The Coq proof assistant user's guide : version 5.6

Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Benjamin Werner, Christine Paulin-Mohring

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1) 39 63 55 11

Rapports Techniques

1 9 9 2

ème anniversaire

N° 134

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

THE COQ PROOF ASSISTANT USER'S GUIDE

Version 5.6

Gilles DOWEK
Amy FELTY
Hugo HERBELIN
Gérard HUET
Christine PAULIN-MOHRING
Benjamin WERNER

Décembre 1991

The Coq Proof Assistant
User's Guide

Version 5.6 *

Gilles DOWEK – Amy FELTY – Hugo HERBELIN

Gérard HUET – Christine PAULIN-MOHRING – Benjamin WERNER

Projet Formel

INRIA-Rocquencourt† — **CNRS - ENS Lyon** ‡

*This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks" and by MRT Programme de Recherches Coordonnées "Programmation Avancée et Outils pour l'Intelligence Artificielle".

†B.P. 105, 78153 Le Chesnay CEDEX, France.

‡LIP URA 1398 du CNRS, 46 Allée d'Italie, 69364 Lyon CEDEX 07, France.

Cock-a-doodle-doo,
My dame has lost her shoe;
My master's lost his fiddlestick,
And knows not what to do.
— Mother Goose

Contents

Introduction

Coq is a proof assistant for higher-order logic, allowing the development of computer programs consistent with their formal specification. It is the result of about ten years of research of the Formel project. We shall briefly survey here three main aspects: the *logical language* in which we write our axiomatizations and specifications, the *proof assistant* which allows the development of verified mathematical proofs, and the *program extractor* which synthesizes computer programs obeying their formal specifications, written as logical assertions in the language.

The logical language used by Coq is a variety of type theory, called the *Calculus of Inductive Constructions*. Without going back to Leibniz and Boole, we can date the creation of what is now called mathematical logic to the work of Frege and Peano at the turn of the century. The discovery of antinomies in the free use of predicates or comprehension principles prompted Russell to restrict predicate calculus with a stratification of *types*. This effort culminated with *Principia Mathematica*, the first systematic attempt at a formal foundation of mathematics. A simplification of this system along the lines of simply typed λ-calculus occurred with Church's *Simple Theory of Types*. The λ-calculus notation, originally used for expressing functionality, could also be used as an encoding of natural deduction proofs. This Curry-Howard isomorphism was used by N. de Bruijn in the *Automath* project, the first full-scale attempt to develop and mechanically verify mathematical proofs. This effort culminated with Jutting's verification of Landau's *Grundlagen* in the 1970's. Automated theorem-proving was pioneered in the 1960's by Davis and Putnam in propositional calculus. A complete mechanization (in the sense of a semi-decision procedure) of classical first-order logic was proposed in 1965 by J.A. Robinson, with a single uniform inference rule called *resolution*. Resolution relies on solving equations in free algebras (i.e. term structures), using the *unification algorithm*. Many refinements of resolution were studied in the 1970's, but few convincing implementations were realized, except of course that PROLOG is in some sense issued from this effort. The most notable computer-aided proof-checkers developed in the 1970's were LCF, designed by R. Milner and his colleagues at U. Edinburgh, specialized in proving properties about denotational semantics recursion equations, and the Boyer and Moore theorem-prover, an automation of primitive recursion over inductive data types. While the Boyer-Moore theorem-prover attempted to synthesize proofs by a combination of automated methods, LCF constructed its proofs through the programming of *tactics*, written in a high-level functional meta-language, ML. During this period, notable achievements in proof theory saw the emergence of two type-theoretic frameworks; the first one, Martin-Löf's *Intuitionistic Theory of Types*, attempts a new foundation of mathematics on constructive principles. The second one, Girard's polymorphic λ-calculus $F\omega$, is a very strong functional system in which we may represent higher-order logic proof structures. Combining both systems in a higher-order extension of the Automath languages, T. Coquand presented in 1985 the first version of the *Calculus of Constructions*, CoC. This strong logical system allowed powerful axiomatizations, but direct inductive definitions were not possible, and inductive notions had to be defined indirectly through functional encodings, which introduced inefficiencies and awkwardness. The formalism was extended in 1989 by Coquand and Paulin with primitive inductive definitions, leading to the current *Calculus of Inductive Constructions*. This extended formalism is not rigorously defined here. Rather, numerous concrete examples are discussed. We refer the interested reader to relevant research papers for more information about the formalism, its meta-theoretic properties, and semantics. However, it should not be necessary to understand this theoretical material in order to write specifications. It is possible to understand the

Calculus of Inductive Constructions at a higher level, as a mixture of predicate calculus, inductive predicate definitions presented as typed PROLOG, and recursive function definitions close to the language ML.

A first implementation of CoC was started in 1984 by G. Huet and T. Coquand. Its implementation language was CAML, a functional programming language from the ML family designed at INRIA in Rocquencourt. The core of this system was a proof-checker for CoC seen as a typed λ-calculus, called the *Constructive Engine*. This engine was operated through a high-level notation permitting the declaration of axioms and parameters, the definition of mathematical types and objects, and the explicit construction of proof objects encoded as λ-terms. A section mechanism, designed and implemented by G. Dowek, allowed hierarchical developments of mathematical theories. This high-level language was called the *Mathematical Vernacular*. Furthermore, an interactive *Theorem Prover* permitted the incremental construction of proof trees in a top-down manner, subgoaling recursively and backtracking from dead-alleys. The theorem prover executed tactics written in CAML, in the LCF fashion. A basic set of tactics was predefined, which the user could extend by his own specific tactics. This system (Version 4.10) was released in August 1989. Since then, the system has been extended to deal with the new calculus with inductive types by C. Paulin, with corresponding new tactics for proofs by induction. A new standard set of tactics has been streamlined, and the vernacular extended for tactics execution. A new user-interface, relying on a CAML-X interface by D. de Rauglaudre, has been designed and implemented by A. Felty. It allows operation of the theorem-prover through the manipulation of windows, menus, mouse-sensitive buttons, and other widgets. The first part of this manual is a user's guide to these theorem-proving facilities. An appendix contains many transcripts of actual proofs developed with this system.

The salient feature which clearly distinguishes our proof assistant from say LCF or Boyer and Moore's, is its possibility to extract programs from the constructive contents of proofs. This computational interpretation of proof objets, in the tradition of Bishop's constructive mathematics, is based on a realisability interpretation, in the sense of Kleene, due to C. Paulin. The user must just mark his intention by separating in the logical statements the assertions stating the existence of a computational object from the logical assertions which specify its properties, but which may be considered as just comments in the corresponding program. Given this information, the system automatically extracts a functional term from a consistency proof of its specifications. This functional term may be in turn compiled into an actual computer program, in CAML or some other functional language, using a translation package designed and implemented by B. Werner. This methodology of extracting programs from proofs is a revolutionary paradigm for software engineering. Program synthesis has long been a theme of research in artificial intelligence, pioneered by R. Waldinger. The Tablog system of Z. Manna and R. Waldinger allows the deductive synthesis of functional programs from proofs in tableau form of their specifications, written in a variety of first-order logic. Development of a systematic *programming logic*, based on extensions of Martin-Löf's type theory, was undertaken at Cornell U. by the Nuprl team, headed by R. Constable. The first actual program extractor, PX, was designed and implemented around 1985 by S. Hayashi from Kyoto University. It allows the extraction of a LISP program from a proof in a logical system inspired by the logical formalisms of S. Feferman. Interest in this methodology is growing in the theoretical computer science community. We can foresee the day when actual computer systems used in applications will contain certified modules, automatically generated from a consistency proof of their formal specifications. We are however still far from being able to use this methodology in a smooth interaction with the standard tools from software engineering, i.e. compilers, linkers,

run-time systems taking advantage of special hardware, debuggers, and the like. We hope that Coq can be of use to researchers interested in experimenting with this new methodology. The second part of this manual explains how to extract ML programs from Coq proofs.

Running the System

How to get Coq

The simplest way is to import it by FTP (File Transfer Program). If your site has access to FTP, here is the sequence of commands to follow:

```
ftp nuri.inria.fr*
Connected to nuri.inria.fr.
220 nuri FTP server ready.
Name (seti.inria.fr:huet):†
Password: myself@mymachine.myinstitution.mycountry
331 Guest login ok, access restriction apply
ftp> cd INRIA/coq
250 CWD command successful.
ftp> binary
ftp> get README
ftp> get coq.tar.Z
ftp> quit
```

If you do not have access to FTP, write us, preferably by e-mail to `coq@margaux.inria.fr`, specifying your equipment and system.

The file coq.tar.Z, once uncompressed, must be de-archived by tar, which will give you a directory `coq`, with sub-directories `DOC`, `SRC`, `CORES`, `EXAMPLES`, and a number of files: `README`, `coq`, `constr`. `DOC` contains the dvi format of this manual, `SRC` contains the CAML source files, `EXAMPLES` contains a selection of examples. `CORES` will contain entries for every machine architecture you want to run Coqon. For instance, if you are on a sun4 machine, the installation of the system as explained below will create an entry `coq.sun4.core`.

Getting the system to run

In order to run the system, you need CAML version 3.1. CAML may also be obtained by FTP. Do, similarly as above: `ftp> cd /lang/caml`

```
250 CWD command successful.
ftp> get README
ftp> cd V3.1
```

*If your name server does not find `nuri`, you may access with absolute addressing: `ftp 128.93.1.26`

†here you type in `anonymous`

```
ftp> get README.FIRST
```

This last file tells you which further files to get in order to get a proper distribution for your architecture.

We shall assume here that your computer or workstation is running the UNIX operating system, with the X window interface, version 11, release 4, or an equivalent such as SUN OpenWindows.

Once you have CAML installed, you have to install Coq. In directory `coq/SRC`, do `make coq` to create the right core image. This assumes that your UNIX knows the command `/bin/arch` naming the architecture. Otherwise, replace the value of `ARCH` in the Makefile by any string naming your architecture.

Once Coqis installed, you may run it by calling the command `coq` from the directory `coq`. You should get, after a few seconds, the banner:

```
Coq                                    - Version 5.6 du 10 Decembre 1991
```

and then the CAML prompt `ml #`.

If this does not work, check the contents of the command file `coq`, which should read:

```
caml V3.1 -big -r CORES/coq.$ARCH.core 40
```

The trouble may come from `caml` not being known. Make the proper PATH adjustment, abbrev declaration, or link. The trouble may be an inconsistency of core images. For instance, the image `coq.core` is not loadable from older versions of CAML.

If you have trouble with the X library from CAML, a simplified system may be obtained with `make constr`. This simplified system does not contain the X window interface.

Navigating in the system

Initially, you are in CAML's top-level, indicated by its prompt sign `ml #`. You may in this state execute any CAML top-level phrase. For instance, if you want to run the system from an Emacs shell window, you should execute initially `echo false;;`.

From CAML's top level, you may exit permanently in a graceful manner, by typing in `quit();;`, followed by RETURN, or forcibly by sending the QUIT signal (usually bound to key CTRL \). You may interrupt with the INT signal (usually bound to key CTRL C). You may escape to the surrounding shell by sending the STOP signal (usually bound to key CTRL Z). In this last case, you may reenter your CAML session with the shell command `fg`.

In CAML's top level, you may load tactic files, using the standard commands `load` and `compile`. You may also operate directly the Constructive Engine in a step-by-step fashion, but this is not the usual mode of operation, which is to give Vernacular commands. Vernacular commands may be loaded from a file, say `th1.v`, by the command `V "th1"`. This is the standard way to initialize a mathematical theory, by loading in the corresponding vernacular file which defines its vocabulary, axioms, definitions, theorems. For instance, initially the system loaded a file `Prelude.v`, which defines basic logical connectives and inference rules, and a file `Nat.v` defining the type of natural numbers.

Such vernacular commands may also be executed interactively, by entering the main loop of the system, by the CAML command `coq();;`. You are now in the vernacular's top-level loop, indicated by the prompt `Coq <`. All vernacular commands are terminated with a period. You may come back to CAML from the vernacular loop with the command `Drop`.

A manual of the Constructions Vernacular is given below in Section 1.1. This basic vernacular

is a higher-level notation that compiles mathematical definitions, variable and axiom declarations, and proofs into operations of the Constructive Engine.

After a little practice with the use of the basic vernacular, the user may attempt to use the Tactics Theorem Prover. This is a goal-directed inference engine, in the spirit of Prolog, but with a proof-search mechanism driven by tactics in the spirit of the LCF proof assistant. Basic tactics are predefined, and the user may extend this initial set of tactics by writing his own tactics and tacticals in CAML.

The theorem prover is entered from the vernacular loop by the command `Goal`, which sets the initial goal you are trying to prove. Most tactics are accessible from the vernacular loop, with their own syntax. More complex tactics may be defined and executed from CAML. It is always possible to escape to CAML from the vernacular by typing in: `#(... CAML top-level phrase ...)`.

When the proof is completed, with the tactics assistance, the theorem may be entered in the current context under the name `name` by the vernacular command `Save name`. You may then go to the next proof, etc. You may backtrack from dead-ends using the command `Undo`, and you may completely abort the current search for a proof with the command `Abort`. If you want to abort the current proof, but restart with the same top-level goal, use command `Restart`.

It is possible to record a proof development in a vernacular file. The command `Open transcript` will open the file `transcript.v`, without truncating it. From then on, all vernacular commands will be appended to this file. Recording ceases with executing command `Close`. The recording of an interactive proof is not done step by step, but proof by proof, at the time when the `Save` command is executed. This way, dead alleys backtracked from with `Undo` are not recorded. Using the Tactics Theorem-Prover is described in Section 1.5.

It is possible to drive the theorem prover from a visual interface with the X window system. The standard settings assume you have a color screen. If you have a black and white screen, you must type in `COLOR:=false;;` to CAML first. You enter the interface from the CAML loop by executing `X();;`. This opens two windows on your screen. One, labeled `Coq`, is used to drive the theorem prover. The other, labeled `Context`, is used to print sections of the current context. You may exit from the interface by pressing the `Quit` button in the `Coq` window, or by sending the INT signal. A complete manual of the interface is provided below in Section 1.7.

It is possible to extract from a proof an algorithm, which corresponds to its constructive contents. This algorithm may actually be constructed, in the form of a computer program in a dialect of the ML language. This facility is explained in Section 2.2 below.

If you have difficulties in installing or operating the system, or if you discover an anomalous behaviour, you may send electronic mail to `coq@margaux.inria.fr`, with a clear description of the trouble, including the machine/system you are running, and the banner of the `Coq` system you are running.

Chapter 1

The System Coq as a Proof Checker

1.1 A Typed λ-calculus

In the System Coq one can express and prove propositions (such as "two is even" or "even(two)"). These propositions concern objects (here "two"). These objects are represented by terms of a typed λ-calculus: the Calculus of Inductive Constructions.

1.1.1 Parameters and Terms

Base objects are declared with the instructions **Parameter** and **Inhabits**.

```
Parameter nat.
Inhabits Set.

Parameter O.
Inhabits nat.

Parameter S.
Inhabits nat -> nat.
```

Remark that since the lexical conventions of the system Coq forbid the numeral 0 as an identifier we have used the letter O.

A shorter syntax for these declarations is:

```
Parameter nat:Set.

Parameter O:nat.

Parameter S:nat -> nat.
```

In this calculus the types are terms. They all belong to a single predefined type **Set**.

Once base objects are declared, terms may be built with three constructions: application, λ-abstractions and product formation.

- Application is the application of a functional term to an argument. An example is `(S O)`.

• Abstraction permits the formation of functional terms. An example is `[x:nat](S (S x))` which denotes the function which maps a natural number x to the successor of its successor.

• Product formation permits one to build types for functional terms. An example is `nat -> nat`. All these products are of type `Set`.

1.1.2 Contexts

A term in which parameters occur is well-formed only when these parameters have been declared. For instance the term `(plus x x)` is well-formed only when the parameters `plus` and `x` are declared.

At any point in the use of the system Coq the list of the parameters already declared form the current *context*. (Actually the current context is the list of all the parameters, definitions, axioms and theorems already declared.) Notice that in the term `[x:nat](plus x x)` the parameter `x` does not have to be declared because it is bound by λ-abstraction. So the term `[x:nat](plus x x)` is well typed in the context $[nat : Set; plus : nat \rightarrow nat \rightarrow nat]$.

1.1.3 Reduction

Let us consider the terms `[x:nat](S (S x))` and `(S 0)`. The first one is a function that maps every natural number to the successor of its successor, the second is the natural "one". When we apply the first to the second, we get the term `([x:nat](S (S x)) (S 0))`. This term can be reduced to `(S (S (S 0)))` by the rule of replacement of formal parameters by actual arguments. In the following, such equivalent terms will be identified.

1.1.4 Polymorphism, Type Constructors and Dependent Types

In this λ-calculus types are terms and thus it is possible to abstract over the type `Set` and also to abstract types. For instance the term `[x:T]x` can be abstracted over the type variable `T` to give the term `[T:Set][x:T]x` which is the *polymorphic* identity.

When we want to give a type to this term a problem occurs. This term is a function. The domain of this function is the type `Set`. But the codomain of this function cannot be defined simply since it depends on the value to which this function is applied. Thus the arrow notation is not sufficient here and we have to give a richer syntax for such function types.

We let this term be of type `(T:Set)(T -> T)` where the notation `(x:P)Q` is an extension of the arrow notation `P -> Q`.

The typing rule for application is that when a term of type `(x:P)Q` is applied to a term `t` of type `P`, the result is of type $Q[x \leftarrow t]$.

In the previous example, when `[T:Set][x:T]x` is applied to the term `nat` we get the term `[x:nat]x` whose type is `(nat -> nat)` which is $(T \rightarrow T)[T \leftarrow nat]$.

When the type of the result does not depend on the value of the argument, (for instance when `[x:nat]x` is applied to any natural number, the type of the result is always `nat`) we write `(nat -> nat)` instead of `(x:nat)nat`. Thus `(P -> Q)` is really an abbreviation for `(x:P)Q` when `x` has no occurrence in `Q`.

Note that this function type construction is also a binding operator and thus `x` is not free in `(x:P)Q`.

1.1.5 Abbreviations

In the process of constructing a term we may want to use definitions. A definition relates a name to a term. Then this name can be used and everything proceeds as if all the occurrences of this name were replaced by the abbreviated term.

This can be done with the `Definition` and `Body` instructions.

```
Definition plus_two.
Body [x:nat](S (S x)).
```

A shorter syntax for this definition is:

```
Definition plus_two = [x:nat](S (S x)).
```

A more explicit syntax where the type is given with the term can also be used:

```
Definition plus_two = [x:nat](S (S x)):nat -> nat.
```

or:

```
Definition plus_two:nat -> nat = [x:nat](S (S x)).
```

Then the term `(plus_two (S O))` is equivalent to `([x:nat](S (S x)) (S O))` i.e. is equivalent to `(S (S (S O)))`.

1.1.6 Local Declarations and Local Definitions

When we define a term representing a function, rather than writing it `[x:nat]t`, it is sometimes easier to write it `t` under the local declaration `x:nat`. This can be done by inserting the declaration `x:nat` between the two parts of the definition. In order to indicate that this declaration is local, the keyword `Parameter` is replaced by the keyword `Variable`.

```
Definition plus_two.
   Variable x.
   Inhabits nat.
Body (S (S x)).
```

The scope of a variable `x` is restricted to the definition of `plus_two`. Out of this definition the variable `x` is discharged, so that the term `plus_two` is bound to `[x:nat](S (S x))`.

This can be compared with the usage in mathematics: "Let x be a natural number, (*plus_two* x) is the successor of the successor of x" instead of: "Let *plus_two* be the function that maps every natural to the successor of its successor".

Local definitions can be stated in the same way, the keyword `Definition` being replaced by `Local`. The scope of the defined symbol is also restricted to the definition. Outside of this definition the occurrences of the defined symbols are replaced by the term they abbreviate.

1.1.7 Sections

When we want a local declaration or definition to be shared by several definitions we may use the paragraph mechanism. A paragraph is opened with the instruction `Section` *name* and closed with the instruction `Leave` *name*. For instance:

```
Section Sums.
   Variable x:nat.
   Definition plus_two = (S (S x)).
   Definition plus_three = (S (S (S x))).
Leave Sums.
```

will define `plus_two = [x:nat](S (S x))` and `plus_three = [x:nat](S (S (S x)))`.

In some cases this section mechanism is too clumsy. For instance, assume we have a function `plus:nat -> nat -> nat`:

```
Parameter plus:nat -> nat -> nat.
```

Assume that we want to define the two functions: `f = [x:nat][y:nat](plus (plus x y) y)` and `g = [x:nat](plus (plus x x) x)` The commands:

```
Section two_functions.
   Variables x,y:nat.
   Definition f = (plus (plus x y) y).
   Definition g = (plus (plus x x) x).
Leave two_functions.
```

will define the functions:
`f = [x:nat][y:nat](plus (plus x y) y)` and `g = [x:nat][y:nat](plus (plus x x) x)`. An undesired `[y:nat]` has been added to the definition of g. Indeed the section mechanism is not supposed to remark that the variable y does not occur in the term g. When such a test is desired you have to use the keyword `End` instead of `Leave`. For instance

```
Section two_functions.
   Variables x,y:nat.
   Definition f = (plus (plus x y) y).
   Definition g = (plus (plus x x) x).
End two_functions.
```

will define `f = [x:nat][y:nat](plus (plus x y) y)` and `g = [x:nat](plus (plus x x) x)`.

Remark that this keyword `End` cannot be used when one wants to define a function which does not use all of its arguments. For instance if we want to define the function `p = [x:nat][y:nat]x`. The commands:

```
Section first_projection.
   Variables x,y:nat.
   Definition p = x.
End first_projection.
```

will define the function `p = [x:nat]x`. Here, the keyword `Leave` should have been used.

