotient r) (quotient r').
Lemma foqc_axioms r f: X->
  function f -> source f = substrate r ->
  f \coP (section_canon_proj r).
Lemma foqcs_axioms r r' f:
  function f -> source f = substrate r -> target f = substrate r' ->
  (canon_proj r' \co f) \coP (section_canon_proj r).

Lemma foq_f r f b: X->
  lf_axiom f (substrate r) b ->
  function (function_on_quotient r f b).
Lemma foqs_f r r' f: X-> X' ->
  lf_axiom f (substrate r)(substrate r') ->
  function (function_on_quotients r r' f).
Lemma foqc_f r f: X-> X' ->
  source f = substrate r ->
  function (fun_on_quotient r f).
Lemma foqcs_f r r' f: X-> X' ->
  function f -> source f = substrate r -> target f = substrate r' ->
  function (fun_on_quotients r r' f).
Lemma foq_V r f b x: X->
  lf_axiom f (substrate r) b ->
  inc x (quotient r) ->
  Vf (function_on_quotient r f b) x = f (rep x).
Lemma foqc_V r f x: X ->
   function f ->
  source f = substrate r -> inc x (quotient r) ->
  Vf (fun_on_quotient r f) c = Vf f (rep x).
Lemma foqs_V r r' f x: X -> X' ->
  lf_axiom f (substrate r)(substrate r') ->   inc x (quotient r) ->
   Vf (function_on_quotients r r' f) x = class r' (f (rep x)).
Lemma foqcs_V r r' f x: X-> X' ->
  function f -> source f = substrate r -> target f = substrate r' ->
  inc x (quotient r) ->
  Vf (fun_on_quotients r r' f) x = class r' (Vf f (rep x)).
```

More lemmas; statement `fun_on_quotient_pr4` is the diagram on the right part of (fun on quotient) on page 119.

```
Lemma fun_on_quotient_pr r f x:
  Vf f x = fun_on_rep (fun _ => Vf f x) (Vf (canon_proj r)x).
Lemma fun_on_quotient_pr2 r r' f x:
  Vf (canon_proj r')  (Vf f x) =
  fun_on_reps r' (fun _ => Vf f x) (Vf (canon_proj r) x).
Lemma composable_fun_proj r f b:
  lf_axiomf f (substrate r) b ->
  (function_on_quotient r f b) \coP (canon_proj r).
Lemma composable_fun_projs r r' f:
  fl_axiomf f (substrate r) (substrate r') ->
  (function_on_quotients r r' f) \coP (canon_proj r).
Lemma composable_fun_projc r f:
  compatible_with_equiv f r ->
  (fun_on_quotient r f) \coP (canon_proj r).
Lemma composable_fun_projcs r r' f:
  compatible_with_equivs f r r'->
  (fun_on_quotients r r' f) \coP (canon_proj r).
Lemma fun_on_quotient_pr3 r f x:
  inc x (substrate r) ->   compatible_with_equiv f r ->
  Vf f x = Vf (fun_on_quotient r f) (Vf  (canon_proj r) x).
Lemma fun_on_quotient_pr4 r r' f:
  compatible_with_equivs f r r'->
  (canon_proj r') \co f = (fun_on_quotients r r' f) \co (canon_proj r).
Lemma fun_on_quotient_pr5 r r' f x:
  compatible_with_equivs f r r'->
  inc x (substrate r) ->
  Vf (canon_proj r') (Vf f x)  =
  Vf (fun_on_quotients r r' f) (Vf (canon_proj r) x).
Lemma compose_fun_proj_ev r f b x:
  compatible_with_equiv (Lf f (substrate r) b) r ->
  inc x (substrate r) ->
  lf_axiom f (substrate r) b ->
  Vf (function_on_quotient r f b \co canon_proj r) x = f x.
Lemma compose_fun_proj_ev2 r r' f x:
  compatible_with_equivs (BL f (substrate r) (substrate r')) r r' ->
  lf_axiom f (substrate r) (substrate r') ->
  inc x (substrate r) ->
  inc (f x) (substrate r') ->
  Vf (canon_proj r') (f x)  =
  Vf (function_on_quotients r r' f \co canon_proj r) x.
Lemma compose_fun_proj_eq r f b:
  compatible_with_equiv (Lf f (substrate r) b) r ->
  lf_axiom f (substrate r) b ->
  (function_on_quotient r f b) \co (canon_proj r) =
    Lf f (substrate r) b.
Lemma compose_fun_proj_eq2 r r' f:
  lf_axiom f (substrate r) (substrate r') ->
  compatible_with_equivs (Lf f (substrate r) (substrate r')) r r'->
  (function_on_quotients r r' f) \co (canon_proj r) =
  (canon_proj r') \co (Lf f (substrate r) (substrate r')).
```



(canonical decomposition)

Assume now that f is a function from E to F, and $\sim$ the associated equivalence, for which x and y are equivalent if $f(x) = f(y)$. Then f is compatible and we can define f on the quotient. If we denote it by $\bar{f}$, and if $\bar{x}$ is the class of x then $\bar{f}(\bar{x}) = f(x)$. From $\bar{f}(\bar{x}) = \bar{f}(\bar{y})$ we get $f(x) = f(y)$, so that x and y are in the same class: hence $\bar{f}$ is injective. If we restrict this function to the image F′ of f we get a bijection, say f'. The diagram (canonical decomposition) says that if we compose the projection π from E to E/ $\sim$, the bijection f' into F′ and the inclusion map from F′ to F, then we get f. If f is surjective then F = F′ and we can simplify a bit: only three arrows are needed. Moreover, there is no need to restrict $\bar{f}$ (this is shown on the right part of the diagram).

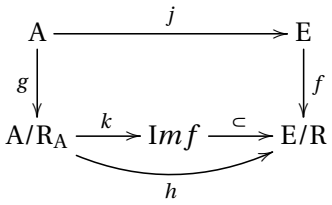
```
Lemma compatible_ea f:
  function f ->
  compatible_with_equiv f (equivalence_associated f).
Lemma ea_foq_fi f:
  function f ->
  injection (fun_on_quotient (equivalence_associated f) f).
Lemma ea_foq_on_im_fb f:
  function f ->
  bijection (restriction2 (fun_on_quotient (equivalence_associated f) f)
    (quotient (equivalence_associated f)) (range (graph f))).
Lemma canonical_decompositiona f (r:= equivalence_associated f):
  function f ->
    function ((restriction2 (fun_on_quotient r f)
      (quotient r) (range (graph f)))
    \co (canon_proj r)).
Lemma canonical_decomposition f  (r:= equivalence_associated f):
  function f ->
  f = (canonical_injection (range (graph f))(target f))
      \co (restriction2 (fun_on_quotient r f) (quotient r) (range (graph f))
           \co (canon_proj r)).
Lemma surjective_pr7 f:
  surjection f ->
  canonical_injection (range (graph f))(target f) = identity (target f).
Lemma canonical_decompositiona f (r:= equivalence_associated f):
  function f ->
    function (compose (restriction2 (fun_on_quotient r f)
      (quotient r) (range (graph f)))
    (canon_proj r)).
Lemma canonical_decomposition_surj f (r:= equivalence_associated f):
  surjection f ->
  f =  (restriction2 (fun_on_quotient r f) (quotient r) (target f))
       \co (canon_proj r).
Lemma canonical_decompositionb f (r:= equivalence_associated f):
  function f ->
    restriction2 (fun_on_quotient r f)  (quotient r) (target f) =
    (fun_on_quotient r f).
Lemma canonical_decomposition_surj2 f (r:= equivalence_associated f):
  surjection f ->
  f = (fun_on_quotient r f) \co (canon_proj r).
```

6.6 Inverse image of an equivalence relation; induced equivalence relation

If ϕ is a function from E to F, S an equivalence on F, and u the canonical projection from F to F/S, the inverse image of S by ϕ is the equivalence R associated to $u \circ \phi$, characterized by $x \overset{R}{\sim} y$ if and only if $\phi(x) \overset{S}{\sim} \phi(y)$. If X is a class modulo S then $\phi^{-1}\langle X\rangle$ is a class modulo R (if nonempty) and conversely.

```
Definition inv_image_relation f r :=
  equivalence_associated (canon_proj r \co f).
Definition iirel_axioms f r :=
  [/\ function f, equivalence r & substrate r = target f].

Lemma iirel_f f r:
  iirel_axioms f r -> function (canon_proj r \co f).
Lemma iirel_relation f r:
  iirel_axioms f r -> equivalence (inv_image_relation f r).
Lemma iirel_substrate f r:
  iirel_axioms f r -> substrate (inv_image_relation f r) = source f.
Lemma iirel_relatedP f r; iirel_axioms f r ->  forall x y,
  (related (inv_image_relation f r) x y <->
   [/\ inc x (source f), inc y (source f) & related r (Vf f x) (Vf f y)]).
Lemma iirel_classP f r: iirel_axioms f r -> forall x,
  (classp (inv_image_relation f r) x <->
 exists y, [/\ classp r y,
    nonempty (y \cap (range (graph f)))
    & x = inv_image_by_fun f y]).
```



(induced equivalence)

Let R be an equivalence on E, A a subset on E, and j the inclusion map $A \to E$. The inverse image of R by j is called the relation *induced* on A and is denoted by R_A. If x and y are in A, then they are related by R_A if and only if they are related by R. Classes for R_A are nonempty sets of the form $A \cap X$ where X is a class for R. The inclusion map is compatible with the relations. Let f and g be the canonical projections and h the function on the quotient. This function is injective, its range is the range of f. Hence h is the composition of a bijection k with the inclusion map.

```
Definition induced_relation r a :=
  inv_image_relation (canonical_injection a (substrate r)) r.
Definition induced_rel_axioms r a :=
   equivalence r /\ sub a (substrate r).
Definition canonical_foq_induced_rel r a :=
  restriction2 (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r)))
  (quotient (induced_relation r a))
  (image_by_fun (canon_proj r) a).
```

```
Section InducedRelation.
Variables (r a: Set).
Hypothesis ira: induced_rel_axioms r a.

Lemma induced_rel_iirel_axioms:
  iirel_axioms (canonical_injection a (substrate r)) r.
Lemma induced_rel_equivalence:
  equivalence (induced_relation r a).
Lemma induced_rel_substrate:
  substrate (induced_relation r a) = a.
Lemma induced_rel_relatedP u v:
  related (induced_relation r a) u v <->
    [/\ inc u a, inc v a & related r u v].
Lemma induced_rel_classP x:
  (classp (induced_relation r a) x <->
  exists y, [/\ classp r y, nonempty (y \cap a) & x = (y \cap a)]).
Lemma compatible_injection_induced_rel:
  compatible_with_equivs (canonical_injection a (substrate r))
  (induced_relation r a) r.
Lemma foq_induced_rel_fi:
  injection (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r))).
Lemma foq_induced_rel_image:
  image_by_fun  (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r))) (quotient (induced_relation r a))
  = image_by_fun (canon_proj r) a.
Definition canonical_foq_induced_rel r a :=
  restriction2 (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r)))
  (quotient (induced_relation r a))
  (image_by_fun (canon_proj r) a).
Lemma canonical_foq_induced_rel_fb:
  bijection (canonical_foq_induced_rel r a).
End InducedRelation.
```

6.7 Quotients of equivalence relations

We say that a relation S is *finer* than R if S implies R. We say that an equivalence r is finer than s if $\overset{s}{\sim}$ implies $\overset{r}{\sim}$, i.e., if for all x and y, $x \overset{s}{\sim} y$ implies $x \overset{r}{\sim} y$. If r and s are equivalences on a same set, this is equivalent to $s \subset r$. If we denote by $\mathrm{C}_s x$ the class of x for s, it is also: for each x, there is an y such that $\mathrm{C}_s x \subset \mathrm{C}_r y$. Equivalently: each $\mathrm{C}_r y$ is saturated by s. We give two examples.

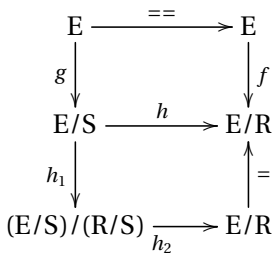
```
Definition finer_equivalence s r:=
  forall x y, related s x y -> related r x y.

Definition finer_axioms s r :=
  [/\ equivalence s,  equivalence r & substrate r = substrate s].
Lemma coarsest_equivalence r:
   equivalence r -> finer_equivalence r (coarse (substrate r)).
Lemma finest_equivalence r:
   equivalence r -> finer_equivalence (diagonal (substrate r)) r.

Section FinerEquivalence.
```

```
Variable (r s: Set).
Hypothesis fa: finer_axioms s r.

Lemma finer_sub_equivP:
  (finer_equivalence s r <-> sub s r).
Lemma finer_sub_equivP2:
  (finer_equivalence s r <->
    (forall x, exists y, sub(class s x)(class r y))).
Lemma finer_sub_equivP3:
  (finer_equivalence s r <->
    forall y, saturated s (class r y)).
```



(quotient of equivalences)

Assume that R and S are two equivalences on E, S finer than R, and let f and g be the canonical projections. Then f is compatible with S. This gives a surjective function h that satisfies $h(\mathrm{C_S}\,x) = \mathrm{C_R}\,x$.

```
Lemma compatible_with_finer:
  finer_equivalence s r ->
  compatible_with_equiv (canon_proj r) s.
Lemma foq_finer_f:
  finer_equivalence s r -> function(fun_on_quotient s (canon_proj r)).
Lemma foq_finer_V x:
  finer_equivalence s r ->  inc x (quotient s) ->
  Vf (fun_on_quotient s (canon_proj r)) x = class r (rep x).
Lemma foq_finer_fs:
  finer_equivalence s r -> surjection (fun_on_quotient s (canon_proj r)).
End FinerEquivalence.
```

On the quotient we can consider the equivalence induced by h. This will be denoted R/S. We have $\mathrm{C_S}\,x \overset{\mathrm{R/S}}{\sim} \mathrm{C_S}\,y$ if and only if $x \overset{\mathrm{R}}{\sim} y$; this is the same as $g(x) \overset{\mathrm{R/S}}{\sim} g(y)$. We have $x \in (\mathrm{E/S})/(\mathrm{R/S})$ if and only if there exists $y \in \mathrm{E/R}$ such that $y = g(y)$. We can consider the canonical decomposition of $h = j \circ h_2 \circ h_1$. Since h is surjective, we can simplify this as $h = h_2 \circ h_1$; here h_1 is the canonical projection of E/S onto (E/S)/(R/S).

```
Definition quotient_of_relations r s :=
  equivalence_associated (fun_on_quotient s (canon_proj r)).

Lemma cqr_aux s x y u:
  equivalence s -> sub y (substrate s) ->
  x = image_by_fun (canon_proj s) y ->
  (inc u x <-> (exists2 v, inc v y & u = class s v)).

Section QuotientRelations.
Variables (r s: Set).
Hypotheses (fa:finer_axioms s r) (fe: finer_equivalence s r).
```

```
Lemma quo_rel_equivalence:
  equivalence (quotient_of_relations r s).
Lemma quo_rel_substrate:
  substrate (quotient_of_relations r s) = (quotient s).
Lemma quo_rel_relatedP x y:
  related (quotient_of_relations r s) x y <->
    [/\ inc x (quotient s), inc y (quotient s) & related r (rep x) (rep y)].
Lemma quo_rel_related_bisP x y:
  inc x (substrate s) -> inc y (substrate s) ->
  ( related (quotient_of_relations r s) (class s x) (class s y)
    <-> related r x y).

Lemma quo_rel_class_bisP x:
  ( inc x (quotient (quotient_of_relations r s)) <->
   exists2 y, inc y (quotient r) & x = image_by_fun (canon_proj s) y).
```

Let S be an equivalence on E and g the canonical projection. Let T be an equivalence on the quotient, and R the inverse image of T by g. This is a relation on E, S is finer than R and R/S is nothing else than T.

```
Lemma quotient_canonical_decomposition
   (f := fun_on_quotient s (canon_proj r))
   (qr := quotient_of_relations r s):
  f =  (fun_on_quotient qr f) \co (canon_proj qr).
End QuotientRelations.

Lemma quotient_of_relations_pr s t
  (r := inv_image_relation (canon_proj s) t):
  equivalence s -> equivalence t -> substrate t = quotient s ->
  t = quotient_of_relations r s.
```

6.8 Product of two equivalence relations

Given two relations R and R′, we can define R × R′ by $(x,x')^{\mathrm{R}\times\mathrm{R}'}_{\sim}(y,y')$ if and only if $x\overset{\mathrm{R}}{\sim}y$ and $x'\overset{\mathrm{R}'}{\sim}y'$. This relation is reflexive, symmetric, antisymmetric, transitive of both relations are. Thus, we get a preorder, an order, or an equivalence from two such relations. If the substrates are E and E′, then the substrate of the product if E × E′ in the these cases.

```
Definition prod_of_relation r r':=
  graph_on
    (fun x y=> inc (J(P x)(P y)) r /\ inc (J(Q x)(Q y)) r')
  ((substrate r) \times (substrate r')).

Lemma order_product2_sr1 f g:
  preorder f -> preorder g ->
  substrate (prod_of_relation f g) = (substrate f) \times (substrate g).
Lemma order_product2_sr f g:
  order f -> order g ->
  substrate (prod_of_relation f g) = (substrate f) \times (substrate g).
Lemma substrate_prod_of_rel r r':
  equivalence r -> equivalence r' ->
  substrate (prod_of_relation r r') =  (substrate r)\times (substrate r')
```

```
Lemma equivalence_prod_of_rel r r':
  equivalence r -> equivalence r' ->
  equivalence (prod_of_relation r r').
Lemma order_product2_preorder f g:
  preorder f -> preorder g -> preorder (prod_of_relation f g).
Lemma order_product2_or f g:
  order f -> order g -> order (prod_of_relation f g).
```

A class in the product is a product of classes.

```
Lemma prod_of_rel_P r r' a b:
  related (prod_of_relation r r') a b <->
  [/\ pairp a, pairp b, related r (P a) (P b) & related r' (Q a) (Q b)].
Lemma related_prod_of_relP1 r r' x x' v :
  related (prod_of_relation r r') (J x x') v <->
  (exists y y', [/\ v = J y y',  related r x y & related r' x' y']).
Lemma related_prod_of_relP2 r r' x x' v:
  related (prod_of_relation r r') (J x x') v <->
    inc v ((im_of_singleton r x) \times (im_of_singleton r' x')).
Lemma class_prod_of_relP2 r r':
  equivalence r -> equivalence r' -> forall x,
  (classp (prod_of_relation r r') x  <->
    exists u v, [/\ classp r u, classp r' v & x = u \times v]).
```

With the same notations, let π and π' be the canonical projections. We can consider the function $\pi \times \pi'$, it maps (x, y) to $(\pi(x), \pi'(x))$: its target is $(E/R) \times (E/R')$. This function is not the canonical projection π'' associated to $R \times R'$, whose target is $(E \times E)/(R \times R')$. However there is a bijection h such that $\pi \times \pi' = h \circ \pi''$.

```
Lemma ext_to_prod_rel_f r r':
  equivalence r -> equivalence r' ->
  function (ext_to_prod(canon_proj r)(canon_proj r')).
Lemma ext_to_prod_rel_V r r' x x':
  equivalence r -> equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  Vf (ext_to_prod(canon_proj r)(canon_proj r')) (J x x') =
  J (class r x) (class r' x').
Lemma compatible_ext_to_prod r r':
  equivalence r -> equivalence r' ->
   compatible_with_equiv (ext_to_prod (canon_proj r) (canon_proj r'))
     (prod_of_relation r r').
Lemma compatible_ext_to_prod_inv r r' x x':
  equivalence r -> equivalence r' ->
  pairp x -> inc (P x) (substrate r) -> inc (Q x)  (substrate r') ->
  pairp x' -> inc (P x') (substrate r) -> inc (Q x')  (substrate r') ->
  Vf (ext_to_prod (canon_proj r) (canon_proj r')) x =
  Vf (ext_to_prod (canon_proj r) (canon_proj r')) x'
  -> related (prod_of_relation r r') x x'.
Lemma related_ext_to_prod_rel r r':
  equivalence r -> equivalence r' ->
  equivalence_associated (ext_to_prod(canon_proj r)(canon_proj r')) =
  prod_of_relation r r'.
Lemma decomposable_ext_to_prod_rel r r': (* 51 *)
  equivalence r -> equivalence r' ->
```

```
exists h,  [/\ bijection h,
  source h = quotient (prod_of_relation r r'),
  target h = (quotient r) \times (quotient r') &
  h \co (canon_proj (prod_of_relation r r')) =
  ext_to_prod(canon_proj r)(canon_proj r')].
```

6.9 Classes of equivalent objects

Let $\sim$ be an equivalence relation; we do not assume that it has a graph. Let θx be the generic object associated to x. In Bourbaki's notation, this is $\tau_y(x \sim y)$. We could implement this via `chooseT`. Assume $x \sim x'$. Then $x \sim y$ and $x' \sim y$ are equivalent, and the properties of τ say $\theta x = \theta x'$. The quantity θx is the class of objects equivalent to x. Bourbaki notes that "$x \sim x$ and $x' \sim x'$ and $\theta x = \theta x'$" is equivalent to $x \sim x'$.

Assume now that there is a set T such that $y \sim y$ implies that there exists $x \in$ T such that $x \sim y$. Let Θ be the set of all θx for $x \in$ T. If $y \sim y$, there exists $x \in$ T such that $x \sim y$, hence $\theta x = \theta y$ and thus $\theta y \in \Theta$. If $x \sim x$, then θx is the unique $z \in \Theta$ such that $x \sim z$.

Assume that $x \sim y$ implies $Ax = Ay$. We can consider the set of all Ax such that $x \sim x$. If f maps t to At, then we have $Ax = f(\theta x)$. Bourbaki says that if we have an equivalence relation on a set E, then we can choose for Ax the class of x, and f becomes a bijection from Θ into the quotient set.

We write θx and Ax instead of $\theta(x)$ and $A(x)$ in order to emphasize the fact that these objects are not functions. However, θx is a set. No code is associated to this section. It seems that this section is not used in the remaining of the work of Bourbaki; for instance, if we consider the relation X is equipotent to Y, then θX is the cardinal of X. Bourbaki proves the existence of the cardinal by repeating the arguments previously exposed in this section.

Chapter 7

Exercises

We start with some properties of the Theory of Sets, not used elsewhere. We show that $\mathrm{Coll}_y(y \notin y)$ is false. This implies that there is no set x such that for all y we have $y \in x$, but on the contrary, there is a set x such that for no y we have $y \in x$ (this being the empty set). Then we show that for every property p, we have $p(x)$, provided that $x \in \emptyset$.

Assume that there is a set x such that $y \in x$ is equivalent to $y \notin y$. Let q be the property $x \in x$. By definition, q is equivalent to its negation. Thus we have p that says $q \implies \neg q$ and p' that says $\neg q \implies q$. If h is a proof of q then $p(h)$ is a proof of $\neg q$ and $p(h,h)$ is false. Thus $\mathrm{H} : h \mapsto p(h,h)$ is a proof that q is false, and $p'(\mathrm{H})$ is a proof that q is true. Thus $\mathrm{H}(p'(\mathrm{H}))$ is a proof of false.

```
Lemma not_collectivizing_notin:
  ~ (exists z, forall y, inc y z <-> not (inc y y)).
Proof.
case=> x hx; move: (hx x) => [p p'].
pose H:=  (fun h : inc x x => (p h h));exact (H (p' H)).
Qed.

Lemma collectivizing_special :
  (exists x, forall y, ~ (inc y x)) /\  ~ (exists x, forall y, inc y x).
Proof.
split; first by exists emptyset;  apply: in_set0.
move=> [x Px]; apply: not_collectivizing_notin.
exists (Zo x (fun z => ~ (inc z z))) => z.
by split;[ case /Zo_P | move => zz; apply:Zo_i].
Qed.

Lemma emptyset_pra x (p: property):
  inc x emptyset -> (p x).
Proof. case;case. Qed.
```

The two objects `False` and `True` have type `Prop`, but can be considered as sets. They have exactly zero and one element; in particular, `False` is equal to the emptyset.

```
Lemma rel_False: emptyset = False.
Proof.
apply: extensionality.
  by move => t /in_set0.
move => t; case; case.
Qed.
```

```
Lemma rel_True: singleton (Ro I) = True.
apply: extensionality.
  move => t /set1_P ->; apply: R_inc.
move => t [a <-]; apply /set1_P.
have -> //: a = I by case:a.
Qed.
```

7.1 Section 1

1. *Show that the relation* $(x = y) \iff (\forall X)((x \in X) \implies (y \in X))$ *is a theorem.*

Comment. In Bourbaki, you can prove $x = x$ (this is the first theorem) or $(\forall x)(x = x)$ (this is different theorem). In Coq, we can prove only the second property. We try to be as close as possible to the Bourbaki statement by using a section. The quantifiers are still present, but invisible. This looks like the axiom of extent for the relation $\ni$; implication $\Rightarrow$ is trivial; implication $\Leftarrow$ is a consequence of a weaker property, where we restrict X to be a singleton, which reads then: $(\forall z)((x = z) \implies (y = z))$.

```
Section exercise1_1.
Variable x y:Set.

Lemma exercise1_1: (x=y) <-> (forall X, inc x X -> inc y X).
Proof.
split; first  by  move=> ->.
by move=> spec_sub; symmetry; apply: set1_eq; apply: spec_sub; fprops.
Qed.
End exercise1_1.
```

2. *Show that* $\emptyset \neq \{x\}$ *is a theorem. Deduce that* $(\exists x)(\exists y)(x \neq y)$ *is a theorem.*

Comment. The first claim is really $(\forall x)(\emptyset \neq \{x\})$. Note that the "axiom of the singleton" (for each x there is a set that has a unique element, namely x) asserts that the number of sets is not finite.

```
Lemma exercise1_2: exists x y:Set, x <> y.
Proof.
have theorem:forall x:Set, emptyset <> singleton x.
  by move=> x esx; empty_tac1 x.
by exists emptyset; exists (singleton emptyset); apply: theorem.
Qed.
```

3. *Let* A *and* B *be two subsets of a set* X*. Show that the relation* $B \subset \complement A$ *is equivalent to* $A \subset \complement B$ *and that the relation* $\complement B \subset A$ *is equivalent to* $\complement A \subset B$*.*

Comment. The notation $\complement A$ is an abuse of language for $X - A$. It suffices to prove one implication, all other follow. We give here a different proof: we show that $B \subset \complement A$ is equivalent to $A \cap B = \emptyset$ and $\complement A \subset B$ is equivalent to $A \cup B = X$. The result follows by commutativity of union and intersection.

```
Lemma exercise1_3 X A B: sub A X -> sub B X ->
  ((sub (X -s B) A <-> sub (X -s A) B)     /\
   (sub B (X -s A) <-> sub A (X -s B))).
```

```
Proof.
have aux1: forall a b, sub a X -> sub b X ->
   (sub (compl a) b <-> a \cup b = X).
  move => a b aX bX; split.
    move => s1; set_extens t; first by case /setU2_P => ts; fprops.
    rewrite - (setU2_Cr aX); case /setU2_P => ts; fprops.
  by rewrite /compl => <- t /setC_P [/setU2_P] [].
have aux2: forall a b, sub a X -> sub b X ->
   (sub b (compl a)  <-> a \cap b = emptyset).
  move => a b aX bX; split.
    move => s1; apply /set0_P =>t /setI2_P [ta tb].
    by move /setC_P: (s1 _ tb) => [].
  move => abe t tb; apply /setC_P; split; fprops.
  move => ta; empty_tac1 t.
move => ax bx; split.
   apply: (iff_trans (aux1 _ _ bx ax)); rewrite setU2_C.
   by apply: iff_sym; apply/aux1.
apply: (iff_trans (aux2 _ _ ax bx)); rewrite setI2_C.
by apply: iff_sym; apply/aux2.
```

4. *Prove that the relation* $X \subset \{x\}$ *is equivalent to "*$X = \{x\}$ *or* $X = \emptyset$*".*

Comment. This has been proved in the main text. If X is a non-empty subset of $\{x\}$, and z is in X, then $z = x$.

```
Lemma exercise1_4 X x:
  sub X (singleton x) <-> (X = singleton x \/ X = emptyset).
Proof.
split; last by case => ->; fprops.
move => asx; case (emptyset_dichot X); first by right.
by move => nea; left; apply: set1_pr1 => // z /asx /set1_P.
Qed.
```

5. *Prove that* $\emptyset = \tau_X(\tau_x(x \in X) \notin X)$.

Comment. We shall give a proof that uses `choose`, which not exactly the same as Bourbaki's τ function. Hence we start, informally, with a Bourbaki proof. We have to show

$$\tau_X(\neg(\exists x)(\neg(x \notin X))) = \tau_X(\neg(\exists x)(x \in X)),$$

(by definition of $\emptyset$, $\forall$ and $\exists$). Write this as $\tau_X(\neg(\exists x)P) = \tau_X(\neg(\exists x)Q)$. According to Scheme S7, it suffices to prove $(\forall X)(\neg(\exists x)P \iff \neg(\exists x)Q)$. Fix X. Criterion C24 says that $\neg x \notin X$ is equivalent to $x \in X$, i.e., $P \iff Q$. From Criterion C31 it follows that $(\exists x)P \iff (\exists x)Q$. Criterion C23 implies $\neg(\exists x)P \iff \neg(\exists x)Q$. Qed.

The expression $\tau_x(x \in X)$ is denoted by `rep` in Coq. Write this as $r(X)$. From $y \in Y$, it follows $r(Y) \in Y$. Let $p(X)$ stand for $r(X) \notin X$. By double negation, if $r(Y)$ is true, then Y must be empty. We must show that $Y = \tau_X p$ is empty; it suffices to prove $p(\tau_X p)$, which follows from $p(\emptyset)$.

```
Lemma  exercise1_5:
  emptyset = choose (fun X => ~ (inc (rep X) X)).
Proof.
have rep_pr: forall Y y, inc y Y -> inc (rep Y) Y.
  by  move=> Y y yY;  apply: (choose_pr (p:=inc^~ Y)); exists y.
```

```
have Ye: forall Y,  ~ (inc (rep Y) Y) -> emptyset = Y.
  move => y ye; symmetry.
  by apply /set0_P; move=> t; dneg aux; apply (rep_pr _ _ aux).
apply: Ye; apply: (choose_pr (p:= fun z => ~  inc (rep z) z)).
exists emptyset; case; case.
Qed.
```

We give here a shorter proof. Note that `in_set0` say that no element is in the empty set, and `rep_i` says `rep x` is in x if x is non-empty.

