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

José Grimm

**N° 6999 — version 4**

initial version July 2009 — revised version December 2010

Algorithmics, Programming, Software and Architecture

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[*]

Domain : Algorithmics, Programming, Software and Architecture

Équipe-Projet Apics

Rapport de recherche n° 6999 — version 4 — initial version July 2009 — revised version December 2010 — 229 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/apics/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. There are small differences, marked in footnotes.

**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)

[*] Email: Jose.Grimm@sophia.inria.fr

# 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/apics/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 $\sigma$-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 $\sigma$ 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 $\sigma$ and $\sigma'$; 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 $\sigma$: it should depend on $X'$ rather than on A and $s$. The compatibility condition is then a property of X, $\sigma(X,Y)$ and $\sigma(X',Y)$ 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 $\sigma(X',Y)$ into $\sigma(X',Y)$ 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 three proofs 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.
```

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 $H \vdash A \land B$ by the two tasks $H \vdash A$ and $H \vdash B$, or $H \vdash A \lor B$ by any of the two tasks $H \vdash A$ or $H \vdash B$. If assumption $H_i$ is $A \land B$ it can be replaced by the two assumptions $H_A$ and $H_B$ asserting A and B; if assumption $H_i$ is $A \lor B$, 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 $\land$ and $\lor$ 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 $x \cup y$ is $\bigcup\{x, y\}$, so that $a \in x \cup y$ means $a \in \bigcup\{x, y\}$, so that there is $t$ such that $a \in t$ and $t \in \{x, y\}$. Now $t \in \{x, y\}$ implies $t = x$ or $t = y$, from which we deduce $a \in x$ or $a \in y$. In all three Coq proofs, you can see how definitions are unfolded, the *doubleton_or* theorem is added to the list of assumptions, 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 last proof uses the ssreflect style of programming. It is characterized by the following two properties: each line of the proof is formed of a single sentence (a sequence of semi-colon separated statements that ends with a period); all local names are given explicitly ($x$, $y$, $t$, *td*) instead of being computed by Coq (like *o*, *e*, *H2*, etc). Moreover, the code is indented according to the number of tasks.

Note that all proofs use 4 names for the local variables ($x$, $y$, $a$ and $t$), and respectively 6, 4 and 2 names for the assumptions (by the way, assumption $H_0$ sometimes means $a \in x \cup y$ and $a \in t$). In the first proof all these 6 names are used in the proof script, while in the last proof, only the name *tb* is reused. The script of the last proof is more robust than the other ones, and would have been so, even if we allowed ssrcoq to select the 5 other names. The last proof is slightly longer the the second one (on average, the proofs using the ssreflect style are longer by 7%).

## 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 $(\mathrm{X}_\iota)_{\iota\in\mathrm{I}}(\mathrm{X}_\iota \in \mathfrak{G})$. 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 $\mathrm{pr}_1$ 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 not 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 $(\mathrm{X}_\iota)_{\iota\in\mathrm{I}}$ and $(\mathrm{X}_i)_{i\in\mathrm{I}})$. We can replace these unreadable old German letters by more significant ones. We must also replace I by something else, because this is a reserved letter in Coq. 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 $\mathrm{I} = \mathrm{A}\times\mathrm{B}$, all four identifiers are reserved letters in Simpson's framework.

Quantities named R, B, X, Y, and Z by Simpson have been renamed to Ro, Bo, Xo, Yo and Zo. Quantity A has been removed (it was a prefix version of &). Quantity E has been renamed Bset then Set: this is the type of a Bourbaki set. It will still be denoted by $\mathscr{E}$ here. In our framework, the reserved single-letter identifiers are I J L O P Q S V W.

Coq reserves the letter I as a proof of *True*, the letter O as the integer 0 and the letter S for the function $n \mapsto n+1$ on integers. An ordered pair with values $x$ and $y$ is a term $z$ that has two projections $\mathrm{pr}_1 z = x$ and $\mathrm{pr}_2 z = y$. The constructor is called *kpair*[3] in Coq, and the destructors are called *pr1* and *pr2*. We shall reserve the letters J for the constructor and P, Q for the destructors, so that *J (P z) (Q z) = z* for all pairs $z$ (see section 2.9 for details).

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 construction is so important that we reserve the letter L for it. 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 BL. These two objects $f$

[3]This was called 'pair' in Simpson and in version 1 of this report, 'bpair' in versions 2 and 3

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 the letters V and W respectively. In this document, we shall use standard notations, for instance $\mathrm{pr}_1$ and $\mathrm{pr}_2$ for P and Q, when they exist, calligraphic letters like $\mathscr{V}$ or $\mathscr{W}$ for some objects like V and W, and a slanted font like *is_function* for the general case. Note that *J x y* is a Coq expression meaning the application of J to both arguments *x* and *y*.

There a possibility to change the Coq parser and pretty printer so that `(x,y)` is read as `pair x y`, and `{ x : A | P }` is read as the set of all $x$ in A satisfying the predicate P. We shall not use this feature here. In fact, these are standard notations in Coq for notions that are related but not exactly identical to ours.

Note: in what precedes, the term "reserved" has to be understood as "cannot be used everywhere". In an expression like *Lemma test: forall V, sub V V,* the variable V is a bound variable, and can be replaced by any name, except keywords like *forall*, or other terms that appear in the formula (like *sub*); however *test* or *Lemma* are perfect variables names. We prove such a theorem by saying: if $v$ is any set, then $v \subset v$ is true by some argument. The import/export mechanism of Coq is so that we can replace $v$ by constants like I, O or S (defined in some other modules) but not of V, W, *inc*, or any theorem name of our framework. We can use J, L, P because these are notations.

## 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 $\square$, $\tau$, $\vee$, $\neg$ (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 | $\tau_{\text{letter}}$ (Relation) | Ssign Term$_1$ ... Term$_n$

Relation := $\neg$ Relation | $\vee$ Relation Relation | Rsign Term$_1$ ... Term$_n$

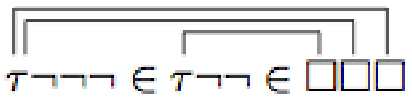
Each sign has to be followed by the appropriate number of terms: $\square$ takes none, $\in$ and $=$ are followed by two terms, and one can extend Bourbaki to non-standard analysis [6] by introducing a specific sign $^{\text{st}}$ of weight 1 qualifying the relation that follows to be standard. Each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

We shall see below that $\tau_x(\mathrm{R})$ has to be interpreted as the expression where all occurrences of $x$ in R are replaced by $\square$ and linked to the $\tau$. Parentheses are removed. This has one advantage: there is no $x$ in $\tau_x(\mathrm{R})$, hence substitution rules become trivial. For instance, the function $x \mapsto x+y$ is constructed by using $\tau$, 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 $\alpha$-*conversion*. Two $\alpha$-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 $A_1A_2A_3\dots$. 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 $A_1A_2\dots A_n$, that contains a *proof*, which is a sub-sequence $A'_1A'_2\dots A'_m$ 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 $A'_1A'_2\dots A'_m$ 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 $\tau_x(R)$ (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 $\tau_x(x \in A' \implies x \in A'')$ (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 $\tau$ is the scope of its operator, thus $\neg\neg \in \square\square$. This means that the two squares are in the scope of both $\tau$, are are linked to the second and first $\tau$ respectively. The third square is in the scope of the first $\tau$ only, hence is linked to the first $\tau$. Formal mathematics in Bourbaki is so complicated that the $\square$ symbol is, in reality, never used.



Denote by $(\boldsymbol{B}|\boldsymbol{x})\ \boldsymbol{A}$ the assembly obtained by replacing $\boldsymbol{x}$, wherever it occurs in $\boldsymbol{A}$, by the assembly $\boldsymbol{B}$. 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 $(\boldsymbol{x}'|\boldsymbol{x})\ \boldsymbol{A}$ provided that $\boldsymbol{x}'$ does not appear in $\boldsymbol{A}$ (informally: since $x$ does not appear in $\tau_x(A)$, 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 $(\boldsymbol{T}|\boldsymbol{x})\ \boldsymbol{A}$ is a term or a relation whenever $\boldsymbol{A}$ is a term or a relation, $\boldsymbol{T}$ is a term, $\boldsymbol{x}$ is a letter.

Abbreviations are allowed, so that $\vee\neg$ can be replaced by $\implies$, 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 $\iff AB$ is the same as $\wedge \implies AB \implies BA$, 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 (we shall use & instead of $\wedge$, since it is easier to type). Note that in Coq, $A \to B$ is the type of a function from A to B but also means $A \implies B$. There is no limit on the number of abbreviations (Bourbaki invented ∅ 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 ∨¬¬∨¬***A***¬***BA***, by virtue of CF5 and CF9. Using abbreviations, this relation can be written as $\Longrightarrow$ ∧***ABA***. 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 $A \cup B \cup C$. This can be interpreted as $(A \cup B) \cup C$ or $A \cup (B \cup C)$. These are two distinct objects that happen to be equal: formally, the relation $(A \cup B) \cup C = A \cup (B \cup C)$ is true. Similarly $A \vee B \vee C$ is ambiguous, but it happens, according to C24, that $(A \vee B) \vee C$ and $A \vee (B \vee C)$ 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 \/ as infix notation for ∨, 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 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 *trans_eq*; 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 $\boldsymbol{A} \implies \boldsymbol{B}$ 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 $\mathrm{A} \implies \mathrm{B}$ is not well-formed. Until the end of E.II.5, Bourbaki uses a special font as in $\boldsymbol{A} \implies \boldsymbol{B}$ to emphasize that $\boldsymbol{A}$ and $\boldsymbol{B}$ are to be replaced by something else.

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

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

is a proof that contains B. Assume that we have two annotated proofs $\mathrm{R}_i$ and $\mathrm{S}_j$, where A is $\mathrm{R}_n$ and $\mathrm{A} \implies \mathrm{B}$ is $\mathrm{S}_m$. 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 $\mathrm{R}_n$ and $\mathrm{S}_m$. 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 $\boldsymbol{P} \implies \boldsymbol{Q}$ and $\boldsymbol{Q} \implies \boldsymbol{R}$. From axiom scheme S4, we get $(\boldsymbol{Q} \implies \boldsymbol{R}) \implies ((\boldsymbol{P} \implies \boldsymbol{Q}) \implies (\boldsymbol{P} \implies \boldsymbol{R}))$. Applying Criterion C1 gives $(\boldsymbol{P} \implies \boldsymbol{Q}) \implies (\boldsymbol{P} \implies \boldsymbol{R})$. Applying it again gives $\boldsymbol{P} \implies \boldsymbol{R}$. If $\mathrm{R}_1, \mathrm{R}_2, \dots, \mathrm{R}_n$ and $\mathrm{S}_1, \mathrm{S}_2, \dots, \mathrm{S}_m$ are proofs of $\boldsymbol{P} \implies \boldsymbol{Q}$ and $\boldsymbol{Q} \implies \boldsymbol{R}$ then a proof of $\boldsymbol{P} \implies \boldsymbol{R}$ is

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

Here $\mathrm{A}_4$ and $\mathrm{D}_y$ 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 $\mathrm{A}_4$, we must give the values of three arguments of the axiom scheme S4, in the case of detachment $\mathrm{D}_y$ 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 $\mathrm{A} \implies \mathrm{B}$ 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 "$(\forall \mathrm{A})(\forall \mathrm{B})((\mathrm{A}\text{ and }\mathrm{B}) \implies \mathrm{A})$".

If P and Q are propositions, one can show that $\neg\neg\mathrm{P} \Rightarrow \mathrm{P}$, $((\mathrm{P} \Rightarrow \mathrm{Q}) \Rightarrow \mathrm{P}) \Rightarrow \mathrm{P}$, $\mathrm{P} \vee \neg\mathrm{P}$, $\neg(\neg\mathrm{P} \wedge \neg\mathrm{Q}) \Rightarrow \mathrm{P} \vee \mathrm{Q}$ and $(\mathrm{P} \Rightarrow \mathrm{Q}) \Rightarrow (\neg\mathrm{P} \vee \mathrm{Q})$ 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 in the next chapter how to implement sets in Coq.

In Coq, we can quantify everything so that criterion C21 becomes a theorem, that 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 proof script is converted into a proof tree:

```
example =
fun (A B : Prop) (H : A /\ B) => and_ind (fun (H0 : A) (_ : B) => H0) H
     : forall A B : Prop, A /\ B -> A
```

This has the form "name = proof : value". The last line is the value of the theorem. The second line is the proof. 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). 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
```

According to this definition, *and_ind* takes three implicit arguments, named A, B and P, its value is a function that takes two arguments, a function $f_2$ and a proof of $A \wedge B$. Here $f_2$ is a function that maps A and B to P. The result is a proof of P. Arguments A, B and P are deduced from $f_2$ and need not be given (on the second line you see expressions of the form `(A:=A)`, this is because *and_rect* has three implicit arguments, that must be explicitly given). With our function $f_2$, P is the first argument hence A. We show here the proof tree of the last variant of *union2_or*. As you can see, proofs in Coq are often hard to read. The important point is that Coq is a proof assistant: it is a system in which inventing a proof is easy. Here in an example.

```
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
```

**Quantified theories** As mentioned above, Bourbaki defines $\tau_x(\mathrm{R})$ as the construction obtained by replacing all $x$ in R by $\square$, adding $\tau$ in front, and drawing a line between $\tau$ 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 $\alpha$-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:T, R* is equivalent in Coq to *ex(fun x:T=>R)*. Here, *ex* is an inductive object. If P is the argument, and if for some witness $x$, $\mathrm{P}(x)$ is true, then *ex P* is defined. From this, we deduce that for some $x$, $\mathrm{P}(x)$ is true. Assume that $f : \mathrm{A} \to \mathrm{B}$ is a surjective function. This means that for each $y$ in B there is an $x$ in A that $f$ maps to $y$. This does not mean that there is a $g$ such that for all $y$, $f(g(y)) = y$. The existence of $g$ is related to the "axiom of choice".

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:

```
nat_ind =
fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> 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 9.1 for details). The first scheme says that if P is a property depending on a variable $z$, if $x = y$, then $P(x)$ and $P(y)$ are equivalent. The second scheme says that if Q and R are two properties depending on a variable $z$, if $Q(z)$ and $R(z)$ are equivalent for all $z$, then $\tau_z(Q) = \tau_z(R)$. The axiom says that if $x \in A$ is equivalent to $x \in B$ then $A = B$ (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 $\tau$ 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 $\tau_z$ 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 $\alpha$ and $\beta$ 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 $\tau$.

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 $P(a, x)$. If 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 $\tau$; There are two exceptions: for defining cardinals and ordinals.

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 quantites $r(X)$, $r(Y)$ 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.

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

In our framework, few objects are defined via $\tau$, 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 $(\forall x)(\boldsymbol{R} \iff \boldsymbol{R})$ 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 $\mathbf{R}$, 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 $(x = y \land y = z) \implies x = z$. 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 *refl_equal*, other theorems are *sym_eq* and *trans_eq*.

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 our 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 $\alpha$-conversion).

We shall add 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). Let $\mathscr{T}$ and $\mathscr{T}'$ be two theories with and without this axiom. If T is a theorem of $\mathscr{T}$, consider T$'$, the same object where P = Q has been replaced by P $\iff$ Q. If we have a proof of T, and substitute = by $\iff$ whenever needed, we get of proof of T$'$ in $\mathscr{T}'$, because Coq uses only Theorems 1, 2 and 3, Scheme S6, Criterion C44, which are true for equivalence.

We shall add an axiom that says if $f$ and $g$ are two functions, of type A $\to$ B, and if $f(x) = g(x)$ whenever $x$ is of type A, then $f = g$. In particular, if $f$ and $g$ are propositional functions, if $f(x)$ is equivalent to $g(x)$ for all $x$, the previous axiom says $f(x) = g(x)$, thus $f = g$. We deduce $\tau_x f = \tau_x g$, which is axiom scheme S7. As explained above, it is only used to defined cardinals and ordinals. Carlos Simpson introduced the following two axioms; the first one is an extension of the previous axiom; the second one says that if $x$ and $y$ are two proofs of the same theorem, they are equal.

```
(*
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 $\mathrm{A}(p)$ and $\mathrm{B}(q)$ 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 $y = \mathrm{A}(p)$ and $y = \mathrm{B}(q)$, and let's apply $\tau_y$. If P is true, it has proof a $p$ and $y = \mathrm{A}(p) = a$; if P is false, then ¬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 ¬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, $\mathrm{X}_i$ 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 ≤ is of type *bool* rather than *Prop*). We do not use the *bool* datatype, thus cannot use the if-then-else construction.

**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.

# Chapter 2

# Sets

This chapter describes the content of the file *set1.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

**Types.** In Coq, each object has a type, for instance 0 is of type *nat*, and the type of *nat* is *Set*; the type of a boolean like *True* is *Prop*; the type of *Set* or *Prop* is *Type*. Our sets will be of type *Set*[1]. For instance *nat* is a set, but neither 0, nor *True*, nor the abstraction $x \mapsto x+1$ nor a record containing sets[2]. The definition given here is not used anymore.

```
Definition Bset:=Set.
```

**Is element of.** 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 $\{x \mid \mathrm{R}(x)\}$ or $\mathscr{E}_x(\mathrm{R}(x))$.

[1]In the original version of C. Simpson, it was *Type*, so that *True* was a set; as a consequence there exists an element $x$ that is in True but not in False, or vice-versa. This is not needed.

[2]In a previous version, a function was such a record, and we could not consider sets of functions

Some relations are not collectivizing, for instance $x \not\in x$. In fact, if we assume that this is equivalent to $x \in y$, replacing $x$ by $y$ gives: $y \not\in y$ is equivalent to $y \in y$, which is absurd. Almost all sets defined by Bourbaki are obtained by application of Axiom A3 (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 $x$, $y$, $a_1, \dots, a_k$, then for all $x_1$, $x_2$, ..., $x_k$, if we denote by $\mathrm{R}(x, y)$ the relation $\mathrm{E}(x, y, x_1, \dots x_n)$, 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 $\mathrm{X}_a$ such that $a \in \mathrm{X}_a$ and a set $\mathrm{X}_b$ 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 chapter, 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 Coq integers 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. The Coq notation *O:nat* says that O is of type nat, we want it to be the same as $z \in \mathbb{N}$ where $z$ stands for 0 and $\mathbb{N}$ for *nat*. In Bourbaki, $z \in \mathbb{N}$, $\mathbb{N} \in z$ and $\mathbb{N} \in \mathbb{N}$ are legal statements; the first one is true, the other ones are false. In order to avoid paradoxes, people sometimes use a hierarchy of sets: if $x \in y$ and $x$ is at level $n$, then $y$ must be at level $n+1$. Thus $x \in x$ is illegal or false. The axiom of foundation states that if $x$ is not empty, there exists $y \in x$ such that the intersection of $x$ and $y$ is empty; it forbids the existence of a sequence of sets with $x_i \in x_{i+1}$ and $x_n \in x_0$. This axiom is independent of all other ones. We shall not use it. However, in Coq there is a type hierarchy, so that there is no $a$ whose type is itself, and no pairs $a$ and $b$ such that $a$ is of type $b$ and $b$ of type $a$. In [5] there is a theorem that says: if the axiom of foundation is true, then X is an ordinal if and only if for all $u$ and $v$ in X we have $u \in v$ or $u = v$ or $v \in u$ and for all $u \in \mathrm{X}$ we have $u \subset \mathrm{X}$. Such a statement becomes impossible in the case of a set hierarchy.

We have the property that *O:nat* is the same as $z \in \mathbb{N}$ if we chose $\mathbb{N}$ to be *nat* and $z$ a function of 0, say $\mathscr{R}0$. We postulate the existence of such a function; note that the type of 0 is *nat* while the type of $\mathscr{R}0$ is *Set*, i.e. the same as (in fact, compatible with) the type of *nat*. Let's define $\mathbb{N}_2$ as the set of even numbers; we have $\mathbb{N}_2 \subset \mathbb{N}$, and $z \in \mathbb{N}_2$. Let $0_2$ be zero in the type *even*. We have $z = \mathscr{R}0 = \mathscr{R}0_2$. This relation will be a consequence of the definition of $0_2$. We want $\mathscr{R}0 \neq \mathscr{R}1$, since otherwise this mechanism is useless. Thus we introduce a parameter and an axiom. Later on[3] we shall define $\mathscr{R}0$ and $\mathscr{R}1$; we shall prove that the axiom is satisfied in this case.

```
Parameter Ro : forall x : Set, x -> Set.
Axiom R_inj : forall (x : Set) (a b : x), Ro a = Ro b -> a = b.
```

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 *In* or *Included* in *Ensembles*, but *inc* or *sub* in our framework.

```
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.
```

**Extensionality.** The axiom of extent is the same as in Bourbaki: if $x \subset y$ and $y \subset x$ then $x = y$. We add another one for functions; if two functions take the same values everywhere we declare them equal.

```
Axiom extensionality : forall a b : Set, sub a b -> sub b a -> a = b.
Axiom arrow_extensionality :
  forall (x y : Type) (u v : x -> y), (forall a : x, u a = v a) -> u = v.
```

Given a type $t$ or a set $x$, it will be declared *nonempty* if there is a witness, i.e., an instance of $t$ or an element of $x$.

```
Inductive nonemptyT (t : Type) : Prop :=
    nonemptyT_intro : t -> nonemptyT t.
Inductive nonempty (x : Set) : Prop :=
    nonempty_intro : forall y : Set, inc y x -> nonempty x.
```

**The axiom of choice.** We assert the existence of a function $\mathscr{C}_\mathrm{T}$ of two parameters $p$ and $q$, depending on a type $t$, that satisfies the following property: if there exists an object $x$ of type $t$ such that $p(x)$ is true, then $c = \mathscr{C}_\mathrm{T}(p,q)$ is an object of type $t$ such that $p(c)$ is true. Note. If we take for $p$ the constant function True, we get: if there exists an element of type $t$, then $c_q$ is of type $t$. But, by construction, $c_q$ is of type $t$. From this, one could deduce that every type is non-empty. Our definition requires that $q$ be of type $t$, so that we get: if $q$ is of type $t$, then $c_q$ is of type $t$; which sounds better. We emphasize that our choice function depends both on $p$ and $q$. This is the equivalent of Bourbaki's $\tau$.

```
Parameter chooseT : forall (t : Type) (p : t -> Prop)(q:nonemptyT t),  t.

Axiom chooseT_pr :
  forall (t : Type) (p : t -> Prop) (ne : nonemptyT t),
    ex p -> p (chooseT p ne).
```

**Images.** The scheme of selection and union is the following: Given a relation $\mathrm{R}(x,y)$; assume that for fixed $y$, we have a set $\mathrm{E}_y$ such that $\mathrm{R}(x,y)$ implies $x \in \mathrm{E}_y$. Then, for every Y, there is a set $\mathrm{Z}_\mathrm{Y}$ containing all $x$ for which there is an $y \in \mathrm{Y}$ such that $\mathrm{R}(x,y)$. 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

[3]Only in version 1 of the software

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 axioms.

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

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

**Double negation axiom.** A property is either true or false. Thus two sets are equal or not.

```
Axiom excluded_middle : forall P : Prop, ~ ~ P -> P.

Lemma excluded_middle': forall A B:Prop, (~ B->A)->(~A->B).
Lemma p_or_not_p : forall P : Prop, P \/ ~ P.
Lemma equal_or_not: forall  x y:Set, x= y \/ x<> y.
Lemma inc_or_not: forall  x y:Set, inc x y \/ ~ (inc x y).
```

We already explained that $P = Q$ is the same as $P \iff Q$ for propositions; the idea is that we can use the *rewrite* and *autorewrite* tactics.

```
Axiom iff_eq : forall P Q : Prop, (P -> Q) -> (Q -> P) -> P = Q.
```

## 2.2 Module constructions

The definition that follows says that $p$ is a predicate, satisfied by a unique object of type $t$. This will be extended later to objects of different types, for instance functions.

```
Definition exists_unique t (p : t->Prop) :=
  (ex p) & (forall x y : t, p x -> p y -> x = y).
```

This defines $x \subsetneq y$.

```
Definition strict_sub (a b : Set) :=  (a <> b) & (sub a b).
```

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 : forall x, sub x x.
Lemma sub_trans : forall a b c, sub a b -> sub b c -> sub a c.
Lemma strict_sub_trans1 :
  forall a b c, strict_sub a b -> sub b c -> strict_sub a c.
Lemma strict_sub_trans2 :
  forall a b c, sub a b -> strict_sub b c -> strict_sub a c.
```

**Empty sets and types.** We say that a set is *empty* if it has no element; by extensionality, it is unique. Bourbaki proves existence of the empty set by noting that for every set $y$, and every property P, there is a set containing all elements of $y$ with the property P. In particular, we can define the complement of $x$ in $y$; taking $x = y$ gives the empty set. 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 $b : x$. In version 3, the empty set is also *Empty_set*.

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

Given a set $x$, we have $b \in x$ if $b = \mathscr{R}a$ for some $a : x$. As a consequence, if $a : x$ then $\mathscr{R}a \in x$. In particular, $x$ is not empty. On the other hand, if $b \in x$ there is an $a$ with $a : x$.

```
Lemma R_inc : forall (x : Set) (a : x), inc (Ro a) x.
Lemma nonemptyT_not_empty : forall x : E, nonemptyT x -> ~ empty x.
Lemma nonemptyT_not_empty0 : forall x:Set, nonemptyT x = nonempty x.
Lemma inc_nonempty : forall x y, inc x y -> nonemptyT y.
Lemma emptyset_sub_any:  forall x, sub emptyset x.
```

**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) (inc_nonempty hyp).

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

If H is a proof of $y \in x$, then $\mathscr{B}$ H is of type $x$; in particular the type $x$ cannot be empty, and $x$ cannot be the empty set. The proof of the next lemma uses the excluded-middle axiom in order to replace $\neg\forall y \neg \mathrm{P}$ by $\exists y\, \mathrm{P}$, where P is $y \in x$. Finally, we have a lemma that says that a set is either empty or nonempty.

```
Lemma not_empty_nonemptyT : forall x, ~ empty x -> nonemptyT x.
Lemma emptyset_pr: forall x, ~ inc x emptyset .
Lemma is_emptyset: forall x, (forall y, ~ (inc y x)) -> x = emptyset.
Lemma emptyset_dichot : forall x, (x = emptyset \/ nonempty x).
Lemma not_nonempty_empty: nonempty emptyset -> False.
```

**Reasoning by cases.** Let T be a type, P a proposition, $a$ a function that maps a proof of P into T and $b$ a function that maps a proof of $\neg$P into T. Since P is true or false, there is a proof $p$ of P or a proof $q$ of its negation, thus $a(p)$ or $b(q)$ is an object of type T; as a consequence the type T is non-empty. We use the axiom of choice to construct $x = \mathscr{C}_\mathrm{C}(a, b)$, an object of type T, such that for every proof $p$ of P we have $a(p) = x$ and for every proof $q$ of $\neg$P we have $b(q) = x$.

```
Definition by_cases_pr (T : Type) (P : Prop) (a : P -> T)
  (b : ~ P -> T) (x : T) :=
  (forall p : P, a p = x) & (forall q : ~ P, b q = x).

Lemma by_cases_nonempty :
 forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T), nonemptyT T.
Definition by_cases (T : Type) (P : Prop) (a : P -> T)  (b : ~ P -> T) :=
  chooseT (fun x : T => by_cases_pr a b x) (by_cases_nonempty a b).
```

Let HT be the assumption that $a(p)$ is independent of the proof $p$ of P and HF the assumption that $b(q)$ is independent of the proof of $\neg$P. The object $\mathscr{C}_C(a,b)$ is well-defined (thus satisfies our claim) only if both assumptions HT and HF are true. However, if P is true, then HF is trivial, since there is no proof of $\neg$P.

```
Lemma by_cases_if :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (p : P),
    (forall p1 p2: P, a p1 = a p2) ->
    by_cases a b = a p.
Lemma by_cases_if_not :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (q : ~ P),
    (forall p1 p2: ~P, b p1 = b p2) ->
    by_cases a b = b q.
```

**Choosing a representative or the empty set.** We simplified a bit the original code (see section 9.3). We construct $\mathscr{C}(p)$, that behaves like $\tau_x$(P): if there is an $x$ with $p(x)$, then $p(\mathscr{C}(p))$ is true; we add the condition that $\mathscr{C}(p)=\emptyset$, if there is no such $x$.

```
Definition choose (p:Set -> Prop) :=
  chooseT (fun x => (ex p -> p x) & ~(ex p) -> x = emptyset)
  (nonemptyT_intro emptyset).

Lemma choose_pr : forall p, ex p -> p (choose p).
Lemma choose_not : forall p, ~(ex p) -> choose p = emptyset.
```

The axiom of choice is used in essentially three cases. Example of the first case: let E be an ordered set, and F a subset of E. We define $\inf_E$ F via the axiom of choice for a certain property $p(x,\mathrm{F})$, for which $p(x,\mathrm{F})$ and $p(y,\mathrm{F})$ imply $x=y$. In this case, the value $\mathscr{C}(p)$ is irrelevant, all we need if that it satisfies $p$ whenever $p$ is satisfied at all. In the second case, the axiom of choice is equivalent to Zermelo's theorem, namely that there exists a function $r(\mathrm{X})$ defined for any set X, depending only on X such that $r(\mathrm{X})\in\mathrm{X}$ whenever X is non-empty. In fact, Zermelo's theorem says that any set E can be well-ordered. If X is a nonempty subset of E, then $\inf_E \mathrm{X}\in\mathrm{X}$ and this gives a choice function on X. Now $\inf_E \mathrm{E}\in\mathrm{E}$ if E is non-empty, and this gives a global choice function.

Finally, let $\mathrm{R}(x,y)$ be an equivalence relation, and let $\mathrm{D}(x)$ be the class of all objects equivalent to $x$; let $f(x)$ be $f(x)=\tau_y(\mathrm{R}(x,y))$. Then $\mathrm{R}(x,y)$ if and only if $f(x)=f(y)$. Assume first that $\mathrm{D}(x)$ is a set. In this case, $\mathrm{R}(x,y)$ is the same as $\mathrm{D}(x)=\mathrm{D}(y)$, and we can define $f$ by applying the choice function to the set $\mathrm{D}(x)$. In some cases, $\mathrm{D}(x)$ is not a set (for instance, the case of cardinals). Our definition makes sense provided that $\mathscr{C}(p)=\mathscr{C}(q)$ if $p(x)$ and $q(x)$ are equivalent whatever $x$. We have an axiom that says $p=q$. This axiom is rarely used, the important point here is that it can be used to prove axiom scheme S7 (see section 9.1).

```
Lemma arrow_EP_exten: forall (p q: Set -> Prop),
  (forall x, p x <-> q x) -> p = q.
Lemma choose_equiv: forall (p q: Set -> Prop),
  (forall x, p x <-> q x) -> choose p = choose q.
```

We consider a variant, $\mathscr{C}_{\mathbb{N}}(p)$, where $x \in \mathbb{N}$; if no element satisfies $p$ then $\mathscr{C}_{\mathbb{N}}(p) = 0$.

```
Definition choosenat p :=
  chooseT (fun u => (ex p -> p u)  &  ~(ex p) -> u = 0)  (nonemptyT_intro 0).
Lemma choosenat_pr : forall p, ex p -> p (choosenat p).
```

**Representatives of nonempty sets.** Fix $z$. Let $p(x)$ be the property $x \in z$. If $z$ is not empty, there is an $y$ such that $p(y)$, hence $\mathscr{C}(p)$ satisfies $p$, this is an element of $z$. This will be denoted by *rep z*.

```
Definition rep (x : Set) := choose (fun y : Set => inc y x).

Lemma nonempty_rep : forall x, nonempty x -> inc (rep x) x.
```

The first of these lemmas is obvious; the second has already been used; it relies on the excluded-middle axiom.

```
Lemma not_exists_pr : forall p : Set -> Prop, (~ ex p) = (forall x : Set, ~ p x).
Lemma exists_proof : forall p : Set -> Prop, ~ (forall x : Set, ~ p x) -> ex p.
```

**Defining a term depending on a boolean value.** This defines a function $\mathscr{Y}(\mathrm{P}, x, y)$ via $\mathscr{C}_{\mathrm{C}}(f, g)$. Here P is a property, $f$ the function that returns $x$ for every proof of P and $g$ the function that returns $y$ for every proof of $\neg$P. As a consequence $\mathscr{Y}(\mathrm{P}, x, y)$ is $x$ if P is true and $y$ otherwise.

```
Definition Yo : Prop -> Set -> Set -> Set :=
  fun P x y => by_cases (fun _ : P => x) (fun _ : ~ P => y).

Lemma Y_if : forall (P : Prop) (hyp : P) x y z, x = z -> Yo P x y = z.
Lemma Y_if_not : forall (P : Prop) (hyp : ~ P) x y z, y = z -> Yo P x z = y.

Lemma Y_if_rw : forall (P:Prop) (hyp :P) x y, Yo P x y = x.
Lemma Y_if_not_rw : forall (P:Prop) (hyp : ~P) x y, Yo P x y = y.
```

A variant[4] 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.
```

We replace the two axioms describing *IM* by a single one.

```
Lemma IM_rw:  forall (x : Set) (f : x -> Set) (y : Set),
  inc y (IM f) = exists a : x, f a = y.
```

[4]New in version 3

**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 $\forall y, \exists X, \forall x, R \implies x \in X$. The conclusion is that for every Y, the relation $\exists y, y \in Y \wedge R$ is collectivizing in $x$. Said otherwise, for every Y, there is a set Z such that $x \in Z$ is equivalent to the existence of $y \in Y$ such that R. A simple case is when R does not depend on $y$. Then, the assumption $\forall x, R(x) \implies x \in X$ implies the existence of Z such that $x \in Z$ is equivalent to $R(x)$. In particular, if $Q(x)$ is any relation, there is a set Z such that $x \in Z$ is equivalent to $x \in Y \wedge Q(x)$. Here is the Coq implementation.[5]

```
(*
Record Zorec (x : Set) (f : x -> Prop) : Set :=
    {Zohead : x; Zotail : f Zohead}.
Definition Zo  := fun (x:Set) (p:Set -> Prop)
  => let P := Zorec (fun a : x => p (Ro a)) in IM (fun t : P => Ro (Zohead t)).
*)
```

In version 3, we changed a bit the definition; it is now:

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

The object $\mathscr{Z}(x, p)$ is the image of some function $g$. This means that $y \in \mathscr{Z}(x, p)$ if and only if there is a $t$ such that $y = g(t)$. If $t$ is the record defined by $a$, it holds $a$ of type $x$ and $p(\mathscr{R}a)$. The function $g(t)$ is then $\mathscr{R}a$. Thus $y = g(t)$ says $y \in x$; it implies $p(y)$. The reverse is true.

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

```
Lemma Z_inc :  forall x p y, inc y x -> p y -> inc y (Zo x p).
Lemma Z_sub :  forall x p,   sub (Zo x p) x.
Lemma Z_all :  forall x p y, inc y (Zo x p) ->  (inc y x) & (p y).
Lemma Z_rw: forall  x p y, inc y (Zo x p) = (inc y x & p y).
```

This defines an auxiliary function *Yy*, with arguments $f$ and $x$. It depends on a property P. If $p$ is a proof of P, the function returns $f(p)$, and if P is false, it returns $x$.[6]

```
Definition Yy : forall P : Prop, (P -> Set) -> Set -> Set :=
  fun P f x => by_cases f (fun _ : ~ P => x).

Lemma Yy_if :
 forall (P : Prop) (hyp : P) (f : P -> Set) x z,
   (forall p1 p2: P, f p1 = f p2) ->
   f hyp = z -> Yy f x = z.
Lemma Yy_if_not :
  forall (P : Prop) (hyp : ~ P) (f : P -> Set) x z, x = z -> Yy f x = z.
```

[5]In the original version, the type of $f$ was $x \to \mathscr{E}$. This is no more possible, since a proposition is not a set.
[6]In version 3, we added the requirement that $f(p)$ be independent of $p$.

We define here $\mathscr{X}(f,y)$ as $Yy(g,\emptyset)$. Given a proof H of $y \in x$, $\mathscr{B}$H is of type $x$; we assume that $f$ is defined for such objects, and pose $g(\mathrm{H}) = f(\mathscr{B}\mathrm{H})$. This is $\mathscr{X}(f,y)$ by definition. If $y = \mathscr{R}z$, where $z$ is of type $x$, we know $\mathscr{B}\mathrm{H} = z$. Said otherwise: $\mathscr{X}(f)$ is some function F, defined for every $y$, whose value is $\emptyset$ if $y \notin x$, and $\mathrm{F}(\mathscr{R}z) = f(z)$ if $z : x$.

```
Definition Xo (x : Set) (f : x -> Set) (y : Set) :=
  Yy (fun hyp : inc y x => f (Bo hyp)) emptyset.

Lemma X_eq :
  forall (x : Set) (f : x -> Set) (y z : Set),
    (exists a : x,  (Ro a = y) & (f a = z)) -> Xo f y = z.

Lemma X_rewrite : forall (x : Set) (f : x -> Set) (a : x), Xo f (Ro a) = f a.
```

## 2.3 Module Little

It is trivial to construct an object *two_points* with two constructors. This gives a set with two distinct elements. We call this the canonical doubleton, the elements will be *TPa* and *TPb*.

```
Inductive two_points : Set :=  | two_points_a | two_points_b.

Definition TPa := Ro two_points_a.
Definition TPb := Ro two_points_b.

Lemma two_points_pr: forall x,
  inc x two_points = (x= TPa \/ x = TPb).
Lemma two_points_distinct: TPa <> TPb.
Lemma two_points_distinctb: TPb <> TPa.
Lemma inc_TPa_two_points: inc TPa two_points.
Lemma inc_TPb_two_points: inc TPb two_points.
```

Given two elements $x$ 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 the canonical doubleton. In Bourbaki, Axiom A2 says that such a set exists. By extensionality, *two_points* is a doubleton.

```
Definition doubleton (x y : Set) :=
  IM (fun t => two_points_rect (fun _ : two_points => Set) x y t).

Lemma doubleton_first : forall x y, inc x (doubleton x y).
Lemma doubleton_second : forall x y, inc y (doubleton x y).
Lemma doubleton_or : forall x y z, inc z (doubleton x y) -> z = x \/ z = y.
Lemma doubleton_rw : forall x y z : Set,
  inc z (doubleton x y) = (z = x \/ z = y).

Lemma two_points_pr2: doubleton TPa TPb =  two_points.

Lemma doubleton_inj :  forall x y z w : Set,
    doubleton x y = doubleton z w ->  (x = z & y = w) \/  (x = w & y = z).

Lemma doubleton_singleton : forall x, doubleton x x = singleton x.
```

We originally defined a *singleton* as the image of a set with one element. We now proceed as in Bourbaki, namely to define it as *doubleton x x*. We denote this as $\{x\}$. By construction

$z \in \{x\} \iff z = x$. From this one can deduce that a singleton is nonempty, and we have an extensionality property.

```
(*
   Inductive one_point : Set :=  one_point_intro : one_point.
   Definition singleton (x : Set) := IM (fun p : one_point => x).
*)

Definition singleton x := doubleton x x.

Lemma doubleton_singleton : forall x, doubleton x x = singleton x.
Lemma singleton_inc : forall x, inc x (singleton x).
Lemma singleton_eq : forall x y, inc y (singleton x) -> y = x.
Lemma singleton_inj : forall x y, singleton x = singleton y -> x = y.
Lemma singleton_rw: forall x y, inc y (singleton x) = (y = x).
Lemma nonempty_singleton: forall x, nonempty (singleton x).
Lemma sub_singleton: forall x y, inc x y -> sub (singleton x) y.
Lemma is_singleton_pr: forall y x,
 ( (inc x y) & (forall z, inc z y -> z = x)) -> y = singleton x.
```

If $x \in z$ and $y \in z$ then $\{x, y\} \subset z$. A doubleton is nonempty. We have $\{x, y\} = \{y, x\}$.

```
Lemma nonempty_doubleton: forall x y, nonempty (doubleton x y).
Lemma sub_doubleton: forall x y z,
  inc x z -> inc y z -> sub (doubleton x y) z.
Lemma doubleton_symm: forall a b,
  doubleton a b = doubleton b a.
```

## 2.4 Module Complement

The *complement* in $a$ of $b$, denoted $a \setminus b$ or sometimes $a - b$, is the subset of $a$ formed of elements not in $b$; it is the set of all elements in $a$ but not in $b$. If $x$ is in $a$ but not in $a \setminus b$, then it is in $b$. If $a \setminus b$ is empty, then $a \subset b$.

```
Definition complement (a b : Set) := Zo a (fun x : Set => ~ inc x b).

Lemma complement_rw :
 forall a b x, inc x (complement a b) = (inc x a & ~ inc x b).
Lemma use_complement :
 forall a b x, inc x a -> ~ inc x (complement a b) -> inc x b.
```

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

```
Lemma sub_complement: forall a b, sub (complement a b) a.
Lemma strict_sub_nonempty_complement : forall x y,
  strict_sub x y -> nonempty (complement y x).

Lemma double_complement: forall a x,
  sub a x -> complement x (complement x a) = a.
Lemma complement_itself : forall x,  complement x x = emptyset.
Lemma complement_emptyset : forall x,  complement x emptyset = x.
Lemma empty_complement: forall a b,
  complement a b = emptyset -> sub a b.
Lemma not_inc_complement_singleton: forall a b,
  ~ (inc b (complement a (singleton b))).
Lemma complement_monotone : forall a b x,
  sub a x -> sub b x -> (sub a b = sub (complement x b) (complement x a)).
```

## 2.5 Module Image

This module contained initially 8 lemmas, and only 2 of them will be used in what follows. 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 : Set -> Set) := IM (fun a : x => f (Ro a)).

Lemma inc_fun_image : forall x f a, inc a x -> inc (f a) (fun_image x f).
Lemma fun_image_rw : forall  f x y,
  inc y (fun_image x f) = exists z, (inc z x & f z = y).
```

## 2.6 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 $x$. C. Simpson considered the set of cuts (see page 194). We changed the definition as follows. Consider a set X with two elements A and B, and all functions with $f$ values in X; then $f^{-1}(\mathrm{A})$, the set of all elements $z \in x$ such that $f(z) = \mathrm{A}$, is a subset of $x$ and every subset of $x$ has this form.

```
(* Definition powerset (x : Set) := IM (fun p : x -> Prop => cut p). *)
Definition powerset (x : Set) :=
  IM (fun p : x -> two_points =>
    Zo x (fun y : Set => forall hyp : inc y x, p (Bo hyp) = two_points_a)).
```

The characteristic property is that $y \subset x$ if and only if $y \in \mathfrak{P}(x)$.

```
Lemma powerset_inc : forall x y, sub x y -> inc x (powerset y).
Lemma powerset_sub : forall x y, inc x (powerset y) -> sub x y.
Lemma powerset_inc_rw : forall x y, inc x (powerset y) = sub x y.
Lemma inc_x_powerset_x: forall x, inc x (powerset x).

Lemma powerset_monotone: forall a b,
  sub a b -> sub (powerset a) (powerset b).
Lemma powerset_emptyset :
  powerset emptyset = singleton emptyset.
```

## 2.7 Module Union

Bourbaki defines the *union* $\bigcup\limits_{\iota\in \mathrm{I}} \mathrm{X}_\iota$ of a family of sets. This means that we have a set I and a mapping $\iota \mapsto \mathrm{X}_\iota$ defined for $i \in \mathrm{I}$. If the mapping is the identity, which is the case considered here, we get the union of a set of sets, denoted by $\bigcup \mathrm{X}$. The union exists as a direct consequence of S8 (Scheme of Selection and Union). The assumption is $(\forall i)(\exists \mathrm{Z})(\forall x)(i \in \mathrm{I}$ and $x \in \mathrm{X}_i) \implies x \in \mathrm{Z}$ (take $\mathrm{Z} = \mathrm{X}_i$). The conclusion is the existence of a set containing all elements satisfying $(\exists i)(i \in \mathrm{I}$ and $x \in \mathrm{X}_i)$. Instead of an axiom, we use the following construction:

```
Record Union_integral (x : Set) : Set :=
  {Union_param : x; Union_elt : Ro Union_param}.
Definition union (x : Set) :=
  IM (fun i : Union_integral x => Ro (Union_elt (x:=x) i)).
```

A *Union_integral* record contains two fields, say $p$ and $e$. Let $q = \mathscr{R}p$. Since $p$ is type $x$, we have $q \in x$. An object $y$ is in the union if $y = \mathscr{R}e$ for some integral record. Since $e$ is of type $q$, this means $y \in q$. The next two lemmas are then obvious; the third one is easy. Bourbaki considers the union of subsets of $x$; according to the last lemma, this is a subset of $x$.

```
Lemma union_inc : forall x y a, inc x y -> inc y a -> inc x (union a).
Lemma union_exists : forall x a,
    inc x (union a) -> exists y,  inc x y & inc y a.
Lemma union_rw: forall x a,
  inc x (union a)= (exists y, inc x y & inc y a).
Lemma union_sub: forall x y, inc x y -> sub x (union y).
Lemma sub_union : forall x z,
   (forall y, inc y z -> sub y x)-> sub (union z) x.
```

The union a family of two sets X and Y 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 $B \subset A \cup B$.

```
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.
Lemma union2_first : forall x y a, inc a x -> inc a (union2 x y).
Lemma union2_second : forall x y a, inc a y -> inc a (union2 x y).
Lemma union2_rw : forall a b x,
  inc x (union2 a b) = (inc x a \/ inc x b).
Lemma union2sub_first: forall a b, sub a (union2 a b).
Lemma union2sub_second: forall a b, sub b (union2 a b).
Lemma union2idem:  forall x, union2 x x = x.
Lemma union2comm:  forall x y, union2 x y = union2 y x.
Lemma union2_sub: forall x y,  sub x y = (union2 x y = y).
```

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

```
Definition tack_on x y := union2 x (singleton y).

Lemma tack_on_or : forall x y z : Set, inc z (tack_on x y) ->
  (inc z x \/ z = y).
Lemma tack_on_rw: forall x y z,
  (inc z (tack_on x y) ) = (inc z x \/ z = y).

Lemma inc_tack_on_x: forall a b, inc a (tack_on b a).
Lemma inc_tack_on_sub: forall a b, sub b (tack_on b a).
Lemma inc_tack_on_y: forall a b y, inc y b -> inc y (tack_on b a).

Lemma tack_on_when_inc: forall x y, inc y x -> tack_on x y = x.
Lemma tack_on_sub: forall x y z, sub x z -> inc y z -> sub (tack_on x y) z.
Lemma tack_on_complement: forall x y, inc y x ->
  x = tack_on (complement x (singleton y)) y.
```

## 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 the next 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. C. Simpson has chosen *rep X*, this makes the intersection of an empty family undefined. There is in fact a better solution: it suffices to take for E the union of the family. This 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 union_empty: union emptyset = emptyset.
Lemma intersection_empty: intersection emptyset = emptyset.
Lemma intersection_forall :
  forall x a y, inc a (intersection x) -> inc y x -> inc a y.
Lemma intersection_sub : forall x y, inc y x -> sub (intersection x) y.
```

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

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

Lemma intersection2_inc : forall x y a,
  inc a x -> inc a y -> inc a (intersection2 x y).
Lemma intersection2_first :  forall x y a,
  inc a (intersection2 x y) -> inc a x.
Lemma intersection2_second : forall x y a,
  inc a (intersection2 x y) -> inc a y.
Lemma intersection2_both: forall x y a,
  inc a (intersection2 x y) -> (inc a x & inc a y).
Lemma intersection2_rw: forall x y a,
  inc a (intersection2 x y) = (inc a x & inc a y).

Lemma intersection2sub_first: forall a b, sub (intersection2 a b) a.
Lemma intersection2sub_second: forall a b, sub (intersection2 a b) b.
Lemma intersection2idem: forall x, intersection2 x x = x.
Lemma intersection2comm: forall x y, intersection2 x y = intersection2 y x.
Lemma intersection2_sub: forall x y,
  sub x y = (intersection2 x y = x).


Lemma union_singleton: forall x,
  union (singleton x) = x.
Lemma intersection_singleton: forall x,
  intersection (singleton x) = x.
```

## 2.9 Module Pair

This module has been changed more than once. We shall explain a bit the different variants. We define here an operator J and two projectors P and Q (these are notations for some Coq functions whose names are irrelevant here). The quantity $J\,x\,y$ is denoted by $(x, y)$, while $P\,z$ and $Q\,z$ are denoted $\text{pr}_1$ and $\text{pr}_2 z$. The operator satisfies: if $(x, y) = (x', y')$ then $x = x'$ and $y = y'$. We say that $z$ is a pair if there exists $x$ and $y$ such that $z = (x, y)$. In this case $x = \text{pr}_1 z$ and $y = \text{pr}_2 z$.

Implementation 1. We follow the 1956 Edition of Bourbaki (Translated as [2]), define J via an axiom, P and Q via the axiom of choice (if $z$ is a pair, we select the unique $x$ such that there is a $y$ such that $z = (x, y)$.

```
(*
  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').
*)
```

We define a pair and state the two properties.

```
Definition is_pair (u : Set) := exists x, exists y, u = J x y.
Lemma pr1_def: forall a b c d, J a b = J c d -> a = c.
Lemma pr2_def: forall a b c d, J a b = J c d -> b = d.
```

This is how we can define P and Q.

```
(*
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)).
*)
```

Implementation 2. We define a pair to be $\{\{x\}, \{\emptyset, \{y\}\}\}$

```
(*
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_distincta:forall x y z w,
  pair_first x y <> pair_second z w.
Lemma pair_distinct:forall x y,
  pair_second x y <>  pair_first x y.

Lemma pr1_injective : forall x y z w : Set,
  pair x y = pair z w -> x = z.
Lemma pr2_injective : forall x y z w,
  pair x y = pair z w -> y = w.
*)
```

The definition above was used by C. Simpson. It has the particularity that the pair $(x, y)$ is a doubleton $\{a, b\}$ with $a = \{x\}$ and $b = \{\emptyset, \{y\}\}$. Note that $a$ has one element while $b$ has two elements, so that $(x, y)$ is a set with two distinct elements. This property was used in Version 1. One can replace $a$ by $a = \{\{x\}\}$. With $a$ and $b$ exchanged, this gives the definition used by Wiener in 1914. It has the advantage, in the case of a hierarchy of sets, that if $x$ and $y$ are at the same level, then so are $a$ and $b$.

One can use $a = \{x\}$ and $b = \{x, y\}$, so that a pair is $\{\{x\}, \{x, y\}\}$. This definition was introduced by Kuratowski in 1923, and used in [3]. This satisfies the same properties as above (initially called *pr1_injective*, but renamed to *pr1_def* since the projector is not injective).

Implementation 3 is as follows.

```
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).
Notation J := kpair.
Notation P := kpr1.
Notation Q := kpr2.
```

Before version 4, union and intersection were defined after pairs; thus we has to re-order the modules. We define $(x, y)$ as the doubleton $\{a, b\}$ with $a = \{x\}$ and $b = \{x, y\}$ as proposed by Kuratowski. We note that $a \cup b = b$ while $a \cap b = a$, so that $a$ and $b$ can be deduced from the unordered pair $\{a, b\}$. The union of the elements of $a$ is $x$, so that the first projection is well defined. Let $c$ be the complement of $a$ in $b$. If $c$ is empty, then $y = x$, otherwise $c = \{y\}$, so that the second projector is well-defined.

```
Lemma kpr0_pair: forall x y, intersection (J x y) = singleton x.
Lemma pr1_pair: forall x y, P (J x y) = x.
Lemma pr2_pair: forall x y, Q (J x y) = y.
```

The following properties are independent of the variants used.

```
Lemma pair_recov : forall u, is_pair u -> J (P u) (Q u) = u.
Lemma pair_is_pair : forall x y, is_pair (J x y).
Lemma is_pair_rw : forall x, is_pair x = (x = J (P x) (Q x)).
Lemma pr1_pair : forall x y, P (J x y) = x.
Lemma pr2_pair : forall x y, Q (J x y) = y.

Lemma pair_extensionality : forall a b,
  is_pair a -> is_pair b -> P a = P b -> Q a = Q b -> a = b.
```

A set of pairs is sometimes called a graph (or a relation in the original work of C. Simpson). We denote by $\mathcal{V}(x, g)$ an element $y$, if it exists, such $(x, y)$ is in the graph $g$. If there is an $y$ such that $(x, y)$ is in the graph, then $(x, \mathcal{V}(x, g))$ is in the graph. Otherwise, $\mathcal{V}(x, g) = \emptyset$. Later on, we shall define the domain as the set of all $x$ for which there a $y$; a graph is said functional (on its domain E) if for every $x$ (in the set E) there is a unique $y$ such that $(x, y)$ is in the graph.

```
Definition V (x f : Set) := choose (fun y : Set => inc (J x y) f).

Lemma V_inc : forall x z f,
 (exists y, inc (J x y) f) -> z = V x f -> inc (J x z) f.
```

```
Lemma V_or : forall x f,
  (inc (J x (V x f)) f) \/
  ((forall z, ~(inc (J x z) f)) & (V x f = emptyset)).
```

## 2.10 Module Cartesian

**Note:** Implementation has changed in Version 3, for the old definition, see section 9.3.

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

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

Lemma product_inc_rw : forall x y z,
  inc x (product y z) = (is_pair x & inc (P x) y & inc (Q x) z).
Lemma product_pr : forall a b u,
  inc u (product a b) -> (is_pair u & inc (P u) a & inc (Q u) b).
Lemma product_inc :  forall a b u,
  is_pair u -> inc (P u) a -> inc (Q u) b -> inc u (product a b).
Lemma product_pair_pr :  forall a b x y,
  inc (J x y) (product a b) -> (inc x a & inc y b).
Lemma product_pair_inc :  forall a b x y,
  inc x a -> inc y b -> inc (J x y) (product a b).
Lemma pair_in_product: forall a b c, inc a (product b c) -> is_pair a.
Lemma product_pair_rw : forall a b x y : Set,
  inc (J x y) (product a 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 empty_product1: forall y,   product emptyset y = emptyset.
Lemma empty_product2: forall x,   product x emptyset = emptyset.
Lemma empty_product_pr: forall x y,
   product x y = emptyset -> (x = emptyset \/ y= emptyset).
```

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

```
Lemma product_monotone_left: forall x x' y,
  sub x x' -> sub (product x y) (product x' y).
Lemma product_monotone_right: forall x y y',
  sub y y' -> sub (product x y) (product x y').
Lemma product_monotone: forall x x' y y',
  sub x x' -> sub y y' -> sub (product x y) (product x' y').
Lemma product_monotone_left2: forall x x' y, nonempty y ->
  sub (product x y) (product x' y) -> sub x x'.
Lemma product_monotone_right2: forall x y y', nonempty x ->
  sub (product x y) (product x y') -> sub y y'.
```

# Chapter 3

# Functions

We describe in this Chapter some basic properties of functions and functional graphs (part of the file *func.v* by Carlos Simpson, without definitions that seemed useless for the Bourbaki project). The module defining equivalence relations will be described in Chapter 6. In version 4, we renamed some lemmas from 'function' to 'fgraph'.

## 3.1 Module Function

A *graph* is a set of pairs. The *domain* and *range* are the images of the first and second projection. A set $f$ satisfies the *fgraph* property if it is a graph and if the first projection is injective. This means that if $(a,b) \in f$ and $(a,b') \in f$ then $b = b'$ (claim that will be proved in the next Chapter).

```
Definition is_graph r := forall y, inc y r -> is_pair y.
Definition domain f  := fun_image f P.
Definition range f := fun_image f Q.
Definition fgraph f :=
  is_graph f & (forall x y, inc x f -> inc y f -> P x = P y -> x = y).
```

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

```
Lemma domain0_rw: forall r x, inc x (domain r) = exists y, inc y r & P y =x.
Lemma range0_rw: forall r x, inc x (range r) = exists y, inc y r & Q y =x.
```

Some properties of a functional graph.

```
Lemma fgraph_is_graph : forall f, fgraph f -> is_graph f.
Lemma fgraph_pr: forall f x y y',
  fgraph f -> inc (J x y) f -> inc (J x y') f -> y = y'.
```

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

```
Lemma domain_rw: forall r x,
  is_graph r -> inc x (domain r) = (exists y, inc (J x y) r).
Lemma range_rw: forall r y,
  is_graph r -> inc y (range r) = (exists x, inc (J x y) r).
```

These lemmas are obvious from the definitions.

```
Lemma inc_pr1_domain : forall f x,
  inc x f -> inc (P x) (domain f).
Lemma inc_pr2_range : forall f x,
  inc x f -> inc (Q x) (range f).
```

The first lemma says that if $x$ is in the graph, then $x$ is a pair whose second component is $\mathcal{V}(\mathrm{pr}_1 x, f)$. The second lemma says that $y$ is in the range if and only if it is $\mathcal{V}(x, f)$ for some $x$ in the domain. If $x$ is in the domain, then $(x, \mathcal{V}(x, f))$ is in the graph.

```
Lemma in_graph_V : forall f x,
  fgraph f -> inc x f -> x = J (P x) (V (P x) f).

Lemma frange_inc_rw : forall f y,
  fgraph f -> inc y (range f) = (exists x, inc x (domain f) & y = V x f).
Lemma fdomain_pr1 : forall f x,
  fgraph f -> inc x (domain f) -> inc (J x (V x f)) f.

Lemma pr2_V : forall f x,
  fgraph f -> inc x f -> Q x = V (P x) f.

Lemma inc_V_range:  forall f x,
  fgraph f -> inc x (domain f) -> inc (V x f) (range f).
```

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 : forall f g, fgraph g -> sub f g -> fgraph f.
Lemma sub_graph_domain : forall f g, sub f g -> sub (domain f) (domain g).
Lemma sub_graph_range : forall f g, sub f g -> sub (range f) (range g).
Lemma sub_graph_ev: forall f g x,
  fgraph g -> sub f g -> inc x (domain f) -> V x f = V x g.
Lemma fgraph_sub : forall f g,
  fgraph f -> fgraph g ->
  sub  (domain f) (domain g) ->
  (forall x, inc x (domain f) -> V x f = V x g) -> sub f g.
Lemma fgraph_exten: forall f g,
  fgraph f -> fgraph g -> domain f = domain g ->
  (forall x, inc x (domain f) -> V x f = V x g) -> f = g.
```

Given two sets A and B, the range of the union is the union of the ranges. The same holds for the domain. If the sets are functional graphs, the union is a functional graph, provided that the intersection of the domains is empty.

```
Lemma range_union2: forall a b,
   range (union2 a b) = union2 (range a) (range b).
Lemma domain_union2: forall a b,
   domain (union2 a b) = union2 (domain a) (domain b).
Lemma fgraph_union2: forall a b, fgraph a -> fgraph b ->
  intersection2 (domain a) (domain b) = emptyset ->
  fgraph (union2 a b).
```

¶ Inverse image of a set $a$ by a graph $f$, denoted $\overset{-1}{f}\langle a\rangle$ or simply $f^{-1}\langle a\rangle$. This is a part of the domain, characterized by the property that $x \in f^{-1}\langle a\rangle$ if and only if $\mathcal{V}(x, f) \in a$.

```
Definition inverse_image (a f : Set) :=
  Zo (domain f) (fun x  => inc (V x f) a).

Lemma inverse_image_sub :   forall a f,
  sub (inverse_image a f) (domain f).

Lemma inverse_image_rw :   forall a f x,
  inc x (inverse_image a f)= ( inc x (domain f) & inc (V x f) a).
Lemma inverse_image_inc :  forall a f x,
  inc x (domain f) -> inc (V x f) a -> inc x (inverse_image a f).
Lemma inverse_image_pr : forall a f x,
  inc x (inverse_image a f) -> inc (V x f) a.
```

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 $f$.