1.1.8 Inductive Types

If we look at the functions that can be represented by a term in this language we get a very poor set of functions. It contains the constant functions, the projections, the successor function and it is closed by composition. One important construction is missing: inductive definitions. In order to get induction we must give a better definition of natural numbers: we shall define natural numbers as an inductive type.

This part only contains useful examples. A more detailed explanation of the rules for inductive definitions can be found in section 1.8.

Declaration of an inductive type

Coq provides the possibility to define inductive types like the type of natural numbers, of lists, trees, ... An inductive type is specified by giving the signature of its constructors. The effect of the declaration of an inductive type in Coq is quite similar to the effect of the declaration of a concrete type in ML. A new type, terms for the constructors of this type and a destructive operation for an element of the type using a match-like structure are added to the environment.

For example, we may define the type of unary natural numbers built from 0 and successor as:

```
Inductive Set nat = O : nat | S : nat->nat. (* Beware: O stands for O *)
```

Since the scheme of inductive types in Coq is more general than the scheme in ML, the syntax of the declaration is slightly more complicated. In ML only the type of the argument of the constructor is indicated. For example, a similar structure of natural numbers will be declared in CAML as

```
type nat = O | S of nat;;
```

In Coq, we indicate the complete type of the constructor.

It is possible to define a constructor with more than one argument, so we do not need the intermediate product constructor like in ML. For instance the type of lists of natural numbers is defined with :

```
Inductive Set natlist = nil : natlist | cons : nat -> natlist -> natlist.
```

It is possible to introduce inductive types depending on parameters, for example polymorphic lists by :

```
Inductive Set list [A:Set]
        = nil : (list A) | cons : A -> (list A) -> (list A).
```

In this definition, `list` will be a type constructor which associates to each type `A` an inductive type with two constructors. The occurrences of `list` in the type of the constructors must appear only in sub-expressions of the form `(list A)`.

Inductive definitions are not necessarily recursive. The product is a special case of an inductive definition with one constructor which takes two arguments:

```
Inductive Set prod [A,B:Set] = pair : A->B->(A*B).
```

In Coq `A*B` (resp. `<A,B>(a,b)`) is an alternative syntax for `(prod A B)` (resp. `(pair A B a b)`).

Another example of a non-recursive inductive definition is the disjunct sum of two sets. This set is generated by two constructors corresponding to the left and right injections.

```
Inductive Set sum [A,B:Set] = inl : A -> (A+B) | inr : B -> (A+B).
```

In Coq `A+B` is an alternative syntax for `(sum A B)`.

Examples of programs using inductive types

We just showed how to declare an inductive definition. We now explain how to take apart an object of an inductive type.

A closed element of an inductive type in reduced form is a term $(c_i\ t_1 \ldots t_n)$ with c_i a constructor. The principle of the destructive operation is the following one : to define a function on an inductive type, it is sufficient to give the value of the function for terms starting with each constructor.

The basic idea is intuitive but the syntax is, at the moment, rather difficult. The general rule is: if the type of the term `t` is an inductive type and if `P` is a type then `<P>Match t with` is a term of the Calculus of Inductive Definitions. Its type depends on the type of `t`.

The match operation contains the usual destructuring operations on terms like in ML. The patterns are restricted to the form of a constructor applied to variables. The match must check all the cases in the order of the constructors as listed in the type declaration.

Let us give a few examples:

The product type. If we want to define a function `f` over the product type `A*B`, we can describe the value of `f` for a pair term `<A,B>(a,b)` with `a` and `b` new variables of type `A` and `B`.

If `p` has type `A*B` then `<P>Match p with` has type `(A->B->P)->P`. This term allows the access to both components of an element of `A*B`. The term `(<P>Match <A,B>(a,b) with f)` is reducible to `(f a b)`. For example the first projection can be written as:

```
Fst : (A,B:Set)(A*B)->A
    = [A,B:Set][u:A*B](<A>Match u with (* x,y *) [x:A][y:B]x).
```

Let `t` be a term possibly containing `x` and `y` as free variables. The construction `(<P>Match u with [x:A][y:B]t)` is analogous to the ML scheme `match u with (x,y)->t` often written as `let (x,y) = u in t`.

The sum type. If we want to define a function `f` over the disjunct sum type `A+B`, we can describe the value of `f` for terms `(inl A B a)`, `(inr A B b)` (corresponding to the left and right injections) with `a` and `b` new variables of type `A` and `B`.

If `p` has type `(sum A B)` then `<P>Match p with` has type `(A->P)->(B->P)->P`. This term allows a definition by cases on `p`. The term `(<P>Match (inl A B a) with f1 f2)` is reducible to `(f1 a)` and the term `(<P>Match (inr A B b) with f1 f2)` is reducible to `(f2 b)`. Let `t1` be a term possibly containing `x` as a free variable and `t2` be a term possibly containing `y` as a free variable. The construction `(<P>Match u with [x:A]t1 [y:B]t2)` is analogous to the ML scheme `match u with (inl x) -> t1 | (inr y) -> t2`. To emphasize this analogy, we usually add comments (inside `(* *)`) in the Coq construction and write:

```
(<P>Match u with (* inl x *) [x:A]t1 (* inr y *) [y:B]t2)
```

The type of natural numbers. If the inductive definition is really recursive like the one of natural numbers, we want more than just the possibility to define a function by cases (whether the argument is 0 or $(S\ n)$). The point is that the language does not provide a general recursion scheme like in ML. It only allows a recursive definition of a function on natural numbers which follows a primitive recursive scheme. This scheme is also expressed with the `Match` syntax.

Primitive recursion means that given two terms f and g, we may define a function H on natural numbers which satisfies the following equations:

$$(H\ 0) = f \qquad (H\ (S\ n)) = (g\ n\ (H\ n))$$

If we want H to be a function from natural numbers to a type P, then f has to be of type P and g a function which takes as arguments a natural number and a term of type P and gives a term of type P. In Coq, the function H will be represented by the following term:

```
Definition H : nat -> P
 = [n:nat](<P>Match n with (* 0 *) f (* S p *) g)
```

For instance the definition of the addition of two natural numbers can be given as:

```
Definition plus : nat->nat->nat
 = [n,m:nat](<nat>Match n with (* 0 *) m
                               (* S p *) [p:nat][pluspm:nat](S pluspm))
```

Once more, the strings `(* 0 *)` and `(* S p *)` are just comments in this syntax.

If `n` has type `nat`, then the expression `<P>Match n with` denotes a term of the calculus whose type is `P->(nat->P->P) ->P`. If `f` has type `P` and `g` has type `nat->P->P` then the term `H` defined by `[n:nat](<nat>Match n with f g)` is such that `(H 0)` is convertible with (i.e. reduces to) `f` and `(H (S n))` is convertible with `(g n (H n))`. With `nat` for the type `P`, it is used to define primitive recursive functions. In our example `(plus 0 m)` is convertible with `m` (the first argument of the match operation) and `(plus (S p) m)` is convertible with `(S (plus p m))` (the second argument `[p:nat][pluspm:nat](S pluspm)` applied to `p` and to the recursive call `(plus p m)`).

Another example is the predecessor function which is specified by the equations:

$$(P\ 0) = 0 \qquad (P\ (S\ n)) = n$$

It is be represented in the Coq system as the term:

```
Definition pred : nat->nat
 = [n:nat](<nat>Match n with (* 0 *) 0 (* S p *) [p:nat][predp:nat]p)
```

This scheme is not restricted to the definition of an integer. It may also be used to define a function. We can directly represent primitive recursive functionals of any order. This possibility allows also a direct representation of functions. Assume that we want to define the difference `minus` between two natural numbers. It may be specified as: `(minus 0 m)` is equal to `0`, `(minus (S p) 0)` is equal to `(S p)` and `(minus (S p) (S q))` is equal to `(minus p q)`. The most direct way to represent `minus` following these equations is to remark that `(minus (S p))` can be defined as a term depending only on `p` and `(minus p)`. Thus the definition of `(minus n)` follows a primitive recursive scheme. If `G` is the term:

```
[p:nat][minusp:nat->nat][m:nat]
  (<nat>Match m with (* O *) (S p)
                     (* S q *) [q:nat][minusSpq:nat](minusp q))
```

then (G p (minus p) O) is equal to (S p) and (G p (minus p) (S q)) is equal to (minus p q). So (G p (minus p)) has the expected behavior of (minus (S p)). Let us take for the definition of minus:

```
[n:nat](<nat->nat>Match n with (* O *) [m:nat]O
                               (* S p *) G)
```

We then directly get, through the internal conversion rule, the expected equalities. The same kind of analysis can be used for the definition of a non-primitive recursive function like the Ackermann function. The expected equalities are (ack O m) is equal to (S m), (ack (S p) O) is equal to (ack p (S O)) and (ack (S p) (S q)) is equal to (ack p (ack (S p) q)). We let the reader check that these equalities are satisfied by the following term:

```
[n:nat]
(<nat->nat>Match n with
         (* O *)  S
       (* S p *)  [p:nat][ackp:nat->nat][m:nat]
                  (<nat>Match m with (* O *) (ackp (S O))
                                     (* S q *) [q:nat][ackSpq:nat](ackp ackSpq)))
```

The type of lists. The case of lists as defined above is an extension of the case of natural numbers. If p has type (list A) then <P>Match p with has type P->(A->(list A)->P->P)->P. This term is analogous to the primitive recursive scheme for lists. We get: (<P>Match nil with f g) reduces to f and (<P>Match (cons A a l) with f g) reduces to (g a l (<P>Match l with f g)). For instance the length of a list can easily be defined by:

```
length : (A:Set)(list A)->nat
  = [A:Set][l:(list A)]
    (<nat>Match l with (* nil *) O
                       (* cons a m *) [a:A][m:(list A)][lgm:nat](S lgm))
```

Primitive Constructions

In the initial prelude of the system, the types nat and bool are defined as inductive types. So are the cartesian product and the disjoint sum of two sets. See the file Prelude.v at the beginning of the appendix for more information.

1.2 Propositions

After writing objects we can write propositions concerning these objects.

1.2.1 Atomic Propositions and Predicates

An atomic proposition is built by applying a predicate to some objects. For instance, if P is a one place predicate over the type nat, and n a term of type nat, then (P n) is a proposition.

In Coq these propositions are terms, the type of propositions is a predefined sort: Prop.

Parameters of sort Prop can be declared in the usual way:

```
Parameter A:Prop.
Parameter B:Prop.
```

Predicates and relations are terms of a functional type. For instance, a one place predicate over the type nat has type (nat -> Prop). Let us for instance declare a binary relation over type nat.

```
Parameter Lower_eq : nat -> nat -> Prop.
```

The term (Lower_eq 0 (S 0)) is a proposition. The term (Lower_eq (S 0) 0) is also a well-formed proposition, and the distinction between true and false propositions will be made later.

1.2.2 Minimal Logic

Propositions are built from atomic propositions with connectives and quantifiers. In minimal logic we use only one connective: implication and one quantifier: the universal quantifier. The other connectives and quantifers will be studied in the next section. If P and Q are two propositions then P -> Q is a proposition. The proposition P -> Q should be read "P implies Q". For instance (Lower_eq 0 0) -> (Lower_eq 0 0) is a proposition. If P is a proposition where a free variable x of type T may occur, (x:T)P is a proposition. The proposition (x:T)P should be read "for all x in T, P". For instance (x:nat)(Lower_eq 0 x) is the proposition usually written $\forall x \in \mathcal{N}\ 0 \leq x$. In Coq it is possible to quantify over any data type. For instance (f:nat -> nat)(Lower_eq 0 (f 0)) is a proposition. It is also possible to quantify over predicates and propositions. For instance (P:nat -> Prop)(P 0) -> ((n:nat)(P n) -> (P (S n))) -> (x:nat)(P x) is a proposition expressing the induction schema of arithmetic. Note that implication and the universal quantifier have the same binding strength and are both right-associative. The proposition (P:Prop)P expresses that all propositions are true (This proposition is itself certainly false, except in an inconsistent context). The symbol -> is overloaded, since it is used both for functional type formation and implication. In the same way the notation (x:T)P is used both for functional type formation and universal quantification.

1.2.3 Connectives and Quantifiers

- If P and Q are two propositions, then P /\ Q is the proposition "P and Q".
- If P and Q are two propositions, then P \/ Q is the proposition "P or Q".
- If P is a proposition then ~P is the proposition "not P".
- False is the absurd proposition.
- True is the tautological proposition.
- If P is a term of type Prop in the current context extended with the declaration $x : T$, then <T> Ex ([x:T]P) is the proposition "there exists an x in T such that P".

1.2.4 Equality

Equality is a predicate of type : `(T:Set)(T -> T -> Prop)`. So if a and b are terms of type A then `(eq A a b)` is a proposition. A shorter syntax for this proposition is `<A>a=b`.

1.3 Axioms

Before proving theorems we have to assume axioms. This can be done with the instructions `Axiom` and `Assumes`.

```
Axiom u.
Assumes A->B.

Axiom v.
Assumes A.

(* Lower_eq is antisymmetric *)

Axiom Antisym.
Assumes (x:nat)(y:nat)(Lower_eq x y) -> (Lower_eq y x) -> (eq nat x y).

(* Lower_eq is transitive *)

Axiom Trans.
Assumes (x:nat)(y:nat)(z:nat)(Lower_eq x y) -> (Lower_eq y z)
                                            -> (Lower_eq x z).
```

1.4 Proving Theorems

This section shows how Coq can be used to check explicit formal proofs written in natural deduction style. Readers familiar with natural deduction may skip this section in a first reading. Proofs are developed in practice with the help of an interactive Theorem Prover, which is described in Section 1.5.

1.4.1 Proving Theorems in Natural Deduction

Using an axiom

- If there is in the current context an axiom `u` of statement `P` then we can prove `P`.

A proof of this theorem is simply written `u`.

Implication

- If there is in the current context axioms or theorems `u` and `v` of statements `P -> Q` and `P`, then we can prove `Q`.

A proof of this theorem is written `(u v)`.

```
Theorem b.
Statement B.
Proof (u v).
```

- If we could prove in the current context extended with the declaration of an axiom `P` a theorem of statement `Q` then we prove in the current context a theorem `P -> Q`.

To write a proof of this theorem, we consider the theorem `Q` and we insert between the instructions `Theorem` and `Statement` the declaration of the axiom `P`. The axiom `P` may be used locally in the proof of `Q` but cannot be used anywhere else. A local axiom is declared between the instructions `Theorem` and `Statement` with the instructions **`Hypothesis`** and **`Assumes`**.

For instance, let us prove the theorem `B -> B` in a context where `B` is a proposition.

```
Theorem I.
   Hypothesis x.
   Assumes B.
Statement B.
Proof x.
```

Note that the statement of the theorem is written `B` and not `B -> B` and the given proof is a proof of `B`, but since the hypothesis `x` is local to the theorem `I`, the theorem is finally a proof of `B -> B`.

Universal Quantification

- If there is in the current context an axiom or a theorem `u` of statement `(x:T)P` (meaning $\forall x : T \cdot P$) and the term `t` has type `T` in this context then we can prove `P` where `x` has been instantiated by `t`.

A proof of this theorem is written `(u t)`.

```
Parameter e.
Inhabits nat.

Parameter f.
Inhabits nat.
```

We can prove the theorem **`(Lower_eq e f) -> (Lower_eq f e) -> (<nat>e=f)`**.

```
Theorem A1.
Statement (y:nat)((Lower_eq e y) -> (Lower_eq y e) -> (<nat> e = y)).
Proof (Antisym e).

Theorem A2.
Statement ((Lower_eq e f) -> (Lower_eq f e) -> (<nat> e = f)).
Proof (A1 f).
```

- If we can write in the current context extended with the declaration of a variable `x:T` a theorem of statement `P` then we can prove in the current context a theorem `(x:T)P`.

The use of this rule is exactly the same as the one for introduction of implication. The only difference is that the local declaration is a variable and not an hypothesis. We have already seen that a local variable was declared with the instructions `Variable` and `Inhabits`.

For instance let us prove `(C:Prop)(C -> C)`.

```
Theorem I.
   Variable C.
   Inhabits Prop.
   Hypothesis x.
   Assumes C.
Statement C.
Proof x.
```

Conjunction

- If there are in the current context axioms or theorems `u` and `v` of statements `P` and `Q` respectively, then we can prove `P /\ Q`.

A proof of this theorem is written `(conj P Q u v)`.

```
Theorem z.
Statement (A/\B).
Proof (conj A B v w).
```

- If there is in the current context an axiom or a theorem `u` of statement `P /\ Q` then we can prove `P`.

A proof of this theorem is written `(proj1 P Q u)`.

```
Theorem p.
Statement A.
Proof (proj1 A B z).
```

- If there is in the current context an axiom or a theorem `u` of statement `P /\ Q` then we can prove `Q`.

A proof of this theorem is written `(proj2 P Q v)`.

```
Theorem q.
Statement B.
Proof (proj2 A B z).
```

Disjunction

- If there is in the current context an axiom or a theorem **u** of statement **P** then we can prove **P \/ Q**.

A proof of this theorem is written **(or_introl P Q u)**.

```
Theorem y.
Statement (A\/B).
Proof (or_introl A B v).
```

- If there is in the current context an axiom or a theorem **u** of statement **Q** then we can prove **P \/ Q**.

A proof of this theorem is written **(or_intror P Q u)**.

```
Theorem z.
Statement (A\/B).
Proof (or_intror A B w).
```

- If there are in the current context axioms or theorems **u, v** and **w** of statements respectively **(P \/ Q)**, **(P -> C)** and **(Q -> C)** then we can prove **C**.

A proof of this theorem is written **(or_ind P Q C u v w)**.

```
Variable C.
Inhabits Prop.

Axiom u'.
Assumes (A->C).

Axiom v'.
Assumes (B->C).

Theorem w'.
Statement C.
Proof (or_ind A B C u' v' z).
```

Absurd and Negation

- If there is in the current context an axiom or a theorem **u** of statement **False** and **P** is any proposition in this context then we can prove **P**.

A proof of this theorem is written **(False_ind P u)**.

```
Axiom abs.
Assumes False.

Theorem a.
Statement A.
Proof (False_ind A abs).
```

- If there are in the current context axioms or theorems of statements `P` and `~P` then we can prove `False`.
- If we can write in the current context extended with the local hypothesis `P` a theorem of statement `False` then we can prove `~P` in the current context.

Actually since we consider `~P` as an abbreviation for `P -> False`, these two rules are particular cases of the rules for `->`. So the way we write proofs constructed with these rules is just a particular case of the way used for `->`.

Existential quantification

- If in the current context `t` is of type `T`, `P` is a proposition in the current context extended with the declaration `x:T`, and there is in the current context an axiom or a theorem `u` of statement `P[x ← t]` then we can prove `<T> Ex ([x:T]P)` (which means $\exists x : T \cdot P$).

A proof of this theorem is written `(ex_intro T ([x:T](P x)) t u)`.

```
Variable T.
Inhabits Set.

Variable t.
Inhabits T.

Variable Q.
Inhabits T->Prop.

Axiom u.
Assumes (Q t).

Theorem E.
Statement (<T> Ex ([x:T](Q x))).
Proof (ex_intro T ([x:T](Q x)) t u).
```

- If there are in the current context axioms or theorems `u` and `v` of statements respectively `<T> Ex ([x:T]P)` and `(x:T)P->C` where `x` does not appear free in `C` then we can prove `C`.

  A proof of this theorem is written `(ex_ind T ([x:T]P) C v u)`.

```
Variable C.
Inhabits Prop.

Axiom v.
Assumes (x:T)(Q x)->C.

Theorem E'.
Statement C.
Proof (ex_ind T Q C v E).
```

Equality

- If there are in the current context axioms or theorems `u` and `v` of statements `(<A> a = b)` and `(P a)` respectively then we can prove `(P b)`.

A proof of this theorem is written `(eq_ind A a P v b u)`.

Lemmas

In the process of proving a theorem we may want to use lemmas. These lemmas can be theorems themselves, but generally we do not want to keep them outside the scope of the theorem. Such local theorems can be inserted between the statement and the proof of a theorem in which they are defined using the instructions **Remark**, **Statement**, **Proof**.

Summary of the Instructions

	Declarations	Definitions	Axioms	Theorems
Global	**Parameter** **Inhabits**	**Definition** **Body**	**Axiom** **Assumes**	**Theorem** **Statement** **Proof**
Local	**Variable** **Inhabits**	**Local** **Body**	**Hypothesis** **Assumes**	**Remark** **Statement** **Proof**

1.4.2 λ-terms as Proofs

In the previous section we have seen how to write proofs of theorems. The notation given for each natural deduction rule may look a bit mysterious. Moreover we have seen that some constructions were overloaded, for instance the arrow `->` was used both for functional types and for implication, and the application $(u\ v)$ was used both for function application and for the rule $\rightarrow$-elim. We are now going to justify the syntax of the proofs and the overloading of these symbols.

Heyting's Semantics for Minimal Propositional Logic

Let us consider first the propositions only built from propositional variables and implication: $\rightarrow$.

In order to prove theorems we have three natural deduction rules: the rule concerning the use of an axiom and the rules concerning implication.

Heyting's semantics proposes to associate to each propositional variable a set: the set of its proofs (this set may be empty if the proposition cannot be proved) and to define recursively the set of proofs of $P \rightarrow Q$ as the set of functions which map every proof of P to a proof of Q (here also this set may be empty).

When we assume an axiom P, we postulate that there is an element in the set of proofs of P. When we want to prove a theorem Q we try to exhibit, using the elements given with the axioms, an element of the set of the proofs of this proposition.