```
Lemma exercise1_5:
  emptyset = choose (fun X => ~ (inc (rep X) X)).
Proof.
pose p := fun z => ~  inc (rep z) z.
have pe: p emptyset by exact: in_set0.
move:(choose_pr (ex_intro p emptyset pe)) => pcp.
case (emptyset_dichot (choose p)) => // ney; by move: (rep_i ney).
Qed.
```

6. *Consider* $(\forall y)(y = \tau_x((\forall z)(z \in x \iff z \in y)))$. *Show that this axiom* A1′ *implies the axiom of extent* A1.

Comment. We introduce an axiom, equivalent to Axiom Scheme S7, that says that if P and Q are equivalent propositions, then $\tau_x P = \tau_x Q$. Write $R(x,y)$ for $(\forall z)(z \in x \iff z \in y)$. Let A and B be two sets such that $A \subset B$ and $B \subset A$, so that R(A,B) holds. It follows, by transitivity of equivalence, that for all x, $R(x,A)$ is equivalent to $R(x,B)$ so that (S7) gives $\tau_x R(x,A) = \tau_x R(x,B)$. Axiom A1′ says $A = \tau_x R(x,A)$ and $B = \tau_x R(x,B)$. If follows $A = B$.

```
Section Ex1_6.
Hypothesis choose_equiv: forall (p q: property),
  (forall x, p x <-> q x) -> choose p = choose q.

Lemma exercise1_6:
  (forall y, y = choose (fun x => (forall z, (inc z x)<->(inc z y))))
  -> (forall a b : Set, sub a b -> sub b a -> a = b).
Proof.
move=> hyp a b; rewrite /sub => sab sba.
rewrite (hyp a) (hyp b).
apply: choose_equiv;  move=> x.
split; move=> aux z; rewrite aux; split;  auto.
Qed.

End Ex1_6.
```

7.2 Section 2

1. *Let* $\boldsymbol{R\{x,y\}}$ *be a relation, the letters* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *being distinct; let* $\boldsymbol{z}$ *be a letter distinct from* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *which does not appear in* $\boldsymbol{R\{x,y\}}$. *Show that the relation* $(\exists \boldsymbol{x})(\exists \boldsymbol{y})\boldsymbol{R\{x,y\}}$ *is equivalent to*

$$(\exists \boldsymbol{z})(\boldsymbol{z} \text{ is an ordered pair and } \boldsymbol{R}\{\mathrm{pr}_1 \boldsymbol{z}, \mathrm{pr}_2 \boldsymbol{z}\})$$

and the relation $(\forall \boldsymbol{x})(\forall \boldsymbol{y})\boldsymbol{R\{x,y\}}$ *is equivalent to*

$$(\forall \boldsymbol{z})(\boldsymbol{z} \text{ is an ordered pair } \implies (\boldsymbol{R}\{\mathrm{pr}_1 \boldsymbol{z}, \mathrm{pr}_2 \boldsymbol{z}\})).$$

Comment. Compare this with the section "Function of two variables".

```
Lemma exercise2_1 (R: relation):
  ((exists x, exists y, R x y) <-> (exists z, pairp z /\ R(P z) (Q z))) /\
  ((forall x, forall y, R x y) <-> (forall z, pairp z -> R(P z) (Q z))).
Proof.
split;split.
- move=> [x] [y] Rxy;  exists (J x y); aw;fprops.
- by move => [z [zp Rz]]; exists (P z); exists (Q z).
- move=> hyp z _; apply: hyp.
- move => hyp x y; move: (hyp _ (pair_is_pair x y)); aw.
Qed.
```

2. *(a) Show that the relation* $\{\{x\},\{x,y\}\} = \{\{x'\},\{x',y'\}\}$ *is equivalent to* $x = x'$ *and* $y = y'$*.*
(b) Let $\mathscr{T}_0$ *be the theory of sets, and let* $\mathscr{T}_1$ *be the theory which has the same schemes and explicit axioms as* $\mathscr{T}_0$*, except for the axiom A3. Show that if* $\mathscr{T}_1$ *is not contradictory, then* $\mathscr{T}_0$ *is not contradictory.*

Comment. In the French version, Bourbaki defines the pair (x,y) as $\{\{x\},\{x,y\}\}$ and proves (a) as Proposition 1 (thus Propositions in this section are numbered differently in the two editions). In the English version, there is a specific sign (that looks a bit like $\supset$) that defines a pair, and an axiom A3. Part (b) of the exercise is then: if the French version is not contradictory, then the English version is neither.

```
Definition xpair (x y : Set) :=
  doubleton (singleton x) (doubleton x (singleton y)).

Lemma exercise2_2 x y z w:
  (xpair x y = xpair z w) <-> (x = z /\ y = w).
Proof.
split; last by  move=> [] -> ->.
move => eq.
have fp2: inc (singleton x) (xpair z w) by  rewrite -eq /xpair; fprops.
have sp2: inc (doubleton x (singleton y)) (xpair z w).
  by rewrite -eq /xpair; fprops.
have xz: x=z.
   case /set2_P: fp2; first by apply: set1_inj.
   by move=> sd; symmetry; apply: set1_eq;  ue.
split=>//.
rewrite xz in sp2.
case /set2P:sp2 => hyp.
  symmetry.
  have syz: (singleton y = z) by apply: set1_eq; ue.
  have: (inc (doubleton z (singleton w)) (xpair x y)).
    by rewrite eq /xpair; fprops.
  rewrite xz /xpair hyp; move/set1_P => zwz.
  have: (singleton w = z) by apply: set1_eq; ue.
  by rewrite - syz; apply: set1_inj.
apply: set1_inj.
have sp3:  (inc (singleton w)  (doubleton z (singleton y))) by  ue.
case /set2_P: sp3 => sp4; last by symmetry.
have sp5: (inc (singleton y)  (doubleton z (singleton w))) by ue.
by case /set2_P: sp5; try ue.
Qed.
```

7.3 Section 3

1. *Show that the relations $x \in y$, $x \subset y$, $x = \{y\}$ have no graph with respect to x and y.*

Assume that r is a relation, and G is a set containing all related pairs. Then r has a graph, namely the subset of all elements (x, y) of G that are related. We replace "has no graph" by "there is no such G". We say that r is "universal" if any x is related to some y.

```
(* Definition has_no_graph (r:relation):=
   ~(exists G, is_graph G /\ forall x y, r x y <-> inc (J x y) G). *)
Definition has_no_graph (r:relation):=
 ~(exists G, forall x y, r x y -> inc (J x y) G).
Definition is_universal (r:relation):=
  forall x, exists y, r x y \/ r y x.
```

Assume that r is a universal relation, and $r(x, y)$ implies $J(x, y) \in X$. Let D be the union of the domain of range of X. The relation $r(x, y)$ implies that both x and y are in D. Since r is universal, every set is in D, absurd.

```
Lemma is_universal_pr r:  is_universal r -> has_no_graph r.
Proof.
move=> u [X h].
case: (proj2 collectivizing_special).
exists ((domain X) \cup (range X)).
move: (u y) => [x [] /h jg]; apply /setU2_P; [left | right];ex_tac.
Qed.
```

The result is now trivial.

```
emma exercise3_1:
  [/\  has_no_graph (fun x y => inc x y),
       has_no_graph (fun x y => sub x y) &
       has_no_graph (fun x y => x = singleton y) ].
Proof.
split; apply: is_universal_pr; move=> x;
  [ exists (singleton x) | exists x |  exists (singleton x) ] ; fprops.
Qed.
```

2. *Let* G *be a graph. Show that the relation* $X \subset \mathrm{pr}_1 G$ *is equivalent to* $X \subset G^{-1}\langle G\langle X\rangle\rangle$.

```
Lemma exercise3_2 G X: sgraph G ->
  ( sub X (domain G) <->
    sub X (direct_image (inverse_graph G) (direct_image G X))).
Proof.
move=>G X gG.
split; move=> hyp t ts; move: (hyp _ ts).
  move/(domainP gG)=> [y Jg]; apply/dirim_P;exists y.
    apply/dirim_P; ex_tac.
  by apply/igraph_pP.
move/dirim_P => [x _] /igraph_pP h; ex_tac.
Qed.
```

3. *Let* G, H *be two graphs. Show that the relation* $\mathrm{pr}_1 H \subset \mathrm{pr}_1 G$ *is equivalent to* $H \subset H \circ G^{-1} \circ G$. *Deduce that* $G \subset G \circ G^{-1} \circ G$.

```
Lemma exercise3_3a G H: sgraph G -> sgraph H ->
  ( sub (domain H) (domain G) <->
    sub H (H \cg ((inverse_graph G) \cg G))).
Proof.
move=> gG gH.
split => h t ts.
  move: (gH _ ts) => Jt; rewrite - Jt  in ts.
  have: (inc (P t) (domain G)) by apply: h; ex_tac.
  move /(domainP gG)=> [y JG]; apply /compg_P; split => //; ex_tac.
  by apply /compg_pP; ex_tac; apply/igraph_pP.
move /(domainP gH): ts => [y JH].
move: (h _ JH) => /compg_pP [z /compg_pP [u q _] _]; ex_tac.
Qed.


Lemma exercise3_3b G: sgraph G ->
  sub G (G \cg ((inverse_graph G) \cg G)).
Proof. move=> gG; apply/(exercise3_3a gG gG); fprops. Qed.
```

4. *If* G *is a graph show that* $\emptyset \circ G = G \circ \emptyset = \emptyset$ *and that* $G^{-1} \circ G = \emptyset$ *if and only if* $G = \emptyset$.

For the first two relations, we need not G be a graph.

```
Lemma exercise3_4a G:
  (G \cg emptyset = emptyset /\
   emptyset \cg G = emptyset).
Proof.
split; apply /set0_P => x /compg_P [_].
  by move => [y /in_set0].
by move => [y _ /in_set0].
Qed.


Lemma exercise3_4b G: sgraph G ->
  ((inverse_graph G) \cg G = emptyset <-> G = emptyset).
Proof.
move=> gG; split => h; last by rewrite h; apply: (proj1 (exercise3_4a _)).
apply /set0_P => x xG; empty_tac1 (J (P x) (P x)).
move:(eq_ind_r (inc^~ G) xG (gG x xG)) => px.
by apply/compg_pP; exists (Q x) => //; apply /igraph_pP.
Qed.
```

5. *Let* A, B *be two sets,* G *a graph.*
Show that $(A \times B) \circ G = G^{-1}\langle A\rangle \times B$ *and* $G \circ (A \times B) = A \times G\langle B\rangle$.

Comment. Note that G need not be a graph here.

```
Lemma exercise3_5 G A B:
  ((A \times B) \cg G = (inverse_image G A) \times B /\
   G \cg (A \times B)  = A \times (direct_image G B)).
Proof.
split; set_extens x.
- move /compg_P => [px [y yG /setXp_P [pa pb]]].
  apply/setX_P;split => //; apply/iim_graph_P; ex_tac.
- move /setX_P => [px /iim_graph_P [y uA JG] QB].
  apply /compg_P; split => //; ex_tac; fprops.
```

```
- move /compg_P => [px [y /setXp_P [pa pb pc]]].
  apply /setX_P;split => //; apply/dirim_P; ex_tac.
- move  /setX_P => [px pxa /dirim_P [y ya yb]].
  apply/compg_P; split => //; ex_tac; fprops.
Qed.
```

6. *For each graph* G *let* G′ *be the graph* $(\mathrm{pr}_1 G \times \mathrm{pr}_2 G) - G$. *Show that* $(G^{-1})' = (G')^{-1}$, *and that* $G \circ (G^{-1})' \subset \Delta'_B$, $(G^{-1})' \circ G \subset \Delta'_A$, *if* $A \supset \mathrm{pr}_1 G$ *and* $B \supset \mathrm{pr}_2 G$. *Show that* $G = (\mathrm{pr}_1 G) \times (\mathrm{pr}_2 G)$ *if and only if* $G \circ (G^{-1})' \circ G = \emptyset$.

```
Definition complement_graph G :=
  ((domain G) \times (range G)) -s G.

Definition commutes_at (f g: Set -> Set) x:= f (g x) = g (f x).

Lemma complement_graph_g G: sgraph (complement_graph G).
Proof. by  move => t /setC_P [] /setX_P [ok _] _. Qed.


Lemma exercise3_6a G: sgraph G -> commutes_at complement_graph inverse_graph G.
Proof.
move => gG.
have gc: sgraph (complement_graph G) by apply: complement_graph_g.
rewrite /commutes_at/complement_graph (igraph_range gG)(igraph_domain gG).
set_extens t.
  move /setC_P =>  [/setX_P [pt pa pb] pc].
  apply /igraphP; split => //; apply/ setC_P;split; first by fprops.
  by move /igraph_pP; rewrite pt.
move /igraphP => [px /setC_P [/setXp_P [pa pb] pc]].
apply/setC_P; rewrite - px; split; [ fprops | by  move /igraph_pP].
Qed.

Lemma exercise3_6b G B: sgraph G -> sub (range G) B ->
  sub (G \cg (complement_graph (inverse_graph G)))
      (complement_graph (diagonal B)).
Proof.
move=> gG srB; rewrite exercise3_6a // => t.
move /compg_P => [pt [y /igraph_pP /setC_P [/setXp_P [pa pb] pc] pd]].
apply/setC_P; split; last by move /diagonal_i_P => [_ _ eq]; case pc; ue.
move:(@identity_sgraph B); rewrite - diagonal_is_identity => aux.
apply /setX_i => //.
  apply/(domainP aux); exists(P t); apply /diagonal_pi_P; fprops.
apply/(rangeP aux);  exists(Q t);apply /diagonal_pi_P.
split => //; apply: srB; ex_tac.
Qed.

Lemma exercise3_6c A G: sgraph G -> sub (domain G) A ->
  sub ((complement_graph (inverse_graph G)) \cg G)
      (complement_graph (diagonal A)).
Proof.
move=> gG sd.
rewrite (exercise3_6a gG) => t.
move /compg_P => [pt [y pa /igraph_pP /setC_P [/setXp_P [pb pc] pd]]].
apply/setC_P; split; last by move /diagonal_i_P => [_ _ eq];case pd; ue.
move:(@identity_sgraph A); rewrite - diagonal_is_identity => aux.
apply /setX_i => //.
```

```
   apply/(domainP aux); exists(P t); apply /diagonal_pi_P.
  split => //; apply: sd; ex_tac.
apply/(rangeP aux);  exists(Q t); apply /diagonal_pi_P; fprops.
Qed.

Lemma exercise3_6d G: sgraph G ->
  ( G = (domain G) \times (range G) <->
    G \cg ((complement_graph (inverse_graph G)) \cg G)
    = emptyset ).
Proof.
move=> gG; rewrite  (exercise3_6a gG).
set (K:= complement_graph G).
transitivity (K = emptyset).
  rewrite /K /complement_graph; split.
  move => <-; apply setC_v.
  move => h; move:(empty_setC h) => aux; apply: extensionality => //.
  apply: (sub_graph_setX gG).
split.
  move=> ->; rewrite igraph0.
  by move: (exercise3_4a G) => [p1 p2]; rewrite p2 p1.
move=> ce; apply /set0_P => x xK.
move: (xK); move /setC_P => [] /setX_P [pa]
     /(domainP gG) [u J1G] /(rangeP gG) [v J2G] _.
empty_tac1 (J v u); apply /compg_pP; ex_tac; apply /compg_pP; ex_tac.
by apply/igraph_pP; rewrite pa.
Qed.
```

7. *A graph* G *is functional if and only if for each set* X *we have* $G\langle G^{-1}\langle X\rangle\rangle \subset X$.

```
Lemma exercise3_7 G: sgraph G ->
  (fgraph G <-> forall X, sub (direct_image G (inverse_image G X)) X)).
Proof.
move=>gG; split.
  move=> fgG X x /dirim_P [y /iim_graph_P [u ux pug] pxg].
  by rewrite (fgraph_pr fgG pxg pug).
move=> hyp; split =>// x y xG yG sP.
move:(gG _ xG) (gG _ yG)=> px py.
apply: pair_exten=>//; apply: set1_eq.
apply: (hyp (singleton (Q y))).
apply/dirim_P; exists (P x); last by rewrite px.
by apply /iim_graph_P;  exists (Q y); fprops; rewrite sP py.
Qed.
```

8. *Let* A, B *be two sets, let* Γ *be a correspondence between* A *and* B, *and let* Γ' *be a correspondence between* B *and* A. *Show that if* $\Gamma'(\Gamma(x)) = \{x\}$ *for all* $x \in A$ *and* $\Gamma(\Gamma'(y)) = \{y\}$ *for all* $y \in B$, *then* Γ *is a bijection of* A *onto* B *and* Γ' *is the inverse mapping.*

Comment. There is an abuse of notation here (see exercise 11). In some cases $\Gamma(x)$ denotes $\Gamma\langle\{x\}\rangle$ and sometimes $\Gamma(X)$ denotes $\Gamma\langle X\rangle$. The proof is a bit longish. In the comments, G and G′ are the graphs.

```
Lemma exercise3_8 G G': correspondence G -> correspondence G' ->
  source G = target G' -> source G' = target G ->
  (forall x, inc x (source G) -> image_by_fun G' (image_by_fun G (singleton x))
    = singleton x) ->
```

```
  (forall x, inc x (source G') -> image_by_fun G (image_by_fun G'(singleton x))
    = singleton x) ->
  [/\ bijection G, bijection G' & G = inverse_fun G'].
Proof.
rewrite /image_by_fun=>  cG cG' sG sG' G'Gx GG'x.
have gG: sgraph (graph G) by fprops.
have gG': sgraph (graph G') by fprops.
```

If $x \in$ A then x is in the domain of G (since $\Gamma'(\Gamma(x))$ is not empty). Same with G and G$'$ exchanged.

```
have sGdgG: source G = domain (graph G).
  apply: extensionality; last by fprops.
  move=> x xs; move: (set1_1 x); rewrite - (G'Gx _ xs).
  move /dirim_P => [y] /dirim_P [t /set1_P -> aa _]; ex_tac.
have sGdgG': source G' = domain (graph G').
  apply: extensionality; last by fprops.
  move=> x xs; move: (set1_1 x); rewrite - (GG'x _ xs).
  move /dirim_P => [y] /dirim_P [t /set1_P -> aa _]; ex_tac.
```

We show $(x, y) \in$ G and $(y, z) \in$ G$'$ implies $x = z$; same with G and G$'$ exchanged.

```
have JGG':forall x y z, inc (J x y)(graph G) -> inc (J y z)(graph G') -> x = z.
  move=> x y z Jxy Jyz.
  have xG: inc x (source G) by rewrite sGdgG; ex_tac.
  symmetry; apply: set1_eq.
  rewrite - (G'Gx _ xG); apply /dirim_P; ex_tac;  apply /dirim_P; ex_tac.
  fprops.
have JG'G:forall x y z, inc (J x y)(graph G') -> inc (J y z)(graph G) -> x = z.
  move=> x y z Jxy Jyz.
  have xG: inc x (source G') by rewrite sGdgG'; ex_tac.
  symmetry; apply: set1_eq.
  rewrite - (GG'x _ xG); apply /dirim_P; ex_tac;  apply /dirim_P; ex_tac.
  fprops.
```

We show: if $x \in$ A there is a y such that $(x, y) \in$ G and $(y, x) \in$ G$'$.

```
have xGy:  (forall x, inc x (source G) -> exists2 y,
    inc (J x y) (graph G) & inc (J y x) (graph G')).
  move=> x xsG; move: (set1_1 x).
  rewrite - (G'Gx _ xsG); move /dirim_P => [y /dirim_P [z /set1_P -> pb pc]].
  ex_tac.
have xG'y:  (forall x, inc x (source G') -> exists2 y,
    inc (J x y) (graph G') & inc (J y x) (graph G)).
  move=> x xsG;  move: (set1_1 x).
  rewrite - (GG'x _ xsG); move /dirim_P => [y /dirim_P [z /set1_P -> pb pc]].
  ex_tac.
```

We show $(x, y) \in$ G and $(x, z) \in$ G implies $y = z$.

```
have fgG: fgraph (graph G).
  split=>//; move=> x y xG yG Pxy.
  have px: pairp x by apply: gG.
  have py: pairp y by apply: gG.
  apply: pair_extensionality =>//.
```

```
  rewrite  - px in xG.
  rewrite  - py -Pxy in yG.
  have Pxs: inc (P x) (source G) by rewrite sGdgG;  ex_tac.
  move: (xGy _ Pxs) => [z _ J2g].
  rewrite -  (JG'G _ _ _ J2g xG).
  by rewrite - (JG'G _ _ _ J2g yG).
have fgG': fgraph (graph G').
  split=>//; move=> x y xG yG Pxy.
  have px: pairp x by apply: gG'.
  have py: pairp y by apply: gG'.
  apply: pair_extensionality =>//.
  rewrite  - px in xG.
  rewrite  - py -Pxy in yG.
  have Pxs: inc (P x) (source G') by rewrite sGdgG';  ex_tac.
  move: (xG'y _ Pxs) => [z _ J2g].
  rewrite -  (JGG' _ _ _ J2g xG).
  by rewrite - (JGG' _ _ _ J2g yG).
```

We show $(x, y) \in G$ and $(y, x) \in G'$ are equivalent.

```
have fg: function G by [].
have fg': function G' by [].
have GiG:  (graph G = inverse_graph(graph G')).
  set_extens x xs.
    have px: pairp x by apply: gG.
    rewrite - px in xs |- *; apply/igraph_pP.
    have Ps: inc (P x) (source G) by rewrite sGdgG; ex_tac.
    move: (xGy _ Ps)=> [y J1 J2].
    by rewrite -(JG'G _ _ _ J2 xs).
  have gi: (sgraph (inverse_graph (graph G'))) by fprops.
  have px: pairp x by apply: gi.
  move: xs; rewrite - px; ;move/igraph_pP => xs; rewrite -px.
  have Ps: inc (P x) (source G) .
    by rewrite  sG; apply: corresp_sub_range=>//; ex_tac.
  move: (xGy _ Ps)=> [y J1 J2].
  by aw;rewrite (JG'G _ _ _  xs J1).
have GiG2: (G = inverse_fun G').
  rewrite /inverse_fun - sG sG' -GiG.
  by symmetry; apply: corresp_recov1.
```

Bijectivity of Γ is easy.

```
have bG: bijection G.
  split.
    split=>//; move=> x y xs ys sW.
    move: (Vf_pr3 fg xs) => HGx.
    move: (Vf_pr3 fg ys) => HGy; rewrite - sW in HGy.
    have Ws: inc (Vf G x) (source G') by rewrite sG';  fprops.
    move: (xG'y _ Ws) => [z J1 J2].
    by rewrite (JGG' _ _ _ HGx J1) (JGG' _ _ _ HGy J1).
  apply: surjective_pr5 =>// x.
  rewrite - sG' => xs.
  move: (xG'y _ xs) =>  [z J1 J2].
  rewrite /related; ex_tac; apply: (p1graph_source fg J2).
have GiG3: G' = inverse_fun G by rewrite GiG2 ifun_involutive.
by split => //; rewrite GiG3; apply: inverse_bij_fb.
Qed.
```

9. *Let* A, B, C, D *be sets,* f *a mapping of* A *into* B, g *a mapping of* B *into* C, h *a mapping of* C *into* D. *If* $g \circ f$ *and* $h \circ g$ *are bijections, show that all of* f, g, h *are bijections.*

```
Lemma exercise3_9 f g h:
  function f -> function g -> function h->
  source g = target f -> source h = target g ->
  bijection (g \co f) -> bijection (h \co g) ->
  [/\ bijection f, bijection g & bijection h].
Proof.
move=> ff fg fh sgtf shtg bgf bhg.
have cgf : g \coP f by [].
have chg : h \coP g by [].
have ig: injection g.
  by move: bhg=>[ia sa]; apply: (right_compose_fi chg ia).
have sg: surjection g.
  by move: bgf=>[ia sa]; apply: (left_compose_fs cgf sa).
have bg: bijection g by split.
split => //.
  apply: (right_compose_fb cgf bgf bg).
apply: (left_compose_fb chg  bhg bg).
Qed.
```

10. *Let* A, B, C *be sets,* f *a mapping of* A *into* B, g *a mapping of* B *into* C, h *a mapping of* C *into* A. *Show that if two of the three mappings* $h \circ g \circ f$, $g \circ f \circ h$, $f \circ h \circ g$ *are surjections and the third is an injection, then* f, g, h *are all bijections.*

The French version claims that the same conclusion holds if two of the three mappings are injections and the third is a surjection. We assume here $h \circ g \circ f$ injective, $g \circ f \circ h$ surjective and $f \circ h \circ g$ injective or surjective. Other cases are equivalent, by renaming variables.

```
Lemma exercise3_10 f g h:
  function f -> function g -> function h->
  source g = target f -> source h = target g ->  source f = target h ->
  injection (h \co (g \co f)) ->
  surjection (g \co  (f \co h)) ->
  (injection (f \co (h \co g))
   \/ surjection (f \co (h \co g))) ->
  [/\ bijection f, bijection g & bijection h].
Proof.
move=> ff fg fh sgtf shtg sfth ihgf sgfh is_fgh.
have cfh: f \coP h by [].
have chg: h \coP g by [].
have cgf: g \coP f by [].
rewrite  compfA // in ihgf.
have fhg: function (h \co g) by  fct_tac.
have chgf:  (h \co g) \coP f by hnf; aw.
move: (right_compose_fi chgf ihgf) => inf.
have ffh: function (f \co h) by fct_tac.
have cgfh: g \coP (f \co h) by hnf; aw.
move: (left_compose_fs  cgfh sgfh) => sg.
```

In both cases we know that f is injective and g surjective. If $f \circ h \circ g$ is injective, we deduce g injective; but surjectivity of g says $f \circ h$ injective; hence f is surjective. Injectivity of g in the second relation says $f \circ h$ surjective. Thus f, g and $f \circ h$ are injective and surjective; the result follows.

```
case is_fgh.
  rewrite compfA// => ifhg.
  have cfhg: (f \co h) \coP g by hnf; aw.
  move: (right_compose_fi  cfhg ifhg) => ig.
  move: (left_compose_fs2 cgfh sgfh  ig)=> sfh.
  move: (left_compose_fs cfh sfh) =>sf.
  move: (left_compose_fi2 cfhg ifhg sg) => ifh.
  have bfh: (bijection (f \co h)) by [].
  have bf: (bijection f)  by [].
  have bg: (bijection g)  by [].
  move: (right_compose_fb cfh bfh bf).
  done.
```

The second case is similar.

```
move=> sfhg.
have cfhg: (f \coP (h \co g)) by hnf; aw.
move: (left_compose_fs cfhg  sfhg) => sf.
have bf: (bijection f)  by [].
move: (left_compose_fs2 cfhg sfhg inf) => shg.
move:(left_compose_fi2 chgf ihgf sf) => ihg.
move: (right_compose_fi chg ihg) => ig.
have bg: (bijection g)  by [].
have bhg: (bijection (h \co g)) by [].
move: (left_compose_fb chg bhg bg).
done.
Qed.
```

11. **Find the error in the following argument: let* **N** *denote the set of all natural numbers and let* A *denote the set of all integers* $n > 2$ *for which there exists three strictly positive integers* x, y, z *such that* $x^n + y^n = z^n$*. Then the set* A *is not empty (in other words, "Fermat's last theorem" is false). For let* $\mathrm{B} = \{\mathrm{A}\}$ *and* $\mathrm{C} = \{\mathbf{N}\}$*;* B *and* C *are sets consisting of a single element, hence there is a bijection* f *of* B *onto* C*. We have* $f(\mathrm{A}) = \mathbf{N}$*; if* A *were empty we would have* $\mathbf{N} = f(\emptyset) = \emptyset$ *which is absurd.**

We have $f\langle\emptyset\rangle = \emptyset$ and $f(\emptyset) = \mathbf{N}$. Writing the first relation as $f(\emptyset) = \emptyset$ creates an ambiguity, but has not as consequence that $\emptyset$ is equal to **N**.

7.4 Section 4

1. *Let* G *be a graph. Show that the following three propositions are equivalent: (a)* G *is a functional graph, (b) if* X, Y *are any two sets, then* $\mathrm{G}^{-1}(\mathrm{X} \cap \mathrm{Y}) = \mathrm{G}^{-1}(\mathrm{X}) \cap \mathrm{G}^{-1}(\mathrm{Y})$*. (c) The relation* $\mathrm{X} \cap \mathrm{Y} = \emptyset$ *implies* $\mathrm{G}^{-1}(\mathrm{X}) \cap \mathrm{G}^{-1}(\mathrm{Y}) = \emptyset$*.*

```
Lemma exercise4_1a g: sgraph g  ->
  (functional_graph g <-> {morph inverse_image g : x y / x \cap y}).
Proof.
move=> gg.
have gig: sgraph (inverse_graph g) by  fprops.
split.
  move=> fgg x y; set_extens t.
    move /iim_graph_P => [u []] /setI2_P [ux uy] jg.
```

```
    apply /setI2_P;split;apply /iim_graph_P; ex_tac.
  move /setI2_P => [/iim_graph_P [u ux ua]/iim_graph_P [v vx va]].
  rewrite -(fgg _ _ _ ua va) in vx.
  apply /iim_graph_P; exists u; fprops.
move=> hyp x y y' gxy gxy'.
move: (hyp (singleton y)(singleton y')).
set u:= _ \cap _ => hyp1.
have:inc x (inverse_image g u).
  rewrite /u hyp1;apply: setI2_i; apply /iim_graph_P.
    exists y; fprops.
    exists y'; fprops.
by move /iim_graph_P => [t /setI2_P [/set1_P  <- /set1_P  <-]].
Qed.


Lemma exercise4_1b g: sgraph g  ->
  (functional_graph g <-> (forall x y, disjoint x y ->
    disjoint (inverse_image g x) (inverse_image g y))).
Proof.
rewrite /disjoint;move=> gg; split.
  move /(exercise4_1a gg) => h x y ie; rewrite /disjoint -h ie.
  by rewrite /inverse_image dirim_set0.
move=> hyp x y y' gxy gxy'.
have gig: sgraph (inverse_graph g) by fprops.
case (emptyset_dichot ((singleton y) \cap (singleton y'))).
  move=>aux; move:(hyp _ _ aux).
  set v:=  _ \cap _.
  have xv: (inc x v).
    rewrite /v; apply: setI2_i; apply /iim_graph_P; ex_tac.
    by move => ve; move: xv; rewrite ve => /in_set0.
by move=> [z /setI2_P [/set1_P -> /set1_P ->]].
Qed.
```

2. *Let* G *be a graph. Show that for each set* X *we have* $G(X) = \mathrm{pr}_2(G \cap (X \times \mathrm{pr}_2 G))$ *and* $G(X) = G(X \cap \mathrm{pr}_1 G)$.

Comment. There is no need to assume that G is a graph.