```
Definition fgraph_create (x : Set) (p : Set) :=
  fun_image x (fun y => J y (p y)).
Notation L := function_create.

Lemma L_inc_rw: forall x p y,
  inc y (L x p) = exists z, inc z x & J z (p z) = y.
Lemma L_fgraph : forall p x, fgraph (L x p).
Lemma L_domain : forall x p, domain (L x p) = x.
Lemma L_V_rw :  forall x p y,
  inc y x -> V y (L x p) = p y.
```

The range of $\mathscr{L}_x f$ is the image $f\langle x\rangle$ (according to Section 2.5; on page 42 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 L_range :  forall p x,
  range (L x p) = fun_image x p.
Lemma L_create : forall  a f,
  L a (fun x => V x (L a f)) = L a f.
Lemma L_range_rw: forall sf f a,
  inc a (range (L sf f))  = exists b, inc b sf & f b = a.
Lemma L_V_out : forall x f y,
  ~inc y x -> V y (L x f) = emptyset.
Lemma L_recovers :  forall f,
  fgraph f -> L (domain f) (fun x : Set => V x f) = f.
Lemma L_exten1 : forall a b f g,
  a = b ->  (forall x, inc x a -> f x = g x) ->
  L a f = L b g.
```

¶ We denote by $g \circ f$ the *composition* of the two functions. It maps $x$ to $g(f(x))$. In the case of graphs, the evaluation function is $\mathscr{V}(\mathscr{V}(x, f), g)$; note that the order is reversed. The domain is the set of all $x$ in the domain of $f$ that are mapped to the domain of $g$, it is the inverse image of the domain of $g$ by $f$. We do not like this definition, thus introduce an alternate one, that agrees if functions are composable. Note that the last lemma makes no assumptions on $f$ and $g$. The easy case is when the two objects are composable (in particular, they are functions). In this case the domain of $g \circ f$ is the domain of $f$.

```
Definition fcomposable (f g : Set) :=
  fgraph f & fgraph g & sub (range g) (domain f).

Definition fcompose (f g : Set) :=
  L (inverse_image (domain f) g) (fun y => V (V y g) f).
Definition gcompose g f := L(domain f)  (fun y => V (V y f) g).

Lemma fcompose_fgraph : forall f g, fgraph (fcompose f g).
Lemma fcompose_domain :  forall f g,
  domain (fcompose f g) = inverse_image (domain f) g.
Lemma fcompose_range:  forall f g, fgraph f ->
  sub (range (fcompose f g)) (range f).
Lemma fcomposable_domain : forall f g,
  fcomposable f g -> domain (fcompose f g) = domain g.
Lemma alternate_compose: forall g f,
  fcomposable g f -> gcompose g f = fcompose g f.
Lemma fcompose_ev : forall x f g,
  inc x (domain (fcompose f g)) -> V x (fcompose f g) = V (V x g) f.
```

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) := L x (fun y : Set => y).

Lemma identity_fgraph : forall x, fgraph (identity_g x).
Lemma identity_domain : forall x, domain (identity_g x) = x.
Lemma identity_range: forall x, range (identity_g x) = x.
Lemma identity_ev : forall x a, inc x a -> V x (identity_g a) = x.
```

Given two sets $f$ and $x$, one can consider the set of all $y \in f$ satisfying $\text{pr}_1 y \in x$. This makes sense if $f$ is a graph, it is called the *restriction* of $f$ to $x$. In fact, since this is a subset of $f$, it is a functional graph whenever $f$ is. Its domain is the intersection of $f$ and $x$. On the restriction domain the function takes the same value as the restriction.

```
Definition restr f x :=
  Zo f (fun y=> inc (P y) x).

Lemma restr_inc_rw : forall f x y,
  inc y (restr f x) = (inc y f & inc (P y) x).
Lemma restr_sub : forall f x,
   sub (restr f x) f.
Lemma restr_fgraph : forall f x,
  fgraph f -> fgraph (restr f x).
Lemma restr_graph: forall x r,
  is_graph r -> is_graph (restr r x).
Lemma restr_domain : forall f x,
  fgraph f ->  domain (restr f x) = intersection2 (domain f) x.
Lemma restr_domain1 : forall f x,
  fgraph f -> sub x (domain f) -> domain (restr f x) = x.
Lemma restr_ev : forall f u x,
  fgraph f -> sub u (domain f) -> inc x u ->
    V x (restr f u) = V x f.
Lemma restr_ev1 : forall f u x,
  fgraph f -> inc x  (domain f) -> inc x u ->
    V x (restr f u) = V x f.
```

```
Lemma fgraph_sub_eq : forall r s,
  fgraph r -> fgraph s -> sub r s ->
    sub (domain s) (domain r) -> r = s.
Lemma fgraph_sub_V : forall f g x,
  fgraph g -> inc x (domain f) -> sub f g ->  V x f = V x g.
Lemma restr_to_domain : forall f g,
  fgraph f -> fgraph g -> sub f g ->  restr g (domain f) = f.
Lemma restr_to_domain2 : forall x f,
  fgraph f -> sub x (domain f) ->  L x (fun i  => V i f) = restr f x.
Lemma double_restr: forall f a b, fgraph f ->
  sub a b -> sub b (domain f) ->
  (restr (restr f b) a) = (restr f a).
```

The union of functional graphs is a graph, provided that some compatibility condition holds. The domain is the union of the domains. The range is the union of the ranges. We consider the special case of a union of a graph and the singleton $\{(x, y)\}$.

```
Lemma domain_union : forall z, domain (union z) =
  union (fun_image z domain).

Lemma tack_on_domain : forall f x y,
  domain (tack_on f (J x y)) = tack_on (domain f) x.

Lemma range_union : forall z, range (union z) =
  union (fun_image z range).

Lemma tack_on_range : forall f x y,
  range (tack_on f (J x y)) = tack_on (range f) y.

Lemma tack_on_fgraph : forall f x y,
  fgraph f -> ~inc x (domain f) ->
  fgraph (tack_on f (J x y)).
```

Given a function that takes an argument of type $x$, we know how to convert it to a function defined on the set $x$. We can then take its graph.

```
Definition tcreate (x:Set) (f:x->Set) :=
  L x (fun y => (Yy (fun (hyp : inc y x) => f (Bo hyp)) emptyset)).

Lemma tcreate_value_type : forall x (f:x->Set) y,
  V (Ro y) (tcreate f) = f y.

Lemma tcreate_value_inc : forall x (f:x->Set) y (hyp : inc y x),
  V y (tcreate f) = f (Bo hyp).
Lemma tcreate_domain : forall x (f:x->Set), domain (tcreate f) = x.
```

# Chapter 4

# 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).

We consider here some properties of sections 1 (Collectivizing relations) and 2 (Ordered pairs) not implemented by Carlos Simpson in [2].

A property $\mathrm{P}(y)$ is collectivizing if there is a set $x$ such that $\mathrm{P}(y)$ is equivalent to $y \in x$. There are properties that are not collectivizing, for instance $y \notin y$. The second lemma says that there is a set containing no set but there is no set containing all sets.

```
Lemma not_collectivizing_notin:
  ~ (exists z, forall y, inc y z = not (inc y y)).
Lemma collectivizing_special :
  (exists x, forall y, ~ (inc y x)) &  ~ (exists x, forall y, inc y x).
```

Additional properties of the empty set. We have $\emptyset \subset x$ for every $x$, but the converse is true only if $x$ is the empty set. Since $x \in \emptyset$ is absurd, everything can be deduced from it.

```
Lemma emptyset_pr: forall x,  inc x emptyset ->  False.
Lemma emptyset_pra: forall x (p: EP),  inc x emptyset -> (p x).
Lemma sub_emptyset : forall x,  sub x emptyset = (x = emptyset).
```

## 4.1 Graphs and correspondences

A graph $r$ is a set of pairs; if the pair $(x, y)$ is an element of $r$ we say that $x$ and $y$ are related by $r$. This will be used essentially when $r$ is the graph of a relation.

```
Definition related r x y := inc (pair x y) r.
```

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 2.5; it is the domain of $r$.

```
Theorem range_domain_exists: forall r,
  is_graph r ->
  (exists_unique (fun a=> (forall x, inc x a = (exists y, inc (J x y) r))) &
    exists_unique (fun b=> (forall y, inc y b  = (exists x, inc (J x y) r)))).
```

A graph is a subset of the product of the domain by the range. A graph is empty if and only if its domain or range is empty. A functional graph is a graph.

```
Lemma sub_graph_prod: forall r, is_graph r ->
  sub r (product (domain r)(range r)).
Lemma empty_graph1: forall r, is_graph r ->
  (domain r = emptyset) = (r = emptyset).
Lemma empty_graph2: forall r, is_graph r ->
  (range r = emptyset) =  (r = emptyset).
Lemma graph_fgraph : forall f, fgraph f -> is_graph f.
```

The emptyset is a functional graph, with empty range and domain.

```
Lemma emptyset_graph: is_graph (emptyset).
Lemma emptyset_range: range emptyset = emptyset.
Lemma emptyset_domain: domain emptyset = emptyset.
Lemma emptyset_fgraph: fgraph emptyset.
```

A product $x \times y$ is a graph. The domain is $x$, the range is $y$. Note that, if one set is empty, then the product is empty, see above. It is a functional graph if the range is a singleton.

```
Lemma product_is_graph: forall x y,
  is_graph (product x y).
Lemma product_related: forall x y a b,
  related (product x y) a b = (inc a x & inc b y).
Lemma product_domain: forall x y,
  nonempty y-> domain (product x y) = x.
Lemma product_range: forall x y,
  nonempty x -> range (product x y) = y.
Lemma constant_function_p1: forall x y,
  fgraph (product x (singleton y)).
```

The *diagonal* of $x$, denoted $\Delta_x$, is the set of all pairs $(a, a)$, with $a \in x$. This is the graph of the identity function on $x$, domain and range being $x$. In what follows, we shall use *identity_g*, but keep the word 'diagonal' in some theorem names.

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

Lemma diagonal_is_identity: forall x, diagonal x = identity_g x.
Lemma inc_diagonal_rw: forall x u,
  inc u (identity_g x) = (is_pair u & inc (P u) x & P u = Q u).
Lemma inc_pair_diagonal: forall x u v,
  inc (J u v) (identity_g x)  = (inc u x & u = v).
Lemma identity_graph:  forall x, is_graph (identity_g x).
```

For Bourbaki, a *correspondence* between A and B is a triple $\Gamma = (G, A, B)$ where the domain of G is a subset of A and the range is a subset of B. The three conditions "G is a graph whose domain is a subset of A and whose range is a subset of B", $G \subset A \times B$, and $G \in \mathfrak{P}(A \times B)$ are equivalent.

```
Definition corr_propb s t g:=  sub g (product s t).
Lemma corr_propa: forall x y z,
  corr_propb x y z = inc z (powerset (product x y)).
Lemma corr_propcc: forall  s t g,
  sub g (product s t) =  (is_graph g & sub (domain g) s & sub (range g) t).
```

Here we use a triple, with the condition $G \subset A \times B$.

```
Definition is_triple  f := is_pair f & is_pair (Q f).
Definition source x := P (Q x).
Definition target x := Q (Q x).
Definition graph x := P x.
Definition corresp s t g  := J g (J s t).

Definition is_correspondence f  :=
  is_triple f i & sub (graph f) (product (source f) (target f)).
```

The important property here is that, if $f$ is a triple, if we extract the source, target and graph, and construct a correspondence, we get $f$ (this is the analogous of *pair_recov* for a triple).

```
Lemma is_triple_corr: forall s t g, is_triple (corresp s t g).
Lemma corresp_source: forall s t g, source (corresp s t g) = s.
Lemma corresp_target: forall s t g, target (corresp s t g) = t.
Lemma corresp_graph: forall s t g, graph (corresp s t g) = g.
Lemma corresp_recov: forall f, is_triple f ->
  corresp(source f) (target f) (graph f) = f.
Lemma corresp_recov1: forall f, is_correspondence f ->
  corresp (source f) (target f) (graph f) = f.
```

We list here the basic properties of correspondences.

```
Lemma corr_propc: forall f,  let g := graph f in
  is_correspondence f ->
  (is_graph g & sub (domain g) (source f) & sub (range g) (target f)).

Lemma corresp_create: forall s t g,
  sub g (product s t) -> is_correspondence (corresp s t g).
Lemma corresp_is_graph: forall g,
  is_correspondence g -> is_graph (graph g).
Lemma corresp_sub_range: forall g,
  is_correspondence g -> sub (range (graph g)) (target g).
Lemma corresp_sub_domain: forall g,
  is_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 set_of_correspondences (x y:Set) :=
  product(powerset (product x y))
  (product (singleton x) (singleton y)).
Lemma set_of_correspondences_rw: forall x y z,
  inc z (set_of_correspondences x y) =
   (is_correspondence z & source z = x & target z = y).
Lemma set_of_correspondences_propa: forall f,
  is_correspondence f ->
  inc f (set_of_correspondences (source f) (target f)).
Lemma sof_value_pra:  forall x y z,
  let f:= sof_value x y z in
  inc z (set_of_correspondences x y) ->
    (is_correspondence f & source f = x &  target f = y &  f = z).
```

Given a function $f : a \to b$, we construct $\mathscr{L} f$, the associated correspondence.

```
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)  := corresp a b (gacreate f).

Lemma acreate_corresp: forall (a b:Set) (f:a->b),
  is_correspondence (acreate f).
Lemma source_acreate : forall (a b :Set)(f:a->b),  source (acreate f) = a.
Lemma target_acreate : forall (a b:Set) (f:a->b),  target (acreate f) = b.
```

¶ Direct image of a set by a functional object. 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 second definition, $f$ is a correspondence, and we consider the image by its graph. In the last definition, $f$ is a correspondence, and we take the image of the source (the case where $f$ is a mapping has been considered in Section 2.5).

```
Definition image_by_graph f u:=
  Zo (range f) (fun y=>exists x, inc x u & inc (J x y) f).

Definition image_by_fun f u :=
  image_by_graph (graph f) u.

Definition image_of_fun f :=
  image_by_graph (graph 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 image functor is increasing (we use here the term "functor" rather than function, since it is a mapping without graph).[1]

```
Lemma image_by_graph_rw: forall u r y,
  inc y (image_by_graph r u) = exists x, (inc x u & inc (J x y) r).
Lemma sub_image_by_graph: forall u r,
  sub (image_by_graph r u)  (range r).
Lemma image_by_graph_domain: forall r, is_graph r ->
  image_by_graph r (domain r) = range r.
Lemma image_by_emptyset: forall r,
  image_by_graph r emptyset =  emptyset.
Lemma image_by_nonemptyset: forall u r,
  is_graph r -> nonempty u -> sub u (domain r)
  -> nonempty (image_by_graph r u).
Theorem image_by_increasing: forall u u' r,
```

[1]In the initial version of the theorem, we assume $r$ to be a graph

```
  sub u u' -> sub (image_by_graph r u) (image_by_graph r u').
Lemma image_of_large: forall u r,  is_graph r ->
  sub (domain r) u -> image_by_graph r u = range r.
```

Given a graph $r$ and an element $x$, the set of all $y$ in $r$ whose first projection is $x$ is called the *cut*. This is $r\langle\{x\}\rangle$. 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_singleton r x := image_by_graph r (singleton x).

Lemma im_singleton_pr: forall r x y,
  inc y (im_singleton r x) = inc (J x y) r.
Lemma im_singleton_inclusion: forall r r', is_graph r -> is_graph r' ->
  (forall x, sub (im_singleton r x) (im_singleton r' x)) = sub r r'.
```

## 4.2 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 $(y, x) \in G$. We follow the definition of Bourbaki; he says that this set exists when G is a graph, because it is a subset of the product of the range and domain. We can also consider the image of the mapping $(x, y) \to (y, x)$. Both definitions agree if G is a graph.

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

Lemma inverse_graph_alt: forall r, is_graph r ->
  inverse_graph r = fun_image r (fun z => J(Q z) (P z)).
```

Some trivialities to start with.

```
Lemma inverse_graph_is_graph: forall r, is_graph (inverse_graph r).
Lemma inverse_graph_rw: forall r y, is_graph r ->
  inc y (inverse_graph r) = (is_pair y &  inc (J (Q y)(P y)) r).
Lemma inverse_graph_pair: forall r x y,
  inc (J x y) (inverse_graph r) = inc (J y x) r.
Lemma inverse_graph_pr2: forall r x y,
  related (inverse_graph r) y x = related r x y.
```

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 inverse_graph_involutive: forall r, is_graph r ->
  inverse_graph (inverse_graph r) = r.
Lemma range_inverse: forall r, is_graph r ->
  range (inverse_graph r) = domain r.
Lemma domain_inverse: forall r, is_graph r ->
  domain (inverse_graph r) = range r.
Lemma inverse_graph_emptyset:
  inverse_graph (emptyset) = emptyset.
Lemma inverse_product: forall x y,
  inverse_graph (product x y) = product y x.
Lemma inverse_identity_g: forall 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 inverse_source: forall f, source (inverse_fun f) = target f.
Lemma inverse_target: forall f, target (inverse_fun f) = source f.
Lemma inverse_correspondence: forall m,
  is_correspondence m -> is_correspondence (inverse_fun m).
Lemma graph_inverse: forall m: correspondenceC,
  graph(inverse_fun m) = inverse_graph(graph m).
Lemma inverse_fun_involutive: forall m,
  is_correspondence m -> inverse_fun (inverse_fun m) = m.
```

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 inv_image_by_graph r x :=
  image_by_graph (inverse_graph r) x.

Definition inv_image_by_fun r x:=
  inv_image_by_graph(graph r) x.

Lemma inv_image_by_fun_pr: forall r x,
  inv_image_by_fun r x = image_by_fun (inverse_fun r) x.
Lemma inv_image_graph_rw: forall x r y,
  (inc y (inv_image_by_graph r x)) = (exists u, inc u x & inc (J y u) r)).
Lemma inv_image_fun_rw: forall x r y,
  (inc y (inv_image_by_fun r x)) = (exists u, inc u x & inc (J y u) (graph r)).
```

## 4.3 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 an $y$ such that $(x, y)$ is in the first graph and $(y, z)$ is in the second. (this agrees with the previous definition in good cases). It is a subset of the product of the domain of the first graph and the range of the second. Note: the first graph is $G_1$, it is the second argument of *compose_graph*. These properties are obvious.[2]

```
Definition compose_graph r' r :=
 Zo(product(domain r)(range r'))(fun w => exists y,
  (inc (J (P w) y) r & inc (J y (Q w)) r')).

Lemma composition_is_graph: forall r r',
  is_graph (compose_graph r r').
Lemma inc_compose: forall r r' x,
  inc x (compose_graph r' r) =
```

[2]In a previous version, we assumed in some lemmas that $r$ or $r'$ are graphs

```
  (is_pair x &( exists y, inc (J (P x) y) r& inc (J y (Q x)) r')).
Lemma compose_related:forall r r' x z,
  (related (compose_graph r' r) x z=
  exists y, related r x y & related r' y z).
Lemma compose_domain1: forall r r',
  sub (domain (compose_graph r' r)) (domain r).
Lemma compose_range1:forall r r',
  sub (range (compose_graph r' r)) (range r').
```

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

```
Theorem inverse_compose:forall r r',
  inverse_graph (compose_graph r' r) =
  compose_graph (inverse_graph r)(inverse_graph r').
```

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

```
Theorem composition_associative:forall r r' r'',
  is_graph r -> is_graph r' -> is_graph r'' ->
  compose_graph r'' (compose_graph r' r) =
  compose_graph (compose_graph r'' r') r.
```

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 image_composition: forall r r' x,
  image_by_graph(compose_graph r' r) x = image_by_graph r' (image_by_graph r x).

Lemma compose_domain:forall r r',
  is_graph r' ->
  domain (compose_graph r' r) = inv_image_by_graph r (domain r').

Lemma compose_range: forall r r',
  is_graph r ->
  range (compose_graph r' r) =   image_by_graph r' (range r).

Lemma inverse_direct_image: forall r x,
  is_graph r -> sub x (domain r) ->
  sub x (inv_image_by_graph r (image_by_graph r x)).

Lemma composition_increasing: forall r r' s s',
  sub r s -> sub r' s' -> sub (compose_graph r' r) (compose_graph s' s).
```

The property that $f$ and $f'$ are two correspondences where the target of $f$ is the source of $f'$ will be called *composableC*. We can write them as $f = (G, A, B)$ and $f' = (G', B, C)$. 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; two lemmas are needed; the first one says $f^{-1} \circ f'^{-1}$ is defined, the other one says that it is the LHS.

```
Definition composableC r' r :=
  is_correspondence r & is_correspondence r' & source r' = target r.
Definition compose r' r :=
  corresp (source r)(target r') (compose_graph (graph r')(graph r)).
Lemma compose_correspondence: forall r' r,
  is_correspondence r -> is_correspondence r' ->
  is_correspondence (compose r' r).
Lemma compose_of_sets: forall r' r x,
  image_by_fun(compose r' r) x = image_by_fun r' (image_by_fun r x).
Lemma inverse_compose_cor: forall r r',
  inverse_fun (compose r' r) =  compose (inverse_fun r)(inverse_fun r').
```

Denote by $\Delta_A$ the diagonal of A and by $I_A$ the identity correspondence defined by $(\Delta_A, A, A)$.

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

Lemma identity_corresp: forall x,
  is_correspondence (identity_fun x).
```

If $f$ is a correspondence between A and B then $f \circ I_A$ and $I_B \circ f$ are equal to $f$. In particular $I_A \circ I_A = I_A$.

```
Lemma identity_source: forall x, source (identity x) = x.
Lemma identity_target: forall x, target (identity x) = x.
Lemma compose_identity_left: forall m,
   is_correspondence m -> compose (identity (target m)) m = m.
Lemma compose_identity_right: forall m,
  is_correspondence m -> compose m (identity (source m)) = m.
Lemma compose_identity_identity: forall x,
  compose (identity x) (identity x) =  (identity x).
Lemma identity_self_inverse: forall x,
  inverse_fun (identity x) =  (identity x).
```

## 4.4 Functions

We say that $r$ 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 3.1, that says that if $z$ and $z'$ are in $r$, then $\text{pr}_1 z = \text{pr}_1 z'$ implies $z = z'$. Remember that $\mathcal{V}_r x$ denotes the object $v$ (if it exists) such that $x$ is related to $v$.

```
Definition functional_graph r :=
  forall x y y', related r x y -> related r x y' ->  y=y'.

Lemma is_functional: forall r,
  (is_graph r & functional_graph r) = (fgraph r).
```

A *function* is a correspondence $f = (G, A, B)$ 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 $G(x)$. Here we use either $\mathcal{V}_G x$ or $\mathcal{W}_f x$. Note: since *source* is only defined for correspondences, by type inference the expression *is_function f* implies that $f$ is a correspondence. Note: Bourbaki says [2, p. 82] “we shall often use the word ‘function’ in place of ‘functional graph’ ”.

```
Definition is_function f :=
    is_correspondence f & fgraph (graph f) & source f = domain (graph f).

Lemma function_fgraph: forall f ,
  is_function f -> fgraph (graph f).
Lemma function_graph: forall f,
  is_function f -> is_graph (graph f).
Lemma is_function_pr: forall s t g,
  fgraph g -> sub (range g) t -> s = domain g ->
  is_function (corresp s t g).
Lemma f_domain_graph: forall f, is_function f -> domain (graph f) = source f.

Lemma f_range_graph: forall f, is_function f -> sub (range (graph f))(target f).
Lemma image_by_fun_source: forall f, is_function f ->
  image_by_fun f (source f) =  range (graph f).
Lemma related_inc_source: forall f x y,
  is_function f -> related (graph f) x y -> inc x (source f).

Lemma is_function_functional: forall f, is_correspondence f ->
  is_function f = (forall x, inc x (source f) ->
    exists_unique (fun y => related (graph f) x y)).
```

All properties of $\mathscr{V}$ give a corresponding one for $\mathscr{W}$. All lemmas listed here are trivial. Let $f = (\mathrm{G},\mathrm{A},\mathrm{B})$ be a function. If $x \in \mathrm{A}$ then $(x, \mathscr{W}_f x) \in \mathrm{G}$, $\mathscr{W}_f x \in \mathrm{range(G)}$ and $\mathscr{W}_f x \in \mathrm{B}$. If $y \in \mathrm{range(G)}$, there exists $x$ such that $y = \mathscr{W}_f x$. If $z \in \mathrm{G}$ then $z = (\mathrm{pr}_1 z, \mathscr{W}_f \mathrm{pr}_1 z)$, $\mathrm{pr}_2 z = \mathscr{W}_f \mathrm{pr}_1 z$, and $\mathrm{pr}_1 z \in \mathrm{A}$. If $(x, y) \in \mathrm{G}$ then $y = \mathscr{W}_f x$, $x \in \mathrm{A}$ 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 $y = \mathscr{W}_f x$.

```
Definition W x f := V x (graph f).

Lemma W_pr3: forall f x,
  is_function f-> inc x (source f) -> inc (J x (W x f)) (graph f).
Lemma inc_W_range_graph:forall f x, is_function f -> inc x (source f)
  -> inc (W x f) (range (graph f)).
Lemma inc_W_target: forall f x, is_function f -> inc x (source f)
  -> inc (W x f) (target f).
Lemma range_inc_rw: forall f y, is_function f ->
  inc y (range (graph f)) = exists x:E, (inc x (source f) & y = W x f).
Lemma in_graph_W: forall f x,
  is_function f -> inc x (graph f) -> x = (J (P x) (W (P x) f)).
Lemma W_pr2: forall f x, is_function f ->
  inc x (graph f) -> Q x = W (P x) f.
Lemma inc_pr1graph_source1:forall f x, is_function f ->
  inc x (graph f) ->  inc (P x) (source f).
Lemma W_pr: forall f x y, is_function f ->
  inc (J x y) (graph f) -> W x f = y.
Lemma inc_pr2graph_target: forall f x y, is_function f ->
  inc (J x y) (graph f) ->  inc y (target f).
Lemma inc_pr2graph_target1:forall f x, is_function f ->
  inc x (graph f) ->  inc (Q x) (target f).
Lemma inc_pr1graph_source1:forall f x, is_function f ->
  inc x (graph f) ->  inc (P x) (source f).
Lemma W_image: forall f x y, is_function f -> sub x (source f) ->
  (inc y (image_by_fun f x) = exists u, inc u x & W u f = y).
Lemma image_of_fun_pr: forall f, image_of_fun f = image_by_fun f (source f).
Lemma sub_image_target:
  forall f, is_function f -> sub (image_of_fun f) (target f).
```

```
Lemma sub_image_target1: forall g x, is_function g ->
  sub (image_by_fun g x) (target g).
```

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

```
Lemma function_exten3: forall f g,
  is_function f -> is_function g -> graph f = graph g ->
  target f = target g -> source f = source g ->
  f = g.
Lemma function_exten: forall f g,
  is_function f -> is_function g -> source f = source g ->
  target f = target g -> (forall x, inc x (source f) -> W x f = W x g)
  -> f = g.
Lemma function_exten1: forall f g,
  is_function f -> is_function g -> graph f = graph g ->
  target f = target 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 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 image_singleton: forall f x,
  is_function f -> inc x (source f) ->
  image_by_fun f (singleton x) = singleton (W x f).


Lemma inv_image_complement: forall g x,
  is_function g ->
  inv_image_by_fun g (complement (target g) x) =
  complement (source g) (inv_image_by_fun g 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, we can consider the function $\mathscr{L}_{\mathrm{A;B}} h$ from A to B whose graph is $\mathscr{L}_{\mathrm{A}} h$, provided that $x \in \mathrm{A}$ implies $h(x) \in \mathrm{B}$ (in the example, B must contain the even integers); this condition will be denoted by *transf_axiom* (see Section 4.6). 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 $\mathscr{L}_{\mathrm{A;B}} h$ will be denoted by $\mathscr{L} f$. In the Coq source, this is *acreate*. We shall see in a moment that $f$ can be obtained from $g = \mathscr{L} f$ by the formula $f = \mathscr{M}_{\mathrm{A;B}} g$. Lemma *W_acreate* says that the following diagram (left part) commutes.

$$\begin{tikzcd} \mathrm{A} \arrow[r, "f=\mathscr{M}_{\mathrm{A;B}}g"] \arrow[d, "\mathscr{R}"'] & \mathrm{B} \arrow[d, "\mathscr{R}"] \\ \mathrm{E} \arrow[r, "\mathrm{W}\cdot\mathscr{L}f"] & \mathrm{E} \end{tikzcd} \qquad \begin{tikzcd} \mathrm{A} \arrow[r, "\mathscr{M}_{\mathrm{A;B}}g"] & \mathrm{B} \\ \mathrm{R\_inc} \arrow[u, "\mathscr{B}"] \arrow[r, "g=\mathscr{L}f"] & \mathrm{W\_mapping} \arrow[u, "\mathscr{B}"'] \end{tikzcd} \qquad \text{(a/b create)}$$

```
Lemma prop_acreate: forall (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 : forall (A B:Set) (f:A->B),  is_function(acreate f).
Lemma acreate_W : forall (A B:Set) (f:A->B) (x:A),
  W (Ro x) (acreate f) = 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 $\mathrm{A} \to \mathrm{B}$, and if $g = \mathscr{L}h$, then $\mathscr{M}g$ is a function $\mathrm{A}' \to \mathrm{B}'$, where $\mathrm{A}'$ 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 $x \in$ source($f$). The expression *inc_W_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 $\mathscr{R}$ 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:is_function f) :=
  fun x:source f => Bo (inc_W_target H (R_inc x)).

Lemma prop_bcreate1: forall f (H:is_function f) (x:source f),
  Ro(bcreate1 H x) = W (Ro x) f.
Lemma bcreate_inv1: forall f (H:is_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 *W_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 $\mathscr{M}_{\mathrm{A;B}}\mathscr{L}f = f$. Note: the proof of the lemma is very short. It uses the *arrow_extensionality* axiom, so that we show in fact $f'(a) = f(a)$ for all $a$ of type A. It uses the *R_inj* axiom, so that we prove $\mathscr{R}f'(a) = \mathscr{R}f(a)$. Using *prop_back2* the lhs is the value on $\mathscr{R}a$ of $\mathscr{L}f$, which is the rhs thanks to *W_acreate*. The last lemma says that $\mathscr{M}f = \mathscr{M}_{\mathrm{A;B}}f$ for some obvious A and B.

```
Lemma W_mapping: forall f A B (Ha:source f =A)(Hb:target f =B) x,
  is_function f -> inc x A -> inc (W x f) B.
Lemma acreate_source: forall (A B:Set) (f:A->B), source (acreate f)= A.
Lemma acreate_target: forall (A B:Set) (f:A->B), target (acreate f)= B.

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

Lemma prop_bcreate2: forall f A B
  (H:is_function f) (Ha:source f =A)(Hb:target f=B)(x:A),
  Ro(bcreate H Ha Hb x) = W (Ro x) f.

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


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

```
Lemma bcreate_eq: forall f (H:is_function f),
  bcreate1 H = bcreate H (refl_equal (source f)) (refl_equal (target f)).
```

Let's consider some examples of functions. The empty function is the only function from the empty set to itself; its graph is empty. Note that if the graph is empty, so is the source.[3]

```
Definition empty_functionC : emptyset -> emptyset := fun x => x.
Definition empty_function:= acreate empty_functionC.

Lemma empty_function_function: is_function empty_function.
Lemma empty_function_graph: graph empty_function = emptyset.
Lemma special_empty_function: forall f, is_function f ->
  graph f = emptyset -> source f = emptyset.

Lemma empty_function_prop:
  bcreate empty_function_function (acreate_source empty_functionC)
  (acreate_target empty_functionC)
  = empty_functionC.
```

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

```
Lemma identity_function: forall x,
  is_function (identity x).
Lemma identity_W: forall x y,
  inc y x -> W y (identity x) = y.
```

We define *identityC a* to be the identity on $a$ as a Coq function. By default, the argument $a$ is implicit; we make it explicit.

```
Definition identityC (a:Set): a->a := fun x => x.
Implicit Arguments identityC [].

Lemma w_identity: forall a x,  identityC a x = x.
Lemma identity_prop: forall a, acreate (identityC a) = identity a.
Lemma identity_prop2: forall a,
  bcreate (identity_function a) (identity_source a) (identity_target a) =
  identityC a.
```

¶ If $a \in x$ and $b \in x$ imply $a = b$, we say that $x$ is a small set. It is either empty or has a single element. We say that a function is *constant* if it takes at most one value; in other words that the range is a small set (if the source is not empty, the range is nonempty).

The constant function $C_{xy}a$ maps $b$ of type $x$ to $a$ of type $y$; for this reason, arguments $x$ and $y$ are implicit. The Bourbaki function $\Gamma = (x \times \{a\}, x, y)$ needs the assumption $a \in y$. Hence $a$ and $y$ are implicit. We make all parameters explicit.

```
Definition small_set x :=
  forall u v, inc u x -> inc v x -> u = v.

Definition is_constant_function f :=
  (is_function f ) &
```

[3]Some trivial results, such as: "the source of the empty function is the empty set", have been removed in V3 since they are consequence of the *simpl* tactic. These lemmas can be found in the last chapter. In version 4, this cannot be applied any more, since the source of a function is now $\mathrm{pr}_1 \circ \mathrm{pr}_2$

```
  (forall x x', inc x (source f) -> inc x' (source f) -> W x f = W x' f).

Definition constant_functionC x y (a:y)  :=  fun _:x => a.
Implicit Arguments constant_functionC [].
Definition constant_function (x y a:Set) (H:inc a y) :=
  acreate (constant_functionC x y (Bo H)).
Implicit Arguments constant_function [].
```

These are the basic properties of constant functions.

```
Lemma constant_source: forall x y a (H:inc a y),
  source (constant_function x y a H) = x.
Lemma constant_target: forall x y a  (H:inc a y),
  target (constant_function x y a H) = y.
Lemma constant_graph: forall x y a  (H:inc a y),
  graph (constant_function x y a H) = product x (singleton a).
Lemma constant_function_fun: forall x y a (H: inc a y),
  is_function(constant_function x y a H).
Lemma constant_W: forall x y a (H:inc a y) b,
  inc b x ->  W b (constant_function x y a H) = a.
Lemma w_constant_functionC: forall  x y (a:y)  (z:x),
  constant_functionC x y a z = a.
```

We give now the link between constant functions, and the property of being constant. Every constant function is of the form $C_{xy}a$ for some $a$ if $x$ is not empty. In the case of a Bourbaki function, instead of saying "there exists $a \in y$" we say "there exists $a$ of type $y$" from which we deduce an element and the proof that it is in $y$.

```
Lemma constant_function_pr: forall f,
  is_function  f -> (is_constant_function f =
    small_set (range (graph f))).
Lemma constant_constant_fun: forall x y a (H: inc a y),
 is_constant_function(constant_function x y a H).
Lemma constant_fun_constantC: forall  x y a,
   is_constant_functionC (constant_functionC x y a).
Lemma constant_function_prop2: forall (x y:Set) (a:y),
  bcreate (constant_function_fun x (R_inc a)) (constant_source x (R_inc a))
  (constant_target x (R_inc a)) = constant_functionC x y a.
Lemma constant_fun_prC: forall x y (f:x->y)(b:x), is_constant_functionC f ->
  exists a:y, f= constant_functionC x y a.
Lemma nonempty_target: forall f,
  nonempty(graph f) -> inc (rep (target f)) (target f).
Lemma constant_fun_pr: forall f (H:nonempty(graph f)),
  is_constant_function f ->
  exists a: target f,
  f= constant_function (source f) (target f) (Ro a) (R_inc a).
```

## 4.5 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 4.7.

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 $f(u)$.

```
Definition composeC a b c (g:b->c) (f: a->b):= fun x:a => g (f x).
Lemma compositionC_associative: forall a b c d (f: c->d)(g:b->c)(h:a->b),
  composeC (composeC f g) h = composeC f (composeC g h).
Lemma compose_id_leftC: forall (a b:Set) (f:a->b),
  composeC (identityC b) f = f.
Lemma compose_id_rightC: forall (a b:Set) (f:a->b),
  composeC f (identityC a) = f.
```

$$\begin{tikzcd}
x \arrow[r, "\mathrm{I}_{x;y}"] \arrow[d, "\mathscr{R}"] & y \\
\text{R\_inc} \arrow[r, "\subset"] & \text{H\_sub} \arrow[u, "\mathscr{B}"']
\end{tikzcd}$$
(inclusion)

We now define the inclusion $\mathrm{I}_{xy}$. 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 $\mathrm{I}_{xz}$).

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

Lemma inclusionC_pr: forall x y (H: sub x y) (a:x),
  Ro(inclusionC H a) = Ro a.
Lemma inclusionC_identity: forall x,
  inclusionC (sub_refl (x:=x)) = identityC x.
Lemma inclusionC_compose: forall x y z (Ha:sub x y)(Hb: sub y z),
  composeC (inclusionC Hb)(inclusionC Ha) = 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 $(\mathrm{G}, \mathrm{A}, \mathrm{B})$ and $(\mathrm{G}', \mathrm{A}', \mathrm{B}')$. If $\mathrm{A} = \mathrm{A}'$ and $\mathrm{G} = \mathrm{G}'$, the functions agree on A. Conversely, the property "$\mathrm{A} \subset \mathrm{A}'$ and the functions agree on A" is the same as $\mathrm{G} \subset \mathrm{G}'$. Thus if $\mathrm{A} = \mathrm{A}'$ we have $\mathrm{G} = \mathrm{G}'$. If moreover $\mathrm{B} = \mathrm{B}'$, the functions are the same.

$$\begin{tikzcd}
 & \mathrm{A} \arrow[r, "f"] & \mathrm{B} \arrow[dr, "\mathscr{R}"] & \\
\mathrm{X} \arrow[ur, "\mathrm{I}_{\mathrm{XA}}"] \arrow[dr, "\mathrm{I}_{\mathrm{XA}'}"'] & & & \mathrm{E} \\
 & \mathrm{A}' \arrow[r, "f'"] & \mathrm{B}' \arrow[ur, "\mathscr{R}"'] &
\end{tikzcd}$$
(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')) &
  (forall a, inc a x -> W a f = W a f').

Definition restrictionC (x a b:Set) (f:a->b)(H: sub x a) :=
  composeC f (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: forall f f',
  is_function f -> is_function f' -> graph f = graph f' ->
  agrees_on (source f) f f'.

Lemma function_exten2: forall f f',
  is_function f -> is_function f' ->
  (f =f')= ((source f =source f') & (target f= target f') &
    (agrees_on (source f) f f')).

Lemma sub_function: forall f g,
  is_function f -> is_function g ->
  (sub (graph f) (graph g)) = (agrees_on (source f) f g).
```

¶ We consider here the restriction of a function to a subset of its domain. We need some lemmas in order to define it. We have two possibilities: the target of the new function can be either the target of the old function, or the range.

```
Definition restriction f x :=
  corresp x (target f) (restr (graph f) x).
Definition restriction1 f x :=
  corresp x (image_by_fun f x) (restr (graph f) x).

Lemma restr_domain2: forall f x,
  is_function f-> sub x (source f) ->
  domain (restr (graph f) x) = x.
Lemma restr_range: forall f x,
  is_function f-> sub x (source f) ->
  sub (range (restr (graph f) x)) (target f).

Lemma restriction_function: forall f x,
  is_function f -> sub x (source f) ->
  is_function (restriction f x).

Lemma restriction1_function: forall f x,
  is_function f -> sub x (source f) ->
  is_function (restriction1 f x).
Lemma restriction_W: forall f x a,
  is_function f -> sub x (source f) ->
  inc a x ->  W a (restriction f x) = W a f.
Lemma restriction1_W: forall f x a,
  is_function f -> sub x (source f) ->
  inc a x ->  W a (restriction1 f x) = W a f.
```

We say that $g = (\mathrm{G},\mathrm{C},\mathrm{D})$ *extends* $f = (\mathrm{F},\mathrm{A},\mathrm{B})$ if $\mathrm{F} \subset \mathrm{G}$ and $\mathrm{B} \subset \mathrm{D}$. This implies $\mathrm{A} \subset \mathrm{C}$. 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 :=
  (is_function f) & (is_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 source_extends: forall f g,
  extends g f -> sub (source f) (source g).

Lemma W_extends: forall f g x,
  extends g f -> inc x (source f) -> W x f = W x g.

Lemma extendsC_pr : forall  (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: forall f x,
  is_function f -> sub x (source f) ->
  extends f (restriction f x).
Lemma function_extends_restC: forall (x a b:Set) (f:a->b)(H:sub x a),
  extendsC f (restrictionC f H) H.
Lemma agrees_same1: forall f g x, agrees_on x f g -> sub x (source f).
Lemma agrees_same2: forall f g x, agrees_on x f g -> sub x (source g).
Lemma agrees_same_restriction: forall f g x,
  is_function f -> is_function g -> agrees_on x f g ->
  target f = target g ->
  restriction f x = restriction g x.
Lemma agrees_same_restrictionC: forall (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 = restrictionC g Hb.
Lemma restriction_graph1: forall f x,
  is_function f -> sub x (source f) ->
  graph (restriction_function f x) =
  intersection2 (graph f)(product x (target f)).
Lemma restriction_recobvers: forall f x,
  is_function f -> sub x (source f) ->
  restriction_function f x = create x (target f)
  (intersection2 (graph f)(product x (target f))).
```

¶ Given a function $f = (G, A, B)$ and two sets X and Y, we can consider $(G \cap (X \times B), X, Y)$. This is a function if $X \subset A$, $Y \subset B$ and the image of X by $f$ is a subset of Y (a name is given to this condition). The function 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.

```
Definition restriction2 f x y :=
  corresp x y (intersection2 (graph f) (product x (target f))).
```

```
Definition restriction2_axioms f x y :=
  is_function f &
  sub x (source f) & sub y (target f) & sub (image_by_fun f x) y.

Lemma restriction2_graph: forall f x y,
  sub (graph (restriction2 f x y)) (graph f).
Lemma inc_graph_restriction2: forall f x y a b,
  (inc (J a b)  (graph (restriction2 f x y))) =
  (inc (J a b) (graph f) & inc a x & inc b (target f)).
Lemma restriction2_props: forall  f x y,
  restriction2_axioms f x y ->
  domain (intersection2 (graph f) (product x (target f))) = x.
Lemma restriction2_function: forall  f x y,
  restriction2_axioms f x y ->
  is_function (restriction2 f x y).    (* 19 *)
Lemma restriction2_W: forall f x y a,
  restriction2_axioms f x y ->
  inc a x -> W a  (restriction2 f x y) = W a f.
Lemma function_rest_of_prolongation: forall f g,
  extends g f -> f = restriction2 g (source f) (target f).
```

$$\begin{tikzcd} a \arrow[r, "f"] \arrow[d, "\mathrm{I}_{ac}"'] & b \arrow[d, "\mathrm{I}_{bd}"] \\ c \arrow[r, "\mathrm{R}_{cd}f"'] & d \end{tikzcd}$$ (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: forall (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) = W (Ro x) (acreate f).
Lemma restriction2C_pr1: forall (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'),
  composeC f (inclusionC Ha) = composeC (inclusionC Hb) (restriction2C f Ha H).
```

## 4.6 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 $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{T}$ nor $\boldsymbol{F}$). On the other hand, $\boldsymbol{x}$ may appear in $\boldsymbol{T}$, it does not appear in $\boldsymbol{A}$, $\boldsymbol{B}$, nor $\boldsymbol{F}$.

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 3.1). The function $(\mathrm{F}, \mathrm{A}, \mathrm{C})$ 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 $\mathscr{L}_{\mathrm{A;C}}$. In version 4, the prefix *af_* has been replaced by *bl_*.

```
Definition BL f a b :=
  corresp a b (L a f).

Lemma bl_source : forall f a b, source (BL f a b) = a.
Lemma bl_target : forall f a b, target (BL f a b) = b.
Lemma bl_graph1: forall f a b c,
  inc c (graph (BL f a b)) -> c = J (P c) (f (P c)).
Lemma bl_graph2: forall f a b c,
  inc c a -> inc (J c (f c)) (graph (BL f a b)).
Lemma bl_graph3: forall f a b c,
  inc c (graph (BL f a b)) -> inc (P c) a.
Lemma bl_graph4: forall f a b c,
  inc c (graph (BL 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 transf_axioms f a b :=
  forall c, inc c a -> inc (f c) b.

Lemma bl_function: forall f a b,
  transf_axioms f a b -> is_function (BL f a b).

Lemma bl_W: forall f a b c,
  transf_axioms f a b ->
  inc c a-> W c (BL f a b) = f c.
Lemma bl_recovers: forall f,
  is_function f -> BL (fun z => W z f)(source f)(target f) = f.
```

We consider here an example of a function defined by a term, the first and second projection, denoted $\mathrm{pr}_1$ and $\mathrm{pr}_2$, on the range and target.[4]

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

Lemma first_proj_W: forall g x,
  inc x g -> W x (first_proj g) = P x.
Lemma second_proj_W: forall g x,
  inc x g -> W x (second_proj g) = Q x.
Lemma first_proj_function:forall g,  is_function (first_proj g).
Lemma second_proj_function :forall g,   is_function (second_proj g).
```

[4]In Version 3, we assumed $g$ to be a graph, this is not needed

## 4.7 Composition of two functions. Inverse function

$$\begin{tikzcd}
A \arrow[r, "\mathscr{L}f"'] \arrow[rr, bend left, "\mathscr{L}(g\circ f)"] \arrow[d, "\mathscr{R}"'] & B \arrow[r, "\mathscr{L}g"'] \arrow[d, "\mathscr{R}"] & C \arrow[d, "\mathscr{R}"] \\
E \arrow[r, "f"] \arrow[rr, bend right, "g\circ f"'] & E \arrow[r, "g"] & E
\end{tikzcd} \tag{composition}$$

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 *fcomposable*. 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 :=
  is_function g & is_function f &  source g = target f.

Lemma composable_pr: forall f g, composable g f ->
  fcomposable (graph g) (graph f).
Lemma composable_pr1: forall f g, composable g f ->
  graph (compose g f) = fcompose (graph g) (graph f).
Lemma compose_domain:forall g f,
  composable g f->
  domain (graph(compose g f)) = domain (graph f).

Lemma compose_source: forall f g, source (compose g f) = source f.
Lemma compose_target: forall f g, target (compose g f) = target g.

Theorem compose_function: forall g f, composable g f
  -> is_function (compose g f).

Lemma compose_W: forall g f x,  composable g f ->
  inc x (source f) -> W x (compose g f) = W (W x f) g.

Lemma composable_acreate:forall (a b c:Set) (f: a-> b)(g: b->c),
  composable (acreate g) (acreate f).
Lemma compose_acreate:forall (a b c:Set) (g: b->c)(f: a-> b),
  compose (acreate g) (acreate f) = acreate(composeC g f).
```

Composition is associative, and identity is a unit.

```
Lemma compose_assoc: forall f g h,
  composable f g -> composable g h ->
  compose (compose f g) h = compose f (compose g h).
Lemma compose_id_left: forall m,
  is_function m-> compose (identity (target m)) m = m.
Lemma compose_id_right: forall m,
  is_function m-> compose m  (identity (source m)) = m.
```

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 injective f:=
  is_function f &
  (forall (x y), inc x (source f) -> inc y (source f) ->
    W x f = W y f -> x = y).

Definition surjective f :=
  is_function f & image_of_fun f = target f.

Definition bijective f :=
  injective f & surjective f.

Definition equipotent x y :=
  exists z, bijective z & source z = x & target z = y.


Lemma inj_is_function: forall f, injective f -> is_function f.
Lemma surj_is_function: forall f, surjective f-> is_function f.
Lemma bij_is_function: forall f, bijective f-> is_function f.


Lemma injective_pr: forall f x x' y,
  injective f -> related (graph f) x y -> related (graph f) x' y -> x = x'.
Lemma injective_pr3: forall f x x' y,
  injective f -> inc (J x y) (graph f) -> inc (J x' y) (graph f) -> x = x'.
Lemma injective_pr_bis: forall f,
  is_function f ->
  (forall x x' y, related (graph f) x y -> related (graph f) x' y -> x = x')
  -> injective f.
Lemma surjective_pr: forall f y,
  surjective f -> inc y (target f) ->
  exists x, inc x (source f) & related (graph f) x y .
Lemma surjective_pr2: forall f y,
  surjective f -> inc y (target f) -> exists x, inc x (source f) & W x f = y.
Lemma surjective_pr3: forall f,
  surjective f -> range (graph f) = target f.
Lemma surjective_pr4: forall f,
  is_function f-> range (graph f) = target f -> surjective f.
Lemma surjective_pr5: forall f,
  is_function f -> (forall y, inc y (target f) ->
  exists x, inc x (source f) & related (graph f) x y) -> surjective f.
Lemma surjective_pr6: forall f,
  is_function f-> (forall y, inc y (target f) ->
  exists x, inc x (source f) & W x f = y) -> surjective f.
Lemma bl_injective:  forall f a b, transf_axioms f a b ->
  (forall u v, inc u a-> inc v a -> f u = f v -> u = v) ->
  injective (BL f a b).
Lemma bl_surjective: forall f a b, transf_axioms f a b ->
  (forall y, inc y b -> exists x, inc x a & f x = y) ->
  surjective (BL f a b).
Lemma bl_bijective: forall f a b, transf_axioms f a b ->
  (forall u v, inc u a-> inc v a -> f u = f v -> u = v) ->
  (forall y, inc y b -> exists x, inc x a & f x = y) ->
  bijective (BL f a b).
Lemma bijective_pr: forall f y,
  bijective f -> inc y (target f) ->
  exists_unique (fun x=> inc x (source f) & W x f = y).

Lemma function_exten4: forall f g, source f = source g ->
  surjective f -> surjective g ->
  (forall x, inc x (source f) -> W x f = W x g) -> f = g.
```

Let's consider the case of Coq functions. We do not need as many lemmas, because they are trivialities. Functors *acreate* and *bcreate* map bijections to bijections. We say that two sets are *equipotent* if there is a bijection between them. Which definition of bijection used is irrelevant.

```
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.


Lemma bijectiveC_pr: forall (a b:Set) (f:a->b) (y:b),
  bijectiveC f ->   exists_unique (fun x:a=> f x = y).

Lemma bcreate_injective: forall f a b
  (H:is_function f)(Ha:source f =a)(Hb:target f =b),
  injective f -> injectiveC (bcreate H Ha Hb).
Lemma bcreate_surjective: forall f a b
  (H:is_function f)(Ha:source f =a)(Hb:target f =b),
  surjective f -> surjectiveC (bcreate H Ha Hb).
Lemma bcreate_bijective: forall f a b
  (H:is_function f)(Ha:source f =a)(Hb:target f =b),
  bijective f -> bijectiveC (bcreate H Ha Hb).

Lemma bcreate1_injective: forall f (H:is_function f),
  injective f -> injectiveC (bcreate1 H).
Lemma bcreate1_surjective: forall f (H:is_function f),
  surjective f -> surjectiveC (bcreate1 H).
Lemma bcreate1_bijective: forall f (H:is_function f),
  bijective f -> bijectiveC (bcreate1 H).

Lemma acreate_injective: forall (a b:Set) (f:a->b),
  injectiveC f -> injective (acreate f).
Lemma acreate_surjective: forall (a b:Set) (f:a->b),
  surjectiveC f -> surjective (acreate f).
Lemma acreate_bijective: forall (a b:Set) (f:a->b),
  bijectiveC f -> bijective (acreate f).
Lemma equipotentC: forall x y, equipotent x y = exists f:x->y, bijectiveC f.
```

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 $\mathrm{Y} = f(\mathrm{X})$.

```
Lemma identity_bijective: forall x,
  bijective (identity x).
Lemma identityC_bijective: forall x,
  bijectiveC (identityC x).
Lemma restriction2_injective: forall f x y,
  injective f -> restriction2_axioms f x y
  -> injective (restriction2 f x y).
Lemma restriction2_surjective: forall f x y,
  restriction2_axioms f x y ->
  source f = x -> image_of_fun f = y ->
  surjective (restriction2 f x y).

Lemma restriction1_surjective: forall f x,
  is_function f -> sub x (source f) ->
  surjective (restriction1 f x).
Lemma restriction1_bijective: forall f x,
  injective f -> sub x (source f) ->
  bijective (restriction1 f x).
```

¶ 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 tack_on_f f x y:=
  corresp (tack_on (source f) x)
  (tack_on (target f) y)  (tack_on (graph f) (J x y)).

Lemma tack_on_corresp: forall  f x y,
  is_correspondence f ->  is_correspondence (tack_on_f f x y).
Lemma tack_on_function: forall  f x y,
  is_function f -> ~(inc x (source f)) -> is_function (tack_on_f f x y).
Lemma tack_on_W_in: forall  f x y u,
  is_function f -> ~(inc x (source f)) -> inc u (source f) ->
  W u (tack_on_f f x y) = W u f.
Lemma tack_on_W_out: forall  f x y,
  is_function f -> ~(inc x (source f)) ->  W x (tack_on_f f x y) = y.


Lemma tack_on_V_out: forall  f x y,
  fgraph f -> ~ (inc x (domain f)) ->
  V x (tack_on f (J x y)) = y.
Lemma tack_on_V_in: forall  f x y u,
  fgraph f -> ~ (inc x (domain f)) -> inc u (domain f) ->
  V u (tack_on f (J x y)) = V u f.

Lemma tack_on_surjective: forall  f x y,
  surjective f -> ~(inc x (source f)) -> surjective (tack_on_f f x y).
Lemma tack_on_f_injective: forall x f g a b,
  is_function f -> is_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.

Lemma tack_on_g_injective: forall x f g a b,
  fgraph f -> fgraph g ->
  domain f = domain g -> ~ (inc x (domain f)) ->
  (tack_on f (J x a) = tack_on g (J x b)) -> f = g.
Lemma restr_tack_on: forall f x a,
  fgraph f -> ~ (inc x (domain f)) ->
  restr (tack_on f (J x a)) (domain f) = f.
Lemma tack_on_restr: forall f x E, fgraph f -> ~ (inc x E) ->
   domain f = tack_on E x->
   tack_on (restr f E) (J x (V x f)) = f.
```

¶ 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 52.

```
Definition canonical_injection a b :=
  corresp a b (identity_g a).
Lemma ci_function: forall a b, sub a b ->
  is_function  (canonical_injection a b).
Lemma ci_W: forall a b x,
  sub a b -> inc x a -> W x (canonical_injection a b) = x.
Lemma ci_injective: forall a b,
  sub a b -> injective (canonical_injection a b).
Lemma ci_range: forall a b, sub a b ->
  range (graph (canonical_injection a b)) = a.
```

¶ 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 :=
  BL (fun x=> J x x) a (product a a).
Lemma graph_diag_app: forall a x,
  inc x  (graph (diagonal_application a)) =
  (is_pair x & inc (P x) a & Q x = J (P x) (P x)).
Lemma diag_app_function: forall a, is_function (diagonal_application a).
Lemma diag_app_injective: forall a,  injective (diagonal_application a).
Lemma diag_app_W:forall a x,
  inc x a -> W x (diagonal_application a) = J x x.
Lemma diag_app_range: forall a,
  range (graph (diagonal_application a)) = diagonal a.
```

¶ Both projections $\mathrm{pr}_1$ and $\mathrm{pr}_2$ are surjective by construction (note that $g$ is not required to be a graph).

The first projection on G is injective if only if G is a functional graph.

```
Lemma second_proj_surjective: forall g,
  surjective (second_proj g).
Lemma first_proj_surjective: forall g ,
  surjective (first_proj g).
Lemma injective_first_proj: forall g,
  is_graph g -> (injective (first_proj g) = functional_graph g).
```

¶ If G is a graph, the map $(x, y) \mapsto (y, x)$ maps G onto $G^{-1}$ (as noted when we defined the inverse graph). From this, we get a bijective function. Bourbaki considers the following function: we fix $a$, and map $x$ to $(x, a)$. This is a bijection between X and the product $X \times \{b\}$. This could be restated as: X is equipotent to $X \times Y$ when Y is a singleton.

```
Definition inv_graph_canon g :=
   BL (fun x=> J (Q x) (P x)) g (inverse_graph g).
```

```
Lemma inv_graph_canon_W: forall g x,
  is_graph g ->inc x g -> W x (inv_graph_canon g) = J (Q x) (P x).
Lemma inv_graph_canon_function:  forall g,
  is_graph g -> is_function (inv_graph_canon g).
Lemma inv_graph_canon_bijective:  forall g,
  is_graph g -> bijective (inv_graph_canon g).

Lemma bourbaki_ex5_17: forall a b,
  bijective ( BL (fun x=> J x b) a (product a (singleton b))).

Lemma equipotent_aux: forall f a b,
  bijective (BL f a b) -> equipotent a b.
Lemma equipotent_prod_singleton:
  forall a b, equipotent a (product a (singleton b)).
Lemma restriction1_pr: forall f,
  is_function f -> restriction2 f (source f) (image_by_fun f (source f)) =
  restriction1 f (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_function: forall f,
  bijective f -> is_function (inverse_fun f).
Theorem inv_function_bijective: forall f,
  is_function f -> is_function (inverse_fun f) -> bijective f.
```

In the case of a Coq function $f$, its inverse is defined by $f^{-1} = \mathscr{M}_{b;a}((\mathscr{L}f)^{-1})$. We need a bunch of trivial lemmas in order to use $\mathscr{M}$. This function satisfies $\mathscr{L}(f^{-1}) = (\mathscr{L}f)^{-1}$.

We have $f^{-1}(f(x)) = x$. By injectivity of $\mathscr{R}$, it suffices to show that $\mathscr{R}f^{-1}(f(x)) = \mathscr{R}x$. Using *bcreate*, it suffices to show that $\mathscr{W}_{\tilde{f}^{-1}}(\mathscr{R}f(x)) = \mathscr{R}x$. Denoting $y = \mathscr{R}x$ and $g = \tilde{f}$, we have to show that $\mathscr{W}_{g^{-1}}(\mathscr{W}_g(y)) = y$. This is a consequence of $g^{-1} \circ g = \mathrm{I}$, relation to be shown later. But since we know that $g$ and $g^{-1}$ are functions, we can restate this as follows: the pair $(\mathscr{W}_g(y), y)$ is in the inverse graph of $g$, which is the same as: the pair $(y, \mathscr{W}_g(y))$ is in the graph of $g$, which is true. From this we deduce $f(f^{-1}(f(x))) = f(x)$, and the surjectivity of $f$ gives $f \circ f^{-1} = \mathrm{I}$.

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

Definition inverseC a b (f:a->b)(H:bijectiveC f): b->a :=
  bcreate(bijective_inv_aux H)(bijective_source_aux f)
  (bijective_target_aux f).

Lemma inverseC_pra: forall (a b:Set) (f:a->b)(H:bijectiveC f) (x:a),
  (inverseC H) (f x) = x.
Lemma inverseC_prb: forall (a b:Set) (f:a->b)(H:bijectiveC f) (x:b),
  f ((inverseC H) x) = x.
Lemma inverseC_prc: forall (a b:Set) (f:a-> b) (H:bijectiveC f),
  inverse_fun(acreate f) = acreate(inverseC H).
```

```
Lemma bij_left_inverseC: forall (a b:Set) (f:a->b)(H:bijectiveC f) ,
  composeC (inverseC H) f = identityC a.
Lemma bij_right_inverseC: forall (a b:Set) (f:a->b)(H:bijectiveC f) ,
  composeC f (inverseC H) = identityC b.
```

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: forall (a b:Set) (f:a->b) g g',
  composeC  g f = identityC a -> composeC  f g' = identityC b ->
  bijectiveC f.
Lemma bijective_double_inverseC1: forall (a b:Set) (f:a->b) g g'
  (Ha: composeC  g f = identityC a)(Hb: composeC  f g' = identityC b),
  g = inverseC(bijective_double_inverseC Ha Hb)
  & g' = inverseC(bijective_double_inverseC Ha Hb).
Lemma bijective_inverseC: forall (a b:Set) (f:a->b)(H:bijectiveC f),
  bijectiveC (inverseC H).
Lemma inverse_fun_involutiveC:forall (a b:Set) (f:a->b) (H: bijectiveC f),
  f = inverseC(bijective_inverseC H).
```

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 inverse_bij_is_bij:forall f,
  bijective f ->  bijective (inverse_fun f).