For instance, if we have the two axioms $A \to B$ and A and we want to prove B, then we have elements f in the set of proofs of $A \to B$ and a in the set of proofs of A. The function f maps every proof of A to a proof of B, and so $(f\ a)$ is a proof of B.

As a second example, we do not assume any axiom. The identity function $[x]x$ maps every proof of A to a proof of A, so it is a proof of the proposition $A \to A$.

It is very easy to show by induction over the length of π that a proof π of a proposition P written in natural deduction can always be translated into a λ-term which is a proof of this proposition and that if a proposition P is proved in a context Γ then the free variables of its proof-term are proofs of the axioms of Γ.

Curry-Howard Isomorphism for Minimal Propositional Logic

Let us now consider the types of the proofs of some proposition P.

As we have in the previous section introduced for each propositional variable A *the set of its proofs*, we now introduce *the type of its proofs*, this type is also written A.

Since the set of proofs of $P \to Q$ is the set of functions from P to Q, we let $P \to Q$ be the type of the proofs of $P \to Q$.. So we may unify a proposition and the type of its proofs, by unifying the functional arrow and the implicational arrow.

Then it is easy to prove that all the proof-terms introduced in the previous section are well-typed in simply typed λ-calculus and that their types are the propositions they prove.

For instance if f is a proof of $A \to B$ and a is a proof of A then f has type $A \to B$ and a has type A. So the proof of B, $(f\ a)$ has type B.

Moreover every term of type P can be translated to a proof of P. Terms and proofs can therefore be identified.

Hence we have defined an isomorphism between propositions and types, proofs and terms, variable declarations and axiom assumptions, definitions and theorems. This isomorphism is called the Curry-Howard isomorphism.

Universal Quantification

Heyting's semantics suggests the representation of a proof of $\forall x : T \cdot P$ by a function which maps every term of type T to a proof of $P[x \leftarrow t]$.

These proofs look like proofs of $Q \to P$. The main difference is that when f is a proof of $Q \to P$ and t a proof of Q, t occurs in the proof $(f\ t)$ of P, but not in the proposition P itself, and when f is a proof of $\forall x : T \cdot P$ the term t occurs in the proof $(f\ t)$ and may also occur in the proposition $P[x \leftarrow t]$.

Thus these terms are not well-typed in the simply typed λ-calculus. Fortunately we have seen that the Calculus of Constructions is an extension of simply typed λ-calculus allowing dependent types, polymorphism and type constructors, and in which such terms may be typed.

Thus in the Calculus of Constructions each natural deduction proof can be represented and each term can be seen as a proof of its type.

In order not to confuse the proof terms of the underlying logic with the elements of the mathematical structures which we axiomatize, we shall use two distinct sorts: `Prop` is the type of propositions of the logic, whereas `Set` is the type of data type specifications. Thus there are two sorts of (first-order) quantification: `(x:T)P`, with `T` of type `Set`, is of type `Prop` (resp. `Set`) if `P` is of type `Prop` (resp. `Set`) in a context where `x` is of type `T`. This second possibility corresponds to the

usual notion of an indexed family of sets. It is also permitted to form `(x:Q)P`, with `Q` of type `Prop`, but this construction is natural only when `x` does not occur in `P`, in which case it is equivalent to the arrow `Q->P` of implication.

It is also possible to write higher-order quantifications, quantifying over types such as `Prop`, `Set`, `Prop->Prop`, etc. We shall see an example in the next section.

Connectors and Quantifiers

Since it is possible in Coq to form products and to quantify over any type, we can give types to the connectives and quantifiers and express the natural deduction rules as propositions. For instance for conjunction we have:

$$and : Prop \to Prop \to Prop$$

The $\wedge$-introduction rule is:

$$(P : Prop)(Q : Prop)(P \to Q \to (and\ P\ Q))$$

and the $\wedge$-elimination rules are:

$$(P : Prop)(Q : Prop)((and\ P\ Q) \to P)$$

and:

$$(P : Prop)(Q : Prop)((and\ P\ Q) \to Q)$$

Notice that these rules are expressed in minimal logic. So it is sufficient to find four terms:

$$and : Prop \to Prop \to Prop$$

$$conj : (P : Prop)(Q : Prop)(P \to Q \to (and\ P\ Q))$$

$$proj1 : (P : Prop)(Q : Prop)((and\ P\ Q) \to P)$$

and:

$$proj2 : (P : Prop)(Q : Prop)((and\ P\ Q) \to Q)$$

to get conjunction for free.

We could for instance declare parameters of these types, but it is easy to show that they contain closed terms, using either inductive types or polymorphism.

1.5 The Tactics Theorem Prover

Proving theorems by giving the exact proof as a λ-term becomes tedious very quickly as the complexity of the theorems increases.

For this purpose, a Theorem Prover, inspired from LCF and similarly based on the notion of tactics, has been developed in the core of the Coq system. By means of the Theorem Prover, the essential steps of a proof become the application of lemmas, theorems or axioms, reasoning by induction, and reduction of a constant to its definition. Most interactions with the underlying λ-calculus structure is avoided. All this makes the development of a proof more friendly and closer to usual reasoning. Moreover, automatic proof search is provided.

The Tactics Theorem Prover deals with only one goal at a time. To this goal corresponds a proof tree, first composed of a single leaf, and built step by step by tactics. In fact, a tactic is a function which builds a proof of a given goal from proofs of more elementary subgoals. New unproven leaves corresponding to subgoals are therefore generated by successive applications of tactics. To each of these subgoals is associated a local context, allowing the discharge of quantified variables and of premises in order to use them easily when need be.

Tactics are presented as commands of the vernacular. Each of them has its own keyword and often takes some arguments. Like vernacular commands, they are terminated by a period. By default, they apply to the first subgoal, but it is always possible to deal with the other subgoals. After trying to apply a tactic to a goal, either it fails and an error message is displayed, or the new state of the Theorem Prover with the remaining subgoals is displayed.

Backtracking is possible, since the Theorem Prover keeps the previous states of a proof search session. It is also possible to restart a proof from the beginning. Unless you abort, you must completely prove a goal in order to be able to finish a session of the Theorem Prover. Once proved, the procedure to follow is to save the goal with its proof in the current context of Coq. Then the Theorem Prover forgets all traces of the goal and of its proof, and is free to deal with another goal. It is not permitted to leave some part of the proof incomplete, but it is possible to declare axioms during a session of the Theorem Prover, and to use these axioms to artificially solve parts of the current goal. It is also possible to declare definitions, or directly prove theorems without escaping the Theorem Prover. However, these declarations, and axiom declarations too, are fragile: they will be kept if you backtrack during the proof, but they will be lost if you restart the proof from the beginning, or abort it, and moreover, you cannot reset them without first aborting the goal.

The Theorem Prover also offers the possibility to record a session. It is a very helpful but fragile facility. This facility is all the more useful since it is possible to use the Theorem Prover not only in an interactive mode but also by reading commands and tactics from a vernacular file. For this purpose, it is also possible to disable the displaying of the successive states of the proof search. This very much speeds up the loading delay.

A panorama of the tactics

The Tactics Theorem Prover offers seven kinds of tactics, as well as a set of tacticals which can be used to combine tactics or alter their effect.

- Introduction tactics. These are the tactics which discharge the hypotheses and variables of the goal into the local context.
- Exact tactics. These are the tactics which allow one to give a complete exact proof of a subgoal.
- Resolution tactics. These are the tactics which reduce the proof to more elementary proofs by applying an already proved theorem or an axiom.
- Elimination tactics. These are tactics which reduce inductive constants to their constructors. They correspond to reasoning by cases, as well as reasoning by induction.
- Convertibility tactics. These are the tactics which change a goal by replacing a constant by its definition, by computing some subexpressions of the goal in order to simplify it, or

by abstracting a parameter in order to reason by induction on it. In λ-calculus terms, they correspond to tactics which change a goal into an equivalent one, modulo $\beta\delta$-conversion and modulo inductive constant elimination rules.

- Context convertibility tactics. These are the tactics which change, not the goal, but the hypotheses or parameters of its context. Currently they corresponds only to tactics which replace constants in hypotheses by their definitions.

- Automatic search tactics.

  These include two high-level tactics which try automatic resolution from a given set of more elementary tactics.

- Tacticals. In addition, we must mention the possibility for the user to define his own tactics by writing them in the metalanguage CAML. This manual does not explain how to program such tactics.

1.5.1 An interactive session of the Tactics Theorem Prover

Here, we will explain by means of interactively developed examples how to work with the Coq Theorem Prover.

Moreover, for more examples of use of the Tactics Theorem Prover, we recommend to the reader to consult the vernacular files `Prelude.v`, `Specif.v` and `Nat.v` (those which define the initial state of Coq) as well as the report presenting the vernacular proof of Gilbreath Trick (see file `Shuffle.v`). These files are listed in the appendix of this document.

Entering the Tactics Theorem Prover : Goal

First you need to be in the vernacular's top-level loop.

```
Coq <
```

You then stipulate what you want to prove. Let us take for instance one well-known tautology (corresponding to the S combinator of combinatory logic).

```
Coq < Goal (A,B,C:Prop)(A->B->C)->(A->B)->A->C.
```

This command has the effect that the current goal becomes the given statement.

Aborting the Tactics Theorem Prover : Abort

You cannot consider several goals at the same time. If you want to change the goal, you first need to abort the current one.

```
Coq < Abort.
Current goal aborted
```

Displaying the current goal : Show

Then you can verify that there is no longer a current goal by means of Show :

```
Coq < Show.
No current goal
```

Let us give the goal again and show it :

```
Coq < Goal (A,B,C:Prop)(A->B->C)->(A->B)->A->C.

Coq < Show.
1 subgoal
  (A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C
```

Discharge of universally quantified variables and premises : Intros

The Intros tactic discharges universally quantified variables and hypotheses of the goal in the local context. This allows easy access of these variables and hypotheses, simply by designating them by their name when need be.

Let us apply this tactic :

```
Coq < Intros.
```

After the use of a tactic, the system shows you what remains to be proved. In our case, we have to prove C under the hypotheses A, A->B and A->B->C, where A, B and C are propositional variables.

```
1 subgoal
  C
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
```

Refinement by application of an axiom, a theorem, or an assumption : Apply

It is possible now to use another tactic. It appears that applying the hypothesis H may be fruitful. This is done by Apply followed by the name of the hypothesis :

```
Coq < Apply H.
2 subgoals
  A
  ============================
    H1 : A
    H0 : A->B
```

```
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
subgoal 2 is:
  B
```

Exact resolution : Exact

Now, we have two subgoals to solve. The first one, to which the tactics apply by default, is directly provable by the hypothesis `H1`. We indicate this by the **Exact** tactic :

```
Coq < Exact H1.
1 subgoal
  B
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
```

Here `H1` is an hypothesis, but in general, **Exact** may take any proof term of Coq as argument.

Backtracking : Undo

Backtracking is done by `Undo`. Here we backtrack in order to explain another tactic applicable to the goal `A`.

```
Coq < Undo.
2 subgoals
  A
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
subgoal 2 is:
  B
```

Exact resolution by an hypothesis : Assumption

Since `H1` is an hypothesis of the local context, it is possible to use the **Assumption** tactic. In fact, this tactic looks for an hypothesis of the local context which can complete the goal, modulo

replacements of constants by their definition, or computing of certain subexpressions. This avoids having to give explicitly the name of the hypothesis :

```
Coq < Assumption.
1 subgoal
  B
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
```

Composition of tactics : ";"

Next we want to solve the second subgoal by using the hypothesis H0. It will remain to prove A, which is one of the hypotheses of the context and thus provable by **Assumption**. It is possible to apply a sequence of tactics one after the other by separating them by semi-colons. In this case, the second tactic will be applied to each subgoal generated by the first, and so on for the remaining tactics after the second semi-colon. For instance, in our case :

```
Coq < Apply H0; Assumption.
Goal proved!
```

Restarting a proof search : Restart

For illustration purposes, we will now restart the proof :

```
Coq < Restart.
Current goal restarted

Coq < Show.
1 subgoal
  (A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C
```

Composition of tactics (continued)

We will now see what happens if we had given the tactics **Intros**, **Apply H** and **Assumption** in a row.

```
Coq < Intros; Apply H; Assumption.
Error No such assumption
```

An error is generated because **Assumption** is applied to the two subgoals resulting from the application of **Apply H**, and one of them is not provable by assumption.

Let us note also that the sequence **Intros; Apply H; Assumption.** is considered as a unique command, in such a way that if one tactic fails, the complete command fails and the subgoal is not modified. In fact :

```
Coq < Show.
1 subgoal
  (A:Prop)(B:Prop)(C:Prop)(A->B->C)->(A->B)->A->C
```

Capture of a possible failure : Try

One way to avoid that is to use Try which catches the failure of a tactic if it occurs. We invoke now the new command line :

```
Coq < Intros; Apply H; Try Assumption.
1 subgoal
  B
  ============================
    H1 : A
    H0 : A->B
    H : A->B->C
    C : Prop
    B : Prop
    A : Prop
```

We now finish the proof :

```
Coq < Apply H0; Assumption.
Goal proved!
```

Saving a proved goal in the global context : Save

When the proof is completed, we can give it a name and save it in the global context of Coq.

```
Coq < Save Combinatoric_S.
Combinatoric_S is defined
```

The Theorem Prover is now freed. The goal just proved is forgotten :

```
Coq < Show.
No current goal
```

Elimination of inductive constants : Elim

Logical or, written as \/, is a constant which is defined in the initial context of Coq (more exactly, it is defined by loading the vernacular file `Prelude.v`). To it, an elimination theorem is associated, whose statement is `(C:Prop)(A->C)->(B->C)->(A\/B)->C` (remembering the or-elimination rule of the natural deduction). Instead of using the tactic Apply applying this elimination theorem, it is possible to use the tactic Elim :

Let us prove, for instance, the commutativity of disjunction :

```
Coq < Goal (A,B:Prop)(A\/B)->(B\/A).

Coq < Intros.
1 subgoal
  B\/A
  ============================
    H : A\/B
    B : Prop
    A : Prop

Coq < Elim H.
2 subgoals
  A->(B\/A)
  ============================
    H : A\/B
    B : Prop
    A : Prop
subgoal 2 is:
  B->(B\/A)
```

The proof may now be completed by using the introduction theorems for `or` (see the section about inductive definitions for more details).

Displaying of a secondary subgoal : Show

Let us profit of this example to point out that the context of local hypotheses printed by the theorem prover is just the one pertaining to the first subgoal. In this example, the second subgoal has the same local context as the first one, but it is not displayed. In general, each subgoal has its own local context. If you want to see the local context of some subgoal, just give `Show` its number as explicit parameter. For instance, here,we can show the second subgoal with its context using the command `Show 2.` :

```
Coq < Show 2.
subgoal 2 is:
  B->(B\/A)
  ============================
    H : A\/B
    B : Prop
    A : Prop
```

Dealing with a secondary subgoal : "n :"

It is also possible to apply a tactic to a subgoal other than the first one, by typing n : before the tactic.

```
Coq < 2:Intro; Apply or_introl; Assumption.
```

```
1 subgoal
  A->(B\/A)
  ============================
    H : A\/B
    B : Prop
    A : Prop

Coq < Intro; Apply or_intror; Assumption.
Goal proved!
```

Reasoning by induction on an inductive predicate : Induction

Another way to reduce an inductively defined hypothesis to its primitive components is by using the tactic `Induction`. In our example the argument `1` of `Induction` means that all quantified variables of the goal up to the first non dependent hypothesis are introduced in the local context and that this last hypothesis must be eliminated.

```
Coq < Restart.
Current goal restarted

Coq < Induction 1.
2 subgoals
  A->(B\/A)
  ============================
    H : A\/B
    B : Prop
    A : Prop
subgoal 2 is:
  B->(B\/A)
```

We have now obtained the same as what we got by doing `Intros`, then `Elim H`.

Or introductions : Left, Right

We can now finish the proof by using the predefined tactics of `\/` introduction doing automatically introductions instead of using the names generated by the introduction theorems.

```
Coq < 2:Left; Assumption.
1 subgoal
  A->(B\/A)
  ============================
    H : A\/B
    B : Prop
    A : Prop

Coq < Right; Assumption.
Goal proved!
```

Let us also mention the existence of the tactic `Split` which introduces logical "and" as well as products.

Automatic proof search : Hint, Auto

`Auto` is a tactic which tries automatic search from a list of hints.

Let us restart the proof of the commutativity of `or` and let us declare the theorems `or_introl` and `or_intror` as hints. This is done by `Hint` :

```
Coq < Restart.
Current goal restarted

Coq < Hint or_introl or_intror.
```

Remark : this `Hint` declaration is not necessary because `or_introl` and `or_intror` are declared as hints in the initial prelude of the system.

We can now finish it in one step :

```
Coq < Induction 1; Auto.
Use : Intro ; Apply or_intror ; Assumption
Use : Intro ; Apply or_introl ; Assumption
Goal proved!
```

Remark that Coqgives you a trace of the execution of the `Auto` tactic with the `Use` comments.. From this trace, you may see that `Auto` combines `Intro` with `Assumption` in addition to applying the current hints.

Since we shall not restart the proof again, let us save it :

```
Coq < Save or_commut.
or_commut is defined
```

Dealing with equality

Leibniz' equality is a constant which is defined by means of the vernacular file `Prelude.v` in the initial context of the system. It is defined as an inductive predicate in such a way that doing substitution by equals is possible by means of inductive constant elimination tactics. More precisely `Elim H` where `H` is a term if type `t = u` rewrites in the current goal all occurrences of u in t.

```
Coq < Goal (n,p:nat)(<nat>p=n)->(<nat>(S p)=(S n)).

Coq < Intros.
1 subgoal
  <nat>(S p)=(S n)
  ============================
    H : <nat>p=n
    p : nat
    n : nat
```

```
Coq < Elim H.
1 subgoal
  <nat>(S p)=(S p)
  ============================
    H : <nat>p=n
    p : nat
    n : nat
```

And n is replaced by p everywhere.

```
Coq < Apply refl_equal.
Goal proved!
```

Automatic resolution of trivialities : Trivial

`Trivial` works with the same list of specified tactics or theorems as `Auto` but deals only with those which generate no subgoals, such as `Assumption` or `Apply refl_equal`. For instance, if we come back to the previous example:

```
Coq < Undo.
1 subgoal
  <nat>(S p)=(S p)
  ============================
    H : <nat>p=n
    p : nat
    n : nat

Coq < Hint refl_equal.

Coq < Trivial.
Use : Apply refl_equal
Goal proved!

Coq < Save S_equal.
S_equal is defined
```

Dealing with symmetry of equality

Consider the following closely related problem:

```
Coq < Goal (A:Set)(x,y:A)(<A>x=y)->(P:A->Prop)(P y)->(P x).

Coq < Intros.
1 subgoal
  (P x)
  ============================
```

```
    H0 : (P y)
    P : A->Prop
    H : <A>x=y
    y : A
    x : A
    A : Set
```

Elimination of H would replace y by x but not x by y. The tactic **Rewrite** is what is needed here:

```
Coq < Rewrite H; Assumption.
Goal proved!
```

A more explicit syntax is **Rewrite -> H**, and the dual syntax **Rewrite <- H** gives a more readable reverse rewriting than its equivalent **Elim H**.

Another natural solution is to use the tactic **Replace** as follows:

```
Coq < Undo.
1 subgoal
  (P x)
  ============================
    H0 : (P y)
    P : A->Prop
    H : <A>x=y
    y : A
    x : A
    A : Set

Coq < Replace x with y; Assumption.
Goal proved!
```

Elimination of a not yet proved statement : ElimType

A still less direct way would be to use **ElimType** :

```
Coq < Undo.
1 subgoal
  (P x)
  ============================
    H0 : (P y)
    P : A->Prop
    H : <A>x=y
    y : A
    x : A
    A : Set

Coq < ElimType (<A>y=x); Trivial.
```

```
Use : Assumption
Use : Idtac
1 subgoal
  <A>y=x
  ============================
    H0 : (P y)
    P : A->Prop
    H : <A>x=y
    y : A
    x : A
    A : Set
```

The remaining subgoal may then be proven by explicit appeal to the symmetry of equality:

```
Coq < Apply sym_equal; Assumption.
Goal proved!
```

Automatic proof search (continued) : Immediate

Some theorems when applied several times do not generate simpler subgoals. It is the case, for instance, for symmetry of equality. To avoid that the automatic search tactic **Auto** does unnecessary search, it is possible to add this theorem to the hint list by means of **Immediate**. This means that **Auto** will apply it only if the remaining subgoals are immediately provable by an hypothesis, an axiom or a theorem which does not generate subgoals.

```
Coq < Restart.
Current goal restarted

Coq < Immediate sym_equal.

Coq < Intros; ElimType (<A>y=x); Auto.
Use : Assumption
Use : Assumption
Use : Apply sym_equal ; Trivial
Goal proved!

Coq < Save sym_equal_elim.
sym_equal_elim is defined
```

Reasoning by induction on data types : Induction

Elimination of inductive **Set** is like other induction theorems. This gives an elegant way to do induction proofs. First, here again is the definition of **nat**:

```
Coq < Inductive Set nat = 0 : nat | S : nat -> nat.

Coq < Goal (n:nat)(<nat>n=0)\/(<nat>Ex([n':nat](<nat>n=(S n')))).
```

```
Coq < Induction n.
2 subgoals
  (<nat>0=0)\/<nat>Ex([n':nat](<nat>0=(S n')))
  ============================
    n : nat
subgoal 2 is:
  (y:nat)
   ((<nat>y=0)\/<nat>Ex([n':nat](<nat>y=(S n'))))->
    ((<nat>(S y)=0)\/<nat>Ex([n':nat](<nat>(S y)=(S n'))))
```

Two subgoals have been generated, one for the base case ($n = 0$) and one for the induction step. The base case is done by left or-introduction then reflexivity of equality. `Auto` can prove it alone :

```
Coq < Auto.
Use : Apply or_introl ; Apply refl_equal
1 subgoal
  (y:nat)
   ((<nat>y=0)\/<nat>Ex([n':nat](<nat>y=(S n'))))->
    ((<nat>(S y)=0)\/<nat>Ex([n':nat](<nat>(S y)=(S n'))))
  ============================
    n : nat
```

The induction step is more readable after discharging all hypotheses:

```
Coq < Intros.
1 subgoal
  (<nat>(S y)=0)\/<nat>Ex([n':nat](<nat>(S y)=(S n')))
  ============================
    H : (<nat>y=0)\/<nat>Ex([n':nat](<nat>y=(S n')))
    y : nat
    n : nat
```

In fact for this proof, the induction hypothesis is not used. As expected `Induction` also allows pattern matching.

```
Coq < Right.
1 subgoal
  <nat>Ex([n':nat](<nat>(S y)=(S n')))
  ============================
    H : (<nat>y=0)\/<nat>Ex([n':nat](<nat>y=(S n')))
    y : nat
    n : nat
```

Resolution tactics with parameters : Apply with, Exists

Introduction of the existential quantifier needs a "witness". A way to give this witness is to use the `with` option of `Apply`, which provides the instantiation:

```
Coq < Apply ex_intro with y.
1 subgoal
  <nat>(S y)=(S y)
  ============================
    H : (<nat>y=0)\/<nat>Ex([n':nat](<nat>y=(S n')))
    y : nat
    n : nat
```

The reflexivity of equality is given as a hint in the prelude, so `Trivial` completes the proof:

```
Coq < Trivial.
Use : Apply refl_equal
Goal proved!

Coq < Save O_or_Successor.
O_or_Successor is defined
```

We could have used the predefined tactic "`Exists y`" instead of `Apply ex_intro with y`.