```
Lemma exercise4_2a g x:
  direct_image g x = range (g \cap (x \times (range g))).
Proof.
set_extens y.
  move /dirim_P => [a ax pg]; apply/funI_P;  exists (J a y); aw.
  apply /setI2_P; split => //; apply:setXp_i => //; ex_tac.
move /funI_P => [a  /setI2_P [pg /setX_P [pa pb pc]] ->].
by apply/dirim_P; ex_tac; rewrite pa.
Qed.

Lemma exercise4_2b g x:
  direct_image g x = direct_image g (x \cap (domain g)).
Proof.
set_extens t; move /dirim_P => [y ys Jg];  apply/dirim_P.
  ex_tac; apply /setI2_P; split=> //; ex_tac.
move /setI2_P: ys => [pa pb]; ex_tac.
Qed.
```

3. *Let* X, Y, Y′, Z *be four sets. Show that* $(Y' \times Z) \circ (X \times Y) = \emptyset$ *if* $Y \cap Y' = \emptyset$ *and that* $(Y' \times Z) \circ (X \times Y) = X \times Z$ *if* $Y \cap Y' \neq \emptyset$.

```
Lemma exercise4_3a x y y' z: disjoint y y' ->
  (y' \times z) \cg (x \times y) = emptyset.
Proof.
rewrite /disjoint; move=> ie; apply /set0_P.
move=> t => /compg_P [_ [u /setXp_P [_ uy] /setXp_P [uy' _]]].
by empty_tac1 u.
Qed.

Lemma exercise4_3b x y y' z: nonempty(y \cap y') ->
  (y' \times z) \cg (x \times  y) = x \times z.
Proof.
move=> [t] /setI2_P [ty ty'].
set_extens u.
  move /compg_P => [pu [v /setXp_P [pa _] /setXp_P [_ pb]]].
  by apply:setX_i.
move => /setX_P [pu Px Qy]; apply/compg_P; split => //; exists t; fprops.
Qed.
```

4. *Let* $(G_\iota)_{\iota \in I}$ *be a family of graphs. Show that for every set* X *we have* $(\bigcup_{\iota \in I} G_\iota)\langle X \rangle = \bigcup_{\iota \in I} G_\iota \langle X \rangle$, *and that for every object* x, $(\bigcap_{\iota \in I} G_\iota)\langle \{x\} \rangle = \bigcap_{\iota \in I} G_\iota \langle \{x\} \rangle$. *Give an example of two graphs* G, H *and a set* X *such that* $(G \cap H)\langle X \rangle \neq G\langle X \rangle \cap H\langle X \rangle$.

We have already shown that $G \mapsto G\langle x \rangle$ is a morphism for the union of two sets. We show there that it is a morphism in the general case. We introduce a definition.

We have to show that $(\exists y \in X)(\exists i)(x, y) \in G_i$ is the same as $(\exists i)(\exists y \in X)(x, y) \in G_i$.

```
Definition graph_morph op ui g :=
    op (ui g) = ui (Lg (domain g) (fun i => op (Vg g i))).

Lemma exercise4_4a g x: graph_morph (direct_image^~x) unionb g.
Proof.
set_extens y.
  move => /dirim_P [a ax /setUb_P [u ud Jv]].
  apply: (@setUb_i _ u); bw; apply /dirim_P; ex_tac.
move /setUb_P => [z]; rewrite Lg_domain => zd; bw.
move /dirim_P => [u ux Jv]; apply /dirim_P; ex_tac; union_tac.
Qed.
```

We have to show that $(\exists y \in X)(\forall i)(x, y) \in G_i$ is the same as $(\forall i)(\exists y \in X)(x, y) \in G_i$. We cannot exchange quantifiers. However, if X is a singleton $\{u\}$, $y \in X$ is equivalent to $y = u$, and this commutes.

```
Lemma exercise4_4b g x: singletonp x ->
  graph_morph (direct_image^~x) intersectionb g.
Proof.
move=> [y ->]; set_extens t.
  move => /dirim_P [a /set1_P ->].
  case (emptyset_dichot g) => gne.
    by rewrite gne setIb_0 => /in_set0.
  move => pi; apply: setIb_i.
     move /domain_set0P: gne => [u udg].
     pose ff i := direct_image (Vg g i) (singleton y).
```

```
     exists (J u (ff u)); apply /funI_P; ex_tac.
  bw; move => i idg; bw; apply/dirim_P; exists y; fprops.
  exact (setIb_hi pi idg).
set f := Lg _ _.
have dfdf: domain f = domain g by rewrite /f; bw.
case (emptyset_dichot g) => gne.
  by rewrite /f gne domain_set0 /Lg funI_set0 setIb_0 => /in_set0.
move => ti; apply /dirim_P; exists y; first by fprops.
apply/(setIb_P gne) => i idg; move: (idg); rewrite -dfdf=>  idf.
by move: (setIb_hi ti idf); rewrite /f; bw; move /dirim_P => [u /set1_P ->].
Qed.
```

Let us turn now to the example. We want to find X, G and H such that $p(X) \neq q(X)$. We have $p(X) = p(X')$ and $q(X) = q(X')$ where X' is the intersection of X and the domain of G or H. We know $p(X) = q(X)$ if X is a singleton. Thus X, G and H must have at least two elements. We give here the minimal solution: X has two elements, G is the identity in X, and H permutes the elements.

```
Lemma exercise4_4c: exists z,  not {morph (direct_image ^~z): x y / x \cap y}.
Proof.
set (x:=TPa); set  (y:= TPb); set z := (doubleton x y).
exists z.
set (G:= doubleton(J x x)(J y y)); set  (H:=  doubleton(J x y)(J y x)).
move => h; move: {h} (h G H).
have ->: direct_image G z = z.
  set_extens u.
    move=> /dirim_P [v vz /set2_P] [] h; rewrite (pr2_def h) /z; fprops.
  case /set2_P => h; apply /dirim_P; exists u; rewrite h /z /G; fprops.
have -> :direct_image H z = z.
  set_extens u.
    move=> /dirim_P [v vz /set2_P] [] h; rewrite (pr2_def h) /z; fprops.
  case /set2_P => ->; apply /dirim_P;  [ exists y | exists x];
     rewrite /H;fprops.
rewrite setI2_id => bad.
move: (set2_1 x y); rewrite -/z -bad; move/dirim_P => [t _ /setI2_P[pa]].
have : P (J t x) = Q (J t x) by case/set2_P: pa => ->; aw.
aw => ->; case/set2_P => h; [move: (pr2_def h) | move: (pr1_def h)]; fprops.
Qed.
```

5. *Let* $(G_\iota)_{\iota\in I}$ *be a family of graphs and let* H *be a graph. Show that*

$$(\bigcup_{\iota\in I} G_\iota)\circ H = \bigcup_{\iota\in I}(G_\iota\circ H) \qquad \textit{and} \qquad H\circ(\bigcup_{\iota\in I} G_\iota) = \bigcup_{\iota\in I}(H\circ G_\iota).$$

```
Lemma exercise4_5 G H:
   graph_morph (composeg ^~H) unionb G
   /\ graph_morph (composeg H) unionb G.
Proof.
split.
  set_extens x.
    move /compg_P => [px [y ph/setUb_P [z zd JV]]].
    apply/setUb_P; bw; ex_tac; bw; apply /compg_P;split => //; ex_tac.
  move /setUb_P; bw; move => [y ydg]; bw; move /compg_P => [px [t pa pb]].
  apply /compg_P;split=> //;ex_tac; union_tac.
```

```
set_extens x.
  move /compg_P => [px [y /setUb_P [z zd JV] ph]].
  apply/setUb_P; bw; ex_tac; bw; apply /compg_P;split => //; ex_tac.
move /setUb_P; bw; move => [y ydg]; bw; move /compg_P => [px [t pa pb]].
apply /compg_P;split => //;ex_tac; union_tac.
Qed.
```

6. *A graph* G *is functional if and only if for each pair of graphs* H, H′ *we have*

$$(H \cap H') \circ G = (H \circ G) \cap (H' \circ G).$$

Note that $(H \cap H') \circ G \subset (H \circ G) \cap (H' \circ G)$ is true for any graphs.

```
Lemma exercise4_6 G: sgraph G ->
  (fgraph G <->
  {when sgraph &, {morph (composeg ^~G) : H H' / H \cap H'}}).
Proof.
move => gG; split.
  move=> fG H H' _ _; set_extens x.
    move /compg_P => [px [y J1 /setI2_P [J2 J3]]].
    apply: setI2_i; apply /compg_P; split => //;ex_tac.
  move /setI2_P => [] /compg_P [px [y J1 J2 /compg_P [_ [y' J1' J2']]]].
  rewrite  (fgraph_pr fG J1 J1') in J2.
  apply/compg_P; split => //; ex_tac; fprops.
```

Converse. If $(x, y) \in G$ and $(x, y') \in G$ we consider the mappings $y \mapsto x$ and $y' \mapsto x$. Then (x, x) is in $H \circ G$ and $H' \circ G$. Thus $H \cap H'$ is nonempty.

```
move=> hyp; split=>// x y xG yG Pxy.
set (H:= singleton(J (Q x) (P x))).
set (H':= singleton(J (Q y) (P y))).
have gh: sgraph H by move=> t /set1_P ->; fprops.
have gh': sgraph H' by move=> t /set1_P->; fprops.
move: (gG _ xG)(gG _ yG)=> xp yp.
rewrite - xp in xG.
rewrite - yp in yG.
apply: pair_exten=>//.
have p1: inc (J (P x)(P x)) (H \cg G).
  apply /compg_P; split;fprops; aw;ex_tac; rewrite /H;  fprops.
have p2: inc (J (P y)(P y)) (H' \cg  G).
  apply /compg_P; split;fprops; aw;ex_tac; rewrite /H';  fprops.
have p3: (inc (J (P x)(P x)) ((H \cap H') \cg G)).
  by rewrite hyp//; apply: setI2_i => //;rewrite Pxy //.
move: p3; move/compg_P => [_ [z _]]; aw.
move /setI2_P => [] /set1_P r1 /set1_P r2.
by rewrite -(pr1_def r1) -(pr1_def r2).
Qed.
```

Notre that this is also true:

```
Lemma exercise4_6bis G: sgraph G ->
  (fgraph G <-> {morph (composeg^~G) : H H' / H \cap H'}).
```

7. *Let* G, H, K *be three graphs. Prove the relation* $(H\circ G)\cap K\subset (H\cap (K\circ G^{-1}))\circ (G\cap (H^{-1}\circ K))$.

```
Lemma exercise4_7 G H K:
  sub ((H \cg G) \cap K)
     ((H \cap (K \cg (inverse_graph G)))
      \cg  (G \cap ((inverse_graph H) \cg K))).
Proof.
move=> t /setI2_P [] /compg_P [tp [y JG JH]] tK.
apply /compg_P;split => //;rewrite - tp in tK.
by exists y; apply : setI2_i => //; apply/compg_P;
   split;fprops; aw; ex_tac; apply /igraph_pP.
Qed.
```

8. *Let* $\mathfrak{R}=(X_\iota)_{\iota\in I}$ *and* $\mathfrak{S}=(Y_\kappa)_{\kappa\in K}$ *be two coverings of a set* E. *(a) Show that if* $\mathfrak{R}$ *and* $\mathfrak{S}$ *are partitions of* E *and if* $\mathfrak{R}$ *is finer than* $\mathfrak{S}$, *then for every* $\kappa\in K$ *there exists* $\iota\in I$ *such that* $X_\iota\subset Y_\kappa$. *(b) Give an example of two coverings* $\mathfrak{R}$ *and* $\mathfrak{S}$ *such that* $\mathfrak{R}$ *is finer than* $\mathfrak{S}$ *but such that the property stated in (a) is not satisfied. (c) Give an example of two partitions* $\mathfrak{R}$ *and* $\mathfrak{S}$ *such that for every* $\kappa\in K$ *there exists* $\iota\in I$ *such that* $X_\iota\subset Y_\kappa$, *but such that* $\mathfrak{R}$ *is not a refinement of* $\mathfrak{S}$.

The French version does not assume that $\mathfrak{R}$ is a partition. We must however assume $Y_\kappa\neq\emptyset$.

```
Lemma exercise4_8a r s x:
  covering r x -> covering s x ->
  partition_w_fam s x ->  coarser_cg s r ->
  nonempty_fam s ->
  forall k, inc k (domain s) ->
     exists2 i, inc i (domain r) & sub (Vg r i) (Vg s k).
Proof.
move=> [fgr co1] [fgs co2] [fgL md usx] [_ _ co] alne k kds.
move: (alne _ kds)=> [y ysk].
have yx: inc y x by rewrite -usx;apply: (@setUb_i _ k);bw.
have yu: (inc y (unionb r)) by apply: co1.
move:  (setUf_hi yu)=> [z zdr yrz].
move: (co _ zdr)=> [i ids rsi].
have yri: inc y (Vg s i) by apply: rsi.
move: md; rewrite /mutually_disjoint; bw=> aux; case (aux _ _ kds ids).
  by move=>  ->; ex_tac.
move=> h; red in h.
by empty_tac1 y; bw; aw; split.
Qed.
```

We consider a covering R, and take for S the union of R and another set. Then R is finer than S.

```
Hint Rewrite variant_d variant_V_a variant_V_b:  bw.

Lemma exercise4_8b (a:= C0) (b:= C1)
  (x:= doubleton a b)
  (r:= Lg (singleton a) (fun _ => x))
  (s:= variantL a b x (singleton a)) :
    [/\ covering r x, covering s x,
      coarser_cg s r,
      nonempty_fam s &
```

```
      ~  (forall k, inc k (domain s) ->
            exists i, inc i (domain r) /\ sub (Vg r i) (Vg s k))].
Proof.
have ba: b<> a by rewrite /a/b; apply: TP_ne1.
rewrite /r/s/x;split.
- split; fprops; move=> t tx; apply: (@setUb_i _ a); fprops; bw; fprops.
- split; fprops; move=> y yx;apply: (@setUb_i _ a); fprops; bw; fprops.
- split; [ fprops | fprops | bw].
  move=> t /set1_P  ->; exists a; bw;fprops.
- move=> k; bw; case /set2_P=> ->; bw; [apply: set2_ne |  apply: set1_ne].
- have bd: (inc b (doubleton a b)) by fprops.
  bw; move=> h; move: (h _ bd)=> [i [/set1_P ->]]; bw; fprops => xa.
  by move: (xa _ bd) => /set1_P.
Qed.
```

Second counter example. The mapping $\kappa \mapsto \iota$ is injective. If I and K have the same number of elements, both partitions are equivalent. If K has a single element, then R is finer than S. Thus we need S_1 and S_2, $R_1 \subset S_1$, $R_2 \subset S_2$ and R_3 that is neither in S_1 nor in S_2, thus has an element in S_1 and another one in S_2. Thus E has at least four elements; in the initial version we used the following definitions:

```
Inductive four_points : Set :=  | fpa  | fpb | fpc |  fpd.
Inductive three_points : Set :=  | tpa  | tpb | tpc.
```

We use here the ordinals zero, one, two and three.

```
Lemma exercise4_8c
  (x:= C4)
  (r:= (Lg C3
     (fun i=>  Yo (i = C0) (singleton C0)
      (Yo (i = C2) (singleton C1) (doubleton C2 C3)))))
  (s:= variantL C0 C1 (doubleton C0 C2) (doubleton C1 C3)):
  [/\ partition_w_fam s x,
      partition_w_fam r x,
     (forall k, inc k (domain s) ->
       exists2 i, inc i (domain r) & sub (Vg r i) (Vg s k)) &
     ~(coarser_cg s r)].
```

We prove some obvious properties like $S_a \cap S_b = \emptyset$.

```
Proof.
move:C2_neC01 => [n1 n2].
move:C3_neC012 => [n3 n4 n5].
have nba: C1 <> C0 by fprops.
have sab:  (disjoint (Vg s C0) (Vg s C1)).
  rewrite /s; bw; apply: disjoint_pr=>u ud1 ud2.
  case /set2_P: ud1=> h; case /set2_P: ud2; rewrite h; auto.
have ra: inc C0 C3 by apply /C3_P; in_TP4.
have rb: inc C1 C3 by apply /C3_P; in_TP4.
have rc: inc C2 C3 by apply /C3_P; in_TP4.
have dab: disjoint (Vg r C0) (Vg r C1).
  rewrite /r; bw; Ytac0; Ytac0; Ytac0; Ytac0.
  apply: disjoint_pr=> u /set1_P -> /set2_P; case; auto.
have dac: disjoint (Vg r C0) (Vg r C2).
  rewrite /r; bw; Ytac0; Ytac0; Ytac0; Ytac0.
```

```
  apply: disjoint_pr=> u  /set1_P -> /set1_P; auto.
have dcb: disjoint (Vg r C2) (Vg r C1).
  rewrite /r; bw; Ytac0; Ytac0; Ytac0; Ytac0.
  apply: disjoint_pr=> u /set1_P -> /set2_P; case; auto.
split.
```

The first step is to prove that S is a partition. It has two elements $S_a = \{a, c\}$ and $S_b = \{b, d\}$. For each x, there is a ν such that $x \in S_\nu$. It is respectively a, b, a and b.

```
(* S partition *)
  rewrite /s;split; fprops.
    rewrite /variantL;red; bw; move=> i j ids jds.
    case /set2_P: ids => ->; case /set2_P: jds =>->; auto.
    by right; apply:disjoint_S.

  set_extens y => ys.
    case (setUb_hi ys); bw; move=> z zd.
    case /set2_P: zd => ->; bw=> yd; case /set2_P: yd => ->; apply/C4_P;in_TP4.
   case /C4_P: ys; move => ->.
     by apply :(@setUb_i _ C0);bw; fprops.
     by apply :(@setUb_i _ C1);bw; fprops.
     by apply :(@setUb_i _ C0);bw; fprops.
     by apply :(@setUb_i _ C1);bw; fprops.
```

We prove now that R is a partition. Since R has three elements it is a bit longer (we must show that 6 pairs of sets are disjoint). We have $R_a = \{a\}$ and $R_b = \{c, d\}$, $R_c = \{b\}$. For each x, there is a ν such that $x \in R_\nu$. It is respectively a, c, b and b.

```
(* R partition *)
   split; first by rewrite /r; fprops.
    red; rewrite {1 2} /r;  bw; move=> i j idr jdr.
    case /C3_P:idr => ->;case /C3_P:jdr;try move => ->; auto;
      try case => ->; auto;
      by right; apply:disjoint_S.
  set_extens t => ts.
    move: (setUb_hi ts); rewrite /r;bw; move => [y ydr]; bw.
    case /C3_P:ydr => ->; Ytac0; Ytac0;
        try move /set1_P ->; try case/set2_P => ->; apply /C4_P;in_TP4.
  rewrite /r; case /C4_P: ts;
     [set v := C0 | set v := C2 | set v := C1 | set v := C1 ];
     move => ->; apply: (@setUb_i _ v); bw; Ytac0; Ytac0; fprops.
```

We show that for all $\kappa \in K$ there exists $\iota \in I$ such that $X_\iota \subset Y_\kappa$. This is $R_a \subset S_a$ and $R_c \subset S_b$. After that, we show that R_b is not a subset of any S_ι.

```
(* property *)
  rewrite /s/r; bw;move => k kds; case /set2_P: kds =>->.
    exists C0 => //; bw; Ytac0; Ytac0 => t /set1_P ->; fprops.
  exists C2 => //; bw; Ytac0; Ytac0 => t /set1_P ->; fprops.
(* not refinement *)
move=> [_ _ ]; rewrite {1}/r;bw => cc.
move: (cc _ rb) => [i]; rewrite /s; bw=> ids.
rewrite /r; bw; Ytac0; Ytac0.
case /set2_P: ids=> ->; bw => h.
   move: (h _ (set2_2 C2 C3)) => /set2_P; case; auto.
move: (h _ (set2_1 C2 C3)) => /set2_P; case; auto.
Qed.
```

7.5 Section 5

* *Montrer que si* X, Y *sont deux ensembles tels que* $\mathfrak{P}(X) \subset \mathfrak{P}(Y)$, *on a* $X \subset Y$.

This exercise appears only in the French version. The converse is true, so that we show $\mathfrak{P}(X) \subset \mathfrak{P}(Y) \iff X \subset Y$.

```
Proof.
move=> sxy z zx.
have p2: sub (singleton z) y.
 by apply /setP_P; apply: sxy; apply /setP_P => t /set1_P ->.
apply: (p2 z); fprops.
Qed.
Lemma powerset_mono A B: sub A B -> sub (powerset A)(powerset B).
Proof.
move=> sAB t /setP_P ta; apply/setP_P; apply:(sub_trans ta sAB).
Qed.
```

* *Soient* E *un ensemble* f *une application de* $\mathfrak{P}(E)$ *dans lui-même telle que la relation* $X \subset Y$ *entraîne* $f(X) \subset f(Y)$. *Soit* V *l'intersection des ensembles* $Z \subset E$ *tels que* $f(Z) \subset Z$ *et soit* W *la réunion des ensembles* $Z \subset E$ *tels que* $Z \subset f(Z)$. *Montrer que* $f(V) = V$ *et* $W = f(W)$ *et que pour tout ensemble* $Z \subset E$ *tel que* $f(Z) = Z$ *on a* $V \subset Z \subset W$.

This exercise appears only in the French version. We prove (in the second part of this report) the following theorem of Tarski: let F be a complete lattice, and $f : F \to F$ an increasing function. Then the set of fixpoints of f is a complete lattice. This exercise considers the case where F is the powerset of E (ordered by inclusion), and shows that f has a least and a greatest fixpoint, namely V, the intersection of the sets $Z \subset E$ for which $f(Z) \subset Z$ and W, the union of the sets $Z \subset E$ such that $Z \subset f(Z)$.

```
Lemma exercise5_f2 f x v w:
  function f -> source f = (powerset x) -> target f = powerset x ->
  (forall a b, inc a (powerset x) -> inc b (powerset x) -> sub a b
    -> sub (Vf f a) (Vf f b)) ->
  v = intersection(Zo (powerset x) (fun z=> sub (Vf f z) z)) ->
  w = union(Zo (powerset x) (fun z=> sub z (Vf f z))) ->
  [/\ Vf f v = v, Vf f w = w & (forall z, sub z x -> Vf f z = z ->
    (sub v z /\ sub z w))].
Proof.
move=> ff sf tf fprop vd wd.
set (q:= (Zo (powerset x) (fun z => sub (Vf f z) z))).
have xpx: inc x (powerset x) by apply:setP_Ti.
have xiq: inc x q.
  rewrite /q; apply: Zo_i=>//.
  by apply: /setP_P; rewrite -tf; apply: Vf_target =>//; rewrite sf.
have neq:nonempty q by exists x.
set (p:= (Zo (powerset x) (fun z  => sub z (Vf f z)))).
have fzv:forall z, sub z x -> Vf f z = z -> sub v z.
  move => z zx Wz.
  have zq:inc z q by apply: Zo_i; [by apply /setP_P | rewrite  Wz; fprops].
  by rewrite vd; apply: setI_s1.
have fzw:forall z, sub z x -> Vf f z = z -> sub z w.
  move => z zx Wz.
  have zp: inc z p by apply: Zo_i; [by apply /setP_P | rewrite  Wz; fprops].
  by rewrite wd; apply: setU_s1.
```

```
have qW: forall z, inc z q -> inc (Vf f z) q.
   move=> z /Zo_P [] /setP_P zx Wzz.
   have aux: sub (Vf f z) x by apply: (@sub_trans z).
   by apply: Zo_i; [ apply/setP_P | apply: fprop => //; apply/setP_P].
have pW: forall z, inc z p -> inc (Vf f z) p.
   move=> z /Zo_P [] /setP_P zx Wzz.
   have aux: inc (Vf f z)  (powerset x).
     by rewrite -tf; apply: Vf_target=>//;rewrite sf; apply /setP_P.
   by apply: Zo_i => //; apply: fprop => //; apply/setP_P.
have vp: inc v (powerset x) by  apply /setP_P; rewrite vd; apply: setI_s1.
have wp: inc w (powerset x).
  by  apply /setP_P; rewrite wd;  apply: setU_s2 => y /Zo_P []/setP_P.
have pv:sub (Vf f v) v.
  move=> t tW; rewrite vd; apply: setI_i=>// y /Zo_P [yp sW].
  have vy: sub v y by rewrite vd; apply: setI_s1; apply: Zo_i=>//.
  by apply: sW;apply: (fprop _ _ vp yp vy).
have pw:sub w (Vf f w).
  move=> t; rewrite {1} wd=> /setU_P [y ty] /Zo_P [yp yW].
  have tw: (sub y w) by rewrite wd;apply: setU_s1; apply: Zo_i=>//.
  by move: (fprop _ _ yp wp tw); apply; apply: yW.
split.
    apply: extensionality=>//.
    have vq: (inc v q) by rewrite /q;apply: Zo_i.
    by move: (qW _ vq)=> aux; rewrite {1} vd;apply: setI_s1.
  apply: extensionality=>//.
  have iwp: inc w p by rewrite /p; apply: Zo_i.
  by move: (pW _ iwp) => aux; rewrite {2} wd;apply: setU_s1.
move=> z zw wz; split; fprops.
Qed.
```

1. *Let* $(X_\iota)_{\iota\in I}$ *be a family of sets. Show that if* $(Y_\iota)_{\iota\in I}$ *is a family of sets such that* $Y_\iota \subset X_\iota$ *for each* $\iota \in I$ *then* $\prod_{\iota\in I} Y_i = \bigcap_{\iota\in I} \mathrm{pr}_\iota^{-1}(Y_\iota)$.

```
Lemma exercise5_1 I x y:
  (forall i, inc i I -> sub (y i) (x i)) -> nonempty I ->
  productf I y =
  intersectionf I (fun i=> inv_image_by_fun (pr_i (Lg I x) i) (y i)).
Proof.
move=> syxi neI.
have fgL: fgraph (Lg I x) by fprops.
have fpj: forall j, inc j I->function (pr_i (Lg I x) j).
  move=> j jI; apply: pri_f=>//;bw.
set_extens t.
  move /setXf_P=> [fgt dt iVy]; apply: setIf_i=>//.
  move=> j jI; apply /iim_graph_P.
  exists (Vg t j); first by apply: iVy.
  have jd: inc j (domain  (Lg I x)) by bw.
  have tp:(inc t (productb (Lg I x))).
    by apply/setXb_P; split; bw => i iI; bw; apply: syxi=>//; apply: iVy.
  by rewrite -(pri_V fgL jd tp); Wtac; rewrite /pr_i lf_source.
have rI: inc (rep I) I by apply: rep_i.
move => h; move:(setIf_hi h  rI) => /iim_graph_P [u uy Jg].
move: (p1graph_source (fpj _ rI) Jg).
rewrite /pr_i;aw; move /setXf_P=>[fgt dt iVV].
apply/setXf_P;split => // i idt.
move: (setIf_hi h  idt) => /iim_graph_P [v vi Jgv].
```

```
move: (Vf_pr (fpj _ idt) Jgv); rewrite pri_V =>//; bw.
  by move=> <-.
by apply/setXb_P; split; bw => k ki; bw; apply: iVV.
```

2. *Let* A, B *be two sets. For each subset* G *of* $A \times B$ *let* $\tilde{G}$ *be the mapping* $x \mapsto G\langle\{x\}\rangle$ *of* A *into* $\mathfrak{P}(B)$. *Show that the mapping* $G \mapsto \tilde{G}$ *is a bijection from* $\mathfrak{P}(A \times B)$ *onto* $(\mathfrak{P}(B))^A$.

Note that $\tilde{G}$ is in $\mathscr{F}(A;\mathfrak{P}(B))$. The French edition says: let $\tilde{G}$ be the graph of the mapping etc, so that $\tilde{G}$ is in $(\mathfrak{P}(B))^A$.

```
Lemma exercise5_2 a b:
  bijection (Lf(fun g => Lg a (fun x => direct_image g (singleton x)))
    (powerset (a \times b)) (gfunctions a (powerset b))).
Proof.
set tilde := L _ _.
apply: lf_bijective.
```

We first prove that the mapping $G \mapsto \tilde{G}$ is a function.

```
    move=> c /setP_P cp.
    set faux:=(Lf (fun x=> direct_image c (singleton x)) a (powerset b)).
    suff: (inc (graph faux) (gfunctions (source faux) (target faux))).
       by rewrite /faux /Lf; aw.
    apply: gfun_set_i ;apply: lf_function => t ta; apply/setP_P => u.
    by move /dirim_P => [x _ pb]; move/setXp_P: (cp _ pb) => [].
```

We prove that the mapping is injective.

```
  move /setP_P => up /setP_P => vp fxy.
  set_extens x => xs.
   move /setX_P: (up _ xs) => [px Px Qx].
    have: inc (Q x) (Vg fy (P x)).
      by rewrite -fxy /fx; bw; apply/dirim_P; ex_tac; rewrite px.
    by rewrite /fy; bw; move/dirim_P=> [w /set1_P ->]; rewrite px.
  move /setX_P: (vp _ xs) => [px Px Qx].
  have: inc (Q x) (Vg fx (P x)).
     by rewrite fxy /fy; bw; apply/dirim_P; ex_tac; rewrite px.
    by rewrite /fx; bw; move/dirim_P=> [w /set1_P ->]; rewrite px.
```

We prove that the mapping is surjective.