Lemma composable_f_inv:forall f,
  bijective f ->  composable f (inverse_fun f).
Lemma composable_inv_f: forall f,
  bijective f ->  composable (inverse_fun f) f.
Lemma bij_right_inverse:forall f,
  bijective f ->  compose f (inverse_fun f) = identity (target f).
Lemma bij_left_inverse:forall f,
  bijective f ->  compose (inverse_fun f) f = identity (source f).

Lemma W_inverse: forall f x y,
  bijective f -> inc x (target f) ->
  (y = W x (inverse_fun f)) -> (x = W y f).
Lemma W_inverse2: forall f x y,
  bijective f -> inc y (source f) ->
  (x = W y f)-> (y = W x (inverse_fun f)).
Lemma W_inverse3: forall f x,
  bijective f -> inc x (target f) -> inc (W x (inverse_fun f)) (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: forall a b (f:a->b),
  bijectiveC f -> is_function (inverse_fun (acreate f)).
Lemma bijective_source_aux: forall a b (f:a->b),
  source (inverse_fun (acreate f)) = b.
Lemma bijective_target_aux: forall 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 $x \subset \mathrm{A}$ (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 $y \subset \mathrm{B}$. Equality holds if $f$ is surjective.

```
Lemma sub_inv_im_source:forall f y,
  is_function f -> sub y (target f) ->
  sub (inv_image_by_graph (graph f) y) (source f).

Lemma direct_inv_im: forall f y,
  is_function f ->  sub y (target f) ->
  sub (image_by_fun f (image_by_fun (inverse_fun f) y)) y.

Lemma direct_inv_im_surjective: forall f y,
  surjective  f -> sub y (target f) ->
  (image_by_fun f (image_by_fun (inverse_fun f) y)) = y.

Lemma inverse_direct_image:forall f x,
  is_function f -> sub x (source f) ->
  sub x (image_by_fun (inverse_fun f) (image_by_fun f x)).
Qed.

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

## 4.8 Retractions and sections

$$\begin{tikzcd} \mathrm{A} \arrow[r, "f"] \arrow[rr, bend right, "\mathrm{I_A}"'] & \mathrm{B} \arrow[r, "r"] & \mathrm{A} \end{tikzcd} \qquad \begin{tikzcd} \mathrm{B} \arrow[r, "s"] \arrow[rr, bend right, "\mathrm{I_B}"'] & \mathrm{A} \arrow[r, "f"] & \mathrm{B} \end{tikzcd} \qquad \text{(retraction/section)}$$

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 r f :=
  composable r f & compose r f = identity (source f).

Definition is_right_inverse s f :=
  composable f s & compose f s = identity (target f).

Definition is_left_inverseC (a b:Set) r (f:a->b) := composeC r f = identityC a.
Definition is_right_inverseC (a b:Set) s (f:a->b):= composeC f s = identityC b.
```

```
Lemma target_left_inverse: forall r f,
  is_left_inverse r f ->  target r = source f.
Lemma source_right_inverse: forall s f,
  is_right_inverse s f ->  source s = target f.
Lemma W_right_inverse: forall s f x,
  is_right_inverse s f -> inc x (target f) -> W (W x s) f = x.
Lemma W_left_inverse: forall r f x,
  is_left_inverse r f -> inc x (source f) -> W (W x f) r = x.
Lemma w_right_inverse: forall (a b:Set) s (f:a->b) (x:b),
  is_right_inverseC s f ->  f (s x) = x.
Lemma w_left_inverse: forall (a b:Set) r (f:a->b) (x:a),
  is_left_inverseC r f -> 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 $\mathrm{A} = \emptyset$, 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 $\mathrm{A} \neq \emptyset$. We start with the easy case.

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

Theorem inj_if_exists_left_inv: forall f,
  (exists r, is_left_inverse r f) -> injective f.
Theorem surj_if_exists_right_inv: forall f,
  (exists s, is_right_inverse s f) -> surjective 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 $f$. Assume $f$ surjective, so that there exists $y$ such that $f(y) = x$. We use the axiom of choice; call this $h(x)$. We have $f(h(x)) = x$, so that $h$ is a right inverse. In order to apply the axiom of choice, we have to ensure that the type $a$ is non-empty; this is an assumption for the left inverse, and dynamically checked for the right inverse (for any $x$, if its type allows it to be an argument of $h$, then $b$ is non-empty, and by surjectivity $a$ is non-empty and $h(x)$ exists).

The original code used some auxiliary lemmas, it is shown in section 9.3.

```
Definition left_inverseC (a b:Set) (f: a->b)(H:nonemptyT a)
  (v:b) := (chooseT (fun u:a => (~ (exists x:a, f x = v)) \/ (f u = v)) H).
Lemma left_inverseC_pr:forall (a b:Set) (f: a->b) (H:nonemptyT a) (u:a),
  f(left_inverseC f H (f u)) = f u.
Lemma left_inverse_comp_id: forall (a b:Set) (f:a->b)  (H:nonemptyT a),
  injectiveC f ->  composeC (left_inverseC f H) f = identityC a.
Lemma exists_left_inv_from_injC: forall (a b:Set) (f:a->b),nonemptyT 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 _ => nonemptyT_intro x end).
Lemma right_inverse_pr:  forall (a b:Set) (f: a->b) (H:surjectiveC f) (x:b),
  f(right_inverseC H x) = x.
```

```
Lemma right_inverse_pr:  forall (a b:Set) (f: a->b) (H:surjectiveC f) (x:b),
  f(right_inverseC H x) = x.
Lemma right_inverse_comp_id: forall (a b:Set) (f:a-> b) (H:surjectiveC f),
  composeC f (right_inverseC H)  = identityC b.
Lemma exists_right_inv_from_surjC: forall (a b:Set) (f:a-> b)(H:surjectiveC f),
  exists s, is_right_inverseC s f.
```

Bourbaki shows existence of a left inverse of the function $f : \mathrm{A} \to \mathrm{B}$ by considering the subset of $\mathrm{B} \times \mathrm{A}$ formed of all pairs $(x, y)$ such that $y \in \mathrm{A}$ and $y = f(x)$ or $y = e$ and $x \in \mathrm{B} \setminus f\langle\mathrm{A}\rangle$, where $e \in \mathrm{A}$ (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: forall f,
  injective f ->nonempty (source f) -> exists r:E, is_left_inverse r f.
Theorem exists_right_inv_from_surj: forall f,
  surjective  f -> exists s:E, is_right_inverse s f.
Theorem exists_left_inv_from_inj_alt: forall f,
  injective f -> nonempty (source f) -> exists r, is_left_inverse r f.  (* 41 *)
Theorem exists_right_inv_from_surj_alt: forall f,
  surjective  f -> exists s, is_right_inverse s f.   (* 17 *)
```

¶ 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: forall g f,
  composable g f -> composable f g -> compose g f = identity (source f)
  -> compose f g = identity (source g)
  ->(bijective f & bijective g & g = inverse_fun f).   (* 20 *)

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

Lemma right_inverse_from_left: forall r f,
  is_left_inverse r f -> is_right_inverse f r.
Lemma left_inverse_from_right: forall s f,
  is_right_inverse s f -> is_left_inverse f s.
Lemma left_inverse_surjective: forall f r,
  is_left_inverse r f -> surjective r.
Lemma right_inverse_injective: forall f s,
  is_right_inverse s f -> injective s.
Lemma section_unique: forall f s s',
  is_right_inverse s f -> is_right_inverse s' f ->
  range (graph s) = range (graph s') ->s = 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 composeC_inj: forall (a b c:Set) (f:a->b)(f':b->c),
  injectiveC f-> injectiveC f' -> injectiveC (composeC f' f).
Lemma composeC_surj: forall (a b c:Set) (f:a->b)(f':b->c),
  surjectiveC f-> surjectiveC f' -> surjectiveC (composeC f' f).
Lemma left_inverse_composeC: forall (a b c:Set)
  (f:a->b) (f':b->c)(r:b->a)(r':c->b),
  is_left_inverseC r' f' -> is_left_inverseC r f ->
  is_left_inverseC (composeC r r') (composeC f' f).
Lemma right_inverse_composeC: forall (a b c:Set)
  (f:a->b) (f':b->c)(s:b->a)(s':c->b),
  is_right_inverseC s' f' -> is_right_inverseC s f ->
  is_right_inverseC (composeC s s') (composeC f' f).
Lemma inj_right_composeC: forall (a b c:Set) (f:a->b) (f':b->c),
  injectiveC (composeC f' f) -> injectiveC f.
Lemma surj_left_composeC: forall (a b c:Set) (f:a->b) (f':b->c),
  surjectiveC (composeC f' f) -> surjectiveC f'.
Lemma surj_left_compose2C: forall (a b c:Set) (f:a->b) (f':b->c),
  surjectiveC (composeC f' f) -> injectiveC f' -> surjectiveC f.
Lemma inj_left_compose2C: forall (a b c :Set)(f:a->b) (f':b->c),
  injectiveC (composeC f' f) -> surjectiveC f -> injectiveC f'.
Lemma left_inv_compose_rfC: forall (a b c:Set) (f:a->b) (f':b->c)(r'': c->a),
  is_left_inverseC r'' (composeC f' f) ->
  is_left_inverseC (composeC r'' f') f.
Lemma right_inv_compose_rfC : forall (a b c:Set) (f:a->b) (f':b->c)(s'': c->a),
  is_right_inverseC s'' (composeC f' f) ->
  is_right_inverseC (composeC f s'') f'.
Lemma left_inv_compose_rf2C: forall (a b c:Set) (f:a->b) (f':b->c)(r'': c->a),
  is_left_inverseC r'' (composeC f' f) -> surjectiveC f ->
  is_left_inverseC (composeC f r'') f'.
Lemma right_inv_compose_rf2C : forall (a b c:Set) (f:a->b) (f':b->c)(s'': c->a),
  is_right_inverseC s'' (composeC f' f) -> injectiveC f'->
  is_right_inverseC (composeC s'' f') f.
```

Now the same results, in Bourbaki notations.

```
Theorem compose_injective: forall f f',
  injective f-> injective f' -> composable f' f ->
  injective (compose f' f).
Theorem compose_surjective: forall f f',
  surjective f-> surjective f' -> composable f' f ->
  surjective (compose f' f).
Lemma compose_bijective: forall f f',
  bijective f-> bijective f' -> composable f' f ->
  bijective (compose f' f).
Lemma left_inverse_composable: forall f f' r r',  composable f' f ->
  is_left_inverse r' f' -> is_left_inverse r f -> composable r r'.
Lemma right_inverse_composable: forall f f' s s',  composable f' f ->
  is_right_inverse s' f' -> is_right_inverse s f -> composable s s'.
Theorem left_inverse_compose: forall f f' r r', composable f' f ->
  is_left_inverse r' f' -> is_left_inverse r f ->
  is_left_inverse (compose r r') (compose f' f).
Theorem right_inverse_compose: forall f f' s s', composable f' f ->
  is_right_inverse s' f' -> is_right_inverse s f ->
  is_right_inverse (compose s s') (compose f' f).
Theorem inj_right_compose: forall f f',
  composable f' f-> injective (compose f' f) -> injective f.
Theorem surj_left_compose: forall f f',
  composable f' f-> surjective (compose f' f) -> surjective f'.
Theorem left_inv_compose_rf: forall f f' r'',
  composable f' f-> is_left_inverse r'' (compose f' f) ->
  is_left_inverse (compose r'' f') f.
Theorem right_inv_compose_rf : forall f f' s'',
  composable f' f-> is_right_inverse s'' (compose f' f) ->
  is_right_inverse (compose f s'') f'.
Theorem surj_left_compose2: forall f f',
  composable f' f-> surjective (compose f' f) -> injective f' -> surjective f.
Theorem inj_left_compose2: forall f f',
  composable f' f-> injective (compose f' f) -> surjective f -> injective f'.
Theorem left_inv_compose_rf2: forall f f' r'',
  composable f' f-> is_left_inverse r'' (compose f' f) -> surjective f ->
  is_left_inverse (compose f r'') f'. (* 23 *)
Theorem right_inv_compose_rf2 : forall f f' s'',
  composable f' f-> is_right_inverse s'' (compose f' f) -> injective f'->
  is_right_inverse (compose s'' f') f. (* 27 *)
```

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

```
Lemma bij_right_compose: forall f f',
  composable f' f-> bijective (compose f' f) -> bijective f' ->bijective f.
Lemma bij_left_compose: forall f f',
  composable f' f-> bijective (compose f' f) -> bijective f ->bijective f'.
```

Next three lemmas show that equipotency is an equivalence relation.

```
Lemma equipotent_reflexive: forall x, equipotent x x.
Lemma equipotent_symmetric:  forall a b,
  equipotent a b -> equipotent b a.
Lemma equipotent_transitive: forall a b c,
  equipotent a b -> equipotent b c -> equipotent a c.
```

$$\begin{tikzcd}
\mathrm{E} \arrow[d, bend right, "g"'] \arrow[dr, "f"] & \\
\mathrm{F} \arrow[u, bend right, "s"'] \arrow[r, "h"'] & \mathrm{G}
\end{tikzcd}
\qquad
\begin{tikzcd}
& \mathrm{E} \arrow[d, bend right, "r"'] \\
\mathrm{G} \arrow[ur, "f"] \arrow[r, "h"'] & \mathrm{F} \arrow[u, bend right, "g"']
\end{tikzcd}$$

(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: forall (a b c:Set) (f:a->b)(g:a->c),
  surjectiveC g ->
  (exists h, composeC h g = f) =
  (forall (x y:a), g x = g y -> f x = f y).
Theorem exists_left_composable: forall f g,
  is_function f -> surjective g -> source f = source g ->
  (exists h:E, composable  h g & compose h g = f) =
  (forall (x y:E), inc x (source g) -> inc y (source g) ->
    W x g = W y g -> W x f = W y f).  (* 13 *)

Lemma exists_left_composable_auxC: forall (a b c:Set) (f:a->b) (g:a-> c) s h,
  surjectiveC g -> is_right_inverseC s g ->
  composeC h g = f -> h = composeC f s.
Theorem exists_left_composable_aux: forall f g s h ,
  is_function f -> surjective g -> source f = source g ->
  is_right_inverse s g ->
  composable  h g -> compose h g = f -> h = compose f s.

Lemma exists_unique_left_composableC: forall (a b c:Set) (f:a->b)(g:a->c) h h',
  surjectiveC g -> composeC h g = f -> composeC h' g = f ->
  h = h'.
Theorem exists_unique_left_composable: forall f g h h',
  is_function f -> surjective g -> source f = source g ->
  composable  h g -> compose h g = f ->
  composable  h' g -> compose h' g = f -> h = h'.

Lemma left_composable_valueC: forall (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 s g  ->  h = composeC f s ->
  composeC h g = f.
Theorem left_composable_value: forall f g s h,
  is_function f -> surjective g -> source f = source g ->
  (forall (x y:E), inc x (source g) -> inc y (source g) ->
    W x g = W y g -> W x f = W y f) ->
  is_right_inverse s g  ->   h = compose f s-> compose h 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: forall (a b c:Set) (f:a->b) (g:c->b) h r,
  injectiveC g ->  is_left_inverseC r g  -> composeC g h = f
  -> h = composeC r f.
Theorem exists_right_composable_aux: forall f g h r,
  is_function f -> injective g -> target f = target g ->
  is_left_inverse r g  -> composable g h ->  compose g h = f
  -> h = compose r f.

Lemma exists_right_composable_uniqueC: forall (a b c:Set) (f:a->b)(g:c->b) h h',
  injectiveC g -> composeC g h = f -> composeC g h' = f -> h = h'.
Theorem exists_right_composable_unique: forall f g h h',
  is_function f -> injective g -> target f = target g ->
  composable g h ->  compose g h = f ->
  composable g h' ->  compose g h' = f -> h = h'.

Lemma exists_right_composableC: forall (a b c:Set) (f:a->b) (g:c->b),
  injectiveC g ->
  (exists h, composeC g h = f) =  (forall u, exists v, g v = f u).
Theorem exists_right_composable: forall f g,
  is_function f -> injective g -> target f = target g ->
  (exists h, composable  g h & compose g h = f) =
  (sub (range (graph f)) (range (graph g))).   (* 44 *)

Lemma right_composable_valueC: forall (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 = composeC r f ->  composeC g h = f.    (* 17 *)
Theorem right_composable_value: forall f g r h,
  is_function f -> injective g -> target g = target f ->
  is_left_inverse r g ->
  (sub (range (graph f)) (range (graph g))) ->
  h = compose r f ->,
  compose g h = f.
```

## 4.9 Functions of two arguments

By definition, all functions take one argument. Consider for example addition of integers. Its type is $nat \to nat \to nat$, so that addition is a function that takes an argument $a$, and returns a function that takes an argument $b$ and returns the sum $a+b$. An other example is the pair constructor of type $A \to B \to A * B$. We can consider the function of type $nat * nat \to nat$, that associates to a pair $(a, b)$ the sum $a+b$. This may also be called a function of two arguments. Switching between these two points of view is called currying or uncurrying.

An operator of type $A \to B \to C$ will be called a function of two arguments; for instance the union of two sets (where all three types are Set), or composition of function (where the types are *correspondenceC*). When we consider functions in the Bourbaki sense (with a source, a target and a graph), the second point of view will be used. More precisely, we interpret $A \to B$ to mean that the source is A and the target is B. A function of two arguments of type $A * B \to C$, is thus a function whose source is $A \times B$ and its target is C. For each element of A we get a function $B \to C$. Similarly, for each element of B we get a function $A \to C$. These two definitions of a function of two arguments are equivalent, it will be formalized in Chapter 6.

```
Definition partial_fun2 f y :=
  BL(fun x=> W (J x y) f) (im_singleton(inverse_graph (source f)) y) (target f).

Definition partial_fun1 f x :=
  BL(fun y=> W (J x y) f)(im_singleton (source f) x) (target f).

Lemma partial_fun1_axioms: forall f x, is_function f -> is_graph(source f) ->
  transf_axioms (fun y=> W (J x y) f)(im_singleton (source f) x) (target f).
Lemma partial_fun1_function: forall f x, is_function f -> is_graph(source f) ->
  is_function (partial_fun1 f x).
Lemma partial_fun1_W: forall f x y, is_function f -> is_graph(source f) ->
  inc (J x y) (source f) -> W y (partial_fun1 f x) = W (J x y) f.
Lemma artial_fun2_axioms: forall f y, is_function f -> is_graph(source f) ->
  transf_axioms (fun x=> W (J x y) f)(im_singleton(inverse_graph (source f)) y)
  (target f).
Lemma partial_fun2_function: forall f y, is_function f -> is_graph(source f) ->
```

```
  is_function (partial_fun2 f y).
Lemma partial_fun2_W: forall f x y, is_function f -> is_graph(source f) ->
  inc (J x y) (source f) -> W x (partial_fun2 f y) = W (J x y) f.
```

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 :=
  BL(fun z=> J (W (P z) u)(W (Q z) v))
  (product (source u)(source v))
  (product (target u)(target v)).
Lemma ext_to_prod_function: forall u v,
  is_function  u -> is_function v ->
  is_function (ext_to_prod u v).
Lemma ext_to_prod_W: forall u v a b,
  is_function u -> is_function v-> inc a (source u) -> inc b (source v)->
  W (J a b) (ext_to_prod u v)  =  J (W a u) (W b v).
Lemma ext_to_prod_W2: forall u v c,
  is_function u -> is_function v->
  inc c (product (source u)(source v)) ->
   W c (ext_to_prod u v) =  J (W (P c) u) (W (Q c) v).
Lemma ext_to_prod_range: forall u v,
  is_function u -> is_function v->
  range (graph (ext_to_prod u v)) =
    product (range (graph u))(range (graph v)).  (* 14 *)
```

$$\begin{tikzcd}
a \arrow[d, "u"] & a\times b \arrow[l, "\mathrm{pr1C}"'] \arrow[r, "\mathrm{pr2C}"] \arrow[d, "u\times v"] & b \arrow[d, "v"] \\
a' \arrow[r, "\mathrm{J}"'] & a'\times b' & b' \arrow[l, "\mathrm{J}"]
\end{tikzcd}$$
(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 $\mathscr{E}\to\mathscr{E}\to\mathscr{E}$) but is not a function of two arguments (its type is not $\mathscr{E}\times\mathscr{E}\to\mathscr{E}$).

```
Lemma ext_to_prod_propP: forall a a' (x: product a a'),  inc (P (Ro x)) a.
Lemma ext_to_prod_propQ: forall a a' (x: product a 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'))  (product b b').


Definition pr1C a b:= fun x:product a b => Bo(ext_to_prod_propP x).
Definition pr2C a b:= fun x:product a 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: forall (a b:Set) (x:product a b),
  Ro x = J (Ro (pr1C x)) (Ro (pr2C x)).
Lemma pr1C_prop: forall (a b:Set) (x:product a b),  Ro (pr1C x) =  P (Ro x).
Lemma pr2C_prop: forall (a b:Set) (x:product a b),  Ro (pr2C x) =  Q (Ro x).
Lemma prJ_prop: forall (a b:Set) (x:a)(y:b),   Ro(pairC x y) = J (Ro x) (Ro y).
Lemma prJ_recov: forall (a b:Set) (x:product a b), pairC (pr1C x) (pr2C x) = x.
Lemma ext_to_prod_prop:
  forall (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: forall u v,
  injective u -> injective  v-> injective (ext_to_prod u v).
Lemma ext_to_prod_surjective: forall u v,
  surjective u -> surjective  v-> surjective (ext_to_prod u v).
Lemma ext_to_prod_bijective: forall u v,
  bijective u -> bijective  v-> bijective (ext_to_prod u v).
Lemma ext_to_prod_inverse: forall u v,
  bijective u -> bijective  v->
  inverse_fun (ext_to_prod u v) =
  ext_to_prod (inverse_fun u)(inverse_fun v). (* 21 *)
Lemma composable_ext_to_prod2: forall u v u' v',
  composable u' u -> composable v' v ->
  composable (ext_to_prod u' v') (ext_to_prod u v).
Lemma compose_ext_to_prod2: forall u v u' v',
  composable u' u ->   composable v' v ->
  compose (ext_to_prod u' v') (ext_to_prod u v) =
  ext_to_prod (compose u' u)(compose v' v).  (* 19 *)
```

Same lemmas for Coq functions.

```
Lemma injective_ext_to_prod2C: forall (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: forall (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:  forall (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: forall (a b c a' b' c':Set) (u:b-> c)(v:a->b)
  (u':b'-> c')(v':a'->b'),
  composeC (ext_to_prodC u u') (ext_to_prodC v v') =
  ext_to_prodC (composeC u v)(composeC u' v').
Lemma inverse_ext_to_prod2C: forall (a b a' b':Set) (u:a->a')(v:b->b')
  (Hu: bijectiveC u)(Hv:bijectiveC  v),
  inverseC (bijective_ext_to_prod2C Hu Hv)=
  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: forall f, is_function f ->
   image_of_fun f = range (graph f).
```

```
Lemma canonical_decomposition1: forall f g i,
  is_function f ->
  g = restriction2 f (source f) (range (graph f)) ->
  i = canonical_injection (range (graph f)) (target f) ->
  (composable i g & f = compose i g & injective i & surjective g &
  (injective f -> bijective g )).  (* 24 *)
```

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: forall (a b:Set) (f:a->b) (x:a), inc (Ro (f x)) (imageC f).
Lemma imageC_exists: forall (a b:Set) (f:a->b) x,
  inc x (imageC f) -> exists y:a,  x = Ro (f y).
Lemma sub_image_targetC: forall (a b:Set) (f:a->b), sub (imageC f) b.

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

Lemma restriction_to_image_pr: forall (a b:Set) (f:a->b) (x:a),
  Ro(restriction_to_image f x) = Ro (f x).
Lemma canonical_decomposition1C: forall (a b:Set) (f:a->b)
  (g:a-> imageC f)(i:imageC f ->b),
  g = restriction_to_image f ->
  i = inclusionC (sub_image_target (f:=f))  ->
  (injectiveC i & surjectiveC g &
  (injectiveC f -> bijectiveC g )).
```

# Chapter 5

# Union and intersection of a family of sets

Bourbaki defines union, intersection and products of a family of sets. A family is just a function, that is, a source, a target, and a functional graph. The definition of the union [2, p. 90] is: “let $(\mathrm{X}_\iota)_{\iota\in\mathrm{I}}$ be a family of sets. The set [...] is called the union of the family and denoted by $\bigcup\limits_{\iota\in\mathrm{I}}\mathrm{X}_\iota$.” Intersection is similarly denoted by $\bigcap\limits_{\iota\in\mathrm{I}}\mathrm{X}_\iota$. In this notation $\iota$ is a dummy variable, X is the graph, I the source (called “the index set”), and the target is never mentioned. In fact, the definitions are independent of the target. Given a graph G, it is possible to construct a function with graph G (just use domain and range as source and target). Note that we do not have any choice for the source. The associated union will be independent of these choices.

For this reason, in what follows, *unionb X* is the union associated to the graph X. Using *unionb* requires that X be a functional graph. Assume now that I is a set, and X a mapping. Then $\mathscr{L}_\mathrm{I}\mathrm{X}$ is a functional graph, whose index set is I. For this reason, we introduce *unionf I X*. This better matches the definition of Bourbaki. In some cases, Bourbaki considers the family $(\mathrm{X}_{f(\kappa)})_{\kappa\in\mathrm{K}}$. This has to be understood as the family $\mathrm{X}\circ f$.

We consider another variant, *uniont* where X is a function on the type I with values into a set. This will be our primary definition. We must then show that all these definitions are equivalent, and the same as the original *union* (as defined by C. Simpson in Sections 2.7 and 2.8).

Intersection is only defined over a nonempty index set. We have tried to impose this restriction in the definition, but this gives theorems that are two complicated. As a result, we define intersection even over the empty set.

## 5.1 Definition of the union and intersection of a family of sets

We define here *uniont* $f$, *unionf I X* and *unionb G* as follows. The first definition is a variant of the union, as defined in section 2.7. A *Uintegral* record contains $z$ and $e$ of type $f(z)$, so that $\mathscr{R}e \in f(z)$. Hence the union is just the image of the function that associates to each record the quantity $\mathscr{R}e$.

The intersection of a function $f$ defined on a type I, denoted by *intersectiont* $f$, is the set of all $y \in \cup f$ such that $y \in f(z)$ for all $z$.

Given a set I and a mapping X defined on sets, we define *unionf I X* and *intersectionf I X* as the union and intersection of the functions defined on the type I by composing X with $\mathscr{R}$. If G is a graph, we define *unionb G* and *intersection G* as *unionf I X* and *intersectionf I X*, where I is the domain and X the evaluation function. It will be shown that *intersection X*, where X is a set of sets, is the intersection of the identity function on X. Similarly, *union X* is the union of the identity function.

```
Record Uintegral (I :Set)(f :I->Set) : Set := {UI_z : In; UI_elt : f UI_z}.

Definition uniont (I:Set)(f : I->Set) :=
   IM (fun i : Uintegral f => Ro (UI_elt i)).

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

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

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

```
Lemma uniont_rw: forall (In:Set) (f:In->Set) x,
  inc x (uniont f) = exists z,  inc x (f z).
Lemma unionf_rw: forall x i f,
  inc x (unionf i f) =  exists y, inc y i & inc x (f y).
Lemma unionb_rw: forall x f,
  inc x (unionb f) =  exists y, inc y (domain f) & inc x (V y f).
Lemma uniont_inc : forall (In:Set) (f : In->Set) y x,
  inc x (f y) -> inc x (uniont f).
Lemma uniont_exists : forall (In:Set) (f : In->Set) x,
  inc x (uniont f) -> exists y:In, inc x (f y).
Lemma unionf_inc: forall x y i f,
  inc y i -> inc x (f y) -> inc x (unionf i f).
Lemma unionf_exists:forall x i f,
  inc x (unionf i f) ->  exists y, inc y i & inc x (f y).
Lemma unionb_inc: forall x y f,
  inc y (domain f) -> inc x (V y f) ->  inc x (unionb f).
Lemma unionb_exists: forall x f,
  inc x (unionb f) -> exists y, inc y (domain f) & inc x (V y f).
```

Same lemmas for the intersection. All these lemmas are obvious from the definitions and the link between $\mathscr{R}$ and $\mathscr{B}$. An important point: all the definitions are meaningless if the domain is empty; for completeness, we show that the intersection is then empty. Note that th e domain is empty if and only if the graph is empty.

```
Lemma nonempty_domain: forall x,
  nonempty (domain x) = nonempty x.
Lemma intersectiont_rw: forall (I:Set) (f:I-> Set) x, nonemptyT I ->
  inc x (intersectiont f) = (forall j, inc x (f j)).
Lemma intersectionf_rw : forall I f x,  nonempty I ->
  inc x (intersectionf I f) = (forall j, inc j I -> inc x (f j)).
```

```
Lemma intersectionb_rw : forall g x, nonempty g ->
  inc x (intersectionb g) = (forall i, inc i (domain g) -> inc x (V i g)).
Lemma intersectiont_inc : forall (I:Set) (f:I-> Set) x, nonemptyT I ->
  (forall j, inc x (f j)) -> inc x (intersectiont f).
Lemma intersectiont_forall : forall (I:Set) (f:I-> Set) x j,
  inc x (intersectiont f) -> inc x (f j).
Lemma intersectionf_inc :forall I f x, nonempty I ->
  (forall j, inc j I -> inc x (f j)) -> inc x (intersectionf I f).
Lemma intersectionf_forall :forall I f x j,
  inc x (intersectionf I f) -> inc j I -> inc x (f j).
Lemma intersectionb_inc : forall g x,  nonempty g ->
  (forall i, inc i (domain g) -> inc x (V i g)) -> inc x (intersectionb g).
Lemma intersectionb_forall : forall g x i,
  inc x (intersectionb g) -> inc i (domain g) -> inc x (V i g).

Lemma intersectiont_empty:  forall (I:Set) (f:I-> Set),
  ~ (nonemptyT I) -> intersectiont f = emptyset.
Lemma intersectionf_empty:  forall f,
  intersectionf emptyset f = emptyset.
Lemma intersectionb_empty: forall f,
  f = emptyset -> intersectionb f = emptyset.
```

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

```
Lemma uniont_extensionality: forall (I:Set) (f: I-> Set) (f':I->Set),
  (forall i, f i = f' i) -> uniont f = uniont f'.
Lemma unionf_extensionality: forall sf f f',
  (forall i, inc i sf -> f i = f' i) -> unionf sf f = unionf sf f'.
Lemma unionb_extensionality: forall f f',
  f = f' -> unionb f = unionb f'.
Lemma intersectiont_extensionality:
  forall (I:Set) (f:I-> Set) (f':I-> Set),
    (forall i, f i = f' i) -> (intersectiont f) = (intersectiont f').
Lemma intersectionf_extensionality:  forall I f f,
  (forall i, inc i I -> f i = f' i) ->
  intersectionf I f  = intersectionf I f'.
Lemma intersectionb_extensionality: forall g g',
  g = g' -> intersectionb g = intersectionb g'.
```

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 uniont_sub:  forall (In:Set) (f: I-> Set) i,
  sub (f i) (uniont f).
Lemma intersectiont_sub:  forall (I:Set) (f: I-> Set) i,
  sub (intersectiont f) (f i).
Lemma sub_uniont:  forall (I:set) (f: I-> Set) x,
  (forall i, sub (f i) x) ->  sub (uniont f) x.
Lemma sub_intersectiont:  forall (I:Set)(f: I-> Set) x,
  nonemptyT I ->
  (forall i, sub x (f i)) ->  sub x (intersectiont f).
Lemma intersectiont_sub2:  forall (I:Set) (f: In-> Set) x,
   (forall i, sub (f i) x) ->  sub (intersectiont f) x.
Lemma sub_uniont2:  forall (I:Set) (f: I-> Set) x,
  nonemptyT I -> (forall i, sub x (f i)) -> sub x (uniont f).
```

If the index set is empty, so is the union.

```
Lemma empty_uniont1: forall (In:Set) (f: In-> Set),
  nonempty (uniont f) -> nonemptyT In.
Lemma empty_unionf1: forall sf f,
  nonempty (unionf sf f) -> nonempty sf.
Lemma empty_unionf: forall sf f,
  sf = emptyset -> unionf sf f = emptyset.
```

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

```
Theorem uniont_rewrite: forall (In K:Set) (f: K->In) (g:In ->Set),
  surjectiveC f ->
  uniont g = uniont (fun k:K => g(f k)).
Theorem intersectiont_rewrite: forall (I K:Set) (f: K->I) (g:I ->Set),
  surjectiveC f ->
  intersectiont g  = intersectiont (fun k:K => g(f k)).
```

The Bourbaki statement about union is *unionb_rewrite1*. In the second lemma we just assume that $f$ is a functional graph. Note that we dropped the requirement that $g$ must be a functional graph (we use *gcompose* instead). In the case of intersection, if $g$ is empty, so is $g \circ f$, and conversely.

```
Lemma unionb_rewrite1: forall f g,
  is_function f -> fgraph g -> range (graph f) = domain g ->
  unionb g = unionb (fcompose g (graph f)).
Lemma unionb_rewrite: forall f g,
  fgraph f -> range f = domain g ->
  unionb g = unionb (gcompose g f).
Lemma intersectionb_rewrite: forall f g,
  fgraph f -> range f = domain g ->
  intersectionb g = intersectionb (gcompose g f).
```

Let $f$ be a constant function and $x \in \mathrm{I}$. Then the intersection and union of $f$ on I is $f(x)$. This is trivial to show. But we shall follow Bourbaki: we first consider the case where I is a singleton. Then we shall prove that a constant function $h$ can be written as $h = g \circ f$ where the image of $f$ is a singleton, hence $f$ is surjective, and the union (or intersection) of $h$ is that of $g$. The conclusion is then obvious, since the source of $g$ is a singleton.

```
Definition is_singleton x := exists u, x = singleton u.

Lemma uniont_constant: forall (I:Set) (f:I ->Set) (x:I),
  is_constant_functionC f -> uniont f = f x.
Lemma intersectiont_constant: forall (I:Set) (f:I ->Set) (x:In),
  is_constant_functionC f -> intersectiont f = f x.

Lemma singleton_type_inj: forall x (y:singleton x)(z:singleton x),   y = z.
Lemma uniont_singleton:
  forall (a;Set) (x:a) (f: singleton (Ro x) -> Set),
  uniont f = f (Bo (singleton_inc (Ro x))).
Lemma intersectiont_singleton:
  forall (a:Set)(x:a) (f: singleton (Ro x) -> Set),
  intersectiont f = f (Bo (singleton_inc (Ro x))).
Lemma unionf_singleton:forall f x,
  unionf (singleton x) f = f x.
Lemma intersectionf_singleton:forall f x,
  intersectionf (singleton x) f = f x.
```

```
Lemma  constant_function_pr2:
  forall x h, inc x (source h) -> is_constant_function h ->
  exists g, exists f,
  (composable g f & h = compose g f & surjective f & is_singleton (target f))
Lemma constant_function_pr3: forall (a:Set) (h:a->Set) (x:a),
  is_constant_functionC h ->
  exists f: a->singleton (Ro x),
   exists g:singleton (Ro x)->Set,
     (forall u:a, h u = g (f u)) & (g (Bo (singleton_inc (Ro x))) = h 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.

A trivial consequence concerns union and intersection of a singleton.

```
Lemma union_prop: forall x,  union x = unionf x (fun u => u).
Lemma intersection_prop: forall x, nonempty x ->
  intersection x = intersectionf x (fun u => u).
Lemma unionb_alt: forall f, fgraph f -> unionb f = union (range f).
Lemma unionb_identity: forall x, unionb (identity_g x) = union x.
```

## 5.2 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 need A non-empty.

```
Lemma union_monotone: forall (In:Set) (f g:In->Set),
  (forall i, sub (f i) (g i)) -> sub (uniont f) (uniont g).
Lemma intersection_monotone: forall (I:Set)(f g:I->Set),
  (forall i, sub (f i) (g i)) -> sub (intersectiont f)(intersectiont g).
Lemma union_monotone2: forall A B f,
  sub A B -> sub (unionf A f) (unionf B f).
Lemma intersection_monotone2: forall A B f,
  sub A B -> nonempty A ->
  sub (intersectionf B f) (intersectionf A f).
```

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 $J_\lambda$ 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 union_assoc: forall sf sg f g,
  sf = unionf sg g ->
  unionf sf f = unionf sg (fun l => unionf (g l) f).
Theorem intersection_assoc: forall sf sg f g,
  (forall i, inc i sg -> nonempty (g i)) ->
  sf = unionf sg g ->
  intersectionf sf f = intersectionf sg (fun l => intersectionf (g l) f).
```

Proposition 3 [2, p. 94] says that if $\Gamma$ is a correspondence, $\Gamma\langle\bigcup X_\iota\rangle = \bigcup\Gamma\langle X_\iota\rangle$ and $\Gamma\langle\bigcap X_\iota\rangle \subset \bigcap\Gamma\langle X_\iota\rangle$. Proposition 4 [2, p. 95] says that we have equality if $\Gamma$ is the inverse of a function, and, as a consequence, if $\Gamma$ is an injective function. In fact, we use the canonical decomposition $\Gamma = i\circ g$, where $g$ is the restriction of $\Gamma$ on its image (hence is bijective), and $i$ is the inclusion map from the image of $\Gamma$ to its target (see lemma *canonical_decomposition1*). Then $\Gamma\langle x\rangle = g^{-1}\langle x\rangle$ for every set $x$. In the case of intersection, if I is empty, the theorems say $\emptyset = \emptyset$.

```
Theorem image_of_union: forall (I:Set) (f:I->Set) g,
  is_correspondence g ->
  image_by_fun g (uniont f) =
  uniont (fun i => image_by_fun g (f i)).
Theorem image_of_intersection: forall (I:Set) (f:I->Set) g,
  is_correspondence g ->
  sub (image_by_fun g (intersectiont f))
  (intersectiont  (fun i => image_by_fun g (f i))).
Theorem inv_image_of_intersection: forall (I:Set) (f:I->Set) g,
  is_function g ->
  (inv_image_by_fun g (intersectiont f)) =
  (intersectiont (fun i => inv_image_by_fun g (f i))).
Lemma inj_image_of_intersection: forall (I:Set) (f:I->Set) g,
  injective g ->
  (image_by_fun g (intersectiont f))
  = (intersectiont (fun i => image_by_fun g (f i))).
```

## 5.3 Complements of unions and intersections

Assume $X_\iota \subset X$, $Y_\iota = X\setminus X_\iota$. Then the intersection (resp. union) of the $X_\iota$ is the union (resp. intersection) of the $Y_\iota$ as subsets of X. This is Proposition 5 [2, p. 96].

```
Theorem complementary_union: forall (I:Set) (f:I-> Set) x,
  (forall i, sub (f i) x) ->  nonemptyT I ->
  complement x (uniont f) = intersectiont (fun i=> complement x (f i)).
Theorem complementary_intersection: forall (I:Set) (f:I-> Set) x,
  (forall i, sub (f i) x) ->  nonemptyT I ->
  complement x (intersectiont f) = uniont (fun i=> complement x (f i)).
Lemma complementary_union1: forall sf f x,
  nonempty sf ->  (forall i, inc i sf -> sub (f i) x) ->
  complement x (unionf sf f) = intersectionf sf (fun i=> complement x (f i)).
Lemma complementary_intersection1:  forall sf f x,
  nonempty sf ->  (forall i, inc i sf -> sub (f i) x) ->
  complement x (intersectionf sf f) = unionf sf (fun i=> complement x (f i)).
```

## 5.4 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. Some formulas are implemented in the file *set1.v*.

```
Lemma union_of_twosets_aux: forall x y f,
  unionf(doubleton x y) f = union2 (f x) (f y).
Lemma intersection_of_twosets_aux: forall x y f,
  intersectionf(doubleton x y) f = intersection2 (f x) (f y).
Lemma union_of_twosets: forall x y,
  unionf(doubleton x y)(fun w => w) = union2 x y.
Lemma intersection_of_twosets: forall x y,
  intersectionf(doubleton x y)(fun w => w) = intersection2 x y.
Lemma union_doubleton: forall x y,
  union2 (singleton x)(singleton y) = doubleton x y.
```

We have:

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

We have:

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

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

```
Lemma union2assoc:  forall x y z,
  union2 x (union2 y z) = union2 (union2 x y) z.
Lemma intersection2assoc:  forall x y z,
  intersection2 x (intersection2 y z) = intersection2 (intersection2 x y) z.
Lemma intersection_union_distrib1:  forall x y z,
  union2 x (intersection2 y z) = intersection2 (union2 x y) (union2 x z).
Lemma intersection_union_distrib2:  forall x y z,
  intersection2 x (union2 y z) = union2 (intersection2 x y) (intersection2 x 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:

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

```
Lemma union2_comp: forall x y z,
  complement z (union2 x y) = intersection2 (complement z x)(complement z y).
Lemma intersection2_comp: forall x y z,
  sub x z -> sub y z ->
  complement z (intersection2 x y) = union2 (complement z x)(complement z 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 union2_complement: forall x z,
  sub x z -> union2 x (complement z x) = z.
Lemma intersection2_complement: forall x z,
  sub x z -> intersection2 x (complement z x) = emptyset.
```

```
Lemma image_of_union2: forall g x y,
  is_correspondence g ->
  image_by_fun g (union2 x y) = union2 (image_by_fun g x)(image_by_fun g y).
Lemma image_of_intersection2: forall g x y,
  is_correspondence g ->
  sub (image_by_fun g (intersection2 x y))
  (intersection2 (image_by_fun g x) (image_by_fun g y)).
Lemma inv_image_of_intersection2: forall g x y,
  is_function g ->
  inv_image_by_fun g (intersection2 x y) =
  intersection2 (inv_image_by_fun g x)(inv_image_by_fun g y).
Lemma inj_image_of_intersection2: forall g x y,
  injective g ->
  image_by_fun g (intersection2 x y)
  = intersection2 (image_by_fun g x)(image_by_fun g 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 inv_image_of_comp: forall f x,
  is_function f -> sub x (target f)  ->
  inv_image_by_fun f (complement (target f) x) =
  complement (inv_image_by_fun f (target f))(inv_image_by_fun f x).
Lemma inj_image_of_comp: forall f x,
  injective f -> sub x (source f)  ->
  image_by_fun f (complement (source f) x) =
  complement (image_by_fun f (source f))(image_by_fun f x).
```

## 5.5 Coverings

A *covering* of a set X is a family $X_\iota$ whose union contains X. We give two definitions, in the first case, the family is defined by a function, and in the second one, by the graph of a function. We show that these definitions agree.

```
Definition covering f x :=  fgraph f & sub x (unionb f).
Definition covering_f sf f x :=  sub x (unionf sf f).
Definition covering_s f x :=  sub x (union f).
Lemma covering_pr: forall f x,
   fgraph f -> covering f x = covering_s (range f) x.
Lemma covering_f_pr: forall sf f x,
  covering_f sf f x = covering_s (range (L sf f)) x.
```

We say that a covering $(Y_\kappa)_{\kappa\in K}$ refines $(X_\iota)_{\iota\in I}$ if for all $\kappa$ there is $\iota$ 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_c 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_covering sf f sg g :=
  forall j, inc j sg -> exists i, inc i sf & sub (g j) (f i).

Definition coarser_c y y' :=
  forall a, inc a y' -> exists b, inc b y & sub a b.
```

```
Lemma coarser_same: forall sf f sg g ,
  coarser_covering sf f sg g = coarser_c (range (L sf f))(range (L sg g)).
Lemma coarser_transitive : forall y y' y'',
  coarser_c y y' -> coarser_c y' y'' -> coarser_c y y''.

Lemma sub_covering: forall Ia Ib f x,
  covering_f Ia f x ->  covering_f Ib f x -> sub Ib Ia ->
  coarser_covering Ia f Ib f.
```

Given two families $X_\iota$ and $Y_\kappa$, 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 $X_\iota$ and $Y_\kappa$ of Z we find a covering $i(X_\iota, Y_\kappa)$ of Z that refines $X_\iota$ and $Y_\kappa$, this is the supremum for the coarser ordering (ordering are defined in the second part of this report).

```
Definition intersection_covering f g :=
  fun z => intersection2 (f (P z)) (g (Q z)).

Definition intersection_covering2 x y:=
  range(L (product x y)(intersection_covering (fun w => w) (fun w => w))).
Lemma intersection_covering2_pr: forall x y z,
  inc z (intersection_covering2 x y) =
  exists a, exists b, inc a x & inc b y & intersection2 a b = z.
Lemma intersection_is_covering: forall sf f sg g x,
  covering_f sf f x -> covering_f sg g  x ->
  covering_f (product sf sg) (intersection_covering f g) x.
Lemma intersection_covering_coarser1: forall sf f sg g x,
  covering_f sf f x -> covering_f sg g  x ->
  coarser_covering sf f (product sf sg) (intersection_covering f g).
Lemma intersection_covering_coarser2: forall sf f sg g x,
  covering_f sf f x -> covering_f sg g  x ->
  coarser_covering sg g (product sf sg) (intersection_covering f g).
Lemma intersection_covering_coarser3: forall sf f sg g  sh h x,
  covering_f sf f x -> covering_f sg g x -> covering_f sh h  x ->
  coarser_covering sf f sh h ->
  coarser_covering sg g  sh h ->
  coarser_covering (product sf sg) (intersection_covering f g) sh 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_c* (which is defined on the set of partitions, as will be seen later).

```
Lemma product_is_covering2: forall u v x,
  covering_s u x -> covering_s v  x ->
  covering_s (intersection_covering2 u v) x.
Lemma intersection_cov_coarser1: forall f g x,
  covering_s f x -> covering_s g x ->
  coarser_c f (intersection_covering2 f g).
Lemma intersection_cov_coarser2: forall f g x,
  covering_s f x -> covering_s g x ->
  coarser_c g (intersection_covering2 f g).
Lemma intersection_cov_coarser3: forall f g h x,
  covering_s f x -> covering_s g x -> covering_s h x ->
  coarser_c f h ->  coarser_c g h ->
  coarser_c (intersection_covering2 f g) h.
```

If $X_\iota$ 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: forall sf f g,
  surjective g -> covering_f sf f (source g)
  -> covering_f sf (fun w => image_by_fun g (f w)) (target g).
Lemma inv_image_of_covering: forall sf f g,
  is_function g -> covering_f sf f (target g)
  -> covering_f sf (fun w => inv_image_by_fun g (f w)) (source g).
Lemma product_of_covering: forall sf f sg g x y,
  covering_f sf f x -> covering_f sg g y ->
  covering_f (product sf sg) (fun z => product (f (P z)) (g (Q z)))
  (product x y).
```

Proposition 7 [2, p. 99] says that if $X_\iota$ is a covering of E, then two functions that agree on each $X_\iota$ agree on E, and a function defined on each $X_\iota$ can be extended to E if the obvious compatibility conditions hold.

We modified the theorem, by adding the condition that the graph is the union of the graphs, and the range is the union of the ranges.

```
Definition function_prop f s t:=
  is_function f & source f = s & target f = t.
Definition function_prop_sub f s t:=
  is_function f & source f = s & sub (target f) t.

Lemma agrees_on_covering: forall sf f x g g',
  covering_f sf f x -> is_function g -> is_function g' ->
  source g = x -> source g' = x ->
  (forall i, inc i sf ->  agrees_on (intersection2 x (f i)) g g') ->
  agrees_on x g g'.

Lemma extension_covering1: forall sf f t h,
  (forall i, inc i sf -> function_prop (h i)(f i) t ) ->
  (forall i j, inc i sf -> inc j sf ->
    agrees_on (intersection2 (f i) (f j)) (h i) (h j)) ->
  exists g, function_prop g (unionf sf f) t &
    graph g = (unionf sf (fun i => (graph (h i)))) &
    range(graph g) = (unionf sf (fun i => (range (graph (h i))))) &
    (forall i, inc i sf -> agrees_on (f i) g (h i)).  (* 47 *)

Lemma extension_covering: forall sf f t h,
  (forall i, inc i sf -> function_prop (h i) (f i) t) ->
  (forall i j, inc i sf -> inc j sf ->
    agrees_on (intersection2 (f i) (f j)) (h i) (h j)) ->
  exists_unique (fun g => function_prop g (unionf sf f) t &
    (forall i, inc i sf -> agrees_on (f i) g (h i))).
```

## 5.6 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. Here *partition_fam* is a family $X_\iota$ of mutually disjoint sets, whose union is E; *partition_s* is a set of sets with this property, and a *partition* is a set of non-empty sets.

```
Definition disjoint x y := intersection2 x y = emptyset.
Definition mutually_disjoint f :=
  (forall i j, inc i (domain f) -> inc j (domain f) ->
    i = j \/ (disjoint (V i f) (V j f))).
Definition partition_s y x:=
  (union y = x) &
  (forall a b, inc a y -> inc b y -> a = b \/ disjoint a b).
Definition partition y x:=
  (union y = x) &
  (forall a, inc a y -> nonempty a) &
  (forall a b, inc a y -> inc b y -> a = b \/ disjoint a b).
Definition partition_fam f x:=
  fgraph f & mutually_disjoint f & unionb f = x.
```

We list below some properties of partitions. The important property if that if $(X_\iota)$ is a partition of E, each element of E is in a unique $X_\iota$.

```
Lemma mutually_disjoint_prop: forall f,
  (forall i j y, inc i (domain f) -> inc j (domain f)  ->
    inc y (V i f) -> inc y (V j f) -> i=j) ->
  mutually_disjoint f.
Lemma mutually_disjoint_prop2: forall x f,
  (forall i j y, inc i x -> inc j x  ->
    inc y (f i) -> inc y (f j) -> i=j) ->
  mutually_disjoint (L x f).
Lemma mutually_disjoint_prop1: forall f, is_function f ->
  (forall i j y, inc i (source f) -> inc j (source f)  ->
    inc y (W i f) -> inc y (W j f) -> i=j) ->
  mutually_disjoint (graph f).

Lemma partitionset_pr: forall y x,
  partition y x -> partition_s y x.
Lemma partition_same: forall y x,
  partition_s y x -> partition_fam (identity_g y) x.
Lemma partition_same2: forall y x,
  partition_fam y x -> partition_s (range y) x.
Lemma partitions_is_covering: forall y x,
  partition_s y x -> covering_s y x.
Lemma partition_fam_is_covering: forall y x,
  partition_fam y x -> covering y x.
Lemma partition_inc_exists: forall f x y,
  partition_fam f x -> inc y x -> exists i, (inc i (domain f) & inc y (V i f)).
Lemma partition_inc_unique: forall f x i j y,
  partition_fam f x -> inc i (domain f) -> inc y (V i f) ->
  inc j (domain f) -> inc y (V j f) -> i = j.
```

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

```
Lemma coarser_reflexive : forall y, coarser_c y y.
Lemma coarser_transitive : forall y y' y'',
  coarser_c y y' -> coarser_c y' y'' -> coarser_c y y''.
Lemma coarser_antisymmetric : forall y y' x,
  partition y x ->  partition y' x ->
  coarser_c y y' -> coarser_c y'  y ->  y = y'.
```

We construct here a function that maps $a$ to $x$ and $b$ to $y$. This function is well-defined if $a = x$ and $b = y$, since it is the identity on the doubleton $\{x, y\}$. It is also well defined if $a$ and *b* are distinct elements. We will use the fact that *two_points* is a set with exactly two elements *TPa* and *TPb*.

```
Definition variant a x y := (fun z:Set => Yo (z = a) x y).
Definition variantL a b x y := L (doubleton a b) (variant a x y).
Definition variantLc f g:=  Lvariant TPa TPb f g.

Definition partition_with_complement x j :=
  variantLc j (complement x j).

Lemma variant_if_rw: forall a x y z,
  z = a -> variant a x y z = x.
Lemma variant_if_not_rw: forall a x y z,
  z <> a -> variant a x y z = y.
Lemma variant_V_a : forall a b x y,
  V a (variantL a b x y)  = x.
Lemma variant_V_b : forall a b x y,
  b <> a ->  V b (variantL a b x y)  = y.
Lemma variant_fgraph : forall a b x y,
  fgraph (variantL a b x y).
Lemma variant_domain : forall a b x y,
  domain (variantL a b x y) = doubleton a b.

Lemma variantLc_fgraph : forall x y,
  fgraph (variantLc x y).
Lemma variantLc_domain: forall f g,
  domain (variantLc f g) = two_points.
Lemma variantLc_domain_nonempty: forall f g,
  nonempty (domain (variantLc f g)).
Lemma variant_V_ca : forall x y,  V TPa (variantLc x y) = x.
Lemma variant_V_cb : forall x y,  V TPb (variantLc x y) = y.
Lemma variant_if_rw1: forall  x y, variant TPa x y TPa = x.
Lemma variant_if_not_rw1: forall  x y, variant TPa x y TPb = y.
```

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).

```
Lemma disjoint_pr: forall a b,
  (forall u, inc u a -> inc u b -> False) -> disjoint a b.
Lemma disjoint_complement : forall x y,
  disjoint y (complement x y).
Lemma disjoint_symmetric : forall x y,
  disjoint x y -> disjoint y x.
Lemma is_partition_with_complement: forall x j,
  sub j x -> partition_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 $X_\iota$ is a partition family, then the mapping $\iota \mapsto X_\iota$ is injective (we use the fact that $X_\iota$ is not empty). Inverse images of disjoint sets by a function are disjoint.

```
Definition largest_partition x := range(L x (fun z => singleton z)).
Definition smallest_partition x :=   (singleton x).
Lemma partition_smallest: forall x,
   nonempty x -> partition_set (smallest_partition x) x.
```

```
Lemma largest_partition_pr: forall x z,
  inc z (largest_partition x) = exists w, inc w x & singleton w = z.
Lemma partition_largest: forall x, partition_set (largest_partition x) x.
Definition injective_graph f:=
  fgraph f &
  (forall x y, inc x (domain f) -> inc y (domain f) ->
    V x f = V y f -> x = y).

Lemma injective_partition: forall f x,
  partition_fam f x -> (forall i,  inc i (domain f) -> nonempty (V i f))
  -> injective_graph f.
Lemma partition_fam_partition: forall f x,
  partition_fam f x -> (forall i,  inc i (domain f) -> nonempty (V i f))
  -> partition(range f) x.
Lemma inv_image_disjoint :  forall g x y,
  is_function g ->  disjoint x y ->
  disjoint (inv_image_by_fun g x)(inv_image_by_fun g 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 $f_\iota$. The assumption is that $f_\iota$ is a function defined on $X_\iota$, with target T. We give a variant (without uniqueness) where the target of $f_\iota$ is a subset of T. The set of functions $\mathscr{F}$ will be defined later.

```
Theorem extension_partition: forall f x t h,
  partition_fam f x ->
  (forall i, inc i (domain f) -> function_prop (h i) (V i f) t) ->
  exists_unique (fun g => function_prop g x t &
    (forall i, inc i (domain f) -> agrees_on (V i f) g (h i))).
Theorem extension_partition1: forall f x t h,
  partition_fam f x ->
  (forall i, inc i (domain f) -> function_prop_sub (h i) (V i f) t) ->
  exists g, function_prop g x t &
    (forall i, inc i (domain f) -> agrees_on (V i f) g (h i)).
```

## 5.7 Sum of a family of sets

Proposition 9 [2, p. 100] says that, for any family $X_\iota$, there exists a family $X'_\iota$ of sets equipotent to $X_\iota$, that are mutually disjoint, and a set X that is the union of these sets. After that we define 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 $X_\iota$ 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.

```
Theorem disjoint_union_lemma : forall f,
  fgraph f -> exists g, exists x,
    fgraph g &  x = unionb g &
    (forall i, inc i (domain f) -> equipotent (V i f) (V i g))
    & mutually_disjoint g.   (* 52 *)

Definition disjoint_union_fam f :=
  L (domain f)(fun i => product (V i f) (singleton i)).
```

```
Definition disjoint_union f :=
  unionb (disjoint_union_fam f).

Lemma disjoint_union_disjoint:
  forall f, fgraph f ->
    mutually_disjoint(disjoint_union_fam f).

Lemma du_index_pr:forall f x, inc x (disjoint_union f) ->
  (inc (Q x) (domain f) &  inc (P x) (V (Q x) f) & is_pair x).
Lemma inc_disjoint_union: forall f x y,
  inc y (domain f) -> inc x (V y f) ->
  inc (J x y) (disjoint_union f).

Lemma partion_union_disjoint:  forall f, fgraph f ->
  partition_fam (disjoint_union_fam f)  (disjoint_union f).
Theorem disjoint_union_pr: forall f,
  fgraph f -> exists x,
    source x = disjoint_union f &
    target x = unionb f &
    surjective x &
    (mutually_disjoint f -> bijective x).   (* 46 *)
```

# Chapter 6

# Product of a family of sets

## 6.1 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 2.6.

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 :=
  BL(image_by_fun f) (powerset (source f)) (powerset (target f)).
Lemma etp_axioms: forall f,  is_correspondence f ->
  transf_axioms (image_by_fun f) (powerset (source f))
  (powerset (target f)).
Lemma etp_function: forall f,
  is_correspondence f -> is_function  (extension_to_parts f).
Lemma etp_W: forall f x,
  is_correspondence f -> sub x (source f)
  -> W x (extension_to_parts f) = image_by_fun f x.
Lemma etp_composable: forall f g,
  composableC g f ->
  composable (extension_to_parts g)  (extension_to_parts f).
Lemma etp_compose: forall f g,
  composableC g f ->
  compose (extension_to_parts g)  (extension_to_parts f)
  = extension_to_parts (compose g f).
Lemma etp_identity: forall x,
  extension_to_parts (identity x) = identity (powerset x).
Lemma composable_for_function: forall f g, composable g f -> composableC g f.