Abstraction of terms : Pattern

`Apply` only does first-order matching, and thus fails in the following example:

```
Coq < Axiom induction :
Coq <   (P:nat->Prop)(P 0)->((m:nat)(P m)->(P (S m)))->(n:nat)(P n).
induction is assumed

Coq < Definition plus [n,m:nat](<nat>Match n with m [n',q:nat](S q)).
plus is defined

Coq < Goal (n:nat)(<nat>(plus n 0)=(plus 0 n)).

Coq < Intro.
1 subgoal
  <nat>(plus n 0)=(plus 0 n)
  ============================
    n : nat
```

Remark that `Apply induction` would in fact succeed here, but not with the expected result. To make apparent the intended structure `(P n)` in this goal, one can use `Pattern` :

```
Coq < Pattern n.
1 subgoal
  ([n:nat](<nat>(plus n O)=(plus O n)) n)
  ============================
    n : nat
```

Then, application of the induction principle generates the right subgoals:

```
Coq < Apply induction.
2 subgoals
  <nat>(plus O O)=(plus O O)
  ============================
    n : nat
subgoal 2 is:
  (m:nat)
   (<nat>(plus m O)=(plus O m))->(<nat>(plus (S m) O)=(plus O (S m)))
```

The base case is trivial :

```
Coq < Trivial.
Use : Apply refl_equal
1 subgoal
  (m:nat)
   (<nat>(plus m O)=(plus O m))->(<nat>(plus (S m) O)=(plus O (S m)))
  ============================
    n : nat
```

Simplification and computing : Simpl

Some expressions of the induction step can be simplified. For instance `(plus (S m) n)` reduces to `(S (plus m n))`. The tactic `Simpl` performs such a reduction.

```
Coq < Simpl.
1 subgoal
  (m:nat)(<nat>(plus m O)=m)->(<nat>(S (plus m O))=(S m))
  ============================
    n : nat
```

Abstraction of terms (continued)

The proof can now be finished by a substitution of `m` with `(plus m O)`. But only the second occurrence of `m` needs to be replaced. This is not directly allowed by `Elim` since this tactic would abstract all the occurrences of `m`.

```
Coq < Intros.
1 subgoal
  <nat>(S (plus m O))=(S m)
```

```
  ============================
    H : <nat>(plus m O)=m
    m : nat
    n : nat

Coq < Elim H.
1 subgoal
  <nat>(S (plus (plus m O) O))=(S (plus m O))
  ============================
    H : <nat>(plus m O)=m
    m : nat
    n : nat
```

This leads nowhere !

The term `m` occurs (several times) in the goal. We want to use the elimination of equality to replace the term `m` by the term `(plus m O)`, but we only want to replace one occurrence of `m` and not all of its occurrences. In order to use the elimination of equality we have to write the goal as `(P m)` and then the tactic `Elim` will transform it into `(P (plus m O))` but by default the tactic `Elim` will take `P = [m:nat]G` (where `G` is the goal) and substitute all the occurrences of `m`. So we use the tactic `Pattern` to put the second occurrence of `m` in evidence and write `G = (P m)` such that `(P (plus m O))` is the goal G in which only the second occurrence of m is substituted by `(plus m O)`.

```
Coq < Undo.
1 subgoal
  <nat>(S (plus m O))=(S m)
  ============================
    H : <nat>(plus m O)=m
    m : nat
    n : nat

Coq < Pattern 2 m.
1 subgoal
  ([n:nat](<nat>(S (plus m O))=(S n)) m)
  ============================
    H : <nat>(plus m O)=m
    m : nat
    n : nat

Coq < Elim H.
1 subgoal
  <nat>(S (plus m O))=(S (plus m O))
  ============================
    H : <nat>(plus m O)=m
    m : nat
    n : nat
```

This is now trivial and we can save the proof:

```
Coq < Trivial.
Use : Apply refl_equal
Goal proved!

Coq < Save plus_O_commut.
plus_O_commut is defined
```

Recording mode : Open, Close

It would have been possible to record in a file, say `demo.v` all the previous interactive development. For instance, assume that we give the command **Open demo** just before the first goal declaration and the command `Close` just after the `sym_equal_elim` proof. The file `demo.v` would contain the following lines:

```
Goal (A,B,C:Prop)(A->B->C)->(A->B)->A->C.
Intros; Apply H; Try Assumption.
Apply H0; Assumption.
Save Combinatoric_S.
Hint or_introl or_intror.
Goal (A,B:Prop)(A\/B)->(B\/A).
Induction 1; Auto.
Save or_commut.
Hint refl_equal.
Goal (n,p:nat)(<nat>p=n)->(<nat>(S p)=(S n)).
Intros.
Elim H.
Trivial.
Save S_equal.
Immediate sym_equal.
Goal (A:Set)(x,y:A)(<A>x=y)->(P:A->Prop)(P y)->(P x).
Intros; ElimType (<A>y=x); Auto.
Save sym_equal_elim.
```

Of course, no trace of displaying or backtracking by means of **Undo**, **Restart**, **Abort**, is recorded. Only "clean" proofs are recorded. Such a clean proof may be read successfully with the command `Load`.

It is also possible to open a recording file in the middle of a proof, but with no guarantee that the corresponding file will load successfully. Also, if you `Close` the recording file before completion of a proof and successful `Saving`, the partial proof steps will be recorded in the file.

Note: the argument of the **Open** command is the name of the recording file in the current working directory (with suffix `.v`). If the file does not exist, it is created; if it exists, it is appended to. If you want to address a file with an explicit absolute or relative path, use the CAML command `open_vernacular`.

1.5.2 Description of the tactics

Introduction tactics

- `Intro`

  This tactic is the basic introduction tactic.

  It assumes that the subgoal is not an atomic proposition and fails if it is atomic.

  If the subgoal is a quantified proposition, then it discharges the quantified variable into the local context associated to the subgoal.

  If the name, say `x` of the variable is already used either in the current context of Coq, or in the local context, it chooses a name `x`n where n is the first number such that `x`n is a new name.

  If the subgoal is an implication, i.e. of the form `P->Q`, it introduces an hypothesis `H:P` into the local context and the subgoal becomes `Q`.

  If the hypothesis name `H` is already used either in the current context of Coq, or in the local context, it chooses a name `H`n where n is the first number such that `H`n is a new name.

- `Intro` *name*

  This tactic works like `Intro` except that it forces the name of the variable or the hypothesis to be *name*. It fails if the name *name* is already used.

- `Intros`

  This tactics repeats `Intro` as often as it is possible. It never fails.

  It is a synonym of `Repeat Intro`.

- `Intros` $name_1 \ldots name_n$

  This tactics repeats `Intro` *name* successively with the names $name_1 \ldots name_n$. It fails if there are more names than variables or hypotheses to introduce in the local context.

- `Intros until` *name*

  This tactics repeats `Intro` until it introduces the variable named *name*. It fails if *name* does not exist.

Exact tactics

- `Exact` *term*

  This tactic allows the user to give an exact proof term. It fails if the term is not a correct proof of the subgoal. Of course, the correctness of the proof is checked modulo unfolding of constants and computation of terms.

  It can be useful when a goal is very simple to prove and the tactics of Coq are not powerful enough to solve it with the expected simplicity (for instance for some cases of substitutions by equals).

- **Assumption**

  This tactic looks for a proof by an assumption in the local context. It fails if no hypothesis of the local context proves the goal.

The special case of Instantiate

- **Instantiate** *term*

  This command, in fact, is not a tactic : in contrast with tactics, it applies to all the subgoals simultaneously. It is the only way to proceed when variables, so called "meta"-variables (standing for incomplete proof terms) are left in the subgoals. Effectively it is the only command which can solve subgoals with "meta"-variables and which can propagate the instantiation of them through the other subgoals. See **Apply** below for some examples of its use.

Resolution tactics

- **Apply** *term*

  This tactic is the basic resolution tactic. The goal must not be a product. Let *statement* be the type of *term*. Usually *term* is a proof or an hypothesis and its type *statement* is the statement it proves or assumes. Be careful that if *statement* is just an atomic proposition, then its head constant is unfolded, and this unfolding is repeated as long as the *statement* stays atomic.

  For instance :

```
Coq < Definition not_not_not [A:Prop]~~~A.
not_not_not is defined

Coq < Goal (A:Prop)(not_not_not A)->A->False.

Coq < Intros.
1 subgoal
  False
  ============================
    H0 : A
    H : (not_not_not A)
    A : Prop

Coq < Apply H.
1 subgoal
  ~~A
  ============================
    H0 : A
    H : (not_not_not A)
    A : Prop
```

And `not_not_not` than `~` have been unfolded in `H` before resolution with `False`.

`Apply` tries a first-order matching of the goal with the conclusion of *statement*. The tactic fails if the matching fails. If the matching is successful, then `Apply` generates as many subgoals as the number of premises in *statement*. When the head term of the conclusion of *statement* is not a constant but a variable then the good way to use `Apply` is to "prepare" the subgoal first with `Pattern` (cf the description of `Pattern`).

The Theorem Prover appears to be dealing with proofs and theorems, but in fact, because of the identification of proofs and λ-terms, propositions and types, it also deals with types from which an inhabitant is searched.

Of course, proving that a type is inhabited is usually trivial in practical examples since the types used in practical examples (`nat`, `bool`, `nat -> nat`, etc.) have well-known inhabitants (as `0`, `true`, `[x:nat]0`, etc.) and we do not need a tactic theorem prover to construct these objects. But these goals may be generated as subgoals by the tactic theorem prover itself. For instance when we want to prove the proposition `(Lower_eq 0 (S (S 0)))` using the axiom:

```
Trans:(x:nat)(y:nat)(z:nat)(Lower_eq x y) -> (Lower_eq y z) ->
                                                (Lower_eq x z)
```

then the term used substituted to y in this proof cannot be infered by the matching of `(Lower_eq 0 (S (S 0)))` with `(Lower_eq x z)`, and `nat` appears as a subgoal. In these cases the "proof" of this subgoal appears in the other generated subgoals `(Lower_eq a y)` and `(Lower_eq y c)`. These dependencies are represented by `Meta(`n`)` where n is a number. In our cases the subgoals are `(Lower_eq a (Meta(1)))` and `(Lower_eq (Meta(1)) c)`.

In this case, the "philosophy" of tactics is lost, since subgoals are no longer mutually independent. If such subgoals with "meta"-variables occur, the only safe way to solve them is by using the command `Instantiate` which is not a tactic and which applies to all subgoals at the same time as opposed to only applying to one particular goal as a tactic does (cf the description of `Instantiate` for further details).

To preserve the philosophy of tactics it is recommended to use `Apply` *term* `with` $term_1$ ... $term_n$ which is described below, and which avoids the generation of interdependent subgoals.

For instance :

```
Coq < Goal (<nat> Ex ([x:nat](<nat>x=0))).

Coq < Apply ex_intro.
2 subgoals
  nat
subgoal 2 is:
  <nat>Meta(3)=0

Coq < Instantiate 0.
1 subgoal
  <nat>0=0
```

and similarly :

```
Coq < Undo.
2 subgoals
  nat
subgoal 2 is:
  <nat>Meta(3)=0

Coq < 2:Instantiate (refl_equal nat 0).
Goal proved!
```

while the pure tactics do not propagate an instantiation, leading possibly to unsuccessful proof attempts :

```
Coq < Undo.
2 subgoals
  nat
subgoal 2 is:
  <nat>Meta(3)=0

Coq < Exact (S 0).
1 subgoal
  <nat>Meta(3)=0
```

The presence of meta-variables forbids a goal from being solved except by means of the command `Instantiate`:

```
Coq < Instantiate (refl_equal nat 0).
Goal proved!

Coq < Show Proof.
(ex_intro nat [x:nat](<nat>x=0) (S 0) (refl_equal nat 0))
```

In this case, the Theorem Prover is not careful about correctness. However, saving will not be possible, since `Save` verifies the proof before saving it.

- `Apply` *term* `with` $term_1 \ldots term_n$

  When variables from a theorem are not deducible by matching with the goal, this tactic permits them to be given explicitly.

  term must be a theorem or an axiom (or an hypothesis). Let *statement* be its statement.

  The number n of arguments of this tactic must be exactly the number of variables really used in *statement* but not present in its conclusion.

  Moreover, the terms must be given in the order of their quantification in *statement.*

  For instance with the goal:

```
Coq < Goal (<nat> Ex ([x:nat](<nat>x=0))).
```

instead of using the tactic `Apply ex_intro` and then the command `Instantiate 0` we can use the tactic `Apply ex_intro with 0` which will also produce the subgoal `<nat>0=0`.

Unlike `Apply`, this tactic does not unfold the statement if it is just an atomic proposition.

- `Apply` *term* `with` $name_1$`:=`$term_1$ ... $name_n$`:=`$term_n$

  This is a variant of `Apply` *term* `with`, where variables are designed by their name (as it appears on the screen) instead of beeing given in the order of their quantification. In this case, the variables may be any variable occuring in the statement of the theorem, not necessary all non deductible one, and not necessary all non dependent one in the conclusion, at the opposite of the previous syntax of `Apply` *term* `with`.

  Example :

```
1 subgoal
  <nat>z=0
  ============================
    z : nat
    H : (x:nat)(y:nat)(P:nat->Prop)(P x)->(P y)

Coq < Apply H with y:=z x:=0.
1 subgoal
  <nat>0=0
  ============================
    z : nat
    H : (x:nat)(y:nat)(P:nat->Prop)(P x)->(P y)
```

- `Cut` *statement*

  When an unproved statement *statement* must be used to solve a subgoal, this tactic generates a new subgoal of statement *statement*, and replaces the current subgoal by the same subgoal with the additional premise *statement*.

  Example :

```
Coq < Goal (<nat>(S 0)=(S (S 0))).

Coq < Cut (<nat>0=(S 0)).
2 subgoals
  (<nat>0=(S 0))->(<nat>(S 0)=(S (S 0)))
subgoal 2 is:
  <nat>0=(S 0)
```

- `Generalize` *name*

  When *name* is the name of a variable of the context from which depends the current subgoal, it replaces it by the same subgoal but quantified by the variable of name *name*.

  Example :

```
Coq < Goal (x,y:nat)(<nat>x=y).

Coq < Intros.
1 subgoal
  <nat>x=y
  ============================
    y : nat
    x : nat

Coq < Generalize x.
1 subgoal
  (x0:nat)(<nat>x0=y)
  ============================
    y : nat
    x : nat
```

Remark : This tactic also works for hypotheses of the context.

Special tactics for predefined theorems

- **Left**

  This tactic proceeds to prove a disjunction by left introduction.

- **Right**

  This tactic proceeds to prove a disjunction by right introduction.

- **Split**

  This tactic proceeds to prove a conjunction by splitting it in proofs of its two components.

- **Exists** *term*

  This tactic provides a way to prove an existential statement by giving the witness element.

  It works for all predefined existential quantification of Coq.

- **Reflexivity**

  This tactic proves a statement of the form **<T>x=x** or **<T>x==x**.

- **Symmetry**

  This tactic reduces a goal **<T>x=y** to the goal **<T>y=x** (and the same for **<T>x==y**).

- **Transitivity** *term*

  This tactic reduces a goal **<T>x=y** to the goals **<T>x=***term* and **<T>***term*=y (and the same for **<T>x==y**).

Elimination tactics

- `Elim` *term*

  This tactic is the basic elimination tactic. *term* must prove or assume a statement *statement*. The conclusion *ind* of this statement must be an inductively defined term. `Elim` tries to apply the elimination theorem of *ind* on the goal. It fails if this is not possible or not allowed. (see the section about inductive definitions for more on the automatically generated elimination theorems).

  When the elimination theorem is dependent, *statement* must not be a product. Otherwise, it generates as many subgoals as the number of premises of *statement*, as `Apply` does.

  `Elim` first tries a first-order matching with the goal, (allowing the use of `Pattern` before applying this tactic). If this matching fails, it tries second order matching. If the conclusion of the elimination theorem of *ind* is `(P t1 ... tn)` (see the section about inductive definitions for further explanation), then it abstracts all the occurrences of `t1`,... ,`tn` in the goal, in order to put the goal in the form `(Q t1 ... tn)` do first order matching. In this case, it works as if `Elim` does `Pattern t1 ... tn` before the matching.

- `Elim` *term* `with` $term_1 \ldots term_n$

  This tactic allows the user to explicitly give the quantified variables of the statement stated by *term*, when they are not deducible by matching with the goal.

- `ElimType` *statement*

  When an unproved statement *statement* must be used to be eliminated, this tactic generates a new subgoal of statement *statement* and *statement* is eliminated on the subgoal to which `ElimType` applies.

  It has the same effect as `Cut` *statement* `; Intro Hyp; Elim Hyp` except that it does not make visible the hypothesis `Hyp` in the local context.

- `Induction` *name*

  This is a short name for `Intros until` *name* `; Pattern` *name* `; Elim` *name*.

- `Induction` *n*

  does the same but induction is done on the *n*th non dependent premiss of the goal.

Dealing with equality Predicates of equality are inductively predefined. Here follow special tactics dealing with them.

- `Rewrite <-` *term*

  If *term* proves a statement which conclusion is an equality `<T>x=y` (or `<T>x==y`) then `Rewrite` works as `Elim` *term* does, i.e. it replaces `y` by `x` everywhere in the goal.

- `Rewrite ->` *term*

  Similar to the previous one, but rewrites in a left to right manner. The indication `->` is optional in this case.

- `Replace` $term_1$ `with` $term_2$

  This tactic replaces the occurrences of $term_1$ in the goal by $term_2$. It then generates a new subgoal `<`T`>`$term_1$`=`$term_2$, where T is the common type of $term_1$ and $term_2$. This subgoal is solved, if possible, by `Assumption` or by symmetry of equality then `Assumption`. Otherwise it is left as an unproven subgoal.

  It works for both predefined equality on `Set` and predefined equality on `Type`.

  Remark : If the subgoal is of the form `(P` $term_1$`)`, then only this occurrence of $term_1$ will be replaced, even if there are other occurrences. This allows the user to use `Pattern` before applying this tactic (see `Elim`).

The special case of `False` `False` is an inductive proposition. Here follows a special tactic dealing with it.

- `Absurd` *statement*

  This is a tactic which proves the goal by contradiction, i.e. the goal is proved by elimination of `False`, and `False` itself comes from proofs of both *statement* and `~`*statement*. Therefore this tactic generates the subgoals *statement* and `~`*statement*.

  It has the same effect as `ElimType False; Cut` *term*

Convertibility tactics

There are three forms of convertibility between goals :

Convertibility w.r.t. expansion of a constant.

Convertibility w.r.t. instantiation of the parameters of a function by its arguments (for instance `([x:nat](S x) O)` is convertible to `(S O)`).

Convertibility w.r.t. the recursion schemes associated to inductive definitions (for instance, if `pred` is the predecessor defined in the vernacular file `Nat.v`, then `(pred (S n))` is convertible to `n`).

All the tactics described here are concerned with one or more of these forms of convertibility.

- `Unfold` $n_1^1 \ldots n_1^{p_1}$ $name_1 \ldots n_q^1 \ldots n_q^{p_q}$ $name_q$

  This tactic replaces successively the occurrences $n_1^i \ldots n_p^i$ in the goal of each constant $name_i$ by its definition.

  By default, if no occurrence is given, then all the occurrences will be unfolded.