```
move=> y ys; move: (gfun_set_hi ys)=> [f [fs sf tg gf]].
set (g:=Zo (a \times b) (fun z => inc (Q z) (Vg y (P z)))).
have gp: inc g (powerset (a \times b)) by apply/setP_P;apply: Zo_S.
rewrite -gf; ex_tac; apply: fgraph_exten; fprops.
  bw; aw.
red; rewrite - (proj33 fs) sf => x xa; bw;rewrite gf;set_extens u.
  move=> h; apply/dirim_P; exists x; first by fprops.
  apply: Zo_i; aw; apply: setXp_i => //.
  rewrite - sf in xa; move: (Vf_target fs xa).
  by rewrite tg /Vf gf; move/setP_P; apply.
move /dirim_P => [v /set1_P ->] /Zo_P []; aw.
```

3. * *Let* $(X_i)_{1\leq i\leq n}$ *be a finite family of sets. For each subset* H *of the index set* $[1,n]$ *let* $P_H = \bigcup_{i\in H} X_i$ *and* $Q_H = \bigcap_{i\in H} X_i$. *Let* $\mathfrak{F}_k$ *be the set of subsets of* $[1,n]$ *which have* k *elements. Show that*

$$\bigcup_{H\in\mathfrak{F}_k} Q_H \supset \bigcap_{H\in\mathfrak{F}_k} P_H \text{ if } k \leq (n+1)/2$$

and that

$$\bigcup_{H\in\mathfrak{F}_k} Q_H \subset \bigcap_{H\in\mathfrak{F}_k} P_H \text{ if } k \geq (n+1)/2. \quad *$$

See part two of this report of an answer.

7.6 Section 6

1. *For a graph* G *to be the graph of an equivalence relation on a set* E*, it is necessary and sufficient that* $pr_1 G = E$, $pr_2 G = E$, $G \circ G^{-1} \circ G = G$ *and* $\Delta_E \subset G$ *(*Δ_E *being the diagonal of* E*).*

Comment. The condition $pr_2 G = E$ was missing in the English version [2]. It is necessary: consider the graph with two elements (a,a) and (b,a).

```
Lemma exercise6_1 x g: sgraph g ->
  ((equivalence g /\ substrate g = x) <->
  [/\ domain g = x, range g = x,
    g \cg ((inverse_graph g) \cg g) = g &
    sub (diagonal x) g]).
Proof.
move=> gg; split.
  move=> [eg sg]; split => //.
  -  by rewrite (domain_sr eg).
  - rewrite - sg /substrate; set_extens t => ts; first by fprops.
    case /setU2_P:ts => // /(domainP gg) [y Jh]; apply/(rangeP gg).
    exists y; equiv_tac.
  - set_extens y.
      move /compg_P => [py [z /compg_pP [u pa /igraph_pP pb pc]]].
      have J4: inc (J u z) g by equiv_tac.
      have J5: inc (J (P y) z) g by equiv_tac.
      have: inc (J (P y) (Q y)) g by equiv_tac.
      by rewrite py.
    move=> yg.
    have py: pairp y by  apply: gg.
    have  yv:J (P y) (Q y) = y by aw.
    rewrite - py; apply /compg_pP; exists (P y); last by ue.
    apply /compg_pP; exists (Q y);  [|  apply/igraph_pP]; ue.
  - move => t /diagonal_i_P [pt Pt PQt].
    by rewrite -pt -PQt;  rewrite - sg in Pt; equiv_tac.
```

Now the converse

```
move=> [dg [rg [cg si]]].
have sg: (substrate g  = x) by rewrite /substrate dg rg; apply: setU2_id.
split=>//.
have p1: forall u, inc u x -> inc (J u u) g.
  by move=> u ux; apply: si; apply /diagonal_pi_P.
have p2: symmetricp g.
```

```
  move=> a b ab; red in ab.
  have Jag: (inc (J a a) g) by apply: p1; rewrite -dg; aw;  ex_tac.
  have Jbg: (inc (J b b) g) by apply: p1; rewrite -rg; aw;  ex_tac.
  red;  rewrite -cg; apply /compg_pP; ex_tac; apply /compg_pP; ex_tac.
  by apply /igraph_pP.
have p3: transitivep g.
  move=> a b c ab bc; rewrite -cg; apply /compg_pP.
  exists a => //; apply /compg_pP; exists b => //.
    by apply: p1; rewrite - sg; substr_tac.
  by apply/igraph_pP; apply (proj2 p2).
by apply:symmetric_transitive_equivalence.
Qed.
```

2. *If* G *is a graph such that* $G \circ G^{-1} \circ G = G$ *show that* $G^{-1} \circ G$ *and* $G \circ G^{-1}$ *are graphs of equivalences on* $\mathrm{pr}_1 G$ *and* $\mathrm{pr}_2 G$ *respectively.*

We first compute the substrate of the relations.

```
Lemma exercise6_2 g: sgraph g ->
  compose_graph g (compose_graph (inverse_graph g) g) = g ->
  [/\ equivalence ((inverse_graph g) \cg g),
    substrate ((inverse_graph g) \cg g) = domain g ,
    equivalence (g \cg (inverse_graph g)) &
    substrate (g \cg (inverse_graph g)) = range g].
Proof.
move=> gg cg.
have gig:sgraph (inverse_graph g) by apply: igraph_graph.
have gcigg:sgraph ((inverse_graph g) \cg g) by apply: compg_graph.
have gcgig: sgraph (g \cg (inverse_graph g))  by apply: compg_graph.
have t3:forall x y z t, related g x y -> related g z y -> related g z t ->
    related g x t.
  move=> x y z t xy zy zt; red; rewrite -cg; apply/compg_pP.
    by exists z=>//;apply/compg_pP;exists y => //;apply /igraph_pP.
have s1: substrate ((inverse_graph g) \cg g) = domain g.
  set_extens x.
    case /(substrate_P gcigg) => [] [y /compg_pP [z J1] /igraph_pP J2];
       /igraph_pP J2; ex_tac.
  move/(domainP gg) => [y Jg].
  have Jxx: (inc (J x x) ((inverse_graph g) \cg g)).
    by apply/compg_pP; ex_tac; apply /igraph_pP.
  apply: (arg1_sr Jxx).
have s2:substrate (g \cg (inverse_graph g)) = range g.
  set_extens x.
    case/(substrate_P gcgig)=> [][y /compg_pP [z /igraph_pP J1 J2]];
    ex_tac.
  move/(rangeP gg) => [y Jg].
  have Jxx: (inc (J x x) (g \cg (inverse_graph g))).
    by apply/compg_pP; ex_tac; apply /igraph_pP.
  apply: (arg1_sr Jxx).
```

We apply proposition 1. Γ is an equivalence if $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$. If Γ is the composition of G and G^{-1} in any order, the relation is true. The second is a consequence of the assumption and associativity of composition.

```
split => //; rewrite equivalence_pr; split;
      try rewrite compg_inverse igraph_involutive //.
```

```
  by rewrite - compgA cg.
by rewrite compgA in cg; by rewrite compgA  cg.
Qed.
```

3. *Let* E *be a set,* A *a subset of* E*, and* R *the equivalence relation associated with the mapping* $X \mapsto X \cap A$ *of* $\mathfrak{P}(E)$ *into* $\mathfrak{P}(E)$*. Show that there exists a bijection from* $\mathfrak{P}(A)$ *onto the quotient set* $\mathfrak{P}(E)/R$.

If $\sim$ is the equivalence associated, then B and B′ are related if they have the same intersection with A. If $u \in A$, we can consider the set of all B whose intersection with A is u as a class. This is our bijection (called canonical in the French edition). **Note:** Since intersection is commutative, we use here $X \mapsto A \cap X$.

```
Definition intersection_with x a :=
  Lf(intersection2 a) (powerset x)(powerset x).
Definition intersection_with_canon x a :=
  Lf (fun b => Zo(powerset x)(fun c=> c = a \cap c))
  (powerset a)(quotient (equivalence_associated (intersection_with x a))).
```

We start with some preliminaries.

```
Lemma exercise6_3 a x:
  sub a x -> bijection (intersection_with_canon x a).
Proof.
move=> ax.
have ta: lf_axiom (intersection2 a) (powerset x) (powerset x).
  move=> y /setP_P ay; apply /setP_i; apply: sub_trans ay ;apply: subsetI2r.
have fai: function (intersection_with x a) by apply: lf_function.
set r:= equivalence_associated (intersection_with x a).
have er: equivalence r by apply: graph_ea_equivalence.
have aux: forall y, sub y a -> y =  a \cap y by move => y; move/setI2id_Pr.
have rr: forall  u v, related r u v <->
    [/\ inc u (powerset x), inc v (powerset x)  &  a \cap u = a \cap v].
   move => u v; split.
      move/(ea_relatedP fai); rewrite lf_source; move => [pa pb].
     by rewrite /intersection_with; aw.
   move => [pa pb pc]; apply/(ea_relatedP fai).
   by rewrite /intersection_with; aw.
```

We show that we have a function.

```
apply: bl_bijective.
    move=> y /setP_P=> ya;set w:= Zo _ _ .
    have new: nonempty w.
      exists y;apply: Zo_i; [apply/setP_P; apply: (sub_trans ya ax) | auto].
    have swp: sub w (powerset x) by apply: Zo_S.
    have rp: inc (rep w) (powerset x) by  apply: swp;apply: rep_i.
    apply /(setQ_P er); split => //.
      move: rp;rewrite graph_ea_substrate /intersection_with; aw.
    have ira: (a \cap (rep w) = y).
      have: (inc (rep w) w) by apply: rep_i.
      by move /Zo_hi => ->.
    set_extens z.
       move => zw; apply /(class_P er); apply /rr;split => //; first by apply: swp.
       by rewrite ira; move /Zo_P: zw => [].
    by move/(class_P er)/rr => [pa pb pc]; apply: Zo_i => //; rewrite -pc ira.
```

We prove injectivity.

```
  move=> u v /setP_P ua /setP_P va;  set fs:= Zo _ _ => eql.
  have iua: u = a \cap u by apply: aux.
  have: inc u fs by  apply: Zo_i => //; apply/setP_P; apply:(sub_trans ua ax).
  by rewrite eql; move/ Zo_hi => ->.
```

We prove now the surjectivity.

```
move=> y /(setQ_P er) cy.
have ip: inc (a \cap (rep y)) (powerset a) by apply/setP_P; apply subsetI2l.
move: (graph_ea_substrate fai); rewrite -/r  lf_source => sr2.
ex_tac; symmetry;set_extens t.
  move /Zo_P => []; move /setP_P => pd pe.
  apply: (rel_in_class2 er cy); apply/rr;split => //; last by apply /setP_P.
  rewrite - sr2; exact (proj1 cy).
move => ty; apply /Zo_i; first by rewrite - sr2; apply: (sub_class_sr er cy ty).
by move: (rel_in_class er cy ty) => /rr  [_ _].
Qed.
```

4. *Let* G *be the graph of an equivalence on a set* E*. Show that if* A *is a graph such that* $A \subset G$ *and* $\mathrm{pr}_1 A = E$ *(resp.* $\mathrm{pr}_2 A = E$*) then* $G \circ A = G$ *(resp.* $A \circ G = G$*); furthermore, if* B *is any graph, we have* $(G \cap B) \circ A = G \cap (B \circ A)$ *(resp* $A \circ (G \cap B) = G \cap (A \circ B)$*).*

```
Lemma exercise6_4 g a b x:
  let comm F G b := F (G b) = G ( F b) in
  equivalence g -> sgraph a -> sgraph b -> substrate g = x -> sub a g ->
  [/\ (domain a = x -> g \cg a = g),
   (range a = x -> a \cg g = g),
  (domain a = x -> comm (composeg^~a) (intersection2 g) b) &
  (range a = x ->  comm (composeg a) (intersection2 g) b)].
Proof.
move=> comp inter eg ga gb sg ag.
have gg: sgraph g by fprops.
split
move=> ax; set_extens y.
  move /compg_P=> [py [z J1a J2g]].
  move: (ag _ J1a) => J1g.
  rewrite - py; equiv_tac.
 move=> yg; move: (gg _ yg)=> py; apply/compg_P.
split =>//.
have : (inc (P y) (domain a)) by rewrite ax - sg; substr_tac.
move/(domainP ga)=> [z Ja]; exists z =>//.
move: (ag _ Ja)=> Jg.
have J2g:inc (J z (P y)) g by  equiv_tac.
rewrite - py in yg; equiv_tac.
```

Second claim.

```
move=> ax; rewrite /comp; set_extens y.
  move /compg_P => [py [z J1g J2a]].
  move: (ag _ J2a) => J2g.
  rewrite - py; equiv_tac.
move=> yg; move: (gg _ yg)=> py; apply/compg_P =>//.
have : (inc (Q y) (range a)) by rewrite ax - sg; substr_tac.
```

```
move/(rangeP ga); move=> [z Ja]; split => //;exists z =>//.
move: (ag _ Ja)=> Jg.
have J2g:inc (J (Q y) z) g by  equiv_tac.
rewrite - py in yg; equiv_tac.
```

Third claim.

```
move=> ax;  set_extens y.
  move /compg_P => [py [z J1a/setI2_P [J2g J3b]]]; apply/setI2_i.
  move: (ag _ J1a) => J1g; rewrite - py; equiv_tac.
  apply/compg_P;split=>//;exists z=>//.
move=> /setI2_P [yg] /compg_P [py [z J1a J2b]].
apply/compg_P; split => //; exists z => //; apply:setI2_i => //.
move: (ag _ J1a)=> Jg.
have J2g:inc (J z (P y)) g by  equiv_tac.
rewrite -  py in yg; equiv_tac.
```

Last claim.

```
move=> ax; set_extens y.
  move /compg_P => [py [z /setI2_P [J2g J3b] J1a]]; apply/setI2_i.
  move: (ag _ J1a) => J1g; rewrite - py; equiv_tac.
  apply/compg_P;split=>//;exists z =>//.
move=> /setI2_P [yg] /compg_P [py [z J1a J2b]].
apply/compg_P; split => //; exists z => //; apply:setI2_i => //.
move: (ag _ J2b)=> Jg.
have J2g:inc (J (Q y) z) g by  equiv_tac.
rewrite - py in yg; equiv_tac.
Qed.
```

5. *Show that every intersection of graphs of equivalences on a set* E *is the graph of an equivalence on* E. *Give an example of two equivalences on a set* E *such that the union of their graphs is not the graph of an equivalence on* E.

We have already shown the first property. Let's show that the union of two symmetric relations is symmetric.

```
Lemma symmetric_union a b: symmetricp a -> symmetricp b ->
  symmetricp (a \cup b).
Proof.
by move=>  sa sb x y;  case /setU2_P=> h; apply/setU2_P;
   [left; apply: sa | right; apply: sb ].
Qed.
```

We show here that if $G \subset E \times E$, then the substrate of $G \cup \Delta_E$ is E.

```
Lemma substrate_union_diag: x g:
  sub g (coarse x) -> substrate (g \cup (identity_g x)) = x.
Proof.
move=> gc.
have gg: sgraph g by move=> t tg; move: (gc _ tg) => /setX_P [].
have gu: sgraph (g \cup (diagonal x)).
  by move=> t; case /setU2_P; [ auto | move/diagonal_i_P => []].
set_extens y.
  case /(substrate_P gu) => [] [z] /setU2_P [].
```

```
  - by move => h;  move: (gc _ h); move /setXp_P=> [].
  - by move/diagonal_pi_P => [].
  - by move => h;  move: (gc _ h); move /setXp_P=> [].
  - by move/diagonal_pi_P => [ h <-].
move=> yx.
have aux: inc (J y y) (g \cup (diagonal x)).
  by apply: setU2_2; apply /diagonal_pi_P; split.
substr_tac.
Qed.
```

If a and b are in E, we can consider $\Delta_E \cup \{(a,b),(b,a)\}$. Its substrate is E.

```
Definition special_equivalence a b x :=
  (doubleton (J a b) (J b a)) \cup (diagonal x).

Lemma substrate_special_equivalence a b x:
  inc a x -> inc b x ->  substrate(special_equivalence a b x) = x.
Proof.
move=> ax bx; rewrite/ special_equivalence.
apply: substrate_union_diag.
by move=> t /set2_P => [][] ->; apply/setXp_i.
Qed.
```

We show that this is an equivalence.

```
Lemma special_equivalence_ea a b x:
  inc a x -> inc b x -> equivalence(special_equivalence a b x).
Proof.
move=> ax bx.
have gs: sgraph (special_equivalence a b x).
  move=> t; move/setU2_P; case; first by case/set2_P => ->; fprops.
  by move  /diagonal_i_P => [].
have pair_symm: forall a b c d, J a b = J c d -> J b a = J d c.
  move=> u v u' v' eql.
  apply: pair_exten; fprops; aw.
    apply: (pr2_def eql).
  apply: (pr1_def eql).
apply: symmetric_transitive_equivalence => //.
  move=> u v h; case/setU2_P: (h).
    case /set2_P => ww; apply/setU2_P; left;
       rewrite (pr1_def ww)(pr2_def ww); fprops.
  by move => /diagonal_pi_P [_ uv]; move: h;rewrite uv.
move=> u v w ra rb.
case /setU2_P: (ra); last by move => /diagonal_pi_P [_ ->].
case /setU2_P: (rb); last by move => /diagonal_pi_P [_ <-].
move => h1 h2; apply/setU2_P.
case /set2_P: h1 => h11; rewrite (pr2_def h11);
  case /set2_P: h2 => h22; rewrite (pr1_def h22).
- left; fprops.
- by right; apply /diagonal_pi_P.
- by right; apply /diagonal_pi_P.
- left; fprops.
Qed.
```

If we have two such equivalences with (a,b) and (a,c), transitivity of the union would imply that b and c are related in one of the two graphs. If all three elements are distinct this is not possible.

```
Lemma exercise6_5
  (x := C3)
  (g1:= special_equivalence C0 C1 x)
  (g2:= special_equivalence C0 C2 x):
      [/\ equivalence g1, equivalence g2, substrate g1 = x,
        substrate g2 = x &  ~ (equivalence (g1 \cup g2))].
Proof.
split.
        apply: special_equivalence_ea; apply /C3_P; in_TP4.
      apply: special_equivalence_ea; apply /C3_P; in_TP4.
    rewrite substrate_special_equivalence //; apply /C3_P; in_TP4.
  rewrite substrate_special_equivalence //; apply /C3_P; in_TP4.
move=> bad.
have p1: (related (g1 \cup g2) C1 C0).
 apply /setU2_P; left;apply/setU2_P; left; fprops.
have p2: (related (g1 \cup  g2) C0 C2).
  apply /setU2_P; right;apply/setU2_P; left; fprops.
have :(related (g1 \cup g2) C1 C2) by equiv_tac.
move: C2_neC01 => [n1 n2].
case /setU2_P; case/setU2_P.
by case/set2_P=> eq2; move: (pr2_def eq2); auto.
by move /diagonal_pi_P =>  [_]; auto.
by case/set2_P=> eq2; move: (pr1_def eq2); fprops.
by move /diagonal_pi_P =>  [_]; auto.
Qed.
```

6. *Let* G, H *be the graphs of two equivalences on* E. *Then* G∘H *is the graph of an equivalence on* E *if and only if* G ∘ H = H ∘ G. *The graph* G ∘ H *is then the intersection of all the graphs of equivalences on* E *wich contain both* G *and* H.

We show that if G ∘ H is an equivalence then G ∘ H = H ∘ G. This uses symmetry.

```
Lemma exercise6_6a G H:
  equivalence G -> equivalence H ->
  (equivalence (G \cg H) <-> (G \cg H = H \cg G)).
Proof.
move=> eG eH.
set (K:= G \cg H).
split.
  move => eK.
     have aux: forall a b, inc (J a b) K -> inc (J b a) K
       by move => a b h;equiv_tac.
    set_extens x => xK.
    have px: (pairp x) by apply: (@compg_graph G H).
    move: xK ; rewrite - px => h; move : (aux _ _ h).
    move /compg_pP=> [y JH JG]; apply/compg_pP; exists y => //; equiv_tac.
  move: xK =>/compg_P [px [y JG JH]].
  rewrite - px; apply: aux; apply/compg_pP;exists y => //; equiv_tac.
```

Converse. We use Proposition 1 that says that an equivalence satisfies $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$.

```
move=> eq.
  move: eG eH; rewrite ! equivalence_pr.
move=> [GG iG] [HH iH]; split.
  rewrite {2} /K compgA eq.
  rewrite - (compgA H G G) GG - compgA.
```

```
  by rewrite -/K eq compgA HH.
by rewrite {2}/K compg_inverse -iH -iG.
Qed.
```

We show here that if G and H are equivalences on E, then the substrate of $G \circ H$ is E.

```
Lemma exercise6_6b G H:
  equivalence G -> equivalence H ->  substrate G = substrate H ->
  substrate (G \cg H) = substrate G.
Proof.
move=> eG eH sG.
set_extens x.
  have xx:sgraph (G \cg H) by fprops.
  case /setU2_P; [move /(domainP xx) | move /(rangeP xx)];
    move => [z] /compg_pP [t ta tb]; [rewrite  sG | ] ;  substr_tac.
move=> xsG.
have p3: related  (G \cg H) x x.
 apply/compg_pP; exists x;equiv_tac => //; ue.
substr_tac.
Qed.
```

We prove that the composition is the smallest equivalence that contains G and H.

```
Lemma exercise6_6c G H:
  equivalence G -> equivalence H -> substrate G = substrate H ->
  [/\ sub G (G \cg H), sub H (G \cg H)
    & forall W, equivalence W -> sub G W -> sub H W ->
      sub (G \cg H) W].
Proof.
move=> eG eH sG.
have gg: sgraph G by fprops.
have gh: sgraph H by fprops.
have gc: sgraph  (G \cg H) by apply: compg_graph.
ee.
- move=> y yG.
  move:  (gg _ yG) => py.
  rewrite - py in yG.
  apply/ compg_P;split=>//;exists (P y)=>//; equiv_tac=>//.
 rewrite - sG;  substr_tac.
- move=> y yH.
  move:  (gh _ yH) => py.
  rewrite - py in yH.
  apply/compg_P; split=>//;exists (Q y)=>//; equiv_tac=>//.
 rewrite sG;  substr_tac.
- move=> w ew gW hW t.
  move /compg_P=> [tp [y JH JG]].
  move: (gW _ JG) (hW _ JH)=> J1G J2G.
  have: inc (J (P t) (Q t)) w by equiv_tac.
  by rewrite tp.
Qed.
```

We know that the domain of an equivalence is the substrate. We show here that the same is true for the domain.

```
Lemma range_is_substrate g:
  equivalence g -> range g = substrate g.
```

```
Proof.
move=> eg; rewrite /substrate; set_extens x.
  move => pa; fprops.
move:(eg) => [fgg  _ _ _].
case /setU2_P => //. move /(domainP fgg)=> [y Jg].
apply/(rangeP fgg); exists y; equiv_tac.
Qed.
```

If G is an equivalence on E then $G \subset E \times E$.

```
Lemma sub_coarse g:
  equivalence g ->  sub g (coarse (substrate g)).
Proof.
move=> eg;move:(eg) => [fgg  _ _ _].
move: (sub_graph_setX fgg).
by rewrite range_is_substrate // domain_sr.
Qed.
```

The set of all graphs of equivalences on E is a subset of $\mathfrak{P}(E \times E)$, according to the two previous lemmas. We can consider the intersection of all these equivalences that contain G or H (there is at least one, the coarsest equivalence). The intersection is the smallest.

```
Lemma exercise6_6d G H:
  equivalence G -> equivalence H -> substrate G = substrate H ->
  G \cg H = H \ch G ->
  (G \cg H) = intersection(Zo (powerset (coarse (substrate G)))
    (fun W => [/\ equivalence W, sub G W & sub H W])).
Proof.
move=> eG eH sG cGH.
set (E:= substrate G).
have sGE: sub G (coarse E) by rewrite /E; apply: sub_coarse.
have sHE: sub H (coarse E) by rewrite /E sG; apply: sub_coarse.
move: (exercise6_6c eG eH sG)=> [sGc sHc lew].
set_extens t => ts.
  apply: setI_i.
    exists (coarse E); apply: Zo_i; first by apply: setP_Ti.
    split=> //;apply: coarse_equivalence.
  by move=> y /Zo_P [_ [ey gy hy]]; apply: (lew _ ey gy hy).
move: cGH;rewrite - exercise6_6a // => cGH.
apply: (setI_hi  (y:=(G \cg H)) ts); apply: Zo_i; last by done.
apply /setP_P;rewrite /E - (exercise6_6b eG eH sG); apply: sub_coarse=>//.
Qed.
```

7. *Let* G_0, G_1, H_0, H_1 *be the graphs of four equivalences on a set* E *such that* $G_1 \cap H_0 = G_0 \cap H_1$ *and* $G_1 \circ H_0 = G_0 \circ H_1$. *For each* $x \in E$, *let* R_0 *(resp.* S_0*) be the relation induced on* $G_1(x)$ *(resp.* $H_1(x)$*) by the equivalence relation* $(x, y) \in G_0$ *(resp.* $(x, y) \in H_0$*). Show that there exists a bijection of the quotient set* $G_1(x)/R_0$ *onto the quotient set* $H_1(x)/S_0$. *(if* $A = G_1(x) \cap H_1(x)$*, show that both quotient sets are in one-to-one correspondence with the quotient set of* A *by the equivalence relation induced by* R_0 *on* A*; this relation is equivalent to that induced by* S_0 *on* A*).*

This exercice is missing in the French edition. We think that the exercise is wrong, but do not have a counterexample.

```
Remark exercise6_7 G0 G1 H0 H1 E x:
```

```
  equivalence G0 -> substrate G0 = E ->
  equivalence H0 -> substrate H0 = E ->
  equivalence G1 -> substrate G1 = E ->
  equivalence H1 -> substrate H1 = E ->
  G1 \cap H0 = G0 \cap H1 ->
  G1 \cg H0 = G0 \cg H1 ->
  inc x E -> (
    let G1x := direct_image G1 (singleton x) in
      let H1x := direct_image H1 (singleton x) in
        let R0 := induced_relation G0 G1x in
          let S0 := induced_relation H0 H1x in
            equipotent (quotient R0) (quotient S0)).
Proof.
Abort.
```

8. *Let* E, F *be two sets, let* R *be an equivalence relation on* F*, and let* f *be a mapping of* E *into* F*. If* S *is the equivalence relation which is the inverse image of* R *under* f*, and if* $A = f\langle E\rangle$*, define a canonical bijection of* E/S *onto* A/R_A.

The first thing to do is to show that S and R_A are equivalence relations.

```
Lemma exercise6_8 f r:
  equivalence r -> function f -> target f = substrate r ->
 (exists g, bijection_prop g (quotient (inv_image_relation f r))
        (quotient (induced_relation r (image_of_fun f)))).
Proof.
move =>  er ff tf.
set (s := inv_image_relation f r).
set (A:= (image_of_fun f)).
set (Ra := induced_relation r A).
have ia: (iirel_axioms f r) by red; intuition.
have rf: A = range (graph f). Check f_range_graph.
  rewrite /A/image_of_fun -image_by_fun_source // image_by_fun.
have es:equivalence s by rewrite /s; apply: iirel_relation.
have iA: induced_rel_axioms r A.
  by split => //; rewrite -tf rf; apply: corresp_sub_range; move: ff=> [h _].
have eR: equivalence Ra by rewrite /Ra;apply: induced_rel_equivalence.
```

Let's quote the properties of `iirel_classP` and `induced_rel_classP`: If X is a class modulo R then $f^{-1}\langle X\rangle$ is a class modulo S (if nonempty) and conversely. Classes for R_A are nonempty sets of the form $A\cap X$ where X is a class for R. If a is a class for S we take X such that $a = f^{-1}\langle X\rangle$, and consider $b = A\cap X$. This gives our function. We can do the reverse operation.

We denote by $f_1(a,X)$ the property $a = f^{-1}\langle X\rangle$, $a\cap A\neq\emptyset$ and $X\in F/R$. We denote by $f_2(a)$ a class that satisfies this property, from which we deduce $f_3(a)$ a class for R_A.

```
set (f1:= fun x y =>  [/\ classp r y,
    nonempty (y \cap A) & x = inv_image_by_fun f y]).
have qsp:forall x, inc x (quotient s) -> exists y, f1 x y.
  by move=> x /(setQ_P es); move /(iirel_classP ia); rewrite -rf.
set (f2:= fun x => choose (fun y => f1 x y)).
have f2p: (forall  x, inc x (quotient s) -> f1 x (f2 x)).
  move=> x xq; rewrite /f2;apply: choose_pr; apply: (qsp _ xq).
set (f3:= fun x => (f2 x) \cap A).
have f3p: (forall x,  inc x (quotient s) -> inc (f3 x) (quotient Ra)).
  move=> x xs; rewrite /Ra; move: (f2p _ xs) => [pa pb pc].
```

```
  apply/(setQ_P eR); apply/(induced_rel_classP iA).
  by exists (f2 x).
```

It is now obvious to find a function from E/S to A/R_A.

```
set (g:= Lf f3 (quotient s) (quotient Ra)).
have sgf: sgraph (graph f) by fprops.
exists g; rewrite /g;split; aw; apply: lf_bijective =>//.
```

Our function is injective. Let $X = f_2(a)$ and $X' = f_2(a')$. From $g(a) = g(a')$ we get $f_3(a) = f_3(a')$, namely $X \cap A = X' \cap A$. This is a nonempty set, it constains an element of the form $f(z)$. We have $a = f^{-1}\langle X\rangle$ and $a' = f^{-1}\langle X\rangle'$. These two classes have a common element z, hence are equal.

```
  move => u v uq vq; rewrite /f3 => ii.
  move: (f2p _ uq)(f2p _ vq); rewrite /f1; move=> [cfu niu uv][cfv niv vv].
  move: niv=> [y] /setI2_P [y2v yiA].
  have y2u: inc y (f2 u).
     apply: (@setI2_1 (f2 u) A);rewrite ii; fprops.
  have : inc y (range (graph f)) by rewrite -rf.
  move /(rangeP sgf)=> [x Jg].
  have xb: (inc x v) by rewrite vv; apply/iim_graph_P;ex_tac.
  have xu:(inc x u) by rewrite uv; apply/iim_graph_P; ex_tac.
  move : uq vq; move/(setQ_P es) => c1 /(setQ_P es) => c2.
  case (class_dichot es c1 c2) => // dj; red in dj.
  by empty_tac1 x; apply: intersection2_inc.
```

Surjectivity is easy. Take $y \in A/R_A$. There is some $x \in F/R$ such that $y = x \cap A$ and we want to find $u \in E/S$ such that $g(u) = x \cap A$, $u = f^{-1}\langle x\rangle$. Define $u = f^{-1}\langle x\rangle$. The construction of g uses the axiom of choice, so that we must show uniqueness, namely $x = f_2(u)$. This is a consequence of the fact these two classes have a common element.