Theorem etp_surjective: forall f,
  surjective f -> surjective (extension_to_parts f).
Theorem etp_injective: forall f,
  injective f -> injective (extension_to_parts f).
```

## 6.2 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 set_of_functions x y :=
  Zo (set_of_correspondences x y)
     (fun z => fgraph (graph z)& x = domain (graph z)).
Definition set_of_permutations E :=
  Zo (set_of_functions E E)(fun z=> bijective z).
Lemma set_of_functions_rw: forall x y f,
  inc f (set_of_functions x y) =
  (is_function  f &  source f = x & target f = 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 set_of_gfunctions x y :=
  Zo (powerset (product x y))(fun z => fgraph z & x = domain z).
Lemma inc_set_of_gfunctions: forall f,
  is_function f -> inc(graph f) (set_of_gfunctions (source f) (target f)).
Lemma set_of_gfunctions_inc: forall x y z,
  inc z (set_of_gfunctions x y) -> exists f,
    is_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 set_of_sub_functions x y:=
  unionf(powerset x)(fun z=> (set_of_functions z y)).
Lemma set_of_sub_functions_rw: forall x y f,
  inc f (set_of_sub_functions x y) =
  (is_function  f &  sub (source f) x & target f = y).
```

We list below some properties of exceptional functions. Remember that a small set has at most one element. We then show that there is an obvious bijection between $\mathscr{F}(E; F)$ and $F^E$.

```
Lemma empty_source_graph: forall f,
  is_function f -> source f = emptyset -> graph f = emptyset.
Lemma empty_target_graph: forall f,
  is_function f -> target f = emptyset -> graph f = emptyset.

Lemma set_of_functions_extens: forall x y a b,
  inc a (set_of_functions x y) -> inc b (set_of_functions x y) ->
  graph a = graph b -> a = b.

Lemma small_set_of_functions_source: forall y,
  small_set (set_of_functions emptyset y).
Lemma small_set_of_functions_target: forall x,
  small_set (set_of_functions x emptyset).
Lemma empty_set_of_functions_target: forall x y,
  (x = emptyset \/ nonempty y) -> nonempty (set_of_functions x y).
Lemma graph_axioms: forall x y,
  transf_axioms graph (set_of_functions x y)  (set_of_gfunctions x y).
Lemma graph_bijective: forall x y,
  bijective (BL graph (set_of_functions x y) (set_of_gfunctions x y)).
Lemma set_of_functions_equipotent: forall x y,
  equipotent (set_of_functions x y) (set_of_gfunctions x y).
```

$$\begin{tikzcd} E \arrow[r, "f"] & F \arrow[d, "v"] \\ E' \arrow[u, "u"] \arrow[r, "f'"'] & F' \end{tikzcd} \qquad \text{(compose3function)}$$

Given $f \in \mathscr{F}(E;F)$, we construct $f' \in \mathscr{F}(E';F')$ 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 :=
  BL (fun f => compose (compose v f) u)
  (set_of_functions (target u) (source v))
  (set_of_functions (source u) (target v)).

Lemma c3f_axioms: forall u v,
  is_function u -> is_function v ->
  transf_axioms (fun f => compose (compose v f) u)
  (set_of_functions (target u) (source v))
  (set_of_functions (source u) (target v)).
Lemma c3f_function: forall u v,
  is_function u -> is_function v -> is_function (compose3function u v).
Lemma c3f_W: forall u v f,
  is_function u -> is_function v ->
  is_function f -> source f = target u -> target f = source v ->
  W f (compose3function u v) = compose (compose v f) u.
Theorem c3f_injective: forall u v,
  surjective u -> injective v -> injective  (compose3function u v).   (* 25 *)
Theorem c3f_surjective: forall u v,
  (nonempty (source u) \/ (nonempty (source v)) \/ (nonempty (target v))
    \/ target u = emptyset) ->
  injective u -> surjective v -> surjective  (compose3function u v).    (* 43 *)
Lemma c3f_bijective: forall u v,
  bijective u -> bijective v -> bijective  (compose3function u v).
```

We now define the canonical bijections from $\mathscr{F}(B\times C;A)$ into $\mathscr{F}(B;\mathscr{F}(C;A)$ or $\mathscr{F}(C;\mathscr{F}(B;A))$. For any function $f(x,y)$ we can fix one of the variables to get a function.

```
Definition partial_fun_axioms f :=
  is_function f & exists a, exists b, source f = product a b.
Definition first_partial_fun f y:=
  BL(fun x => W (J x y) f) (domain (source f)) (target f).
Definition second_partial_fun f x:=
  BL(fun y => W (J x y) f) (range (source f)) (target f).
Definition first_partial_function f:=
  BL(fun y => first_partial_fun f y) (range (source f))
  (set_of_functions (domain (source f)) (target f)).
```

```
Definition second_partial_function f:=
  BL(fun x => second_partial_fun f x) (domain (source f))
  (set_of_functions (range (source f)) (target f)).
Definition first_partial_map b c a:=
  BL (fun f=> first_partial_function f)
  (set_of_functions (product b c) a)
  (set_of_functions c (set_of_functions b a)).
Definition second_partial_map b c a:=
  BL (fun f=>  second_partial_function f)
  (set_of_functions (product b c) a)
  (set_of_functions b (set_of_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_axioms_pr : forall f,
  partial_fun_axioms f ->
  source f = product (domain (source f))(range (source f)).
Lemma fpf_axioms: forall f y,
  partial_fun_axioms f -> inc y (range (source f)) ->
  transf_axioms (fun x => W (J x y) f) (domain (source f)) (target f).
Lemma spf_axioms: forall f x,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  transf_axioms (fun y => W (J x y) f) (range (source f)) (target f).
Lemma fpf_function :forall f y,
  partial_fun_axioms f -> inc y (range (source f)) ->
  is_function (first_partial_fun f y).
Lemma spf_function :forall f x,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  is_function (second_partial_fun f x).
Lemma fpf_W :forall f x y,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  inc y (range (source f)) ->
  W x (first_partial_fun f y) = W (J x y) f.
Lemma spf_W :forall f x y,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  inc y (range (source f)) ->
  W y (second_partial_fun f x) = W (J x y) f.
```

The next lemmas show that both $x \mapsto f_x$ and $y \mapsto f_y$ are functions.

```
Lemma fpfa_axioms: forall f,
  partial_fun_axioms f ->
  transf_axioms (fun y => first_partial_fun f y)(range (source f))
  (set_of_functions (domain (source f)) (target f)).
Lemma spfa_axioms: forall f ,
  partial_fun_axioms f ->
  transf_axioms (fun x => second_partial_fun f x)(domain (source f))
  (set_of_functions (range (source f)) (target f)).
Lemma fpfa_function: forall f,
  partial_fun_axioms f -> is_function (first_partial_function f).
Lemma spfa_function: forall f ,
 partial_fun_axioms f -> is_function (second_partial_function f).
Lemma fpfa_W: forall f y,
  partial_fun_axioms f -> inc y  (range (source f)) ->
  W y (first_partial_function f) = first_partial_fun f y.
```

```
Lemma spfa_W: forall f x,
  partial_fun_axioms f -> inc x (domain (source f)) ->
  W x (second_partial_function f) = 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_axioms: forall a b c,
  nonempty b -> nonempty c ->
  transf_axioms (fun f=> first_partial_function f)
  (set_of_functions (product b c) a)
  (set_of_functions c (set_of_functions b a)).
Lemma spfb_axioms: forall a b c,
  nonempty b -> nonempty c ->
  transf_axioms (fun f=> (fun f=> second_partial_function f)
  (set_of_functions (product b c) a)
  (set_of_functions b (set_of_functions c a)).
Lemma fpfb_function: forall a b c,
  nonempty a -> nonempty b -> is_function (first_partial_map a b c).
Lemma spfb_function: forall a b c,
  nonempty a -> nonempty b -> is_function (second_partial_map a b c).
Lemma fpfb_W: forall  a b c f,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  W f (first_partial_map a b c) = first_partial_function f.
Lemma spfb_W: forall a b c f,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  W f (second_partial_map a b c) = second_partial_function f.
Lemma fpfb_WW: forall a b c f x,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  inc x (product a b) ->
  W (P x)  (W (Q x) (W f (first_partial_map a b c))) = W x f.
Lemma spfb_WW: forall a b c f x,
  nonempty a -> nonempty b -> inc f (set_of_functions (product a b) c) ->
  inc x (product a b) ->
  W (Q x) (W (P x) (W f (second_partial_map a b c))) = W x f.
```

We now prove the main result, Proposition 3 of [2, p. 103].

```
Theorem fpfa_bijective: forall a b c,
  nonempty a -> nonempty b -> bijective (first_partial_map a b c).     (* 38 *)
Theorem spfa_bijective: forall a b c,
  nonempty a -> nonempty b -> bijective (second_partial_map a b c).    (* 38 *)
```

## 6.3 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 $X_1,\dots,X_n$ is a tuple $(x_1,\dots,x_n)$, 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 three definitions of the product, in the same way as we gave three definitions for the union or the intersection. Note that the family $(X_\iota)$ can be defined by a function $f : a \to E$, but the family $(x_\iota)$ cannot, because there is not set containing such objects; an element of a product is the graph of a function. We define below *gbcreate*, a variant of *gacreate*, that is the graph of *acreate*, that converts $f : a \to E$ into a graph.

```
Definition gbcreate (a:Set) (f:a->Set) := (IM (fun y:a => J (Ro y) (f y))).
Lemma gbcreate_rw: forall (a:Set) (f:a->E) x,
  inc x (gbcreate f) = exists y:a, J (Ro y) (f y) =x.
Lemma gbcreate_graph: forall (a:Set) (f:a-> Set), is_graph (gbcreate f).
Lemma gbcreate_fgraph: forall (a:Set) (f:a-> Set), fgraph (gbcreate f).
Lemma gbcreate_domain : forall (a:Set) (f:a-> Bsezt), domain (gbcreate f) = a.
Lemma gbcreate_V : forall (a:Set) (f:a-> Set) (x:a), V (Ro x) (gbcreate f) = f x.
```

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 $\mathfrak{P}(I \times A)$ that are are graphs of functions with this property. If all $X_\iota$ 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.

In the basic definition, the family is defined by the graph of a function, but in some cases, it is easier to consider a function defined on the type I. If I has two elements, the product is isomorphic to the product of two sets. Contrarily to the union and intersection, the order of the family is important.

```
Definition productb f:=
  Zo (powerset (product (domain f) (unionb f)))
  (fun z => fgraph z & domain z = domain f
    & forall i, inc i (domain f) -> inc (V i z) (V i f)).

Definition productt (I:Set) (f:I->Set):= productb (gbcreate f).
Definition productf sf f :=  productb (L sf f).
```

We list below some basic properties of products.

```
Lemma productb_rw: forall f x,  fgraph f ->
  inc x (productb f) =
  (fgraph x &  domain x = domain f &
    forall i, inc i (domain x) -> inc (V i x) (V i f)).
Lemma productt_rw: forall (In:Set) (f:In->Set) x,
  inc x (productt f) =
  (fgraph x &  domain x = In  & forall i:In, inc (V (Ro i) x) (f i)).
Lemma productf_rw: forall sf f x,
  inc x (productf sf f) =
  (fgraph x &  domain x = sf &
    forall i, inc i (domain x) -> inc (V i x) (f i)).
```

We give here an extensionality properties for elements of a product.

```
Lemma productt_exten: forall (I:Set) (f:I->Set) x x',
  inc x (productt f) -> inc x' (productt f) ->
  (forall i:I, V (Ro i) x = V (Ro i) x') ->
  x = x'.
Lemma productb_exten: forall f x x',
  fgraph f ->
  inc x (productb f) -> inc x' (productb f) ->
  (forall i, inc i (domain f) -> V i x = V i x') ->
  x = x'.
Lemma productf_exten: forall sf f x x',
  inc x (productf sf f) -> inc x' (productf sf f) ->
  (forall i, inc i sf -> V i x = V i x') ->
  x = x'.
Lemma trivial_fgraph: forall f, L emptyset f = emptyset.
```

We define now $\mathrm{pr}_\iota$, the $\iota$-th projection of a product; it is like $\mathrm{pr}_1$ and $\mathrm{pr}_2$ for the product of two sets. Let $f$ be an element of the product and $\iota$ an index; we have $\mathrm{pr}_\iota f = f_\iota$, where $f_\iota$ 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 $\mathrm{X}_\iota$.

Note. Since we have three types of products, we have three types of partial products, and three theorems for each property. We consider here only one case.

```
Definition pr_i f i:=  BL(V i) (productb f) (V i f).

Lemma pri_axioms: forall f i,
  fgraph f -> inc i (domain f) ->
  transf_axioms (V i)(productb f)(V i f).
Lemma pri_function: forall f i,
  fgraph f -> inc i (domain f) ->
  is_function (pr_i f i).
Lemma pri_W: forall f i x,
  fgraph f -> inc i (domain f) -> inc x (productb f) ->
  W x (pr_i f i) = V i x.
```

If the sets $\mathrm{X}_\iota$ are non empty, so is the product, and conversely. The idea is that we have $x_\iota \in \mathrm{X}_\iota$ for some $x_\iota$, and we can construct a function $\iota \mapsto x_\iota$.

```
Lemma product_trivial :
  productb emptyset = singleton emptyset.
Lemma product2_trivial: forall f,
  productb (L emptyset f) = singleton emptyset.
Lemma product_nonempty: forall f,
  fgraph f ->  (forall i, inc i (domain f) -> nonempty (V i f)) ->
  nonempty (productb f).
Lemma product_nonempty2: forall f,
  fgraph f ->  nonempty (productb f) ->
  (forall i, inc i (domain f) -> nonempty (V i f)).
Lemma productt_nonempty: forall (In:Set) (f:In->Set),
  (forall i, nonempty (f i)) ->  nonempty (productt f).
Lemma productt_nonempty2: forall (In:Set) (f:In->Set),
  nonempty (productt f) -> (forall i, nonempty (f i)).
```

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 $\iota$. Then $\prod_{\iota\in\mathrm{I}} \mathrm{E} = \mathrm{E}^{\mathrm{I}}$.

```
Lemma graphset_pr1: forall a b z,
  inc z (set_of_gfunctions a b) =
  (fgraph z & domain z = a & sub z (product a b)).
Lemma graphset_pr2: forall a b z,
  inc z (set_of_gfunctions a b) =
  (fgraph z & domain z = a & sub (range z) b).
Lemma product_sub_graphset:  forall f x,
  fgraph f -> sub (unionb f) x ->
  sub (productb f) (set_of_gfunctions (domain f) x).
Lemma product_eq_graphset:  forall f x,
  fgraph f -> (forall i, inc i(domain f) -> V i f = x) ->
  productb f = set_of_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 isomorphic to $X_\alpha$. The definition *product1a* constructs a product with one set given the parameters $x = X_\alpha$ and $a = \alpha$. If $I = \{\alpha\}$, then the product is equal to $x^I$ since I is a singleton. We construct a function from $x$ into the product that associates to each $y$ the constant function $y$.

```
Definition product1 (x a:Set) := productt  (fun _:singleton a => x).
Lemma product1_pr: forall x a,
  product1 x a= set_of_gfunctions (singleton a) x.
Lemma product1_rw: forall x a y,
  inc y (product1 x a) = (fgraph y & domain y = singleton a
    & inc (V a y) x).
Definition product1_canon x a :=
  BL (fun i : Set => L(singleton a) (fun _ : Set => i))  x (product1 x a).
Lemma product1_canon_axioms: forall x a,
 transf_axioms (fun i =>  L(singleton a) (fun _ : Set => i)) x (product1 x a).
Lemma product1_canon_function: forall x a,
  is_function (product1_canon x a).
Lemma W_product1_canon: forall x a i,
  inc i x -> W i (product1_canon x a) =
  L(singleton a) (fun _ : Set => i).
Lemma product1_canon_bijective: forall x a,
  bijective(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, consider only the case where I is the basic doubleton (we might as well consider the case where $\alpha = \emptyset$ and $\beta = \{\emptyset\}$, which is what Bourbaki suggests whenever two distinct sets are needed).

```
Definition product2 x y :=
  productf two_points (variant TPa x y).
Definition product2_canon x y :=
  BL (fun z => (variantLc (P z) (Q z)))  (product x y) (product2 x y).

Lemma product2_rw: forall x y z,
  inc z (product2 x y) = (fgraph z & domain z = two_points &
    inc (V TPa z) x & inc (V TPb z) y).
Lemma product2_canon_axioms: forall x y,
  transf_axioms (fun z  => Lvariantc (P z) (Q z))
  (product x y) (product2 x y).
Lemma product2_canon_function: forall x y,
  is_function (product2_canon x y).
Lemma product2_canon_W: forall x y z,
  inc z (product x y) -> W z (product2_canon x y) = variantLc (P z) (Q z).
Lemma product2_canon_bijective: forall x y,
  bijective (product2_canon x y).
```

If each $X_\iota$ is a singleton, so is the product $\prod X_\iota$.

```
Lemma is_singleton_rw: forall x,
  is_singleton x = (nonempty x & (forall a b, inc a x -> inc b x -> a = b)).
Lemma product_singleton: forall f,
  fgraph f -> (forall i, inc i (domain f) -> is_singleton (V i f))
  -> is_singleton (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 constant_graph s x :=
  L s (fun _ => x).
Definition is_constant_graph f :=
  fgraph f &
  (forall x x', inc x (domain f) -> inc x' (domain f) -> V x f = V x' f).
Definition diagonal_graphp e i :=
  Zo(set_of_gfunctions i e) is_constant_graph.
Definition constant_functor i e:=
  BL(fun x => constant_graph i x) e (set_of_gfunctions i e).

Lemma constant_graph_function: forall f,
  is_constant_function f -> is_constant_graph(graph f).
Lemma constant_graph_V: forall s x y,
  inc y s -> V y (constant_graph s x) = x.
Lemma constant_graph_small_range: forall f,
  is_constant_graph f -> small_set(range f).
Lemma diagonal_graph_rw: forall e i x,
  inc x (diagonal_graphp e i) =
  (is_constant_graph x & domain x = i & sub (range x) e).
Lemma constant_graph_is_constant: forall x y,
  is_constant_graph(constant_graph x y).
Lemma cf_injective : forall i e,
  nonempty i -> injective (constant_functor i e).
```

Proposition 4 [2, p. 104] says: Given a family $X_\iota$ 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 :=
  BL (fun x => >compose_graph x (graph u))
  (productb f) (productf (source u) (fun k => V (W k u) f)).
Lemma compose_V: forall u v x,
  fgraph u -> fgraph v -> fgraph (compose_graph u v) ->
  inc x (domain (compose_graph u v)) ->
  V x (compose_graph u v) = V (V x v) u.

Lemma pc_axioms0: forall f u c,
  fgraph f -> bijective u -> target u = domain f ->
  inc c (productb f)  ->
  let g:= compose (corresp (domain c) (range c) c) u  in
    is_function g & compose_graph c (graph u) = graph g
    (forall i, inc i (source u) ->
      V i (graph g) = V (V i (graph u)) c).
Lemma pc_axioms: forall f u,
  fgraph f -> bijective u -> target u = domain f ->
  transf_axioms (fun x => compose_graph x (graph u))
     (productb f) (productf (source u) (fun k  => V (W k u) f)).
Lemma pc_function: forall f u,
  fgraph f -> bijective u -> target u = domain f ->
  is_function(product_compose f u).
Lemma pc_W: forall f u x,
  fgraph f -> bijective u -> target u = domain f ->
  inc x (productb f) -> W x (product_compose f u) = compose_graph x (graph u).
Lemma pc_WV: forall f u x i,
  fgraph f -> bijective u -> target u = domain f ->
  inc x (productb f) -> inc i (source u) ->
   V i (W x (product_compose f u)) = V  (W i u) x.
Lemma pc_bijective: forall f u,
  fgraph f -> bijective u -> target u = domain f ->
  bijective (product_compose f u).   (* 33 *)
```

## 6.4 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 graph_exten: forall r r',
  is_graph r -> is_graph r' ->
  (r = r') =  forall u v, related r u v = related r' u v.
Lemma restriction_graph2 : forall f j,
  fgraph f -> sub j (domain f) ->
  L j (fun x  => V x f) = compose_graph f (identity_g j).
Lemma restr_domain1 : forall f x,
  fgraph f -> sub x (domain f) -> domain (restr f x) =  x.
```

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 $\mathrm{pr}_J$.

```
Definition restriction_product f j := productb (restr f j).
Definition pr_j f j :=
  BL  (fun z => restr z j) (productb f)(restriction_product f j).
Lemma prj_axioms: forall f j,
  fgraph f -> sub j (domain f) ->
  transf_axioms  (fun z => restr z j)
  (productb f)(restriction_product f j).
Lemma prj_function: forall f j,
  fgraph f -> sub j (domain f) -> is_function (pr_j f j).
Lemma prj_W: forall f j x,
  fgraph f -> sub j (domain f) -> inc x (productb f)
  -> W x (pr_j f j) =  (restr x j).
Lemma prj_WV: forall f j x i,
  fgraph f -> sub j (domain f) -> inc x (productb f) -> inc i j
  -> V i (W x (pr_j f j))  =  V i x.
```

Propositions 6 and 5 [2, p. 105] state that if $X_\iota$ is nonempty for $\iota \not\in J$, 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 $\mathrm{pr}_J$ is surjective. A special case is when J has a single element $\alpha$. Then $\mathrm{pr}_J$ is the composition of $\mathrm{pr}_\alpha$ and the canonical function that identifies a product of a single set with this set. Thus $\mathrm{pr}_\alpha$ is surjective. A consequence is that if $X_\iota \subset Y_\iota$ then $\prod X_\iota \subset \prod Y_\iota$ (the converse is true if no $X_\iota$ is empty).

```
Theorem extension_partial_product: forall f j g,
  fgraph f -> (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  fgraph g -> domain g = j -> sub j (domain f) ->
  (forall i, inc i j -> inc (V i g) (V i f)) ->
  exists h, domain h = domain f &
    fgraph h & (forall i, inc i (domain f) -> inc (V i h) (V i f)) &
    (forall i, inc i j -> V i h  = V i g).    (* 45 *)
Theorem prj_surjective: forall f j,
  fgraph f -> (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  sub j (domain f) -> surjective (pr_j f j).  (* 15 *)
Lemma pri_surjective: forall f k,
  fgraph f -> (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  inc k (domain f) -> surjective (pr_i f k).     (* 37 *)

Lemma productb_monotone1: forall f g,
  fgraph f -> fgraph g  -> domain f = domain g ->
  (forall i, inc i (domain f) -> sub (V i f) (V i g))
  -> sub (productb f) (productb g).
Lemma productb_monotone2: forall f g,
  fgraph f -> fgraph g  -> domain f = domain g ->
  (forall i, inc i (domain f)  -> nonempty (V i f)) ->
  sub (productb f) (productb g)  ->
  (forall i, inc i (domain f) -> sub (V i f) (V i g)).
```

## 6.5 Associativity of products of sets

Consider a family $X_\iota$. Assume that the index set I is the union of sets $J_\lambda$. For each $\lambda$, we can consider the function $\mathrm{pr}_{J_\lambda}$. 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 $\lambda$ and write it as $(\mathrm{pr}_{J_\lambda} f)_\lambda$. Thus we get a function

$$f \mapsto (\mathrm{pr}_{J_\lambda} f)_{\lambda\in L} \qquad \prod_{\iota\in I} X_\iota \to \prod_{\lambda\in L}\Big(\prod_{\iota\in J_\lambda} X_\iota\Big)$$

It is a bijection if the sets $J_\lambda$ 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_fam g (domain f).

Definition prod_assoc_map f g :=
  BL(fun z => (L (domain g) (fun l => W z (pr_j f (V l g)))))
  (productb f)
  (productf (domain g) (fun l => (restriction_product f (V l g)))).
Lemma pam_axioms: forall f g,
  prod_assoc_axioms f g ->
  transf_axioms(fun z => (L (domain g) (fun l => W z (pr_j f (V l g)))))
  (productb f)
  (productf (domain g) (fun l => (restriction_product f (V l g)))).
Lemma pam_function: forall f g,
  prod_assoc_axioms f g ->
  is_function (prod_assoc_map f g).
```

```
Lemma pam_W: forall f g x,
  prod_assoc_axioms f g -> inc x (productb f) ->
  W x (prod_assoc_map f g) =  (L (domain g) (fun l => W x (pr_j f (V l g)))).
Lemma pam_injective: forall f g,
  prod_assoc_axioms f g ->
  injective(prod_assoc_map f g).
Theorem pam_bijective: forall f g,
  prod_assoc_axioms f g ->
  bijective(prod_assoc_map f g).    (* 59 *)
```

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 $\mathrm{pr}_{I_1}$.

```
Lemma variantLc_prop: forall x y,
  variantLc x y = L two_points (variant TPa x y).
Lemma prod_assoc_map2: forall f g,
  prod_assoc_axioms f g -> domain g = two_points
  -> equipotent (productb f)
  (product (restriction_product f (V TPa g))(restriction_product f (V TPb g))).

Lemma first_proj_bijective: forall x y,
  is_singleton y -> bijective (first_proj (product x y)).
Lemma prj_bijective: forall f j,
  fgraph f -> sub j (domain f) ->
  (forall i, inc i (complement (domain f) j) -> is_singleton (V i f)) ->
  bijective (pr_j f j).     (* 65 *)
```

## 6.6 Distributivity formulae

Let $((X_{\lambda,\iota})_{\iota\in J_\lambda})_{\lambda\in L}$ be a family of families of sets. Let $I = \prod J_\lambda$. We assume that L and I are not 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 $\lambda$ such $x$ is in the intersection over $J_\lambda$, hence $x \in X_{\lambda,\mu}$, whatever $\mu$; in particular it could be $f(\lambda)$. Conversely, Bourbaki assumes that $x$ is not in the LHS; he considers the set $\{\iota \in J_\lambda \mid x \not\in X_{\lambda,\iota}\}$. 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: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> nonempty (domain (V l f))) ->
  unionf (domain f) (fun l => intersectionb (V l f)) =
  intersectionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (unionf (domain f) (fun l => V (V l g) (V l f)))).      (* 25 *)
Lemma distrib_inter_union: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> nonempty (domain (V l f))) ->
  intersectionf (domain f) (fun l => unionb (V l f)) =
  unionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (intersectionf (domain f) (fun l => V (V l g) (V l f)))).
```

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 12 lines for union, and 6 for intersection).

```
Lemma distrib_union2_inter: forall f g,
  fgraph f -> fgraph g -> nonempty (domain f) -> nonempty (domain g) ->
  union2 (intersectionb f)(intersectionb g) =
  intersectionf(product (domain f)(domain g)) (fun z =>
    (union2 (V (P z) f) (V (Q z) g))).
Lemma distrib_inter2_union: forall f g,
  fgraph f -> fgraph g -> nonempty (domain f) -> nonempty (domain g) ->
  intersection2 (unionb f)(unionb g) =
  unionf(product (domain f)(domain g)) (fun z =>
    (intersection2 (V (P z) f) (V (Q z) g))).
```

We say here that the union and intersection of a singleton {X} is X. We also gives additional properties of the empty set. These will be used in the next theorem for exceptional situations.

```
Lemma trivial_product1: forall f, productf emptyset f = singleton emptyset.
Lemma unionf_emptyset: forall f, unionf emptyset f =  emptyset.
Lemma nonempty_product3: forall sf f i,
  inc i sf -> f i = emptyset ->  productf sf f = emptyset.
```

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

$$\prod_{\lambda\in L}\Big(\bigcup_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcup_{f\in I}\Big(\prod_{\lambda\in L} X_{\lambda,f(\lambda)}\Big)$$

$$\prod_{\lambda\in L}\Big(\bigcap_{\iota\in J_\lambda} X_{\lambda,\iota}\Big) = \bigcap_{f\in I}\Big(\prod_{\lambda\in L} X_{\lambda,f(\lambda)}\Big)$$

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: forall f,
  fgraph f ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  productf (domain f) (fun l => unionb (V l f)) =
  unionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (productf (domain f) (fun l => V (V l g) (V l f)))).     (* 30 *)
Theorem distrib_prod_intersection: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> nonempty (domain (V l f))) ->
  productf (domain f) (fun l => intersectionb (V l f)) =
  intersectionf (productf (domain f) (fun l => (domain (V l f))))
  (fun g => (productf (domain f) (fun l => V (V l g) (V l f)))).    (* 29 *)
```

Let $X_\lambda$ 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 $X_\lambda$.

```
Lemma partition_product: forall f,
  fgraph f -> nonempty (domain f) ->
  (forall l, inc l (domain f) -> fgraph (V l f)) ->
  (forall l, inc l (domain f) -> (partition_fam (V l f) (unionb (V l f)))) ->
  partition_fam(L(productf (domain f) (fun l => domain (V l f)) )
    (fun g => (productf (domain f) (fun l => V (V l g) (V l f)))))
  ( productf (domain f) (fun l => unionb (V l f))).    (* 13 *)
```

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: forall f g,
  fgraph f -> fgraph g ->
  nonempty (domain f) -> nonempty (domain g) ->
  product2 (unionb f)(unionb g) =
  unionf(product (domain f)(domain g))
      (fun z =>  (product2 (V (P z) f) (V (Q z) g))).
Lemma distrib_prod2_inter: forall f g,
  fgraph f -> fgraph g ->
  nonempty (domain f) -> nonempty (domain g) ->
  product2 (intersectionb f)(intersectionb g)=
  intersectionf(product (domain f)(domain g)) (fun z =>
    (product2 (V (P z) f) (V (Q z) g))).   (* 16 *)

Lemma distrib_product2_union: forall f g,
  fgraph f -> fgraph g ->
  product (unionb f)(unionb g) =
  unionf(product (domain f)(domain g)) (fun z =>
    (product (V (P z) f) (V (Q z) g))).             (* 27 *)
Lemma distrib_product2_inter: forall f g,
  fgraph f -> fgraph g -> nonempty (domain f) -> nonempty (domain g) ->
  product (intersectionb f)(intersectionb g) =
  intersectionf(product (domain f)(domain g)) (fun z =>
    (product (V (P z) f) (V (Q z) g))).             (* 25 *)
```

Proposition 10 [2, p. 110] says that the intersection of a product is the product of the intersection.

$$\prod_{\iota\in I}\Big(\bigcap_{\kappa\in K} X_{\iota,\kappa}\Big) = \bigcap_{\kappa\in K}\Big(\prod_{\iota\in I} X_{\iota,\kappa}\Big)$$

This is a special case of the general distributivity formula where the set $J_\lambda$ is independent of $\lambda$. 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.

```
Theorem distrib_inter_prod: forall f sa sb,
  fgraph f -> nonempty sb ->
  intersectionf sb (fun k => productf sa (fun i=> V (J i k) f)) =
  productf sa (fun i => intersectionf sb (fun k=> V (J i k) f)).
```

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 productf_extension : forall sf1 f1 sf2 f2,
  L sf1 f1 = L sf2 f2 -> productf sf1 f1 = productf sf2 f2.

Lemma distrib_prod_inter2_prod: forall f g,
  fgraph f -> fgraph g -> domain f = domain g ->
  nonempty (domain f) ->
  intersection2 (productb f) (productb g) =
  productf (domain f) (fun i => intersection2 (V i f) (V i g)).

Lemma distrib_inter_prod_inter: forall f g,
    fgraph f -> fgraph g -> domain f = domain g ->
    nonempty (domain f) ->
    product2 (intersectionb f) (intersectionb g) =
    intersectionf (domain f) (fun i => product2 (V i f) (V i g)). (* 23 *)
```

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 $\prod F_\iota \times \prod F'_\iota$ into $\prod(F_\iota \times F'_\iota)$. We need a bunch of lemmas in order to prove that this mapping is a bijection.

```
Definition prod_of_function x x':=
  L(domain x)(fun i => J (V i x) (V i x')).

Definition prod_of_products_canon f f':=
  BL(fun w => prod_of_function (P w) (Q w))
  (product (productb f) (productb f'))
  (productf (domain f)(fun i => product (V i f) (V i f'))).

Definition prod_of_product_aux f f' :=
  fun i =>  (product (W i f) (W i f')).

Definition prod_of_prod_target f f' :=
  fun_image(source f)(prod_of_product_aux f f').

Definition prod_of_products f f' :=
  BL (prod_of_product_aux f f')(source f)(prod_of_prod_target f f').

Lemma prod_of_prod_inc_target: forall f f' x,
  inc x (prod_of_prod_target f f') =
  (exists i, inc i (source f) & product (W i f) (W i f') = x).
```

```
Lemma prod_of_products_function: forall f f',
  is_function (prod_of_products f f').

Lemma prod_of_products_W: forall f f' i,
  inc i (source f) ->
  W i (prod_of_products f f') =  product (W i f) (W i f').
Lemma prod_of_products_fam_pr: forall f f' x,
  is_function x -> source x = source f ->
  target x = union (prod_of_prod_target f f') ->
  inc (graph x) (productb (graph (prod_of_products f f'))) =
  forall i, inc i (source f) ->
    (is_pair (W i x) &  inc (P (W i x)) (W i f) &  inc (Q (W i x)) (W i f')).
Lemma prod_of_function_axioms:forall x x' f f',
  is_function f -> is_function f' -> source f = source f' ->
  inc (graph x) (productb (graph f)) ->  inc (graph x') (productb (graph f')) ->
  transf_axioms (fun i => J (W i x) (W i x'))
     (source f) (union (prod_of_prod_target f f')).
Lemma prod_of_function_W: forall x x' f f' i,
  is_function f -> is_function f' -> source f = source f' ->
  inc x (productb (graph f)) ->  inc x' (productb (graph f')) ->
  inc i (source f) ->
  V i (prod_of_function x x') =  J (V i x) (V i x').
Lemma prod_of_function_function: forall x x' f f',
  is_function f -> is_function f' -> source f = source f' ->
  inc x (productb (graph f)) ->  inc x' (productb (graph f')) ->
  inc (prod_of_function x x')
  (productb (graph (prod_of_products f f'))).
Lemma popc_target_aux:forall f f',
 is_function f -> is_function f' -> source f = source f' ->
 productb(L (domain (graph f))
   (fun i => product (V i (graph f)) (V i (graph f')))) =
 productb(graph (prod_of_products f f')).
Lemma popc_target:forall f f',
  is_function f -> is_function f' -> source f = source f' ->
  target (prod_of_products_canon (graph f) (graph f')) =
  (productb (graph (prod_of_products f f'))).
Lemma popc_axioms:forall f f',
 is_function f -> is_function f' -> source f = source f' ->
  transf_axioms(fun w => prod_of_function (P w) (Q w))
  (product (productb (graph f)) (productb (graph f')))
  (productb (graph (prod_of_products f f'))).
Lemma popc_W:forall f f' w,
  is_function f -> is_function f' -> source f = source f' ->
  inc w (product (productb (graph f)) (productb (graph f'))) ->
  W w (prod_of_products_canon (graph f) (graph f')) =
  prod_of_function (P w) (Q w).

Lemma popc_bijection: forall f f',
  is_function f -> is_function f' -> source f = source f' ->
  bijective (prod_of_products_canon (graph f) (graph f')).    (* 75 *)
```

## 6.7 Extensions of mappings to products

$$\begin{tikzcd} X_\iota \arrow[r, "f_\iota"] & Y_\iota \\ \prod X_\iota \arrow[u, "\mathrm{pr}_\iota"] \arrow[r, "(f_\iota)_{\iota\in I}"'] & \prod Y_\iota \arrow[u, "\mathrm{pr}_\iota"'] \end{tikzcd}$$ (extension)

Assume that $X_\iota$, $Y_\iota$ and $f_\iota$ are families with the same index I. We assume that $f_\iota$ is a functional graph with source $X_\iota$ and target $Y_\iota$. 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=> V (V i x)  (f i).
Definition ext_map_prod In src trg f :=
  BL (fun x => L In (ext_map_prod_aux x f))
    (productf In src ) (productf In trg ).

Definition ext_map_prod_axioms In src trg f :=
  forall i, inc i In -> (fgraph (f i) & domain (f i) = src i &
    sub (range (f i)) (trg i)).

Lemma ext_map_prod_taxioms: forall In src trg f,
  ext_map_prod_axioms In src trg f ->
  transf_axioms (fun x => L In (ext_map_prod_aux x f))
    (productf In src) (productf In trg).
Lemma ext_map_prod_function: forall In src trg f,
  ext_map_prod_axioms In src trg f -> is_function ( ext_map_prod In src trg f).
Lemma ext_map_prod_W: forall In src trg f x,
  ext_map_prod_axioms In src trg f ->
  inc x (productf In src) ->
  W x (ext_map_prod In src trg f) = L In (ext_map_prod_aux x f).
Lemma ext_map_prod_WV: forall In src trg f x i,
  ext_map_prod_axioms In src trg f ->
  inc x (productf In src) -> inc i In ->
  V i (W x (ext_map_prod In src trg f)) = V (V i x)  (f i).
```

Proposition 11 [2, p. 111] says that composition of extensions is extension of compositions. Bourbaki uses this property to show that if all $f_\iota$ are injective, so is the extension, by exhibiting a left inverse. We use a direct proof because it is easier (note that $f_\iota$ is not a function, just the graph of a function).

```
Lemma ext_map_prod_composable: forall  In p1 p2 p3  g f h,
  ext_map_prod_axioms In p1 p2 f ->
  ext_map_prod_axioms In p2 p3 g ->
  (forall i, inc i In -> h i = fcompose (g i) (f i)) ->
  (forall i, inc i In -> fcomposable (g i) (f i)) ->
  ext_map_prod_axioms In p1 p3 h.

Lemma ext_map_prod_compose: forall  In p1 p2 p3  g f h,
  ext_map_prod_axioms In p1 p2 f ->
  ext_map_prod_axioms In p2 p3 g ->
  (forall i, inc i In -> h i = fcompose (g i) (f i)) ->
  (forall i, inc i In -> fcomposable (g i) (f i)) ->
  compose (ext_map_prod In p2 p3 g)  (ext_map_prod In p1 p2 f) =
  (ext_map_prod In p1 p3 h).      (* 29 *)

Lemma ext_map_prod_injective: forall  In p1 p2 f,
  ext_map_prod_axioms In p1 p2 f ->
  (forall i, inc i In -> injective_graph (f i)) ->
  injective (ext_map_prod In p1 p2 f).
Lemma ext_map_prod_surjective: forall  In p1 p2 f,
  ext_map_prod_axioms In p1 p2 f ->
  (forall i, inc i In -> range (f i) = p2 i) ->
  surjective (ext_map_prod In p1 p2 f).     (* 21 *)
```

Let $f$ be a function from E to A, where A is a product $X_\iota$ over I. Consider the function $\text{pr}_\iota \circ f$ from E to $X_\iota$. 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 $f_\iota$ is a family of functions from E to $X_\iota$, and $\overline{f}$ is its extension to the products, then $\text{pr}_\iota \circ (\overline{f} \circ d) = f_\iota$. The mapping from $f$ to $\overline{f}$ is a bijection between $(\prod X_\iota)^E$ and $\prod X_\iota^E$.

We prove the following facts one after the other. If $f$ is a function from E to A, then $\text{pr}_\iota \circ f$ is a function from E to $X_\iota$. To $g \in A^E$ we associate a function from E to A. The mapping $\iota \mapsto G(\text{pr}_\iota \circ f)$, where G denotes the graph of a function, is an element of $\prod X_\iota^E$. We have a function $A^E \to \prod X_\iota^E$. We define $\overline{f}$. We show $f = \overline{f} \circ d$. We show $\text{pr}_\iota \circ (\overline{f} \circ d) = f_\iota$. We finally show that the mapping is bijective.

```
Definition fun_set_to_prod src F :=
  BL(fun f =>
       L(domain F)( fun i=> (graph (compose (pr_i F i)
          (corresp src (productb F) f)))))
     (set_of_gfunctions src (productb F))
    (productb (L (domain F) (fun i=> set_of_gfunctions src (V i F)))).

Lemma fun_set_to_prod1: forall F f i,
  fgraph F -> inc i (domain F) ->
  is_function f -> target f = productb F ->
  (is_function (compose (pr_i F i) f) &
   source (compose (pr_i F i) f) = source f &
   target (compose (pr_i F i) f) = V i F&
   (forall x, inc x (source f) ->  W x  (compose (pr_i F i) f) = V i (W x f))).

Lemma fun_set_to_prod2: forall src  F f gf,
  fgraph F -> inc gf (set_of_gfunctions src (productb F)) ->
  f = (corresp src (productb F) gf) ->
  (is_function f & target f = productb F & source f = src).
Lemma fun_set_to_prod3: forall src  F,
  fgraph F -> transf_axioms(fun f =>
       L(domain F)( fun i=> (graph (compose (pr_i F i)
          (corresp src (productb F) f)))))
     (set_of_gfunctions src (productb F))
    (productb (L (domain F) (fun i=> set_of_gfunctions src (V i F)))).
Lemma fun_set_to_prod4: forall src  F,
  fgraph F -> ( is_function (fun_set_to_prod src F)
   & source (fun_set_to_prod src F) = (set_of_gfunctions src (productb F))
   & target (fun_set_to_prod src F)
    = (productb (L (domain F) (fun i=> set_of_gfunctions src (V i F))))).
Definition fun_set_to_prod5 F f :=
  ext_map_prod (domain F) (fun i=> source f)(fun i=> V i F)
  (fun i => (graph (compose (pr_i F i) f))).
Lemma fun_set_to_prod6: forall F f,    (* 40 *)
  fgraph F -> is_function f -> target f = productb F ->
  (is_function (fun_set_to_prod5 F f) &
    composable (fun_set_to_prod5 F f) (constant_functor (domain F)(source f) )&
    compose (fun_set_to_prod5 F f) (constant_functor (domain F)(source f)) =f).
Lemma fun_set_to_prod7: forall src F f g,
  fgraph F ->
  (forall i, inc i (domain F) -> inc (f i) (set_of_gfunctions src (V i F))) ->
  g = ext_map_prod (domain F) (fun i=> src)(fun i=> V i F) f ->
  (forall i, inc i (domain F) ->
    f i = graph (compose (pr_i F i) (compose g (constant_functor
           (domain F) src) ))).      (* 67 *)
Lemma fun_set_to_prod8: forall src  F,
  fgraph F -> bijective (fun_set_to_prod src F).      (* 72 *)
```

# Chapter 7

# Equivalence relations

The code of the first two sections of this chapter was originally written by Carlos Simpson[1]. He used *is_relation* where we now write *is_graph*. A relation between two objects $x$ and $y$, often denoted by $x \sim y$, is a function of type $\mathscr{E} \to \mathscr{E} \to$ 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.

## 7.1 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 relation.

```
Definition substrate r :=  union2 (domain r) (range r ).
```

We have some characterizations of the substrate. Only the last one requires that $r$ be a graph.

```
Lemma inc_pr1_substrate : forall r y,  inc y r -> inc (P y) (substrate r).
Lemma inc_pr2_substrate : forall r y,  inc y r -> inc (Q y) (substrate r).
Lemma inc_arg1_substrate: forall r x y,  related r x y -> inc x (substrate r).
Lemma inc_arg2_substrate: forall r x y,  related r x y -> inc y (substrate r).
Lemma substrate_smallest : forall 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 inc_substrate_rw : forall r x,
  is_graph r -> inc x (substrate r) =
  ((exists y, inc (J x y) r) \/  (exists y, inc (J y x) r)).
```

We say that a property $\sim$ is *symmetric* if $a \sim b$ implies $b \sim a$, *transitive* if $a \sim b$ and $b \sim c$ implies $a \sim c$, *reflexive* on $x$ if $a \in x$ is equivalent to $a \sim a$. We say that $\sim$ is an *equivalence relation* if it is symmetric and transitive. We say that it is an *equivalence relation on* E if moreover it is reflexive on E.

```
Definition reflexive_r (r:Set -> Set -> Prop) x :=
  forall y, inc y x = r y y.
```

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

```
Definition symmetric_r (r:Set -> Set -> Prop) :=
  forall x y, r x y -> r y x.
Definition transitive_r (r:Set -> Set -> Prop) :=
  forall x y z, r x y -> r y z -> r x z .
Definition equivalence_r (r:Set -> Set -> Prop) :=
  symmetric_r r & transitive_r r.
Definition equivalence_re (r:Set -> Set -> Prop) x :=
  equivalence_r r & reflexive_r r x.
```

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.[2]

```
Definition is_reflexive r :=
  is_graph r & (forall y, inc y (substrate r) ->related r y y).
Definition is_symmetric r :=
  is_graph r  &  (forall x y, related r x y -> related r y x).
Definition is_transitive r :=
  is_graph r &
  (forall x y z, related r x y -> related r y z -> related r x z).
Definition is_equivalence r :=
  is_reflexive r & is_transitive r &  is_symmetric r.
```

Some trivial consequences of the definitions.

```
Lemma equivalence_is_graph : forall r, is_equivalence r -> is_graph r.
Lemma reflexive_inc_substrate : forall r x,
  is_reflexive r -> inc x (substrate r) = inc (J x x) r.
Lemma reflexive_ap :  forall r x,
  is_reflexive r -> inc x (substrate r) -> related r x x.
Lemma reflexive_ap2 :  forall r x y,
  is_reflexive r ->  related r x y -> related r x x.
Lemma symmetric_ap : forall r x y,
  is_symmetric r -> related r x y -> related r y x.
Lemma transitive_ap : forall r x z,
  is_transitive r ->
  (exists y, related r x y & related r y z) ->
  related r x z.
```

Some lemmas that show that if a graph is symmetric and transitive then it is reflexive.

```
Lemma symmetric_transitive_reflexive : forall r,
  is_symmetric r -> is_transitive r -> is_reflexive r.
Lemma equivalence_relation_pr1 : forall r,
  is_graph r ->
  (forall x y, related r x y -> related r y x) ->
  (forall x y z, related r x y -> related r y z -> related r x z) ->
  is_equivalence r.
Lemma symmetric_transitive_equivalence : forall r,
  is_symmetric r -> is_transitive r -> is_equivalence r.
```

Some trivial properties of an equivalence.[3]

[2]In a previous version, we added the property *is_graph*, that can be deduced from the other ones
[3]Names changed in V3, was reflexivity or transitivity

```
Lemma reflexivity_e : forall r u,
  is_equivalence r -> inc u (substrate r) -> related r u u.
Lemma symmetricity_e : forall r u v,
  is_equivalence r -> related r u v -> related r v u.
Lemma transitivity_e : forall r u v w,
  is_equivalence r ->  related r u v -> related r v w -> related r u w.
```

In the case of an equivalence, the characterization of the substrate is easy.

```
Lemma domain_is_substrate: forall g,
 is_equivalence g -> domain g = substrate g.
Lemma substrate_sub : forall r s,
  sub r s -> sub (substrate r) (substrate s).
Lemma inc_substrate : forall r x,
  is_equivalence r ->
  inc x (substrate r) = (exists y, related r x y).
```

If $r$ has some properties then $\overset{r}{\sim}$ has the corresponding ones.

```
Lemma reflexive_reflexive: forall r,
  is_reflexive r -> reflexive_r (related r) (substrate r).
Lemma symmetric_symmetric: forall r,
  is_symmetric r -> symmetric_r (related r).
Lemma transitive_transitive: forall r,
  is_transitive r -> transitive_r (related r).
Lemma equivalence_equivalence: forall r,
  is_equivalence r -> equivalence_re (related r)(substrate r).
```

We say that $r$ is the graph of $\sim$ if $x \overset{r}{\sim} y$ is the same as $x \sim y$, whatever $x$ and $y$. For every set E, we can define a set $r = g_{\mathrm{E}}(\sim)$, the graph of $\sim$ on E. This is the graph of some relation whose substrate is a subset of E.

```
Definition is_graph_of g (r:Set -> Set -> Prop):=
  forall u v, r u v = related g u v.

Definition graph_on (r:Set -> Set -> Prop) x
  := Zo(product x x)(fun w => r (P w)(Q w)).

Lemma graph_on_graph: forall r x,
  is_graph(graph_on r x).
Lemma graph_on_rw0: forall r x a b,
  inc (J a b) (graph_on r x)  = (inc a x & inc b x & r a b).
Lemma graph_on_rw1: forall r x a b,
  related  (graph_on r x) a b = (inc a x & inc b x & r a b).
Lemma graph_on_substrate: forall r x,
  sub (substrate (graph_on r x)) x.
```

Assume that $\sim$ is an equivalence relation on E. Then $g_{\mathrm{E}}(\sim)$ is the graph of $\sim$. The relation associated to this graph is $\sim$. Assume that $\sim$ is an equivalence relation that has a graph $g$, then it is an equivalence relation on the domain of $g$ (which is also the substrate of $g$). This graph is an equivalence. We shall often use the fact that an equivalence relation on E is associated to an equivalence.

```
Lemma equivalence_has_graph0: forall r x,
  equivalence_re r x -> is_graph_of (graph_on r x) r.
Lemma graph_on_rw2: forall r x u v,
  equivalence_re r x ->
  related (<graph_on r x) u v = r u v.

Lemma equivalence_has_graph: forall r x,
  equivalence_re r x -> exists g, is_graph_of g r.
Lemma equivalence_if_has_graph: forall r g,
  is_graph g ->  is_graph_of g r ->
  equivalence_r r -> equivalence_re r (domain g).
Lemma equivalence_if_has_graph2: forall r g,
  is_graph g ->   is_graph_of g r ->
  equivalence_r r -> is_equivalence g.
Lemma equivalence_has_graph2:forall r x,
  equivalence_re r x -> exists g,
   is_equivalence g & (forall u v, r u v = related g u v).
```

If we take a set E and equality on E (i.e., the relation "$x \in \mathrm{E}$ and $y \in \mathrm{E}$ and $x = y$" between $x$ and $y$), this gives an equivalence relation (it is alo an order). The associated equivalence is the diagonal of E; this is the graph of the identity function. Note that if $r$ is a reflexive graph on E and is functional, it is the identity graph. This is the equivalence with the smallest classes (we shall see that to each equivalence can be associated a partition, and partitions can be compared, so that equivalences can be compared. We have found the finest one).

```
Definition restricted_eq x := fun u => fun v => inc u x & u = v.

Lemma diagonal_equivalence1: forall x,
  equivalence_re(restricted_eq x) x.
Lemma graph_of_restricted_eq: forall x,
  is_graph_of(identity_g x)(restricted_eq x).
Lemma diagonal_equivalence: forall x,
  is_equivalence (identity_g x).
Lemma diagonal_substrate: forall x, substrate(identity_g x) = x.
```

We can consider the relation on E for which all elements are related. Its graph is $\mathrm{E} \times \mathrm{E}$.

```
Definition coarse u := product u u.

Lemma coarse_substrate : forall u, substrate (coarse u) = u.
Lemma coarse_graph: forall x, is_graph (coarse x).

Lemma coarse_related : forall u x y,
  related (coarse u) x y = (inc x u & inc y u).
Lemma coarse_equivalence : forall u,
  is_equivalence (coarse u).


Lemma inter2_is_graph1: forall x y, is_graph x ->
  is_graph (intersection2 x y).
Lemma inter2_is_graph2: forall x y, is_graph y ->
  is_graph (intersection2 x y).
Lemma union2_is_graph: forall x y, is_graph x -> is_graph y ->
  is_graph (union2 x y).
Lemma sub_graph_coarse_substrate: forall r,
  is_graph r -> sub r (coarse (substrate r)).
```

Equipotency is an equivalence relation without a graph.

```
Lemma equipotent_equivalence: equivalence_r equipotent.
```

The fifth example in Bourbaki is: Suppose $\mathrm{A} \subset \mathrm{E}$; then ($x \in \mathrm{E}\backslash\mathrm{A}$ and $y = x$) or ($x \in \mathrm{A}$ and $y \in \mathrm{A}$) is a equivalence on E.

```
Lemma equivalence_relation_bourbaki_ex5: (* 33 *)
  forall a e, sub a e ->
  is_equivalence (
    Zo(product e e)(fun y=>
      (inc (P y)(complement e a) & (P y = Q y)) \/ (inc (P y) a & inc(Q y) a))).
```

Consider a non-empty family of relations $(\sim_i)_{i\in\mathrm{I}}$. We can consider the relation $\forall i, x \sim_i y$. This is equivalence (resp. an equivalence on E) if each $\sim_i$ is an equivalence (resp. an equivalence on E).

```
Lemma inter_rel_graph : forall z,
  nonempty z -> (forall r, inc r z -> is_graph r) ->
  is_graph (intersection z).
Lemma inter_rel_rw : forall z x y,
  nonempty z ->
  related (intersection z) x y =
  (forall r, inc r z -> related r x y).
Lemma inter_rel_reflexive : forall z,
  nonempty z -> (forall r, inc r z -> is_reflexive r) ->
  is_reflexive (intersection z).
Lemma inter_rel_substrate : forall z e,
  nonempty z -> (forall r, inc r z -> is_reflexive r) ->
  (forall r, inc r z -> substrate r = e) ->
  substrate (intersection z) = e.
Lemma inter_rel_transitive : forall z,
  nonempty z -> (forall r, inc r z -> is_transitive r) ->
  is_transitive (intersection z).
Lemma inter_rel_symmetric : forall z,
  nonempty z -> (forall r, inc r z -> is_symmetric r) ->
  is_symmetric (intersection z).
Lemma inter_rel_equivalence : forall z,
  nonempty z -> (forall r, inc r z -> is_equivalence r) ->
  is_equivalence (intersection z).
```

We can consider the set of all equivalences on E. It is not empty.

```
Definition all_relations x :=  powerset (product x x).
Definition all_equivalence_relations x :=
  Zo (all_relations x) (fun r => (is_equivalence r)
    & (substrate r = x)).

Lemma inc_all_relations : forall r x,
  inc r (all_relations x) = (is_graph r & sub (substrate r) x).
Lemma inc_all_equivalence_relations : forall r x,
  inc r (all_equivalence_relations x) =
  (is_equivalence r & (substrate r = x)).
Lemma inc_coarse_all_equivalence_relations : forall u,
  inc (coarse u) (all_equivalence_relations u).
```

Proposition 1 [2, p. 114] says that a correspondence $\Gamma$ between X and X is an equivalence on X if and only if X is the domain of $\Gamma$, $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$. We prove this property for graphs rather than correspondences.

```
Lemma selfinverse_graph_symmetric: forall r,
  (is_symmetric r = (r= inverse_graph r)).
Lemma idempotent_graph_transitive: forall r,
  is_graph r-> (is_transitive r = sub  (compose_graph r r) r).
Theorem equivalence_pr: forall r,
  (is_equivalence r  = ((compose_graph r r) = r &  r= inverse_graph r)).
```

## 7.2 Equivalence classes; quotient set

Let $f$ be a function on E; the relation $f(x) = f(y)$ is an equivalence relation on E. It has a graph $\mathrm{F}^{-1} \circ \mathrm{F}$, where F is the graph of $f$. We shall denote it by $\sim_f$.

```
Definition eq_rel_associated f :=
  fun x => fun y => (inc x (source f) & (inc y (source f)) & (W x f = W y f)).
Definition equivalence_associated f :=
  compose_graph (inverse_graph (graph f)) (graph f).

Lemma ea_equivalence: forall f,
  is_function f -> equivalence_re(eq_rel_associated f)(source f).
Lemma graph_of_ea: forall f,
  is_function f ->
  is_graph_of (equivalence_associated f) (eq_rel_associated f).
Lemma graph_ea_equivalence: forall f,
  is_function f->
  is_equivalence (equivalence_associated f).
Lemma graph_ea_substrate: forall f,
  is_function f->
  substrate (equivalence_associated f) = source f.
Lemma ea_related:forall f x y,
  is_function f ->
  related (equivalence_associated f) x y =
  (inc x (source f) & inc y (source f) &  W x f = W y f).
```

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 $(x, y) \in \mathrm{G}$ (also known as the cut of G at $x$) 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 $\mathrm{G}_x$ 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:Set) := fun_image (Zo r (fun z => P z = x)) Q.


Lemma inc_class : forall r x y,
  is_equivalence r -> inc y (class r x) = related r x y.
Lemma inc_class0 : forall r x y,
  is_equivalence r -> inc y (class r x) = related r x y.
Lemma class_is_cut: forall r x, is_equivalence r ->
  class r x = im_singleton r x.
Lemma sub_class_substrate: forall r x,
  is_equivalence r -> sub(class r x) (substrate r).
Lemma nonempty_class_symmetric : forall r x,
  is_equivalence r ->
  nonempty (class r x) = inc x (substrate r).
Lemma related_class_eq1: forall r u v, is_equivalence r ->
  related r u v -> class r u = class r v.
Lemma related_class_eq : forall r u v,
  is_equivalence r ->
  related r u u ->
  related r u 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 that there is a class $x$ such that $a \in x$ and $b \in x$. If $x \in$ E/R and $y \in x$ then $\hat{x} \overset{\mathrm{R}}{\sim} y$.

```
Definition quotient r := fun_image (substrate r) (class r).
Definition is_class r x := is_equivalence r
  & inc (rep x) (substrate r)  & x = class r (rep x).

Lemma inc_rep_itself:forall r x,  is_equivalence r ->
  inc x (quotient r) -> inc (rep x) x.
Lemma non_empty_in_quotient: forall r x,
  is_equivalence r -> inc x (quotient r) -> nonempty x.
Lemma is_class_class : forall r x, is_equivalence r ->
  inc x (substrate r) -> is_class r (class r x).
Lemma inc_quotient : forall r x,  is_equivalence r ->
  inc x (quotient r) = is_class r x.
Lemma in_class_related : forall r y z,
  is_equivalence r ->
  related r y z = (exists x, is_class r x & inc y x & inc z x).
Lemma related_rep_in_class:forall r x y,
  is_equivalence r -> 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 is_class_rw : forall r x, is_equivalence r ->
  is_class r x = (nonempty x & sub x (substrate r) &
    (forall y z, inc y x -> (inc z x = related r y z)) ).
Lemma class_dichot : forall r x y,
  is_class r x -> is_class r y -> (x = y \/ disjoint x y).
Lemma inc_class_quotient : forall r x,
  is_equivalence r -> inc x (substrate r) ->
  inc (class r x) (quotient r).
```

```
Lemma inc_in_quotient_substrate : forall r x y,
  is_equivalence r -> inc x y -> inc y (quotient r)
  -> inc x (substrate r).
Lemma union_quotient : forall r,
  is_equivalence r ->   union (quotient r) = substrate r.
Lemma inc_rep_substrate : forall r x, is_equivalence r ->
  inc x (quotient r) -> inc (rep x) (substrate r).
Lemma class_rep : forall r x, is_equivalence r ->
  inc x (quotient r) -> class r (rep x) = x.
Lemma inc_itself_class : forall r x,
  is_equivalence r ->inc x (substrate r) ->  inc x (class r x).
Lemma related_rep_class : forall r x, is_equivalence r ->
  inc x (substrate r) -> related r x (rep (class r x)).
Lemma related_rep_rep : forall r u v,
  is_equivalence r -> inc u (quotient r) -> inc v (quotient r) ->
  related r (rep u) (rep v) = (u = v).
Lemma related_rw : forall r u v,
  is_equivalence r -> related r u v =
  (inc u (substrate r) & inc v (substrate r) & class r u = class r v).
Lemma is_class_pr: forall r x y,
  is_equivalence r -> inc x y -> inc y (quotient r)
  -> y = class r x.
```

The *canonical projection* is the mapping $x \mapsto \bar{x}$ from E onto E/R. An important property is that this function is surjective[4].

```
Definition canon_proj(r:Set):= BL(fun x=> class r x)
  (substrate r) (quotient r).

Lemma canon_proj_function: forall r,
  is_equivalence r -> is_function (canon_proj r).
Lemma canon_proj_W: forall r x,
  is_equivalence r ->
  inc x (substrate r) -> W x (canon_proj r) = class r x.
Lemma related_graph_canon_proj: forall r x y,
  is_equivalence r -> inc x (substrate r) -> inc y (quotient r) ->
  inc (J x y) (graph (canon_proj r)) = inc x y.
Lemma canon_proj_show_surjective:forall r x,
  is_equivalence r-> inc x (quotient r)
  ->W (rep x)(canon_proj r) =x.

Lemma canon_proj_surjective:forall r,
  is_equivalence r-> surjective (canon_proj r).
```

The next lemma says that if $\mathrm{A} \subset \mathrm{E}$ and $x \subset \bar{\mathrm{A}}$ (where $\bar{\mathrm{A}}$ is the set of all $\bar{a}$ for $a \in \mathrm{A}$) then $x \in \mathrm{E/R}$. We then state Criterion 55 [2, p. 115]: $u \overset{\mathrm{R}}{\sim} v$ 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 $\mathrm{R}\{x,y\} \iff (p(x) = p(y))$". The correct statement would be: $\mathrm{R}\{x,y\}$ if and only if $x \in \mathrm{E}$ and $y \in \mathrm{E}$ and $p(x) = p(y)$. The proof is a bit strange. It starts with: "let $x$ and $y$ be elements of E such that $(x, y) \in \mathrm{G}$. Then $x \in \mathrm{E}$ and $y \in \mathrm{E}$; let us show..."