  For instance (recall that `~` is the way we write `not`, that `refl` stands for "to be a reflexive relation" and `gt` stands for "to be greater than") :

```
Coq < Goal (~~~(refl nat gt))-> ~(refl nat gt).

Coq < Unfold 2 4 not gt 1 refl.
1 subgoal
 (~(~(x:nat)(~(le x x)))->False)->(refl nat [n:nat][m:nat](~(le n m)))->False
```

But be careful in which order the constants are given :

```
Coq < Undo.
1 subgoal
  (~~~(refl nat gt))->(~(refl nat gt))

Coq < Unfold gt 2 4 not 1 refl.
1 subgoal
  (~(~(x:nat)(le x x)->False)->False)->(~(refl nat [n:nat][m:nat](~(le n m))))
```

- `Change` *statement*

  This tactic replaces the goal by *statement*. *statement* and the goal have to be convertible following the β-rule, δ-rule and elimination rules for inductive terms, it fails otherwise.

- `Red`

  This tactic replaces the head constant of the conclusion of the goal by its definition.

- `Simpl`

  This tactic simplifies the goal by instantiating the formal parameters of functions by the corresponding arguments and by applying the recursion schemes for inductive constants as much as possible. It never fails.

  Example :

```
Coq < Goal (n:nat)([p:nat](<nat>n=(pred (S p))) n).

Coq < Simpl.
1 subgoal
  (n:nat)(<nat>n=n)
```

- `Hnf`

  This tactic reduces the head of the goal (by expansion of the head constant, or by simplification if the goal is of the form `([x:A]P y)`) until it becomes either universally quantified, an implication, or the application of a predicate variable to its arguments.

- `Pattern` $n_1^1 \ldots n_1^{p_1}$ $term_1$ $\ldots$ $n_q^1 \ldots n_q^{p_q}$ $term_q$

  This tactic abstracts $term_1 \ldots term_q$ in the goal. The numbers $n_1^1 \ldots n_1^{p_1} \ldots n_q^1 \ldots n_q^{p_q}$ are given to specify exactly which occurrences of $term_1 \ldots term_q$ have to be taken into account.

  By default, if no occurrence is given, then all the occurrences are taken into account.

  For instance:

```
Coq < Variable x:nat.
x is assumed
```

```
Coq < Goal (<nat>x=x)->(<nat>x=x).

Coq < Pattern 1 3 x 0 x 4 x.
1 subgoal
  ([n:nat][n0:nat][n1:nat](<nat>n=x)->(<nat>n=n1) x x x)
```

This tactic is helpful for putting a subgoal into the form $(Q\ t_1\ \ldots t_n)$. It is useful when a theorem or an axiom is applied whose statement has a conclusion of the form $(P\ x_1\ \ldots x_n)$, where P and x_i are all variables. This makes the matching done by Apply between the statement and the goal trivial.

Context convertibility tactics

- **Unfold** $n_1^1 \ldots n_1^{p_1}$ $term_1 \ldots n_q^1 \ldots n_q^{p_q}$ $term_q$ *name* **in** *namehyp*

  * is allowed instead of a list of occurrences.

- **Change** *statement* **in** *namehyp*

- **Red in** *namehyp*

- **Simpl in** *namehyp*

- **Hnf in** *namehyp*

These tactics have similar behavior as the one described in the previous section, but they apply to the hypothesis of name *namehyp* instead of applying to the current subgoal.

Remark : Be careful not to use in as a name : this is a reserved keyword !

Automatic search tactics

These tactics use a hint list built by means of the commands **Hint** or **Immediate**. Hints are theorems or axioms which these tactics try to apply. Each hint has a priority, usually the number of subgoals it generates.

- **Auto**

  This tactic tries to apply successively, by order of priority, the theorems or axioms of the hint list. If successful, it recursively tries to apply the hints once again to the generated subgoals until a complete proof is found. It has a maximal search depth of 5.

  It leaves the goal unchanged if the search fails.

- **Auto** n

  This tactic is the same as the previous one except that the depth of the search is forced to n.

- **Trivial**

  This tactic tries to apply the hints of priority 0, i.e. the theorems generating no subgoals. It also tries to solve by **Assumption**.

  It leaves the goal unchanged if no proof is found.

In Verbose mode, the `Auto` and `Trivial` tactics provide a trace mechanism. They print a message explaining the sequence of elementary tactics that were used. An elementary tactics corresponds to what was stored in the search table using the `Hint` or `Immediate` commands.

The message start with `Use : .` It continues with what we call a *printed message* which is one of the following sentences: "`Idtac`" if the auto tactics did not succeed and consequently did not change the goal. "*tactic*" if it corresponds to the elementary tactic used. "*tactic*; *printed messages list*" if the `Auto` tactics combined several levels. A *printed messages list* is either a *printed message* or has the following syntax:

"`[`*printed message 1*`|`...`|`*printed message n*`]`"

It means that the tactic *tactic* generates n subgoals, the ith subgoal was solved using the tactic corresponding to the message "*printed message i*".

If you use a compound tactic "*tactic*`; Auto.`", the tactic `Auto` will be applied to each subgoal generated by "*tactic*". You will get a message for each application of the tactic.

Handling the hint list

- `Hint` $name_1 \ldots name_n$

  This command adds tactics `Apply` $name_1$, ..., `Apply` $name_n$ to the hint list. A priority is assigned to each hint, usually the number of subgoals it generates.

  Beware: `Hint` declarations may be lost at the closing of a section. Redeclare them if they need to.

- `Immediate` *name*

  This command adds the theorem or axiom *name* to the hint list, with the indication to use it only if the subgoals it generates are immediately provable by an hypothesis or a theorem that does not generate subgoals.

  Be careful: `Immediate` declarations may be lost at the closing of a section. Redeclare them if need be.

- `Hint Unfold` $name_1 \ldots name_n$

  This adds the hints $name_1 \ldots name_n$ to the hint list. Here, the hints are not for resolution with a theorem or an axiom, but used to unfold all the occurrences of the constants named $name_1 \ldots name_n$.

  The priority of these hints is arbitrarily fixed to 4.

- `Erase` $name_1 \ldots name_n$

  This removes the theorems or axioms $name_1 \ldots name_n$ from the hint list.

- `Print Hint`

  This displays the list of theorems used for the automatic search tactics, sorted by their head constant. The associated priority is also displayed. If a theorem has been entered several times in the list, it appears only once.

For instance, typing `Print Hint` just after entering the system will give the list of theorems declared as hints in the initial context of Coq :

```
Coq < Print Hint.
For gt -> gt_Sn_n,0 gt_Sn_0,0 le_S_gt,1 gt_n_S,1
For and -> conj,2
For sum -> inr,1 inl,1
For not -> n_Sn,0 0_S,0 le_Sn_n,0 le_Sn_0,0
For eq -> refl_equal,0 eq_add_S,1 eq_S,1 sym_equal,1
For T -> I,0
For le -> le_pred_n,0 le_0_n,0 le_n_Sn,0 le_n,0 le_trans_S,1 le_S_n,1 le_n_S,1
 le_S,1
For sumor -> inright,1 inleft,1
For or -> or_intror,1 or_introl,1
For sumbool -> right,1 left,1
For PROD -> PAIR,2
For prod -> pair,2
```

Tacticals : constructors of tactics

- *tactic*$_1$; *tactic*$_2$

  This construction applies *tactic*$_2$ to all the subgoals generated by *tactic*$_1$. It associates to the left.

- *tactic*$_1$ `Orelse` *tactic*$_2$

  This composed tactic tries to apply *tactic*$_1$ and, in case of failure, applies *tactic*$_2$ without catching a possible failure. It associates to the left.

- `Try` *tactic*

  This tactical catches a possible failure of *tactic*.

- `Repeat` *tactic*

  This tactical repeats the tactic *tactic* as long as it does not fail.

- `Do` *n tactic*

  This allows to repeat *n* times the tactic *tactic*. It fails if it is not possible to repeat *tactic n* times without failure.

Advanced user's tactics

We just inform the reader that new tactics may be written in CAML. This facility is not documented here. We refer the interested (and adventurous) user to the study of Coq CAML source files.

- `By` *caml_tactic*

  This syntax allows the use of tactics written in CAML. Such an expression behaves like a built-in tactic and may be combined with others tactics by means of tacticals.

1.5.3 Commands of the Theorem Prover

1.5.4 Handling the goal environment

- `Goal` *term*

  This declares a goal to prove to the Tactics Theorem Prover. As long as this goal is not either proved and saved or aborted, all `Reset` commands are disabled.

- *n*: *tactic*

  This applies the tactic *tactic* to the *n*th subgoal. If the tactic fails, an error message is displayed. Otherwise, the newly generated and remaining subgoals are displayed.

  tactic may be any tactic built from tacticals and basic tactics.

  n: `Instantiate` *term* is also allowed, but we remind that `Instantiate` is not a tactic : it can only be used alone, and not in composition with other tactics by means of tacticals.

  By default, if *n*: is omitted, it is the first subgoal that is considered.

- `Show`

  This displays the principal subgoal with its local context and the statement of the secondary subgoals. Subgoals are numbered from the principal one.

- `Show` *n*

  This displays the *n*th subgoal with its local context.

- `Undo`

  This causes the Theorem Prover to return to the previous step of the proof. The Theorem Prover keeps a memory of at most 12 states. Therefore, it is not possible to undo more than 12 times.

  Axioms, theorems, constants and hints declarations are not concerned with `Undo`.

  Remark that sometimes a tactic (as for instance `Auto`) may do nothing and keep the set of goals unchanged. Undoing such a "transformation" will also keep the set of goals unchanged.

- `Undo` *n*

  This does the same as `Undo`, but *n* times.

- `Restart`

  This restarts the proof at the beginning. Axioms, theorems, constants and hints declarations introduced since the beginning of the proof search session are kept.

- `Abort`

  This aborts the current goal. Axioms, theorems, constants and hints declarations introduced since the beginning of the proof search session are kept. One must use the command `Reset` below in order to erase them.

Declaration of a completely proven theorem in the global context

- `Save Theorem` *name*
- `Save` *name*

  These two commands (which are synonymous) save a completely proved theorem in the context of the constructive engine.

  This has the same effect as the vernacular command :

  `Theorem` *name*

  `Statement` *statement*

  `Proof` *proof*

  where *statement* was the term given with `Goal` and *proof* is the term constructed by the Theorem Prover.

  Moreover, they clear away the current goal of the environment of the Theorem Prover.

  All `Hint` and `Immediate` commands and all declarations introduced during the session, if not forgotten by a possible `Restart`, are kept in the current context of Coq.

- `Save Remark` *name*

  This command works as the previous ones but the scope of the theorem is restricted to the current section.

Silent mode

The silent mode turns off displaying of the state after a tactic has been applied successfully. It is very useful for speeding up the loading of vernacular files containing tactics.

- `Silent`

  This command turns off the displaying.

- `Verbose`

  This command turns the normal displaying mode on.

File recording of a interactive session

The following two commands allow the recording of an interactive session in a file.

- `Open` *filename*

  This command opens the recording mode and all the axioms, theorems, constants, goal commands, save commands, and hint declarations that you give from this time on will be appended to the file named `filename.v` in the current directory. If this file does not exist, it is created. If you want to address a file with an explicit absolute or relative path, use the CAML command `open_vernacular`.

  The spacing or blank lines that you type are kept, but the comments `(* ... *)` are removed.

If the goal is not aborted, the axioms, theorems or constants declared during a proof search session will be written just before the goal declaration. This is consistent with the state of the global context after the proof is completed.

Similarly for `Hint` and `Immediate` declarations. But in the case of `Abort`, these commands will not be recorded when no axioms, theorems, or constants declarations were given between them and the goal declaration. This is a (subtle) anomaly.

- `Close`

  This closes the recording mode.

1.6 Miscellaneous commands of the Coq system

Printing the state of the context

The current context can be printed with the following commands :

- `Print`

  This prints all the axioms or parameters and then all the theorems and definitions declared after the last axiom or parameter. If there is no axiom or parameter in the context, then all declarations are displayed. Proofs of theorems and bodies of definitions are displayed as well as theorem statements and definition types.

- `Info` *name* : is equivalent to `Print` *name* except if *name* is an inductive definition in which case it displays also its constructors and the allowed eliminations.

- `Print All`

  This prints the entire context. Proofs of theorems and bodies of definitions are not displayed. Only axiom or theorem statements and parameter or definition types are displayed.

- `Print Section` *name*

  This prints the entire context like `Print All`, but only from the beginning of the open section *name*.

- `Print` *name*

  This prints the body of the constant *name* with its type (which may be the proof of the theorem *name* with its statement). If *name* is a parameter, its type is printed. (If *name* is an axiom, its statement is printed.)

- `Check` *name*

  This prints the statement of the axiom or theorem *name* (or the type of the parameter or definition *name*).

- `Search` *name*

  This prints all the theorems with head constant *name*.

- `Inspect` *num*

  Like `Print All` this prints the context but only includes the *num* last items.

Reseting the context

These commands are used to reset the system to a previous state, in order to correct mistakes, for instance.

- `Reset` *name*

  This resets the system to the state it was in before writing the item *name*.

- `Reset After` *name*

  This resets the system to the state it was in after writing *name*.

- `Reset Section` *name*

  This resets the system to the state it was in before opening the section *name*.

- `Reset Initial`

  This resets the system to its initial state, (i.e. the initial context corresponding to the standard prelude.)

Term evaluation

- `Eval` *term*

  This command evaluates *term* by instantiating the formal parameters of functional subterms by the corresponding arguments.

  For instance :

```
Coq < Eval ([x:nat](pred x) (S 0)).
      = (pred (S 0))
      : nat.
```

- `Compute` *term*

  This command evaluates *term* by instantiating the formal parameters of functional subterms by the corresponding arguments and by applying recursion schemes.

  For instance :

```
Coq < Compute ([x:nat](pred x) (S 0)).
      = 0
      : nat.
```

Loading a vernacular file

- `Load` *filename*

  This loads the vernacular file `filename.v` in the current working directory. A system error occurs if the file does not exist. Use the CAML command `cd"path";;` to change the current directory (or `#(cd"path")` directly from the vernacular top-level loop, or else more directly `#(V"path/filename")` to load a vernacular file not in the current directory from the vernacular top-level loop).

Exiting the vernacular top-level of Coq system

- `Drop` This returns to the CAML top-level. The CAML prompt is :

  ```
  ml #
  ```

When we later go back to Coq by the command `coq();;` everything previously developed is recovered, i.e. the state of the proof-engine is not affected by the changes performed in the caml top-level.

1.7 The Coq Interface

A first prototype of a multi-window interface for the X-window system is included in Version 5.6 of the Coq system. The current implementation provides a flexible mechanism for goal-directed proof development. Most of the operations available correspond directly to the application of tactics or to other Coq commands that operate on goals.

In describing interface operations, we will always mention the corresponding Coq command. For a complete description of any command, we refer the reader to the corresponding section elsewhere in this document.

1.7.1 Starting Up The Interface

The interface works on both black and white and color screens. The default is color. If you have a black and white screen, begin by typing the following to the `ml` top level.

```
ml # COLOR := false;;
```

This command also works on color screens and will give the black and white version of the interface.

To start up the interface, type:

```
ml # X();;
```

A function `goX` is also provided in order to specify a screen on which the interface should appear. This function takes a string as argument corresponding to the machine name. For example,

```
ml # goX "pouilly:0"
```

causes the interface windows to appear on the screen of the machine named pouilly.

By typing either of these commands, two windows will appear on the screen: the top level or main window, and a context window. These windows (as well as all others that may appear later) can be moved, made smaller or larger, iconified, etc. according to the specifications of the X-window manager.

1.7.2 The Context Window

The context window is originally empty. It contains three buttons corresponding to three kinds of operations: Print, Inspect, and Reset. By clicking and holding down the mouse button, a menu will appear with the list of operations. To apply an operation, while holding down the mouse button, move the mouse to the desired operation and release the button. The operations are listed and

some description is given below. Those that are not explained correspond directly to the command of the same name. Many operations take additional arguments (indicated by an underscore in the menu entry). An input window will appear, prompting for these arguments. Click on `OK` or type return to end the input and execute the operation.

The `Print` Menu.

```
Print
Print _
Print Hint
Print All
Print Section _
Check _
Search _
```

The `Print` button applies the `Print` command, while the `Print _` button opens the input window allowing the user to enter the name of an item in the context. The `Print All` operation is different from the others in that the context window will be cleared before the `Print All` command is executed displaying the entire context.

The `Inspect` Menu. This menu contains entries for the `Inspect` command with various specified arguments. It also allows the user to enter an arbitrary argument.

The `Reset` Menu.

```
Reset _
Reset After _
Reset Section _
Reset Initial
```

Scrolling Text. In addition, the context window contains a scroll bar. Scroll backward one line at a time by placing the mouse anywhere in the scroll region and clicking on the left button, and forward one line at a time by clicking on the right button. To move the text to a particular location, position the mouse at the desired place in the scroll region and click on the middle button. Hold this button down and move the mouse up and down for finer control. Several other text windows of the interface contain scroll bars which all work similarly.

1.7.3 The Main Window

The top level window contains several buttons and menus corresponding to various system commands, and an output window where output from the system is printed.

The `Goal` Button. The `Goal` button corresponds to the `Goal` command and prompts the user for a goal. The windows for tactic-driven proof synthesis are then opened. See Section 1.7.4 for a description of goal-directed proof synthesis.

The `TacProver` Button. This button opens the windows for tactic-driven proof synthesis on the current goal or goals. If there is more than one subgoal, one of them is chosen as the current goal. The user can freely change the current goal by clicking on the appropriate goal in the subgoal window. See Section 1.7.4.

The window interface is independent of the tty interface in the sense that it is possible to go back and forth between the two. For example, a user may begin proof synthesis using the window interface, and then continue using the tty interface. If the user then returns to the window interface, the list of subgoals will be those that remained at the end of the tty session.

The `Open` and `Close` Buttons. The `Open` button corresponds to the `Open` command and will prompt the user for the file name. The `Close` button closes the currently opened file, if there is one.

The `Hint` Menu. The following operations for adding and removing items to be used by the automatic tactics are included in this menu.

```
Hint _
Immediate _
Hint Unfold _
Erase _
Print Hint
```

The `Vernac` Button. The main window is incomplete. The intent is to eventually include all commands that do not operate on a goal in this window. For example, a flexible way to introduce definitions, hypotheses, etc., should be provided at this level. In the meantime, a button `Vernac` is provided, which prompts the user for input corresponding to an arbitrary command.

The `Abort` Button. This button corresponds to the `Abort` command which aborts the current goal.

The `Quit` Button. The `Quit` button closes all windows and returns to the `ml` top level. If the window interface is started up again, it will return to the state it was in just before exiting.

1.7.4 The Proof Synthesis Windows

Two windows will appear when proof synthesis is started by either the `Goal` or `TacProver` buttons in the main window. The first is the Current Goal Window which displays the current goal along with its local hypotheses. It also contains buttons for applying tactics to the current goal and other operations. Five of these buttons contain menus for the different categories of tactics: Introduction, Resolution, Induction, Convertibility, and Automatic/Exact. The tactics are listed and some description is given below. A sixth button is provided for entering more complex tactic expressions.

The second window is the Subgoal Window which contains the list of all subgoals. The current subgoal is marked by `X`. To change the current subgoal, simply click on the button marked with the number of the desired subgoal. This goal and its local hypotheses will be displayed in the Current

Goal Window. All tactics will be applied to this newly chosen goal until it is completed or until the user chooses to replace it by another subgoal from the subgoal list. In the case when a tactic is applied to the current goal resulting in multiple subgoals, the user must choose which one will be the next current goal before proof synthesis can proceed.

Introduction Tactics.

```
Intro
Intro _
Intros
Intros _
Intros until _
```

The `Intro` button applies the `Intro` command, while the `Intros` button applies `Intros`. The corresponding `Intro _` and `Intros _` prompt the user for the names of the hypotheses. Similarly, `Intros until` prompts the user for its argument.

Resolution Tactics.

```
Apply _
Cut _
Generalize _
Left
Right
Split
Exists _
Reflexivity
Symmetry
Transitivity _
```

The input to tactic `Apply` is a term, or a term followed by `with` followed by additional arguments. The input to `Exists` and `Transitivity` are terms. The input to `Replace` is a term followed by `with` followed by another term.

Induction Tactics.

```
Elim _
ElimType _
Induction _
Rewrite -> _
Rewrite <- _
Replace _
Absurd _
```

`Elim` is similar to `Apply` above. The user is prompted for input. The first argument can be followed by `with` and additional input. The input to `Induction` can be either a number or variable name.

Convertibility Tactics.

```
Unfold _
Change _
Red
Simpl
Hnf
Pattern _
```

The `Unfold` tactic prompts for input, which like `Apply` and `Elim` can take several forms: in this case, an optional occurrence number, followed by a constant, optionally followed by `in` and the name of a hypothesis.

Automatic/Exact Tactics.

```
Auto
Auto _
Trivial
Exact _
Assumption
Instantiate _
```

Tactic Expressions. This button provides a menu of high level operations that allow tactics to be composed in various ways.

```
_ Orelse _
Try _
Repeat _
Do _
Other
```

Selecting any of these buttons will cause an input window to appear. Selecting `Do`, `Try`, or `Repeat` will cause the corresponding initial string to appear in the input window, while `Other` will open the input window with no initial string. Choose the desired tactics one at a time by selecting them from the five tactic menus. Enter any required arguments by typing them directly in this input window. Use the button marked ; to enter a semi-colon between tactics. Alternatively, a semi-colon as well as any other input can be typed in directly by hand. Only click on `Orelse` after the first argument has been entered. This can be done, for example, by clicking on `Other`, entering the first argument, and then clicking on `Orelse` before entering the second argument. Finally, click on `OK` or type return to execute the tactic expression.

For example, it is possible to apply the expression:

```
Do 2 Intro ; Repeat Split ; Auto.
```

by the following series of operations.