```
move=> y; move /(setQ_P eR) /(induced_rel_classP iA)=> [x [cx nex yi]].
set (u:= inv_image_by_fun f x).
have uq: inc u (quotient s).
   by apply/ (setQ_P es) /(iirel_classP ia); exists x; rewrite -rf.
ex_tac.
rewrite /f3 yi.
move:(f2p _ uq); rewrite /f1; move=> [cf2 ni ui].
move: nex=> [t] /setI2_P [tx].
rewrite {1} rf ; move/ (rangeP sgf)=> [z Jg].
have: inc z u by apply/iim_graph_P; ex_tac.
rewrite  {1} ui; move/iim_graph_P => [t' t'2u Jg'].
have tt': t = t' by move: (Vf_pr ff Jg) (Vf_pr ff Jg') => <-.
suff: f2 u = x by move=>->.
case(class_dichot er cf2 cx)=> // di; red in di.
empty_tac1 t; apply setI2_i =>//; ue.
Qed.
```

9. *Let* F, G *be two sets, let* R *be an equivalence relation of* F, *let* p *be the canonical mapping of* F *onto* F/R *and let* f *be a surjection of* G *onto* F/R. *Show that there exists a set* E, *a surjection* g *of* E *onto* F *and a surjection* h *of* E *onto* G *such that* $p \circ g = f \circ h$.

The set E is the disjoint union of F and G, we write it as $E_a \cup E_b$.

```
Lemma exercise6_9 F G p f r:
  equivalence r ->  F = substrate r -> p = canon_proj r ->
  surjection f -> source f = G -> target f = quotient r ->
  exists E g h,
  [/\ surjection_prop g E F, surjection_prop h E G & p \co g = f \co h].
Proof.
move=> er sr xr sjf sf tf.
have ff: function f by fct_tac.
set (a:= C0); set (b:= C1).
have ba: b <> a by rewrite /a /b; apply: two_points_distinctb.
set Ea:= F \times (singleton a).
set Eb:= G \times (singleton b).
set E:= Ea \cup Eb.
have gE: sgraph E by move => T /setU2_P; case; move /setX_P => [ok _].
have xep: forall x, inc x E -> (Q x =a \/ Q x = b).
  by move=> x /setU2_P; case; move /setX_P => [_ _ ] /set1_P; auto.
have xgp:forall x, inc x G -> inc (Vf f x) (quotient r).
  move=> x xg; rewrite - tf;apply: Vf_target => //; ue.
have xgp1:forall x, inc x G -> inc (rep (Vf f x)) F.
 move=> x xG; rewrite sr;fprops.
```

We consider the function g; it is the identity on E_a if we identify E_a with F, so that the image is F. Let $x \in \mathrm{E}_b$; we can identify E_b with G, hence assume $x \in \mathrm{G}$ so that $f(x) \in \mathrm{F/R}$. We define $g(x)$ to be a representative of the class of $f(x)$. This is an element of F. We have $p(g(x)) = f(x)$.

```
set (gz :=fun z=> Yo (Q z = a) (P z) (rep (Vf f (P z)))).
have gzP:forall z, inc z Ea -> gz z = P z.
  move=> z /setX_P [_ _] /set1_P h; rewrite /gz Y_true //.
have gzp': forall z, inc z Ea -> inc (gz z) F.
  by move=> z zEa; rewrite gzP//; move /setX_P : zEa => [_ ok _].
have gzQ:forall z, inc z Eb -> gz z = rep (Vf f (P z)).
  move=> z/setX_P [_ _] /set1_P => h; rewrite /gz Y_false //;  ue.
have gzq':forall z, inc z Eb -> inc (gz z) F.
  by move=> z zE; rewrite gzQ //; apply: xgp1;  move /setX_P : zE => [_ ok _].
have tag:lf_axiom gz E F.
  move=> t; case /setU2_P; [apply: gzp'| apply: gzq'].
set (g:= Lf gz E F).
have sj: surjection g.
  rewrite /g;apply: lf_surjective =>//;  move=> y yF.
  have p1: inc  (J y a) Ea by rewrite /Ea; fprops.
  have p2: inc  (J y a) E by rewrite /E; aw; intuition.
  by ex_tac;  rewrite gzP; aw.
have gp: forall x, inc x Eb -> Vf (canon_proj r) (gz x)  = Vf f (P x).
  move=> x xEb.
  have gzs:  inc (gz x) (substrate r) by rewrite - sr; apply: gzq'.
  have xE: inc x E by move: xEb; rewrite /E;aw; intuition.
  aw;rewrite gzQ //; apply: class_rep=>//; apply: xgp.
  by move /setX_P : xEb => [_ ok _].
```

We define now h similarly.

```
set (ha:= fun x => rep (inv_image_by_fun f(singleton(Vf (canon_proj r)x)))).
have haF:forall x, inc x F ->
    ha x = rep (inv_image_by_fun f (singleton (class r x))).
  move=> x xF; rewrite /ha; aw; ue.
```

```
have haF':forall x, inc x F ->
    sub (inv_image_by_fun f (singleton(class r x))) G.
  move=> x xF t /iim_graph_P [u _ jg]; rewrite - sf; Wtac.
have haF'': forall x, inc x F ->
    inc (ha x) (inv_image_by_fun f (singleton (class r x))).
  move => x xF; rewrite haF //; apply: rep_i.
  have ct: inc (class r x) (target f) by rewrite tf; rewrite sr in xF; fprops.
  move:((proj2 sjf) _  ct)=> [u us]; move => <-.
  exists u; apply /iim_graph_P; ex_tac; apply: Vf_pr3=>//.
have haG: forall x, inc x F -> inc (ha x) G.
  by move=> x xF; apply: (haF' _ xF); apply: haF'' =>//.
set(hz:= fun z=> Yo (Q z = a) (ha (P z)) (P z)).
have hzG: forall z, inc z E -> inc (hz z) G.
  rewrite /hz;move=> z /setU2_P [] /setX_P [_ pa] /set1_P ->; Ytac0 => //.
   by apply: haG.
set(h:=Lf hz E G).
have sh: surjection h.
  rewrite /h;apply: lf_surjective=>//.
  move=> y yG.
  have JEb:inc (J y b) Eb by rewrite /Eb;aw; fprops.
  have JE: (inc (J y b) E) by rewrite /E; aw; intuition.
  by ex_tac; rewrite /hz; aw; rewrite Y_false //.
have WWh: forall x, inc x Ea -> Vf f (hz x) = Vf (canon_proj r)  (P x).
  move=> x xEa.
  have xE: inc x E by rewrite /E; aw; intuition.
  have Ps: inc (P x) (substrate r) by rewrite - sr -gzP//; apply: gzp'.
  rewrite/h /hz; aw.
  move /setX_P: xEa=> [px PF] /set1_P ->; Ytac0.
  move /iim_graph_P: (haF'' _ PF) => [u ] /set1_P <- Jg; Wtac.
```

We are now ready to prove the main result.

```
exists E; exists g; exists h; rewrite /surjection_prop/g/h;aw;split => //.
have cpg: p \coP g.
  split; first by rewrite xr;apply: canon_proj_f.
     by fct_tac.
  rewrite xr /g; aw; ue.
have cfh: composable f h by split => //; try fct_tac; rewrite /h; aw.
have sg: source g = source h by  rewrite /g/h; aw.
have tp: target p = target f by  rewrite xr; aw.
move: sj => [fg _].
apply: function_exten; try fct_tac; aw.
```

The non-obvious point is to show $p(g(x)) = f(h(x))$.

```
move=> x xE /=; aw.
move /setU2_P: (xE) => [] xE'.
  have Ps: inc (P x) (substrate r) by rewrite - sr -gzP //; apply: gzp'.
  rewrite WWh // /g; aw; rewrite gzP // xr; aw.
rewrite xr gp /h /hz; aw =>//; rewrite Y_false //.
by move /setX_P: xE'=> [_ _ ] /set1_P ->.
Qed.
```

10. *(a) if* $\mathrm{R}\{x,y\}$ *is any relation, then "*$\mathrm{R}\{x,y\}$ *and* $\mathrm{R}\{y,x\}$*" is a symmetric relation. Under what condition is it reflexive on a set* E*?*

(b) Let $R\{x,y\}$ be a reflexive and symmetric relation on a set E. Let $S\{x,y\}$ be the relation "There exists an integer $n>0$ and a sequence $(x_i)_{0\le i\le n}$ of elements of E such that $x_0=x$, $x_n=y$ and for each index i such that $0\le i<n$, $R\{x_i,x_{i+1}\}$". Show that $S\{x,y\}$ is an equivalence relation on E and that its graph is the smallest of all graphs of equivalences on E which contain the graph of R. The equivalence classes with respect to S are called the connected components of E with respect to the relation R.

*(c) Let $\mathfrak{F}$ be the set of subsets A of E such that for each pair of elements (y,z) such that $y\in A$ and $z\in E-A$, we have " not $R\{y,z\}$". For each $x\in E$ show that the intersection of the sets $A\in\mathfrak{F}$ such that $x\in A$ is the connected component of x with respect to the relation R.**

Part a is trivial.

```
Lemma set1_pr2: forall a X, inc a X -> small_set X -> X = singleton a.
Proof.
  by move => w W tX sX; apply (set1_pr tX) => u zX; exact: (sX _ _ zX tX).
Qed.

Section Exercice6_10.
Lemma Exercise6_10_a (r:relation):
  symmetric_r (fun  x y => r x y /\ r y x).
Proof. by move=> x y; case. Qed.

Lemma exercise6_10_b r E:
  reflexive_re r E -> reflexive_re (fun  x y => r x y /\ r y x) E.
Proof.  move => rr y; split; [by move/rr | by case; move /rr]. Qed.
```

We consider now a context in which R is reflexive and symmetric on E.

```
Variables (R:relation) (E:Set).
Hypotheses (A1: reflexive_re R E)(A2: symmetric_r R)
  (A3: forall x y, R x y -> inc x E).
```

Defining the relation S is easy.

```
Inductive chain:Type :=
  chain_pair: Set -> Set -> chain
 | chain_next: Set> -> chain -> chain.
Fixpoint chain_head x :=
  match x with chain_pair u _ => u | chain_next u _ => u end.
Fixpoint chain_tail x :=
  match x with chain_pair _ u => u | chain_next _ u => chain_tail u end.
Fixpoint chained_r x :=
  match x with chain_pair u v => R u v
    | chain_next u v => R u (chain_head v) /\ chained_r v
 end.
Definition relS x y := exists c:chain,
  [/\ chained_r c, chain_head c = x  & chain_tail c = y].
```

For the transitivity, we need to concatenate lists.

```
Fixpoint concat_chain x y : chain :=
  match x with chain_pair u _ => chain_next u y
| chain_next u v => chain_next u  (concat_chain  (x:=v) y) end.
```

```
Lemma head_concat x y:
  chain_head (concat_chain x y) = chain_head x.
Proof.  by case x. Qed.

Lemma tail_concat x y:
  chain_tail (concat_chain x y) = chain_tail y.
Proof.  by elim x. Qed.

Lemma chained_concat x y:
  chained_r x -> chained_r y -> chain_tail x = chain_head y ->
  chained_r (concat_chain x y).
Proof.
move=> cx cy txhy;elim: x cx txhy => [a b cp ct| a c r cp ct].
  split=>//; by  rewrite -ct //.
by move: cp => [pa pb];split; [rewrite head_concat | apply: r].
Qed.

Lemma transitiveS y x z: relS x y -> relS y z -> relS x z.
Proof.
move=> [c [cc hcx tcy]][c' [cc' hcy  tcz]].
exists (concat_chain c c'); split => //.
    apply: chained_concat=>//;  ue.
  by rewrite head_concat.
by rewrite tail_concat.
Qed.
```

For the symmetry, we need to reverse the list. One way to reverse the list L is to start with an empty list L′, and recursively add the head of L to the head of L′, as long as L is not empty. In this case, L and L′ have at least two elements, this gives some special cases to deal with.

```
Fixpoint reconc_chain (x y:chain) :chain:=
  match x with chain_pair u v => chain_next v (chain_next u y)
   | chain_next u v => reconc_chain v (chain_next u y) end.

Lemma tail_reconc x y: chain_tail (reconc_chain x y) = chain_tail y.
Proof. by move: x y; elim=> [a b y | a c r] // y; by rewrite r. Qed.
Lemma head_reconc x y: chain_head (reconc_chain x y) = chain_tail x.
Proof. by move: x y; elim => [a b y | a c r] // y; by rewrite r. Qed.
Lemma chained_reconc x y: chained_r x -> chained_r y ->
  R (chain_head y) (chain_head x) ->  chained_r (reconc_chain x y).
Proof.
move: x y; elim => [a b y c cy | P c r]=>//=; auto.
move=> y [rPh cc] cy RhyP; apply: r=>//; split => //; apply: A2.
Qed.
```

We define now the reverse.

```
Fixpoint chain_reverse x:=
  match x with chain_pair u v => chain_pair v u
    | chain_next u v =>
      match v with chain_pair u' v' => chain_next v' (chain_pair u' u)
        | chain_next u' v' => reconc_chain v' (chain_pair u' u)
      end end.

Lemma head_reverse x: chain_head (chain_reverse x) = chain_tail x.
Proof. elim x =>// y;elim =>// P c h h1 /=; apply: head_reconc. Qed.
```

```
Lemma tail_reverse x: chain_tail (chain_reverse x) = chain_head x.
Proof. elim x =>// y;elim =>// P c h h1 /=; apply: tail_reconc. Qed.

Lemma chained_reverse x: chained_r x -> chained_r (chain_reverse x).
Proof.
elim x; first by move=> a b; simpl;  auto.
move=> a; elim; first by move => b c; simpl; intuition.
move=> b c hr hr1 /=  [Rab [Rbc cc]].
apply: chained_reconc=>//; simpl; auto.
Qed.

Lemma symmetricS x y: relS x y -> relS y x.
Proof.
move=> [c [cc hcx tcy]].
exists (chain_reverse c); split.
    apply: chained_reverse =>//.
  rewrite head_reverse //.
rewrite tail_reverse //.
Qed.
```

We make use of A3 for the first time here. It says that if x is related by S, it is in E. As a consequence our relation is an equivalence relation and its graph is an equivalence on E.

```
Lemma equivalenceS: equivalence_re relS E.
Proof.

split; first by split; red; [ apply: symmetricS | apply: transitiveS].
move=> x; split.
  by move=> xE; exists (chain_pair x x);split => //;  apply /A1.
move=>  [c [cc hcx _]].
elim: c cc hcx => [a b | a c _] /= h <-; [|move: h=> [h _]]; apply: (A3 h).
Qed.

Definition Sgraph := graph_on relS E.
Lemma equivalence_Sgraph: equivalence Sgraph.
Proof.
apply equivalence_from_rel; split; [apply: symmetricS | apply: transitiveS ].
Qed.

Lemma substrate_Sgraph: substrate Sgraph = E.
Proof.
apply: graph_on_sr => x xE.
by exists (chain_pair x x); split => //; apply/ A1.
Qed.
```

We can now show that this is the smallest relation. If r is an equivalence implied by R, the transitivity says that two elements (in particular head and tail) of a `chained_r` chain are related by r.

```
Lemma S_is_smallest r: equivalence r ->
  (forall x y, R x y -> inc (J x y) r) -> sub Sgraph r.
Proof.
move => r er pr p.
have aux:(forall w, chained_r w -> inc (J (chain_head w) (chain_tail w)) r).
  elim => [a b | a c h [aux cc]] //=; first by apply: pr.
```

```
  move: (h cc)(pr _ _ aux) => r1 r2; equiv_tac.
move /Zo_P => [pp  [c [cc [hcx htx]]]].
have <-: (J (P p)(Q p) = p) by move/setX_P: pp => [pp _].
by rewrite -hcx -htx ; apply: aux.
Qed.
```

We define here the set $\mathfrak{F}$ and some set C(x). We have to show that this is the class of x for S.

```
Definition setF:= Zo (powerset E)(fun A => forall y z, inc y A ->
  inc z (E -s A) -> not (R y z)).
Definition connected_comp x := intersection(Zo setF (fun A => inc x A)).
```

We first rewrite the condition on $\mathfrak{F}$, then prove that every element of $\mathfrak{F}$ is stable by S, hence contains equivalence classes. Each equivalence class is in $\mathfrak{F}$. The result is then obvious.

```
Lemma setF_pr A a b:
  inc A setF -> inc a A -> R a b -> inc b A.
Proof.
move /Zo_P => [] /setP_P AE Ap aA Rab.
case (inc_or_not b A)=> // nba.
have bc: inc b (E -s A) by apply:setC_i =>//; apply: (A3 (A2 Rab)).
by case (Ap _ _  aA bc).
Qed.

Lemma setF_pr2 A a b:
  inc A setF -> inc a A -> relS a b -> inc b A.
Proof.
move=> As aA [c [cc hcx <-]].
rewrite - hcx in aA; clear hcx.
elim: c cc aA.
   move=> u v /= Ruv uA; apply: (setF_pr As uA Ruv).
move=> u c h /= [uh cc] uA.
apply: h=>//;apply: (setF_pr As uA uh).
Qed.

Lemma setF_pr3 A a: inc A setF -> inc a A -> sub (class Sgraph a) A.
Proof.
move=> As aA t /(class_P equivalence_Sgraph) /(graph_on_P2 equivalenceS).
apply: (setF_pr2 As aA).
Qed.

Lemma setF_pr4 a: inc a E -> inc (class Sgraph a) setF.
Proof.
move=> aE; rewrite /setF.
move: equivalence_Sgraph => e1.
move: equivalenceS => e2.
apply: Zo_i.
  apply/setP_P; rewrite - substrate_Sgraph; apply: (sub_class_substrate e1).
move=> y z ya /setC_P [zE nzc]; dneg yz; apply/(class_P e1).
suff: related Sgraph z a by move=> aux; equiv_tac.
have : related Sgraph y a by  move/(class_P e1):ya=> h; equiv_tac.
move /(graph_on_P2 e2) => ra; apply/(graph_on_P2 e2); apply: transitiveS ra.
by exists (chain_pair z y); split => //; apply A2.
Qed.
```

```
Lemma connected_comp_class x: inc x E ->
  class Sgraph x = connected_comp x.
Proof.
move=> xE;set_extens t; rewrite /connected_comp.
  move=> tc;apply: setI_i.
    exists E; apply: Zo_i =>//; rewrite /setF; apply: Zo_i.
      aw; applt: setP_Ti.
    by move=> y z yE /setC_P [];
  move=> y /Zo_P [yS xy];apply: ((setF_pr3 yS xy) _ tc).
move: equivalence_Sgraph => eq.
have cx:(inc  (class Sgraph x)  (Zo setF (fun A => inc x A))).
  apply: Zo_i; first by apply: setF_pr4.
  apply/(class_P eq); rewrite - substrate_Sgraph in xE; equiv_tac.
move=> h;apply: (setI_hi h cx).
Qed.
```

11. *(a) Let* $\mathrm{R}\{x, y\}$ *be a reflexive and symmetric relation on a set* E. R *is said to be intransitive of order 1 if for any four distinct elements* x, y, z, t *of* E, *the relations* $\mathrm{R}\{x, y\}$, $\mathrm{R}\{x, z\}$, $\mathrm{R}\{x, t\}$, $\mathrm{R}\{y, z\}$ *and* $\mathrm{R}\{y, t\}$ *imply* $\mathrm{R}\{z, t\}$. *A subset* A *of* E *is said to be stable with respect to the relation* R *if* $\mathrm{R}\{x, y\}$ *for all* x *and* y *in* A. *If* a *and* b *are two distinct elements of* E *such that* $\mathrm{R}\{a, b\}$ *show that the set* $\mathrm{C}(a, b)$ *of elements* $x \in \mathrm{E}$ *such that* $\mathrm{R}\{a, x\}$ *and* $\mathrm{R}\{b, x\}$ *is stable and that* $\mathrm{C}(x, y) = \mathrm{C}(a, b)$ *for each pair of distinct elements* x, y *of* $\mathrm{C}(a, b)$. *The sets* $\mathrm{C}(a, b)$ *(for each ordered pair* (a, b) *such that* $\mathrm{R}\{a, b\}$*) and the connected components (Exercise 10) with respect to* R *which consist of a single element are called the constituents of* E *with respect to the relation* R. *Show that the intersection of two distinct constituents of* E *contains at most one element and that if* A, B, C *are three mutually distinct constituents at least one of the sets* $\mathrm{A} \cap \mathrm{B}$, $\mathrm{B} \cap \mathrm{C}$, $\mathrm{C} \cap \mathrm{A}$ *is empty.*

(b) Conversely, let $(\mathrm{X}_\lambda)_{\lambda \in \mathrm{L}}$ *be a covering of a set* E *consisting of non-empty subsets of* E *having the following properties: (1) if* λ *and* μ *are two distinct indices,* $\mathrm{X}_\lambda \cap \mathrm{X}_\mu$ *contains at most one element; (2) if* λ, μ, ν *are three distinct letters, then at least one of the three sets* $\mathrm{X}_\lambda \cap \mathrm{X}_\mu$, $\mathrm{X}_\mu \cap \mathrm{X}_\nu$, $\mathrm{X}_\nu \cap \mathrm{X}_\lambda$ *is empty. Let* $\mathrm{R}\{x, y\}$ *be the relation "There exists* $\lambda \in \mathrm{L}$ *such that* $x \in \mathrm{X}_\lambda$ *and* $y \in \mathrm{X}_\lambda$*"; show that* R *is reflexive on* E, *symmetric and intransitive of order 1, and that the* X_λ *are the constituents of* E *with respect to* R.

*(c) * Similarly, a relation* $\mathrm{R}\{x, y\}$ *which is reflexive and symmetric on* E *is said to be intransitive of order* $n-3$ *if, for every family* $(x_i)_{1 \le i \le n}$ *of distinct elements of* E, *the relations* $\mathrm{R}\{x_i, x_j\}$ *for each pair* $(i, j) \neq (n-1, n)$ *imply* $\mathrm{R}\{x_{n-1}, x_n\}$. *Generalize the results of (a) and (b) to intransitive relations of any order. Show that a relation which is intransitive of order* p *is also intransitive of order* q *for all* $q > p$.*

This is a follow-up to the previous exercise. We still assume that R is reflexive and symmetric on E (i.e., A1, A2 and A3 are assumed). We give a short definition and show that it is equivalent to the long one.

```
Definition intransitive1 := forall x y z t,
  x <> y ->  R x y -> R x z -> R x t -> R y z -> R y t -> R z t.

Lemma intransitive1pr :
  let intransitive_alt:= forall x y z t,
    x <> y -> x <> z -> x <> t -> y <> z -> y <> t -> z <> t ->
    inc x E -> inc y E -> inc z E -> inc t E ->
    R x y -> R x z -> R x t -> R y z -> R y t -> R z t in
    intransitive1 <-> intransitive_alt.
```

```
Proof.
rewrite /intransitive1; split.
move=> h x y z t  H0 _ _ _ _ _ _ _ _ _ H10 H11 H12 H13 H14.
apply: (h x y z t H0 H10 H11 H12 H13 H14).
move=> h x y z t nxy xy xz xt yz yt.
move: (A3 xy) (A3 yz)(A3 (A2 xz))(A3 (A2 yt)) => xE yE sE tE.
case (equal_or_not x z) => nxz; first by  ue.
case (equal_or_not x t) => nxt; first by apply: A2; ue.
case (equal_or_not y z) => nyz; first by ue.
case (equal_or_not y t) => nyt; first by apply: A2; ue.
case (equal_or_not z t)=> nzt; first  by rewrite nzt  -A1.
apply: (h x y z t) =>//.
Qed.
```

We now define and study C(a, b).

```
Definition stableR A:= forall a b, inc a A -> inc b A -> R a b.
Definition Cab a b:= Zo E (fun x => R a x /\ R b x).

Lemma Cab_stable a b: a<> b -> R a b -> intransitive1 ->
  stableR (Cab a b).
Proof.
move=> nab Rab i1; rewrite /Cab=> u v.
move /Zo_P=> [_ [r1 r2]] /Zo_P [_ [r3 r4]]; apply: (i1 a b u v) =>//.
Qed.

Lemma Cab_trans a b x y: a<> b -> R a b -> intransitive1 ->
  x<> y -> inc x (Cab a b) -> inc y (Cab a b) -> (Cab a b)= (Cab x y).
Proof.
move=> nab rab i1 nxy /Zo_P [xE [r1 r2]] /Zo_P [yE [r3 r4]].
set_extens t; move /Zo_P=> [tE [r5 r6]]; apply/Zo_i => //; split.
- apply: (i1 a b x t) =>//.
- apply: (i1 a b y t) =>//; apply: A2.
- apply: (i1 x y a t) =>//; first apply: (i1 a b x y)=>//; apply: A2=> //.
- apply: (i1 x y b t) =>//; first apply: (i1 a b x y)=>//; apply: A2=> //.
Qed.
```

A constituent is either a C or a connected component that has a single element. Let's characterize these. The non-trivial point here is to show that, if x is related to no other element than itself by R, the same is true for S. Hence, consider a chain from x to y. By symmetry, we have a chain from y to x for which we can use induction (if $y \sim x$, then $x = y$ by symmetry of R and equality; if $y \sim z$ and z is chained to x, we get $z = x$ by induction, hence $y \sim x$ and we proceed as above).

```
Lemma singleton_component A: sub A E ->
  ((inc A (quotient Sgraph) /\ singletonp A) <->
   (exists2 a, A = singleton a & forall b, R a b -> a = b)).
Proof.
move=> AE.
move: equivalence_Sgraph => e1.
move: equivalenceS => e2.
split.
  move=> [Asq [x Asx]]; exists x => //.
  move=> b Rb.
  have : related Sgraph x b.
```

```
    by apply/(graph_on_P2 e2); exists (chain_pair x b).
  move /(in_class_relatedP e1) => [y [cy xy]].
  have <- : A = y.
    move: Asq => /(setQ_P e1) => cA; case  (class_dichot e1 cy cA)=> //.
    move=> dy; red in dy; empty_tac1 x; apply:setI2_i => //.
    rewrite Asx; fprops.
  by rewrite Asx; move /set1_P.
move=> [x As Ap]; rewrite As; split; last by exists x.
have xse: inc x (substrate Sgraph).
  rewrite substrate_Sgraph; apply: AE; rewrite As; fprops.
have Aq: forall b, R b x -> b = x.
  by move => b ba; rewrite (Ap b) //; apply: A2.
suff: (class Sgraph x = singleton x).
  move => <-; apply /(setQ_P e1); apply: (class_class e1 xse).
apply: set1_pr; first by  apply /(class_P e1); equiv_tac.
move => w; move /(class_P e1) => aux.
have : (related Sgraph w x) by equiv_tac.
move /(graph_on_P2 e2) => [c [cc <-]].
elim: c cc.
  by move=> u v /= uv vx; rewrite vx in uv; apply: Aq.
by move=> p c h1 /= [Rp cc] tc; apply: Aq; rewrite - (h1 cc tc).
Qed.
```

The intersection of two distinct constituents has at least one element. This is obvious if the constituents are singletons. Consider $C(a,b)$ and $C(a',b')$. Assume that they contain u and v. If these elements are distinct then $C(a,b) = C(u,v) = C(a',b')$.

```
Definition is_constituant A :=
  (exists a, [/\ A = singleton a, inc a E & forall b, R a b -> a = b]) \/
  (exists a b, [/\ A = Cab a b, a<> b & R a b]).

Lemma constituant_inter2 A B:
  is_constituant A -> is_constituant B -> intransitive1 ->
  A = B \/ small_set (A \cap B).
Proof.
move=> cA cB i1.
case (equal_or_not A B); first (by auto); move => AB;right; move=> u v.
case cA.
  move=>[a [Aa aE ap]]; rewrite Aa.
  by move/setI2_P => [/set1_P -> _] /setI2_P [/set1_P -> _].
case cB.
   move=>[c [Ac cE cp]] _; rewrite Ac.
   by move/setI2_P => [_ /set1_P ->] /setI2_P [_ /set1_P ->].
move=> [a [b [Aab nab Rab]]] [a' [b' [Aab' nab' Rab']]].
case (equal_or_not u v)=>// nuv.
rewrite Aab Aab';move => /setI2_P [uA uB] /setI2_P [vA vB].
case AB; rewrite Aab' Aab.
rewrite (Cab_trans nab Rab  i1 nuv uB vB).
by rewrite (Cab_trans nab' Rab'  i1 nuv uA vA).
Qed.
```

Consider now the intersection of three constituents A, B and C. In the proof, we first eliminate the case where some of these sets are identical. Then the intersections are small sets (a singleton or empty). Bourbaki asks to show that at least one intersection is empty. The French edition of Bourbaki adds a last case: *ou les trois ensembles sont identiques*, which reads: the three intersections are identical, since this case can happen.

```
Lemma constitutant_inter3 A B C:
  is_constituant A -> is_constituant B -> is_constituant C ->  intransitive1 ->
  A = B \/ A = C \/ B = C \/  A \cap B = emptyset
  \/ A \cap C = emptyset \/ B \cap C = emptyset
  \/ (A \cap B = A \cap C  /\ B \cap C =  A \cap C).
Proof.
move=> cA cB cC i1.
case (equal_or_not A B); [by left| move=> nAB; right].
case (equal_or_not A C); [by left| move=> nAC; right].
case (equal_or_not B C); [by left| move=> nBC; right].
have ssAB: small_set (A \cap B).
  case (constituant_inter2 cA cB i1) =>//.
have ssAC: small_set (A \cap C).
  case (constituant_inter2 cA cC i1) =>//.
have ssBC: small_set (B \cap C).
  case (constituant_inter2 cB cC i1) =>//.
```

If A is a component $\{x\}$ and if $x \in C(a,b)$ then x is related to at least two distinct elements, absurd. Thus, the case wheer one set is a singleton is easy.