```
Lemma sub_im_canon_proj_quotient: forall r a x,
  is_equivalence r -> sub a (substrate r) ->
```

[4]The case where R is the equivalence associated to a correspondence is not considered anymore in Version 3.

```
  inc x (image_by_fun (canon_proj r) a) ->
  inc x (quotient r).
Lemma related_e_rw: forall r u v,
 is_equivalence r -> (related r u v =
   (inc u (source (canon_proj r)) &
     inc v (source (canon_proj r)) &
     W u (canon_proj r) = W v (canon_proj r))).
```

If we consider the equivalence associated with the equality on a set E then each class is a singleton and E/R is equipotent to E.

```
Lemma diagonal_class: forall x u,
  inc u x -> class (identity_g x) u = singleton u.
Lemma canon_proj_diagonal_bijective: forall x,
  bijective (canon_proj (identity_g x)).
```

Consider now the equivalence associated with $\mathrm{pr}_1$. We consider a product $E \times F$ and the relation $(x, y) \sim (x, z)$ for every $x$, $y$ and $z$. This is an equivalence on $E \times F$, and 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_eqr (x y:Bet) :=
  eq_rel_associated(first_proj (product x y)).

Definition first_proj_eq (x y :Set) :=
  equivalence_associated (first_proj (product x y)).

Lemma first_proj_eq_pr: forall x y a b,
  first_proj_eqr x y a b =
  (inc a (product x y) & inc b (product x y) & P a = P b).
Lemma first_proj_graph: forall x y,
  is_graph_of(first_proj_eq x y)(first_proj_eqr x y).
Lemma first_proj_equivalence: forall x y,
  is_equivalence (first_proj_eq x y).
Lemma first_proj_eq_related: forall x y a b,
  related (first_proj_eq x y) a b =
  (inc a (product x y) & inc b (product x y) & P a = P b).
Lemma first_proj_substrate: forall x y,
  substrate(first_proj_eq x y) = product x y.
Lemma first_proj_class:forall x y z,
  nonempty y ->
  is_class (first_proj_eq x y) z =
  exists u, inc u x & z = product (singleton u) y.   (* 22 *)
Lemma first_proj_equiv_proj:  forall x y,
  nonempty y->
  bijective (BL (fun u => product (singleton u) y)
    x (quotient (first_proj_eq x y))).
```

For any equivalence, the quotient is a partition of the substrate.

```
Lemma sub_quotient_powerset: forall r,
  is_equivalence r -> sub (quotient r) (powerset (substrate r)).
Lemma partition_from_equivalence: forall r,
  is_equivalence r ->
  partition(quotient r)(substrate r).
```

We consider now the converse. Let $f$ be a function and F its graph. Assume that F is a partition of X. We shall write $X_i$ instead of $f(i)$. 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 has a graph on X, say $r$. Then $r$ is an equivalence on X. We have $a \in \overline{b}$ (i.e., $a$ and $b$ are related by $r$) if and only if there is an $i$ such that $a \in X_i$ and $b \in X_i$. If a set $a$ is a class of $r$, then it has the form $X_i$. The converse is true if $X_i$ is not empty. In other words, the image of $f$ is the quotient $X/r$. We restate it as follows: if $f$ is a function and the graph of $f$ is a true partition of X, 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 (W i f) & inc y (W i f).

Definition partition_relation f x :=
  graph_on (in_same_coset f) x.

Lemma partition_inc_unique1: forall f x i j y,
  is_function f -> partition_fam (graph f) x ->
  inc i (source f) -> inc y (W i f) ->
  inc j (source f) -> inc y (W j f) -> i = j.

Lemma isc_reflexive: forall f x, is_function f ->
  partition_fam (graph f) x -> reflexive_r (in_same_coset f) x.
Lemma isc_symmetric: forall f,  symmetric_r (in_same_coset f).
Lemma isc_transitive: forall f x, is_function f ->
  partition_fam (graph f) x -> transitive_r (in_same_coset f).
Lemma isc_equivalence: forall f x, is_function f ->
  partition_fam (graph f) x -> equivalence_re (in_same_coset f) x.
Lemma partition_rel_graph: forall f x,
  is_function f ->  partition_fam (graph f) x ->
  is_graph_of (partition_relation f x) (in_same_coset f).
Lemma partition_relation_pr: forall f x a b,
  is_function f -> partition_fam (graph f) x ->
  related (partition_relation f x) a b = in_same_coset f a b.
Lemma partition_is_equivalence: forall f x,
  is_function f -> partition_fam (graph f) x ->
  is_equivalence (partition_relation f x).
Lemma partition_relation_substrate: forall f x,
  is_function f -> partition_fam (graph f) x ->
  substrate (partition_relation f x)= x.
Lemma partition_relation_class: forall f x a,
  is_function f -> partition_fam (graph f) x ->
  is_class (partition_relation f x) a
  -> exists u, inc u (source f) & a = W u f.
Lemma partition_relation_class2: forall f x u,
  is_function f -> partition_fam (graph f) x ->
  inc u (source f) -> nonempty (W u f)
  -> is_class (partition_relation f x)  (W u f).
Lemma partition_fun_bijective: forall f x,
  is_function f -> partition_fam (graph f) x ->
  (forall u, inc u (source f) -> nonempty (W u f))
  -> bijective (BL (fun u => W u f)
    (source f) (quotient (partition_relation f x))).
```

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 :=
   is_function f & partition_fam (graph f) x & sub s x
  & (forall i, inc i (source f) -> is_singleton (intersection2 (W i f) s)).

Definition representative_system_function g f x :=
  injective g & (representative_system (range (graph g)) f x).

Lemma rep_sys_function_pr: forall g f x i,
  representative_system_function g f x -> inc i (source f)
  -> exists_unique (fun a=> inc a (source g) & inc (W a g) (W i f)).
Lemma rep_sys_function_pr2: forall g f x,
  injective g -> is_function f -> partition_fam (graph f) x -> sub (target g) x
  -> (forall i, inc i (source f)
    -> exists_unique (fun a=> inc a (source g) & inc (W a g) (W i f)))
  -> representative_system_function g f x.
Lemma section_canon_proj_pr: forall g f x y r,
  r = partition_relation f x ->  is_function f -> partition_fam (graph f) x
  -> is_right_inverse g (canon_proj r) ->
  inc y x ->
  related r y (W (class r y) g). (* 15 *)
Lemma section_is_representative_system_function: forall g f x,
  is_function f -> partition_fam (graph f) x
  -> is_right_inverse g (canon_proj (partition_relation f x)) ->
  (forall u, inc u (source f) -> nonempty (W u f)) ->
  representative_system_function g f x.    (* 33 *)
```

## 7.3 Relations compatible with an equivalence relation

We say that P($x$) is *compatible* with ∼ if P($x$) and $x \sim y$ imply P($y$). Every property is compatible with the equality.

```
Definition compatible_with_equiv_p (p: Set -> Prop)(r:Set) :=
  forall x x', p x -> related r x x' -> p x'.

Lemma trivial_equiv: forall p x,
  compatible_with_equiv_p p (diagonal x).
```

If $p$ is compatible with ∼, we can define P($t$) on the quotient E/R of ∼ by:

$$t \in E/R \text{ and } (\exists x)(x \in t \text{ and } P\{x\})$$

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 $p(x)$. This is Criterion C56. 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) & exists x, inc x t & p x.
```

```
Lemma rel_on_quo_pr: forall  p r t,
  is_equivalence r -> compatible_with_equiv_p p r ->
  relation_on_quotient p r t = (inc t (quotient r) &  forall x, inc x t -> p x).
Lemma rel_on_quo_pr2: forall p r y,
  is_equivalence r -> compatible_with_equiv_p p r ->
  (inc y (substrate r) & relation_on_quotient p r (W y (canon_proj r))) =
  (inc y (substrate r) & p y).
```

## 7.4 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:Set) := compatible_with_equiv_p (fun y=> inc y x) r.

Lemma saturated_pr: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  (saturated r x) =  (forall y, inc y x -> sub (class r y) x).
Lemma saturated_pr2: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  (saturated r x) =  exists y, (forall z, inc z y -> is_class r z)
    & x = union y.
```

Given a function $f$ and a set X, we consider $X_f$ to be $f^{-1}\langle f\langle X\rangle\rangle$. We have $y \in X_f$ 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 class_is_inv_direct_value: forall r x,
  is_equivalence r -> inc x (substrate r) ->
  class r x = inverse_direct_value (canon_proj r) (singleton x).   (* 21 *)
Lemma saturated_pr3: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  saturated r x =  (x= inverse_direct_value (canon_proj r) x).  (* 30 *)
Lemma saturated_pr4: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  saturated r x =  (exists 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_union: forall r x,
  is_equivalence r -> (forall y, inc y x -> sub y (substrate r)) ->
  (forall y, inc y x -> saturated r y) ->
  (sub (union x) (substrate r) &  saturated r (union x)).
Lemma saturated_intersection : forall r x,
  is_equivalence r -> nonempty x ->
  (forall y, inc y x -> sub y (substrate r)) ->
  (forall y, inc y x -> saturated r y) ->
  (sub (intersection x) (substrate r) &  saturated r (intersection x)).
Lemma saturated_complement : forall r a,
  is_equivalence r -> sub a (substrate r) -> saturated r a ->
  saturated r (complement (substrate r) 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:Set):=
  inverse_direct_value (canon_proj r) x.
Lemma saturation_of_pr: forall r x,
  is_equivalence r -> sub x (substrate r) ->
  saturation_of r x =
  union (Zo (quotient r)(fun z=> exists i, inc i x & z = class r i)).   (* 21 *)
Lemma saturation_of_smallest: forall r x,
  is_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)).      (* 24 *)

Definition union_image x f:=
  union (Zo x (fun z=> exists i, inc i (source f) & z = W i f)).

Lemma saturation_of_union: forall r f g,
  is_equivalence r -> is_function f -> is_function g ->
  (forall i, inc i (source f) -> sub (W i f) (substrate r)) ->
  source f = source g ->
  (forall i, inc i (source f) -> saturation_of r (W i f) = W i g)
  -> saturation_of r (union_image (powerset(substrate r)) f) =
  union_image (powerset(substrate r)) g.     (* 20 *)
```

## 7.5 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:Set) :=
  BL rep (quotient r) (substrate r).
Lemma section_canon_proj_axioms:forall r,
  is_equivalence r ->
  transf_axioms rep (quotient r) (substrate r).
Lemma section_canon_proj_W: forall r x,
  is_equivalence r ->
  inc x (quotient r) -> W x (section_canon_proj r) = (rep x).
Lemma function_section_canon_proj: forall r,
  is_equivalence r -> is_function (section_canon_proj r).
Lemma right_inv_canon_proj: forall r,
  is_equivalence r ->
  is_right_inverse (section_canon_proj r) (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 $x \overset{R}{\sim} x'$ 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 64).

```
Definition compatible_with_equiv f r :=
  is_function f & source f = substrate r  &
  forall x x', related r x x' -> W x f = W  x' f.

Lemma compatible_with_equiv_pr: forall f r,
  is_function f -> source f = substrate r  ->
  compatible_with_equiv f r =
  (forall y, compatible_with_equiv_p (fun x => y = W x f) r).
Lemma compatible_constant_on_classes: forall f r x y,
  is_equivalence r ->
  compatible_with_equiv f r -> inc y (class r x) -> W x f = W y f.
Lemma compatible_constant_on_classes2: forall f r x,
  is_equivalence r -> compatible_with_equiv f r ->
  is_constant_function(restriction f (class r x)).
Lemma compatible_with_proj: forall f r x y,
  is_equivalence r -> compatible_with_equiv f r ->
  inc x (substrate r) -> inc y (substrate r) ->
  W x (canon_proj r) = W y (canon_proj r) -> W x f = W y f.
```

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 $F/s$. 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' :=
  is_function f & target f =  substrate r' &
  compatible_with_equiv (compose (canon_proj r') f) r.
Lemma compatible_with_pr:forall r r' f x y,
  is_equivalence r ->  is_equivalence r' ->
  compatible_with_equivs f r r' ->
  related r x y -> related r' (W x f) (W y f).
Lemma compatible_with_pr2:forall r r' f,
  is_equivalence r ->  is_equivalence r' ->
  is_function f ->
  target f = substrate r'-> source f = substrate r->
  (forall x y,  related r x y -> related r' (W x f) (W y f)) ->
  compatible_with_equivs f r r'.
Lemma compatible_with_proj3 :forall r r' f x y,
  is_equivalence r -> is_equivalence r' ->
  compatible_with_equivs f r r'->
  inc x (substrate r) -> inc y (substrate r) ->
  W x (canon_proj r) = W y (canon_proj r) ->
  class r' (W x f) = class r' (W y f).
```

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 216).

```
Definition fun_on_quotient r f :=
  compose f (section_canon_proj r).

Lemma exists_fun_on_quotient: forall f r,
  is_equivalence r -> is_function f -> source f = substrate r ->
  compatible_with_equiv f r =
  (exists h, composable  h (canon_proj r) & compose h (canon_proj r) = f).
Lemma exists_unique_fun_on_quotient: forall f r h,
  is_equivalence r -> compatible_with_equiv f r ->
  composable  h (canon_proj r) -> compose h (canon_proj r) = f ->
  h = fun_on_quotient r f.
Lemma compose_foq_proj :forall f r,
  is_equivalence r ->  compatible_with_equiv f r ->
  compose (fun_on_quotient r f)  (canon_proj r) = f.
```

$$\begin{tikzcd} E \arrow[r, "f"] \arrow[d, "\pi"'] & E' \\ E/r \arrow[ur, "f'"'] & \end{tikzcd} \qquad \begin{tikzcd} E \arrow[r, "f"] \arrow[d, "\pi"'] & E' \arrow[d, "\pi'"] \\ E/r \arrow[r, "f''"'] & E'/r' \end{tikzcd}$$

(fun on quotient)

Assume that $f$ is a function from E into E$'$ on which we have equivalence relations $r$ and $r'$. Let $\pi$ and $\pi'$ 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=> W (f(rep x)) (canon_proj r’).
Definition function_on_quotient r f b  :=
  BL(fun_on_rep f)(quotient r)(b).
Definition function_on_quotients r r’ f :=
  BL(fun_on_reps r’ f)(quotient r)(quotient r’).
Definition fun_on_quotients r r’ f :=
  compose (compose (canon_proj r’) f) (section_canon_proj r).
Lemma foq_axioms: forall r f b, X->
  transf_axioms f (substrate r) b ->
  transf_axioms (fun_on_rep f) (quotient r) b.
Lemma foqs_axioms: forall r r’ f, X -> X’ ->
  transf_axioms f (substrate r)(substrate r’) ->
  transf_axioms (fun_on_reps r’ f) (quotient r) (quotient r’).
Lemma foqc_axioms: forall r f, X->
  is_function f -> source f = substrate r ->
  composable f (section_canon_proj r).
Lemma foqcs_axioms:forall r r’ f, X-> X’->
  is_function f -> source f = substrate r -> target f = substrate r’ ->
  composable (compose (canon_proj r’) f) (section_canon_proj r).

Lemma foq_function:forall r f b, X->
  transf_axioms f (substrate r) b ->
  is_function (function_on_quotient r f b).
Lemma foqs_function: forall r r’ f, X-> X’ ->
  transf_axioms f (substrate r)(substrate r’) ->
  is_function (function_on_quotients r r’ f).
```

```
Lemma foqc_function: forall r f, X-> X' ->
  source f = substrate r ->
  is_function (fun_on_quotient r f).
Lemma foqcs_function:forall r r' f, X-> X' ->
  is_function f -> source f = substrate r -> target f = substrate r' ->
  is_function (fun_on_quotients r r' f).
Lemma foq_W:forall r f b x, X->
  transf_axioms f (substrate r) b ->
  inc x (quotient r) ->
  W x (function_on_quotient r f b) = f (rep x).
Lemma foqc_W:forall r f x, X ->
   is_function f ->
  source f = substrate r -> inc x (quotient r) ->
  W x (fun_on_quotient r f) = W (rep x) f.
Lemma foqs_W: forall r r' f x, X -> X' ->
  transf_axioms f (substrate r)(substrate r') ->   inc x (quotient r) ->
  W x (function_on_quotients r r' f) = class r' (f (rep x)).
Lemma foqcs_W:forall r r' f x, X-> X' ->
  is_function f -> source f = substrate r -> target f = substrate r' ->
  inc x (quotient r) ->
  W x (fun_on_quotients r r' f) = class r' (W (rep x) f).
```

More lemmas; statement *fun_on_quotient_pr4* is the diagram on the right part of (fun on quotient) on page 123.

```
Lemma fun_on_quotient_pr: forall r f x,
  W x f = fun_on_rep (fun w => W x f) (W x (canon_proj r)).
Lemma fun_on_quotient_pr2: forall r r' f x,
  W (W x f) (canon_proj r') =
  fun_on_reps r' (fun w => W x f) (W x (canon_proj r)).
Lemma composable_fun_proj: forall r f b, X->
  transf_axioms f (substrate r) b ->
  composable (function_on_quotient r f b) (canon_proj r).
Lemma composable_fun_projs: forall r r' f, X -> X' ->
  transf_axioms f (substrate r) (substrate r') ->
  composable (function_on_quotients r r' f) (canon_proj r).
Lemma composable_fun_projc: forall r f, X->
  compatible_with_equiv f r ->
  composable (fun_on_quotient r f) (canon_proj r).
Lemma composable_fun_projcs: forall r r' f, X-> X' ->
  compatible_with_equivs f r r'->
  composable (fun_on_quotients r r' f) (canon_proj r).
Lemma fun_on_quotient_pr3: forall r f x, X->
  inc x (substrate r) ->   compatible_with_equiv f r ->
  W x f = W (W x (canon_proj r)) (fun_on_quotient r f).
Lemma fun_on_quotient_pr4: forall r r' f, X-> X' ->
  compatible_with_equivs f r r'->
  compose (canon_proj r') f = compose (fun_on_quotients r r' f)(canon_proj r).
Lemma fun_on_quotient_pr5: forall r r' f x, X-> X'->
  compatible_with_equivs f r r'->
  inc x (substrate r) ->
  W (W x f) (canon_proj r') =
  W (W x (canon_proj r)) (fun_on_quotients r r' f).
Lemma compose_fun_proj_ev: forall r f b x, X->
  compatible_with_equiv (BL f (substrate r) b) r ->
  inc x (substrate r) ->
  transf_axioms f (substrate r) b ->
  W x (compose (function_on_quotient r f b) (canon_proj r)) = f x.
Lemma compose_fun_proj_ev2: forall r r’ f x, X-> X’ ->
  compatible_with_equivs (BL f (substrate r) (substrate r’)) r r’ ->
  transf_axioms f (substrate r) (substrate r’) ->
  inc x (substrate r) ->
  inc (f x) (substrate r’) ->
  W (f x) (canon_proj r’) =
  W x (compose (function_on_quotients r r’ f) (canon_proj r)).
Lemma compose_fun_proj_eq: forall r f b, X->
  compatible_with_equiv (BL f (substrate r) b) r ->
  transf_axioms f (substrate r) b ->
  compose (function_on_quotient r f b) (canon_proj r) =
    BL f (substrate r) b.
Lemma compose_fun_proj_eq2: forall r r’ f, X-> X’ ->
  transf_axioms f (substrate r) (substrate r’) ->
  compatible_with_equivs (BL f (substrate r) (substrate r’)) r r’->
  compose (function_on_quotients r r’ f) (canon_proj r) =
  compose (canon_proj r’) (BL f (substrate r) (substrate r’)).
```

$$\begin{tikzcd} \mathrm{E} \arrow[r, "f"] \arrow[d, "\pi"'] & \mathrm{F} \\ \mathrm{E}/\sim \arrow[r, "\equiv", "f'"'] & \mathrm{F}' \arrow[u, "\subset"'] \end{tikzcd} \qquad \begin{tikzcd} \mathrm{E} \arrow[r, "f"] \arrow[d, "\pi"'] & \mathrm{F} \\ \mathrm{E}/\sim \arrow[ur, "\bar f"'] & \end{tikzcd}$$

(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/ ∼, 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: forall f,
  is_function f ->
  compatible_with_equiv f (equivalence_associated f).
Lemma ea_foq_injective: forall f,
  is_function f ->
  injective (fun_on_quotient (equivalence_associated f) f).
Lemma ea_foq_on_im_bijective: forall f,
  is_function f ->
  bijective (restriction2 (fun_on_quotient (equivalence_associated f) f)
    (quotient (equivalence_associated f)) (range (graph f))).   (* 28 *)
Lemma canonical_decompositiona: forall f,
  is_function f ->
  let r:= equivalence_associated f in
    is_function (compose (restriction2 (fun_on_quotient r f)
      (quotient r) (range (graph f)))
    (canon_proj r)).  (* 23 *)
Lemma canonical_decomposition: forall f,
  is_function f ->
  let r:= equivalence_associated f in
  f = compose (canonical_injection (range (graph f))(target f))
  (compose (restriction2 (fun_on_quotient r f)
    (quotient r) (range (graph f)))
  (canon_proj r)). (* 41 *)
Lemma surjective_pr7:  forall f,
  surjective f ->
  canonical_injection (range (graph f))(target f) = identity (target f).
Lemma canonical_decompositiona: forall f,
  is_function f ->
  let r:= equivalence_associated f in
    is_function (compose (restriction2 (fun_on_quotient r f)
      (quotient r) (range (graph f)))
    (canon_proj r)).
Lemma canonical_decomposition_surj: forall f,
  surjective f ->
  let r:= equivalence_associated f in
  f =  compose (restriction2 (fun_on_quotient r f) (quotient r) (target f))
      (canon_proj r).
Lemma canonical_decompositionb: forall f,
  is_function f ->
  let r:= equivalence_associated f in
    restriction2 (fun_on_quotient r f)  (quotient r) (target f) =
    (fun_on_quotient r f).
Lemma canonical_decomposition_surj2: forall f,
  surjective f ->
  let r:= equivalence_associated f in
  f =  (compose (fun_on_quotient r f) (canon_proj r)).
```

## 7.6 Inverse image of an equivalence relation; induced equivalence relation

If $\phi$ 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 $\phi$ 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 (compose (canon_proj r) f).
Definition iirel_axioms f r :=
  is_function f & is_equivalence r & substrate r = target f.

Lemma iirel_function: forall f r,
  iirel_axioms f r -> is_function (compose (canon_proj r) f).
Lemma iirel_relation: forall f r,
  iirel_axioms f r -> is_equivalence (inv_image_relation f r).
Lemma iirel_substrate: forall f r,
  iirel_axioms f r -> substrate (inv_image_relation f r) = source f.
Lemma iirel_related: forall f r x y,
  iirel_axioms f r ->
  related (inv_image_relation f r) x y =
  (inc x (source f) & inc y (source f) &  related r (W x f) (W y f)).
Lemma iirel_class: forall f r x,
  iirel_axioms f r ->
  is_class (inv_image_relation f r) x =
  exists y, is_class r y
    & nonempty (intersection2 y (range (graph f)))
    & x = inv_image_by_fun f y.    (* 40 *)
```

$$\begin{tikzcd} A \arrow[rr, "j"] \arrow[d, "g"'] & & E \arrow[d, "f"] \\ A/R_A \arrow[r, "k"] \arrow[rr, bend right, "h"'] & \mathrm{I}mf \arrow[r, "\subset"] & E/R \end{tikzcd}$$

(induced equivalence)

Let R be an equivalence on E, A a subset on E, and $j$ the inclusion map A → 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:Set) :=
  inv_image_relation (canonical_injection a (substrate r)) r.
Definition induced_rel_axioms(r a :Set) :=
   is_equivalence r & sub a (substrate r).

Lemma induced_rel_iirel_axioms : forall r a,
  induced_rel_axioms r a ->
  iirel_axioms (canonical_injection a (substrate r)) r.
Lemma induced_rel_equivalence: forall r a,
  induced_rel_axioms r a -> is_equivalence (induced_relation r a).
Lemma induced_rel_substrate: forall r a,
  induced_rel_axioms r a -> substrate (induced_relation r a) = a.
Lemma induced_rel_related: forall r a u v,
  induced_axioms_rel r a ->
  related (induced_relation r a) u v =
  (inc u a & inc v a & related r u v).
Lemma induced_rel_class: forall r a x,
  induced_rel_axioms r a ->
  is_class (induced_relation r a) x =
  exists y, is_class r y
    & nonempty (intersection2 y a)
    & x = (intersection2 y a).
Lemma compatible_injection_induced_rel: forall r a,
  induced_rel_axioms r a ->
  compatible_with_equivs (canonical_injection a (substrate r))
  (induced_relation r a) r.
Lemma foq_induced_rel_injective: forall r a,
  induced_rel_axioms r a ->
  injective (fun_on_quotients (induced_relation r a) r
    (canonical_injection a (substrate r))).  (* 21 *)
Lemma foq_induced_rel_image: forall r a,
  induced_axioms_rel r a ->
  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.  (* 47 *)
```

```
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_bijective: forall r a,
  induced_rel_axioms r a -> bijective (canonical_foq_induced_rel r a). (* 15 *)
```

## 7.7 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 $C_s x$ the class of $x$ for $s$, it is also: for each $x$, there is an $y$ such that $C_s x \subset C_r y$. Equivalently: each $C_r y$ is saturated by $s$. We give two examples.

```
Definition finer_equivalence(s r:Set):=
  forall x y, related s x y -> related r x y.

Definition finer_axioms(s r:Set):=
  is_equivalence s & is_equivalence r & substrate r = substrate s.

Lemma finer_sub_equiv: forall s r,
  finer_axioms s r ->
  (finer_equivalence s r) = (sub s r).
Lemma finer_sub_equiv2: forall s r,
  finer_axioms s r ->
  (finer_equivalence s r) =
  (forall x, exists y, sub(class s x)(class r y)).
Lemma finer_sub_equiv3: forall s r,
  finer_axioms s r ->
  (finer_equivalence s r) =
  (forall y, saturated s (class r y)).   (* 22 *)
Lemma finest_equivalence: forall r,
   is_equivalence r -> finer_equivalence (diagonal (substrate r)) r.
Lemma coarsest_equivalence: forall r,
   is_equivalence r -> finer_equivalence r (coarse (substrate r)).
```

$$\begin{tikzcd}
E \arrow[r, "=="] \arrow[d, "g"'] & E \arrow[d, "f"] \\
E/S \arrow[r, "h"] \arrow[d, "h_1"'] & E/R \\
(E/S)/(R/S) \arrow[r, "h_2"'] & E/R \arrow[u, "="']
\end{tikzcd}$$

(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 $h(C_S x) = C_R x$.

```
Lemma compatible_with_finer: forall s r,
  finer_axioms s r ->
  finer_equivalence s r ->
  compatible_with_equiv (canon_proj r) s.
Lemma foq_finer_function: forall s r,
  finer_axioms s r ->
  finer_equivalence s r -> is_function(fun_on_quotient s (canon_proj r)).
Lemma foq_finer_W: forall s r x,
  finer_axioms s r -> finer_equivalence s r ->  inc x (quotient s) ->
  W x (fun_on_quotient s (canon_proj r)) = class r (rep x).
Lemma foq_finer_surjective: forall s r,
  finer_axioms s r ->
  finer_equivalence s r -> surjective(fun_on_quotient s (canon_proj r)).
```

On the quotient we can consider the equivalence induced by $h$. This will be denoted R/S. We have $C_S x \overset{R/S}{\sim} C_S y$ if and only if $x \overset{R}{\sim} y$; this is the same as $g(x) \overset{R/S}{\sim} g(y)$. We have $x \in (E/S)/(R/S)$ if and only if there exists $y \in E/R$ 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:Set):=
  equivalence_associated (fun_on_quotient s (canon_proj r)).

Lemma quotient_of_relations_equivalence:
  forall r s, finer_axioms s r -> finer_equivalence s r ->
    is_equivalence (quotient_of_relations r s).
Lemma quotient_of_relations_substrate:
  forall r s, finer_axioms s r -> finer_equivalence s r ->
    substrate (quotient_of_relations r s) = (quotient s).
Lemma quotient_of_relations_related: forall r s x y,
  finer_axioms s r -> finer_equivalence s r ->
  related (quotient_of_relations r s) x y =
  (inc x (quotient s) & inc y (quotient s) &
    related r (rep x) (rep y)).
Lemma quotient_of_relations_related_bis: forall r s x y,
  finer_axioms s r -> finer_equivalence s r ->
  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 nonempty_image: forall f x,
  is_function f -> nonempty x -> sub x (source f) ->
  nonempty (image_by_fun f x).
Lemma cqr_aux: forall s x y u,
  is_equivalence s -> sub y (substrate s) ->
  x = image_by_fun (canon_proj s) y ->
  inc u x = (exists v, inc v y & u = class s v).
Lemma quotient_of_relations_class_bis: forall r s x,
  finer_axioms s r -> finer_equivalence s r ->
  inc x (quotient (quotient_of_relations r s)) =
  exists y, inc y (quotient r) & x = image_by_fun (canon_proj s) y.  (* 85 *)
```

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: forall r s,
  let f := fun_on_quotient s (canon_proj r) in
```

```
    let qr := quotient_of_relations r s in
  finer_axioms s r -> finer_equivalence s r ->
  f =  (compose (fun_on_quotient qr f) (canon_proj qr)).
Lemma quotient_of_relations_pr: forall s t,
  let r := inv_image_relation (canon_proj s) t in
  is_equivalence s -> is_equivalence t -> substrate t = quotient s ->
  t = quotient_of_relations r s.  (* 25 *)
```

## 7.8 Product of two equivalence relations

Given two relations R and R′, we can define R × R′ by $(x,x')\overset{R\times R'}{\sim}(y,y')$ if and only if $x\overset{R}{\sim}y$ and $x'\overset{R'}{\sim}y'$. In the definition that follows, we consider relations on sets E and E′, and show that this gives a relation on E × E′ (the substrate is not the whole product: if E and E′ have two elements that are related, then the graphs of R and R′ have a single element, the graph of R×R′ has a single element, and its substrate has two elements, while E×E′ has four elements). If R and R′ are equivalence, so is the product, and the substrate is E×E′. A class in the product is a product of classes.

```
Definition substrate_for_prod(r r':Set) :=
  product(substrate r)(substrate r').

Definition prod_of_relation(r r':Set):=
  Zo(product(substrate_for_prod r r')(substrate_for_prod r r'))
  (fun y=> inc (J(P (P y))(P (Q y))) r & inc (J(Q (P y))(Q (Q y))) r').

Lemma prod_of_rel_is_rel: forall r r', is_graph (prod_of_relation r r').
Lemma substrate_prod_of_rel1:  forall r r',
  sub (substrate (prod_of_relation r r'))(substrate_for_prod r r').
Lemma prod_of_rel_pr: forall r r' a b,
  related (prod_of_relation r r')  a b =
  ( is_pair a & is_pair b & related r (P a) (P b) & related r' (Q a) (Q b)).
Lemma substrate_prod_of_rel2:
  forall r r',  is_symmetric r -> is_symmetric r' ->
    substrate (prod_of_relation r r') = substrate_for_prod r r'.
Lemma prod_of_rel_refl:
  forall r r',  is_reflexive r -> is_reflexive r' ->
   is_reflexive (prod_of_relation r r').
Lemma prod_of_rel_sym:
  forall r r',  is_symmetric r -> is_symmetric r' ->
   is_symmetric (prod_of_relation r r').
Lemma prod_of_rel_trans:
  forall r r',  is_transitive r -> is_transitive r' ->
   is_transitive (prod_of_relation r r').
Lemma substrate_prod_of_rel: forall r r',
  is_equivalence r ->is_equivalence r' ->
  substrate (prod_of_relation r r') =  product(substrate r)(substrate r').
Lemma equivalence_prod_of_rel:  forall r r',
  is_equivalence r -> is_equivalence r' ->
  is_equivalence (prod_of_relation r r').
Lemma related_prod_of_rel1: forall r r' x x' v,
  is_equivalence r -> is_equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  related (prod_of_relation r r') (J x x') v =
  (exists y, exists y', v = J y y' & related r x y & related r' x' y').
```

```
Lemma related_prod_of_rel2: forall r r' x x' v,
  is_equivalence r -> is_equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  related (prod_of_relation r r') (J x x') v =
  inc v (product (im_singleton r x) (im_singleton r' x')).
Lemma class_prod_of_rel2: forall r r' x,
  is_equivalence r -> is_equivalence r' ->
  is_class (prod_of_relation r r') x =
  exists u, exists v, is_class r u & is_class r' v & x = product u v.   (* 26 *)
```

With the same notations, let $\pi$ and $\pi'$ 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 $\pi''$ 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_function: forall r r',
  is_equivalence r -> is_equivalence r' ->
  is_function (ext_to_prod(canon_proj r)(canon_proj r')).
Lemma ext_to_prod_rel_W: forall r r' x x',
  is_equivalence r -> is_equivalence r' ->
  inc x (substrate r) -> inc x' (substrate r') ->
  W (J x x') (ext_to_prod(canon_proj r)(canon_proj r')) =
  J (class r x) (class r' x').
Lemma compatible_ext_to_prod: forall r r',
  is_equivalence r -> is_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: forall r r' x x',
  is_equivalence r -> is_equivalence r' ->
  is_pair x -> inc (P x) (substrate r) -> inc (Q x)  (substrate r') ->
  is_pair x' -> inc (P x') (substrate r) -> inc (Q x')  (substrate r') ->
  W x (ext_to_prod (canon_proj r) (canon_proj r')) =
  W x' (ext_to_prod (canon_proj r) (canon_proj r'))
  -> related (prod_of_relation r r') x x'.
Lemma related_ext_to_prod_rel: forall r r',
  is_equivalence r -> is_equivalence r' ->
  equivalence_associated (ext_to_prod(canon_proj r)(canon_proj r')) =
  prod_of_relation r r'.    (* 25 *)
Lemma decomposable_ext_to_prod_rel:forall r r',
  is_equivalence r -> is_equivalence r' ->
  exists h,  bijective h &
    source h = quotient (prod_of_relation r r') &
    target h = product (quotient r) (quotient r')&
    compose h (canon_proj (prod_of_relation r r')) =
    ext_to_prod(canon_proj r)(canon_proj r').   (* 58 *)
```

## 7.9 Classes of equivalent objects

Let $\sim$ be an equivalence relation; we do not assume that it has a graph. Let $\theta x$ 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 $\tau$ say $\theta x=\theta x'$. The quantity $\theta x$ 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 $\Theta$ be the set of all $\theta x$ 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 $\theta x$ 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 $\Theta$ into the quotient set.

We write $\theta x$ and $Ax$ instead of $\theta(x)$ and $A(x)$ in order to emphasize the fact that these objects are not functions. However, $\theta x$ 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 $\theta X$ is the cardinal of $X$. Bourbaki proves the existence of the cardinal by repeating the arguments previously exposed in this section.

# Chapter 8

# 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 $x \in \emptyset$.

```
Lemma not_collectivizing_notin:
  ~ (exists z, forall y, inc y z = not (inc y y)).
Proof.
case=> x hx.
have nxx: (~ inc x x) by move=> xx; move: (xx); rewrite  (hx x).
by apply nxx; rewrite (hx x).
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 emptyset_pr.
move=> [x Px]; apply not_collectivizing_notin.
exists (Zo x (fun z => ~ (inc z z))) => z.
apply iff_eq; rewrite Z_rw; intuition.
Qed.

Lemma emptyset_pra: forall x (p: Set -> Prop),
  inc x emptyset -> (p x).
Proof. by move=> x p xe;   elim (emptyset_pr xe). Qed.
```

## 8.1 Section 1

**1.** *Show that the relation* $(x = y) \iff (\forall \mathrm{X})((x \in \mathrm{X}) \implies (y \in \mathrm{X}))$ *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 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.
apply iff_eq; first  by  move=> ->.
by move=> spec_sub; symmetry; apply singleton_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:Set, exists 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* $\mathrm{B} \subset \complement\mathrm{A}$ *is equivalent to* $\mathrm{A} \subset \complement\mathrm{B}$ *and that the relation* $\complement\mathrm{B} \subset \mathrm{A}$ *is equivalent to* $\complement\mathrm{A} \subset \mathrm{B}$.

**Comment:** The notation $\complement\mathrm{A}$ is an abuse of language for $\mathrm{X}-\mathrm{A}$. One part of the proof needs the law of excluded middle. We write a tactic that proves $a \in b$ by contradiction (it takes n argument, the name of the assumption $a \notin b$).

It suffices to prove the first result (the second claim follows from the double complement rule). It suffices to prove one implication, the other is obtained by symmetry.

```
Ltac em_set h :=
 match goal with  |- inc ?a ?b => case (inc_or_not a b) => // h
end.

Lemma exercise1_3: forall X A B, sub A X -> sub B X ->
  (sub (complement X B) A = sub (complement X A) B     &
  sub B (complement X A) = sub A (complement X B)).
Proof.
move=>X.
suff: forall a b,  sub a X -> sub b X ->
    sub (complement X b) a = sub (complement X a) b.
  move=> sh A B AX BX.
  split; first by apply sh.
  set c:=sh _ _ (sub_complement (a:=X) (b:= A))(sub_complement (a:=X) (b:= B)).
  by move: c; srw.
suff: forall a b,  sub a X -> sub b X ->
    sub (complement X b) a -> sub (complement X a) b.
  by move=> sh A B AX BX ; apply iff_eq; apply sh.
move=> a b aX bX sca t;  srw; move => [ tA nta ]; em_set nintb.
by elim nta; apply sca; srw.
Qed.
```

**4.** *Prove that the relation* $\mathrm{X} \subset \{x\}$ *is equivalent to* “$\mathrm{X} = \{x\}$ *or* $\mathrm{X} = \emptyset$”.

**Comment:** The relation $X \subset \{x\}$ is equivalent to $a \in X \iff a = x$. This needs the law of excluded middle; in other terms, from the proof of $X \subset \{x\}$, it is not possible to deduce which alternative, $X = \{x\}$ or $X = \emptyset$ is true.

```
Lemma exercise1_4: forall a b,
  sub a (singleton b) = (a = singleton b \/ a = emptyset).
Proof.
move=>X x; apply iff_eq.
  case (p_or_not_p (sub (singleton x) X)); first by left; apply extensionality.
  right; apply is_emptyset => t; dneg tX.
  have xX: inc x X by rewrite - (singleton_eq (H0 _ tX)).
  by move=> y; aw=> ->.
case => ->; [ apply sub_refl | apply emptyset_sub_any].
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 $\tau$ 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;  rewrite /rep; apply choose_pr; exists y.
have Ye: forall Y,  ~ (inc (rep Y) Y) -> emptyset = Y.
  move => y ye; symmetry.
  by apply is_emptyset; move=> t; dneg aux; apply (rep_pr _ _ aux).
apply Ye; apply choose_pr; exists emptyset.
case; case.
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.

Write $R(x, y)$ for $(\forall z)(z \in x \iff z \in y)$. Let A and B be two sets. The new axiom says $A = \tau_x R(x, A)$ and $B = \tau_x R(x, B)$. Let's show the axiom of extent: assume $A \subset B$ and $B \subset A$; thus $z \in A \iff z \in B$, hence $R(A, B)$. It follows, by transitivity of equivalence, that for all $x$, $R(x, A)$ is equivalent to $R(x, B)$. Scheme S7 (property *choose_equiv*) implies $\tau_x R(x, A) = \tau_x R(x, B)$, from which $A = B$ follows.

```
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; apply iff_eq;  auto.
Qed.
```

## 8.2 Section 2

**1.** *Let $\boldsymbol{R}\{\boldsymbol{x},\boldsymbol{y}\}$ 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 $\boldsymbol{R}\{\boldsymbol{x},\boldsymbol{y}\}$. Show that the relation $(\exists \boldsymbol{x})(\exists \boldsymbol{y})\boldsymbol{R}\{\boldsymbol{x},\boldsymbol{y}\}$ 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 $(\forall \boldsymbol{x})(\forall \boldsymbol{y})\boldsymbol{R}\{\boldsymbol{x},\boldsymbol{y}\}$ is equivalent to*

$$(\forall \boldsymbol{z})(\boldsymbol{z} \text{ is an ordered pair } \Longrightarrow (\boldsymbol{R}\{\mathrm{pr}_1 \boldsymbol{z}, \mathrm{pr}_2 \boldsymbol{z}\})).$$

**Comment.** Compare this with the section “Function of two variables”.

```
Lemma exercise2_1: forall R:Set-> Set -> Prop,
  (  (exists x, exists y, R x y) = (exists z, is_pair z & R(P z) (Q z)) &
     (forall x, forall y, R x y)  = (forall z, is_pair z -> R(P z) (Q z))).
Proof.
move => R; split;apply iff_eq.
- 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 $\mathcal{T}_0$ be the theory of sets, and let $\mathcal{T}_1$ be the theory which has the same schemes and explicit axioms as $\mathcal{T}_0$, except for the axiom A3. Show that if $\mathcal{T}_1$ is not contradictory, then $\mathcal{T}_0$ 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 : forall x y z w,
  (xpair x y = xpair z w) = (x = z & y = w).
Proof.
move=> x y z w; apply iff_eq; 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 (doubleton_or fp2); first by apply singleton_inj.
   by move=> sd; symmetry; apply singleton_eq;  ue.
split=>//.
rewrite xz in sp2.
case (doubleton_or sp2) => hyp.
  symmetry.
  have syz: (singleton y = z) by apply singleton_eq; ue.
  have: (inc (doubleton z (singleton w)) (xpair x y)).
    by rewrite eq /xpair; fprops.
  rewrite xz /xpair hyp  doubleton_singleton; aw => zwz.
  have: (singleton w = z) by apply singleton_eq; ue.
  by rewrite -syz; apply singleton_inj.
apply singleton_inj.
have sp3:  (inc (singleton w)  (doubleton z (singleton y))) by  ue.
case (doubleton_or sp3) => sp4; last by symmetry.
have sp5: (inc (singleton y)  (doubleton z (singleton w))) by ue.
by case (doubleton_or sp5); try ue.
Qed.
```

## 8.3 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 containg 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”.

```
(* Definition has_no_graph (r:Set -> Set -> Prop):=
   ~(exists G, is_graph G & forall x y, r x y <-> inc (J x y) G). *)
Definition has_no_graph (r:Set -> Set -> Prop):=
 ~(exists G, forall x y, r x y -> inc (J x y) G).
Definition is_universal (r:Set -> Set -> Prop):=
  forall x, exists y, r x y \/ r y x.
```

Assume that $r$ is a universal relation, and $r(x, y)$ implies $\mathrm{J}(x, y) \in \mathrm{X}$. 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: forall r,  is_universal r -> has_no_graph r.
move=> r u [X h].
move:collectivizing_special => [_ p]; elim p.
exists (union2 (domain X)(range X)).
rewrite /domain/range => y.
move: (u y) => [x]; case => rxy.
  by apply union2_first; aw; exists (J y x); split;aw;  apply h.
by   apply union2_second;aw; exists (J x y); split;aw;  apply h.
Qed.
```

The result is now trivial.

```
Lemma 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.
intuition; 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: forall G X, is_graph G ->
  sub X (domain G) =
  sub X (image_by_graph (inverse_graph G) (image_by_graph G X)).
Proof.
move=>G X gG.
have giG :is_graph (inverse_graph G) by fprops.
apply iff_eq; move=> hyp t ts; move: (hyp _ ts); aw.
  move=> [y Jg]; exists y; aw; split =>//; ex_tac.
move=> [x]; aw; move=> [[y _] 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: forall G H, is_graph G -> is_graph H ->
  sub (domain H) (domain G)=
  sub H (compose_graph H (compose_graph (inverse_graph G) G)).
Proof.
move=> G H gG gH.
apply iff_eq => h t ts.
  move: (pair_recov (gH _ ts)) => Jt; rewrite - Jt  in ts.
  have: (inc (P t) (domain G)) by apply h; ex_tac.
  aw; move=> [y JG]; split; first by rewrite -Jt; fprops.
  exists (P t); split=>//; aw;  split; fprops; ex_tac; aw.
move: ts; aw; move=> [y JH]; move: (h _ JH); aw.
move=> [_ [z [p _]]]; move: p; aw; move=> [_ [u [q _]]].
ex_tac.
Qed.

Lemma exercise3_3b: forall G, is_graph G ->
  sub G (compose_graph G (compose_graph (inverse_graph G) G)).
Proof. move=> G gG; rewrite -(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: forall G,
  (compose_graph G emptyset = emptyset &
   compose_graph emptyset G = emptyset).
Proof.
move=> G; split.
  empty_tac x H; move: H; aw; move=> [_ [z [Je _ ]]]; elim (emptyset_pr Je).
empty_tac x H; move: H; aw; move=> [_ [z [ _ Je ]]]; elim (emptyset_pr Je).
Qed.


Lemma exercise3_4b: forall G, is_graph G ->
  (compose_graph (inverse_graph G) G = emptyset) = (G = emptyset).
Proof.
move=> G gG; apply iff_eq => h.
  empty_tac x xG; empty_tac1  (J (P x) (P x)).
  move: (gG _ xG)=> px; aw;split; fprops; exists (Q x); split; aw.
rewrite h; by move: (exercise3_4a (inverse_graph emptyset))=>[res _].
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.** The code presented here is simpler than in Version 1; we do not need to show that some quantities are graphs.

```
Lemma exercise3_5: forall G A B,
  (compose_graph (product A B) G = product (inv_image_by_graph G A) B &
  compose_graph G (product A B)  = product A (image_by_graph G B)).
Proof.
move=>G A B; split; set_extens1 x; aw.
  move=> [px [y [JG ]]]; aw; move=> [_ [? ?]]; intuition; ex_tac.
  move=> [px [[y [uA JG]] QB]]; intuition; ex_tac; fprops.
  move=> [px [y [Jp JG]]]; move: Jp; aw;move=> [_ [? ?]]; intuition; ex_tac.
  move=> [px [ pA [y [xB jB]]]]; intuition;  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 :=
  complement(product (domain G)(range G)) G.

Lemma complement_graph_g : forall G, is_graph (complement_graph G).
Proof.
rewrite /complement_graph=> G x; srw;  rewrite  product_inc_rw; intuition.
Qed.

Lemma exercise3_6a: forall G, is_graph G ->
  complement_graph (inverse_graph G) = inverse_graph(complement_graph G).
Proof.
move => G gG.
have gc: is_graph (complement_graph G) by apply complement_graph_g.
rewrite /complement_graph; bw.
rewrite range_inverse //.
set_extens1 t; srw; rewrite product_inc_rw domain_inverse//.
  move=> [[tp [PrG QdG]] ntiG].
  rewrite inverse_graph_rw //; split=>//; srw; split; fprops.
  by dneg pG; rewrite inverse_graph_rw //.
rewrite inverse_graph_rw // complement_rw product_pair_rw.
rewrite  inverse_graph_rw //.
```

```
intuition.
Qed.

Lemma exercise3_6b: forall G B, is_graph G -> sub (range G) B ->
  sub (compose_graph G (complement_graph (inverse_graph G)))
      (complement_graph (identity_g B)).
Proof.
move=> G B gG srB; rewrite exercise3_6a //.
rewrite /complement_graph => t tc; srw.
rewrite  identity_domain identity_range inc_diagonal_rw.
move: tc; aw; move=> [pt [y [Jig JG]]].
move: Jig; aw; srw; rewrite product_pair_rw.
move=> [[yd Pr] nJ].
intuition; try (apply srB=>//; ex_tac).
elim nJ; ue.
Qed.

Lemma exercise3_6c: forall A G, is_graph G -> sub (domain G) A ->
  sub (compose_graph (complement_graph (inverse_graph G)) G)
      (complement_graph (identity_g A)).
Proof.
move=> A G gG sd.
rewrite exercise3_6a // /complement_graph identity_domain identity_range.
move=> t tc; srw; rewrite inc_diagonal_rw; aw.
move:tc; aw; move =>  [Htp  [y [Jg ]]];  aw; srw;  rewrite product_pair_rw.
move=> [[Qd yr] nJ].
intuition; try (apply sd=>//; ex_tac).
elim nJ; ue.
Qed.

Lemma exercise3_6d: forall G, is_graph G ->
  ( G = product (domain G)(range G) ) =
  (compose_graph G (compose_graph (complement_graph (inverse_graph G)) G)
  = emptyset ).
Proof.
move=> G gG; rewrite  (exercise3_6a gG).
set (K:= complement_graph G).
transitivity (K = emptyset).
  rewrite /K /complement_graph.
  apply iff_eq; first by move=> <-; apply complement_itself.
  move=> hyp.
  by apply extensionality;[ apply sub_graph_prod| apply empty_complement].
apply iff_eq.
  move=> ->; rewrite  inverse_graph_emptyset.
  move: (exercise3_4a G) => [p1 p2].
  by rewrite p2 p1.
move=> ce; empty_tac x xK.
move: (xK); rewrite /K /complement_graph; srw; aw.
move=> [[px [[u J1G] [v J2G]]] nG].
empty_tac1 (J v u); aw; split; fprops;ex_tac; aw; split; fprops; ex_tac; aw.
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: forall g, is_graph g ->
  fgraph g = (forall x, sub (image_by_graph g (inv_image_by_graph g x)) x).
```

```
Proof.
move=>G gG; rewrite /inv_image_by_graph.
apply iff_eq.
  move=> fgG X x; aw; move=> [y [yi JG]].
  move: yi; aw; move=> [z [zX]]; aw => JG'.
  by rewrite (fgraph_pr fgG JG JG').
move=> hyp; split =>//.
move=> x y xG yG sP.
move:(gG _ xG) (gG _ yG)=> px py.
apply pair_extensionality=>//.
apply singleton_eq.
apply (hyp (singleton (Q y))); aw; exists (P x).
split; aw; exists (Q y); rewrite sP; aw; fprops.
Qed.
```

**8.** *Let* A, B *be two sets, let* $\Gamma$ *be a correspondence between* A *and* B*, and let* $\Gamma'$ *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* $\Gamma$ *is a bijection of* A *onto* B *and* $\Gamma'$ *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 $\Gamma(\mathrm{X})$ denotes $\Gamma\langle \mathrm{X}\rangle$. The proof is a bit longish. In the comments, G and G$'$ are the graphs.

```
Lemma exercise3_8: forall G G', is_correspondence G -> is_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) ->
  (bijective G & bijective G' & G = inverse_fun G').
Proof.
rewrite /image_by_fun=> G G' cG cG' sG sG' G'Gx GG'x.
have gG: is_graph (graph G) by fprops.
have gG': is_graph (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.
  have: inc x (singleton x) by fprops.
  rewrite - (G'Gx _ xs);  set (aux:=graph G); aw.
  move=> [y [p _]]; move: p; aw; move=> [z []]; aw; move=> -> p; ex_tac.
have sGdgG': source G' = domain (graph G').
  apply extensionality; last by fprops.
  move=> x xs.
  have: inc x (singleton x) by fprops.
  rewrite - (GG'x _ xs);  set (aux:=graph G'); aw.
  move=> [y [p _]]; move: p; aw; move=> [z []]; aw; move=> -> p; 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 singleton_eq.
  rewrite - (G'Gx _ xG); aw; ex_tac; aw; 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 singleton_eq.
  rewrite - (GG'x _ xG); aw; ex_tac; aw; ex_tac; fprops.
```

We show: if $x \in A$ there is an $y$ such that $(x, y) \in G$ and $(y, x) \in G'$.

```
have xGy:  (forall x, inc x (source G) -> exists y,
    inc (J x y) (graph G) & inc (J y x) (graph G')).
  move=> x xsG.
  have: inc x (singleton x) by fprops.
  rewrite - (G'Gx _ xsG);  set (aux:=graph G'); aw.
  move=> [y [p q]]; move: p; aw; move=> [z []]; aw; move=> -> p; ex_tac.
have xG'y:  (forall x, inc x (source G') -> exists y,
    inc (J x y) (graph G') & inc (J y x) (graph G)).
  move=> x xsG.
  have: inc x (singleton x) by fprops.
  rewrite - (GG'x _ xsG);  set (aux:=graph G'); aw.
  move=> [y [p q]]; move: p; aw; move=> [z []]; aw; move=> -> p; 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: is_pair x by apply gG.
  have py: (is_pair y) by apply gG.
  apply pair_extensionality =>//.
  rewrite  -(pair_recov px) in xG.
  rewrite  -(pair_recov 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: is_pair x by apply gG'.
  have py: (is_pair y) by apply gG'.
  apply pair_extensionality =>//.
  rewrite  -(pair_recov px) in xG.
  rewrite  -(pair_recov 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: is_function G by hnf;intuition.
have fg': is_function G' by hnf;intuition.
have GiG:  (graph G = inverse_graph(graph G')).
  set_extens x xs.
```

```
    have px: is_pair x by apply gG.
    rewrite -(pair_recov px) in xs |- *; aw.
    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: (is_graph (inverse_graph (graph G'))) by fprops.
  have px: (is_pair x) by apply gi.
  rewrite -(pair_recov px) in xs |- *; awi xs.
  have Ps: inc (P x) (source G) .
    by rewrite  sG; apply corresp_sub_range=>//; ex_tac.
  move: (xGy _ Ps)=> [y [J1 J2]].
  by 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: bijective G.
  split.
    split=>//; move=> x y xs ys sW.
    move: (W_pr3 fg xs) => HGx.
    move: (W_pr3 fg ys) => HGy; rewrite -sW in HGy.
    have Ws: inc (W x G) (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 (inc_pr1graph_source fg J2).
have GiG3: G' = inverse_fun G by rewrite GiG2 inverse_fun_involutive.
intuition.
by rewrite GiG3; apply inverse_bij_is_bij1.
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: forall  f g h,
  is_function f -> is_function g -> is_function h->
  source g = target f -> source h = target g ->
  bijective(compose g f) -> bijective (compose h g) ->
  (bijective f & bijective g & bijective h).
Proof.
move=> f g h ff fg fh sgtf shtg bgf bhg.
have cgf : composable g f by hnf ; intuition.
have chg : composable h g by hnf; intuition.
have ig: injective g.
  by move: bhg=>[ia sa]; apply (inj_right_compose chg ia).
have sg: surjective g.
  by move: bgf=>[ia sa]; apply (surj_left_compose cgf sa).
have bg: bijective g by split.
split; first by apply (bij_right_compose cgf bgf bg).
split; last by  apply (bij_left_compose chg  bhg bg).
done.
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: forall  f g h,
  is_function f -> is_function g -> is_function h->
  source g = target f -> source h = target g ->  source f = target h ->
  injective (compose h (compose g f)) ->
  surjective (compose g (compose f h)) ->
  (injective (compose f (compose h  g))
   \/ surjective (compose f (compose h  g))) ->
  (bijective f & bijective g & bijective h).
Proof.
move=> f g h ff fg fh sgtf shtg sfth ihgf sgfh is_fgh.
have cfh: composable f h by hnf; auto.
have chg: composable h g by red; auto.
have cgf: composable g f by red; auto.
rewrite -compose_assoc // in ihgf.
have fhg: is_function (compose h g) by  fct_tac.
have chgf: composable (compose h g) f by hnf; aw; auto.
move: (inj_right_compose chgf ihgf) => inf.
have ffh: is_function (compose f h) by fct_tac.
have cgfh: composable g (compose f h) by hnf; aw; auto.
move: (surj_left_compose  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 - compose_assoc// => ifhg.
  have cfhg: composable (compose f h) g by  hnf; aw; auto.
  move: (inj_right_compose  cfhg ifhg) => ig.
  move: (surj_left_compose2 cgfh sgfh  ig)=> sfh.
  move: (surj_left_compose cfh sfh) =>sf.
  move: (inj_left_compose2 cfhg ifhg sg) => ifh.
  have bfh: (bijective (compose f h)) by hnf.
  have bf: (bijective f)  by hnf.
  have bg: (bijective g)  by hnf.
  move: (bij_right_compose cfh bfh bf).
  auto.
```

The second case is similar.

```
move=> sfhg.
have cfhg: (composable f (compose h g)) by hnf; aw; auto.
move: (surj_left_compose cfhg  sfhg) => sf.
have bf: (bijective f)  by hnf.
move: (surj_left_compose2 cfhg sfhg inf) => shg.
move:(inj_left_compose2 chgf ihgf sf) => ihg.
move: (inj_right_compose chg ihg) => ig.
```

```
have bg: (bijective g)  by hnf.
have bhg: (bijective (compose h g)) by hnf.
move: (bij_left_compose chg bhg bg).
auto.
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**.

## 8.4 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: forall g,  is_graph g  ->
  functional_graph g = (forall x y, inv_image_by_graph g (intersection2 x y)=
    intersection2 (inv_image_by_graph g x) (inv_image_by_graph g y)).
Proof.
move=> sfhg.
have cfhg: (composable f (compose h g)) by hnf; aw; auto.
move: (surj_left_compose cfhg  sfhg) => sf.
have bf: (bijective f)  by hnf.
move: (surj_left_compose2 cfhg sfhg inf) => shg.
move:(inj_left_compose2 chgf ihgf sf) => ihg.
move: (inj_right_compose chg ihg) => ig.
have bg: (bijective g)  by hnf.
have bhg: (bijective (compose h g)) by hnf.
move: (bij_left_compose chg bhg bg).
auto.
Qed.

Lemma exercise4_1a: forall g,  is_graph g  ->
  functional_graph g = (forall x y, inv_image_by_graph g (intersection2 x y)=
    intersection2 (inv_image_by_graph g x) (inv_image_by_graph g y)).
Proof.
move=> g gg.
have gig: is_graph (inverse_graph g) by  fprops.
rewrite /inv_image_by_graph.
apply iff_eq.
  move=> fgg x y; set_extens1 t.
    aw; move=> [z]; aw; move=> [[zx zy] Jg].
    by split; exists z; aw.
  aw; move=> [[a p][b q]]; move: p q; aw; move=> [ax Jg][biy Jg'].
  rewrite -(fgg _ _ _ Jg Jg') in biy.
  by exists a; aw.
move=> hyp x y y' gxy gxy'.
move: (hyp (singleton y)(singleton y')).
set u:=intersection2 _ _.
move=> hyp1.
have:inc x (image_by_graph (inverse_graph g) u).
  by rewrite /u hyp1; aw; split; ex_tac; aw.
aw; move=> [z];  rewrite /u; aw.
by move=> [[zy zy'] _]; rewrite -zy -zy'.
Qed.


Lemma exercise4_1b: forall g,  is_graph g  ->
  functional_graph g = (forall x y, intersection2 x y = emptyset ->
    intersection2 (inv_image_by_graph g x) (inv_image_by_graph g y)=emptyset).
Proof.

move=> g gg; apply iff_eq.
  rewrite exercise4_1a //.
  by move=> p1 x y ie; rewrite -p1 ie /inv_image_by_graph image_by_emptyset.
move=> hyp x y y' gxy gxy'.
have gig: is_graph (inverse_graph g) by fprops.
case (emptyset_dichot (intersection2 (singleton y) (singleton y'))).
  move=>aux. move:(hyp _ _ aux).
  set v:= intersection2 _ _.
  have xv: (inc x v).
    by rewrite /v; apply intersection2_inc; aw; ex_tac; aw.
 by  move=> ve; rewrite ve in xv; elim (emptyset_pr xv).
by move=> [z]; aw; move=> [zy zy']; rewrite -zy -zy'.
Qed.
```

**2.** *Let* $\mathrm{G}$ *be a graph. Show that for each set* $\mathrm{X}$ *we have* $\mathrm{G}(\mathrm{X}) = \mathrm{pr}_2(\mathrm{G} \cap (\mathrm{X} \times \mathrm{pr}_2 \mathrm{G}))$ *and* $\mathrm{G}(\mathrm{X}) = \mathrm{G}(\mathrm{X} \cap \mathrm{pr}_1 \mathrm{G})$.

```
Lemma exercise4_2a: forall g x,  is_graph g  ->
  image_by_graph g x = range(intersection2 g (product x (range g))).
Proof.
move=> g x gg.
set g2 := intersection2 _ _.
have gi: (is_graph g2) by move=> t ; rewrite /g2; aw; intuition.
set_extens1 y; rewrite /g2; aw.
  move=> [z [zx Jg]]; exists z; aw; ee; ex_tac.
move=> [z]; aw; move=>[Jg [pJ [zx _]]]; ex_tac.
Qed.

Lemma exercise4_2b: forall g x,  is_graph g  ->
  image_by_graph g x = image_by_graph g (intersection2 x (domain g)).
Proof.
move=> g x gg; set_extens1 t;aw;  move=> [y [ys Jg]]; exists y; ee.
  apply intersection2_inc=>//;ex_tac.
move: ys; apply intersection2_first.
Qed.
```

**3.** *Let* $\mathrm{X}, \mathrm{Y}, \mathrm{Y}', \mathrm{Z}$ *be four sets. Show that* $(\mathrm{Y}' \times \mathrm{Z}) \circ (\mathrm{X} \times \mathrm{Y}) = \emptyset$ *if* $\mathrm{Y} \cap \mathrm{Y}' = \emptyset$ *and that* $(\mathrm{Y}' \times \mathrm{Z}) \circ (\mathrm{X} \times \mathrm{Y}) = \mathrm{X} \times \mathrm{Z}$ *if* $\mathrm{Y} \cap \mathrm{Y}' \neq \emptyset$.

```
Lemma exercise4_3a: forall  x y y’ z, intersection2 y y’ = emptyset ->
  compose_graph(product y’ z)(product x y) = emptyset.
Proof.
move=> x y y’ z ie; apply is_emptyset; move=> t tc.
move: tc; aw; move=> [pt [u []]]; aw.
move=> [_ [_ uy]] [_[uy’ _]].
Qed.
Lemma exercise4_3b: forall  x y y’ z, nonempty(intersection2 y y’) ->
  compose_graph(product y’ z)(product x y) = product x z.
Proof.
move=> x y y’ z [t]; aw; move=>[ty ty’].
set_extens1 u; aw.
  move=>[pu [v []]]; aw; ee.
move=>[pu [Px Qy]].
split=>//; exists t; aw; 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 to show that $(\exists y\in X)(\exists i)(x,y)\in G_i$ is the same as $(\exists i)(\exists y\in X)(x,y)\in G_i$.