- Select `Do` from the Tactic Expression menu. Note that in addition to `Do`, the initial string appearing in the window will contain the goal number corresponding to the current goal (calculated automatically) followed by a colon.

- Enter 2 from the keyboard.
- Select `Intro` from the menu of Introduction tactics.
- Click on ; in the input window.
- Select `Repeat` from the Tactic Expression menu.
- Select `Split` from the menu of Resolution tactics.
- Again, click on ; in the input window.
- Select `Auto` from menu of Automatic tactics.
- Click on `OK` in the input window.

Alternatively, the entire command can be entered by selecting `Other` to open the input window and then typing this expression by hand, or entering it using cut and paste. (See Section 1.7.5.)

The `Other` button is also useful, for example, to apply expressions such as:

```
Split ; Auto.
```

where an expression containing several tactics begins with a tactic that takes no arguments. In this case, clicking on `Split` without first opening the input window would apply `Split` directly. Here, as is always the case when the input window is open, the complete expression will not be applied until the user selects `OK` or types return.

The `Undo` Button. The `Undo` button corresponds to the `Undo` command. It will undo the proof to the point before the last tactic expression was applied.

It is important to note that this undo facility is somewhat limited. It is linear with respect to time. Thus for example, if a tactic is applied to one subgoal, and then another is applied to a different and independent subgoal, there is no way to undo the operation on the former without first undoing the operation on the latter, even though the two subgoals may be on independent branches of the search tree.

The `Restart` Button. This button corresponds to the `Restart` command which restarts the current goal.

1.7.5 Miscellaneous Operations

Cut and Paste. It is possible to cut and paste text from arbitrary windows to the Coq interface, and from the Coq interface to arbitrary windows. To cut text from any one of the interface windows, position the mouse at the beginning of the text and click on the left mouse button. Release the button, go to the end of the text and click on the right button. Alternatively, click with the left button at the beginning of the text, hold the button down and move to the end of the text, then release. Note: the text will not be highlighted. To paste the current contents of the cut buffer to an input window, click on the middle button.

Using this facility, input to tactics that take arguments can be entered by cutting text from the Context Window, an emacs window, or any other window, and pasting this text in the tactic input window.

It is also possible to use this facility to execute existing example files using the Coq interface. To execute tactic expressions from an emacs file, it suffices to open the tactic expression window by clicking on `Tactic Expression` and selecting `Other`, to cut the text from the emacs file, and to paste it to the input window.

Known limitation: when cutting text from a Coq window, it is not possible to exceed one line. On the other hand, if a section of text containing more than one line is cut from another window (such as emacs), the entire text can be pasted to a Coq input window.

1.8 More on inductive definitions

In part 1.1.8 we saw how to use inductive types to get a direct representation of natural numbers, lists or products. Inductive definitions also allow an internal representation of other notions like inductive predicates and even logical connectives.

This part contains examples of inductive definitions. It also contains more technical information on these definitions like the general syntax of an inductive declaration. We shall also present some difficulties that can arise when using inductive definitions.

1.8.1 Inductive definitions of predicates and relations

During the development of a theory, it is useful to introduce notions like n *is even* or n *is less than* m. Because they often correspond to primitive recursive predicates it is possible to represent them as `<nat>(C n)=1` with `C` the characteristic function of the predicate. But such definitions, if theoretically sufficient, do not lead to very simple proofs. In Coq, the general mechanism of inductive definitions can also be used for the definition of inductive predicates leading to very elegant proofs.

Order on the natural numbers

The inductive definition of the order on natural numbers corresponds to the mathematical definition: "$\leq$ is the smallest relation such that $0 \leq n$ and if $n \leq m$ then $(S\ n) \leq (S\ m)$". When we introduce such a definition, we expect two properties. First the predicate satisfies the specification clauses :

$$\forall n.0 \leq n \qquad \forall n.\forall m.\ n \leq m \Rightarrow (S\ n) \leq (S\ m)$$

Secondly it is the smallest one to satisfy these properties. This means that if R is such that:

$$\forall n.R(0,n) \qquad \forall n.\forall m.R(n,m) \Rightarrow R((S\ n),(S\ m))$$

then $n \leq m \Rightarrow R(n,m)$.

The effect of the following inductive definition will be similar:

```
Inductive Definition LE : nat->nat->Prop
 = LE_0 : (n:nat)(LE 0 n)
 | LE_SS : (n,m:nat)(LE n m)->(LE (S n) (S m)).
```

It declares in the environment the two terms LE_0 and LE_SS whose types are respectively (n:nat)(LE 0 n) and (n,m:nat)(LE n m)->(LE (S n) (S m)). It also introduces a constant LE_ind corresponding to the minimality property (or elimination theorem) whose type is:

```
(R:nat->nat->Prop)
        ((n:nat)(R 0 n))
      ->((n,m:nat)(LE n m)->(R n m)->(R (S n) (S m)))
      ->(n,m:nat)(LE n m)->(R n m)
```

This presentation looks slightly stronger than our informal presentation because we only need to prove
(n,m:nat)(LE n m)->(R n m)->(R (S n) (S m)) instead of the stronger condition:
(n,m:nat)(R n m)->(R (S n) (S m)) but the two schemes actually may be shown to be equivalent.

In general the user will not directly use the combinator LE_ind but the elimination tactic. For example assume that we want to prove:

```
Goal (n,m:nat)(LE n m)->(LE n (S m)).
```

Using the introduction tactic we get an hypothesis H of type (LE n m), while the current goal becomes (LE n (S m)). We may use the elimination tactic Elim H. It will first infer a relation R to which it will apply the minimality principle. In this case R will be [n,m:nat](LE n (S m)). Two subgoals are generated corresponding to the two clauses to verify, namely:

```
(n:nat)(LE 0 (S n))
(n,m:nat)(LE n m)->(LE n (S m))->(LE (S n) (S (S m)))
```

Both are solved trivially using introduction and applying respectively le_0 and le_SS.

1.8.2 Inductive definitions of logical structures

The inductive definition of a type in a programming language or a mathematical predicate is quite natural. The inductive definition of logical structures like conjunction, disjunction or existential quantification, however, may seem strange. Actually, this notion may be related to the analogy between proofs and programs. Because we do not want to introduce this notion now, let us just make the following remark. In natural deduction, each logical connective is presented with introduction and elimination rules. The introduction rules are analogous to the constructors of an inductive type and the elimination rule is analogous to the destructing operation.

In part 1.2.2 we saw how minimal logic was represented in Coq. This logic is enough to axiomatically present the other logical structures. In this part we shall instead show how to define an internal representation for them using inductive definitions.

To say that a logical structure may be defined as an inductive type is simply to remark that the form of the elimination rule can be derived from the specification of the introduction rules.

Propositional Calculus

The following definitions are part of the *Prelude* file which is loaded when starting the system. So you may completely ignore the internal coding of the various logical connectives and just use them via the adequate tactics of introduction (Left, Right, Split) and elimination (Elim).

Conjunction is written as an inductive type with one introduction rule.

```
Inductive Connective and [A,B:Prop] = conj : A->B->(A/\B).
```

There is only one elimination rule which says that in order to prove C from (A/\B) it is sufficient to prove A->B->C. From this scheme we may retrieve the two usual elimination rules (A/\B)->A and (A/\B)->B.

Another example is the definition of the disjunction of two propositions. It has two introduction rules:

```
Inductive Connective or [A,B:Prop]
    = or_introl : A -> (A\/B) | or_intror : B -> (A\/B).
```

The elimination scheme states that in order to prove C from (A\/B) it is sufficient to prove A->C and B->C.

The absurd proposition has no introduction rule. It is represented by an inductive definition without constructors:

```
Inductive Proposition False = .
```

The corresponding elimination rule says that from False we may prove any proposition C.

The tautological proposition has one introduction rule:

```
Inductive Proposition True = I : True.
```

The corresponding elimination rule is the identity propositional function.

First-order connectives

The universal quantifier is a primitive notion of Coqbut the existential quantifier may be defined as an inductive proposition. The existential quantifier has only one introduction rule. It expresses that if we have a term t and a proof of $(P\ t)$ then we have a proof of $\exists x.(P\ x)$. It is defined by:

```
Inductive Connective ex [A:Set;P:A->Prop]
    = ex_intro : (x:A)(P x)->(<A>Ex(P)).
```

Remark that the syntax $< A > Ex(P)$ introduced before is an abbreviation for $(ex\ A\ P)$.

The elimination scheme says that: In order to prove C from <A>Ex(P) it is sufficient to prove: (x:A)(P x)->C.

Defining new composed connectives

A compound proposition like $(\exists x.\exists y.((Q\ y) \wedge (P\ x))) \vee (\forall x.(P\ x))$ may be coded using the previous operator. It may also be written with only one level of inductive type.

```
Inductive Connective Compound [A:Set;P,Q:A->Prop] =
   case_left : (x,y:A)(Q y)->(P x)->(Compound A P Q)
 | case_right : ((x:A)(P x))->(Compound A P Q).
```

It is important to remark the difference between existential and universal quantification on x in the left and right cases.

Definition of equality

It is also possible to define equality as an inductive predicate. It is defined for a set `A` and an element `x` of type `A` as the smallest predicate true for `x`. It says that `x` and `y` are equal if they represent convertible terms. The definition in the system is:

```
Inductive Definition eq [A:Set;x:A] : A->Prop
    = refl_equal : <A>x=x.
```

Again $< A > x = y$ is a short syntax for $(eq\ A\ x\ y)$.

The elimination principle is `eq_ind` of type:

`(A:Set)(x:A)(P:A->Prop)(P x)->(y:A)(<A>x=y)->(P y).`

Here also, this type corresponds to the usual elimination principle for equality. It says that in order to prove `(P u)` from a proof of `<A>t=u` it is sufficient to prove `(P t)`. When applied to a proof of `<A>t=u`, the elimination tactic will first try to rewrite the current goal as `(P u)` for some predicate `P` and then replace the current goal by the new goal `(P t)`.

1.8.3 Various inductive definitions of the same notion

One difficulty with inductive definitions is the fact that in general the same notion admits several possible inductive definitions. To each inductive definition corresponds naturally an induction principle (the minimality property). Different proofs may require the use of different induction principles.

For example, a natural property to prove is the transitivity of the order relation on natural numbers `LE`.

```
Goal (n,m,p:nat)(LE n m)->(LE m p)->(LE n p).
```

The natural step to take is to do an elimination on the proof of `(LE n m)` by `Induction 1`. We shall then need to prove the clause:

`(LE n m)->((LE m p)->(LE n p))->(LE (S m) p)->(LE (S n) p).`

But there is no direct proof of this property. The reader may also check that an induction on `n`, `m` or `p` does not lead to a direct proof.

There is another definition of the order corresponding to the informal definition of "to be greater or equal to `n`." It is the smallest property which is true for `n` and is true for `(S m)` when it is true for `m`. It corresponds to the inductive definition.

```
Inductive Definition le [n:nat] : nat -> Prop
    = le_n : (le n n)
    | le_S : (m:nat)(le n m)->(le n (S m)).
```

Proving the transitivity of this relation is simple:

```
Goal (n,m,p:nat)(le n m)->(le m p)->(le n p).
```

It is done by an elimination on the proof of `(le m p)`, then using the assumption and applying `le_S`. Also the equivalence between the two definitions of the order is not hard to prove.

```
Goal (n,m:nat)(le n m)->(LE n m).
```

After an elimination on the hypothesis `(le n m)`, we have to check two clauses. The property `(LE n n)` is easy to prove by induction on `n`. The property `(LE n m)->(LE n (S m))` was proved before. We now have to prove the other direction:

```
Goal (n,m:nat)(LE n m)->(le n m).
```

We use `Induction 1`. The property `(le 0 m)` is easy to prove by induction on `m`. The property `(le n m)->(le (S n) (S m))` is proved directly after an elimination on the proof of `(le n m)`.

We have shown two possible definitions of the same notion, and we cannot say that one is better than the other. The best way to proceed is to use one or the other induction principle depending on the proof to be done.

A good exercise is to try to show the following property, using the definitions of `plus` and `minus` introduced in 1.1.8:

```
Goal (n,m:nat)(le n m)-><nat>m=(plus n (minus m n)).
```

This is hard to prove directly but if we replace `le` with `LE`, the proof becomes very simple.

The role of parameters

Sometimes between two apparently equivalent definitions, there is one which is definitely better than the other. This problem is related to the role of parameters. In our previous definition of `le`, we could have written:

```
Inductive Definition le' : nat->nat->Prop
    = le'_n : (n:nat)(le' n n)
    | le'_S : (n:nat)(m:nat)(le' n m)->(le' n (S m)).
```

What is the difference between `le` and `le'`? Both are binary relations on natural numbers, and the type of `le_n` and `le'_n` are the same as the types of respectively `le_S` and `le'_S`. In the case of `le`, we define for each `n` a binary inductive predicate. But `le'` is an inductive relation. This difference is reflected in the minimality properties `le_ind` and `le'_ind`.

```
le_ind : (n:nat)(P:nat->Prop)
         (P n)->((m:nat)(le n m)->(P m)->(P (S m)))->(m:nat)(le n m)->(P m).
le'_ind : (R:nat->nat->Prop)
          ((n:nat)(R n n))->((n,m:nat)(le' n m)->(R n m)->(R n (S m)))
          ->(n,m:nat)(le' n m)->(R n m).
```

Now assume that we want to prove a goal $G(t,u)$ which depends on two natural numbers t and u and we want to use the fact that $t \leq u$. If we know `(le t u)` we will only have to prove $G(t,t)$ and $\forall m.G(t,m) \rightarrow G(t,(S\ m))$. But if we use `(le' t u)` we shall have to prove $\forall n.G(n,n)$ and $\forall n.\forall m.G(n,m) \rightarrow G(n,(S\ m))$. These properties are always stronger than the ones required in the `le` case.

Actually it is possible to do a simple proof of `(n,m:nat)(le' n m)->(le n m)` using only an elimination on the proof of `(le' n m)`. Thus the two definitions are equivalent. But the first one (`le`) is always the better.

The problem is to recognize that an argument of an inductive relation can be moved to a parameter position. This can be done when in each type of a constructor and each occurrence of the relation in the type the argument is the same bound variable (in our case n).

1.8.4 Definition of recursive propositions

Proving that $0 \neq 1$

The elimination scheme on an inductive type is quite a powerful method for doing proofs. But it is not always sufficient. For example some expected properties like `~<nat>0=1` are not provable using only these tools. A proof of `0=1` gives us a way to transform a proof of a property `(P 0)` into a proof of `(P 1)` using elimination. In order to derive absurdity from `0=1` it is sufficient to be able to find `P` such that `(P 0)` and `~(P 1)` are provable. For a restricted class of inductive definitions, we introduce the possibility of defining propositions by structural recursion. The syntax is similar to that used for the construction of a program by recursion on an inductive structure.

For natural numbers, we can define a predicate (function from natural numbers to *Prop*) which is the true proposition for `0` and absurdity for `(S p)`.

```
Definition P : nat->Prop
 = [n:nat](<Prop>Match n with (* 0 *) True (* S p *) [p:nat][H:Prop]False).
```

We have that `(P 0)` is internally equivalent to the true proposition `T` and `(P (S 0))` is internally equivalent to the absurd proposition `False`.

Recursive definitions of relations

This possibility to recursively define a proposition gives an alternative way to define some relations. A recursive version for the definition of the order `LE` can be given by:

```
LE' = : nat->nat->Prop
   [n:nat](<nat->Prop>Match n with
           (* 0 *) [m:nat]True
         (* S k *) [k:nat][LEk:nat->Prop][m:nat]
                     (<Prop>Match m with (* 0 *) False
                                         (* S l *) [l:nat][LESkl:Prop](LEk l)))
```

`LE'` is of type `nat->nat->Prop`. We have `(LE' 0 m)` convertible with `True`, `(LE' n 0)` convertible with `False` and `(LE' (S n) (S m))` convertible with `(LE' n m)`. What is missing, as a basic property, is the minimality of the relation for the defining clauses. This can be derived using the appropriate induction on natural numbers.

Not all inductive definitions can be rewritten using a recursive definition. This is the case for our definition of `le`. The two conclusions of the constructors `(le n n)` and `(le n (S m))` do not correspond to exclusive cases of constructors of natural numbers.

Inverting an inductive definition

Very often inductive definitions lead to more elegant proofs because of the elimination constructions. But sometimes an inversion property in an inductive definition is needed, for example the property `(LE (S n) (S m))->(LE n m)` or `~(LE (S n) 0)`. This inversion can be proved in a systematic way using a definition by cases of a relation:

```
Definition LE_fun = [n:nat]
     (<nat->Prop>Match n with
```

```
  (* O *) [m:nat]True
(* S k *) [k:nat][LE_funk:nat->Prop][m:nat]
       (<Prop>Match m with (* O *) False
                           (* S l *) [l:nat][LE_funkl:Prop](LE k l)))
```

We have `(LE_fun O m)` convertible with `True`, `(LE_fun (S n) O)` convertible with `False`, and finally `(LE_fun (S n) (S m))` convertible with `(LE n m)`. Then, using an elimination on the proof of `(LE n m)`, we get a trivial proof of `(n,m:nat)(LE n m)->(LE_fun n m)`. The expected inversion properties are then convertible with instances of this theorem. For example, the proposition `(LE (S n) O)->(LE_fun (S n) O)` is convertible with `~(LE (S n) O)`.

1.8.5 General rules for an inductive definition

This part contains a more precise definition of rules for inductive definitions. It is rather technical and may be skipped unless you are interested in the precise general mechanism of inductive definitions.

In this section s is one of the sorts *Set*, *Prop* or *Type*. The sorts *Set* and *Prop* play a similar role. *Set* is the sort associated to the types of the programming language (the type of natural numbers for example). *Prop* is the type of the propositions of the logic. *Type* is the type of *Prop*. It can be considered as part of the language in the Calculus of Constructions extended with universes. If we use the original Calculus with only three levels, *Type* appears only at the meta level (to give a type to *Prop*, $A \rightarrow Prop, \ldots$).

All terms we shall write in the following will be well-formed terms of the Calculus. We also call a term whose type is a sort a *type*.

Declaration of the type

Let us give some definitions. An *arity* is either a sort or a term $(x : T)A$ with A an arity. In the following A is an arity and X is a variable of type A. If M is of type A, we shall write $(M\ a1 \ldots an)$ to denote a type obtained by application of the term M to the n terms $a1, \ldots, an$.

Let P be a type. X is *strictly positive* in P if P is $(X\ a1 \ldots an)$ or $(x : T)P'$ where X is *strictly positive* in P' and X does not occur in any of the ai or in T.

Let C be a type, C is said to be a type of constructor of X if C is $(X\ a1 \ldots an)$ or $(x : T)C'$ or $P \rightarrow C'$ and C' is a type of constructor of X, X is *strictly positive* in P and X does not occur in any of the ai or in T.

An inductive definition X is specified by an arity A and a list of type of constructors of X $[C1; \ldots; Cp]$. It may be written in the system as $Ind(X : A.\{C1, \ldots, Cp\})$. Each Ci should be a term well-formed in the current environment plus the definition of the variable $X : A$.

The inductive type $Ind(X : A.\{C1, \ldots, Cp\})$ has type A.

The usual way to define an inductive type is to choose a name X for the type and $c_1, ..., c_p$ for the constructors and to use the following scheme:

$$\begin{array}{rcl} \textit{Inductive Definition } X : A = & & c1 : C1 \\ & & \vdots \\ & & \mid cp : Cp. \end{array}$$

Parameters

In the previous examples we have seen that it is possible to add parameters to the definition. They appear as a list after X. The syntax is $[p1 : A1; \ldots ; pk : Ak]$ where each pi can be a single name or a list of names of parameters separated by commas. In a definition with parameters there is an extra condition. Let us write $(X\ p1 \ldots pk)$ for the variable X applied to all the declared parameters. Let X' be a variable of type A. There should exist for each Ci which is a type of constructors of X', a Ci' such that $Ci = Ci'[X'/(X\ p1 \ldots pn)]$. Then the definition with parameters is equivalent to first adding the parameters $p1 : A1; \ldots ; pk : Ak$ to the environment, introducing an inductive definition of $C1', \ldots , Cp'$, and then abstracting all the propositions with respect to the parameters. Hence, inductive definitions with parameters are always reducible to the case without parameters.

Constructors

Let $M = Ind(X : A.\{C1, \ldots , Cp\})$ be a well-formed inductive definition. Then for each integer i less than or equal to p, $Constr(i, M)$ is a well-formed term of type $Ci[x/M]$.

The effect of a general inductive definition scheme is to declare ci as a constant whose value is $Constr(i, M)$.

Elimination schemes

The generic rules for the elimination schemes are quite hard to read, so we will not write them. The examples given earlier give a better idea of this general scheme.

We will describe the kind of elimination which is allowed for a given inductive definition $M = Ind(X : A.\{C1, \ldots , Cp\})$. The primitive notion for elimination is the term $< P > Match\ t\ with$ where t is of type $(M\ a1 \ldots an)$ and P is a term whose type is an arity B. First of all, the elimination is distinguished by the arity of the predicate P with respect to the arity of the inductive definition A. Let us decompose A as $(x1 : A1) \ldots (xn : An)s$. The arity B may be either $(x1 : A1) \ldots (xn : An)s'$ or $(x1 : A1) \ldots (xn : An)(M\ x1 \ldots xn) \rightarrow s'$. In the first case, we shall call $< P > Match\ t\ with$ a non-dependent elimination of sort s' and in the second case, a dependent elimination of sort s'. Obviously, because the arity A can be deduced from the type of t, only the dependency and the sort s' are necessary to characterize the elimination. Dependent elimination corresponds, for example, to the induction principle for natural numbers. Non-dependent elimination corresponds to the construction of a primitive recursive function or the minimality property for inductive predicates. In the dependent elimination case, the constructors of the inductive definition appear, while in the non dependent case, they do not. Another trivial remark is that non-dependent elimination is a particular case of dependent elimination. If P is of type $(x1 : A1) \ldots (xn : An)s'$, we may see $< P > Match\ t\ with$ as a shorthand for $< [x1 : A1] \ldots [xn : An][H : (M\ x1 \ldots xn)](P\ x1 \ldots xn) > Match\ t\ with$. But writing an arbitrary elimination for an arbitrary inductive type is not allowed. We give the rules by case analysis on the sort s of the inductive type. In some sense, they corresponds to the rules in the presentation of the Calculus of Constructions as a Generalized Type System.