```
case: cA.
  move=> [a [Aa aE ap]]; case cB.
    move=> [b [Bb bE bp]].
    left; apply: disjoint_pr=> u ua ub; case nAB.
    by move: ua ub;rewrite Aa Bb; move /set1_P => -> /set1_P ->.
  move => [b1 [b2 [Bbb [nbb Rbb]]]].
  left; apply: disjoint_pr => u; rewrite Aa Bbb; move /set1_P => ->.
  move/Zo_hi=> [R1 R2]; case nbb.
  by rewrite -(ap _ (A2  R1)) (ap _ (A2 R2)).
move => [a1 [a2 [Aaa naa Raa]]].
case: cB.
  move=> [b [Bb bE bp]]; case cC.
    move=> [c [Cc [cE cp]]].
    right;right;left; apply: disjoint_pr=> u; rewrite Bb Cc; move /set1_P=> ->.
    by move/set1_P=> bc; case nBC;rewrite Bb Cc bc.
  move => [c1 [c2 [Ccc ncc Rcc]].
  right; right;left; apply: disjoint_pr => u; rewrite Bb Ccc; move/set1_P=> ->.
  by move /Zo_hi=> [R1 R2]; case ncc; rewrite -(bp _ (A2  R1))(bp _ (A2 R2)).
move => [b1 [b2 [Bbb nbb Rbb]]].
case: cC.
  move=> [c [Cc [cE cp]]].
  right;left;apply: disjoint_pr => u uA uC; move: uC uA; rewrite Aaa Cc.
  move /set1_P=> -> /Zo_hi [R1 R2]; case naa.
  by rewrite -(cp _ (A2  R1)) (cp _ (A2 R2)).
move => [c1 [c2 [Ccc ncc Rcc]]].
```

We assume $A = C(a_1, a_2)$, $B = C(b_1, b_2)$ and $C = C(c_1, c_2)$. Then either all intersections are empty, or there is $c \in A \cap B$, $b \in A \cap C$ and $a \in B \cap C$. We get $A \cap B = \{c\}$ since the intersection is a small set. We have three such relations. We have to show $a = b = c$. Note that one equality implies the other. Our result is true if $c \in C$ (since the $c \in A \cap C = \{b\}$. It is also true if $a = b$ (since then $a \in A \cap B = \{c\}$).

```
case (emptyset_dichot (A \cap B));[ by left |  move=> [c ci]; right].
case (emptyset_dichot (A \cap C));[ by left |  move=> [b bi]; right].
case (emptyset_dichot (B \cap C));[ by left |  move=> [a ai]; right].
have iAB: A \cap B = singleton c by apply: set1_pr2.
```

```
have iAC: A \cap C = singleton b by apply: set1_pr2.
have iBC: B \cap C = singleton a by apply: set1_pr2.
rewrite iAB iAC iBC.
suff: (inc c C).
   move=> cC.
   have cAC: inc c (A \cap C) by move/setI2_P: ci => []; fprops.
   have cBC: inc c (B \cap C) by move/setI2_P: ci => []; fprops.
   by rewrite (ssAC _ _ bi cAC) (ssBC _ _ ai cBC).
case (equal_or_not a b).
  move=> ab.
  have: inc a (A \cap B).
    apply setI2_i; [by rewrite ab;apply: (setI2_1 bi) | apply: (setI2_1 ai)].
  rewrite iAB; move /set1_P => <-; apply: (setI2_2 ai).
move=> nab.
```

The element c is related to a_1 and a_2. This makes 12 relations. We obtain three more relations by intransitivity: elements a, b and c are related. If $a \neq b$ we also deduce that c is related to c_1 and c_2. This says $c \in$ C

```
move: ai bi ci => /setI2_P [aB aC] /setI2_P [bA bC] /setI2_P [cA cB].
move: cA cB bA bC aB aC; rewrite Aaa Bbb Ccc.
move => /Zo_P [cE [Ra1c Ra2c]] /Zo_hi [Rb1c Rb2c].
move => /Zo_hi [Ra1b Ra2b] /Zo_hi [Rc1b Rc2b].
move => /Zo_hi [Rb1a Rb2a] /Zo_hi [Rc1a Rc2a].
move: (i1 _ _ _ _ ncc Rcc Rc1a Rc1b Rc2a Rc2b) => Rab.
move: (i1 _ _ _ _ nbb Rbb Rb1a Rb1c Rb2a Rb2c) => Rac.
move: (i1 _ _ _ _ naa Raa Ra1b Ra1c Ra2b Ra2c) => Rbc.
move: (i1 _ _ _ _ nab Rab (A2 Rc1a) Rac (A2 Rc1b) Rbc) => Rc1c.
move: (i1 _ _ _ _ nab Rab (A2 Rc2a) Rac (A2 Rc2b) Rbc) => Rc2c.
by apply: Zo_i.
Qed.

End Exercice6_10.
```

We consider now part b. Given an assumption on X and E we define a relation R.

```
Definition exercise6_11b_assumption X E:=
  [/\ union X = E,
   (forall A, inc A X -> nonempty A),
   (forall A B, inc A X -> inc B X -> A = B \/ small_set (A \cap B)) &
  (forall A B C, inc A X -> inc B X -> inc C X ->
    ( A=B \/ A = C \/ B = C \/  A \cap B = emptyset
      \/ A \cap C = emptyset
      \/ B \cap C = emptyset
      \/ (A \cap B = A \cap C /\  A \cap B = B \cap C)))].
Definition exercise6_11b_rel X x y := exists A, [/\ inc A X, inc x A & inc y A].
```

We start with trivial facts.

```
Lemma exercise6_11b1 E X:
  exercise6_11b_assumption X E -> reflexive_re (exercise6_11b_rel X) E.
Proof.
move=> [h _] x; rewrite /exercise6_11b_rel -h;split.
  move => xE; move: (setU_hi xE)=> [y ye xy];ex_tac.
move=> [y [yX xy _ ]]; apply: (setU_i xy yX).
```

```
Qed.

Lemma exercise6_11b2 X:
   symmetric_r (exercise6_11b_rel X).
Proof.
move=> E y; rewrite /exercise6_11b_rel.
by move=>[A [Ax xA yA]];  exists A.
Qed.
```

Let's show intransitivity. We assume the four points distinct. We have 5 relations, thus 5 sets. Denote by A_{xy} the set containing x and y. The element z is in A_{xz} and in A_{yz}. while the element t is in A_{xt} and in A_{yt}. If one set of the first list is the same as one set of the second list, the result is true. Otherwise, this gives four inequalities. Each one says that a set is a singleton. Obviously $A_{xz} \cap A_{xt} = \{x\}$ and $A_{yz} \cap A_{yt} = \{y\}$.

We have $x \in x_0 \cap x_1 \cap x_2$, $y \in x_0 \cap x_3 \cap x_4$, $z \in x_1 \cap x_3$ and $t \in x_2 \cap x_4$. We must show that z and t are in a common set. If one of x_1, x_3 is one of x_2, x_4, the result is obvious. We hence get four inequalities between sets. We know that $A \neq B$ implies that the intersection is empty or a singleton. Hence we get $x_1 \cap x_2 = \{x\}$ and $x_3 \cap x_4 = \{y\}$.

```
Lemma exercise6_11b3 E X: exercise6_11b_assumption X E ->
  let R := exercise6_11b_rel X in
    forall  x y z t,
      x <> y ->  x<>z -> x <> t -> y <> z -> y <> t -> z <> t ->
      R x y -> R x z -> R x t -> R y z -> R y t -> R z t.
Proof.
move=> [uX alne i2 i3] R x y z t nxy nxz nxt nyz nyt nzt
 [XY [XYX xXY yXY]]  [XZ [XZX xXZ zXZ]] [XT [XTX xXT tXT]]
 [YZ [YZX yYZ zYZ]]  [YT [YTX yYT tYT]].
case (equal_or_not XZ XT) => XZXT; first by exists XT; split => //; ue.
case (equal_or_not XZ YT) => XZYT; first by exists XZ; split => //; ue.
case (equal_or_not YZ XT) => YZXT; first by exists XT; split => //; ue.
case (equal_or_not YZ YT) => YZYT; first by exists YT; split => //; ue.
have iXZXT:  (XZ \cap XT = singleton x).
  apply: set1_pr2;first by fprops.
  case (i2 _ _ XZX XTX) =>h; [ contradiction | done].
have iYZYT:  (YZ \cap YT = singleton y).
  apply: set1_pr2;first by fprops.
  case (i2 _ _ YZX YTX) =>h; [ contradiction | done].
```

We assume $A_{xy} = A_{xz}$, and study the consequences. We get $A_{xy} = A_{yz}$ since y and z are two distinct elements in both sets. The case $A_{xy} = A_{yt}$ is trivial. Consider $A_{xt} = A_{yt}$; if this is true, we have $A_{xy} = A_{yt}$ since x and y are in both sets; the result is trivial. In the other case, we have three distinct sets $A_{xy} = A_{xz} = A_{yz}$, A_{xt} and A_{yt}. The intersections of two of them are nonempty. Since these intersections contain distinct elements x, y, t, the sets must be the same and the result is trivial.

```
case (equal_or_not XY XZ)=> XYXZ.
  have XYYZ: XY= YZ.
    have yp1:inc y (XY \cap YZ) by fprops.
    have zp1:inc z (XY \cap YZ) by rewrite XYXZ; fprops.
    case (i2 _ _ XYX YZX) =>// h; case nyz;apply: (h _ _ yp1 zp1).
  case (equal_or_not XY YT)=> XYYT; first by exists XY; aw; split => //; ue.
  case (equal_or_not XT YT) => XTYT.
    have xp: inc x (XY \cap YT) by aw; ue.
```

```
    have yp: inc y (XY \cap YT) by aw; ue.
    case (i2 _ _ XYX YTX) =>h; first by contradiction.
    elim nxy;apply: (h _ _ xp yp).
  case (i3 _ _ _ XYX XTX YTX); first by move=> h;exists XY; split => //; ue.
  case; first by move=> h.
  case; first by move=> h.
  case; first by move=> h;empty_tac1 x; aw; intuition.
  case; first by rewrite XYYZ; move=> h; empty_tac1 y; aw; intuition.
  case; first by move=> h; empty_tac1 t; aw; intuition.
  move=> [r1 r2].
  have : inc t (XT \cap YT) by aw; intuition.
  by rewrite -r2 XYYZ;move /setI2_P=> [tp _];  exists YZ.
```

We consider now the case $A_{xy} \neq A_{xz}$. The intersection of these sets is then $\{x\}$. It implies $A_{xz} \neq A_{yz}$ for otherwise y would be in $A_{xy} \cap A_{xz}$. Thus $A_{xz} \cap A_{yz} = \{z\}$.

```
have iXYXZ: (XY \cap XZ = singleton x).
  apply: set1_pr2; fprops; case  (i2 _ _ XYX XZX) => //.
case  (equal_or_not XZ YZ)=> XZYZ.
  have : inc y (singleton x) by rewrite - iXYXZ; aw;intuition; ue.
  move/set1_P => h; elim nxy =>//.
have iXZYZ:  (XZ \cap YZ = singleton z).
   apply: set1_pr2; fprops; case  (i2 _ _ XZX YZX) => //.
```

Now we compare A_{xy} and A_{yt}. Assume first equality. We proceed as above.

```
case (equal_or_not XY YT)=> XYYT.
  have XYXY:  (XY = XT).
    have xp: inc x (XY \cap XT)  by fprops.
    have tp: inc t (XY \cap XT) by rewrite XYYT; fprops.
    case (i2 _ _ XYX XTX) =>// h; elim nxt;apply: (h _ _ xp tp).
  case (equal_or_not XY YZ)=> XYYZ; first by exists XY; aw;split => //; ue.
  case (i3 _ _ _ XYX XZX YZX); first by move=> h;exists XY; intuition; ue.
  case; first by move=> h.
  case; first by move=> h.
  case; first by move=> h;empty_tac1 x; aw.
  case; first by move=> h; empty_tac1 y; aw.
  case; first by move=> h; empty_tac1 z; aw.
  move=> [r1 r2].
  have : inc z (XZ \cap YZ) by aw; intuition.
  rewrite -r2 XYYT;move /setI2_P=> [tp _]; exists YT; by aw.
```

Here $A_{xy} \neq Ayt$. The intersection is $\{y\}$. From this we get $A_{xt} \cap A_{yt} = \{t\}$. (same as proof as above).

```
have iXYYT: (XY \cap YT = singleton y).
  apply: set1_pr2; fprops; case  (i2 _ _ XYX YTX) => //.
case  (equal_or_not XT YT)=> XTYT.
  have : inc x (singleton y) by rewrite -iXYYT; aw;intuition; ue.
  move/set1_P => h; elim nxy =>//.
have iXTYT:  (XT \cap YT = singleton t).
  apply:set1_pr2;fprops:case  (i2 _ _ XTX YTX) => //.
```

One can prove $A_{xz} \cap A_{yt} = A_{xt} \cap A_{yz} = \emptyset$ but this relation is helpless. The only remaining pairs of sets are (A_{xy}, A_{xt}) and (A_{xy}, A_{xt}). The case $A_{xy} = A_{xt} = A_{yz}$ is trivially excluded. The cases $A_{xy} \neq A_{xt}$ and $A_{xy} \neq A_{yz}$ are easy.

```
case (equal_or_not XY XT)=> XYXT.
  case (equal_or_not XY YZ)=> XYYZ; first by elim  YZXT; ue.
  case (i3 _ _ _ XYX XZX YZX); first by move=> h.
  case; first by move=> h.
  case; first by move=> h.
  case; first by move=> h;empty_tac1 x; aw.
  case; first by move=> h;empty_tac1 y; aw.
  case; first by move=> h;empty_tac1 z; aw.
  rewrite iXYXZ iXZYZ; move=> [_ sxz].
  by elim nxz; apply: set1_inj.
case (i3 _ _ _ XYX XTX YTX); first by move=> h.
case; first by move=> h.
case; first by move=> h.
case; first by move=> h;empty_tac1 x; aw.
case; first by move=> h;empty_tac1 y; aw.
case; first by move=> h;empty_tac1 t; aw.
rewrite iXYYT iXTYT; move=> [sy st].
rewrite sy in st;  by elim nyt; apply: set1_inj.
Qed.
```

We show now that the elements of X are the constituents. Let $p_1(u)$ the property that u has the form C(a,b), $p_2(u)$ the property that u is a connected component formed of a single element. If u satisfies these conditions, then $u \in$ X. We are asked to show the converse. Assume that $u \in$ X; if it has at least two elements, it satisfies p_1. Assume that it has a single element x. Assume that there is no other set v containing x; then p_2 is true. Assume now that there is another set v containing x; then p_1 and p_2 are false. (Example: E has two elements a and b, X has two elements $\{a,b\}$ and $\{a\}$). The assumptions on X say: if v and v' are two sets containing x, then the intersection is a singleton. Denote by $p_3(u)$ this condition. It does not imply $u \in$ X.

Thus we prove the following.

```
Lemma exercise6_11b4 E X
 (R := exercise6_11b_rel X)
  (p1 := fun u => (exists a b, [/\ a<> b, R a b & u =
      Zo E (fun x => R a x /\ R b x)]))
  (p2:= fun u => (exists x, [/\ u = singleton x, inc x E &
      forall y, inc y E -> R x y -> x = y]))
  (p3:= fun u => (exists v, [/\ inc v X, u <> v, sub u v & singletonp u])):
  exercise6_11b_assumption X E ->
    [/\  (forall u, inc u X -> p1 u \/ p2 u \/ p3 u ),
      (forall u, p1 u -> inc u X) & (forall u, p2 u -> inc u X)].
```

We show here that singletons satisfy p_2 or p_3.

```
Proof.
move => E X [uXE alne i2 i3] R p1 p2 p3; split.
  move=> u uX.
  case (p_or_not_p (singletonp u)) => su.
    right; case  (p_or_not_p (p3 u)) => p3u; first by intuition.
    left; move: (su) => [x sx].
    rewrite sx; exists x; split => //.
      rewrite -uXE; apply: (@setU_i _  u) =>//; rewrite sx; fprops.
    move=> y yE Rxy; case (equal_or_not x y) =>//.
    move=> xy; move: Rxy=> [A [AX xA yA]].
    case p3u; exists A; split =>//.
```

```
    by dneg uA; move: yA; rewrite -uA sx;move /set1_P.
  by move=> t; rewrite sx; move/set1_P => ->.
```

Our set u is not empty, hence has an element y. We show here that if it has another element x, then $p_1(u)$ is satisfied. If x_0 is related to x and y, there exists two sets x_1 that contains y and x_0, and x_2 that contains x and x_0. We want to show $x_0 \in u$. This is clear if $x_1 = u$ or $x_2 = u$. Assume these two pairs distinct. If $x_1 = x_2$, the intersection $x_1 \cap u$ is a singleton, containing x and y, absurd. We can then use property (2).

```
constructor 1; red.
move: (alne _ uX) => [y yu]; exists y.
case (p_or_not_p (exists2 v, inc v u & v <> y)).
  move=> [x xu xy]; exists x; split; [auto | by exists u |].
  set_extens1 w.
      move=> wu; apply: Zo_i.
      rewrite - uXE; apply: (@setU_i _ u) =>//.
    split;exists u; split => //.
  move /Zo_P=> [wE [ [A [AX xA yA]]  [A' [AX' xA' yA']]]].
  case (equal_or_not A u)=> Au; first by rewrite -Au.
  case (equal_or_not A' u)=> Au'; first by rewrite -Au'.
  have xi: (inc x (u \cap A')) by aw.
  have yi: (inc y (u \cap A)) by aw.
  case (equal_or_not A A') => AA'.
    case (i2 _ _ uX AX)=> aux.
      by elim Au'; rewrite -AA' aux.
    rewrite -AA' in xi.
    by elim  xy; apply:(aux _ _ xi yi).
  move: (i3 _ _ _ AX AX' uX).
  case =>//; case =>//; case =>//.
  case; first by move=> h; empty_tac1 w; aw.
  case; first by move=> h; empty_tac1 y; aw.
  case; first by move=> h; empty_tac1 x; aw.
  move=> [h1 h2].
  rewrite setI2_C -h2 in xi.
  rewrite setI2_C -h1 in yi.
  case (i2 _ _ AX AX')=>// aux.
  elim xy; by apply: (aux _ _ xi yi).
```

To finish, we must show that a nonempty set that is not a singleton has at least two elements.

```
move=> h;elim su; exists y; apply: set1_pr1; first by ex_tac.
move => w wu;case (equal_or_not w y) =>// wy;  by elim h; ex_tac.
```

We show here that $p_1(u)$ implies $u \in$ X. Consider x and x_0 two distinct elements, and $u = \mathrm{C}(x, x_0)$. The two elements x and x_0 are related, this means that they are in a set x_1. We have $u = x_1$. The proof is the same as above.

```
(* last case *)
  move=> u [a [b [nab [A [AX [aA bA]]] uZ]]].
  suff: (u = A) by  move=> ->.
  rewrite uZ; set_extens t.
    move /Zo_P=> [tE [[A' [AX' [aA' bA']]] [A'' [AX'' [aA'' bA'']]].
    case (equal_or_not A A'')=> AA''; first by ue.
    case (equal_or_not A A')=> AA'; first by ue.
```

```
    have aAA: inc a (A \cap A') by fprops.
    have bAA: inc b (A \cap A'') by fprops.
    case (equal_or_not A' A'') => aux.
    case (i2 _ _ AX AX')=> // ss.
      rewrite -aux in bAA; elim nab; apply: (ss _ _ aAA bAA).
    case (i3 _ _ _ AX AX' AX'') =>//; case =>//; case =>//.
    case; first by move=> h; empty_tac1 a; aw.
    case; first by move=> h; empty_tac1 b; aw.
    case; first by move=> h; empty_tac1 t; aw.
    move=> [h1 h2]. rewrite - h1 in  bAA.
    case (i2 _ _ AX AX')=>// ss.
    elim nab; by apply: (ss _ _ aAA bAA).
  move=> tA; apply: Zo_i.
      rewrite -uXE;apply: (@setU_i _ A)=>//.
    by split;exists A.
```

We show that $p_2(u)$ implies $u \in \mathrm{X}$.

```
move=> u  [v [uv vE su]].
move: vE;rewrite  -uXE;  move/setU_P=> [y vy yX].
suff: u = y by move=> ->.
rewrite uv; symmetry; apply:set1_pr => // t tv.
symmetry;apply: su.
  rewrite -uXE;apply: (@setU_i _ y) =>//.
by exists y.
```

Part c. We do not know how to generalize. The last claim is obvious. Assume R intransitive of order $p-3$, let $q > p$ and consider q distinct elements, which are related (with the exception of x_{q-1} and x_q; discard the $q-p$ first elements. The missing relation is true by intransitivity.

Chapter 8

Summary

8.1 The axioms

We give here the list of all axiom schemes.
S1: If $\boldsymbol{A}$ is a relation in $\mathscr{T}$, the relation ($\boldsymbol{A}$ or $\boldsymbol{A}$) $\Longrightarrow$ $\boldsymbol{A}$ is an axiom of $\mathscr{T}$.
S2: If $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations in $\mathscr{T}$, the relation $\boldsymbol{A} \Longrightarrow$ ($\boldsymbol{A}$ or $\boldsymbol{B}$) is an axiom of $\mathscr{T}$.
S3: If $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations in $\mathscr{T}$, the relation ($\boldsymbol{A}$ or $\boldsymbol{B}$) $\Longrightarrow$ ($\boldsymbol{B}$ or $\boldsymbol{A}$) is an axiom of $\mathscr{T}$.
S4: If $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are relations in $\mathscr{T}$, the relation ($\boldsymbol{A} \Longrightarrow \boldsymbol{B}$) $\Longrightarrow$ (($\boldsymbol{C}$ or $\boldsymbol{A}$) $\Longrightarrow$ ($\boldsymbol{C}$ or $\boldsymbol{B}$)) is an axiom of $\mathscr{T}$.
S5: If $\boldsymbol{R}$ is a relation in $\mathscr{T}$, if $\boldsymbol{T}$ is a term in $\mathscr{T}$, and if $\boldsymbol{x}$ a letter, then the relation $(\boldsymbol{T}|\boldsymbol{x})\boldsymbol{R} \Longrightarrow (\exists \boldsymbol{x})\boldsymbol{R}$ is an axiom.
S6: Let $\boldsymbol{x}$ be a letter, let $\boldsymbol{T}$ and $\boldsymbol{U}$ be terms in $\mathscr{T}$, and let $\boldsymbol{R}\{\boldsymbol{x}\}$ a relation in $\mathscr{T}$; then the relation $(\boldsymbol{T} = \boldsymbol{U}) \Longrightarrow (\boldsymbol{R}\{\boldsymbol{T}\} \Longleftrightarrow \boldsymbol{R}\{\boldsymbol{U}\})$ is an axiom.
S7: If $\boldsymbol{R}$ and $\boldsymbol{S}$ are relations in $\mathscr{T}$, and if $\boldsymbol{x}$ is a letter, then the relation $((\forall \boldsymbol{x})(\boldsymbol{R} \Longleftrightarrow \boldsymbol{S})) \Longrightarrow (\tau_{\boldsymbol{x}}(\boldsymbol{R}) = \tau_{\boldsymbol{x}}(\boldsymbol{S}))$ is an axiom.
S8: Let $\boldsymbol{R}$ be a relation, let $\boldsymbol{x}$ and $\boldsymbol{y}$ be distinct letters, and let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be letters distinct from $\boldsymbol{x}$ and $\boldsymbol{y}$ which do not appear in $\boldsymbol{R}$. Then the relation

$$(\forall \boldsymbol{y})(\exists \boldsymbol{X})(\forall \boldsymbol{x})(\boldsymbol{R} \Longrightarrow (\boldsymbol{x} \in \boldsymbol{X})) \Longrightarrow (\forall \boldsymbol{Y})\,\mathrm{Coll}_{\boldsymbol{x}}((\exists \boldsymbol{y})((\boldsymbol{y} \in \boldsymbol{Y}) \text{ and } \boldsymbol{R}))$$

is an axiom.

The French edition has only four axioms since A3 is a theorem.
A1. $(\forall x)(\forall y)((x \subset y \text{ and } y \subset x) \Longrightarrow (x = y))$.
A2. $(\forall x)(\forall y)\mathrm{Coll}_z(z = x \text{ or } z = y)$.
A3. $(\forall x)(\forall x')(\forall y)(\forall y')(((x, y) = (x', y')) \Longrightarrow (x = x' \text{ and } y = y'))$
A4. $(\forall \mathrm{X})\mathrm{Coll}_{\mathrm{Y}}(\mathrm{Y} \subset \mathrm{X})$.
A5. There exists an infinite set.

8.2 The Zermelo Fraenkel Theory

An alternative to the Bourbaki theory is the Zermelo Fraenkel theory. It has the usual interpretation of the quantifiers $\forall$ and $\exists$, but not the symbol τ, thus is missing a choice function. With the notations of [5] the axioms are

B1. $\forall x \forall y[\forall z(z \in x \Longleftrightarrow z \in y) \Longrightarrow x = y]$ (Axiom of extent, A1).

B0. $\forall x \forall y \exists z \forall t[t \in z \Longleftrightarrow (t = x \text{ or } t = y)]$ (Axiom of the pair, A2).

B2. $\forall x \exists y \forall z [z \in y \iff \exists t (t \in x \text{ and } z \in t)]$ (Axiom of the union).

B3. $\forall x \exists y \forall z [z \in y \iff z \subset x]$ (Axiom of the set of subsets, A4).

B4. $\exists x \exists y [\forall z (z \notin y) \text{ and } y \in x \text{ and } \forall u [bu \in x \implies \exists v [v \in x \text{ and } \forall t (t \in v \iff t = u \text{ or } t \in u)]]]$ (Axiom of infinity).

SS. $\forall x_1 \ldots \forall x_k \{\forall x \forall y \forall y' [\mathrm{E}(x, y, x_1, \ldots, x_k) \text{ and } \mathrm{E}(x, y', x_1, \ldots, x_k) \implies y = y'] \implies \forall t \exists w \forall v [v \in w \iff \exists u [u \in t \text{ and } \mathrm{E}(u, v, x_1, \ldots, x_k)]]\}$ (Scheme of Replacement).

SC. $\forall x_1 \ldots \forall x_k \forall x \exists y \forall z [z \in y \iff (z \in x \text{ and } \mathrm{A}(z, x_1, \ldots, x))]$ (Scheme of comprehension).

AC. $\forall a \{[\forall x (x \in a \implies x \neq \emptyset) \text{ and } \forall x \forall y (x \in a \text{ and } y \in a \implies x = y \text{ or } x \cap y = \emptyset)] \implies \exists b \forall x \exists u (x \in a \implies b \cap x = \{u\})\}$ (Axiom of choice).

AF. $\forall x [x \neq \emptyset \implies \exists y (y \in x \text{ and } y \cap x = \emptyset)]$ (Axiom of foundation).

Comments. The Zermelo-Fraenkel theory consists in axioms B1, B2, B3, B4, and scheme SS. From SS, one can deduce SC and B0. The Zermelo theory consists in B1, B0, B2, B3, B4 and SC. It is a weaker theory. Axiom AF is independent of all other axioms, it excludes some weird sets; it is useful in modeling.

Scheme SS depends on a relation E that takes at least two arguments. Fix all parameters but the first two ones. Assume that $\mathrm{E}(x, y)$ is functional in y (i.e., if $\mathrm{E}(x, y) = \mathrm{E}(x, y')$ implies $y = y'$). Rewrite $\mathrm{E}(x, y)$ as $y = f(x)$. The scheme says that for all t, there is a w containing those v of the form $v = f(u)$ for some $u \in t$. Scheme SC says that for every relation $\mathrm{A}(z)$ (that may depend on other parameters), and for every set x there is a set w containing those $v \in x$ that satisfy A.

Consider now axiom B4. The parameter y has to be zero (a.k.a the empty set), and v has to be $u \cup \{u\}$. Denote this by $\mathrm{S}(u)$. Now B4 says: there exists a set x, containing zero, and such that $u \in x \implies \mathrm{S}(u) \in x$. In part two of this report, we shall define pseudo-ordinals. Then the set of finite pseudo-ordinals (which is also the set of finite cardinals with the definition of [5]) is the smallest set satisfying B4. Thus B4 is equivalent to the existence of this set. This axiom is equivalent to A5 (remember that it asserts existence of an infinite set, where "infinite" is a very complicated expression, since it depends on the addition of cardinals, see part two of this report).

Consider now axiom AC. It says that for every set a, if a is formed of non-empty, mutually disjoint sets, there exists a set b that meets each element of a exactly once. Denote by $f(x)$ the unique element of the intersection of x and b. Then (informally) f is a function such that $f(x) \in x$. More formally, the axiom is equivalent to: for every set A, there exists a function $f : \mathfrak{P}(\mathrm{A}) - \emptyset \to \mathrm{A}$ such that $f(x) \in x$. It is also equivalent to say that a product of non-empty sets is non-empty; it is also equivalent to Zermelo's Theorem (every set can be well-ordered, see part 2). We shall use Zermelo's Theorem in order to show that cardinals are well-ordered. A consequence of this fact is the Cantor-Bernstein theorem: if there is an injection from A into B and an injection of B into A, then there is a bijection of A onto B. But this result is independent of AC.

8.3 Changes from previous versions

We show some definitions and theorems, there were either removed or changed, with some explanations. For more details, see the previous version of this document.

The choice function. Let x be a set, $p(x)$ a property. Let $Q(p,x)$ be the property that, if there is a set y such that $p(y)$, then $p(x)$ is true, and if there is no such y, then x is the emptyset. We have two lemmas that say that, if we know that there exists y such $p(y)$ (resp., if we know the converse), then $Q(p,x)$ is equivalent to $p(x)$ (resp. $x = \emptyset$). In both cases, there exists x such that $Q(p,x)$ is true. By the excluded middle law, one of these two cases must be true. This allows us to define the choice function, using an inhabitant of the type of sets (in the definitions of Carlos Simpson below, E is `Type`, the type of sets, EP is `E→Prop`, and `Prop` is an inhabitant of E).

```
Definition refined_pr (p:EP) (x:E) :=
   (ex p -> p x) & ~(ex p) -> x = emptyset.

Lemma refined_pr_if : forall p x, ex p -> refined_pr p x = p x.
Lemma refined_pr_not : forall p x, ~(ex p) -> refined_pr p x = (x = emptyset).
Lemma exists_refined_pr : forall p, ex (refined_pr p).
Definition choose' := fun X : EP => chooseT X (nonemptyT_intro Prop).
Definition choose (p:EP) := choose' (refined_pr p).
```

Reasoning by cases. Let P be a proposition. The expression "$(P \wedge A) \vee (\neg P \wedge B)$" can be written in Coq as '`IF P then A else B`'. Consider now two sets a and b, and the relation '`IF P then x=a else x=b`' as a property $f(x)$ (for fixed P, a and b). If $f(x)$ holds, then either $x = a$, or $x = b$. Assume $a \neq b$. Then $f(a)$ is equivalent to P and $f(b)$ is equivalent to $\neg$P. By the excluded middle law, one of P or $\neg$P holds, so that exactly one of $f(a)$ or $f(b)$ holds. This means that we can apply the axiom of choice, and we get a function Y, that maps the predicate P to one of a or b. In the code that follows, a depend on a proof p of P and b on a proof q of $\neg$P. The proof irrelevance axiom then shows that the result depends only on the truth value of P.