```
Lemma exercise4_4a:  forall g x,
  image_by_graph(unionb g) x =
  unionb (L(domain g) (fun i=> image_by_graph(V i g) x)).
Proof.
move=> g x; set_extens1 y; aw.
  move=> [z [zx]]; rewrite unionb_rw; move=> [u [ud Jv]].
  apply unionb_inc with u; bw; aw; ex_tac.
rewrite unionb_rw; bw; move=> [z [zd]]; bw; aw; move=> [u [ux Jv]].
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. We need an auxiliary result: $G_\iota\langle\{x\}\rangle$ is a nonempty family.

```
Lemma exercise4_4b:  forall g x,
  nonempty g -> is_singleton x ->
  image_by_graph(intersectionb g) x =
  intersectionb (L(domain g) (fun i=> image_by_graph(V i g) x)).
Proof.
move=> g x neg [y] ->; move: (neg) => [p pg];
set f:= L _ _.
have nef: nonempty f.
  have [u udg]: exists u, inc u (domain g).
    exists (P p); rewrite /domain; aw; ex_tac.
  rewrite /f.
  set (ff:=  (fun i=> image_by_graph (V i g) (singleton y))).
  exists (J u (ff u)); rewrite /L; aw; ex_tac.
set_extens1 z; aw.
   rewrite /f;move=>[t []];aw;move=> ->; srw=>//;bw;move=> hyp i idf.
   bw; aw; exists y; aw; ee.
rewrite  intersectionb_rw // /f; bw=>hyp.
```

```
exists y; aw; split=>//.
apply intersectionb_inc=>//.
move=> i idg; move: (hyp _ idg); bw; aw.
by move=> [t];aw; move=> [ty]; rewrite ty.
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:
  let x:=TPa in let y:= TPb in
    let G:= doubleton(J x x)(J y y) in let H:=  doubleton(J x y)(J y x)
      in let z:= doubleton x y in
        image_by_graph (intersection2 G H) z <>
        intersection2 (image_by_graph G z)(image_by_graph H z).
Proof.
move=> x y G H z.
have xy: x <> y by apply two_points_distinct.
have gG: is_graph G by  move=> t ts; case (doubleton_or ts)=>->; fprops.
have gH: is_graph H by  move=> t ts; case (doubleton_or ts)=>->; fprops.
have  iG: image_by_graph G z = z.
  set_extens1 u.
    aw; move=> [v [vz JG]].
    case (doubleton_or JG)=> aux;  rewrite (pr2_def aux); rewrite /z; fprops.
  move=> uz; case (doubleton_or uz)=> h; move:uz;
    rewrite h /G; aw=> h'; ex_tac; fprops.
have iH:image_by_graph H z = z.
  set_extens1 u.
    aw; move=> [v [vz JH]].
    case (doubleton_or JH) => h; rewrite (pr2_def h);  rewrite /z; fprops.
  aw; rewrite /H=> uz;case (doubleton_or uz)=> uz';  [ exists y | exists x];
    rewrite  uz'; split; rewrite /z /H;fprops.
have ie: intersection2 G H = emptyset.
   apply is_emptyset.
   move=> t ti.
   move: (intersection2_both ti)=> [p1 p2]; elim xy.
  case (doubleton_or p1)=> p3; case (doubleton_or p2)=> p4; rewrite p4 in p3;
     try exact  (pr2_def p3); try exact  (pr1_def p3); auto.
  symmetry.  exact  (pr2_def p3). symmetry. exact  (pr1_def p3).
rewrite iG iH ie intersection2idem.
have xz: inc x z by rewrite /z;fprops.
move=> ime; move: xz; rewrite -ime; aw; move=> [t [_ te]].
elim (emptyset_pr te).
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:  forall g h,
  (compose_graph (unionb g) h =
     unionb (L(domain g) (fun i=> compose_graph(V i g) h))
     & compose_graph h (unionb g) =
     unionb (L(domain g) (fun i=> compose_graph h (V i g)))).
Proof.
move=> G H; split.
  set_extens1 x; aw; srw; bw.
    move=> [px [y [JPH]]]; srw; move=>[z [zd JV]].
    by ex_tac;bw; aw; split=>//; exists y.
  move=> [y [ydg]]; bw; aw; move=> [px [z [J1H J2V]]].
  split=>//; exists z; split=>//; union_tac.
```

Second part. The proof is almost identical.

```
set_extens1 x; aw; srw; bw.
  move=> [px [y [aux JPH]]];move: aux; srw; move=>[z [zd JV]].
  by ex_tac;bw; aw; split=>//; exists y.
move=> [y [ydg]]; bw; aw; move=> [px [z [J1H J2V]]].
split=>//; exists z; split=>//; union_tac.
Qed.
```

**6.** *A graph* G *is functional if and only if for each pair of graphs* H, H′ *we have*

$$(\mathrm{H}\cap\mathrm{H}')\circ\mathrm{G}=(\mathrm{H}\circ\mathrm{G})\cap(\mathrm{H}'\circ\mathrm{G}).$$

Note that $(\mathrm{H}\cap\mathrm{H}')\circ\mathrm{G}\subset(\mathrm{H}\circ\mathrm{G})\cap(\mathrm{H}'\circ\mathrm{G})$ is true for any graphs.

```
Lemma exercise4_6:  forall G, is_graph G ->
  fgraph G = (forall H H', is_graph H -> is_graph H' ->
    compose_graph (intersection2  H H') G =
    intersection2 (compose_graph H G) (compose_graph H' G)).
Proof.
 move=> G gG; apply iff_eq.
  move=> fG H H' gH gH'.
  set_extens1 x; aw.
    move=> [xp [y [J1]]]; aw; move=> [J2 J3].
    split; split=>//; ex_tac.
  move=> [[px [y [J1 J2]]] [_ [y'[J1' J2']]]].
  rewrite  (fgraph_pr fG J1 J1') in J2.
  by split=>//;  ex_tac; aw; split.
```

Converse. If $(x,y)\in\mathrm{G}$ and $(x,y')\in\mathrm{G}$ we consider the mappings $y\mapsto x$ and $y'\mapsto x$. Then $(x,x)$ is in $\mathrm{H}\circ\mathrm{G}$ and $\mathrm{H}'\circ\mathrm{G}$. Thus $\mathrm{H}\cap\mathrm{H}'$ 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: is_graph h by  rewrite /h; move=> t; aw=>->; fprops.
have gh': is_graph h' by  rewrite /h'; move=> t; aw=>->; fprops.
move: (gG _ xG)(gG _ yG)=> xp yp.
rewrite -(pair_recov xp) in xG.
rewrite -(pair_recov yp) in yG.
apply pair_extensionality=>//.
```

```
have p1: inc (J (P x)(P x)) (compose_graph h G).
  aw; split;fprops;  ex_tac; rewrite /h;  fprops.
have p2: inc (J (P y)(P y)) (compose_graph h' G).
  aw; split;fprops;  ex_tac; rewrite /h';  fprops.
have p3: (inc (J (P x)(P x)) (compose_graph (intersection2 h h') G)).
  by rewrite hyp//; apply intersection2_inc=>//;rewrite Pxy //.
move: p3; aw; rewrite /h /h'; move=> [_ [z [_]]]; aw.
by move=> [r1 r2]; rewrite -(pr1_def r1) -(pr1_def r2).
Qed.
```

**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:  forall G H K,
  sub(intersection2 (compose_graph H G) K)
     (compose_graph(intersection2 H (compose_graph K (inverse_graph G)))
       (intersection2 G (compose_graph (inverse_graph H) K))).
Proof.
move=> G H K t; aw; move => [[tp [y [JG JH]]] tK].
split=>//.
rewrite - (pair_recov tp) in tK.
exists y; split; aw;ee;  ex_tac; aw.
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:  forall dr r ds s x,
  covering_f dr r x -> covering_f ds s x ->
  partition_fam (L ds s) x ->  coarser_covering ds s dr r ->
  (forall k, inc k ds -> nonempty (s k)) ->
  forall k, inc k ds -> exists i, inc i dr & sub (r i) (s k).
Proof.
move=> dr r ds s x co1 co2 [fgL [md usx]] co alne k kds.
move: (alne _ kds)=> [y ysk].
have yx: inc y x by rewrite -usx;apply unionb_inc with k;bw.
have yu: (inc y (unionf dr r)) by apply co1.
move:  (unionf_exists yu)=> [z [zdr yrz]].
ex_tac.
move: (co _ zdr)=> [i [ids rsi]].
have yri: inc y (s i) by apply rsi.
move: md; rewrite /mutually_disjoint; bw=> aux; case (aux _ _ kds ids).
  by move=>  ->.
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.

```
Lemma exercise4_8b: let a:= TPa in let b:= TPb in
  let x:= doubleton a b in let dr:= singleton a in let r:= fun _ => x
    in let ds:= x in let s:= variant a x (singleton a) in
      (covering_f dr r x & covering_f ds s x &
        coarser_covering ds s dr r &
        (forall k, inc k ds -> nonempty (s k)) &
        ~ (forall k, inc k ds -> exists i, inc i dr & sub (r i) (s k))).
Proof.
move=> a b x dr r ds s; rewrite /ds/dr/x/s.
have ba: b<> a by rewrite /a/b; apply two_points_distinctb.
ee.
-  by move=> t tx; apply unionf_inc with a; fprops.
-  move=> y yx;apply unionf_inc with a; fprops; rewrite  variant_if_rw //.
- move=> t; aw;  move=> ->;exists a; split;fprops; rewrite variant_if_rw //.
- move=> k kx; case (doubleton_or kx) => ->.
  rewrite  variant_if_rw//;apply nonempty_doubleton.
  rewrite  variant_if_not_rw//; apply nonempty_singleton.
- have bd: (inc b (doubleton a b)) by fprops.
  move=> h; move: (h _ bd)=> [i]; aw; move=> [] -> .
  rewrite  variant_if_not_rw //.
  rewrite /r/= =>xa; move: (xa _ bd); aw.
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 if 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; we could use $\emptyset$, $\{\emptyset\}$, $\{\{\emptyset\}\}$ and $\{\{\{\emptyset\}\}\}$, but it is a bit longish to prove that all elements are distinct. We consider here two sets of 3 and 4 elements, and a tatic that solves inequalities.

```
Inductive four_points : Set :=  | fpa  | fpb | fpc |  fpd.
Inductive three_points : Set :=  | tpa  | tpb | tpc.

Ltac disc:=
  match goal with
    |- Ro ?x <> Ro ?y =>
      red; intros;cut(x = y); [ intros hyp; discriminate hyp | apply  R_inj=>//]
    | h:Ro ?x = Ro ?y |- False =>
     cut(x = y); [ intros hyp; discriminate hyp | apply  R_inj=>//]
end.

Lemma exercise4_8c:
  let x:= four_points in let dr:= three_points in
    let r:= fun i=>  Yo (i = (Ro tpa)) (singleton (Ro fpa))
      (Yo (i = (Ro tpc)) (singleton (Ro fpb)) (doubleton (Ro fpc) (Ro fpd)))
      in let ds:= doubleton(Ro fpa) (Ro fpb)
        in let s:= variant (Ro fpa) (doubleton (Ro fpa) (Ro fpc))
          (doubleton (Ro fpb) (Ro fpd))
          in (partition_fam (L ds s) x &
            partition_fam (L dr r) x&
           (forall k, inc k ds -> exists i, inc i dr & sub (r i) (s k))&
           ~( coarser_covering ds s dr r )).
```

The first step is to prove that S is a partition. It has two elements $S_a = \{a, c\}$ and $S_b = \{b, d\}$. The key relation is $S_a \cap S_b = \emptyset$.

```
Proof.
move=> x dr r ds s.
have nba: Ro fpb <> Ro fpa by disc.
have sab:  (disjoint (V (Ro fpa) (L ds s)) (V (Ro fpb) (L ds s))).
  rewrite /ds/s; bw; fprops.
  rewrite  variant_if_rw // variant_if_not_rw; last by apply nba.
  apply disjoint_pr=>u ud1 ud2.
  case (doubleton_or ud1)=> h; case (doubleton_or ud2); rewrite h=>h'; disc.
split.
  split; gprops; split.
    red; bw; move=> i j ids jds.
    case (doubleton_or ids)=>->; case (doubleton_or jds)=>->; auto.
    by right; apply disjoint_symmetric.
```

We show that S is a covering. For each $x$, there is a $v$ such that $x \in S_v$. It is respectively $a$, $b$, $a$ and $b$.

```
rewrite  variant_if_not_rw//];
       move=> yd; case  (doubleton_or yd) =>->; apply R_inc.
  elim ys; move=> u <-; elim u;
   [ set (v:= fpa) | set (v:= fpb) | set (v:= fpa) |  set (v:= fpb) ];
   apply unionb_inc with (Ro v); bw; fprops; bw;
     solve [  rewrite  variant_if_rw//;  fprops |
              rewrite  variant_if_not_rw//; 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 $R_b = \{c, d\}$, $R_c = \{b\}$.

```
have dab: disjoint (r (Ro tpa)) (r (Ro tpb)).
  rewrite /r Y_if_rw // Y_if_not_rw //; try disc.
  rewrite Y_if_not_rw; try disc.
  apply disjoint_pr=> u; aw =>->;rewrite doubleton_rw; case => h; disc.
have dac: disjoint (r (Ro tpa)) (r (Ro tpc)).
  rewrite /r Y_if_rw // Y_if_not_rw //; try disc; rewrite Y_if_rw //.
  apply disjoint_pr=> u; aw =>-> => h; disc.
have dcb: disjoint (r (Ro tpc)) (r (Ro tpb)).
  rewrite /r Y_if_not_rw //; try disc; rewrite  Y_if_rw //.
  rewrite Y_if_not_rw; try disc; rewrite Y_if_not_rw; try disc.
  apply disjoint_pr=> u; aw =>->;rewrite doubleton_rw; case => h; disc.
split.
  split; gprops; split.
    red; rewrite /r; bw; move=> i j idr jdr; bw.
    case idr; case jdr; move=> u <-  v <-; bw.
    elim u; elim v; auto; right; apply disjoint_symmetric=>//.
```

We now show that R is a covering. For each $x$, there is a $v$ such that $x \in R_v$. It is respectively $a$, $c$, $b$ and $b$.

```
  set_extens t ts.
    move: (unionb_exists ts) ;bw; move => [y [ydr]];rewrite /r; bw.
    case ydr=> u <-; elim u.
        rewrite Y_if_rw//; aw=> ->; apply R_inc.
      do 2 (rewrite  Y_if_not_rw; try disc).
      move => h;case (doubleton_or h) =>->; apply R_inc.
    rewrite Y_if_not_rw; try disc;rewrite Y_if_rw//;  aw=> ->; apply R_inc.
  rewrite /r; case ts=> u <-.
  case u ; [set (v:= tpa) | set (v := tpc) | set (v := tpb)
    | set (v:= tpb) ] ; apply unionb_inc with (Ro v); bw; try apply R_inc;
   first (by rewrite Y_if_rw //; fprops);
   rewrite  Y_if_not_rw; try disc;
   first (by rewrite Y_if_rw //; fprops);
   rewrite  Y_if_not_rw; try disc; 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 $R_c \subset S_b$.

```
split.
  rewrite /s/r/ds; move => k kds; case (doubleton_or kds) =>->.
  rewrite  variant_if_rw //; exists (Ro tpa).
    split; first by apply R_inc.
    rewrite  Y_if_rw //;  move=> t; aw;  move=> ->; fprops.
  rewrite  variant_if_not_rw //; exists (Ro tpc).
  split; first by apply R_inc.
  rewrite Y_if_not_rw; try disc;rewrite  Y_if_rw //.
  move=> t; aw;  move=> ->; fprops.
```

Now, we show that $R_b$ is not a subset of any $S_\iota$.

```
move=> cc.
move: (cc _ (R_inc tpb)) => [i [ids]].
rewrite /r;  do 2 ( rewrite  Y_if_not_rw; try disc).
set d:= doubleton (Ro fpc) (Ro fpd).
have cd: inc (Ro fpc) d by rewrite /d;fprops.

have dd: inc (Ro fpd) d by rewrite /d;fprops.
rewrite /s; case (doubleton_or ids)=> ->.
  rewrite  variant_if_rw // => h.
  move: (h _ dd); aw;rewrite doubleton_rw; case=>h'';disc.
rewrite  variant_if_not_rw; (try disc) => h.
move: (h _ cd); aw;rewrite doubleton_rw; case=>h'';disc.
Qed.
```

## 8.5 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.

```
Lemma exercise5_f1: forall x y, sub(powerset x) (powerset y) -> sub x y.
Proof.
move=> x y sxy z zx.
have p1: sub (singleton z) x by move => t; aw; move=> ->.
have p2: sub (singleton z) y by  apply powerset_sub; apply sxy; aw.
apply (p2 z); fprops.
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. It says: let E be set and $f$ a mapping from $\mathfrak{P}(E)$ into itself such that $X \subset Y$ implies $f(X) \subset f(Y)$. Let V be the intersection of the sets $Z \subset E$ for which $f(Z) \subset Z$ and let W be the union of the sets $Z \subset E$ such that $Z \subset f(Z)$. Show that $f(V) = V$ and $W = f(W)$ and that for every set $Z \subset E$ such that $f(Z) = Z$ one has $V \subset Z \subset W$.

```
Lemma exercise5_f2: forall f x v w,
  is_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 (W a f) (W b f)) ->
  v = intersection(Zo (powerset x) (fun z=> sub (W z f) z)) ->
  w = union(Zo (powerset x) (fun z=> sub z (W z f))) ->
  (W v f = v & W w f = w & (forall z, sub z x -> W z f = z ->
    (sub v z & sub z w))).
Proof.

move=> f x v w ff sf tf fprop vd wd.
set (q:= (Zo (powerset x) (fun z => sub (W z f) z))).
have xpx: inc x (powerset x) by aw; fprops.
have xiq: inc x q.
  rewrite /q; apply Z_inc=>//.
  by apply powerset_sub; rewrite -tf; apply inc_W_target =>//; rewrite sf.
have neq:nonempty q by exists x.
set (p:= (Zo (powerset x) (fun z  => sub z (W z f)))).
have fzv:forall z, sub z x -> W z f = z -> sub v z.
  move => z zx Wz.
  have zq:inc z q by rewrite /q; apply Z_inc; aw; rewrite Wz; fprops.
  by rewrite vd; apply intersection_sub.
have fzw:forall z, sub z x -> W z f = z -> sub z w.
  move => z zx Wz.
  have zp: inc z p by rewrite /p; apply Z_inc; aw; rewrite Wz; fprops.
  by rewrite wd; apply union_sub.
have qW: forall z, inc z q -> inc (W z f) q.
   move=> z; rewrite /q ! Z_rw; aw; move=> [zx Wzz].
   have aux: sub (W z f) x by apply sub_trans  with z.
   split=>//; apply fprop=>//;aw.
have pW: forall z, inc z p -> inc (W z f) p.
   move=> z; rewrite /p ! Z_rw; aw; move=> [zx Wzz].
   have aux: inc (W z f) (powerset x).
     by rewrite -tf; apply inc_W_target=>//;rewrite sf; aw.
   awi aux; split=>//; apply fprop=>//; aw.
have vp: inc v (powerset x). by aw; rewrite vd; apply intersection_sub.
have wp: inc w (powerset x).
  by aw;rewrite wd; apply sub_union => y;rewrite Z_rw; aw; move=> [res _].
have pv:sub (W v f) v.
  move=> t tW; rewrite vd; apply intersection_inc=>//.
  move=> y; rewrite Z_rw; move => [yp sW].
  have vy: sub v y by rewrite vd; apply intersection_sub; apply Z_inc=>//.
  by apply sW;apply : (fprop _ _ vp yp vy).
have pw:sub w (W w f).
  move=> t; rewrite {1} wd; srw;move=> [y [ty]]; rewrite Z_rw; move => [yp yW].
  have tw: (sub y w) by rewrite wd;apply union_sub; apply Z_inc=>//.
  by move: (fprop _ _ yp wp tw); apply; apply yW.
split.
  apply extensionality=>//.
  have vq: (inc v q) by rewrite /q;apply Z_inc.
  by move: (qW _ vq)=> aux; rewrite {1} vd;apply intersection_sub.
```

```
split.
  apply extensionality=>//.
  have iwp: inc w p by rewrite /p; apply Z_inc.
  by move: (pW _ iwp) => aux; rewrite {2} wd;apply union_sub.
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: forall id x y,
  (forall i, inc i id -> sub (y i) (x i)) -> nonempty id ->
  productf id y =
  intersectionf id (fun i=> inv_image_by_fun (pr_i (L id x) i) (y i)).
Proof.
move=> I x y syxi neI.
have fgL: fgraph (L I x) by gprops.
have fpj: forall j, inc j I->is_function (pr_i (L I x) j).
  move=> j jI; apply pri_function=>//;bw.
set_extens1 t.
  aw; move=> [fgt [dt iVy]].
  apply intersectionf_inc=>//.
  move=> j jI; rewrite /inv_image_by_fun; aw.
  exists (V j t); split; first by apply iVy; ue.
  have jd: inc j (domain  (L I x)) by bw.
  have tp:(inc t (productb (L I x))).
    aw; ee; bw.
    rewrite dt; move=> i idt; bw; apply syxi=>//; apply iVy; ue.
  by rewrite -(pri_W fgL jd tp); graph_tac; rewrite /pr_i bl_source.
have rI: inc (rep I) I by apply nonempty_rep.
srw; move=> h; move: (h _ rI).
rewrite /inv_image_by_fun; aw; move=> [u [uy Jg]].
move: (inc_pr1graph_source (fpj _ rI) Jg).
rewrite /pr_i; aw; bw; move=>[fgt [dt iVV]]; ee.
rewrite dt;move=> i idt; move: (h _ idt).
rewrite /inv_image_by_fun; aw; move=> [v [vi Jgv]].
move: (W_pr (fpj _ idt) Jgv); rewrite pri_W=>//; bw.
  by move=> ->.
aw; bw;ee.
Qed.
```

**2.** *Let* A, B *be two sets. For each subset* G *of* A × 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: forall a b,
  bijective (BL(fun g => L a (fun x => image_by_graph g (singleton x)))
    (powerset (product a b)) (set_of_gfunctions a (powerset b))).
Proof.
move=> a b; set tilde := BL _ _.
apply bl_bijective.
```

We first prove that the mapping $G \mapsto \tilde{G}$ is a function.

```
    move=> c; aw => cp.
    set faux:=(BL (fun x=> image_by_graph c (singleton x)) a (powerset b)).
    suff: (inc (graph faux) (set_of_gfunctions (source faux) (target faux))).
       by rewrite /faux /BL; aw.
    apply inc_set_of_gfunctions=>//.
    rewrite /faux; apply bl_function=>//; move=> t; aw.
    move=> ta u ; aw; move=> [x]; aw; move=> [xt Jc].
    move: (cp _ Jc); aw; intuition.
```

We prove that the mapping is injective.

```
  move => u v; set fx := L a _; set fy:= L a  _.
  aw; move=> up vp fxy.
  set_extens x xs.
    move: (up _ xs); aw; move=> [px [Px Qx]].
    have: inc (Q x) (V (P x) fy).
      rewrite -fxy /fx;bw; aw; ex_tac; aw.
    rewrite /fy; bw; aw; move=> [w]; aw; move=> [aux]; rewrite aux; aw.
  move: (vp _ xs); aw; move=> [px [Px Qx]].
  have: inc (Q x) (V (P x) fx).
    rewrite fxy /fy;bw; aw; ex_tac; aw.
  rewrite /fx; bw; aw; move=> [w]; aw; move=> [aux]; rewrite aux; aw.
```

We prove that the mapping is surjective.

```
move=> y ys; move: (set_of_gfunctions_inc ys)=> [f [fs [sf [tg gf]]]].
set (g:=Zo (product a b) (fun z => inc (Q z) (V (P z) y))).
have gp: inc g (powerset (product a b)) by aw;rewrite /g;apply Z_sub.
rewrite -gf.
ex_tac; apply fgraph_exten; gprops; fprops.
  symmetry; bw; aw.
bw; move=> x xa; bw.
set_extens1 u; aw.
  move=> [v]; aw; move=> [vx]; rewrite vx /g Z_rw -gf; aw; intuition.
rewrite gf;move=> h; exists x; split;aw.
rewrite /g; apply Z_inc; aw; ee.
by rewrite -sf in xa; move: (inc_W_target fs xa); rewrite tg /W gf; aw; apply.
Qed.
```

**3.** * *Let* $(X_i)_{1\le i\le n}$ *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*

$$\bigcup_{H\in\mathfrak{F}_k} Q_H \supset \bigcap_{H\in\mathfrak{F}_k} P_H \text{ if } k \le (n+1)/2$$

*and that*

$$\bigcup_{H\in\mathfrak{F}_k} Q_H \subset \bigcap_{H\in\mathfrak{F}_k} P_H \text{ if } k \ge (n+1)/2. \quad *$$

Bourbaki defines integers and finite sets only later. We can define finite sets by induction. We could try to say: let I be the index, T be a subset of I, and consider all H equipotent to T. The condition $k \le n/2$ is equivalent to: there is an injection from T into the complementary. The condition is however $k \le (n+1)/2$, this makes the result non-obvious.

## 8.6 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* $\mathrm{pr}_1 G = E$, $\mathrm{pr}_2 G = E$, $G \circ G^{-1} \circ G = G$ *and* $\Delta_E \subset G$ *(*$\Delta_E$ *being the diagonal of* E*).*

**Comment.** The condition $\mathrm{pr}_2 G = E$ was missing in the English version [2]. It is necessary: consider the graph with two elements $(a, a)$ and $(b, a)$.

```
Lemma exercise6_1: forall x g, is_graph g ->
  (is_equivalence g & substrate g = x) =
  (domain g = x & range g = x &
    compose_graph g (compose_graph (inverse_graph g) g) = g
    & sub (identity_g x) g).
Proof.
move=> x g gg; apply iff_eq.
  move=> [eg sg]; ee.
  -  by rewrite (domain_is_substrate eg).
  - rewrite - sg /substrate; set_extens t ts.
      by rewrite union2_rw; right.
    case (union2_or  ts) =>//.
    aw; move=> [y Jh]; exists y; equiv_tac.
  - set_extens1 y.
      aw; move=> [py [z]]; aw; move=> [[_ [u [J1 J2]]] J3]; awi J2.
      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.
      aw.
    move=> yg.
    have py: is_pair y by apply gg.
    have  yv:J (P y) (Q y) = y by aw.
    aw; split=>//; exists (P y); split; last by  rewrite yv.
    aw;split; fprops; exists (Q y); rewrite yv; split=>//; aw.
  - move => t; rewrite inc_diagonal_rw;  move=> [pt [Pt PQt]].
    have <-: (J (P t) (Q t) = t) by aw.
    by rewrite -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 union2idem.
split=>//.
have p1: forall u, inc u x -> inc (J u u) g.
  move=> u ux; apply si;rewrite inc_diagonal_rw; aw;ee.
have p2: is_symmetric g.
  split=>//; 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; aw; split; fprops.
  ex_tac; aw; split; fprops;ex_tac; aw.
have p3: is_transitive g.
  split=>// a b c ab bc; rewrite -cg; aw.
  exists b; split =>//; aw; exists b; split=>//.
  by red; aw; apply p1; rewrite -dg; aw; exists c.
split; split =>//; first by ue.
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: forall g, is_graph g ->
  compose_graph g (compose_graph (inverse_graph g) g) = g ->
  (is_equivalence (compose_graph (inverse_graph g) g) &
    substrate (compose_graph (inverse_graph g) g) = domain g &
    is_equivalence (compose_graph g (inverse_graph g)) &
    substrate (compose_graph g (inverse_graph g)) = range g).
Proof.
move=> g gg cg.
have gig:is_graph (inverse_graph g) by apply inverse_graph_is_graph.
have gcigg:is_graph (compose_graph (inverse_graph g) g).
  by apply composition_is_graph.
have gcgig: is_graph (compose_graph g (inverse_graph g)).
  by apply composition_is_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; aw.
  split; fprops; exists z; split=>//.
  aw; split; fprops; exists y; aw; fprops.
have s1: substrate (compose_graph (inverse_graph g) g) = domain g.
  set_extens1 x.
    rewrite inc_substrate_rw=>//.
    case; move=> [y]; aw;move => [_ [z [J1 J2]]]; [ ex_tac |  awi J2;ex_tac].
  aw; move=> [y Jg].
  have Jxx: (inc (J x x) (compose_graph (inverse_graph g) g)).
    by aw; split; fprops; ex_tac; aw.
  apply (inc_arg1_substrate Jxx).
have s2:substrate (compose_graph g (inverse_graph g)) = range g.
  set_extens1 x.
    rewrite inc_substrate_rw=>//.
    case; move=> [y]; aw;move => [_ [z [J1 J2]]]; [ awi J1;ex_tac | ex_tac].
  aw; move=> [y Jg].
  have Jxx: (inc (J x x) (compose_graph g (inverse_graph g))).
    by aw; split; fprops; ex_tac; aw.
  apply (inc_arg1_substrate Jxx).
```

We apply proposition 1. $\Gamma$ is an equivalence if $\Gamma = \Gamma^{-1}$ and $\Gamma \circ \Gamma = \Gamma$. If $\Gamma$ 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.

```
ee; rewrite equivalence_pr; split;
     try rewrite inverse_compose inverse_graph_involutive //.
  by rewrite -composition_associative cg.
rewrite composition_associative in cg.
by rewrite composition_associative  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 ~ 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).

```
Definition intersection_with x a :=
  BL(fun w=> intersection2 w a) (powerset x)(powerset x).
Definition intersection_with_canon x a :=
  BL (fun b => Zo(powerset x)(fun c=> c = intersection2 c a))
  (powerset a)(quotient (equivalence_associated (intersection_with x a))).
```

We start with some preliminaries.

```
Lemma exercise6_3: forall a x,
  sub a x -> bijective (intersection_with_canon x a).
Proof.
move=> a x ax.
have ta:transf_axioms (fun w0 => intersection2 w0 a)
    (powerset x) (powerset x).
  move=> y; aw=> ay; apply sub_trans with y=>//; apply intersection2sub_first.
have fai: is_function (intersection_with x a).
  by apply bl_function.
set r:= equivalence_associated (intersection_with x a).
have er: is_equivalence r by apply graph_ea_equivalence.
have aux: forall y, sub y a -> y = intersection2 y a.
  by move => y; rewrite intersection2_sub.
```

We show that we have a function.

```
apply bl_bijective.
    move=> y; aw => ya.
    set w:= Zo _ _ .
    have new: nonempty w.
      exists y;rewrite /w;apply Z_inc; aw=>//.
        apply sub_trans with a=>//.
      by apply aux.
    have swp: sub w (powerset x) by rewrite /w; apply Z_sub.
    have rp: inc (rep w) (powerset x) by  apply swp;apply nonempty_rep.
    srw; red; ee.
      by rewrite graph_ea_substrate /intersection_with; aw; awi rp.
    have ira: (intersection2 (rep w) a = y).
      have: (inc (rep w) w) by apply nonempty_rep.
      rewrite  {2} /w Z_rw; intuition.
    set_extens1 z; bw; rewrite ea_related /intersection_with // ? bl_source.
      rewrite {1} /w Z_rw; move=> [zp iza];  awi zp; awi rp;ee; aw.
      by rewrite ira iza.
    move=> [_ [zp]];  awi zp; awi rp; aw; rewrite ira; move=> h.
    by apply Z_inc;aw;rewrite h.
```

We prove injectivity.

```
  move=> u v; aw => ua va.
  set fs:= Zo _ _ .
  move=> eql.
  have iua: u = intersection2 u a by apply aux.
  have: inc u fs by rewrite /fs; apply Z_inc; aw; apply sub_trans with a.
```

We prove now the surjectivity.

```
move=> y; srw;move => [];  rewrite graph_ea_substrate //;  move=> _ [yp1 yp2].
have ip: inc (intersection2 (rep y) a) (powerset a).
  aw; apply intersection2sub_second.
ex_tac.
have si: (source (intersection_with x a)) = powerset x.
  by rewrite /intersection_with; aw.
rewrite si in yp1; awi yp1.
rewrite yp2; set_extens1 t; bw; rewrite ea_related // si -yp2 Z_rw.
   move=> [tp ia]; ee; awi tp; rewrite /intersection_with; aw.
aw; move=> [_ [tx]];rewrite /intersection_with; aw; intuition.
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* $pr_1 A = E$ *(resp.* $pr_2 A = E$*) 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: forall g a b x,
  let comp := compose_graph in let inter:= intersection2 in
  is_equivalence g -> is_graph a -> is_graph b -> substrate g = x -> sub a g ->
  (domain a = x -> comp g a = g &
  range a = x -> comp a g = g &
  (domain a = x -> comp (inter g b) a = inter g (comp b a)) &
  (range a = x -> comp a (inter g b)  = inter g (comp a b))).
Proof.
move=> g a b x comp inter eg ga gb sg ag.
have gg: is_graph g by fprops.
ee.
-  move=> ax; rewrite /comp; set_extens1 y;aw.
     move=> [py [z [J1a J2g]]].
     move: (ag _ J1a) => J1g.
     rewrite - (pair_recov py); equiv_tac.
   move=> yg; move: (gg _ yg)=> py; split =>//.
   have : (inc (P y) (domain a)) by rewrite ax - sg; substr_tac.
   aw; move=> [z Ja];  exists z; split =>//.
   move: (ag _ Ja)=> Jg.
   have J2g:inc (J z (P y)) g by  equiv_tac.
   rewrite - (pair_recov py) in yg; equiv_tac.
```

Second claim.

```
-  move=> ax; rewrite /comp; set_extens1 y;aw.
     move=> [py [z [J1g J2a]]].
     move: (ag _ J2a) => J2g.
     rewrite - (pair_recov py); equiv_tac.
   move=> yg; move: (gg _ yg)=> py; split =>//.
   have : (inc (Q y) (range a)) by rewrite ax - sg; substr_tac.
   aw; move=> [z Ja];  exists z; split =>//.
   move: (ag _ Ja)=> Jg.
   have J2g:inc (J (Q y) z) g by  equiv_tac.
   rewrite - (pair_recov py) in yg; equiv_tac.
```

Third claim.

```
- move=> ax; rewrite /comp/inter; set_extens1 y; aw.
  move=> [py [z]]; aw; move=> [J1a [J2g J3b]].
  split.
    move: (ag _ J1a) => J1g; rewrite - (pair_recov py); equiv_tac.
    split=>//;exists z; split=>//.
  move=> [yg [py [z [J1a J2b]]]].
  split=>//; exists z; aw;ee.
  move: (ag _ J1a)=> Jg.
  have J2g:inc (J z (P y)) g by  equiv_tac.
  rewrite - (pair_recov py) in yg; equiv_tac.
```

Last claim.

```
- move=> ax; rewrite /comp/inter; set_extens1 y; aw.
  move=> [py [z]]; aw; move=> [[J1b J2b] J3a].
  split.
    move: (ag _ J3a) => J3g; rewrite - (pair_recov py); equiv_tac.
    split=>//;exists z; split=>//.
  move=> [yg [py [z [J1b J2a]]]].
  split=>//; exists z; aw;ee.
  move: (ag _ J2a)=> Jg.
  have J2g:inc (J (Q y) z) g by  equiv_tac.
  rewrite - (pair_recov 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: forall a b, is_symmetric a -> is_symmetric b ->
  is_symmetric (union2 a b).
Proof.
move=> a b [ga sa] [gb sb].
split=>//.
  move=> t; aw; case; fprops.
move=> x y; rewrite /related; aw.
by case=> h; [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: forall x g,
  sub g (coarse x) -> substrate (union2 g (identity_g x)) = x.
Proof.

move=> x g gc.
rewrite /coarse in gc.
have gg: is_graph g by move=> t tg; move: (gc _ tg); aw; intuition.
have gu: is_graph (union2 g (identity_g x)).
  move=> t; aw; case; rewrite ? inc_diagonal_rw; intuition.
set_extens1 y.
  rewrite inc_substrate_rw //; case; move=> [z]; aw; case;
```

```
     try (move=> h; move: (gc _ h); aw); intuition; ue.
move=> yx.
have aux: inc (J y y) (union2 g (identity_g x)).
  apply union2_second;  rewrite inc_pair_diagonal; auto.
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 :=
  union2 (doubleton (J a b) (J b a))(identity_g x).
Lemma substrate_special_equivalence:forall a b x,
  inc a x -> inc b x ->  substrate(special_equivalence a b x) = x.
Proof.
move=> a b x ax bx; rewrite/ special_equivalence.
apply substrate_union_diag.
move=> t; rewrite / coarse doubleton_rw; case;move=> ->;fprops.
Qed.
```

We show that this is an equivalence.

```
Lemma special_equivalence_ea:forall a b x,
  inc a x -> inc b x -> is_equivalence(special_equivalence a b x).
Proof.

move=> a b x ax bx.
have gs: is_graph (special_equivalence a b x).
  move=> t; rewrite /special_equivalence; aw.
  rewrite inc_diagonal_rw doubleton_rw; case ; [case; move=> ->|case]; fprops.
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_extensionality; fprops; aw.
    apply (pr2_def eql).
  apply (pr1_def eql).
apply symmetric_transitive_equivalence; split => //.
  move=> u v; rewrite /related/ special_equivalence; aw; rewrite doubleton_rw.
  case.
    case; move => h; rewrite (pair_symm _ _ _ _ h); fprops.
  move=> [uc uv]; right; rewrite  -uv; intuition.
move=> u v w; rewrite /related /special_equivalence;aw; rewrite! doubleton_rw.
case.
  case=>h; rewrite (pr1_def h) (pr2_def h); case.
        case=> h';  rewrite (pr2_def h') (pr1_def h'); intuition.
      move=> [_ bw]; rewrite bw; intuition.
    case => h';rewrite (pr2_def h') (pr1_def h'); intuition.
  move=> [_ aw]; rewrite -aw; intuition.
by move=> [ux uv]; rewrite uv.
Qed.
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: let x := three_points in
```

```
  let g1:= special_equivalence (Ro tpa) (Ro tpb) x in
    let g2:= special_equivalence (Ro tpa) (Ro tpc) x in
      (is_equivalence g1 & is_equivalence g2 & substrate g1 = x &
        substrate g2 = x &  ~ (is_equivalence (union2 g1 g2))).
Proof.
move=> x g1 g2; ee; try (apply special_equivalence_ea; apply R_inc).
    rewrite substrate_special_equivalence //; apply R_inc.
  rewrite substrate_special_equivalence //; apply R_inc.
move=> bad.
have p1: (related (union2 g1 g2) (Ro tpb) (Ro tpa)).
  red; rewrite /g1/special_equivalence  ! union2_rw; intuition.
have p2: (related (union2 g1 g2) (Ro tpa) (Ro tpc)).
  red; rewrite /g1/special_equivalence  ! union2_rw; intuition.
have :(related (union2 g1 g2) (Ro tpb) (Ro tpc)) by equiv_tac.
rewrite /related union2_rw /g1 /g2 /special_equivalence ! union2_rw.
case; case.
- rewrite doubleton_rw; case=> eq2; move: (pr2_def eq2)=> e; disc.
- rewrite  inc_diagonal_rw; aw; move=> [_ [_ e]]; disc.
- rewrite doubleton_rw; case=> eq2; move: (pr1_def eq2)=> e; disc.
- rewrite  inc_diagonal_rw; aw; move=> [_ [_ e]]; disc.
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* $G \circ H = H \circ G$*. The graph* $G \circ H$ *is then the intersection of all the graphs of equivalences on* E *wich contain both* G *and* H*.*

We show that if $G \circ H$ is an equivalence then $G \circ H = H \circ G$. This uses symmetry.

```
Lemma exercise6_6a: forall G H,
  is_equivalence G -> is_equivalence H ->
  (is_equivalence (compose_graph G H) =
    (compose_graph G H = compose_graph H G)).
Proof.
move=> G H eG eH.
set (K:= compose_graph G H).
apply iff_eq.
  move => eK;  set_extens x xK.
    have px: (is_pair x) by apply (@composition_is_graph G H).
    move: xK ; rewrite - (pair_recov px)=> h.
    have : inc (J (Q x) (P x)) K by equiv_tac.
    rewrite /K;aw;move=> [_ [y [JH JG]]]; split=>//; exists y; split;equiv_tac.
  move: xK; aw; move => [px [y [JG JH]]].
  rewrite -(pair_recov px).
  have aux: (inc (J (Q x) (P x)) K).
    rewrite /K; aw;split; fprops;exists y; split; equiv_tac.
  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 composition_associative eq.
  rewrite- (composition_associative G G H) GG -composition_associative.
  by rewrite -/K eq composition_associative HH.
by rewrite {2}/K inverse_compose -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: forall G H,
  is_equivalence G -> is_equivalence H ->  substrate G = substrate H ->
  substrate (compose_graph G H) = substrate G.
Proof.
move=> G H eG eH sG.
set_extens1 x.
  rewrite {1} /substrate union2_rw; aw; fprops.
  case; move=> [y]; aw; move=> [_ [z [J1g J2g]]]; [rewrite sG|]; substr_tac.
move=> xsG.
have p1:related G x x by equiv_tac.
 rewrite sG in xsG.
have p2:related H x x by equiv_tac.
have p3: related  (compose_graph G H) x x.
   red; aw;split; fprops;  exists x. auto.
substr_tac.
Qed.
```

We prove that the composition is the smallest equivalence that contains G and H.

```
Lemma exercise6_6c: forall G H,
  is_equivalence G -> is_equivalence H -> substrate G = substrate H ->
  (sub G (compose_graph G H) & sub H (compose_graph G H)
    &forall W, is_equivalence W -> sub G W -> sub H W ->
      sub (compose_graph G H) W).
Proof.
move=> G H eG eH sG.
have gg: is_graph G by fprops.
have gh: is_graph H by fprops.
have gc: is_graph  (compose_graph G H) by apply composition_is_graph.
ee.
- move=> y yG.
  move:  (gg _ yG) => py.
  rewrite - (pair_recov py) in yG.
  aw;split=>//;exists (P y); split=>//; equiv_tac=>//.
 rewrite - sG;  substr_tac.
- move=> y yH.
  move:  (gh _ yH) => py.
  rewrite - (pair_recov py) in yH.
  aw;split=>//;exists (Q y); split=>//; equiv_tac=>//.
 rewrite sG;  substr_tac.
- move=> w ew gW hW t.
  aw; move=> [tp [y [JH JG]]].
  move: (gW _ JG) (hW _ JH)=> J1G J2G.
  have: inc (J (P t) (Q t))) w by equiv_tac.
  aw.
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: forall g,
  is_equivalence g -> range g = substrate g.
```

```
Proof.
move=> g eg; rewrite /substrate; set_extens1 x; rewrite union2_rw.
  intuition.
case =>//;aw; fprops; move=> [y Jg]; exists y; equiv_tac.
Qed.
```

If G is an equivalence on E then $G \subset E \times E$.

```
Lemma sub_coarse: forall g,
  is_equivalence g ->  sub g (coarse (substrate g)).
Proof.
move=> g eg.
have gg: is_graph g by fprops.
move: (sub_graph_prod gg).
rewrite range_is_substrate // domain_is_substrate //.
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: forall G H,
  is_equivalence G -> is_equivalence H -> substrate G = substrate H ->
  compose_graph G H = compose_graph H G ->
  (compose_graph G H) = intersection(Zo (powerset (coarse (substrate G)))
    (fun W =>  is_equivalence W & sub G W & sub H W)).
Proof.
move=> G H 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 intersection_inc.
    exists (coarse E); apply Z_inc; aw; fprops; ee;apply coarse_equivalence.
  by move=> y; rewrite Z_rw; move=> [_ [ey [gy hy]]]; apply (lew _ ey gy hy).
rewrite - exercise6_6a // in cGH.
apply (intersection_forall  (y:=(compose_graph G H)) ts).
apply Z_inc; aw; ee.
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: forall G0 G1 H0 H1 E x,
  is_equivalence G0 -> substrate G0 = E ->
  is_equivalence H0 -> substrate H0 = E ->
  is_equivalence G1 -> substrate G1 = E ->
  is_equivalence H1 -> substrate H1 = E ->
  intersection2 G1 H0 = intersection2 G0 H1 ->
  compose_graph G1 H0 = compose_graph G0 H1 ->
  inc x E -> (
    let G1x := image_by_graph G1 (singleton x) in
      let H1x := image_by_graph 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: forall f r,
  is_equivalence r -> is_function f -> target f = substrate r ->
  (exists g, bijective g & source g = quotient (inv_image_relation f r) &
    target g = quotient (induced_relation r (image_of_fun f))).
Proof.
move => f r 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:is_equivalence s by rewrite /s; apply iirel_relation.
have iA: induced_rel_axioms r A.
  by red;ee; rewrite -tf rf; apply corresp_sub_range; move: ff=> [h _].
have eR: is_equivalence Ra by rewrite /Ra;apply induced_rel_equivalence.
```

Let's quote the properties of *class_iirel* and *class_induced_rel*: 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=>  fun y => is_class r y
    & nonempty (intersection2 y A) & x = inv_image_by_fun f y).
have qsp:forall x, inc x (quotient s) -> exists y, f1 x y.
  move=> x; rewrite inc_quotient // /s (iirel_class x ia).
  by rewrite -rf /f1.
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 => intersection2 (f2 x) A).
have f3p: (forall x,  inc x (quotient s) -> inc (f3 x) (quotient Ra)).
```

```
  move=> x xs; rewrite /Ra; move: (f2p _ xs).
  rewrite /f1 inc_quotient // induced_rel_class //.
  move=> h; exists (f2 x); intuition.
```

It is now obvious to find a function from E/S to $A/R_A$.

```
set (g:= BL f3 (quotient s) (quotient Ra)).
exists g; rewrite /g;ee; aw; apply bl_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]; aw; move => [y2v yiA].
  have y2u: inc y (f2 u).
    by apply (@intersection2_first (f2 u) A);rewrite ii;aw; split.
  have : inc y (range (graph f)) by rewrite -rf.
  aw; fprops; move => [x Jg].
  have xu: (inc x u) by rewrite uv /inv_image_by_fun; aw; ex_tac.
  have xb:(inc x v)  by rewrite vv /inv_image_by_fun; aw; ex_tac.
  move : uq vq; rewrite ! inc_quotient //; move=> uq vq.
  case (class_dichot uq vq) => // 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; rewrite inc_quotient // /Ra induced_rel_class //.
move=> [x [cx [nex yi]]].
set (u:= inv_image_by_fun f x).
have uq: inc u (quotient s).
  rewrite /s inc_quotient // iirel_class//; exists x; rewrite -rf;ee.
ex_tac.
rewrite /f3 yi.
move:(f2p _ uq); rewrite /f1; move=> [cf2 [ni ui]].
move: nex=> [t]; aw;move=> [tx].
rewrite rf; aw; fprops; move=> [z Jg].
have zu: inc z u by rewrite /u /inv_image_by_fun; aw; ex_tac.
move: zu; rewrite  {1} ui  /inv_image_by_fun; aw; move=> [t' [t'2u Jg']].
have tt': t = t' by move: (W_pr ff Jg) (W_pr ff Jg') => <-.
suff: f2 u = x by move=>->.
case(class_dichot cf2 cx)=> // di; red in di.
empty_tac1 t; apply intersection2_inc =>//; 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: forall F G p f r,
  is_equivalence r ->  F = substrate r -> p = canon_proj r ->
  surjective f -> source f = G -> target f = quotient r ->
  exists E, exists g, exists h,
  (surjective g & surjective h & source g = E & source h = E &  target g = F
    & target h = G & compose p g = compose f h).
Proof.
move=> F G p f r er sr xr sjf sf tf.
set (a:= TPa); set (b:= TPb).
have ba: b <> a by rewrite /a /b; apply two_points_distinctb.
set Ea:= product F (singleton a).
set Eb:=product G (singleton b).
set E:= union2 Ea Eb.
have gE: is_graph E.
  by rewrite /E /Ea /Eb=> t; rewrite union2_rw; aw; intuition.
have xep: forall x, inc x E -> (Q x =a \/ Q x = b).
  move=> x; rewrite /E/Ea/Eb union2_rw; aw; intuition.
have xgp:forall x, inc x G -> inc (W x f) (quotient r).
  move=> x xg; rewrite - tf;apply inc_W_target; [ fct_tac | ue].
have xgp1:forall x, inc x G -> inc (rep (W x f)) 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 $x \in E_b$; we can identify $E_b$ with G, hence assume $x \in G$ so that $f(x) \in F/R$. 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 (W (P z) f))).
have gzP:forall z, inc z Ea -> gz z = P z.
  move=> z;rewrite /Ea/gz; aw; move=> [_ [_ Qz]]; rewrite Y_if_rw //.
have gzp': forall z, inc z Ea -> inc (gz z) F.
  move=> z zEa; rewrite gzP//.
  move: zEa; rewrite /Ea; aw; intuition.
have gzQ:forall z, inc z Eb -> gz z = rep (W (P z) f).
  move=> z;rewrite /Eb/gz; aw; move=> [zp [Pz Qzb]].
  rewrite Y_if_not_rw //;  ue.
have gzq':forall z, inc z Eb -> inc (gz z) F.
  move=> z zE; rewrite gzQ //; apply xgp1.
  move: zE; rewrite /Eb; aw; intuition.
have tag:transf_axioms gz E F.
  move=> t; rewrite/E; aw; case; [apply gzp'| apply gzq'].
set (g:= BL gz E F).
have sj: surjective g.
  rewrite /g;apply bl_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 -> W (W x g) (canon_proj r) = W (P x) f.
  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.
  rewrite /g; aw;rewrite gzQ //; apply class_rep=>//; apply xgp.
  move: xEb; rewrite /Eb; aw; intuition.
```

We define now $h$ similarly.

```
set (ha:= fun x => rep (inv_image_by_fun f(singleton(W x (canon_proj r))))).
```

```
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; rewrite /inv_image_by_fun;aw.
  have ff: is_function f by fct_tac.
  move=> [u []]; aw; rewrite -sf; move=> _ Jg; graph_tac.
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 nonempty_rep.
  have ct: inc (class r x) (target f) by rewrite tf; rewrite sr in xF; gprops.
  move:(surjective_pr2 sjf ct)=> [u [us]]; move => <-.
  exists u; rewrite /inv_image_by_fun; aw.
  ex_tac; apply W_pr3=>//; fct_tac.
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.
  move=> z; rewrite /E/Ea/Eb/hz; aw;case; move=> [pz [PFG Qz]].
    rewrite  Y_if_rw//; apply haG=>//.
  rewrite Y_if_not_rw //; ue.
set(h:=BL hz E G).
have sh: surjective h.
  rewrite /h;apply bl_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_if_not_rw //.
have WWh: forall x, inc x Ea -> W (W x h) f = W (P x) (canon_proj r).
  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: xEa; rewrite /Ea; aw; move=> [px [PF]]; move=> ->.
  rewrite Y_if_rw //.
  move: (haF'' _ PF);rewrite /inv_image_by_fun;aw; move=> [u []]; aw.
  by move=>  <- Jg; move: sjf=> [ff _];rewrite (W_pr ff Jg).
```

We are now ready to prove the main result.

```
exists E; exists g; exists h; ee; try (solve [rewrite /g/h; aw]).
have cpg: composable p g.
  split; first by rewrite xr;apply canon_proj_function.
  split; first by fct_tac.
  rewrite xr /g; aw; ue.
have cfh: composable f h by  red; ee; 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 sjf => [fg _][ff _].
apply function_exten; try fct_tac; aw.
```

The non-obvious point is to show $p(g(x)) = f(h(x))$.

```
move=> x xsg; move: (xsg); rewrite sg=> xsh; aw.
have xE:inc x E by move: xsg;  rewrite /g /E; aw.
```

```
move: (xE); rewrite /E; aw; case => 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_if_not_rw //.
move: xE'; rewrite /Eb;aw; move=> [_ [_ Qb]]; ue.
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* $\mathrm{R}\{x, y\}$ *be a reflexive and symmetric relation on a set* E*. Let* $\mathrm{S}\{x, y\}$ *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* $0 \le i < n$*,* $\mathrm{R}\{x_i, x_{i+1}\}$*". Show that* $\mathrm{S}\{x, y\}$ *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* $y \in \mathrm{A}$ *and* $z \in \mathrm{E} - \mathrm{A}$*, we have " not* $\mathrm{R}\{y, z\}$*". For each* $x \in \mathrm{E}$ *show that the intersection of the sets* $\mathrm{A} \in \mathfrak{F}$ *such that* $x \in \mathrm{A}$ *is the connected component of* $x$ *with respect to the relation* R*.**

Part a is trivial.

```
Section Exercice6_10.
Lemma Exercise6_10_a: forall r:Set -> Set -> Prop,
  symmetric_r (fun  x y => r x y & r y x).
Proof. move=> r x y; intuition. Qed.
Lemma exercise6_10_b: forall r E,
  reflexive_r r E -> reflexive_r (fun  x y => r x y & r y x) E.
Proof.
rewrite /reflexive_r; move=> r E rr y.
rewrite rr; apply iff_eq;intuition.
Qed.
```

We consider now a context in which R is reflexive and symmetric on E.

```
Variables (R:Set -> Set -> Prop) (E:Set).
Variables (A1: reflexive_r 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 {struct x} : 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 : forall x y,
  chain_head (concat_chain x y) = chain_head x.
Proof.  by move=> x y;induction x. Qed.

Lemma tail_concat : forall x y,
  chain_tail (concat_chain x y) = chain_tail y.
Proof.  by move=> x y;induction x. Qed.

Lemma chained_concat: forall x y,
  chained_r x -> chained_r y -> chain_tail x = chain_head y ->
  chained_r (concat_chain x y).
Proof.
move=> x y 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: forall x y z, relS x y -> relS y z -> relS x z.
Proof.
move=> x y z [c [cc [hcx tcy]]][c' [cc' [hcy  tcz]]].
exists (concat_chain c c'); ee.
    apply chained_concat=>//;  ue.
  rewrite head_concat//.
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) {struct x} :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: forall x y, chain_tail (reconc_chain x y) = chain_tail y.
Proof. elim=> [a b y | a c r] // y; by rewrite r. Qed.
Lemma head_reconc: forall x y, chain_head (reconc_chain x y) = chain_tail x.
Proof.  elim => [a b y | a c r] // y; by rewrite r. Qed.
Lemma chained_reconc: forall x y, chained_r x -> chained_r y ->
  R (chain_head y) (chain_head x) ->  chained_r (reconc_chain x y).
Proof.
elim => [a b y c cy | P c r]=>//=; auto.
move=> y [rPh cc] cy RhyP; apply r=>//; red;ee.
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: forall x, chain_head (chain_reverse x) = chain_tail x.
Proof.  elim =>// y;elim =>// P c h h1 /=; apply head_reconc. Qed.

Lemma tail_reverse: forall x, chain_tail (chain_reverse x) = chain_head x.
Proof.  elim =>// y;elim =>// P c h h1 /=; apply tail_reconc. Qed.

Lemma chained_reverse: forall x, chained_r x -> chained_r (chain_reverse x).
Proof.
elim; 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:  forall x y, relS x y -> relS y x.
Proof.
move=> x y [c [cc [hcx tcy]]].
exists (chain_reverse c); ee.
    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; apply iff_eq.
  by move=> xE; exists (chain_pair x x); ee; simpl; rewrite - 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: is_equivalence Sgraph.
Proof.
rewrite /Sgraph; move: (equivalence_has_graph0 equivalenceS) => igo.
have g: (is_graph (graph_on relS E)) by apply graph_on_graph.
apply (equivalence_if_has_graph2 g igo).
by move: equivalenceS=> [].
Qed.

Lemma substrate_Sgraph: substrate Sgraph =E.
Proof.
rewrite /Sgraph; apply extensionality;first by apply graph_on_substrate.
move=> x xE.
have rxx:relS x x by move: equivalenceS=> [r] <-.
rewrite - (graph_on_rw2 x x  equivalenceS) in rxx.
substr_tac.
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: forall r, is_equivalence r ->
  (forall x y, R x y -> inc (J x y) r) -> sub Sgraph r.
Proof.
rewrite /Sgraph/graph_on=> 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.
rewrite Z_rw; move=> [pp  [c [cc [hcx htx]]]].
have <-: (J (P p)(Q p) = p) by awi pp; aw; move: pp;  intuition.
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 (complement E 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: forall A a b,
  inc A setF -> inc a A -> R a b -> inc b A.
Proof.
move=> A a b; rewrite Z_rw; aw; move=> [AE Ap] aA Rab.
case (inc_or_not b A)=> // nba.
have bc: inc b (complement E A)  by srw; split =>//; apply (A3 (A2 Rab)).
by elim (Ap _ _  aA bc).
Qed.

Lemma setF_pr2: forall A a b,
  inc A setF -> inc a A -> relS a b -> inc b A.
Proof.
move=> A a b 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: forall A a, inc A setF -> inc a A -> sub (class Sgraph a) A.
Proof.
move=> A a As aA t; bw;last by apply equivalence_Sgraph.
rewrite /Sgraph graph_on_rw2; last by apply equivalenceS.
apply (setF_pr2 As aA).
Qed.

Lemma setF_pr4: forall a, inc a E -> inc (class Sgraph a) setF.
Proof.
```

```
move=> a aE; rewrite /setF.
move: equivalence_Sgraph => e1.
move: equivalenceS => e2.
apply  Z_inc.
  by aw ; rewrite -substrate_Sgraph; apply sub_class_substrate.
move=> y z ya; srw; move=> [zE nzc]; dneg yz;bw.
suff: related Sgraph z a by move=> aux; equiv_tac.
have : related Sgraph y a by move: ya; bw=> h; equiv_tac.
rewrite /Sgraph ! graph_on_rw2 //.
have: relS z y by exists (chain_pair z y) => //=;auto.
apply transitiveS.
Qed.

Lemma connected_comp_class: forall x, inc x E ->
  class Sgraph x = connected_comp x.
Proof.
move=> x xE;set_extens1 t; rewrite /connected_comp.
  move=> tc;apply intersection_inc.
    exists E; apply Z_inc =>//; rewrite /setF; apply Z_inc.
      aw; fprops.
    move=> y z yE; srw; intuition.
  move=> y; rewrite Z_rw; move=> [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 Z_inc.
   by apply setF_pr4.
  rewrite - substrate_Sgraph in xE; bw; equiv_tac.
Qed.
```

**11.** *(a) Let* $R\{x,y\}$ *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* $R\{x,y\}$, $R\{x,z\}$, $R\{x,t\}$, $R\{y,z\}$ *and* $R\{y,t\}$ *imply* $R\{z,t\}$*. A subset* A *of* E *is said to be stable with respect to the relation* R *if* $R\{x,y\}$ *for all* $x$ *and* $y$ *in* A*. If* $a$ *and* $b$ *are two distinct elements of* E *such that* $R\{a,b\}$ *show that the set* $C(a,b)$ *of elements* $x \in E$ *such that* $R\{a,x\}$ *and* $R\{b,x\}$ *is stable and that* $C(x,y) = C(a,b)$ *for each pair of distinct elements* $x$, $y$ *of* $C(a,b)$*. The sets* $C(a,b)$ *(for each ordered pair* $(a,b)$ *such that* $R\{a,b\}$*) 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* $A \cap B$, $B \cap C$, $C \cap A$ *is empty.*

*(b) Conversely, let* $(X_\lambda)_{\lambda \in L}$ *be a covering of a set* E *consisting of non-empty subsets of* E *having the following properties: (1) if* $\lambda$ *and* $\mu$ *are two distinct indices,* $X_\lambda \cap X_\mu$ *contains at most one element; (2) if* $\lambda$, $\mu$, $\nu$ *are three distinct letters, then at least one of the three sets* $X_\lambda \cap X_\mu$, $X_\mu \cap X_\nu$, $X_\nu \cap X_\lambda$ *is empty. Let* $R\{x,y\}$ *be the relation "There exists* $\lambda \in L$ *such that* $x \in X_\lambda$ *and* $y \in X_\lambda$*"; show that* R *is reflexive on* E*, symmetric and intransitive of order 1, and that the* $X_\lambda$ *are the constituents of* E *with respect to* R.