- $s = Type$: dependent elimination of sorts *Prop* and *Type* is allowed.
- $s = Prop$: non-dependent elimination of sort *Prop* is allowed. Dependent elimination does not make sense because we are not considering a proof of a proposition as an object.

- $s = Set$: Dependent elimination of sorts *Set* and *Prop* is allowed. A dependent elimination of sort *Type* is also allowed for the construction of a restricted set of recursive predicates. The restriction is that the type of constructors of the inductive definition should be what we call "small". This means that all the types of arguments have the sort *Prop* or *Set* (but not *Type*). Without this restriction it would be possible to define a strong sum $\exists X : Prop.P(X)$ with two projections, and thus to get an inconsistency.

Limitation of the definitions

There is a restriction in our definitions on the type of a constructor. We should allow X to be strictly positive in an inductive definition $Ind(Y : B.\{C1, \ldots, Cp\})$ if X is strictly negative in the Ci. This would allow an equivalent definition of lists by:

```
Inductive Set list' = nil : list' | cons :(A*list')->list'.
```

But in this general case, the elimination rules do not have a nice formulation (we need to introduce product and projection to write them). In practical examples there is a way to express the inductive definitions with our definition of strictly positive. The main drawback of this limitation is the impossibility to represent mutually recursive types using a product in this framework.

The role of names

In the current implementation names are not directly related to the inductive definitions. The internal representation does not mention the names of the type or of the constructor. It is even possible to manipulate an inductive type and its constructors without naming them. The equality between two inductive types depends only on the specification of the type of the constructors. This is different from the ML tradition where the booleans and a concrete type with two 0-ary constructors are not identified.

Chapter 2

The System Coq as a Programming Language

We explain here how to use the Calculus of Constructions for the development of certified programs. The first part explains how to derive and check a development in the system Coq. The second part explains how to get a real program from the development and how to execute it.

2.1 Development of programs with logical information

2.1.1 Motivations

In the first chapter we saw that the system Coq contains a programming language and that it is possible to describe functions in it. We also showed how to do proofs in the system. In particular we may do proofs of programs in it. In this part we shall proceed differently. Instead of having proofs which tell us about programs, we shall include proofs inside our programs. Usually in a programming language it is possible to include comments which explain properties of the program or why these properties are satisfied. In the Coq formalism, these comments can be part of the language and thus be mechanically checked.

2.1.2 Examples of specifications

Instead of developing a program as a term of type `nat->nat`, for instance, we shall be able to express more informative types, including logical information on programs. These types will also be called specifications in the following.

Let `P` be a predicate on the natural numbers. `P` is a term of type `nat->Prop`. A typical specification will be `{x:nat|(P x)}`. This expression specifies all the natural numbers `n` such that there exists a proof of `(P n)`. More precisely an element of this specification will have two components. The first is a natural number `n` and the second a proof of `(P n)`. If `A` and `B` are two propositions, another typical specification is `{A}+{B}`. This expression specifies a boolean value which is `left` if there exists a proof of `A` and `right` if there exists a proof of `B`. Actually an element of this specification is either a left injection of a proof of `A` or a right injection of a proof of `B`.

Let `Q` be a relation on natural numbers (of type `nat->nat->Prop`). The specification `(x:nat)(P x)->{y:nat|(Q x y)}` specifies a program which associates to each input `n` of type `nat` such that

there exists a proof of (P n), an output m of type nat such that there exists a proof of (Q n m).

In the same spirit, (x:nat)({(P x)}+{~(P x)}) specifies the characteristic function of the predicate P.

These are the most frequently used specifications but it is also possible to write more sophisticated combinations. For example, let us specify a program for division by two.

Using twice = [m:nat](plus m m), we may write its specification as:

```
(n:nat){m:nat & {<nat>n=(twice m)}+{<nat>n=(S (twice m))}}
```

An element of {m:nat & (Q m)} is a natural number m plus an element of (Q m) which is a specification. This expression specifies a program to transform a natural number n into a natural number m plus a boolean information which is left if <nat>n=(twice m) and right if <nat>n=(S (twice m)). This is different from the specification:

```
(n:nat){m:nat|(<nat>n=(twice m))\/(<nat>n=(S (twice m)))}.
```

where the boolean information does not appear (the disjunction property is a comment).

Specifications as inductive sets

Our definitions of specifications are just particular cases of inductive sets taking propositions as arguments. They are defined by the following declarations in the Specif.v file.

```
Inductive Set sumbool [A,B:Prop]
    = left : A ->({A}+{B}) | right : B->({A}+{B}).
```

{A}+{B} is an abbreviation for (sumbool A B).

```
Inductive Set sig [A:Set;P:A->Prop]
    = exist : (x:A)(P x)->{x:A|(P x)}
```

{x:A|(P x)} is an abbreviation for (sig A P).

```
Inductive Set sigS [A:Set;P:A->Set]
    = existS : (x:A)(P x)->{x:A & (P x)}
```

{x:A & (P x)} is an abbreviation for (sigS A P). Another useful specification is the type of programs which either return an object which satisfies the specification A or indicates that some proposition B is true.

```
Syntax sumor "_+{_}".
Inductive Set sumor [A:Set;B:Prop]
    = inleft : A -> (A+{B}) | inright : B -> (A+{B}).
```

We also have predefined inductive sets for an existential with two predicates:

```
Inductive Set sig2 [A:Set;P,Q:A->Prop]
    = exist2 : (x:A)(P x)->(Q x)->{x:A | (P x) & (Q x)}

Inductive Set sigS2 [A:Set;P,Q:A->Set]
    = existS2 : (x:A)(P x)->(Q x)->{x:A & (P x) & (Q x)}
```

{x:A | (P x) & (Q x)} and {x:A & (P x) & (Q x)} are abbreviations for respectively (sig2 A P Q) and (sig2s A P Q).

2.1.3 Examples of developments

It has been known for a long time that in order to write a correct program, we need to develop the program in a rigorous way. In particular we need to specify it and introduce some invariant properties. The system Coq is a good tool for the formalisation of such rigorous development of programs. The system will allow complete verification of the various proofs encountered in program development.

Let us present a few steps of the development of the "division by two" specification. We start from the specification below containing extra boolean information:

```
(n:nat){m:nat & {<nat>n=(twice m)}+{<nat>n=(S (twice m))}}
```

The proof is by induction on n (we can use `Induction n`). In the base case we have to prove:

```
<nat>{m:nat & {<nat>0=(twice m)}+{<nat>0=(S (twice m))}}
```

With instance `0` for `m`, we generate the subgoal:

```
{<nat>0=(twice 0)}+{<nat>0=(S (twice 0))}
```

Using the `Simpl` tactic, the subgoal is rewritten to:

```
{<nat>0=0}+{<nat>0=(S 0)}
```

which is solved by indicating that we are in the left case plus a trivial proof of `<nat>0=0` (or directly with `Auto`). In the inductive case, we have to prove:

```
(n:nat){m:nat & {<nat>n=(twice m)}+{<nat>n=(S (twice m))}}
      -> {m:nat | {<nat>(S n)=(twice m)}+{<nat>(S n)=(S (twice m))}}
```

We eliminate the induction hypothesis (`Induction 1`). From a mathematical point of view, this corresponds to the following reasoning. We know the property:

"there exists some `m` such that `{<nat>n=(twice m)}+{<nat>n=(S (twice m))}`".

We introduce an object `m` such that `{<nat>n=(twice m)}+{<nat>n=(S (twice m))}`. From the computational point of view we do the recursive call that gives us a natural number `m` plus an object which satisfies the specification `{<nat>n=(twice m)}+{<nat>n=(S (twice m))}`. This generates the subgoal:

```
(n:nat)({<nat>n=(twice m)}+{<nat>n=(S (twice m))})
      -> {m:nat & {<nat>(S n)=(twice m)}+{<nat>(S n)=(S (twice m))}}
```

An elimination on `{<nat>n=(twice m)}+{<nat>n=(S (twice m))}` (using `Induction 1`) leads us to two subgoals:

```
(<nat>n=(twice m))->{m:nat & {<nat>(S n)=(twice m)}+{<nat>(S n)=(S (twice m))}}
(<nat>n=(S (twice m)))->{m:nat & {<nat>(S n)=(twice m)}+{<nat>(S n)=(S (twice m))}}
```

The first subgoal is solved by giving the explicit witness `m`, then indicating that we are in the right case, then applying arithmetical properties. The second subgoal is solved by giving the explicit witness `(S m)`, then indicating that we are in the left case, then applying arithmetical properties. At each step the specifications are used to help find the program. We can get the intended program in pure form by removing the comments. We write it in an ML-like language.

```
let rec div2 = function 0      -> (0,left)
                      | (S n) -> let (m,p) = div2 n in
                                 match p with left  -> (m,right)
                                            | right -> ((S m),left)
```

We may get other programs which for example will not use the boolean information. For instance, the following is an example specification of such a program:

```
(n:nat){m:nat | (<nat>n=(twice m))\/(<nat>n=(S (twice m)))}
```

For example, one may use an induction principle which says that if $P(0)$ and $P(1)$ are true and if $\forall x.P(x) \Rightarrow P(x+2)$ is true then $\forall n.P(n)$ is true. First of all we have to justify this induction principle. To do so, we must prove:

```
(P:nat->Set)(P 0)->(P (S 0))->((n:nat)(P n)->(P (S (S n))))->(n:nat)(P n).
```

In order to apply this induction principle to our goal we first (after the introduction of `n`) write the goal `{m:nat|(<nat>n=(twice m))\/(<nat>n=(S (twice m)))}` as an application `(P n)`. This is done with the tactic `Pattern n`. Then we apply the proof of the induction principle. We get three subgoals:

```
{m:nat|(<nat>0=(twice m))\/(<nat>0=(S (twice m)))}
{m:nat|(<nat>(S 0)=(twice m))\/(<nat>(S 0)=(S (twice m)))}
(n:nat){m:nat|(<nat>n=(twice m))\/(<nat>n=(S (twice m)))}
       ->{m:nat|(<nat>(S (S n))=(twice m))\/(<nat>(S (S n))=(S (twice m)))}
```

The first two goals are solved by giving the witness `0`. For the last goal we first eliminate the induction hypothesis and get an object `m` and a hypothesis

```
H: (<nat>n=(twice m))\/(<nat>n=(S (twice m)))).
```

The goal to prove is:

```
{m:nat|(<nat>(S (S n))=(twice m))\/(<nat>(S (S n))=(S (twice m)))}
```

It is not possible at this point to do an elimination on hypothesis `H`. This is because it is a logical proposition (of type `Prop`), whereas the goal is a specification (of type `Set`). Such an elimination scheme is not allowed in general. The logical statement
`(<nat>n=(twice m))\/(<nat>n=(S (twice m))))` should just be considered a comment. In the previous case, the statement `{<nat>n=(twice m)}+{<nat>n=(S (twice m))}` corresponded to the computation of a boolean value indicating whether `n` is even or odd, it was possible to introduce a conditional on this value. But in this case, we do not need any test in order to provide a witness `(S m)` for the proof of the goal. We have to prove:

```
(<nat>(S (S n))=(twice (S m)))\/(<nat>(S (S n))=(S (twice (S (S m)))))
```

from the hypothesis `(<nat>n=(twice m))\/(<nat>n=(S (twice m))))`. This is done by an elimination on this hypothesis (now the goal is a proposition of type `Prop`, and the elimination is possible) plus arithmetical properties. This part of the development is now inside a comment. The pure program obtained by removing all comments is:

```
let rec quo2 = function 0        -> 0
                      | (S 0)    -> 0
                      | (S (S n)) -> let m = quo2 n in (S m);;
```

2.1.4 The role of dependent types

One way to introduce logical information into a type is, as we have seen before, to mix propositions and types. Another way is to introduce directly a dependent type. A typical example is the type of lists of a given length n. Let us call A the type of the elements in the list. One way to represent such a notion is to consider the specification {l:(list A)|<nat>n=(length A l)}. An element in this type is built from a list l and a proof of <nat>n=(length A l). Another way is to directly represent a type of lists of some length as an inductive function from nat to Set as follows :

```
Inductive Definition llist A : nat->Set
      = nil : (llist 0)
      | cons : (n:nat)A->(llist n)->(llist (S n)).
```

The append function on such lists may be defined as:

```
append = [n,m:nat][l:(llist n)][l':(llist m)]
 (<[k:nat](llist A (plus k m))> Match l with
        (* nil *) l'
(* cons k a lk *) [k:nat][a:A][lk:(llist A k)][appk:(llist A (plus k m))]
                  (cons A (plus k m) a appk))
  : (A:Set)(n,m:nat)(llist A n)->(llist A m)->(llist A (plus n m))
```

The use of such definitions is interesting because the append function is directly typable as an element of (n,m:nat)(llist n)->(llist m)->(llist (plus n m)). We do not have to write both the program of append and the proof that the length of the concatenation of two lists is the sum of the lengths. But this may also lead to unexpected problems. For example two lists with different lengths have different types. Even if these lengths are provably equal but not internally convertible the types are different. This raises a problem when we want to write equality on such lists. For example we may want to prove associativity of the append function. Then the term (append n1 (plus n2 n3) l1 (append n2 n3 l2 l3)) admits for type the term (llist (plus n1 (plus n2 n3))), but (append (plus n1 n2) n3 (append n1 n2 l1 l2) l3) admits for type (llist (plus (plus n1 n2) n3)), and these two types are not convertible. A solution is to introduce a new notion of equality which compares l:(llist n) and l':(llist n'). This can be done with an inductive predicate:

```
Inductive Definition eqllist [k:nat;x:(llist k)] : (l:nat)(llist l)->Prop
       = eql_refl : (eqllist k x k x).
```

With this definition we may express and prove the associativity of the append function.

```
(eqllist (plus n1 (plus n2 n3))
         (append n1 (plus n2 n3) l1 (append n2 n3 l2 l3))
         (plus (plus n1 n2) n3)
         (append (plus n1 n2) n3 (append n1 n2 l1 l2) l3)).
```

The proof is as usual by induction on the list l1.

2.1.5 Relationships between *Prop* and *Set*

In the system Coq, every correct term has a type, and this type itself has a type called a sort, following the presentation of Generalized Type Systems. Here, we have distinguished two sorts *Set* and *Prop*. An object whose type is *Prop* is called a proposition and an object whose type is *Set* is called a specification. A term whose type is a proposition represents a logical proof. A term whose type is a specification represents a program or a program development. These are not separate worlds. A proposition may express properties of programs (or even of program development) so that it is possible to do proofs of programs. Conversely a program may contain logical information.

The obvious restriction in this interpretation of proofs and program development is that a proof of logical information may never be used for constructing a computational part of a program. This distinction is made by type-checking. If a term has type `(C:Set)P` then it is not possible to apply this term to a proposition. Conversely if the term has type `(C:Prop)P`, it is not possible to apply it to a Set.

An advantage of this distinction is the possibility to introduce new axioms which apply only to the logical part of the proofs. For example whereas intuitionistic reasoning is needed for a proof of a specification in order to get a direct functional interpretation of the term, we may use classical reasoning for comments. We just need to introduce the postulate:

```
Axiom classic : (A:Prop)(~~A)->A.
```

There is also a restriction on elimination of inductive types. If a term has type `I` with `I` an inductive definition whose sort is *Prop*, an elimination on this object can only be done in order to build other propositions. Thus, if your goal is a specification, it is not possible to apply the tactic `Elim` to a term whose type is a proposition. The converse is possible. If the goal is a proposition and if the term you want to eliminate has a specification for type, it is possible to apply the elimination.

Self-realizing propositions

Although in general we cannot eliminate a proof object to obtain a specification, there are particular cases where it is consistent with our interpretation to do so. For example from a proof of the absurd proposition it is possible to justify that every specification is correct (every program in this case is correct !). If we know the property `<A>a=b` and we have a program correct with respect to the specification `(P a)` then it is also correct with respect to the specification `(P b)`. This justifies the introduction of the following two axioms.

```
False_rec :(P:Set)False->P.
eq_rec :(A:Set)(a:A)(P:A->Set)(P a)->(b:A)(<A>a=b)->(P b)
```

There are other examples of such propositions. For example an inductive proposition with only one constructor whose arguments are non-informative propositions is a self realizing proposition. In this case we may build a proof of the elimination property on specifications:

```
and_rec = [A,B:Prop][C:Set][F:A->B->C][AB:A/\B]
           (F <A,B>Fst{AB} <A,B>Snd{AB})
        : (A,B:Prop)(C:Set)(A->B->C)->(A/\B)->C
```

For an inductive predicate like LE, which is invertible, we may find a proof of the elimination on specifications whose type is:

```
(P:nat->nat->Set)
        ((n:nat)(P O n))->
        ((n:nat)(m:nat)(LE n m)->(P n m)->(P (S n) (S m)))->
        (n,m:nat)(LE n m)->(P n m)
```

For this we prove `(n,m:nat)(LE n m)->(P n m)` first by induction on `n` and then by induction on `m` using the properties `~(LE (S n) O)` and `(LE (S n) (S m))->(LE n m)`.

2.1.6 Correctness of the extracted programs

The extraction procedure is part of the Coq engine. Each time a term is introduced in the machine, its type is computed as well as its algorithmic contents. The algorithmic part of a program development is what is obtained after removing the logical part (proofs which are part of the programs). In this part of extraction, the types are kept but dependencies on terms in types are removed. We get terms correctly typed in the system F_ω with inductive types.

We have to relate the extracted program to the original specification. This is done at the theoretical level by a realisability interpretation which is not part of the Coq implementation. We shall not describe it in this manual.

Roughly speaking we associate to each specification S a property which describes a set of programs "correct with respect to S". The theory makes sure that the term extracted from a proof of S satisfies this property.

The realisability interpretation corresponds to our intended meaning for a program to be correct with respect to its specification. For example, the term extracted from a proof of $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ with $P(x)$ and $Q(x,y)$ of type *Prop* leads to a unary function f such that $\forall x.P(x) \Rightarrow Q(x,f(x))$ is satisfied. With $P(x)$ of type *Prop*, the term extracted from a proof of $\forall x.(P(x) \vee \neg P(x))$ leads to a boolean function which is the characteristic function of the predicate P.

The interest of a realisability interpretation is not just to make sure that the extracted program is correct but also to give the possibility of interpreting axioms. Assume that there is a program p correct with respect to a specification P and that we develop a proof q of a specification Q under the assumption P. A priori the term t extracted from q is not closed and cannot be executed, but if we replace the occurrences of the assumption in the program t by the program p then we get a closed term which is still correct with respect to the specification Q. So we may use assumptions which are not provable but for which a correct program can be found or we can use a justification which cannot be derived from a proof in the Calculus of Constructions. This justifies, for example, the introduction of the axioms `False_rec` and `eq_rec` in the theory.

We have to say something about the termination property of the extracted programs. The extracted term is still a program correctly typed in the Calculus of Constructions. So we may deduce that it is strongly normalizable. This will no longer be true if we introduce terms containing a fixpoint combinator for realizing axioms. In this case we may introduce termination conditions inside the specification. Indeed some specifications imply the normalizability of the intended programs. For example the set `nat` seen as a specification contains every normalizable term (written for example in a ML-like language with non-terminating programs) that reduces to a natural number $(S^n\ 0)$ for some n. But of course we loose the strong normalisation property.

2.2 The Program extraction facilities

We have explained why it is possible to use Coq to build certified and relatively efficient programs. We now see how the extraction part is done in practice.

2.2.1 Sketching the extraction algorithm

The main ideas have already been exposed previously in section 2.1.2. Having in mind what the extracted program should look like, we may see that the (say ML) compiler will need two kind of informations: on one hand the description of the concrete (inductive) types, i.e. the number of constructors and their respective arities; on the other hand, the actual program parts, i.e. the λ-terms which correspond to the computational parts of proofs.

This means the implemented algorithm scans the proof environement and takes care of two kind of terms:

- The terms of Coq, whose type is of the form $(x_1 : T_1)(x_2 : T_2) \ldots Set$. These terms will be mapped to *types* or *type schemes* of the extracted program. Basic examples are `nat` or `list`.
- The Coq terms whose type has type *Set*. These terms will be mapped to *terms* of the extracted program (for example `add` for addition, `heapsort` for the sorting routine, etc).

The other terms, those of type *Prop* or those whose type has type *Prop* won't have any counterpart in the program. They are what can be considered *comments* of the actual code (i.e. assertions and their validations).

For readers familiar with higher-order types systems, let us mention that the terms which are obtained through this algorithm are typed in Girard's system F_ω enriched with inductive types (sometimes called F_ω^{idt}).

2.2.2 The principles of the implementation

In this implementation, we chose to use existing compilers to execute the extracted programs. This choice was made for various reasons:

- Generating code for existing compilers allows us to take advantage of all the technological know-how used in these compilers in order to get efficient executable code.
- It allows some comparison with the corresponding hand-written code.
- It may suggest some new improvements, such as incorporating other features of the programming language into the logical level.