```
Axiom proof_irrelevance : forall (P : Prop) (q p : P), p = q.
Definition by_cases (T : Type) (P : Prop) (a : P -> T)  (b : ~ P -> T) :=
  chooseT (fun x : T =>  (forall p : P, a p = x)
                       & (forall q : ~ P, b q = x))
    (by_cases_nonempty a b).
Lemma by_cases_if :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (p : P),
    by_cases a b = a p.
Lemma by_cases_if_not :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (q : ~ P),
    by_cases a b = b q.
```

A variant of Y where the arguments are of type A instead of being a set.

```
Definition Yt (A:Type): Prop -> (A->A->A):=
  fun P x y => by_cases (fun _ : P => x) (fun _ : ~ P => y).

Lemma Yt_if_rw : forall (A:Type)(P : Prop) (hyp : P) (x:A) y,
  Yt P x y = x.
Lemma Yt_if_not_rw :  forall (A:Type) (P : Prop) (hyp : ~ P) x  (y:A),
  Yt P x y = y.
```

The intersection of a family of sets $f(z)$ over I is the set of all $x \in$ E, such that, for all $z \in$ I, $x \in f(z)$. If I is non-empty, we choose for E some $f(z_0)$, in the other case, the intersection is the empty set. In the current implementation E is the union of the family, and the definition is much simpler.

```
Definition intersectiont (In:Set)(f : In->Set):=
  by_cases(fun H:nonemptyT In =>
      Zo (f (chooseT_any H)) (fun y => forall z : In, inc y (f z)))
  (fun _:~ nonemptyT In => emptyset).
```

Pairs. The definition of an orderded pair in the English Edition of Bourbaki, [2], is essentially that of the 1956 French Edition. Here the existence of an "ordered pair" is postulated. It is denoted as $\supset xy$, and later on as (x,y). The "axiom of the ordered pair" states that if $(x,y)=(x',y')$, then $x=x'$ and $y=y'$. The unique quantity x (defined by the Axiom of Choice) such that $z=(x,y)$ is called the "first projection", and denoted $\mathrm{pr}_1 z$. The unique quantity y such that $z=(x,y)$ is called the "second projection" and denoted $\mathrm{pr}_2 z$. Thus $z=(\mathrm{pr}_1 z,\mathrm{pr}_2 z)$ is equivalent to "z is an ordered pair".

Note that $\mathrm{pr}_1\emptyset$ and $\mathrm{pr}_2\emptyset$, are two well-defined sets, but whether $\emptyset$ is an ordered set or not is undecidable. Thus $\emptyset=(\mathrm{pr}_1\emptyset,\mathrm{pr}_2\emptyset)$ could be true or false.

```
Parameter J : Set -> Set -> Set.
Axiom axiom_of_pair : forall x y x' y' : Set,
    (J x y = J x' y') -> (x = x' & y = y').
Definition P (u : Set) :=
  choose (fun x : Set => ex (fun y : Set => u = J x y)).
Definition Q (u : Set) :=
  choose (fun y : Set => ex (fun x : Set => u = J x y)).
```

It is possible to define pairs without using an axiom. Essentially (x,y) is a doubleton $\{a,b\}$. If we take $a=\{\{x\}\}$ and $b=\{\emptyset,\{y\}\}$, we get the definition proposed by Wiener in 1914. The implementation of C. Simpson used $a=\{x\}$. Note that a has one element while b has two elements, so that (x,y) is a set with two distinct elements.

```
Definition pair_first (x y:Set):= singleton x.
Definition pair_second (x y:Set):= doubleton emptyset (singleton y).

Definition pair (x y : Set) :=
  doubleton (pair_first x y) (pair_second x y).

Lemma pair_distinct x y:
  pair_second x y <>  pair_first x y.
```

The Kuratowski definition of a pair (x,y) is $z=\{a,b\}$, where $a=\{x\}$ and $b=\{x,y\}$. This one is used in the French version of Bourbaki [3]. We have $\bigcap z=a\cap b=\{x\}$ so that $\bigcup\bigcap z=x$. We have $\bigcup z=a\cup b=\{x,y\}$. The complement of $\bigcap z$ in $\bigcup z$ is either empty (in case $x=y$) or $\{y\}$ (otherwise). This allows us to compute y from z.

In the current version, a different method is used. Consider the set of all t in $\bigcup z$ such that $(\mathrm{pr}_1 z,t)=\bigcup z$. Recall that $\bigcup z=\{x,y\}$, and $\mathrm{pr}_1 z=x$. So we consider the set E of all t, equal to x or y, such that $\{x,t\}=\{x,y\}$. Obviously $y\in$ E. If $t\in$ E, we have either $t=x$ or $t=y$, but in the first case $x=y$. Thus E $=\{y\}$ and $\bigcup$E $=y$.

```
Definition kpair x y := doubleton (singleton x) (doubleton x y).
Definition kpr1 x := union (intersection x).
Definition kpr2 x := let a := complement (union x) (intersection x) in
 Yo (a = emptyset) (kpr1 x) (union a).
```

Cartesian Product. Assume that a is a set, and f of type `Set→Set` is a function. One can consider the property: the pair (x, y) satisfies $x \in a$ and $y \in f(x)$.

```
Definition in_record a f (x : Set) :=
  is_pair x  & inc (P x) a   & inc (Q x) (f (P x)).

Record Cartesian_record a f : Set :=
  {Cartesian_first : a; Cartesian_second : f (Ro Cartesian_first)}.

Definition recordMap a f  (i : Cartesian_record a f) :=
  J (Ro (Cartesian_first i)) (Ro (Cartesian_second i)).

Lemma in_record_ex : forall a f (x : Set),
 in_record a f x -> exists i : Cartesian_record a f, recordMap i = x.

Lemma in_record_bounded : forall a f, Bounded.axioms (in_record a f).
```

There is a set containing all pairs (x, y) that satisfy $x \in a$ and $y \in f(x)$.

```
Definition record a f := Bounded.create (in_record a f).

Lemma record_in :  forall a f x, inc x (record a f) -> in_record a f x.
Lemma record_pr :  forall a f x,
   inc x (record a f) -> (is_pair x & inc (P x) a & inc (Q x) (f (P x))).
Lemma record_inc :  forall a f x, in_record a f x -> inc x (record a f).
Lemma record_pair_pr : forall a f x y,
    inc (J x y) (record a f) -> (inc x a & inc y (f x)).
Lemma record_pair_inc :  forall a f x y,
  inc x a -> inc y (f x) -> inc (J x y) (record a f).
```

A product is just a record where the function f is constant.

```
Definition product (a b : Set) := record a (fun x : Set => b).
```

Module Basic Realization. The following two axioms imply that $\mathscr{R}n$ is the n-th ordinal (in the von Neumann sense) for each natural number n. Thus $\mathscr{R}0 = \emptyset$, $\mathscr{R}1 = \{\emptyset\}$, $\mathscr{R}2 = \{\emptyset, \{\emptyset\}\}$ and so on.

```
Axiom nat_realization_O : forall x : Set, ~ inc x (Ro 0).
Axiom nat_realization_S :
          forall (n : nat) (x : Set),
          inc x (Ro (S n)) = (inc x (Ro n) \/ x = Ro n).
Lemma nat_zero_emptyset : Ro 0 = emptyset.
Lemma R_one_singleton_emptyset : Ro 1 = singleton emptyset.
```

In the framework of Simpson, any type was a set. So, he postulated that $\mathscr{R}x = x$, whenever x is a proposition, and that $\mathscr{R}p = \emptyset$ for any proof p of true (note that this proof is unique).

```
Axiom prop_realization : forall x : Prop, Ro x = x.
Axiom true_proof_realization_empty : forall t : True, Ro t = Ro 0.
```

By extensionality, `False` is $\emptyset$ and `True` is $\{\mathscr{R}\mathrm{I}\}$, where I is a proof of True. The previous axiom then says that `True` is $\{\emptyset\}$. By the excluded middle law, any proposition is true or false, so that `Prop` has exactly two elements, and is $\mathscr{R}2$.

```
Lemma false_emptyset : emptyset = False.
Lemma R_false_emptyset : Ro False = emptyset.
Lemma true_proof_emptyset : forall t : True, Ro t = emptyset.
Lemma true_singleton_emptyset : singleton emptyset = True.
Lemma R_true_singleton_emptyset : Ro True = singleton emptyset.

Lemma R_two_prop : Ro 2 = Prop.
```

Correspondences. In a first implementation, a correspondence was a set, more precisely, a functional graph on a set with three elements, Source, Target and Graph.

```
Definition create x y g:=
  denote Source x (denote Target y (denote Graph g stop)).
Definition like (a:E) := a = create(sourceC a) (targetC a)(graphC a).
Definition correspondence m:=
   like m & is_graph (graph m) & sub (domain (graph m)) (source m)
   & sub (range (graph m)) (target m).
```

Later on, a correspondence was a record; we had conversion functions between correspondences and triples, and an axiom of choice for correspondences.

```
Record correspondenceC:Type :=
    corresp{ source:Set; target:Set; graph :Set }.
Definition corr_value (x:correspondenceC):=
    J(graph x) (J (source x) (target x)).
Definition inv_corr_value z := corresp(P (Q z)) (Q (Q z)) (P z).

Definition choosef (p:correspondenceC -> Prop) :=
  chooseT (fun u=> (ex p -> p u)  &  ~(ex p) -> u = identity_fun emptyset)
  (nonemptyT_intro (corresp emptyset emptyset emptyset)).

Lemma choosef_pr : forall p, (ex p) -> p (choosef p).
```

For Bourbaki, an *equivalence on a set* E is a correspondence whose source and target are both equal to E, and whose graph F is such that the relation $(x, y) \in F$ is an equivalence relation on E. Note: the correspondence is uniquely defined by F, since E is the substrate of F; conversely, given an equivalence F on E, the domain and range of F is E, thus $F \subset E \times E$, and (E, E, F) is a correspondence.

```
  Definition equivalence_cor r:=
    source r = target r &
    equivalence (graph r) & source r = (substrate (graph r)).
  Definition graph_to_eq_cor g := corresp (domain g)(domain g) g.
```

8.4 Tactics

We have two data bases for `autorewrite` and one for `auto`.

```
Ltac aw := autorewrite with aw; trivial.
Ltac bw := autorewrite with bw; trivial.
Ltac fprops := auto with fprops.
```

This tactic `ex_middle u` solves the current goal by assuming in u that it is false. The tactic `dneg u` solves a goal of the form ¬B, assuming that there is an assumption ¬A; it puts u : B as assumption, and ask to prove A.

```
Ltac ex_middle u := match goal with
  |- ?p => case (p_or_not_p p) ; [ done | move => u ]
end.
Ltac dneg u := match goal with
  H : ~ _ |- ~ _ => move => u; apply:H
end.
```

The tactic `set_extens v` solves a goal of the form $a = b$ by application of the axiom of extent for sets. It generates two subgoals: $v \in a \implies v \in b$ and $v \in b \implies v \in a$.

```
Ltac set_extens v:= apply: extensionality=> v.
```

This tactic tries to find an equality that solves the goal via `fprops`.

```
Ltac ue :=
  match goal with
    | H:?a = ?b |- _ => solve [ rewrite  H ; fprops | rewrite - H ; fprops]
end.
```

The tactic `empty_tac1 x` assumes that there an assumption $a = \emptyset$, or $\emptyset = a$, and considers the goal $x \in a$. The tactic `empty_tac2 x` assumes that there an assumption that says a and b are disjoint; and considers $x \in a$ and $x \in b$. The tactic `eq_dichot v` assumes that the goal is $a = b \vee$ C. It asks to prove C assuming that v is $a \neq b$. The tactic `mdi_tac` assumes moreover that C says that two sets are disjoint; it replaces C by the property that no element is in both sets.

```
Ltac empty_tac v := apply /set0_P => v.
Ltac empty_tac0 :=
  match goal with
  |  H:inc _ emptyset |- _ => case (in_set_0 H)
  end.
Ltac empty_tac1 u :=
  case (in_set_0 (x:= u));
  match goal with
      H: ?x = emptyset |- _ => rewrite - H
    | H: emptyset = ?x |- _ =>  rewrite H end ;
  fprops.
Ltac empty_tac2 u :=
  match goal with H: disjoint ?x ?y |- _ =>
     case: (in_set_0 (x:= u)); rewrite - H; apply: setI2_i =>//
  end.
Ltac eq_dichot v:= match goal with |- ?a = ?b \/ _
  => case (equal_or_not a b); first (by left); move=> v; right end.
Ltac mdi_tac v:=  eq_dichot v; apply: disjoint_pr.
```

The following tactics can be used in case where the goal contains `Yo P a b`. The first one generates two subgogals, where P is respectively true or false. The second tries to guess a proof of P or not P.

```
Ltac Ytac eq:=
```

```
  match goal with
  | |- context  [Yo ?p _ _ ] =>
      case: (p_or_not_p p) => eq;
   [rewrite (Y_true eq) | rewrite (Y_false eq) ]
end.

Ltac Ytac0 := match goal with
  | h: ?p |- context  [Yo ?p  _ _ ] => rewrite (Y_true h)
  | h: (~ ?p) |- context  [Yo ?p  _ _ ] => rewrite (Y_false h)
  | h: ?j <> ?i |- context  [Yo (?i = ?j) _ _ ]
     => rewrite (Y_false (sym_not_equal h))
  | |- context  [Yo (?i = ?i) _ _ ] => rewrite (Y_true (refl_equal i))
  | |- context  [Yo (C0 = C1) _ _ ] => rewrite (Y_false TP_ne)
  | |- context  [Yo (C1 = C0) _ _ ] => rewrite (Y_false TP_ne1)
  | |- context  [Yo ?p  ?x ?x ] => rewrite (Y_same p x)
end.
```

This tactic solves a goal of the form A ∨ B ∨ C or A ∨ B ∨ C ∨ D, when one of A, B, C or D holds.

```
Ltac in_TP4:= solve [by constructor 1 | by constructor 2 |
   by constructor 3 |  by constructor 4].
```

This solves goals of the form $\exists x, \mathrm{P}$, or $\exists x, \mathrm{P} \wedge \mathrm{Q}$. One of P or Q could be $x \in \mathrm{Y}$, or $(x, a) \in \mathrm{Y}$.

```
Ltac ex_tac:=
  match goal with
   | H:inc (J ?x ?y) ?z |- exists x, inc (J x ?y) ?z
     => exists x ; assumption
   | H:inc (J ?x ?y) ?z |- exists y, inc (J ?x y) ?z
     => exists y ; assumption
   | H:inc (J ?x ?y) ?z |- ex2 _  (fun t => inc (J t ?y) ?z)
     => exists x ; trivial
   | H:inc (J ?x ?y) ?z |- ex2 _ (fun t => inc (J ?x t) ?z)
     => exists y ; trivial
   | H:inc (J ?x ?y) ?z |- ex2 (fun t => inc (J t ?y) ?z) _
     => exists x ; trivial
   | H:inc (J ?x ?y) ?z |- ex2 (fun t => inc (J ?x t) ?z) _
     => exists y ; trivial
   | H:inc ?x ?y |- ex2 (fun t=> inc t ?y) _
     => exists x ; fprops
   | H:inc ?x ?y |- ex2 _ (fun t => inc t ?y)
     => exists x ; fprops
   | H:inc ?x ?y |- exists x, [/\ inc x ?y, _ & _ ]
     => exists x; split => //
   |  |- ex2 (fun t => inc t (singleton ?y)) _
     => exists y ; fprops
    | H : inc (J ?x ?y) ?g |-  inc ?x (domain ?g)
       => exact: (domain_i H)
    |  H : inc (J ?x ?y) ?g |-  inc ?y (range ?g)
       => exact: (range_i H)
    | H : inc ?x ?y |-  nonempty ?y
       => exists x;assumption
   | |- exists y, inc (J (P ?x) y) _
     => exists (Q x) ; aw
   | |- exists y, inc (J y (Q ?x)) _
```

```
     => exists (P x) ; aw
end.
```

This solves a goal related to a graph of a function.

```
Ltac Wtac :=
  match goal with
    | |-  inc (J ?x (Vf ?f ?x)) (graph ?f) => apply: Vf_pr3 ; fprops
    |  h:inc (J ?x ?y) (graph ?f) |- Vf ?f ?x = ?y
       => symmetry; apply: Vf_pr ; fprops
    |  h:inc (J ?x ?y) (graph ?f) |- ?y  = Vf ?f ?x => apply: Vf_pr ; fprops
    | |- inc (Vf ?f _) (range (graph ?f)) => apply: inc_Vf_range_g ; fprops
    | h1: function ?f, h2: inc ?x (source ?f) |- inc (Vf ?f ?x) (target ?f)
       => apply: (inc_Vf_target h1 h2)
    | h2:target ?f = ?y |- inc (Vf ?f ?x) ?y
       =>  rewrite - h2; Wtac
    | h2: ?y = target ?f  |- inc (Vf ?f ?x) ?y
      =>  rewrite  h2; Wtac
    | h1: inc ?x ?y,  h2: ?y = source ?f |- inc (Vf ?f ?x) (target ?f)
      =>  rewrite h2 in h1; Wtac
    | h1: inc ?x ?y,  h2: source ?f = ?y |- inc (Vf ?f ?x) (target ?f)
      => rewrite - h2 in h1; Wtac
    | |- inc (Vf ?f _) (target ?f)
       => apply: (inc_Vf_target); fprops
    | Ha:function ?X1, Hb: inc (J _ ?X2) (graph ?X1)
      |-  inc ?X2 (target ?X1)
        => apply: (inc_pr2graph_target Ha Hb)
    | Ha:function ?X1, Hb: inc (J ?X2 _) (graph ?X1)
      |-  inc ?X2 (source ?X1)
        => apply: (inc_pr1graph_source Ha Hb)
    | Ha:function ?X1, Hb: inc ?X2 (graph ?X1)
      |-  inc (P ?X2) (source ?X1)
        => apply: (inc_pr1graph_source1 Ha Hb)
    | Ha:function ?X1, Hb: inc ?X2 (graph ?X1)
      |-  inc (Q ?X2) (target ?X1)
        => apply: (inc_pr2graph_target1 Ha Hb)
  end.
```

The next tactic solves a goal of the form: f is a function.

```
Ltac fct_tac :=
  match goal with
    | H:bijection ?X1 |- function ?X1 => exact (bij_function H)
    | H:injection ?X1 |- function ?X1 => exact (inj_function H)
    | H:surjection ?X1 |- function ?X1 => exact (surj_function H)
    | H:function ?X1 |- correspondence ?X1 =>
      by case H
    | H:function ?g |- sub (range (graph ?g)) (target ?g)
      => apply: (f_range_graph H)
    | H:composable ?X1 _ |- function ?X1 => destruct H as [H _ ]; exact H
    | H:composable _ ?X1 |- function ?X1 => destruct H as [_ [H _ ]]; exact H
    | H:composable ?f ?g |- function (compose ?f ?g ) =>
     apply: (fcomp_f H)
    | H:function ?f |- function (compose ?f ?g ) =>
     apply: fcomp_f; apply: conj=>//; apply: conj
    | H:function ?g |- function (compose ?f ?g ) =>
```

```
     apply: fcomp_f; apply: conj; last apply: conj=>//
    | Ha:function ?f, Hb:function ?g |- ?f = ?g =>
      apply: function_exten
  end.
```

This tactic solves goals of the form $x \in \bigcup_{i\in \mathrm{I}} \mathrm{X}_i$, by guessing the value of i.

```
Ltac union_tac:=
  match goal with
    | H:inc ?x (?f ?y) |- inc ?x (uniont ?f)
      => apply: (setUt_i H)
    | Ha : inc ?i (domain ?g), Hb :  inc ?x (Vg ?i ?g) |- inc ?x (unionb ?g)
      => apply: (setUb_i Ha Hb)
    | Ha : inc ?x ?y, Hb :  inc ?y ?a |- inc ?x (union ?a)
      => apply: (setU_i Ha Hb)
    | Ha : inc ?y ?i, Hb :  inc ?x (?f ?y) |- inc ?x (unionf ?i ?f)
      => apply: (setUf_i _ Ha Hb)
    | Ha : inc ?y ?i |- inc ?x (unionf ?i ?f)
      => apply: (setUf_i Ha); fprops
    | Hb : inc ?x (?f ?y) |- inc ?x (unionf ?i ?f)
      => apply: (setUf_i _ Hb); fprops
    | Ha : inc ?i (domain ?g) |- inc ?x (unionb ?g)
      => apply: (setUb_i Ha); fprops
    | Hb : inc ?x (Vg ?i ?g) |- inc ?x (unionb ?g)
      => apply: (setUb_i _ Hb); fprops
    | Hb : inc ?z ?X |- inc ?x (union ?X)
      => apply: (setU_i _ Hb); fprops
    | Ha : inc ?x ?z  |- inc ?x (union ?X)
      => apply: (setU_i Ha); fprops
  end.
```

This helps solving goals that depends on the substrate of a relation.

```
Ltac substr_tac :=
  match  goal with
    | H:inc ?x ?r |- inc (P ?x) (substrate ?r)
      => apply: (inc_pr1_sr H)
    | H:inc ?x ?r |- inc (Q ?x) (substrate ?r)
      => apply: (inc_pr2_sr H)
    | H:related ?r ?x _ |- inc ?x (substrate ?r)
      => apply: (inc_arg1_sr H)
    | H:related ?r _ ?y |- inc ?y (substrate ?r)
      => apply: (inc_arg2_sr H)
    | H:inc(J ?x _ ) ?r|- inc ?x (substrate ?r)
      => apply: (inc_arg1_sr H)
    | H: inc (J _ ?y) ?r |- inc ?y (substrate ?r)
      => apply: (inc_arg2_sr H)
  end.
```

This tatics exploits the properties of an equivalence relation.

```
Ltac equiv_tac:=
  match goal with
    | H: equivalence ?r, H1: inc ?u (substrate ?r) |- related ?r ?u ?u
      => apply: (reflexivity_e H H1)
    | H: equivalence ?r |- inc (J ?u ?u) ?r
```

```
      => apply: reflexivity_e
    | H:equivalence ?r, H1:related ?r ?u ?v |- related ?r ?v ?u
      => apply: (symmetricity_e H H1)
    | H:equivalence ?r, H1: inc (J ?u ?v) ?r |- inc (J ?v ?u) ?r
      => apply: (symmetricity_e H H1)
    | H:equivalence ?r, H1:related ?r ?u ?v, H2: related ?r ?v ?w
      |- related ?r ?u ?w
      => apply: (transitivity_e H H1 H2)
    | H:equivalence ?r, H1:related ?r ?v ?u, H2: related ?r ?v ?w
      |- related ?r ?u ?w
      => apply: (transitivity_e H (symmetricity_e H H1) H2)
    | H: equivalence ?r, H1: inc (J ?u ?v) ?r, H2: inc (J ?v ?w) ?r |-
      inc (J ?u ?w) ?r
      => apply: (transitivity_e H H1 H2)
end.
```

Other tactics.