*(c) * Similarly, a relation* $R\{x,y\}$ *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* $R\{x_i,x_j\}$ *for each pair* $(i,j) \neq (n-1,n)$ *imply* $R\{x_{n-1},x_n\}$*. 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; apply iff_eq.
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: forall a b, a<> b -> R a b -> intransitive1 ->
  stableR (Cab a b).
Proof.
move=> a b nab Rab i1; rewrite /Cab=> u v.
rewrite !Z_rw; move=> [_ [r1 r2]] [_ [r3 r4]].
apply (i1 a b u v) =>//.
Qed.

Lemma Cab_trans: forall 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=> a b x y nab rab i1 nxy; rewrite /Cab ! Z_rw.
move=> [xE [r1 r2]] [yE [r3 r4]].
set_extens1 t; rewrite ! Z_rw; move => [tE [r5 r6]]; split =>//; 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: forall A, sub A E ->
  (inc A (quotient Sgraph) & is_singleton A) =
  (exists a, A = singleton a & forall b, R a b -> a = b).
Proof.
move=> A AE.
move: equivalence_Sgraph => e1.
move: equivalenceS => e2.
apply iff_eq.
  move=> [Asq [x Asx]]; exists x; split=>//.
  move=> b Rb.
  have : related Sgraph x b.
    rewrite /Sgraph graph_on_rw2//; exists (chain_pair x b);simpl; intuition.
  rewrite in_class_related //; move=> [y [cy [xy]]].
  have <- : A = y.
    move: Asq;aw;srw => cA; case  (class_dichot cy cA)=> //.
    move=> dy; red in dy; empty_tac1 x; apply intersection2_inc=>//.
    rewrite Asx; fprops.
  rewrite Asx; aw; intuition.
move=> [x [As Ap]]; rewrite As.
split; last by rewrite /is_singleton; exists x.
srw; split =>//.
have xE: inc x E by apply AE; rewrite As; fprops.
have ->:  (rep (singleton x) = x).
  move:   (nonempty_rep (nonempty_singleton x)); aw.
split; first by rewrite  substrate_Sgraph =>//.
set_extens1 y;  bw;aw.
  move => ->;rewrite - substrate_Sgraph in xE; equiv_tac.
move=> h; symmetry.
have : (related Sgraph y x) by equiv_tac.
rewrite /Sgraph graph_on_rw2//; move=> [c [cc []]]  <-.
elim: c cc.
  move=> u v /= uv vx;move: (A2 uv);rewrite vx; apply Ap.
by move=> p c h1 /= [Rp cc] tc; rewrite -(h1 cc tc) in Rp; apply Ap; apply A2.
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, exists b, A = Cab a b& a<> b & R a b).

Lemma constituant_inter2 : forall A B,
  is_constituant A -> is_constituant B -> intransitive1 ->
  A = B \/ small_set (intersection2 A B).
Proof.
move=> A B 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 aw; move=> [ua _][ub _]; rewrite ua ub.
```

```
case cB.
   move=>[c [Ac [cE cp]]]; rewrite Ac.
  by  aw; move=> _ [_ ua][_ ub]; rewrite ua ub.
move=> [a [b [Aab [nab Rab]]]] [a' [b' [Aab' [nab' Rab']]]].
case (equal_or_not u v)=>// nuv.
rewrite Aab Aab';aw; move => [uA uB][vA vB].
elim 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 : forall A B C,
  is_constituant A -> is_constituant B -> is_constituant C ->  intransitive1 ->
  A = B \/ A = C \/ B = C \/  intersection2 A B = emptyset
  \/ intersection2 A C = emptyset \/ intersection2 B C = emptyset
  \/ (intersection2 A B = intersection2 A C  &
      intersection2 B C =  intersection2 A C).
Proof.
move=> A B C 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 (intersection2 A B).
  case (constituant_inter2 cA cB i1) =>//.
have ssAC: small_set (intersection2 A C).
  case (constituant_inter2 cA cC i1) =>//.
have ssBC: small_set (intersection2 B 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, teh case wheer one set is a singleton is easy.

```
move: cA;case.
  move=> [a [Aa [aE ap]]]; case cB.
    move=> [b [Bb [bE bp]]].
    left; apply disjoint_pr=> u; rewrite Aa Bb; aw; move=> ->.
    by move=> ab; elim nAB; rewrite Aa Bb ab.
  move => [b1 [b2 [Bbb [nbb Rbb]]]].
  left; apply disjoint_pr => u; rewrite Aa Bbb; aw; move=> ->.
  rewrite /Cab Z_rw; move=> [_ [R1 R2]]; elim nbb.
  by rewrite -(ap _ (A2  R1)) (ap _ (A2 R2)).
move => [a1 [a2 [Aaa [naa Raa]]]].
move: cB; case.
  move=> [b [Bb [bE bp]]]; case cC.
    move=> [c [Cc [cE cp]]].
    right;right;left; apply disjoint_pr=> u; rewrite Bb Cc; aw; move=> ->.
    by move=> bc; elim nBC; rewrite Bb Cc bc.
  move => [c1 [c2 [Ccc [ncc Rcc]]]].
  right; right;left; apply disjoint_pr => u; rewrite Bb Ccc; aw; move=> ->.
  rewrite /Cab Z_rw; move=> [_ [R1 R2]]; elim ncc.
  by rewrite -(bp _ (A2  R1)) (bp _ (A2 R2)).
move => [b1 [b2 [Bbb [nbb Rbb]]]].
move: cC; case.
  move=> [c [Cc [cE cp]]].
  right;left;apply disjoint_pr => u uA uC; move: uC uA; rewrite Aaa Cc; aw.
  move=> ->;  rewrite /Cab Z_rw; move=> [_ [R1 R2]]; elim 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 (intersection2 A B));[ by left |  move=> [c ci]; right].
case (emptyset_dichot (intersection2 A C));[ by left |  move=> [b bi]; right].
case (emptyset_dichot (intersection2 B C));[ by left |  move=> [a ai]; right].
have iAB: intersection2 A B= singleton c.
  set_extens1 x; rewrite singleton_rw;
     [move=> xs; apply (ssAB _ _ xs ci) |  by move=> -> ].
have iAC: intersection2 A C = singleton b.
  set_extens1 x; rewrite singleton_rw;
     [move=> xs; apply (ssAC _ _ xs bi) |  by move=> -> ].
have iBC: intersection2 B C = singleton a.
  set_extens1 x; rewrite singleton_rw;
     [move=> xs; apply (ssBC _ _ xs ai) |  by move=> -> ].
rewrite iAB iAC iBC.
suff: (inc c C).
   move=> cC.
   have cAC: inc c (intersection2 A C) by move: ci; aw; intuition.
   have cBC: inc c (intersection2 B C) by move: ci; aw; intuition.
   by rewrite (ssAC _ _ bi cAC) (ssBC _ _ ai cBC).
case (equal_or_not a b).
  move=> ab.
  have: inc a (intersection2 A B) by move: bi ai; rewrite -ab; aw; intuition.
  rewrite iAB; aw; move => <-; apply (intersection2_second 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; aw; move=> [aB aC][bA bC][cA cB].
move: cA cB bA bC aB aC; rewrite Aaa Bbb Ccc /Cab ! Z_rw.
move=> [cE [Ra1c Ra2c]][_ [Rb1c Rb2c]][_ [Ra1b Ra2b]][_ [Rc1b Rc2b]].
move=> [_ [Rb1a Rb2a]][_ [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.
intuition.
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 (intersection2 A B))
  & (forall A B C, inc A X -> inc B X -> inc C X ->
    ( A=B \/ A = C \/ B = C \/ intersection2 A B = emptyset
      \/ intersection2 A C = emptyset
      \/ intersection2 B C = emptyset
      \/ (intersection2 A B = intersection2 A C &
      intersection2 A B = intersection2 B 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: forall E X,
  exercise6_11b_assumption X E -> reflexive_r (exercise6_11b_rel X) E.
Proof.
move=> E X [h _] x; rewrite /exercise6_11b_rel -h;apply iff_eq.
  move => xE; move: (union_exists xE)=> [y [ye xy]];ex_tac.
move=> [y [yX [xy _ ]]]; apply (union_inc xy yX).
Qed.

Lemma exercise6_11b2: forall X,
   symmetric_r (exercise6_11b_rel X).
Proof.
move=> E X y; rewrite /exercise6_11b_rel.
move=>[A [Ax [xA yA]]];  exists A; intuition.
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: forall 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=>E X [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; intuition; ue.
case (equal_or_not XZ YT) => XZYT; first by exists XZ; intuition; ue.
case (equal_or_not YZ XT) => YZXT; first by exists XT; intuition; ue.
case (equal_or_not YZ YT) => YZYT; first by exists YT; intuition; ue.
have iXZXT:  (intersection2 XZ XT = singleton x).
  have xi:  (inc x (intersection2 XZ XT))  by aw; intuition.
  case (i2 _ _ XZX XTX) =>h; first by contradiction.
  set_extens1 a;  rewrite singleton_rw => ai; [apply (h _ _  ai xi) | ue].
have iYZYT:  (intersection2 YZ YT = singleton y).
  have yi:  (inc y (intersection2 YZ YT))  by aw; intuition.
  case (i2 _ _ YZX YTX) =>h; first by contradiction.
  set_extens1 a;  rewrite singleton_rw => ai; [apply (h _ _  ai yi) | ue].
```

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 (intersection2 XY YZ) by aw.
    have zp1:inc z (intersection2 XY YZ) by rewrite XYXZ; aw.
    case (i2 _ _ XYX YZX) =>// h; elim nyz;apply: (h _ _ yp1 zp1).
  case (equal_or_not XY YT)=> XYYT; first by exists XY; aw; ee; ue.
  case (equal_or_not XT YT) => XTYT.
    have xp: inc x (intersection2 XY YT) by aw; ue.
    have yp: inc y (intersection2 XY 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; intuition; 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 (intersection2 XT YT) by aw; intuition.
  rewrite -r2 XYYZ; aw; move=> [tp _].
  exists YZ; by aw; intuition.
```

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: (intersection2 XY XZ = singleton x).
  have px: inc x (intersection2  XY XZ) by aw; intuition.
  case  (i2 _ _ XYX XZX) => h; first by [].
  set_extens1 a;  rewrite singleton_rw => ai; [apply (h _ _  ai px) | ue].
case  (equal_or_not XZ YZ)=> XZYZ.
  have : inc y (singleton x) by rewrite -iXYXZ; aw;intuition; ue.
  aw=> h; elim nxy =>//.
have iXZYZ:  (intersection2 XZ YZ = singleton z).
  have pz: (inc z (intersection2 XZ YZ)) by aw.
  case  (i2 _ _ XZX YZX) => h; first by [].
  set_extens1 a;  rewrite singleton_rw => ai; [apply (h _ _  ai pz) | ue].
```

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 (intersection2 XY XT)  by aw.
    have tp: inc t (intersection2 XY XT) by rewrite XYYT; aw.
    case (i2 _ _ XYX XTX) =>// h; elim nxt;apply: (h _ _ xp tp).
  case (equal_or_not XY YZ)=> XYYZ; first by exists XY; aw; ee; 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; intuition.
  case; first by move=> h; empty_tac1 y; aw; intuition.
  case; first by move=> h; empty_tac1 z; aw; intuition.
  move=> [r1 r2].
  have : inc z (intersection2 XZ YZ) by aw; intuition.
  rewrite -r2 XYYT; aw; move=> [tp _].
  exists YT; by aw; intuition.
```

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: (intersection2 XY YT = singleton y).
  have px: inc y (intersection2  XY YT) by aw; intuition.
  case  (i2 _ _ XYX YTX) => h; first by [].
  set_extens1 a;  rewrite singleton_rw => ai; [apply (h _ _  ai px) | ue].
case  (equal_or_not XT YT)=> XTYT.
  have : inc x (singleton y) by rewrite -iXYYT; aw;intuition; ue.
  aw=> h; elim nxy =>//.
have iXTYT:  (intersection2 XT YT = singleton t).
  have pz: (inc t (intersection2 XT YT)) by aw.
  case  (i2 _ _ XTX YTX) => h; first by [].
  set_extens1 a;  rewrite singleton_rw => ai; [apply (h _ _  ai pz) | ue].
```

On 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; intuition.
  case; first by move=> h;empty_tac1 y; aw; intuition.
  case; first by move=> h;empty_tac1 z; aw; intuition.
  rewrite iXYXZ iXZYZ; move=> [_ sxz].
  by elim nxz; apply singleton_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; intuition.
case; first by move=> h;empty_tac1 y; aw; intuition.
case; first by move=> h;empty_tac1 t; aw; intuition.
rewrite iXYYT iXTYT; move=> [sy st].
rewrite sy in st;  by elim nyt; apply singleton_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: forall E X, exercise6_11b_assumption X E ->
  let R := exercise6_11b_rel X in
    let p1 := fun u => (exists a, exists b, a<> b & R a b & u =
      Zo E (fun x => R a x & R b x)) in
    let p2:= fun u => (exists x, u = singleton x & inc x E&
      forall y, inc y E -> R x y -> x = y) in
    let p3:= fun u => (exists v, inc v X & u <> v & sub u v & is_singleton u) in
      (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 (is_singleton u)) => su.
    right; case  (p_or_not_p (p3 u)) => p3u; first by intuition.
    left; move: (su) => [x sx].
    rewrite sx; exists x;ee.
      rewrite -uXE; apply union_inc with  u =>//; rewrite sx; fprops.
    move=> y yE Rxy; case (equal_or_not x y) =>//.
    move=> xy; move: Rxy=> [A [AX [xA yA]]].
    elim p3u; exists A; ee.
      by dneg uA; move: yA; rewrite -uA sx; aw.
    by move=> t; rewrite sx; aw; move => ->.
```

Our set $u$ is not empty, hence has an element $y$. We show here that if it has another 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).

```
  left; red.
  move: (alne _ uX) => [y yu]; exists y.
  case (p_or_not_p (exists v, inc v u & v <> y)).
    move=> [x [xu xy]]; exists x; ee; first by exists u; ee.
    set_extens1 w.
    move=> wu; apply Z_inc.
        rewrite - uXE; apply union_inc with u=>//.
      split;red;red;exists u; intuition.
    rewrite Z_rw; move => [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 (intersection2 u A')) by aw.
    have yi: (inc y (intersection2 u 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 intersection2comm -h2 in xi.
    rewrite intersection2comm -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; set_extens1 w;aw; last by move=> ->.
  move=> wu;case (equal_or_not w y) =>// wy.
  by elim h; ex_tac.
```

We show here that $p_1(u)$ implies $u \in \mathrm{X}$. 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.

```
split.
  move=> u [a [b [nab [[A [AX [aA bA]]] uZ]]]].
  suff: (u = A) by  move=> ->.
  rewrite uZ; set_extens1 t; rewrite Z_rw.
    move=> [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 (intersection2 A A') by aw.
    have bAA: inc b (intersection2 A A'') by aw.
    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; ee.
      rewrite -uXE;apply union_inc with A=>//.
    exists A; ee.
  exists A; ee.
```

We show that $p_2(u)$ implies $u \in \mathrm{X}$.

```
move=> u  [v [uv [vE su]]].
move: vE;rewrite  -uXE union_rw; move=> [y [vy yX]].
suff: u = y by move=> ->.
rewrite uv;set_extens1 t; aw; first by move=> ->.
move=> tv; symmetry; apply su.
  rewrite -uXE;apply union_inc with y =>//.
exists y; ee.
```

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.

**Original Exercises**.

```
Lemma exercise3_5: forall g a b, is_graph g ->
  (compose_graph (product a b) g = product (inv_image_by_graph g a) b &
  compose_graph g (product a b)  = product a (image_by_graph g b)).

Lemma exercise4_4b:  forall g x,
  fgraph g -> (forall i, inc i (domain g) -> is_graph (V i g)) ->
  nonempty g -> is_singleton x ->
  image_by_graph(intersectionb g) x =
  intersectionb (L(domain g) (fun i=> image_by_graph(V i g) x)).
Lemma exercise4_5:  forall g h,
  fgraph g -> (forall i, inc i (domain g) -> is_graph (V i g)) -> is_graph h->
  (compose_graph (unionb g) h =
     unionb (L(domain g) (fun i=> compose_graph(V i g) h))
     & compose_graph h (unionb g) =
     unionb (L(domain g) (fun i=> compose_graph h (V i g)))).

Lemma exercise4_7:  forall G H K, is_graph G ->is_graph H ->is_graph K ->
  sub(intersection2 (compose_graph H G) K)
     (compose_graph(intersection2 H (compose_graph K (inverse_graph G)))
       (intersection2 G (compose_graph (inverse_graph H) K))).
```

## 8.7 The cardinals, according to Zermelo, 1908

We implement here the a part of the theory of Zermelo as described in his paper "Investigations in the foundations of set theory I". It has seven axioms: Axiom I is the axiom of extent, axiom II corresponds to the axiom of the pair, Axiom III is the axiom of separation SC, axiom IV is the axiom of the powerset, Axiom V is the axiom of the union, axiom VI is the axiom of choice and axiom VII the axiom of infinity. The bold face numbers are the paragraph numbers of the Zermelo paper.

The notations have changed, the union and intersection of two sets is $\mathrm{A}+\mathrm{B}$ and $[\mathrm{A},\mathrm{B}]$, while the union and intersection of a family is $\mathfrak{S}\mathrm{A}$ and $\mathfrak{D}\mathrm{B}$; the powerset is denoted by $\mathfrak{U}\mathrm{T}$. Note that Zermelo uses no ordered pairs, thus no graphs, and his products are different from our products. Here is the definition.

**13.** "Let T be a set whose elements $\mathrm{M},\mathrm{N},\mathrm{R},\ldots$ are various (mutually disjoint) sets, and let $\mathrm{S}_1$ be any subset of its union $\mathfrak{S}\mathrm{T}$. Then it is definite for every element M of T whether the intersection $[\mathrm{M},\mathrm{S}_1]$ consists of a single element or not. Thus all those elements of T that have exactly one element in common with $S_1$ are the elements of a certain subset $T_1$ of T, and it is again definite whether $T_1 = T$ or not. All subsets $S_1$ of $\mathfrak{S}T$ that have exactly one element in common with each element of T then are, according to Axiom III, the elements of a set $P = \mathfrak{P}T$, which according to axioms III and IV is a subset of $\mathfrak{U}\mathfrak{S}T$ and will be called the connection set associated with T, or the product of the sets M, N, R, .... If $T = \{M, N\}$ we write $\mathfrak{P}T = MN$."

Here is the definition of a connection or a product of two sets. We do not require the sets to be disjoint for the definition. In fact, the product of $n$ sets is formed of sets with at most $n$ elements, and we have exactly $n$ elements if the sets are disjoint. Note that the product is independent of the order of the factors.

```
Definition zprod a := Zo (powerset (union a))
  (fun y => forall x, inc x a -> is_singleton (intersection2 y x)).
Definition zprod2 a b:= zprod (doubleton a b).
```

We have $X \in AB$ if and only if $X \subset A \cup B$ and the two sets $X \cap A$ and $X \cap B$ are singletons.

```
Lemma zprod2_pr: forall a b y,
  inc y (zprod2 a b) =
  (sub y (union2 a b) &
    is_singleton (intersection2 y a) &
    is_singleton (intersection2 y b)).
```

We denote by $s_X(A)$ the unique element of $X \cap A$. If $X \in AB$, then $X \cap A = \{s_X(A)\}$ and $X \cap B = \{s_X(B)\}$. For this, we deduce that $s_X(A)$ is in A and X, and also that $s_X(B)$ is in B and X.

```
Definition zpr x a := choose (fun z => (intersection2 x a) = singleton z).
Lemma zpr_prop : forall x a,
  is_singleton (intersection2 x a) -> (intersection2 x a = singleton (zpr x a)).
Lemma zprod2_pr1: forall a b x,
  inc x (zprod2 a b) ->
  ((intersection2 x a = singleton (zpr x a)) &
  (intersection2 x b = singleton (zpr x b))).
Lemma zprod2_pr0: forall a b x,
  inc x (zprod2 a b) ->
  (inc (zpr x a) a & inc (zpr x b) b & inc (zpr x a) x & inc (zpr x b) x).

Lemma zprod2_pr0aa: forall a b x,
  inc x (zprod2 a b) ->inc  (zpr x a) a.
Lemma zprod2_pr0ax: forall a b x,
  inc x (zprod2 a b) ->inc  (zpr x a) x.
Lemma zprod2_pr0bb: forall a b x,
  inc x (zprod2 a b) -> inc  (zpr x b) b.
Lemma zprod2_pr0bx: forall a b x,
  inc x (zprod2 a b) ->inc  (zpr x b) x.
```

Conversely, an element in A and X must be $s_X(A)$. From this we deduce $X = \{s_X(A), s_X(B)\}$. If $x \in A$ and $x \in X$, then $s_X(A) = x$.

```
Lemma zprod2_pr1a: forall a b x z,
  inc x (zprod2 a b) ->
  inc z a -> inc z x -> z = zpr x a.
Lemma zprod2_pr1b: forall a b x z,
  inc x (zprod2 a b) ->
  inc z b -> inc z x -> z = zpr x b.
Lemma zprod2_pr2: forall a b y,
  inc y (zprod2 a b) ->
  y = doubleton (zpr y a) (zpr y b).
```

We characterize here the product AB when B is a singleton $\{b\}$, as the set of all $\{a,b\}$ for $a \in$ A.

```
Lemma intersection_singleton: forall a b c,
  (intersection2 a (singleton b) = singleton c) =
  (inc c a & c = b).
Lemma is_singleton_int: forall a b c,
  inc c a -> inc c b -> (forall u, inc u a-> inc u b -> u = c) ->
  is_singleton (intersection2 a b).
Lemma zprod_singleton: forall M r, ~ inc r M ->
  let N := zprod2 M (singleton r) in
    ( (forall u, inc u M -> inc (doubleton u r) N) &
      (forall x, inc x N -> exists u, inc u M & x = doubleton u r)).
```

**15.** "A *mapping of* M *onto* N is a subset $\phi$ of the product MN such that each element of M+N occurs as an element in one and only one element $\{m,n\}$ of $\phi$. Two elements $m$ and $n$ that occur together in one element of $\phi$ are said to be 'mapped onto each other'. Two sets are said to be *immediately equivalent* if there exists at least one such $\phi$."

The definition is symmetric with respect to M and N. The union of these two sets is uniquely determined by $\phi$ as the union of $\phi$. Zermelo assumes the two sets disjoint. In fact, if M = N, there is only one mapping, the set of all singletons. We add the disjointness condition to the definition of "equivalent" (but this really changes nothing).

```
Definition zmap f a b := sub f (zprod2 a b) &
  (forall x, inc x (union2 a b) -> exists_unique (fun z => inc z f & inc x z)).
Definition ziequivalent a b := disjoint a b & exists f, zmap f a b.

Lemma zmap_symm: forall f a b,
  zmap f a b -> zmap f b a.
Lemma zequiv_symm: forall a b, ziequivalent a b -> ziequivalent b a.
Lemma zmap_pr1 : forall f a b, zmap f a b ->
  union f = union2 a b.
```

Examples: disjoint singletons are equivalent as well as disjoint doubletons.

```
Lemma zmap_example1 : forall a b, a <> b ->
  let A := singleton a in
    let B := singleton b in
    zmap (singleton (doubleton a b)) A B.
Lemma zmap_example2 : forall a b c d, let A := doubleton a b in
  let B := doubleton c d in
    disjoint A B -> a <> b -> c <> d ->
    zmap (doubleton (doubleton a c) (doubleton b d)) A B.
Lemma zequiv_example1 :  forall a b, a <> b ->
  ziequivalent (singleton a) (singleton b).
Lemma zequiv_example2 : forall a b c d, let A := doubleton a b in
  let B := doubleton c d in
    disjoint A B -> a <> b -> c <> d ->
    ziequivalent A B.
```

Assume $f(a) \in B$ whenever $a \in A$, where A and B are two disjoint sets. The set of all $z \in A \cup B$ of the form $\{a, f(a)\}$ with $a \in A$ is an element of AB. Assume $f$ bijective; this set is then a mapping.

```
Definition zbijective F a b :=
  (  (forall x, inc x a -> inc (F x) b)
    & (forall x x', inc x a -> inc x' a -> F x = F x' -> x = x')
    & (forall y, inc y b -> exists x, inc x a & F x = y)).

Lemma zmap_example3: forall F a b, disjoint a b -> zbijective F a b ->
  zequivalent a b.
```

Let's denote by $w_f(x)$ the element such that $w_f(x) \in f$ and $x \in w_f(x)$. If $f$ is a mapping, $x \in A \cup B$, such an object exists and is unique. Wer can restate this as: if $x$ and $y$ are in $f$, if $s_x(A) = s_y(A)$ then $x = y$; the same holds for B.

```
Definition zmap_aux f x := choose (fun z =>  inc z f &inc x z  ).
Lemma zmap_aux_pr1: forall f a b x,
  zmap f a b -> inc x (union2 a b) ->
  (inc (zmap_aux f x) f & inc x (zmap_aux f x)).
Lemma zmap_aux_pr2: forall f a b x y,
  zmap f a b -> inc x (union2 a b) -> inc y f -> inc x y
  -> y = (zmap_aux f x).
Lemma zmap_aux_pr3a: forall f a b x y,
  zmap f a b -> inc x f -> inc y f ->  zpr x a = zpr y a
  -> x = y.
Lemma zmap_aux_pr3b: forall f a b x y,
  zmap f a b -> inc x f -> inc y f ->  zpr x b = zpr y b
  -> x = y.
```

Consider now $s_A(w_f(x))$. This is the unique element of A in $w_f(x)$. This is obviously $x$ if $x \in A$. It is also defined for $x \in B$. We shall sometimes denote this by $f_A(x)$. Similarly $f_B(x)$. We have $f_A(f_B(x)) = x$ if $x \in A$ and $f_B(f_A(x)) = x$ if $x \in B$. This implies that $f_B$ is bijective (in the sense given above).

```
Definition zmap_val f a x:= zpr (zmap_aux f x) a.

Lemma zmap_val_pr1a : forall f a b x, zmap f a b ->  inc x (union2 a b) ->
  inc (zmap_val f a x) a.
Lemma zmap_val_pr1b : forall f a b x, zmap f a b ->  inc x (union2 a b) ->
  inc (zmap_val f b x) b.

Lemma zmap_val_pr1 : forall f a b x, zmap f a b ->  inc x (union2 a b) ->
  (inc (zmap_val f a x) a & inc (zmap_val f b x) b).
Lemma zmap_val_pr2a : forall f a b x, zmap f a b ->  inc x a ->
  (zmap_val f a x) = x.
Lemma zmap_val_pr2b : forall f a b x, zmap f a b ->  inc x b ->
  (zmap_val f b x) = x.

Lemma zmap_val_pr3a : forall f a b x, zmap f a b ->  inc x a ->
  (zmap_val f a (zmap_val f b x)) = x.
Lemma zmap_val_pr3b : forall f a b x, zmap f a b ->  inc x b ->
  (zmap_val f b (zmap_val f a x)) = x.

Lemma zmap_bijective: forall f a b, zmap f a b ->
  zbijective (zmap_val f b) a b.
```

**16.** Zermelo says: “It is definite for two disjoint sets whether they are equivalent or not”, since this is the same checking whether a set Ω is empty or not. We just show here that the set exists.

```
Lemma zmap_set: forall a b, exists s,
  forall f, zmap f a b = inc f s.
```

**17.** If $\phi$ is a mapping $M \to N$, and $M_1$ is a subset of M, there exists a subset $N_1$ of N which is equivalent to $M_1$ via a subset of $\phi$. Note: Zermelo assumes the sets disjoint. We start with a lemma that says that if M and N are disjoint, so are $M_1$ and $N_1$. The equivalence is the set of all $X \in \phi$ such that $s_X(M) \in M_1$, and the set $N_1$ is the set of objects $z \in N$ of the form with $z = s_X(N)$ for some X satisfying this condition. Note that Zermelo doe not use the Replacement Axiom that asserts the existence of the set $\{f(x), x \in A\}$, this makes the definition a bit more complex.

```
Lemma sub_disjoint: forall a b a' b',
  disjoint a b -> sub a' a -> sub b' b -> disjoint a' b'.
Lemma zmap_sub: forall f a b a', zmap f a b -> sub a' a ->
  exists f', exists b',
    (sub f' f & sub b' b & zmap f' a' b').

Lemma zequiv_sub: forall a b a', ziequivalent a b -> sub a' a ->
  exists b',  (sub b' b & ziequivalent a' b').
```

**18.** Assume that $f$ maps A to B, and $g$ maps B to C. For instance, we map $\{a, b\}$ to $\{c, d\}$ and then to $\{b, a\}$, where all four elements aree distinct. Composition is the permutation on $\{a, b\}$. But this is not a mapping according to Zermelo. Thus, in the following lemma, we assume A and C disjoint. The set of all $z = \{s_x(A), s_y(C)\}$ in $\mathfrak{P}(A \cup C)$ such that there exist $x \in f$ and $y \in g$ such that $s_x(B) = s_y(B)$ is a mapping $A \to C$. We give here a simpler proof: both $f_A$ an $g_B$ are bijective, hence the composition is also bijective. Thus we have a bijection $A \to C$, which gives a mapping, since A and C are disjoint.

```
Lemma zmap_transitive: forall a b c, disjoint a c ->
  zequivalent a b -> zequivalent b c -> zequivalent a c.
```

**19.** In ¶10, Zermelo says that for every set M there is a set N such that $N \subset M$ and $N \not\in M$. We can restate this without quantifiers as: the set $\{N \in \mathfrak{P}(M), N \not\in M\}$ is non-empty. It suffices in fact to take the set of all elements of M that do not belong to themeselves. This theorem can be used as: for every set M there is a set N such that $N \not\in M$.

Thus, given M and N, there exists $r$ not in M such that, if $R = \{r\}$, the set MR is disjoint from M and N (an element $x$ of MR has the form $\{y, r\}$, if it is in M or N, it is in $M \cup N$ and $r$ is in the union of this set. The function $x \mapsto \{x, r\}$ is bijective, thus we get a mapping $M \to MR$.

It follows that, for any M and N, there exists $M'$ disjoint from M and N, equivalent to M.

```
Lemma disjointness : forall M, exists N,
  sub N M & ~ inc N M.
Lemma disjointness1 : forall M N, exists r,
  let M1 := zprod2 M (singleton r) in
   ( ~ (inc r M)  & disjoint M M1 & disjoint N M1).
Lemma zmap_example4: forall M r,
  let N := zprod2 M (singleton r) in
```

```
    ~ (inc r M) -> disjoint M N ->
    ziequivalent M N.
Lemma zequiv_example4: forall M N, exists M',
  disjoint N M' & ziequivalent M M'.
```

**19.** Zermelo deduces that there is no set containing all sets equivalent to M, since if T is such a set, we have a set $x$ equivalent to M disjoint from ∪T, thus cannot be in T (note that this argument does not hold if $x$ is empty, so that we start with a lemma: if M is empty, so is $x$).

```
Lemma zequiv_empty: forall M, ziequivalent M emptyset -> M = emptyset.

Lemma zequiv_no_graph: forall M, nonempty M ->
  ~ (exists S, forall M',  ziequivalent M M' -> inc M' S).
```

**21.** Zermelo says that is “definite” the existence of a set R disjoint from both sets M and N and equivalent to them. This justifies the following definition of “mediately equivalent”. This is an equivalence relation.

```
Definition zequiv  M N := exists R, ziequivalent M R &  ziequivalent N R.

Lemma zequiv_reflexive: forall M, zequiv M M.
Lemma zequiv_symmetric: forall M N,
   zequiv  M N -> zequiv  N M.
Lemma zequiv_transitive: forall M N P,
   zequiv  M N -> zequiv N  P -> zequiv M P.
```

# Chapter 9

# Summary

## 9.1 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 $\boldsymbol{A}) \implies \boldsymbol{A}$ is an axiom of $\mathscr{T}$.
S2: If $\boldsymbol{A}$ and $\boldsymbol{B}$ are relations in $\mathscr{T}$, the relation $\boldsymbol{A} \implies (\boldsymbol{A}$ 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 $\boldsymbol{B}) \implies (\boldsymbol{B}$ 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 $(\boldsymbol{A} \implies \boldsymbol{B}) \implies ((\boldsymbol{C}$ or $\boldsymbol{A}) \implies (\boldsymbol{C}$ 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 $(\boldsymbol{T}|\boldsymbol{x})\boldsymbol{R} \implies (\exists \boldsymbol{x})\boldsymbol{R}$ 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 $(\boldsymbol{T} = \boldsymbol{U}) \implies (\boldsymbol{R}\{\boldsymbol{T}\} \iff \boldsymbol{R}\{\boldsymbol{U}\})$ 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 $((\forall \boldsymbol{x})(\boldsymbol{R} \iff \boldsymbol{S})) \implies (\tau_{\boldsymbol{x}}(\boldsymbol{R}) = \tau_{\boldsymbol{x}}(\boldsymbol{S}))$ 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

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

is an axiom.

The French edition has only four axioms since A3 is a theorem.
A1. $(\forall x)(\forall y)((x \subset y$ and $y \subset x) \implies (x = y))$.
A2. $(\forall x)(\forall y)\mathrm{Coll}_z(z = x$ or $z = y)$.
A3. $(\forall x)(\forall x')(\forall y)(\forall y')(((x, y) = (x', y')) \implies (x = x'$ and $y = y'))$
A4. $(\forall \mathrm{X})\mathrm{Coll}_{\mathrm{Y}}(\mathrm{Y} \subset \mathrm{X})$.
A5. There exists an infinite set.

## 9.2 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 $\tau$, thus is missing a choice function. With the notations of [5] the axioms are

B1. $\forall x \forall y[\forall z(z \in x \iff z \in y) \implies x = y]$ (Axiom of extent, A1).

B0. $\forall x \forall y \exists z \forall t[t \in z \iff (t = x$ 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. $\exists x \exists y[\forall z(z \not\in y) \text{ and } y \in x \text{ and } \forall u[u \in x \implies \exists v[v \in x \text{ and } \forall t(t \in v \iff t = u \text{ or } t \in u)]]]$ (Axiom of infinity).

SS. $\forall x_1 \dots \forall x_k\{\forall x \forall y \forall y'[\mathrm{E}(x, y, x_1, \dots, x_k) \text{ and } \mathrm{E}(x, y', x_1, \dots, x_k) \implies y = y'] \implies \forall t \exists w \forall v[v \in w \iff \exists u[u \in t \text{ and } \mathrm{E}(u, v, x_1, \dots, x_k)]]\}$ (Scheme of Replacement).

SC. $\forall x_1 \dots \forall x_k \forall x \exists y \forall z[z \in y \iff (z \in x \text{ and } \mathrm{A}(z, x_1, \dots, x))]$ (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-Frankel 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.

## 9.3 Changes from previous versions

We show here a list of definitions and theorems, there were either removed or changed, with some explanations. In the definitions that follow, *E* means *Set* , *EP* means *Set*→ *Prop* and *EE* means *Set*→ *Set*.

**Reasoning by cases.** In the original version, we had an axiom that says: for any proposition P, if $p$ and $p'$ are two proofs of P, then $p = p'$. Let $a(x)$ be function whose argument $x$ is of type P. We have then $a(p) = a(p')$, and we can denote this by $a(\mathrm{P})$. Similarly, if $b(x)$ is a function whose argument is of type ¬P we can define $b(\neg\mathrm{P})$. The function *by_cases* selects one of $a(\mathrm{P})$ or $b(\neg\mathrm{P})$. We give here the original properties of this function.

```
Axiom proof_irrelevance : forall (P : Prop) (q p : P), p = q.
Lemma by_cases_exists :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T),
    exists x : T, by_cases_pr a b x.
Lemma by_cases_property :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T),
  by_cases_pr a b (by_cases a b).
Lemma by_cases_unique :
  forall (T : Type) (P : Prop) (a : P -> T) (b : ~ P -> T) (x : T),
    by_cases_pr a b x -> by_cases a b = x.
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.
```

**The choose function.** Let $x$ be a set, $p(x)$ a property of sets. Let $\mathrm{Q}(p,x)$ 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 $\mathrm{Q}(p,x)$ is equivalent to $p(x)$ (resp. $x = \emptyset$). In both cases, there exists $x$ such that $\mathrm{Q}(p,x)$ is true. By the excluded middle axiom, one of these two cases must be true. This allow us to define the choice function. This definition makes sense only if we can prove that there exists at least one set; in the original version a set was a *Type* and C. Simpson used *Prop*, this definition uses *Type*. Later one we changed the type of a set to *Set*, and our witness to $\mathbb{N}$, then to $\emptyset$.

```
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 Type).
Definition choose (p:EP) := choose' (refined_pr p).
```

We give here two special cases of the axiom of choice. We consider two types A and B, and a function $f : \mathrm{A} \to \mathrm{B}$; the question is whether there exists a function $g : \mathrm{B} \to \mathrm{A}$ that is the inverse of $f$. This function satisfies $f(g(x)) = x$ or $g(f(x)) = x$. In the first case $f$ must be surjective and $g$ is called a right inverse, in the second case $f$ must be injective and $g$ is called a left inverse. We construct a left inverse by defining $g(y)$ to be some element $u$ of A, if $y$ is not in the the range of $f$, or any $x$ such that $y = f(x)$ otherwise. In the current version the condition $y = u$ has been removed, making the definition simpler; of course, we still need the assumption that A is non-empty. In the case of a right inverse, we removed the auxiliary lemma (which makes the defintion more complicated).

```
Lemma left_inverseC_aux: forall (a b:Set) (f: a->b)(w:a) (v:b),
  exists u:a, (~ (exists x:a, f x = v) & u  = w)  \/ (f u = v).
Definition chooseT_any a (H:nonemptyT a):= (chooseT (fun x:a => x=x) H).

Definition left_inverseC (a b:Set) (f: a->b)(H:nonemptyT a)
  (v:b) := (chooseT (fun u:a => (~ (exists x:a, f x = v) & u  = chooseT_any H)
    \/ (f u = v)) H).
Lemma inverse_value_ex: forall (a b:Set) (f: a->b) (H:surjectiveC f) (x:b),
  nonemptyT a.
Definition right_inverseC (a b:Set) (f: a->b)  (H:surjectiveC f) (x:b) :=
  (chooseT (fun k:a => f k = x) (inverse_value_ex H x)).
```

**Complement.** The first lemma is used to show the second, that says that if it is not the case that both complements are non-empty, then one complement is non-empty, i.e., one set is included in the other.

```
Lemma show_sub_or_aux: forall b c,
   ~ (sub b c \/ sub c b) -> nonempty (complement c b).
Lemma show_sub_or : forall b c,
   (nonempty (complement b c) -> nonempty (complement c b) -> False) ->
   sub b c \/ sub c b.
Lemma non_nonempty_comp_sub: forall a b,
  ~ nonempty (complement a b) -> sub a b.
```

**Pairs.** Our initial definition of a pair made it a set with exactly two elements. This property is not used anymore, this allowed us to change the defintion of a pair, and the following is not true anymore.

```
Lemma inc_pair1: forall x y, inc (pair_first x y) (J x y).
Lemma inc_pair2: forall x y, inc (pair_second x y) (J x y).
Lemma pair_extensionalitya: forall x y u,
  inc u (J x y) -> u = pair_first x y \/ u = pair_second x y.
```

**Cuts.** (This section removed in V4) Let $p$ be the property that $x$ is even. The construction *cut* allows us to define $\mathbb{N}_2$ as the set of all even integers. This means $y \in \mathbb{N}_2$ if $\mathscr{R}y$ is even. Since 0 is even we have $z \in \mathbb{N}_2$, where $z = \mathscr{R}0$. Let $0_2$ be the object of type $\mathbb{N}_2$ such that $\mathscr{R}0_2 = z$. The construction *cut_to* take $0_2$ as argument and returns 0. Later on, we shall see that for any inclusion like $\mathbb{N}_2 \subset \mathbb{N}$ there is a function of type $\mathbb{N}_2 \to \mathbb{N}$.

We define *cut x p* as the set of all elements $y \in x$ such that if H is a proof of $y \in x$, and $z = \mathscr{B}$H, then $p(z)$ is true, (here $p$ is a property on the type $x$). Note that $\mathscr{R}z = y$. Argument $x$ is implicit.

```
Definition cut (x : Set) (p : x -> Prop) :=
  Zo x (fun y : Set => forall hyp : inc y x, p (Bo hyp)).

Lemma cut_sub : forall (x : Set)(p : x -> Prop), sub (cut p) x.
Lemma cut_inc : forall (x : Set)(p : x -> Prop)(y : x), p y -> inc (Ro y) (cut p).
```

Let $y$ be of type *cut p*. In this case *R_inc y* is a proof that $\mathscr{R}y \in$ C. If we apply *cut_sub* we get a proof that $\mathscr{R}y \in x$. We apply $\mathscr{B}$ to this. If this gives $z$, then $z$ is of type $x$, $\mathscr{R}z = \mathscr{R}y$ and $p(z)$ is true.

```
Definition cut_to (x : Set) (p : x -> Prop) (y : cut p) :=
  Bo (cut_sub (p:=p) (R_inc y)).

Lemma cut_to_R_eq : forall (x:Set)(p: x -> Prop) (y: cut p), Ro (cut_to y) = Ro y.
Lemma cut_pr : forall (x : Set) (p : x -> Prop) (y : cut p), p (cut_to y).
```

Let $f$ be a function defined on the type $a$. If $x$ is of type *IM f*, there is some $y : a$ such that $f(y) = \mathscr{R}x$. The function *IM_lift* uses the axiom of choice, and returns such an $y$.

```
Definition IM_lift : forall (a : Set) (f : a -> Set), IM f -> a.
 ir. assert (exists x : a, f x = Ro H). ap IM_exists. ap R_inc.
 assert (nonemptyT a). induction H0. app nonemptyT_intro.
 exact (chooseT (fun x : a => f x = Ro H) H1).
Defined.

Lemma IM_lift_pr :
  forall (a : Set) (f : a -> Set) (x : IM f), f (IM_lift x) = Ro x.
```

**Module Basic Realization.** (This module has been withdrawn in version 2) The four axioms of this paragraph have been introduced by C. Simpson. The first two axioms imply that $\mathscr{R}n$ is the $n$-th ordinal (in the von Neumann sense) for each natural number $n$, these are $\emptyset$, $\{\emptyset\}$, $\{\emptyset,\{\emptyset\}\}$ 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.
```

These two axioms say that $\mathscr{R}$P is P for every proposition, and $\mathscr{R}p$ is the empty set whenever $p$ is a proof of *True*. These axioms are currently ill-typed.

```
Axiom prop_realization : forall x : Prop, Ro x = x.
Axiom true_proof_realization_empty : forall t : True, Ro t = Ro 0.
```

The first lemma says that $\emptyset$ is *False*. Note that the objects have different types, but are equal by extensionality. The realization of *False* is itself, hence the empty set. The realization of every $t$ of type *True* is the empty set. By extensionality, this implies that *True* is the singleton $\{\emptyset\}$. The realization of *True* is itself.

```
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.
```

By definition $\mathscr{R}2$ has two elements, which are $\emptyset$ and $\{\emptyset\}$, said otherwise, *True* and *False*. Since every proposition is *True* or *False*, we get that $\mathscr{R}2$ is *Prop*.

```
Lemma R_two_prop : Ro 2 = Prop.
```

**Module Transposition.** (This module has been withdrawn in version 3).

We define here a function $T_{i\,j\,a}$ that maps $i$ to $j$, $j$ to $i$ and everything else to $a$. All lemmas given here are obvious.

```
Definition create (i j a : Set) := Yo (a = i) j (Yo (a = j) i a).

Lemma not_i_not_j : forall i j a, a <> i -> a <> j -> create i j a = a.
Lemma i_j_j_i : forall i j a, create i j a = create j i a.
Lemma i_j_i : forall i j, create i j i = j.
Lemma i_j_j : forall i j, create i j j = i.
Lemma i_i_a : forall i a, create i i a = a.
Lemma surj : forall i j a, exists b, create i j b = a.
Lemma involutive : forall i j a, create i j (create i j a) = a.
Lemma inj : forall i j a b, create i j a = create i j b -> a = b.
```

**Module Bounded.** This module has been withdrawn in version 3, after changing definition of the cartesian product. The module say that a relation $p$ is collectivizing, under certain conditions, which are related to the Replacement Axiom.

Let $p$ be a predicate and $x$ a set. Assume $p(y)$ is true if and only if $y \in x$. We say that $p$ satisfies the axioms if there is such a set $x$. This set is obviously unique, and will be denoted by *create p*.

```
Definition property (p : Set -> Prop) (x : Set) :=
  forall y : Set, (p y -> inc y x & inc y x -> p y).
Definition axioms (p : Set -> Prop) := ex (property p).
Definition create (p : Set -> Prop) := choose (property p).
```

If $p$ satisfies the axioms then $y$ is in *create p* if and only if $p(y)$. If $p$ is bounded (we can consider different cases) then $p$ satisfies the axioms.

```
Lemma lem1 : forall (p : EP) (y : Set), axioms p -> inc y (create p) -> p y.
Lemma lem2 : forall (p : EP) (y : Set), axioms p -> p y -> inc y (create p).
Lemma inc_create : forall (p : EP) y, axioms p -> inc y (create p) = p y.

Lemma criterion : forall p : EP,
 ex (fun x : Set => forall y : Set, p y -> inc y x) -> axioms p.

Lemma trans_criterion :
  forall (p : EP) (f : EE) (x : Set),
    (forall y : Set, p y -> ex (fun z : Set =>  (inc z x) & (f z = y))) ->
    axioms p.

Lemma little_criterion :
  forall (p : EP) (x : Set) (f : x -> Set),
    (forall y : Set, p y -> exists a : x, f a = y) -> axioms p.
```

**Cartesian Product.** We explain here the initial implementation of the cartesian product; it depended on the module *Bounded*, that has been withdrawn.

Consider two sets X and Y. For every $y \in Y$ we can consider the set of all pairs $(x, y)$ with $x \in X$. We can consider the union of these sets. It is denoted by $X \times Y$. Bourbaki defines the product before the union, but uses the same argument as for the union to show existence of the product.

We consider here the following property: Let $a$ be a set and $f$ a function. We say that $z$ is in the record if $z$ is a pair, say $z = (x, y)$, $x \in a$ and $y \in f(x)$. A *Cartesian_record* holds $u$ and $v$, where $u$ is of type $a$, and $v$ of type $f(\mathscr{R}u)$. Given such an object $i$ we can create the pair $(\mathscr{R}u, \mathscr{R}v)$. Let's denote it by $g(i)$. If $z$ is in the record, it is of the form $g(i)$. This shows that *in_record* is bounded.

```
Definition in_record (a : Set) (f : EE) (x : Set) :=
  is_pair x  & inc (P x) a   & inc (Q x) (f (P x)).

Record Cartesian_record (a : Set) (f : EE) : Set :=
  {Cartesian_first : a; Cartesian_second : f (Ro Cartesian_first)}.

Definition recordMap (a : Set) (f : EE) (i : Cartesian_record a f) :=
  J (Ro (Cartesian_first i)) (Ro (Cartesian_second i)).

Lemma in_record_ex : forall (a : Set) (f : EE) (x : Set),
 in_record a f x -> exists i : Cartesian_record a f, recordMap i = x.

Lemma in_record_bounded :
  forall (a : Set) (f : EE), Bounded.axioms (in_record a f).
```

The record $\mathrm{R}(a, f)$ of $a$ and $f$ is the set of all pairs $(x, y)$ where $x \in a$ and $y \in f(x)$. Following lemmas are trivial.

```
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 is constant.

```
Definition product (a b : Set) := record a (fun x : Set => b).
```

**Module Back.** This module has been withdrawn in version 2.

We define *RBdefault* as a function of $x$, $z$ and $d$, where $d$ is of type $z$. The value is $x$ if $x \in z$ and $\mathscr{R}d$ otherwise. In any case, this is an element of $z$.

```
Definition RBdefault x z (d:z) :=
   Yo (inc x z) x (Ro d).

Lemma RBdefault_in :
  forall x z (d:z), inc x z -> RBdefault x d = x.
Lemma RBdefault_out : forall x z (d:z),
  ~(inc x z) -> RBdefault x d = (Ro d).
Lemma inc_RBdefault : forall x z (d:z),
   inc (RBdefault x d) z.
```

We define now *Bdefault*. If $y$ is the value of *RBdefault*$(x,z,d)$ we know $y \in z$. Applying $\mathscr{B}$ gives $t$ of type $z$. If $x \in z$ then $\mathscr{R}t = x$, otherwise $t = d$. If $x = \mathscr{R}x'$, where $x'$ is of type $z$ then $t = x'$.

```
Definition Bdefault x z (d:z) : z :=
  Bo (inc_RBdefault x d).

Lemma R_Bdefault_in : forall x z (d:z),
  inc x z -> Ro (Bdefault x d) = x.

Lemma Bdefault_out : forall x z (d:z),
  ~(inc x z) -> (Bdefault x d) = d.

Lemma Bdefault_R : forall z (y d:z),
  Bdefault (Ro y) d = y.
```

Now the definition of *Bnat*, as a particular case where $z = \mathbb{N}$, and $d = 0$. To any set $x$ we associate an integer $n$ (an object of type *nat*). If $x \in \mathbb{N}$, then $\mathscr{R}n = x$, otherwise $n = 0$.

```
Definition Bnat x := Bdefault x (z:=nat) 0.

Lemma R_Bnat : forall x, inc x nat ->
  Ro (Bnat x) = x.

Lemma Bnat_out : forall x, ~(inc x nat) ->
   Bnat x = 0.

Lemma Bnat_R : forall (i:nat), Bnat (Ro i) = i.
```

**Module FunctionSet.** This module has been withdrawn in version 3.

We say that $u$ is in $\mathrm{F}(a, f)$ if $u$ is a functional graph, its domain is $a$ and for every $x$, the value $\mathcal{V}(x,u)$ is in $f(x)$. In section 2.10 we have defined the record $\mathrm{R}(a, f)$. It is immediate that $u \subset \mathrm{R}(a, f)$; this has as consequence that the property is bounded.

```
Definition in_function_set (a : Set) (f : EE) (u : Set) :=
  fgraph u
  & domain u = a
  & (forall y, inc y a -> inc (V y u) (f y)).

Lemma in_fs_sub_record :  forall a f u,
 in_function_set a f u -> sub u (record a f).

Lemma in_fs_eq_L :  forall a f u,
 in_function_set a f u -> u = L a (fun y : Set => V y u).

Lemma in_fs_for_L :  forall a g v,
 (forall y, inc y a -> inc (v y) (f y)) ->
 in_function_set a f (L a v).

Lemma in_fs_bounded : forall a f,
 Bounded.axioms (in_function_set a f).
```

Since the property is bounded, there exists a set, denoted $\mathrm{F}(a, f)$ above. It is a subset of $\mathfrak{P}(\mathrm{R}(a, f))$.

```
Definition function_set (a : Set) (f : EE) :=
  Bounded.create (in_function_set a f).

Lemma function_set_iff :  forall  a f u
 inc u (function_set a f) <-> in_function_set a f u.

Lemma function_set_sub_powerset_record :  forall a f,
 sub (function_set a f) (powerset (record a f)).

Lemma function_set_pr : forall a f u,
    inc u (function_set a f) ->
    (in_function_set a f u & fgraph u & domain u = a
      & (forall y, inc y a -> inc (V y u) (f y))).

Lemma function_set_inc :  forall a f u,
    fgraph u -> domain u = a ->
    (forall y, inc y a -> inc (V y u) (f y)) -> inc u (function_set a f).

Lemma in_function_set_inc :  forall a f u,
 in_function_set a f u -> inc u (function_set a f).
```

**Module Notation.** Note. This module has been withdrawn in Version 3. The content has been moved to the file algebra3.

Bourbaki says: a correspondence $f = (\mathrm{F}, \mathrm{A}, \mathrm{B})$ is said to be function if its graph F is a functional graph and if its source A is equal to its domain $\mathrm{pr}_1\mathrm{F}$ [2, p. 81], and has statements of the form: let $f$ be a mapping of A into B, if $f$ is injective and if $\mathrm{A} \neq \emptyset$ then $f$ has a left-inverse. The important point is that a function contains three items (source, graph, etc.), and has some properties as $\mathrm{A} = \mathrm{pr}_1\mathrm{F}$. In Coq this means that *source f = P (graph f)* is true for every function $f$. We do not require *graph f = P f*. In a first implementation, the graph was defined by the function *graphC* below. Then we changed our mind and associated a record to it; the trouble is then that we cannot consider the set of all functions, but must use instead the set of all triples $(\mathrm{F}, \mathrm{A}, \mathrm{B})$ associated to the function.

Consider now the following puzzle. We say that a group is a tuple $(\mathrm{E}, +, -, 0)$ with some properties and that a ring is a tuple $(\mathrm{E}, +, -, 0, *, 1)$ and that a field is a tuple $(\mathrm{E}, +, -, 0, *, 1, /)$. How can we arrange the data structure so that properties true for a group become true for a field?

One solution is the following: we say that a group is tuple of equalities $(u = \mathrm{E}, p = +, m = -, z = 0)$, where $u$, $p$, $m$, $z$ are some constant names (we will use character strings in what follows). The order becomes irrelevant, and we may have additional, useless, elements. Instead of $u = \mathrm{E}$ we use the pair $(u, \mathrm{E})$. In other words, a group is a finite functional graph.

A *notation* is a functional graph, whose domain is *string*. A finite functional graph is defined by the two constructors *stop* and *denote*. Here *stop* is the constant function that associates the empty set to every value and *denote a b f* is the function similar to $f$, but it associates $b$ to $a$.

```
Definition is_notation f :=
  fgraph f & domain f = string.

Definition stop := L string (fun s => emptyset).

Definition denote str obj old :=
  L string (fun s => (Yo (s = str) obj (V s old))).
```

The following four lemmas explain how to use the notation mechanism.

```
Lemma is_notation_stop : is_notation stop.
Lemma is_notation_denote : forall str obj old,
  is_notation old -> is_notation (denote str obj old).

Lemma V_stop : forall x, V x stop = emptyset.
Lemma V_denote_new : forall str obj old x,
  x = str -> inc x string -> V x (denote str obj old) = obj.

Lemma V_denote_old : forall str obj old x,
  ~x=str -> inc x string -> V x (denote str obj old) = V x old.
```

We define here some commonly used fields.

```
Definition Underlying := Ro "Underlying".
Definition Source  := Ro "Source".
Definition Target  := Ro "Target".
Definition Graph  := Ro "Graph".
Definition Arrow  := Ro "Arrow".

Definition Ul (x : E) := V Underlying x.
Definition sourceC (x : E) := V Source x.
Definition graphC (x : E) := V Graph x.
Definition targetC (x : E) := V Target x.
Definition arrowC x := V Arrow x.
```

This may be used later one when defining unary and binary operations on a set.

```
Definition unary (x:Set) (f:EE) := L x f.

Lemma V_unary : forall x f a, inc a x ->
  V a (unary x f) = f a.

Definition binary (x:Set) (f:Set -> Set -> Set) :=
  L x (fun a => (L x (fun b => f b a))).

Lemma V_V_binary : forall x f a b,
  inc a x -> inc b x -> V a (V b (binary x f)) = f a b.
```

**Module Universe.** This module has been withdrawn in the second edition.

A *universe* is some big set. The definition here says that if $u$ is a universe then it must contain many sets.

```
Definition axioms u :=
  (forall x y, inc x u -> inc y x -> inc y u) &
  (forall x (f:x->Set), inc x u -> (sub (IM f) u) -> inc (IM f) u) &
  (forall x, inc x u -> inc (union x) u) &
  (forall x, inc x u -> inc (powerset x) u) &
  inc nat u &
  inc string u.
```

We give here a list of properties of a universe.

```
Lemma inc_trans_u : forall x u, axioms u ->
  (exists y, (inc x y & inc y u)) -> inc x u.

Lemma inc_powerset_u : forall x u, axioms u ->  inc x u ->
   inc (powerset x) u.

Lemma inc_nat_u : forall u, axioms u -> inc nat u.
Lemma inc_R_nat_u : forall (n:nat) u, axioms u -> inc (Ro n) u.
Lemma inc_prop_u : forall u, axioms u -> inc Prop u.
Lemma inc_R_a_prop_u : forall u (p:Prop), axioms u ->  inc (Ro p) u.
Lemma inc_a_prop_u : forall u (p:Prop), axioms u -> inc p u.
Lemma inc_proof_u : forall u (p:Prop) (t:p), axioms u -> inc (Ro t) u.
Lemma inc_string_u : forall u, axioms u -> inc string u.
Lemma inc_subset_u : forall x u, axioms u ->
  (exists y, (inc y u & sub x y)) -> inc x u.
Lemma inc_emptyset_u : forall u, axioms u -> inc emptyset u.
Lemma inc_IM_u : forall x (f:x->Set) u,
  axioms u -> inc x u -> sub (IM f) u ->
  inc (IM f) u.