```
Ltac try_lvariant u:=
  move:u;move/ two_pointsP; case => ->; bw.
Ltac eqtrans u:= apply equipotentT with u; fprops.
Ltac eqsym:= apply: equipotentS.
```

8.5 List of Theorems

We give here the list of all theorems, propositions, lemmas, corollaries, together with the Coq names, a page reference, and the statement (we use French quotes for exact citations).

Section one

Proposition 1 (`sub_refl`) « $x \subset x$ », [23].

Proposition 2 (`sub_trans`) « $(x \subset y$ and $y \subset z) \implies (x \subset z)$ », [23].

Theorem 1 « The relation $(\forall x)(x \notin \mathrm{X})$ is functional in X. » This theorem asserts existence and uniqueness of the empty set, [23].

Section 2

Theorem 1 asserts existence of the product $\mathrm{X} \times \mathrm{Y}$ of two sets, [35].

Proposition 1 (`setX_Sl` and variants) « If A', B' are non-empty sets, the relation $\mathrm{A}' \times \mathrm{B}' \subset \mathrm{A} \times \mathrm{B}$ is equivalent to "$\mathrm{A}' \subset \mathrm{A}$ and $\mathrm{B}' \subset \mathrm{B}$" », [35].

Proposition 2 (`setX_0` and variants) « Let A and B be two sets. The relation $\mathrm{A} \times \mathrm{B} = \emptyset$ is equivalent to "$\mathrm{A} = \emptyset$ or $\mathrm{B} = \emptyset$" », [35]

Section 3

Proposition 1 (`range_domain_exists`) asserts existence and uniqueness of the range and domain of a graph, [43].

Proposition 2 (`dir_im_S`) « Let G be a graph and let X, Y be two sets; then the relation $\mathrm{X} \subset \mathrm{Y}$ implies $\mathrm{G}\langle \mathrm{X} \rangle \subset \mathrm{G}\langle \mathrm{Y} \rangle$ », [45].

Corollary (`dir_im_domain`).

Proposition 3 (`compg_inverse`) « Let G, G′ be two graphs. The inverse of G′∘G is then $\overset{-1}{G}\circ\overset{-1}{G'}$ », [47].

Proposition 4 (`compgA`) is associativity of composition of graphs, [47].

Proposition 5 (`compg_image`) says $(G'\circ G)\langle A\rangle = G'\langle G\langle A\rangle\rangle$, [47].

Proposition 6 (`compf_f`) says « If f is a mapping of A into B and g is a mapping of B into C, then $g\circ f$ is a mapping of A into C », [59].

Proposition 7 (`bijective_inv_function` and `inv_function_bijective`) says « Let f be a mapping of A into B. Then $\overset{-1}{f}$ is a function if and only if f is bijective », [65].

Proposition 8 (`inj_if_exists_left_inv`, and variants) says under which conditions a function has a left or right inverse, [68].

Corollary (`bijective_from_compose`).

Theorem 1 (`inj_compose`) and variants) studies the relationship between injectivity, surjectivity and composition, [69].

Proposition 9 (`exists_left_composable` and variants) explains when a function can be factored through another one, [71].

Section 4

Proposition 1 (`setUt_rewrite`, `setIt_rewrite` and variants) says that if $f : K \to I$ is a function, $(X_\iota)_{\iota\in I}$ a family of sets, then the union and the intersection of the family is the union and the intersection of $X_{f(\kappa)}$ over K, [80].

Proposition 2 (`setUf_A` and `setIf_A`) states associativity of union and intersection, [82].

Proposition 3 (`dirim_setUt` and `dirim_setIt`) says that if Γ is a correspondence, $\Gamma\langle\bigcup X_\iota\rangle = \bigcup\Gamma\langle X_\iota\rangle$ and $\Gamma\langle\bigcap X_\iota\rangle \subset \bigcap\Gamma\langle X_\iota\rangle$, [82].

Proposition 4 (`iim_fun_setIt`) says that equality holds for the inverse image of intersection by a function [82].

Corollary (`inj_image_setIt`).

Proposition 5 (`setCUf2` and `setCIf2`) studies the complementary of unions and intersections, [82].

Proposition 6 (`iim_fun_C`) studies the inverse image of the complementary, [83].

Corollary (`inj_image_C`).

Proposition 7 (`agrees_on_covering` and `extension_covering`) says that if X_ι is a covering of E, then two functions that agree on each X_ι agree on E, and a function defined on each X_ι can be extended to E if the obvious compatibility conditions hold, [85].

Proposition 8 (`extension_partition`) says that if $(X_\iota)_\iota$ is a partition of X and $f_\iota \in \mathscr{F}(X_\iota, T)$, then there exists a unique $f \in \mathscr{F}(X, T)$ that extends every f_ι, [88].

Proposition 9 (`disjoint_union_lemma`) asserts existence of the disjoint union, [88].

Proposition 10 (`disjoint_union_pr`) relates sum and union, [89].

Section 5

Proposition 1 (`etp_fs` and `etp_fi`) says: if f is surjective (resp. injective), then its extension to the set of sets is surjective (resp. injective), [91].

Proposition 2 (`c3f_fi` and `c3f_fs`) states under which conditions $f \mapsto v \circ f \circ u$ is injective or surjective, [93].

Corollary (`c3f_fb`).

Proposition 3 (`fpfa_fb` and `spfa_fb`) says that $\mathscr{F}(B\times C;A)$, $\mathscr{F}(B;\mathscr{F}(C;A)$ and $\mathscr{F}(C;\mathscr{F}(B;A))$ are canonically isomorphic, [93].

Proposition 4 (`pc_fb`) says: Given a family X_ι and a bijection f, the product $\prod X_\iota$ is isomorphic to the product $\prod X_{f(\iota)}$, [98].

Propositions 6 and 5 (`extension_psetX` and `prj_fs`) if X_ι is nonempty for $\iota \notin J$, then pr_J is surjective from the product $\prod_{\iota\in I} X_\iota$ into the partial product $\prod_{\iota\in J} X_\iota$ [99].

Corollary 1 (`pri_fs`).

Corollary 2 (`nonempty_product` and variants).

Corollary 3 (`setXb_monotone1`, `setXb_monotone2`).

Proposition 7 (`pam_fb`) states associativity of the product, [100].

Proposition 8 (`distrib_union_inter` and `distrib_inter_union`) states distributivity of union over intersection and intersection over union, [101].

Corollary (`distrib_union2_inter` and `distrib_inter2_union`).

Proposition 9 (`distrib_prod_union` and `distrib_prod_intersection`) states distributivity of product over union and intersection, [101].

Corollary 1 (`partition_product`).

Corollary 2 (`distrib_prod2_union` and `distrib_prod2_intersection`).

Proposition 10 (`distrib_inter_prod` and `distrib_prod_intersection`) says that the intersection of a product is the product of the intersection, [102].

Corollary (`distrib_prod_inter2_prod` and `distrib_inter_prod_inter`).

Proposition 11 says that composition of extensions is extension of compositions.

Corollary (`injective_ext_map_prod` and `injective_ext_map_prod`).

Section 6

Proposition 1 (`equivalence_cor_pr`) says: « A correspondence Γ between X and X is an equivalence on X if and only if it satisfies the following conditions: (a) X is the domain of Γ; (b) $\Gamma = \Gamma^{-1}$; (c) $\Gamma \circ \Gamma = \Gamma$ », [111].

Criterion C55 (`related_e_rw`) characterizes the canonical projection, [113].

Criterion C56 (`rel_on_quo_pr`) « Let $R\{x,x'\}$ be an equivalence relation on a set E and let $P\{x\}$ be a relation that does not contain the letter x' and is compatible (with respect to x) with the equivalence relation $R\{x,x'\}$. Then, if t does not appear in $P\{x\}$, the relation "$t \in E/R$ and $(\exists x)(x \in t$ and $P\{x\})$" is equivalent to the relation "$t \in E/R$ and $(\forall x)(x \in t$ and $P\{x\})$" ». [116].

Criterion C57 (`exists_unique_fun_on_quotient`) « Let R be an equivalence relation on a set E, and let g be the canonical mapping of E onto E/R. Then a mapping f of E into F is compatible with R is and only if f can be put in the form $h \circ g$, where h is a mapping of E/R into F. The mapping h is uniquely defined by f; if f is any section of g, we have $h = f \circ s$. »[119]

8.6 Notations and Definitions

In many cases we indicate the page on which an object is defined.

Symbols

$x \wedge y$ is often replaced by "and". The Coq equivalent is `/\`.

$x \vee y$ is often replaced by "or". The Coq equivalent is `\/`.

$\neg x$ is often replaced by "not". The Coq equivalent is `~`.

$\square$ is a dummy variable for Bourbaki, [8].

R$\{x\}$ is a Bourbaki notation, meaning that R is a relation that may depend on x. If R is a relation that depends on y, it is also $(x|y)$R.

τ_x(R) is a Bourbaki notation, it is the generic element satisfying R$\{x\}$, [13].

$x \Longrightarrow y$ is represented in Coq by `x -> y`.

$x \mapsto y$ is represented in Coq by `fun x => y`.

$x \to y$ is a Coq notation meaning the type of functions from type x to type y.

$x = y$ is equality. Was used as synonym to $\Longleftrightarrow$.

$(a|b)c$ is a Bourbaki notation, meaning the relation obtained by replacing b by a in c, [9].

$x : y$ is a Coq notation meaning that x is of type y.

$f(x)$ is the value of the function f at point x, parentheses are sometimes omitted.

$f\langle x\rangle$ is the value of f on the set x, see fun_image, image_by_graph, image_by_fun.

$\overset{-1}{f}\langle x\rangle$, see inverse_image.

$(\forall x)$P and `forall x, p` are similar constructions, [14].

$(\exists x)$P and `exists x, p` are similar constructions, [14].

$(\exists! x)$P means `exists_unique`.

$x \in y$ (is element of): see inc.

$x \subset y$ (is subset of): see sub.

$\emptyset$ (empty set): see emptyset.

$\{x, \mathrm{R}\}$ (set of x such that R): see Zo.

$\{x\}$, $\{x, y\}$: see singleton or doubleton.

$a - b$, $a \setminus b$, $\complement a$: see complement.

`a -s1 b`: is the set formed of a by removing element b.

`a +s1 b`: is the set formed of a by adding element b.

(x, y) (ordered pair): see J.

$\bigcup \mathrm{X}$, $\bigcup_{\iota \in \mathrm{I}} \mathrm{X}_\iota$, see union.

$a \cup b$, $a \cap b$, `\cup`, `\cap`, see union2, intersection2.

`a =1g b`, `a =1f b`, see same_Vg and same_Vf.

`x \cf y`, `x \cg y`, `x \co y`, composition of x and y, see composef, composeg, compose.

`x \cfP y`, `x \coP y`, says that x and y can be composed. See composablef, composable.

$f \circ g$, f \co g, f \cf g, f \cg g, see compose, composef, composeg.

$A \times B$, `a \times b`, $u \times v$, $R \times R'$, see product, ext_to_prod, prod_of_relation.

x \Eq y, see equipotent.

Δ_A, see diagonal.

G^{-1} see inverse_graph, inverse_fun or inverseC.

$x \mapsto y$ or $x \to y$ is the function that maps x to y, for instance $x \mapsto \sin(x)$ (source and target are implicit).

$\boldsymbol{x \to T}$ ($\boldsymbol{x \in A}$, $\boldsymbol{T \in C}$), is the function with source A, target C that maps x to T, [57].

$(f_x)_{x \in A}$ is a shorthand for $x \to f(x)$ ($x \in A$); see above, the piece $\boldsymbol{T \in C}$ is implicit.

$\hat{f}$, see extension_to_parts.

F^E, see gfunctions.

$\mathscr{F}(E;F)$ see functions.

$\Phi(E,F)$ see sub_functions.

f_x, f_y sometimes denotes the mappings $y \mapsto f((x,y))$ or $x \mapsto f((x,y))$, implemented as `first_partial_fun`, `second_partial_fun`, [93].

$\tilde{f}$, sometimes denotes the mappings $x \mapsto f_x$ or $y \mapsto f_y$. Implemented as `first_partial_function`, `second_partial_function`, [93].

$f \mapsto \tilde{f}$, implemented as `first_partial_map`, `second_partial_map`, is a bijection from $\mathscr{F}(B \times C;A)$ into $\mathscr{F}(B;\mathscr{F}(C;A))$ or $\mathscr{F}(C;\mathscr{F}(B;A))$, [93].

$\prod_{\iota \in I} X_\iota$ see productt.

$(x_\iota)_{\iota \in I}$ denotes an element of a product indexed by I.

$x \overset{r}{\sim} y$ is sometimes used instead of $r(x,y)$ or $(x,y) \in r$, especially when r is the graph of an equivalence relation.

$g_E(\sim)$, the graph of $\sim$ on E, see graph_on.

$\sim_f$ may denote `eq_rel_associated f`.

$\bar{x}$, may denote the equivalence class of x, see class.

$\hat{x}$ may denote a representative of the equivalence class x.

E/ $\sim$, E/R, see quotient.

R/S see quotient_of_relations.

X_f sometimes means $f^{-1}\langle f\langle X\rangle\rangle$, see inverse_direct_value.

R_A see induced_relation.

¶ is not defined. We use it as a paragraph separator.

Letters

$\mathscr{B}$: see Bo.

$\mathscr{C}_C(a,b)$, $\mathscr{C}_T(p,q)$, $\mathscr{C}(p)$: see by_cases, chooseT and choose.

$C_{xy}a$ stands for `constant_function x y a`, it is the constant function from x to y with value a, [53].

$C_R x$ may denote the equivalence class of x for R, see class.

$\mathrm{Coll}_{\boldsymbol{x}}\boldsymbol{R}$ says that R is collectivizing in x, [17].

$\mathscr{E}$, see Set.

$\mathscr{E}_x(R)$ appears in the English version where $\{x,R\}$ is used in the French version; see Zo.

I_A, see identity.

I_{xy} see inclusionC, canonical_injection.

$\mathscr{L}_X f$, $\mathscr{L} f$, $\mathscr{L}_{A;B} f$ (creating functions): see Lf, Lg, acreate.

$\mathscr{M}f$, $\mathscr{M}_{\mathrm{A;B}}f$ (inverse of $\mathscr{L}$), see bcreate1 and bcreate.

$\mathfrak{P}(x)$, see powerset.

$\mathrm{pr}_1 z$, $\mathrm{pr}_2 z$, $\mathrm{pr}_\mathrm{i} f$, $\mathrm{pr}_\mathrm{J} f$ (projections), see P, Q, pr_i, pr_j.

$\mathscr{R}x$ see Ro.

$\mathrm{R}_{ab}f$, see restriction, [57].

$\mathscr{V}(x,f)$, $\mathscr{V}_f x$ (value of a function), see Vg.

$\mathscr{W}_f x$ (value of a function): see Vf.

$\mathscr{Y}(\mathrm{P},x,y)$ see Yo.

$\mathscr{Z}(x,\mathrm{P})$ see Zo.

Words

`acreate f`, $\mathscr{L}f$, is the correspondence associated to the COQ function f, [51].

`agrees_on x f f'`, `agreeC x f f'` is the property that for all $a \in x$, $f(a)$ and $f'(a)$ are defined and equal, [54].

`alls X p` means that $p(x)$ holds if $x \in \mathrm{X}$, [21].

`allf G p` means that $p(x)$ holds if x is in the range of the functional graph G, [39].

`antisymmetricp r` says that the graph r is antisymmetric, [108].

`bcreate f A B`, $\mathscr{M}_{\mathrm{A;B}}f$, is a kind of inverse of $\mathscr{L}$, [51].

`bcreate1 f`, $\mathscr{M}f$, is a kind of inverse of $\mathscr{L}$, [51].

`bijection f`, `bijectiveC f`, means that f is a bijection, [60].

Bo, $\mathscr{B}$, is an inverse of $\mathscr{R}$, [24].

`by_cases a b`, $\mathscr{C}_\mathrm{C}(a,b)$, defines an object by applying a if P is true, and b if P is false (not used anymore).

`canonical_injection x y`, I_{xy}, is the inclusion map on $x \subset y$, [64].

`canon_proj r`, is the mapping $x \mapsto \bar{x}$ from E onto E/R, the quotient set of r, [113].

`class r x` is the class of x for the equivalence relation r, [111].

`classp r x` says that x is an equivalence class for r, [112]

`choose p`, $\mathscr{C}(p)$, is some x such that $p(x)$ is true, the empty set if no x satisfies p, [24].

`chooseT p q`, $\mathscr{C}_\mathrm{T}(p,q)$, is our basic axiom of choice, [22].

`coarse x` is $x \times x$, [110].

`coarser_cs I J`, `coarser_cg f g`, two definitions that say for all $j \in \mathrm{J}$ there is $i \in \mathrm{I}$ such that $j \subset i$ or for all j there is i such that $g_j \subset f_i$, [84].

`compatible_with_equiv_p p r` means that $p(x)$ and $x \overset{r}{\sim} y$ implies $p(y)$, [116].

`compatible_with_equiv f r` means that $x \overset{r}{\sim} y$ is equivalent to $f(x) = f(y)$, [118].

`compatible_with_equivs f r r'` means that $x \overset{r}{\sim} y$ is equivalent to $f(x) \overset{r'}{\sim} f(y)$, [119].

`complement a b`, $a - b$, $a \setminus b$, $\complement b$, is the set of element of a not in b, [27].

`composable f g`, `f \cfP g`, is the condition on functions f and g for $f \circ g$ to be a function, [59].

`composableC f g`, is composable for correspondences, [47],

`composablef f g`, `f \cfP g`, is composable for functional graphs, [42].

`compose f g`, `f \co g`, $f \circ g$, is the composition of two functions or correspondences, [47],

`composef f g`, `f \cf g`, $f \circ g$, composition of two graphs, [42].

`composeg f g`, `f \cg g`, $f \circ g$, variant of composition of two graphs, [47].

`constant_graph s x` is the graph of the constant function with domain s and value x, [98].

`correspondence f` says that f is a triple (G, A, B), with $G \subset A \times B$, [44].

`correspondences A B` means the set of correspondences from A to B, it is $\mathfrak{P}(A \times B) \times \{A\} \times \{B\}$,[44].

`covering f x`, `covering_f I f x`, `covering_s f x`, three variants of a family of sets (defined by f and I) whose union contains x, [83].

`cstgp f E`, `cstfp f E`, says that f is a constant graph (resp. function) on the set E.

`cut r x` is $r\langle\{x\}\rangle$, replaced by `im_singleton` [45].

`diagonal A`, Δ_A, is the set of all (x, x) such that $x \in A$, [43].

`diagonal_application A` is the diagonal mapping $x \mapsto (x, x)$ of A into Δ_A, [64].

`diagonal_graphp I E` is the set of graphs of constant functions from I to E, [98].

`disjoint x y` means $x \cap y = \emptyset$, [86].

`disjointVeq x y` means disjoint or equal, [86].

`disjoint_union f`, `disjoint_union_fam f` are two variants of the disjoint union of the family of sets f, [89].

`domain f` is the set of x for which there is an y with $(x, y) \in f$, it is $\mathrm{pr}_1\langle f\rangle$, [36].

`doubleton x y`, $\{x, y\}$, is a set with elements x and y, [25].

`empty_function`, `empty_functionC` is the identity on $\emptyset$, [52].

`emptyset`, $\emptyset$, is a set without elements, [23].

`eq_rel_associated f` is the graph of the equivalence relation $f(x) = f(y)$, [111].

`equipotent x y` means that there is a bijection from x to y.

`equivalence r` says that the graph r is an equivalence, [108].

`equivalence_associated f` is the equivalence relation $f(x) = f(y)$, [111].

`equivalence_r r`, `equivalence_re r x`, says that the relation r is an equivalence relation (in x), [107].

`exists_unique p`,$(\exists!x)p$, (this notation is not in Bourbaki) means that there exists a unique x such that $p(x)$, [22].

`extends g f`, `extendsC g f` says $g(x) = f(x)$ whenever $f(x)$ is defined, [56].

`ext_map_prod I X Y g` is the function $(x_\iota)_{\iota\in I} \mapsto (g_\iota(x_\iota))_{\iota\in I}$ from $\prod_I X_\iota$ into $\prod_I Y_\iota$, [104].

`ext_to_prod u v` is the function $(x, y) \mapsto (u(x), v(y))$, sometimes denoted $u \times v$, [73]

`extension_to_parts f`, denotes the function $x \mapsto f\langle x\rangle$, from $\mathfrak{P}(A)$ to $\mathfrak{P}(B)$, [91]

`finer_equivalence s r`, comparison of equivalences, $x \overset{s}{\sim} y$ implies $x \overset{r}{\sim} y$, [124].

`first_proj g` is the function $x \mapsto \mathrm{pr}_1 x$ $(x \in g)$.

`first_proj_equiv x y`, `first_proj_equivalence x y`, is the equivalence associated to `first_proj` on the set $x \times y$, [114].

`fgraph f` says that f is a functional graph, [36].

`fterm`, `fterm2` are the types `Set → Set` and `Set → Set → Set`, [21].

`function f` says that f is a function in the sense of Bourbaki, [48].

`functional_graph f` says that f is a functional graph, [48].

`functions E F`, denoted $\mathscr{F}(E;F)$, is the set of functions $E \to F$, [92].

`fun_image x f`, $f\langle x\rangle$, is the value of f on the set x, [26].

`fun_on_quotient r f`, `function_on_quotient r f b`, `function_on_quotients`, `fun_on_quotients r r' f`, the function obtained from f on passing to the quotient of r (or r and r'), [119], [120].

`fun_set_to_prod E X` is the canonical bijection between $(\prod X_\iota)^E$ and $\prod X_\iota^E$, [105].

`function_prop f s t`, `function_prop_sub f s t`. This is the property that f is a function from s into t, or into a subset of t, [52].

`gfunctions E F`, denoted F^E, is the set of graphs of functions from E to F, [92].

`graph f` is a part of a correspondence or function, [44].

`graph_on r X` is the graph of the relation r restricted to X, [109].

`identity_g A`, I_A, is is the graph of the identity function on the set A, [41].

`identity A`, I_A, is the identity function on the set A, [48].

`IM` stands for the image of a function. Its axioms implement the Axiom Scheme of Replacement, [23].

`image_by_fun f A`, $f\langle A\rangle$, is $\{t, \exists x \in A, t = f(x)\}$, [45].

`image_by_graph f A`, $f\langle A\rangle$ is $\{t, \exists x \in A, (x,t) \in f\}$, [45].

`image_of_fun f`, is the image of f, [45].

`inc x y` or $x \in y$ means that x is an element of y, [19].

`inclusionC x y`, I_{xy}, is the inclusion map on $x \subset y$ as a Coq function, [54].

`induced_relation R A`, R_A, is the equivalence induced by R on A, [123].

`injection f`, `injectiveC f`, means that f is an injection, [60].

`in_same_coset f` is the relation "there exists i such that $x \in f(i)$ and $y \in f(i)$" between x and y, [114].

`intersection X`, $\bigcap X$, is the intersection of a set of sets, [31].

`intersectiont I f`, `intersectionf x f`, `intersectiont g`, $\bigcap_{\iota\in I} X_\iota$ is the set of elements a such that forall $\iota \in I$ we have $a \in X_\iota$, [79].

`intersection2 X Y`, $X \cap Y$, is the intersection of two sets [31].

`intersection_covering`, intersection of coverings, [84].

`inverse_direct_value f X`, X_f, is $f^{-1}\langle f\langle X\rangle\rangle$, [117].

`inverse_graph G`, $\overset{-1}{G}$, inverse graph of the graph G, [45].

`inverse_fun f` or `inverseC a b f H`, $\overset{-1}{f}$, inverse of the function f, [46], [61].

`inverse_image x f`, $\overset{-1}{f}\langle x\rangle$, is the inverse value of f on the set x.

`inv_image_relation f r`, is the inverse image of the relation r under the function f, [123].

`inv_image_by_graph f x`, `inv_image_by_fun r x`, $\overset{-1}{f}\langle x\rangle$, direct image of a set by the inverse function, [46]

`inv_graph_canon G` is the bijection $(x,y) \mapsto (y,x)$ from G to G^{-1}, [65].

`is_left_inverse r f` means that r is a retraction or left-inverse of f, and $r \circ f$ is the identity, [67].

`is_right_inverse s f` means that s is a section or right-inverse of f, and $f \circ s$ is the identity, [67].

`J x y`, or (x,y), is an ordered pair, formed of two items x and y, [34].

`largest_partition x` is the set of all singletons of x.

`left_inverseC`, left inverse of a Coq function, [68].

`Lf f A B`, $\mathscr{L}_{A;B} f$, is function from A to B whose graph is $\mathscr{L}_A f$, [57].

`Lg X f`, $\mathscr{L}_X f$ is the graph formed of all $(x, f(x))$ with $x \in X$, [40].

`lf_axiom f A B` says that for all $x \in$ A we have $f(x) \in$ B, case where $\mathscr{L}_{A;B} f$ is a function, [57].

LHS is the left hand side of an equality.

`Lvariant a b x y`, `variant a x y`, `Lvariantc x y`, these are functions whose range is the doubleton $\{x, y\}$, [87].

`mutually_disjoint f` says that for all distinct i and j, $f(i)$ and $f(j)$ are disjoint, [86]

$x \neq y$, `x <> y` is inequality, [22].

`P z`, $\mathrm{pr}_1 z$ denotes x if z is the pair (x, y), [34].

`pairp x`, says that x is an ordered pair.

`partial_fun1 f y`, `partial_fun1 f x`, partial functions, [72].

`partition y x`, `partition_s y x`,`partition_fam f x`, thee variants that say that y or f is a partition of x, [86].

`partition_relation f x` is the equivalence relation associated to the partition graph(f) of x, [114].

`partition_with_complement X A`, is the partition of X formed of A and its complementary set, [87].

`permutations E`, is the set of bijections E → E, [92].

`powerset x`, $\mathfrak{P}(x)$, is the set of subsets of x, [31].

$\mathrm{pr}_1 z$, $\mathrm{pr}_2 z$ stand for `pr1 z` and `pr2 z`. These are also denoted by P and Q. If z is the pair (x, y), these functions return x and y respectively, [34].

`pr_i f i`, $\mathrm{pr}_i f$, denotes a component of an element of a product., [96].

`pr_j f J`, $\mathrm{pr}_J f$, is the function $(x_\iota)_{\iota\in I} \mapsto (x_\iota)_{\iota\in J}$, [99].

`prod_assoc_map` is the function whose bijectivity is the "theorem of associativity of products", [100].

`prod_of_function u v`, is the function $x \mapsto (u(x), v(x))$, [103].

`prod_of_products_canon F F'`, is the bijection between $\prod \mathrm{F}_\iota \times \prod \mathrm{F}'_\iota$ and $\prod(\mathrm{F}_\iota \times \mathrm{F}'_\iota)$, [103].

`prod_of_relation R R'`, $\mathrm{R} \times \mathrm{R}'$, is the product of two equivalences, [126].

`product A B`, A × B, is the set of all pairs (a, b) with $a \in$ A and $b \in$ B, [35]. See also `ext_to_prod u v`.

`productb g` or `productf I f`, $\prod_{\iota\in I} \mathrm{X}_\iota$ is the product of a family of sets, [95].

`product1 x a` is the product of the family defined on the singleton $\{a\}$ with value x, [97].

`product1_canon x a` is the canonical application from x into `product1 x a`, [97].

`product2 x y` is the product of the family defined on the doubleton $\{a, b\}$ with values x and y, [97].

`product2_canon x y` is the canonical application from $x \times y$ into `product2 x y`, [97].

`product_compose`, auxiliary function used for change of variables in a product, [98].

`property` is the type `Set→Prop`, [21].

`Q z`, $\mathrm{pr}_2 z$ denotes y if z is the pair (x, y), [34].

`quotient R`, E/R, is the set of equivalence classes of R, [112].

`quotient_of_relations r s`, R/S, is the quotient of two equivalences, [125].

`range f` is the set of y for which there is an x with $(x, y) \in f$, it is $\mathrm{pr}_2\langle f\rangle$, [36].

`reflexive_r r x` says that the relation r is reflexive in x, [107].

`reflexivep r` says that the graph r is reflexive, [108].

`related r x y` is a shot-hand for $(x, y) \in r$, [38].

`relation` is the type `Set → Set → Prop`, [21].

`relation_on_quotient p r` is the relation induced by $p(x)$ on passing to the quotient (with respect to x) with respect to R, [116].

`rep x` is an element y such that $y \in x$, whenever x is not empty, [24].

`representative_system s f x` means that, for all i, $s \cap \mathrm{X}_i$ is a singleton, where X_i is a partition of x associated to the function f, [115].

`representative_system_function g f x`, means that g is an injection whose image is a system of representatives (see definition above), [115].

`restr x G` is the restriction to x of the graph G, [41].

`restricted_eq E` is the relation "$x \in$ E and $y \in$ E and $x = y$", [109].

`restriction_function f x` is like `restr`, but f and the restrictions are functions, [55].

`restriction2_axioms f x y` is the condition: f is a function whose source contains x, whose target contains y, moreover $a \in x$ implies $f(a) \in y$, [57].

`restriction2 f x y`, `restriction2C f x y`, restriction of f as a function $x \to y$, [57].

`restrictionC f H` is the restriction to x of the function $f : a \to b$, where H proves $x \subset a$ implicitly, [54].

`restriction_product f j` is the product of the restrictions of $\prod f$ to J, [99].

`restriction_to_image f` is the restriction of the Coq function f to its range, [75].

retraction: see `is_left_inverse`.

RHS is the right hand side of an equality.

`right_inverseC`, right inverse of a Coq function, [68].

`Ro x` or $\mathscr{R}x$ converts its argument x of type u to a set, which is an element of u, [22].

`same_Vg f g`, `same_Vf f g` means: $f(x) = g(x)$ whenever x, [39], [49].

`saturated r x` means: for every $y \in x$, the class of x for the relation r is a subset of x, [116].

`saturation_of r x` is the saturation of x for r, [117].

`second_proj g` is the function $x \mapsto \mathrm{pr}_2 x$ $(x \in g)$.

section: see `is_right_inverse`.

`section_canon_proj R` is the function from E/R into E induced by `rep`, [118].

`Set` is the type of sets, [19].

`sgraph f` says that f is a set of pairs, [36].

`singleton x`, $\{x\}$, is a set with one element, [25].

`singletonp x` means that x is a singleton.

`singl_val p`, means that $p(x)$ and $p(y)$ imply $x = y$, [21].

`singl_val_fp p f`, means that $p(x)$ and $p(y)$ imply $f(x) = f(y)$, [21].

`small_set x` means that x has at most one element, [53].

`smallest_partition x` is the singleton $\{x\}$.

`source f` contains (resp. is equal to) the domain of the graph of a correspondence f (resp. function f) [44], [48].

`ssub x y`, $x \subsetneq y$, means $x \subset y$ and $x \neq y$, [22].

`sub x y`, $x \subset y$, means that x is a subset of y, [19].

`surjection f`, `surjectiveC f`, means that f is a surjection, [60].

`sub_functions E F`, denoted $\Phi(E;F)$ is the set of triples (G,A,F) associated to functions from $A \subset E$ into F, [92].

`substrate r` is the union of the domain and range [107].

`symmetric_r r` says that the relation r is symmetric, [107].

`symmetricp r` says that the graph r is symmetric, [108].

`target f` contains the range of the graph of a correspondence or function f, [44].

`TPa`, `TPb`, `TP`, are respectively 0, 1, 2, sets with zero, one and two elements, [26].

`transitive_r r` says that the relation r is transitive, [107].

`transitivep r` says that the graph r is transitive, [108].

`triple a b c` is the ordered pair $(a,(b,c))$.

`tripleton a b c` is the set a,b,c.

`union X`, $\bigcup X$, is the union of a set of sets, [27],

`uniont I f`, `unionf x f`, `uniont g`, $\bigcup_{\iota\in I} X_\iota$ is the set of elements a such that $a \in X_\iota$ for some $\iota \in I$, [79].

`union2 a b`, $a \cup b$, is the union of two sets, [28].

`Vf x f`, $\mathscr{W}_f x$, is the value at the point x of the function f, [49].

`Vg x f`, $\mathscr{V}(x,f)$ or $\mathscr{V}_f x$, is the value at the point x of the functional graph f, [39].

`variant`, see `Lvariant`.

`Yo P x y`, $\mathscr{Y}(P,x,y)$, is a function that associates to z the value x is P is true, and y if P is false, [30].

`Zo x R`, $\mathscr{Z}(x,R)$, $\mathscr{E}_x(R)$ or $\{x,R\}$: it is the set of all x that satisfy R, [17], [24].

Index

antisymmetric, 107
associative, 28, 32, 47, 59, 82, 100
axiom, 9

bijective, 60

canonical projection, 113
choice, 22, 24, 30, 68, 96, 99, 102, 112
class, 111
commutative, 25, 28, 32
compatible, 116, 118, 119
complement, 27
composition, 42, 46, 47
constant, 53
correspondence, 44
covering, 83

diagonal, 43
disjoint, 33
domain, 38
doubleton, 25, 83, 97, 103

empty, 23–25, 31, 35, 45, 52, 68, 79–81
equipotent, 62, 65
equivalence, 107
extension, 56, 91, 104
extensionality, 16, 22, 23, 25, 41, 80, 96

finer, 124
function, 7, 48

graph, 38

identity, 41, 48
induced, 116, 119, 123
injective, 59
intersection, 31, 78
inverse, 65
involutive, 27, 46, 65

mapping, 7

partition, 86
powerset, 30
product, 35, 95
proof, 8

quotient, 111

range, 38
reflexive, 107
restriction, 41, 53
retraction, 67

saturated, 116
scheme, 9
section, 67
singleton, 25, 65
substrate, 107
sum, 88
surjective, 59
syllogism, 10
symmetric, 107

theorem, 8, 9
theory, 7
transitive, 107

union, 27, 78

Bibliography

[1] Yves Bertod and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[2] N. Bourbaki. *Elements of Mathematics, Theory of Sets*. Springer, 1968.

[3] N. Bourbaki. *Éléments de mathématiques, Théorie des ensembles*. Diffusion CCLS, 1970.

[4] Douglas Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.

[5] Jean-Louis Krivine. *Théorie axiomatique des ensembles*. Presses Universitaires de France, 1972.

[6] Edward Nelson. Internal set theory: a new approach to nonstandard analysis. *Bulletin of the American Mathematical Society*, 1977.

[7] The Coq Development Team. The Coq reference manual. http://coq.inria.fr.

Contents

1 Introduction **3**

1.1 Objectives . . . 3
1.2 Background . . . 4
1.3 Introduction to Coq . . . 4
1.4 Notations . . . 6
1.5 Description of formal mathematics . . . 7
1.6 The theory of sets . . . 17

2 Sets **21**

2.1 Module Axioms . . . 21
2.2 Module constructions . . . 23
2.3 Module Little . . . 25
2.4 Module Image . . . 26
2.5 Module Complement . . . 27
2.6 Module Union . . . 27
2.7 Module Powerset . . . 30
2.8 Module Intersection . . . 31
2.9 Module Pair . . . 33
2.10 Module Cartesian . . . 35
2.11 Module Function . . . 36

3 Correspondences **43**

3.1 Graphs and correspondences . . . 43
3.2 Inverse of a correspondence . . . 45
3.3 Composition of two correspondences . . . 46
3.4 Functions . . . 48
3.5 Restrictions and extensions of functions . . . 53
3.6 Definition of a function by means of a term . . . 57
3.7 Composition of two functions. Inverse function . . . 59

3.8 Retractions and sections 67
3.9 Functions of two arguments 72

4 Union and intersection of a family of sets 77
4.1 Definition of the union and intersection of a family of sets 78
4.2 Properties of union and intersection 81
4.3 Complements of unions and intersections 82
4.4 Union and intersection of two sets 83
4.5 Coverings 83
4.6 Partitions 86
4.7 Sum of a family of sets 88

5 Product of a family of sets 91
5.1 The axiom of the set of subsets 91
5.2 Set of mappings of one set into another 92
5.3 Definition of the product of a family of sets 95
5.4 Partial products 98
5.5 Associativity of products of sets 100
5.6 Distributivity formulae 101
5.7 Extensions of mappings to products 104

6 Equivalence relations 107
6.1 Definition of an equivalence relation 107
6.2 Equivalence classes; quotient set 111
6.3 Relations compatible with an equivalence relation 116
6.4 Saturated subsets 116
6.5 Mappings compatible with equivalence relations 118
6.6 Inverse image of an equivalence relation; induced equivalence relation 123
6.7 Quotients of equivalence relations 124
6.8 Product of two equivalence relations 126
6.9 Classes of equivalent objects 128

7 Exercises 129
7.1 Section 1 130
7.2 Section 2 132
7.3 Section 3 134
7.4 Section 4 141
7.5 Section 5 149
7.6 Section 6 152

8 Summary **179**

8.1 The axioms 179

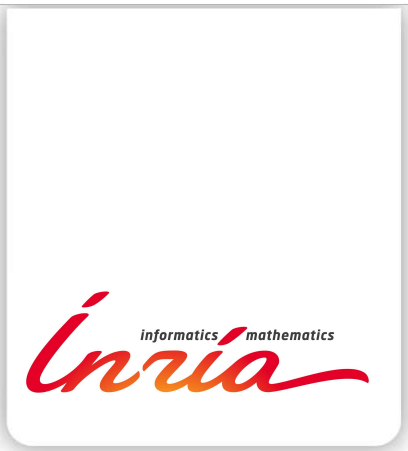
8.2 The Zermelo Fraenkel Theory 179

8.3 Changes from previous versions 180

8.4 Tactics 184

8.5 List of Theorems 189

8.6 Notations and Definitions 192



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399