Definition doubleton_step :forall (x y:Set) (n:nat), Set.
intros. induction n. exact x. exact y.
Defined.

Lemma IM_doubleton_step: forall x y,
IM (doubleton_step x y) = (doubleton x y).

Lemma inc_doubleton_u : forall x y u,
  axioms u -> inc x u -> inc y u -> inc (doubleton x y) u.
Lemma inc_singleton_u : forall x u,
Lemma inc_pair_u : forall x y u,
  axioms u -> inc x u -> inc y u -> inc (pair x y) u.
Lemma sub_u : forall x u, axioms u -> inc x u -> sub x u.
Lemma inc_function_create_u : forall x f u,
   axioms u -> inc x u ->
  (forall y, inc y x -> inc (f y) u) ->
  inc (L x f) u.
Lemma inc_function_tcreate_u : forall x (f:x->Set) u,
  axioms u -> inc x u ->
  (forall y, inc (f y) u) ->
  inc (tcreate f) u.
Lemma inc_pr1_of_pair_u : forall u x,
  axioms u -> (exists y, inc (pair x y) u) -> inc x u.
Lemma inc_pr2_of_pair_u : forall u y,
  axioms u -> (exists x, inc (pair x y) u) -> inc y u.
Lemma inc_V_u : forall f x u,
  axioms u -> inc f u -> inc x u -> inc (V x f) u.
Lemma inc_denote_u : forall s x a u,
  axioms u -> inc a u -> inc s string -> inc x u ->
Lemma inc_binary_u : forall x f u,
  axioms u -> inc x u ->
  (forall y z, inc y x -> inc z x -> inc (f y z) u) ->
  inc (binary x f) u.
Lemma inc_stop_u : forall u,
   axioms u -> inc stop u.
```

**Functional graphs** The diagonal of a set E is the set of all $(x, x)$ for $x \in E$. This is the graph of the identity function on E, and the following properties are redundant.

```
Lemma diagonal_fgraph:  forall x, fgraph (diagonal x).
Lemma diagonal_graph:  forall x, is_graph (diagonal x).
Lemma diagonal_domain: forall x, domain (diagonal x) = x.
Lemma diagonal_range: forall x, range (diagonal x) = x.
Lemma diagonal_V: forall a x, inc a x -> V a (diagonal x) = a.
```

These definitions and lemmas have been withdrawn.

```
Definition is_restriction (f g :Set) :=
  fgraph g &  exists x, f = restr g x.

Lemma is_restriction_pr: forall f g,
  is_restriction f g =  (fgraph f & fgraph g & sub f g).

Definition restriction_graph f r := restr r f.
Lemma restriction_graph_pr: forall x r y,
  (inc y (restriction_graph x r) = (inc y r & inc (P y) x)).
Lemma restricted_graph_is_graph: forall x r,
  is_graph r -> is_graph (restriction_graph x r).
Lemma restriction_graph_is_graph: forall f x,
  is_function f-> fgraph (restr (graph f) x).
```

This is the original statement. Assumption *(sub (source f) (source g)* is redondant.

```
Lemma sub_function: forall f g,
  is_function f -> is_function g ->
  (sub (graph f) (graph g)) = ((sub (source f) (source g))
    & (agrees_on (source f) f g)).
```

**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)(graphCiN a).
Definition correspondence m:=
   like m & is_graph (graph m) & sub (domain (graph m)) (source m)
   & sub (range (graph m)) (target m).
```

After that we decided that a correspondence was no more a set. We had to extent the axiom of choice to functions. Thus we define *choosef*, a variant of $\tau_f(Q)$ that produces the identity function or the empty set if no function $f$ satisfies Q.

```
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).
```

The *sof_value* function creates a correspondence from a source, a target and a triple whose first element is the graph. In the current version, we can use $z$ directlty.

```
Definition sof_value x y z := corresp x y (P z).
Lemma sof_value_pra:  forall x y z,
  let f:= sof_value x y z in
  inc z (set_of_correspondences x y) ->
    (is_correspondence f & source f = x &  target f = y &  f = z).
```

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 $(x,y) \in \mathrm{F}$ 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 $\mathrm{F} \subset \mathrm{E} \times \mathrm{E}$, and $(\mathrm{E},\mathrm{E},\mathrm{F})$ is a correspondence.[1]

```
  Definition equivalence_cor r:=
    source r = target r &
    is_equivalence (graph r) & source r = (substrate (graph r)).
  Definition graph_to_eq_cor g := corresp (domain g)(domain g) g.
```

**Intersection.** The intersection of a function $f$ of type $\mathrm{I} \to \mathscr{E}$ denoted by *intersectiont f*, is the set of all $y \in \mathrm{E}$ such that $y \in f(z)$ for all $z : \mathrm{I}$. The intersection depends on the set E. In the case of intersection of sets, we use for E the representative of the set, this is not possible here. One solution is provided here: either I is empty or not; in the first case, we define the intersection as being empty, otherwise, we use the axiom of choice, select $i : \mathrm{I}$, and use $\mathrm{E} = f(i)$. In Version 4, we changed this, and use for E the union of the family.

```
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).
```

The following is no more used, since direct proofs are shorted.

A corollary is when L has two elements; in the original version, we used the property that a pair is a set with two distinct elements; here we use the canonical doubleton. Consider two families $\mathrm{F} = (\mathrm{F}_\iota)_{\iota\in\mathrm{I}}$ and $\mathrm{G} = (\mathrm{G}_\kappa)_{\kappa\in\mathrm{K}}$, and the function L that maps the canonical doubleton to $\{\mathrm{F},\mathrm{G}\}$. Then $\mathrm{L}(\iota)$ is a functional graph with non-empty domain, provided the same holds for F and G. The distributivity formulas use the set $\mathrm{I} = \prod \mathrm{J}_\lambda$. This set is the product of the domains of the graphs $\mathrm{L}(\iota)$, namely I and J. We show that it is the product of the family with index set $(\mathrm{F},\mathrm{G})$, that maps the first element to I and the second to J.

```
Lemma fgraph_rec_variantLc: forall f g,
  fgraph f -> fgraph g ->
  (forall l, inc l (domain (variantLc f g)) ->
    fgraph (V l (variantLc f g))).
Lemma nonempty_rec_dom_variantLc: forall f g,
```

[1] This notion has been withdrawn in Version 3

```
  nonempty (domain f) -> nonempty (domain g) ->
  forall l, inc l (domain (variantLc f g)) ->
    nonempty (domain (V l (variantLc f g))).
Lemma product_variantLc: forall f g,
  productf (domain (variantLc f g)) (fun l => domain (V l (variantLc f g))) =
  product2 (domain f) (domain g).
```

**Other lemmas.** Most of these theorems assert that (after simplification) we have $x = x$.

```
Lemma Z_pr : forall x p y, inc y (Zo x p) -> p y.
Definition elt x y := inc y x.
Lemma elt_inc : forall x y, elt x y = inc y x.
Lemma union2_idempotent: forall u,  union2 u u= u.


Lemma source_compose: forall r' r,  source(compose r' r) = source r.
Lemma target_compose: forall r' r,  target(compose r' r) = target r'.
Lemma graph_compose: forall r' r ,
  graph(compose r' r) = compose_graph(graph r')(graph r).
Lemma target_identity: forall x, target (identity_fun x) = x.
Lemma source_identity: forall x, source (identity_fun x) = x.
Lemma graph_identity: forall x, graph (identity_fun x) = diagonal x.
Lemma source_empty_function: source empty_function = emptyset.
Lemma target_empty_function: target empty_function = emptyset.
Lemma source_restriction: forall f x,
  source (restriction_function f x) = x.
Lemma target_restriction:forall f x,
  target (restriction_function f x) = target f.
Lemma graph_restriction:forall f x,
  graph (restriction_function f x) =  (restr  (graph f) x).
Lemma source_restriction2:forall f x y,  source (restriction2 f x y) = x.
Lemma target_restriction2:forall f x y,  target (restriction2 f x y) = y.
Lemma af_source: forall f a b,  source (BL f a b) = a.
Lemma af_target: forall f a b,  target (BL f a b) = b.
Lemma source_first_proj: forall g, source (first_proj g) =  g.
Lemma source_second_proj: forall g, source (second_proj g) = g.
Lemma target_first_proj: forall g, target (first_proj g) = domain g.
Lemma target_second_proj: forall g, target (second_proj g) = range g.
Lemma source_ci: forall a b , source (canonical_injection a b)=a.
Lemma target_ci: forall a b, target (canonical_injection a b)=b.
Lemma graph_ci:  forall a b, graph (canonical_injection a b)= diagonal a.
Lemma source_diag_app:forall a, source (diagonal_application a) = a.
Lemma target_diag_app:forall a, target (diagonal_application a) = product a a.
Lemma source_inv_graph_canon: forall g, source (inv_graph_canon g) = g.
Lemma target_inv_graph_canon: forall g,
  target (inv_graph_canon g) = (inverse_graph g).
Lemma source_fpf :forall f y,
  source (first_partial_fun f y) = domain (source f).
Lemma source_spf :forall f y,
  source (second_partial_fun f y) = range (source f).
Lemma source_fpfa :forall f,
  source (first_partial_function f) = range (source f).
Lemma source_spfa :forall f,
  source (second_partial_function f) = domain (source f).
Lemma source_fpfb :forall a b c,
  source (first_partial_map a b c) = (set_of_functions (product a b) c).
```

```
Lemma source_spfb :forall a b c,
  source (second_partial_map a b c) = (set_of_functions (product a b) c).
Lemma target_fpf :forall f y,
  target(first_partial_fun f y) = target f.
Lemma target_spf :forall f y,
  target(second_partial_fun f y) = target f.
Lemma target_fpfa :forall f,  target(first_partial_function f) =
  set_of_functions (domain (source f)) (target f).
Lemma target_spfa :forall f,  target(second_partial_function f) =
  set_of_functions (range (source f)) (target f).
Lemma target_fpfb :forall a b c,  target(first_partial_map a b c) =
  set_of_functions b (set_of_functions a c).
Lemma target_spfb :forall a b c,  target(second_partial_map a b c) =
  set_of_functions a (set_of_functions b c).
Lemma source_pri: forall f i, source(pr_i f i) = productb f.
Lemma target_pri: forall f i, target(pr_i f i) = V i f.
Lemma source_prit: forall In (f:In->Set) i,  source(pr_it f i) = productt f.
Lemma target_prit: forall In (f:In->Set) i,  target(pr_it f (Ro i)) = f i.
Lemma source_product1_canon: forall x a,
  source (product1_canon x a) = x.
Lemma target_product1_canon: forall x a,
  target (product1_canon x a) = (product1 x a).
Lemma source_product2_canon: forall x y,
  source (product2_canon x y) = (product x y).
Lemma target_product2_canon: forall x y,
  target (product2_canon x y) = (product2 x y).
Lemma source_pc: forall f u, source (product_compose f u) = (productb f).
Lemma target_pc: forall f u,
  target (product_compose f u) =
  (productf (source u) (fun k => V (W k u) f)).
Lemma source_prj: forall f j,
  source (pr_j f j ) = (productb f).
Lemma target_prj: forall f j,
  target (pr_j f j ) = restriction_product f j.
Lemma source_pam: forall f g,
  source (prod_assoc_map f g) = productb f.
Lemma target_pam: forall f g,
  target (prod_assoc_map f g) =
  (productf (domain g) (fun l => (restriction_product f (V l g)))).
Lemma source_popc:forall f f',
  source (prod_of_products_canon f f') =
  (product (productb f) (productb f')).
Lemma source_ext_map_prod: forall  In src trg f,
  source (ext_map_prod In src trg f) = (productf In src ).
Lemma target_ext_map_prod: forall  In src trg f,
  target (ext_map_prod In src trg f) = (productf In trg).
Lemma source_ext_to_prod: forall  u v,
  source (ext_to_prod u v) = product (source u) (source v).
Lemma target_ext_to_prod: forall u v,
  target (ext_to_prod u v) = product (target u) (target v).

Definition canon_projc f := BL(fun x=> gclass f x)
  (source f) (quotient (graph f)).
Lemma source_canon_proj: forall r,
  source (canon_proj r) = substrate r.
Lemma target_canon_proj: forall r,
  target (canon_proj r) = quotient r.
Lemma source_canon_projc: forall f,
  source (canon_projc f) = source f.
Lemma target_canon_projc: forall f,
  target (canon_projc f) = quotient (graph f).
Lemma source_section_canon_proj: forall r,
  source (section_canon_proj r) = (quotient r).
Lemma target_section_canon_proj: forall r,
  target (section_canon_proj r) = (substrate r).
Lemma source_foq: forall r f b,
  source (function_on_quotient r f b)=  quotient r.
Lemma source_foqs: forall r r' f,
  source (function_on_quotients r r' f)=  quotient r.
Lemma source_foqc: forall r f,
  source (fun_on_quotient r f)=  quotient r.
Lemma source_foqcs: forall r r' f,
  source (fun_on_quotients r r' f)=  quotient r.
Lemma target_foq: forall r f b,  target (function_on_quotient r f b) =  b.
Lemma target_foqs: forall r r' f ,
   target (function_on_quotients r r' f)=  quotient r'.
Lemma target_foqc: forall r f,  target (fun_on_quotient r f)=  target f.
Lemma target_foqcs: forall r r' f,
  target(fun_on_quotients r r' f)= quotient r'.

Lemma composable_identity_left: forall m,
  is_correspondence m -> composableC (identity_fun (target m)) m.
Lemma composable_identity_right: forall m,
  is_correspondence m -> composableC m  (identity_fun (source m)).
Lemma correspondence_of_restricted_eq: forall x,
  graph_to_eq_cor(diagonal x) = identity_fun x.
Theorem equivalence_cor_pr: forall f,
  is_correspondence f ->
  (equivalence_cor f = (source f = target f &
    (source f = (domain (graph f))) & compose f f = f &
    f = inverse_fun f)).
Lemma restriction_correspondence: forall f x,
  is_function f-> sub x (source f)
  -> is_correspondence(restriction_function f x).
Lemma w_identity: forall a x,  identityC a x = x.
Lemma inc_create_domain: forall sf f a,
  inc a (domain (L sf f))  = inc a sf.
Lemma restriction_V: forall f x i,
  fgraph f -> sub x (domain f) -> inc i x ->  V i (restr f x) = V i f.
```

## 9.4 Tactics

The following bunch of tactics was introduced by C. Simpson. They have in general two letters as in *rw*; they have the form *rwi* when they apply to a given hypothesis.

```
Ltac ir := intros.
Ltac rw u := rewrite u.
Ltac rwi u h := rewrite u in h.
Ltac wr u := rewrite <- u.
Ltac wri u h := rewrite <- u in h.
Ltac ap h := apply h.
Ltac om := omega.
Ltac am := assumption.
Ltac tv := trivial.
Ltac eau := eauto.
Ltac sy := symmetry.
Ltac uf u := unfold u.
Ltac ufi u h := unfold u in h.
Ltac nin h := induction h.
Ltac uh a := red in a.
Ltac au := first [ solve [am] | auto ].
```

Tactics where the last letter is doubled are extensions that call *tv* in order to solve a goal.

```
Ltac app u := ap u; tv.
Ltac apw u v := apply u with v ; tv.
Ltac rww u := rw u; tv.
Ltac rwii u h:= rwi u h; tv.
Ltac wrr u := wr u; tv.
```

In the current version of the software we do not use these tactics anymore. Instead of "*rww rel*" we use "*rewrite rel//*" and all facilities provided by the *ssreflect* package. In the remainder of this paragraph, we shall describe some of the old tactics, and compare them with the new variants (described in the next paragraph).

We define here a tactic *ee* that removes all conjunctions. This may introduce some assumptions named H1, H2, etc. In the current version, amm assumption names are explicit.

```
Ltac EasyDeconj :=
  match goal with
  |  |- (_ & _) => ap conj; [ EasyDeconj | EasyDeconj ]
  |  |- _ => idtac
  end.
Ltac EasyExpand :=
  match goal with
  | id1:(?X1 /\ ?X2) |- _ => nin id1; EasyExpand
  |  |- _ => EasyDeconj
  end.
Ltac ee := EasyExpand.
```

This tactic helps solving $a = b$ by application of the axiom of extent for sets.

```
Ltac set_extens:= app extensionality; unfold sub; intros.
```

The *cp* tactic is a variant of *set* or *pose*. The idea is that, if we copy *b*, this will give something like "*a:=b:c*", that will be converted to "*a:c*". This has been replaced by use of the *move* tactic.

```
Ltac Remind u :=
  set (recalx := u);
   match goal with
   | recalx:?X1 |- _ => assert X1; [ exact recalx | clear recalx ]
   end.
Ltac cp := Remind.
```

The *Ztac* tactic can be used when the assumption or conclusion contains $x \in \{y \in z \,|\, \mathrm{P}(y)\}$; it replaces by: $x \in z$ and $\mathrm{P}(x)$. This has been removed since it was fragile.

```
Ltac Ztac :=
  match goal with
    | id1:(inc _ (Zo _ _)) |- inc _ (Zo _ _ )
      => nin (Z_all id1) ; ap Z_inc; auto
    | id1:(inc _ (Zo _ _)) |- _ => nin (Z_all id1)
    |  |- (inc _ (Zo _ _)) => ap Z_inc; auto
    | _ => idtac
  end.
```

The next tactics have been introduced in Version 3. We were able to reduce by 17% the size of the file *sete2.v* described in the previous chapter.

The tactic *ue* rewrites equation $a = b$ in some cases. The tactic *eee* is more powerful; it first handles conjunctions, then tries to use *ue*; and finally, calls *fprops*, i.e., *auto* with some data base. The *ue* tactic has been simplified: we keep only the rule containing the *solve* tactic.

```
Ltac Use_eq :=
  match goal with
    | H:?a = ?b |- ?f ?a = ?f ?b => rww H
    | H:?a = ?b |- ?f ?a _ = ?f ?b _ => rww H
    | H:?a = ?b |- ?f _ ?a = ?f _ ?b  => rww H
    | H:?a = ?b |- ?f ?a _ _  = ?f ?b _ _  => rww H
    | H:?a = ?b |- ?f _ ?a _   = ?f _  ?b _  => rww H
    | H:?a = ?b |- ?f _ _ ?a  = ?f _ _ ?b  => rww H
    | H:?a = ?b |- _ => solve [ rww  H ; fprops | wrr H ; fprops]
    | Ha : ?a = ?c, Hb : ?b =  ?c |- ?a = ?b => wr Ha ; wr Hb; tv
    | H:?b = _ |- _ ?b => rww H
    | H:?b = _ |- _ _ ?b  =>  rww H
    | H:?b = _ |- _ ?b _ =>  rww H
    | H:?b = _ |- _ _ ?b _ =>  rww H
    | H:?b = _ |- _ ?b _ _ =>  rww H
    | H:?b = _ |- _ _ _ ?b  =>  rww H
end.
Ltac ue := Use_eq.
Ltac eee := ee ; tv; try ue; fprops.
```

The next tactic considers $x \in \{a\}$ or $y \in \{a, b\}$ and replaces $x$ or $y$ by $a$ or $b$ in the goal. It has been removed.

```
Ltac db_tac :=
 match goal with
    | H: inc _ (doubleton _ _ ) |- _ => nin (doubleton_or H); ue
    | H: inc _ (singleton _  ) |- _ => rwi singleton_rw H; ue
end.
```

The *inter2tac* tactic helps solvings goals of the form $x \in \mathrm{A} \cup \mathrm{B}$ or consequences of $x \in \mathrm{A} \cap \mathrm{B}$. It has been removed.

```
Ltac inter2tac :=
  match goal with
    | H:inc ?X1 (intersection2 ?X2 _ ) |- inc ?X1 ?X2
      => ap (intersection2_first H)
    | H:inc ?X1 (intersection2 _ ?X2 ) |- inc ?X1 ?X2
      => ap (intersection2_second H)
    | H:inc ?X1 ?X2 |- inc ?X1 (union2 ?X2 _)
```

```
      => app union2_first
    | H:inc ?X1 ?X2 |- inc ?X1 (union2 _ ?X2)
      => app union2_second
    |   |- inc _ (intersection2 _ _ )
      => app intersection2_inc
    | H:J _ _ = J _ _ |- _
       => solve [ rww (pr1_injective H);fprops | rww (pr2_injective H) ; fprops]
  end.
```

This helps solving equality of functions. Removed.

```
Ltac corr_tac :=
  match goal with
    | H: corr_value _ = corr_value _ |- _
      => rwi correspondence_extensionality1 H
    | |-  corr_value _ = corr_value _
      => rw correspondence_extensionality1
    | Ha: corr_value ?a = ?b, Hb:corr_value ?c = ?d |- ?b = ?d
      => wr Ha; wr Hb ;rww correspondence_extensionality1
end.
```

The first tactic replaces a goal $x = \emptyset$ by a goal *False* under the assumption $y \in x$. The second tactic takes an argument $u$ and proves any goal by asserting $u \in \emptyset$. If an assumption is $\mathrm{A} = \emptyset$, the goal will be $u \in \mathrm{A}$. These tactics have been modified.

```
Ltac empty_tac :=
  match goal with
  |  |-  _ = emptyset => ap is_emptyset; red;ir; idtac
  | _ => idtac
  end.

Ltac empty_tac1 u :=  elim (emptyset_pr (x:= u));
  match goal with H: ?x = emptyset |- _ =>  wr H
    | H: emptyset = ?x |- _ =>  rw H end ; fprops.
```

## 9.5 Tactics V4

These tactics use one of the five data bases (via *autorewrite* or *auto*).

```
Ltac aw := autorewrite with aw;  trivial.
Ltac awi u:= autorewrite with aw in u.
Ltac bw := autorewrite with bw; trivial.
Ltac srw:= autorewrite with sw; trivial.
Ltac fprops := auto with fprops.
Ltac gprops := auto with gprops.
```

These two tactics can be use to solve $a = b$ via the axiom of extent; all we have to show is that for all $v$, the conditions $v \in a$ implies $v \in b$ and conversely. The first tactic uses an argument in which to put the assumption $v \in a$ or $v \in b$.

```
Ltac set_extens v Hv:= apply extensionality=> v Hv.
Ltac set_extens1 v:= apply extensionality=> v.
```

The *dneg* tactic implements the rule: if A $\implies$ B then ¬B $\implies$ ¬A. The *ex_middle* tactic solves $a = b$ or $a \in b$ by contradiction. These tactics take an argument (the name of the assumption).

```
Ltac dneg u := match goal with
  H : ~ _ |- ~ _ => move => u; move :H; apply
end.

Ltac ex_middle u := match goal with
  | |- inc ?a ?b => case (inc_or_not a b) => // u
  | |- ?a = ?b => case (equal_or_not a b)=> // u
end.
```

The *ue* tatics rewrites an equality if it solves the goal. The *ee* tactic recursively splits a conjunction and applies *intuition* if this solves the goal.

```
Ltac ue :=
  match goal with
    | H:?a = ?b |- _ => solve [ rewrite  H ; fprops | rewrite - H ; fprops]
end.
Ltac EasyDeconj :=
  match goal with
  |  |- (_ & _) => apply conj; [ EasyDeconj | EasyDeconj ]
  |  |- _ => try solve [intuition]
  end.
Ltac ee := EasyDeconj.
```

This tactics solves goals involving existence and pairs.

```
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 |- exists x, _ & inc (J x ?y) ?z
     => exists x ; split; trivial
   | H:inc (J ?x ?y) ?z |- exists y, _ & inc (J ?x y) ?z
     => exists y ; split; trivial
   | H:inc (J ?x ?y) ?z |- exists x, inc (J x ?y) ?z & _
     => exists x ; split; trivial
   | H:inc (J ?x ?y) ?z |- exists y, inc (J ?x y) ?z & _
     => exists y ; split; trivial
   | H:inc ?x ?y |- exists x, inc x ?y & _
     => exists x ; split; fprops
   | H:inc ?x ?y |- exists x, _ & inc x ?y
     => exists x ; split; fprops
    |  |- exists x, inc x (singleton ?y) & _
     => exists y ; split; fprops
    | H1 : is_graph ?g, H : inc (J ?x ?y) ?g |-  inc ?x (domain ?g)
       => rewrite (domain_rw _ H1) ; exists y; exact  H
    |  H1 : is_graph ?g,  H : inc (J ?x ?y) ?g |-  inc ?y (range ?g)
       => rewrite (range_rw _ H1); exists x; exact 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 graph_tac :=
  match goal with
    | |-  inc (J ?x (W ?x ?f)) (graph ?f) => apply W_pr3 ; fprops
    |  h:inc (J ?x ?y) (graph ?f) |- W ?x ?f = ?y => apply W_pr ; fprops
    |  h:inc (J ?x ?y) (graph ?f) |- ?y  = W ?x ?f
      => symmetry; apply W_pr ; fprops
    | |- inc (W _ ?f) (range (graph ?f)) => apply inc_W_range_graph ; fprops
    | |- inc (W _ ?f) (target ?f)  => apply inc_W_target ; fprops

    | Ha:is_function ?X1, Hb: inc (J _ ?X2) (graph ?X1)
      |-  inc ?X2 (target ?X1)
        => apply (inc_pr2graph_target Ha Hb)
    | Ha:is_function ?X1, Hb: inc (J ?X2 _) (graph ?X1)
      |-  inc ?X2 (source ?X1)
        => apply (inc_pr1graph_source Ha Hb)
    | Ha:is_function ?X1, Hb: inc ?X2 (graph ?X1)
      |-  inc (P ?X2) (source ?X1)
        => apply (inc_pr1graph_source1 Ha Hb)
    | Ha:is_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:bijective ?X1 |- is_function ?X1 => exact (bij_is_function H)
    | H:injective ?X1 |- is_function ?X1 => exact (inj_is_function H)
    | H:surjective ?X1 |- is_function ?X1 => exact (surj_is_function H)
    | H:is_function ?X1 |- is_correspondence ?X1 =>
      by case H
    | H:is_function ?g |- sub (range (graph ?g)) (target ?g)
      => apply (f_range_graph H)
    | H:composable ?X1 _ |- is_function ?X1 => destruct H as [H _ ]; exact H
    | H:composable _ ?X1 |- is_function ?X1 => destruct H as [_ [H _ ]]; exact H
    | H:composable ?f ?g |- is_function (compose ?f ?g ) =>
     apply (compose_function H)
    | H:is_function ?f |- is_function (compose ?f ?g ) =>
     apply compose_function; apply conj=>//; apply conj
    | H:is_function ?g |- is_function (compose ?f ?g ) =>
     apply compose_function; apply conj; last apply conj=>//
    | Ha:is_function ?f, Hb:is_function ?g |- ?f = ?g =>
      apply function_exten
  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_substrate H)
    | H:inc ?x ?r |- inc (Q ?x) (substrate ?r)
      => apply (inc_pr2_substrate H)
    | H:related ?r ?x _ |- inc ?x (substrate ?r)
      => apply (inc_arg1_substrate H)
    | H:related ?r _ ?y |- inc ?y (substrate ?r)
      => apply (inc_arg2_substrate H)
    | H:inc(J ?x _ ) ?r|- inc ?x (substrate ?r)
      => apply (inc_arg1_substrate H)
    | H: inc (J _ ?y) ?r |- inc ?y (substrate ?r)
      => apply (inc_arg2_substrate H)
  end.
```

This tatics exploits the properties of an equivalence relation.

```
Ltac equiv_tac:=
  match goal with
    | H: is_equivalence ?r, H1: inc ?u (substrate ?r) |- related ?r ?u ?u
      => apply (reflexivity_e H H1)
    | H: is_equivalence ?r |- inc (J ?u ?u) ?r
      => apply reflexivity_e
    | H:is_equivalence ?r, H1:related ?r ?u ?v |- related ?r ?v ?u
      => apply (symmetricity_e H H1)
    | H:is_equivalence ?r, H1: inc (J ?u ?v) ?r |- inc (J ?v ?u) ?r
      => apply (symmetricity_e H H1)
    | H:is_equivalence ?r, H1:related ?r ?u ?v, H2: related ?r ?v ?w
      |- related ?r ?u ?w
      => apply (transitivity_e H H1 H2)
    | H:is_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: is_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.
```

This tactic solves goals of the form $x \in \bigcup_{i \in I} X_i$, by guessing the value of $i$.

```
Ltac union_tac:=
  match goal with
    | H:inc ?x (?f ?y) |- inc ?x (uniont ?f)
      => apply (uniont_inc H)
    | Ha : inc ?i (domain ?g), Hb :  inc ?x (V ?i ?g) |- inc ?x (unionb ?g)
      => apply (unionb_inc Ha Hb)
    | Ha : inc ?x ?y, Hb :  inc ?y ?a |- inc ?x (union ?a)
      => apply (union_inc Ha Hb)
    | Ha : inc ?y ?i, Hb :  inc ?x (?f ?y) |- inc ?x (unionf ?i ?f)
      => apply (unionf_inc _ Ha Hb)
    | Ha : inc ?y ?i |- inc ?x (unionf ?i ?f)
      => apply unionf_inc with y; fprops
    | Ha : inc ?i (domain ?g) |- inc ?x (unionb ?g)
      => apply unionb_inc with i; fprops
    | Ha : inc ?y ?i |- inc ?x (unionf ?i ?f)
      => apply unionf_inc with y; fprops
    | Ha : inc ?x (V ?i ?g) |- inc ?x (unionb ?g)
      => apply unionb_inc with i; fprops
    | Hb : inc ?x (?f ?y) |- inc ?x (unionf ?i ?f)
      => apply unionf_inc with y; fprops
  end.
```

New versions of *empty_tac*.

```
Ltac empty_tac0 :=
  match goal with
  |  H:inc _ emptyset |- _ => elim (emptyset_pr H)
  end.

Ltac empty_tac  v H:=
  match goal with
  |  |-  _ = emptyset => apply is_emptyset; move=> v H
  end.

Ltac empty_tac1 u :=  elim (emptyset_pr (x:= u));
  match goal with H: ?x = emptyset |- _ => rewrite <- H
    | H: emptyset = ?x |- _ =>  rewrite H end ; fprops.
```

Other tactics.

```
Ltac try_lvariant u:=
  rewrite two_points_pr in u; move:u=> [] ->; bw.
Ltac eq_dichot v:= match goal with |- ?a = ?b \/ _
  => case (equal_or_not a b); first (by intuition); move=> v; right end.
Ltac eqtrans u:= apply equipotent_transitive with u; fprops.
Ltac eqsym:= apply equipotent_symmetric.
```

## 9.6 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*) « $x \subset x$ », [20].

Proposition 2 (*sub_trans*) « $(x \subset y$ and $y \subset z) \implies (x \subset z)$ », [20].

Theorem 1 « The relation $(\forall x)(x \notin X)$ is functional in X. » This theorem asserts existence and uniqueness of the empty set, [21].

### Section 2

Theorem 1 asserts existence of the product $X \times Y$ of two sets, [32].

Proposition 1 (*product_monotone* and variants) « If $A'$, $B'$ are non-empty sets, the relation $A' \times B' \subset A \times B$ is equivalent to "$A' \subset A$ and $B' \subset B$" », [32].

Proposition 2 (*empty_product_pr*) « Let A and B be two sets. The relation $A \times B = \emptyset$ is equivalent to "$A = \emptyset$ or $B = \emptyset$" », [32]

## Section 3

Proposition 1 (*range_domain_exists*) asserts existence and uniqueness of the range and domain of a graph, [39].

Proposition 2 (*image_by_increasing*) « Let G be a graph and let X, Y be two sets; then the relation $X \subset Y$ implies $G\langle X\rangle \subset G\langle Y\rangle$ », [42].

Corollary (*image_of_large*).

Proposition 3 (*inverse_compose*) « Let G, G′ be two graphs. The inverse of $G' \circ G$ is then $\overset{-1}{G} \circ \overset{-1}{G'}$ », [45].

Proposition 4 (*composition_associative*) is associativity of composition of graphs, [45].

Proposition 5 (*image_composition*) says $(G' \circ G)\langle A\rangle = G'\langle G\langle A\rangle\rangle$, [45].

Proposition 6 (*is_function_compose*) 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 », [82].

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 », [62].

Proposition 8 (*inj_if_exists_left_inv*, and variants) says under which conditions a function has a left or right inverse, [65].

Corollary (*bijective_from_compose*).

Theorem 1 (*inj_compose*) and variants) studies the relationship between injectivity, surjectivity and composition, [67].

Proposition 9 (*exists_left_composable* and variants) explains when a function can be factored through another one, [69].

## Section 4

Proposition 1 (*uniont_rewrite*, *intersectiont_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, [78].

Proposition 2 (*union_assoc* and *intersection_assoc*) states associativity of union and intersection, [79].

Proposition 3 (*image_of_union* and *image_of_intersection*) says that if Γ is a correspondence, $\Gamma\langle\bigcup X_\iota\rangle = \bigcup \Gamma\langle X_\iota\rangle$ and $\Gamma\langle\bigcap X_\iota\rangle \subset \bigcap \Gamma\langle X_\iota\rangle$, [80].

Proposition 4 (*inv_image_of_intersection*) says equality holds for the inverse image of intersection, [80].

Corollary (*inj_image_of_intersection*).

Proposition 5 (*complementary_union* and *complementary_intersection*) studies the complementary of unions and intersections, [80].

Proposition 6 (*inv_image_of_comp*) studies the inverse image of the complementary, [82].

Corollary (*inj_image_of_comp*).

Proposition 7 (*agrees_on_covering* and *extension_covering*) says that if $X_\iota$ is a covering of E, then two functions that agree on each $X_\iota$ agree on E, and a function defined on each $X_\iota$ can be extended to E if the obvious compatibility conditions hold, [84].

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 $f_\iota$ , [87].

Proposition 9 (*disjoint_union_lemma*) asserts existence of the disjoint union, [87].

Proposition 10 (*disjoint_union_pr*) relates sum and union, [87].

### 9.6.1 Section 5

Proposition 1 (*surjective_etp* and *injective_etp*) says: if $f$ is surjective (resp. injective), then its extension to the set of sets is surjective (resp. injective), [89].

Proposition 2 (*injective_c3f* and *surjective_c3f*) states under which conditions $f \mapsto v \circ f \circ u$ is injective or surjective, [91].

Corollary (*bijective_c3f*).

Proposition 3 (*bijective_fpfa* and *bijective_spfa*) says that $\mathscr{F}(\mathrm{B} \times \mathrm{C};\mathrm{A})$, $\mathscr{F}(\mathrm{B};\mathscr{F}(\mathrm{C};\mathrm{A})$ and $\mathscr{F}(\mathrm{C};\mathscr{F}(\mathrm{B};\mathrm{A}))$ are canonically isomorphic, [91].

Proposition 4 (*bijective_pc*) says: Given a family $\mathrm{X}_\iota$ and a bijection $f$, the product $\prod \mathrm{X}_\iota$ is isomorphic to the product $\prod \mathrm{X}_{f(\iota)}$, [97].

Propositions 6 and 5 (*extension_exists* and *surjective_prj*) if $\mathrm{X}_\iota$ is nonempty for $\iota \not\in \mathrm{J}$, then $\mathrm{pr}_\mathrm{J}$ is surjective from the product $\prod_{\iota\in\mathrm{I}} \mathrm{X}_\iota$ into the partial product $\prod_{\iota\in\mathrm{J}} \mathrm{X}_\iota$ [99].

Corollary 1 (*surjective_pri*).

Corollary 2 (*nonempty_product* and variants).

Corollary 3 (*productb_monotone1*, *productb_monotone2*).

Proposition 7 (*bijective_pam*) states associativity of the product, [99].

Proposition 8 (*distrib_union_inter* and *distrib_inter_union*) states distributivity of union over intersection and intersection over union, [100].

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 $\Gamma$ between X and X is an equivalence on X if and only if it satisfies the following conditions: (a) X is the domain of $\Gamma$; (b) $\Gamma = \Gamma^{-1}$; (c) $\Gamma \circ \Gamma = \Gamma$ », [114].

Criterion C55 (*related_e_rw*) characterizes the canonical projection, [116].

Criterion C56 (*rel_on_quo_pr*) « Let $\mathrm{R}\{x, x'\}$ be an equivalence relation on a set E and let $\mathrm{P}\{x\}$ be a relation that does not contain the letter $x'$ and is compatible (with respect to $x$) with the equivalence relation $\{x, x'\}$. Then, if $t$ does not appear in $\mathrm{P}\{x\}$, the relation "$t \in \mathrm{E}/\mathrm{R}$ and $(\exists x)(x \in t$ and $\mathrm{P}\{x\})$" is equivalent to the relation "$t \in \mathrm{E}/\mathrm{R}$ and $(\forall x)(x \in t$ and $\mathrm{P}\{x\})$" ». [119].

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 $h = f \circ s$. »[122]

## 9.7 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, [7].

$\mathrm{R}\{x\}$ 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 $(x|y)\mathrm{R}$.

$\tau_x(\mathrm{R})$ is a Bourbaki notation, it is the generic element satisfying $\mathrm{R}\{x\}$, [12].

$x \implies y$ 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. We use it as synonym to $\iff$.

$(a|b)c$ is a Bourbaki notation, meaning the relation obtained by replacing $b$ by $a$ in $c$, [8].

$x : y$ 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*.

$(\forall x)\mathrm{P}$ and *forall x, p* are similar constructions, [12].

$(\exists x)\mathrm{P}$ and *exists x, p* are similar constructions, [12].

$(\exists! x)\mathrm{P}$ means sometimes *exists_unique*.

$x \in y$, $x \ni y$ (is element of): see *inc* and *elt*.

$x \subset y$ (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*.

$(x, y)$ (ordered pair): see *J*.

$\bigcup \mathrm{X}$, $\bigcup_{\iota \in \mathrm{I}} \mathrm{X}_\iota$, see *union*.

$a \cup b$, $a \cap b$, see *union2, intersection2*.

$\mathrm{A} \times \mathrm{B}$, $u \times v$, $\mathrm{R} \times \mathrm{R}'$, see *product, ext_to_prod, prod_of_relation*.

$f \circ g$, see *fcompose, gcompose, compose_graph, compose, composeC*.

$\Delta_\mathrm{A}$, see *diagonal*.

$\overset{-1}{\mathrm{G}}$ 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).

$\boldsymbol{x} \to \boldsymbol{T}$ ($\boldsymbol{x} \in \boldsymbol{A}, \boldsymbol{T} \in \boldsymbol{C}$), is the function with source A, target C that maps $x$ to T, [55].

$(f_x)_{x \in \mathrm{A}}$ is a shorthand for $x \to f(x)$ $(x \in \mathrm{A})$; see above, the piece $\boldsymbol{T} \in \boldsymbol{C}$ is implicit.

$\hat{f}$, see *extension_to_parts*

$\mathrm{F}^{\mathrm{E}}$, see *set_of_gfunctions.*

$\mathscr{F}(\mathrm{E};\mathrm{F})$ see *set_of_functions.*

$\Phi(\mathrm{E},\mathrm{F})$ see *set_of_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,* [91].

$\tilde{f}$, sometimes denotes the mappings $x \mapsto f_x$ or $y \mapsto f_y$. Implemented as *first_partial_function, second_partial_function,* [91].

$f \mapsto \tilde{f}$, implemented as *first_partial_map, second_partial_map,* is a bijection from $\mathscr{F}(\mathrm{B}\times\mathrm{C};\mathrm{A})$ into $\mathscr{F}(\mathrm{B};\mathscr{F}(\mathrm{C};\mathrm{A})$ or $\mathscr{F}(\mathrm{C};\mathscr{F}(\mathrm{B};\mathrm{A}))$, [91].

$\prod_{\iota \in \mathrm{I}} \mathrm{X}_\iota$ see *productt.*

$(x_\iota)_{\iota \in \mathrm{I}}$ 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.

$g_{\mathrm{E}}(\sim)$, 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 $x$.

$\mathrm{E}/\sim$, $\mathrm{E}/\mathrm{R}$, see *quotient.*

$\mathrm{R}/\mathrm{S}$ see *quotient_of_relations.*

$\mathrm{X}_f$ sometimes means $f^{-1}\langle f\langle \mathrm{X}\rangle\rangle$, see *inverse_direct_value.*

$\mathrm{R}_{\mathrm{A}}$ see *induced_relation.*

¶ is not defined. We use it as a paragraph separator.

**Letters**

$\mathscr{B}$ see *Bo.*

$\mathscr{C}_{\mathrm{C}}(a,b)$, $\mathscr{C}_{\mathrm{T}}(p,q)$, $\mathscr{C}(p)$: see *by_cases a b, chooseT* and *choose.*

$\mathrm{C}_{xy}a$ stands for *constant_function x y a,* it is the constant function from $x$ to $y$ with value $a$, [50].

$\mathrm{C}_{\mathrm{R}}x$ 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.*

$\mathscr{E}_x(\mathrm{R})$ appears in the English version where $\{x,\mathrm{R}\}$ is used in the French version; see *Zo.*

$\mathrm{I}_{\mathrm{A}}$, see *identity.*

$\mathrm{I}_{xy}$ see *inclusionC, canonical_injection.*

$\mathscr{L}_{\mathrm{X}}f$, $\mathscr{L}f$, $\mathscr{L}_{\mathrm{A};\mathrm{B}}f$ (creating functions): see *L, acreate, BL.*

$\mathscr{M}f$, $\mathscr{M}_{\mathrm{A};\mathrm{B}}f$ (inverse of $\mathscr{L}$), see *bcreate1* and *bcreate.*

$\mathfrak{P}(x)$, see *powerset.*

$\mathrm{pr}_1 z$, $\mathrm{pr}_2 z$, $\mathrm{pr}_\iota f$, $\mathrm{pr}_{\mathrm{J}} f$ (projections), see P, Q, *pr_i, pr_j.*

$\mathscr{R}x$ see *Ro.*

$\mathrm{R}_{ab}f$ (restriction) see [54].

$\mathscr{V}(x,f)$, $\mathscr{V}_f x$ (value of a function): see *V.*

$\mathscr{W}_f x$ (value of a function): see *W*.

$\mathscr{X}(f,y)$, see *Xo*.

$\mathscr{Y}(\mathrm{P},x,y)$ see *Yo*.

$\mathscr{Z}(x,\mathrm{P})$ see *Zo*.

**Words**

*acreate f*, $\mathscr{L} f$, is the correspondence associated to the Coq function $f$, [42].

*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, [52].

*bcreate f A B*, $\mathscr{M}_{\mathrm{A};\mathrm{B}} f$, is a kind of inverse of $\mathscr{L}$, [49].

*bcreate1 f*, $\mathscr{M} f$, is a kind of inverse of $\mathscr{L}$ [49].

*bijective f, bijectiveC f,* means that $f$ is a bijection, [58].

*BL f a b*, $\mathscr{L}_{\mathrm{A};\mathrm{B}} f$, *fun_function f a b*, is function from A to B whose graph is $\mathscr{L}_{\mathrm{A}} f$, [55].

*Bo*, $\mathscr{B}$, is an inverse of $\mathscr{R}$, [21].

*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, [22].

*canonical_injection x y*, $\mathrm{I}_{xy}$, is the inclusion map on $x \subset y$, , [61].

*canon_proj r*, is the mapping $x \mapsto \bar{x}$ from E onto E/R, the quotient set of $r$, [116].

*class r x* is the class of $x$ for the equivalence relation $r$, [114].

*choose p*, $\mathscr{C}(p)$, is some $x$ such that $p(x)$ is true, the empty set if no $x$ satisfies $p$, [22].

*chooseT p q*, $\mathscr{C}_{\mathrm{T}}(p,q)$, is our basic axiom of choice, [19].

*coarse x* is $x \times x$, [112].

*coarser_covering I f J g, coarser_c f g,* two definitions that say for all $j \in \mathrm{J}$ there is $i \in \mathrm{I}$ such that $g_j \subset f_i$ or for all $g_j \in g$ there is $f_i \in f$ such that $g_j \subset f_i$, [82].

*compatible_with_equiv_p p r* means that $p(x)$ and $x \overset{r}{\sim} y$ implies $p(y)$, [119].

*compatible_with_equiv f r* means that $x \overset{r}{\sim} y$ is equivalent to $f(x) = f(y)$, [122].

*compatible_with_equivs f r r'* means that $x \overset{r}{\sim} y$ is equivalent to $f(x) \overset{r'}{\sim} f(y)$, [122].

*complement a b*, $a - b$, $a \setminus b$, $\complement b$, is the set of element of $a$ not in $b$, [26].

*composableC f g, composable f g* is the condition on correspondences (resp. functions) $f$ and $g$ for $f \circ g$ to be a correspondence (resp. function), [45], [57].

*compose_graph f g*, $f \circ g$, composition of two graphs, [44].

*compose f g, composeC f g*, $f \circ g$, is the composition of two functions, [45], [52].

*constant_graph s x* is the graph of the constant function with domain $s$ and value $x$, [97].

*correspondenceC* is a data type with three slots, source, target and graph, [40]

*corr_value f* associates to a correspondence $f$ its triple $(\mathrm{G},\mathrm{A},\mathrm{B})$, [40].

*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 $x$, [82].

*cut x p* is the set of all $x$ that satisfy $p$, [194].

*cut r x* is $r\langle\{x\}\rangle$, replaced by *im_singleton* [43].

*diagonal A*, $\Delta_{\mathrm{A}}$, is the set of all $(x,x)$ such that $x \in \mathrm{A}$, [40].

*diagonal_application A* is the diagonal mapping $x \mapsto (x,x)$ of A into $\Delta_{\mathrm{A}}$, [61].

*diagonal_graphp I E* is the set of graphs of constant functions from I to E, [97].

*disjoint x y* means $x \cap y = \emptyset$, [84].

*disjoint_union f, disjoint_union_fam f* are two variants of the disjoint union of the family of sets $f$, [87].

*domain f* is the set of $x$ for which there is an $y$ with $(x,y) \in f$, it is $\mathrm{pr}_1\langle f\rangle$, [33].

*doubleton x y*, $\{x,y\}$, is a set with elements $x$ and $y$, [25].

*EEE* is a shorthand for the type *Set → Set → Set*.

*EEP* is a shorthand for the type *Set → Set → Prop*.

*elt x y*, $x \ni y$, is the same as $y \in x$, [20].

*empty_function, empty_functionC* is the identity on $\emptyset$, [50].

*emptyset*, $\emptyset$, is a set without elements, [21].

*eq_rel_associated f* is the graph of the equivalence relation $f(x) = f(y)$, [114].

*equipotent x y* means that there is a bijection from $x$ to $y$.

*equivalence_associated f* is the equivalence relation $f(x) = f(y)$, [114].

*equivalence_r r, equivalence_re r x*, says that the relation $r$ is an equivalence relation (in $x$), [109].

*equivalence_corr r* says that the correspondence $r$ is associated to an equivalence, [203].

*exists_unique p*,$(\exists! x)p$, (this notation is not in Bourbaki) means that there exists a unique $x$ such that $p(x)$, [20].

*extends g f, extendsC g f* says $g(x) = f(x)$ whenever $f(x)$ is defined, [54].

*ext_map_prod I X Y g* is the function $(x_\iota)_{\iota\in\mathrm{I}} \mapsto (g_\iota(x_\iota))_{\iota\in\mathrm{I}}$ from $\prod_\mathrm{I} \mathrm{X}_\iota$ into $\prod_\mathrm{I} \mathrm{Y}_\iota$, [105].

*ext_to_prod u v* is the function $(x,y) \mapsto (u(x), v(y))$, sometimes denoted $u \times v$, [71]

*extension_to_parts f*, denotes the function $x \mapsto f\langle x\rangle$, from $\mathfrak{P}(\mathrm{A})$ to $\mathfrak{P}(\mathrm{B})$, [89]

*finer_equivalence s r*, comparison of equivalences, $x \overset{s}{\sim} y$ implies $x \overset{r}{\sim} y$, [128].

*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 $x \times y$, [117].

*fcompose f g*, $f \circ g$, composition of two graphs, without assumption, [35].

*fcomposable f g* says that graphs $g$ and $f \circ g$ have the same domain, [35].

*fgraph f* says that $f$ is a functional graph, [33].

*functional_graph f* says that $f$ is a functional graph, [46].

*fun_image x f*, $f\langle x\rangle$, is the value of $f$ on the set $x$, [27].

*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 $r'$), [122], [123].

*fun_set_to_prod E X* is the canonical bijection between $(\prod \mathrm{X}_\iota)^\mathrm{E}$ and $\prod \mathrm{X}_\iota^\mathrm{E}$, [106].

*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 $t$, [84].

*gcompose f g*, $f \circ g$, composition of two graphs, assumes that range $g$ is a subset of domain $f$, [35].

*graph f* is a part of a correspondence, [40].

*graph_on r X* is the graph of the relation $r$ restricted to X, [111].

*identity A*, $\mathrm{I_A}$, is is the graph of the identity function on the set A, [36].

*identity_fun A*, $\mathrm{I_A}$, is the identity function on the set A, [46].

*IM* stands for the image of a function. Its axioms implement the Scheme of Selection and Union, [20].

*image_by_fun f A*, $f\langle A\rangle$, is $\{t, \exists x \in A, t = f(x)\}$, [42].

*image_by_graph f A*, $f\langle A\rangle$ is $\{t, \exists x \in A, (x, t) \in f\}$, [42].

*image_of_fun f*, is the image of $f$, [42].

*inc x y* or $x \in y$ means that $x$ is an element of $y$, [17].

*inclusionC x y*, $I_{xy}$, is the inclusion map on $x \subset y$ as a Coq function, [52].

*induced_relation R A*, $R_A$, is the equivalence induced by R on A, [127].

*injective f*, *injectiveC f*, means that $f$ is an injection, [58].

*in_same_coset f* is the relation "there exists $i$ such that $x \in f(i)$ and $y \in f(i)$" between $x$ and $y$, [118].

*intersection X*, $\bigcap X$, is the intersection of a set of sets, [29].

*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 $a \in X_\iota$, [76].

*intersection2 X Y*, $X \cap Y$, is the intersection of two sets [29].

*intersection_covering*, intersection of coverings, [83].

*inverse_direct_value f X*, $X_f$, is $f^{-1}\langle f\langle X\rangle\rangle$, [120].

*inverse_graph G*, $\overset{-1}{G}$, inverse graph of the graph G, [43].

*inverse_fun f* or *inverseC a b f H*, $\overset{-1}{f}$, inverse of the function $f$, [44], [62].

*inverse_image x f*, $\overset{-1}{f}\langle x\rangle$, is the inverse value of $f$ on the set $x$, [34].

*inv_image_relation f r*, is the inverse image of the relation $r$ under the function $f$, [126].

*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, [44]

*inv_corr_value t* associates to a $t = (G, A, B)$ its correspondence $f$, [40].

*inv_graph_canon G* is the bijection $(x, y) \mapsto (y, x)$ from G to $G^{-1}$, [61].

*is_class r x* says that $x$ is an equivalence class for $r$, [115]

*is_correspondence f* says that $f$ is associated to a triple (G,A,B), [41]

*is_equivalence r* says that the graph $r$ is an equivalence, [110].

*is_function f* says that $f$ is a function in the sense of Bourbaki, [46].

*is_graph f* says that $f$ is a set of pairs, [33].

*is_graph_of g r* is true if $g$ is the graph of the relation $r$, [111].

*is_left_inverse r f* means that $r$ is a retraction or left-inverse of $f$, and $r \circ f$ is the identity, [64].

*is_reflexive r* says that the graph $r$ is reflexive, [110].

*is_restriction f g* says that $f$ is the restriction of $g$ to some set, [33]

*is_right_inverse s f* means that $s$ is a section or right-inverse of $f$, and $f \circ s$ is the identity, [64].

*is_singleton x* means that $x$ is a singleton.

*is_symmetric r* says that the graph $r$ is symmetric, [110].

*is_transitive r* says that the graph $r$ is transitive, [110].

*J x y*, or $(x, y)$, is an ordered pair, formed of two items $x$ and $y$, [31].

*L X f*, *fcreate X f*, $\mathscr{L}_X f$ is the graph formed of all $(x, f(x))$ with $x \in X$, [35].

*largest_partition x* is the set of all singletons of $x$.

*left_inverseC*, left inverse of a Coq function, [65].

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 $\{x, y\}$, [86].

*mutually_disjoint f* says that for all distinct $i$ and $j$, $f(i)$ and $f(j)$ are disjoint, [84]

$x \neq y$, *neq x y, x <> y* is inequality, [20].

*one_point* is the basic singleton, [26].

*P z*, $\mathrm{pr}_1 z$ denotes $x$ if $z$ is the pair $(x, y)$, [30].

*partial_fun1 f y, partial_fun1 f x,* partial functions, [70].

*partition y x, partition_s y x,partition_fam f x,* thee variants that say that $y$ or $f$ is a partition of $x$, [84].

*partition_relation f x* is the equivalence relation associated to the partition $f$ of $x$, [118].

*partition_with_complement X A,* is the partition of X formed of A and its complementary set, [86].

*powerset x*, $\mathfrak{P}(x)$, is the set of subsets of $x$, [27].

$\mathrm{pr}_1 z$, $\mathrm{pr}_2 z$ 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, [30].

*pr_i f i, pr_it f i*, $\mathrm{pr}_i f$, denotes a component of an element of a product. [95].

*pr_j f J*, $\mathrm{pr}_\mathrm{J} f$, is the function $(x_\iota)_{\iota\in\mathrm{I}} \mapsto (x_\iota)_{\iota\in\mathrm{J}}$, [98].

*prod_assoc_map* is the function whose bijectivity is the “theorem of associativity of products”, [99].

*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, [130].

*product A B*, $\mathrm{A} \times \mathrm{B}$, is the set of all pairs $(a, b)$ with $a \in \mathrm{A}$ and $b \in \mathrm{B}$, [32]. See also *ext_to_prod u v*.

*productt I X, product b g* or *productf I f*, $\prod\limits_{\iota\in\mathrm{I}} \mathrm{X}_\iota$ is the product of a family of sets, [94].

*product1 x a* is the product of the family defined on the singleton $\{a\}$ via value $x$, [96].

*product1_canon x a* is the canonical application from $x$ into *product1 x a*, [96].

*product2 x y* is the product of the family defined on the doubleton $\{a, b\}$ via value $x$ and $y$, [96].

*product2_canon x y* is the canonical application from $x \times y$ into *product2 x y*, [96].

*product_compose*, auxiliary function used for change of variables in a product, [97].

*Q z*, $\mathrm{pr}_2 z$ denotes $y$ if $z$ is the pair $(x, y)$, [30].

*quotient R*, E/R, is the set of equivalence classes of R, [115]

*quotient_of_relations r s*, R/S, is the quotient of two equivalences, [129]

*range f* is the set of $y$ for which there is an $x$ with $(x, y) \in f$, it is $\mathrm{pr}_2\langle f\rangle$, [33].

*reflexive_r r x* says that the relation $r$ is reflexive in $x$, [109].

*related r x y* is a shot-hand for $(x, y) \in r$, [39].

*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, [119]

*rep x* is an element $y$ such that $y \in x$, whenever $x$ is not empty, [23].

*representative_system s f x* means that, for all $i$, $s \cap \mathrm{X}_i$ is a singleton, where $\mathrm{X}_i$ is a partition of $x$ associated to the function $f$, [119].

*representative_system_function g f x*, means that $g$ is an injection whose image is a system of representatives (see definition above), [119].

*restr x G* is the restriction to $x$ of the graph G, [36].

*restricted_eq E* is the relation "$x \in$ E and $y \in$ E and $x = y$", [112].

*restriction_function f x* is like *restr*, but $f$ and the restrictions are functions, [53].

*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 $f(a) \in y$, [54].

*restriction2 f x y*, *restriction2C f x y*, restriction of $f$ as a function $x \to y$, [54].

*restrictionC f H* is the restriction to $x$ of the function $f : a \to b$, where H proves $x \subset a$ implicitly, [52].

*restriction_product f j* is the product of the restrictions of $\prod f$ to J, [98].

*restriction_to_image f* is the restriction of the Coq function $f$ to its range, [73].

retraction: see *is_left_inverse*.

RHS is the right hand side of an equality.

*right_inverseC*, right inverse of a Coq function, [65].

*Ro x* or $\mathscr{R} x$ converts its argument $x$ of type $u$ to a set, which is an element of $u$, [19].

*saturated r x* means: for every $y \in x$, the class of $x$ for the relation $r$ is a subset of $x$, [120].

*saturation_of r x* is the saturation of $x$ for $r$, [121].

*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*, [121].

*Set* or $\mathscr{E}$ is the type of sets, [17].

*set_of_correspondences A B* means the set of triples associated to correspondences from A to B, it is $\mathfrak{P}(\mathrm{A} \times \mathrm{B}) \times \{\mathrm{A}\} \times \{\mathrm{B}\}$, [41].

*set_of_endomorphisms E*, is the set of triples (G, E, E) associated to functions from E into E, [90]

*set_of_functions E F*, denoted $\mathscr{F}(\mathrm{E};\mathrm{F})$, is the set of triples (G, E, F) associated to functions from E into F, [90]

*set_of_gfunctions E F*, denoted $\mathrm{F}^{\mathrm{E}}$, is the set of graphs of functions from E to F, [90]

*set_of_sub_functions E F*, denoted $\Phi(\mathrm{E};\mathrm{F})$ is the set of triples (G, A, F) associated to functions from $\mathrm{A} \subset \mathrm{E}$ into F, [90]

*singleton x*, $\{x\}$, is a set with one element, [26].

*sof_value x y z* converts three elements into a correspondence, [90].

*small_set x* means that $x$ has at most one element, [50].

*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 $f$) [40], [46].

*strict_sub x y*, $x \subsetneq y$, means $x \subset y$ and $x \neq y$, [20].

*sub x y*, $x \subset y$, means that $x$ is a subset of $y$, [17].

*surjective f*, *surjectiveC f*, means that $f$ is a surjection, [58].

*substrate r* is the union of the domain and range [109].

*symmetric_r r* says that the relation $r$ is symmetric, [109].

*target f* contains the range of the graph of a correspondence $f$, [40].

*transf_axioms f A B* says that for all $x \in \mathrm{A}$ we have $f(x) \in \mathrm{B}$, case where $\mathscr{L}_{\mathrm{A;B}} f$ is a function, [55].

*transitive_r r* says that the relation $r$ is transitive, [109].

*two_points* is the basic doubleton, [25].

*union X*, $\bigcup \mathrm{X}$, is the union of a set of sets, [27],

*uniont I f, unionf x f, uniont g*, $\bigcup_{\iota \in \mathrm{I}} \mathrm{X}_\iota$ is the set of elements $a$ such that $a \in \mathrm{X}_\iota$ for some $\iota \in \mathrm{I}$, [76].

*union2 a b*, $a \cup b$, is the union of two sets, [28].

*V x f*, $\mathscr{V}(x, f)$ or $\mathscr{V}_f\, x$, is the value at the point $x$ of the graph $f$, [31].

*variant*, see *Lvariant*.

*W x f*, $\mathscr{W}_f\, x$, is the value at the point $x$ of the function $f$, [47].

*Xo f y*, $\mathscr{X}(f, y)$, this is $f(x)$ if $y = \mathscr{R} x$, [25].

*Yo P x y*, $\mathscr{Y}(\mathrm{P}, x, y)$, is a function that associates to $z$ the value $x$ is P is true, and $y$ if P is false, [23].

*Zo x R*, $\mathscr{Z}(x, \mathrm{R})$, $\mathscr{E}_x(\mathrm{R})$ or $\{x, \mathrm{R}\}$: it is the set of all $x$ that satisfy R, [17] [24].

# Index

# 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

Centre de recherche INRIA Sophia Antipolis – Méditerranée
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399