In order to be able to use existing ML compilers, we defined our own extraction language called FML. FML programs can be generated from Coq proofs of certain specifications using the extraction algorithm, and erasing the type information in the abstractions of the extracted F_ω^{idt} terms.

The extracted terms can be mixed with some hand-written FML code. Execution is made possible by a simple translation to ML code. There is one drawback to this approach: since the ML type system is weaker than F_ω, some extracted programs are not ML-typable. We describe this phenomenon more precisely in section 2.2.10. Yet in practice, this happens very rarely, especially in the case of proofs of actual program specifications.

2.2.3 The main features

FML is the target language of the extraction. It can be viewed as a non-strongly-typed version of the functional core of ML (i.e. λ-calculus + concrete types and pattern-matching). As said above, there is no specific FML compiler or interpreter. The execution of the extracted programs is delegated to existing ML compilers. At present time, it is possible to generate code for the following implementations:

- CAML itself, which has the advantage of extracting into the system's implementation language, but requires some code optimization since the CAML compiler performs strict (or eager) evaluation.
- LML, The lazy ML implementation of Göteborg *.
- GAML, a similar experimental implementation of lazy ML, also based on graph reduction, due to Luc Maranget in INRIA†.

Therefore we shall describe three main kinds of features in this section:

- How to extract FML programs out of proofs.
- How to manipulate these FML programs, and how to write directly in FML.
- How to produce executable ML code out of FML.

We will end up by showing how to use some existing programs of the proof library, and try to illustrate the proof style needed for program extraction.

2.2.4 Using FML

FML has its own toplevel, which is obtained by typing `fml();;` from CAML. The FML prompt is `Fml <`. Let us illustrate the main features through examples.

Basic features and syntax

FML commands end with a period. One can define constants as in ML, and integer expressions are FML terms:

```
ml #fml();;

Fml < let a = 1.
a is defined:
1
Fml < let B = 1*a-3.
B is defined:
1*a-3
Fml < Print B.
```

*The LML compiler is available for free by anonymous ftp at the Chalmers university in Göteborg.

†This implementation is still under development. Interested readers may contact *maranget@margaux.inria.fr*

```
B = 1*a-3
Fml < let c = d.
The constant d is undefined.
```

λ-abstraction is denoted by square brackets, like in Coq, but without the type information.

```
Fml < let f = [x,y,z]x+2*(y-z).
f is defined:
[x][y][z]x+2*y-z
```

We provide the boolean constants, two comparison tests over integers and the classical *if ... then ... else*.

```
Fml < let tr = true.
tr is defined:
true
Fml < let test = if B<a then 2 else (f 1 3 9).
test is defined:
(if B<a then 2 else (f 1 3 9))
```

Finally recursion and local definitions are allowed:

```
Fml < let rec F = [x]if x=0 then 1 else 2*(F x).
F is defined:
REC [x](if  x=0 then  1 else 2*(F x))

Fml < let G = [x,y](let rec g = [z](if z=0 then 1 else (g (z-1))*y) in g x).
G is defined:
[x][y](let rec g = [z]if  (z=0)  then  1 else  ((g (z-1))*y)  in g x)
```

As the reader will have noticed, no evaluation of the constants is performed interactively, as this job will be left to the target ML-compiler. Note also that no type-checking is performed, and thus the user should be careful when defining his own terms.

As we have seen, the command `Print` followed by some identifier prints the term to which this identifier is bound in the current FML environment. In the same way, the command `Env.` displays the whole environment on the screen.

Concrete types and pattern matching

Similarly to ML, one can define concrete types. Here is the well-known definition of unary integers:

```
Fml < Inductive NAT = O | S NAT.
type NAT is defined:
 == inductive
    O | S NAT
```

which means that the type `NAT` has two constructors `O` which takes no argument, and `S` which takes one argument of type `NAT`. One can also define type schemes, like polymorphic lists:

```
Fml < Inductive List a = nil | cons a (List a).
type List is defined:
a  == inductive
      nil | cons a ( List  a)
Fml < Print nil.
nil = Constr{1,List}
Fml < let rec interval = [n](if n=0 then nil else cons n (interval (n-1))).
interval is defined:
REC [n](if n=0 then nil else (cons n (interval (n-1))))
```

The essential point is of course pattern matching:

```
Fml < let tl = [l] match l with
Fml <    nil -> nil
Fml < | (cons a l') -> l'
Fml < end match.
tl is defined:
[l]match l with
      nil -> nil
    | (cons a l') -> l'
    end match
```

It is also possible to define type abbreviations:

```
Fml < ari == nat -> nat.
type ari is defined:
 == nat -> nat
```

The command `Types.` prints the list of the currently defined types.

2.2.5 Extracting toward FML

At the beginning of a session, the FML context contains the code extracted from the prelude. The command `Reset` resets it to that original state. The main command is `Extract`, which extracts all the Coq environment except the prelude (which should be already translated) to the FML environment.

```
Fml < Extract.

Fml < Env.
False_rec = #Exit
pair = [Var1][Var2]Constr{1,prod}<Var1,Var2>
                                    .....
```

It is also possible to extract only parts of the context. This is useful when combined with the semantical attachment facilities (see section 2.2.9). The commands are:

- `Extract Until` followed by the name of the last Coq object to be extracted.

- `Extract From` followed by the name of the last Coq object not to be extracted.
- `Extract From ... To ... .` which translates the part of the Coq context between the two parameters.

While developing his program on the Coq top-level, the user may want to see the extracted terms corresponding to his proofs. He can do so using the following commands:

- `Extraction` : prints the whole extracted context in his F_ω form; i.e. the typed counterpart of the FML program. The same can be achieved with the CAML `print_extraction();;` command.
- `Extraction` *name* : just prints the F_ω program extracted from the term *name*.

In some cases, one may want to empty completely the environment and get rid of the code corresponding to the `Prelude.v` file:

```
Fml < Reset All.

Fml < Env.
```

If one wants to extract from the entire Coq environment (after a `Reset All`), the command is `Extract All`.

2.2.6 Saving FML files

It is possible to save the FML environment in a file by the command `Save` *name*, which will produce a file of name *name* suffixed by `.f`. The saved environment is restored by `Load` *name*.

2.2.7 Generating executable lazy ML

The system has a pretty-printing facility, allowing the user to write the contents of the environment in a given file with lazy ML syntax. This file can be compiled afterwards by the appropriate compiler. A system flag enables the user to generate either LML or GAML code. LML is chosen by default. The CAML commands to switch the flag are:

```
ml #lml;;
<fun> : (unit -> unit)

ml #gaml;;
<fun> : (unit -> unit)
```

To generate a compilable file, the FML command is `Write File` followed by the term which has to be evaluated in the generated program. For example:

```
Fml < Write File 1.
```

will produce an ML program always returning 1.

The file is generated in the current directory. Its default name is `extract.m` (resp. `extract.gl` for GAML). One can also give a precise name, for example:

```
Write dummy File 1.
```

which will change the file name to `dummy.m` (resp. `dummy.gl`).

Last but not least, it is possible to optimize the obtained code by a certain amount of partial evaluation. The FML command is `Optimise`; it tries to evaluate the definitions present in the current environement and possibly expands and deletes the not-strict functions. For "reasonable" programs, this gives shorter, faster and more readable code. The execution time should not be too long. Yet for certain very complex proofs (e.g. Higman's lemma), the optimization may take too much time and memory to be performed.

2.2.8 Generating CAML code

Since CAML is a strict language, the extracted code has to be optimized. So the optimization routine will be called each time the user wants to generate CAML programs. CAML being strict and interactive, it is not necessary to choose the term to be evaluated at code generation time. The FML command is `Write CAML File` followed by the name given to the CAML file to be produced. The CAML function `make_caml_file : string -> unit` performs the same job.

It is also possible to hand the produced CAML code directly to the current CAML toplevel. This is done by the CAML command `extract_caml : unit -> unit`. One has to be a little careful using it, since it generally redefines primitive CAML types such as `bool`, `unit` or `list`. We will show in the examples how that feature may be used in practice.

2.2.9 Realizing axioms

We saw in the first chapter, that it was possible to assume some axioms while developing a proof. Since these axioms can be any kind of proposition or object type, they may perfectly well have some computational content, but of course the system cannot guess the program which realizes them. Therefore, it is possible to tell the system what FML term or type corresponds to a given Coq variable.

For type variables, the command is `Attach` followed by the name of the variable and the FML type. For example:

```
Fml < Attach A Int.
```

will associate the type of built-in integers to *A*. If the user tries to extract from a portion of the Coq context containing a type variable which has not been instantiated, the extraction fails.

For term variables, the command is `Realize` followed by the name of the variable and the corresponding FML term.

These semantical attachments have to be done *before* typing the `Extract` command. A variable which has not been realized will be translated by the FML `#Error` term, corresponding to `fail` in CAML, or the lazy exception in LML or GAML.

Let us try to illustrate this feature by an example: The *Heapsort* program contained in the library is defined for lists of elements of some type variable *A* of type *Set*:

```
Variable A : Set.                                    (in List.v)
```

The specification proof also assumes that there is an order relation *inf* over that type (which has no computational content), and that this relation is total and decidable:

```
Variable   inf : A -> A -> Prop.
Hypothesis inf_total : (x,y:A){(inf x y)}+{(inf y x)}.          (in Heap.v)
```

Now suppose we want to use this specification proof to obtain a sorting program for lists of ML integers; this means A has to be instantiated by the FML type *Int*, and the axiom *inf_total* will be realized using the < operator of FML. Here is how to proceed once the proof is loaded:

```
Fml < Reset.
Fml < Attach A Int.
Fml < Realize inf_total [x,y]if x<y then left else right.
Fml < Extract.
```

It is possible to list the Coq variables having computational content with the Fml `Free Vars.` command or `print_extracted_vars();;` in CAML. For a better understanding of these features, we advise the reader to take a look at the different examples given later on.

2.2.10 Programs that are not ML-typable

The formal extraction algorithm mapping Coq proofs to F_ω^{idt} programs, the extracted code is not proved to be ML-typable. There are in fact two cases which can be problematic:

- If some part of the program is "very" polymorphic, there may be no ML type for it. In that case the extraction to FML works all right but the generated code may be refused by the ML type-checker. A very well known example is the "distr_pair" function: `let dp (a,b) f = (f a,f b)` which cannot be used with its full power in the ML type-system.

- Some definitions of inductive types of F_ω^{idt} may have no counterpart in ML. This happens when there is a quantification over types inside the type of a constructor; for example:

  `Inductive Set anything = dummy : (A:Set)A->anything.`
  which corresponds to the definition of ML dynamics.

The first case is not too problematic; it is still possible to run the programs by switching off the type-checker during compilation. Unless you misused the semantical attachment facilities you should never get any messages like "segmentation fault" for which the extracted code would be to blame. To switch off the type-checker of the LML compiler, use the option *-t*. For CAML, the type-checker is switched off by the command `#open typing true;;` [‡]. One can also use the "magic" but dangerous function `unchecked_coercion` which gives the type `'a` to any object; but this implies changing a little of the extracted code by hand.

The second case is fatal. If some inductive type cannot be translated to FML one has to change the proof (or possibly to "cheat" by some low-level manipulations we would not describe here).

We have to say, though, that in most "realistic" programs, these problems do not occur. For example all the programs of the library are accepted by ML type-checkers except `Higman.v` [§].

[‡]Note that this feature of CAML may in some cases be unsafe.

[§]Should you obtain a not ML-typable program out of a self developed example, we would be interested in seeing it; so please mail us the example: *coq@margaux.inria.fr*

2.3 Some examples

2.3.1 Euclidean Division

The file `Euclid.v` contains proofs of some simple results about the primitive operations on natural numbers, and ends up with the proof of Euclidean division. The natural numbers defined in the example files are unary integers defined by two constructors *O* and *S*. The corresponding CAML definition would be:

```
ml #type nat = O
ml #         | S of nat;;
```

To use the file, we begin by loading it into the Coq environment:

```
ml #V "Euclid";;
```

Once this is done, we can play with the extracted program:

```
ml #fml();;

Fml < Extract.
Fml < Print eucl_dev.

eucl_dev = [b]
            [a]
             (gt_wf a
             [n]
              [H0]
               (sumbool_rec
                   (diveucl_rec (minus n b) b [q][r](divex n b (S q) r)
                   (H0 (minus n b)))
                (divex n b O n)
               (le_or_gt b n)))

Fml < Types.
        .....
diveucl  == inductive 'divex   nat nat
```

The function *eucl_dev* will take two unary numbers as arguments, and return a *diveucl* pair containing the quotient and the rest. We can toy a little in FML to translate the unary numbers into built-in integers and conversely. This allows us to use integers in decimal notation.

```
Fml < let rec to_int = [n]
Fml <   match n with
Fml <     O -> 0
Fml <   | (S m) -> 1+(to_int m)
Fml <  end match.

Fml < let rec to_nat = [N]
```

```
Fml <   (if N=0 then 0 else (S(to_nat (N-1)))).

Fml < let eucl_int = [N,M]
Fml <   (match (eucl_dev (to_nat N)(to_nat M)) with
Fml <     (divex q r)->(pair (to_int q)(to_int r))
Fml <   end match).

Fml < Drop.

ml #extract_caml();;
       ....
ml #eucl_int 3 3;;
(pair_C (1,0)) : (int,int) prod

ml #eucl_int 150 32131;;
(pair_C (214,31)) : (int,int) prod
```

2.3.2 Heapsort

The file (Heap.v) contains the proof of an efficient list sorting algorithm described by Bjerner. It is an adaptation of the well-known *heapsort* algorithm to functional languages.

We start with loading the files:

```
ml #V "Lists";;
      ....
ml #V "Heap";;
```

As we saw in 2.2.9, we have to instantiate or realize by hand some of the Coq variables.

```
ml #fml();;

Fml < Free Vars.

*** [False_rec :(P:Set)P]

*** [eq_rec :(A:Set)A->(P:Set)P->A->P]

*** [A :Set]

*** [inf_total :A->A->sumbool]

*** [carac :A->A->nat]
```

The variables *False_rec* and *eq_rec* come from the prelude and are automatically realized by the system during the extraction. So we have to worry about *A* and *inf_total*. *carac* is not relevant for the computation and it is not necessary to realize it, yet for rigorous readers we give a possible instantiation.

```
Fml < Attach A Int.

Fml < Realize inf_total [x,y] if x<y then left else right.

Fml < Realize carac [x,y] if x=y then 0 else (S 0).

Fml < Extract.

Fml < Write CAML File heapsort.
```

If we want to test the program directly, we can define the list of the first n integers in reverse order:

```
Fml < let rec inter = [n](if n=0 then nil else (cons n (inter (n-1)))).

Fml < Drop.

ml #extract_caml();;
      ....
ml #timers true;;
      ....
ml #heapsort (inter 3000);;
(cons_C
 (1,
  (cons_C
   (2,
    (cons_C
     (3,
      ( ... ...

Evaluation has needed: Runtime: 1.37s GC: 3.34s Conses: 0
```

This shows the program is quite efficient for CAML code [¶].

2.3.3 Mergesort

This program is very similar to *heapsort*, yet the mergesort algorithm used here is slightly better adapted to functional languages. The idea is quite simple:

- One breaks the list to be sorted into a list of lists, each containing just one element. For example

$$[1;3;6;2;4;9] \mapsto [[1];[3];[6];[2];[4];[9]]$$

- The elements of this list of lists are merged together, two by two, until one gets a single sorted list. Here:

$$[[1]\ ;\ [3]\ ;\ [6];\ [2]\ ;[4];[9]] \mapsto [[1;3];[2;6];[4;9]]$$

[¶] These times were obtained on a DecStation.

$$[[1;3];\ [2;6];\ [4;9]] \mapsto [[1;2;3;6];[4;9]]$$
$$[[1;2;3;6];\ [4;9]] \mapsto [[1;2;3;4;6;9]]$$

One can see that the kind of recursion used over lists is not exactly structural recursion, but rather well-founded recursion over the length of the list. Rather than proving this recursion scheme using structural recursion, we here choose to axiomatize it, and realize it afterwards using the construct *let rec* ....

The user may look at the file `wf.v` to see how this is formalized at the logical level. The computational axiom is:

```
Axiom Acc_rec : (A:Set)
                (R:A->A->Prop)
                 (P:A->Set)
                  ((x:A)
                    ((y:A)(R y x)->(Acc A R y))->
                     ((y:A)(R y x)->(P y))->(P x))
                  ->(a:A)(Acc A R a)->(P a)
```

We see the corresponding F_ω type is:

```
(A:Set)(P:Set)A->(A->(A->P)->P)->P.
```

So `Mergesort.v` will be used exactly as `Heap.v` with one more realisation :

```
Fml < Realize Acc_rec [f,x](let rec F = [y](f y F) in (F x)).
```

The computational behavior of the resulting program is quite satisfactory and can be compared with hand-written code:

```
ml #mergesort (inter 3000);;
(cons_C
 (1,
  (cons_C
   (2,
    (cons_C
     (3,
      (cons_C
       (4,
        (cons_C
  ....
Evaluation has needed: Runtime: 0.46s GC: 1.23s
```

2.3.4 Higman's lemma

Originally this proof was not meant to be used as a program. It is an A-translated version of the classical proof of this well-known combinatorial result. The resulting program is deliberately "monstrous". Details about how it can be used are given in the file **`Higman_extractor.ml`**. We just mention this example, as it is an interesting example of a program which is proven correct, and yet the underlying algorithm has not been exactly understood!

Chapter 3

Examples

- The files `Prelude.v` and `Specif.v` contain the basic definitions of logical connectors and equality.
- The file `Nat.v` contains basic results on natural numbers.

These three files are automatically loaded in the initial state of the system.

- The file `List.v` contains basic results on lists.
- The file `Ack.v` contains a development of Ackermann's function.
- The file `Euclid.v` contains a development of euclidian divison.
- The file `Tarski.v` contains a proof of Tarski's theorem.
- The file `Tarski1.v` contains the same proof but developd using the theorem proving facilities.
- The file `Mergesort.v` contains a development of a sorting algorithm using general recursion.
- The file `Shuffle.v` contains a proof of the Gilbreath trick.
- The files `Sch_Set.v` and `Schroeder.v` contain bases of set theory and a proof of Schrœder-Bernstein theorem.

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*          Prelude :  Logical connectives, quantifiers, equality        *)
(*                                                                       *)
(*************************************************************************)

Chapter Prelude.

Section Logic.

Syntax I "<_>Id".
Inductive Proposition True = I : True.

Section Negation.

(* Absurdity *)

Inductive Proposition False =.

(* Negation *)

Definition not [A:Prop]A->False.
 Syntax not "~_".

End Negation.

Section Conjunction.

(* Pairing / Introduction *)

 Syntax and "_/\\_".

Syntax conj "<_,_>{_,_}".
Inductive Connective and [A,B:Prop] = conj : A->B->(A/\B).

Subsection Projections.

Propositions A,B.

Theorem proj1.
Statement (A/\B)->A.
Proof (and_ind A B A [y:A][z:B]y).

Theorem proj2.
Statement (A/\B)->B.
Proof (and_ind A B B [y:A][z:B]z).

End Projections.

Syntax proj1 "<_,_>Fst{_}".
Syntax proj2 "<_,_>Snd{_}".

End Conjunction.
```

```
Section Disjunction.

Syntax or "_\\/_".
Inductive Connective or [A,B:Prop]
    = or_introl : A -> (A\/B) | or_intror : B -> (A\/B).

End Disjunction.

Definition IF = [P,Q,R:Prop](P /\ Q) \/ ((~P) /\ R).
Syntax IF "if _ then _ else ".

(* First-order quantifiers *)

Definition all.
Body [A:Set][P:A->Prop](x:A)(P x).
Syntax all "<_>All(_)".

Syntax ex "<_>Ex(_)".
Inductive Definition ex [A:Set;P:A->Prop] : Prop
    = ex_intro : (x:A)(P x)->(<A>Ex(P)).

Syntax ex2 "<_>Ex2(_,_)".
Inductive Definition ex2 [A:Set;P,Q:A->Prop] : Prop
    = ex_intro2 : (x:A)(P x)->(Q x)->(<A>Ex2(P,Q)).

(* Equality *)

Syntax eq "<_>_=_".
Inductive Definition eq [A:Set;x:A] : A->Prop
    = refl_equal : <A>x=x.


End Logic.

(*****************************************************************************)
(*                                                                           *)
(*                    Basic programming with Set                             *)
(*                                                                           *)
(*****************************************************************************)

(************************)
(* Programming Language *)
(************************)

(* Basic sets *)

Inductive Set unit = tt : unit.

Inductive Set bool = true : bool | false : bool.

Inductive Set nat = O : nat | S : nat->nat.

(* Disjoint sum of two sets *)
```

```
Syntax sum "_+_".
Inductive Set sum [A,B:Set]
    = inl : A -> (A+B) | inr : B -> (A+B).

(* Product of Sets *)

Syntax prod  "_*_".
Syntax pair "<_,_>(_,_)".
Inductive Set prod [A,B:Set] = pair : A->B->(A*B).

Section programming.
   Variables A,B:Set.
   Variable x:A*B.
Theorem fst.
Statement (A*B)->A.
Proof [u:A*B](<A>Match u with (* y,z *) [y:A][z:B]y).

Theorem snd.
Statement (A*B)->B.
Proof [u:A*B](<B>Match u with (* y,z *) [y:A][z:B]z).
End programming.

Syntax fst "<_,_>Fst(_)".
Syntax snd "<_,_>Snd(_)".

Section Prelude_lemmas.

Goal (A:Prop)(C:Prop)A->(~A)->C.
Unfold not ; Intros A C h1 h2.
(*    h2 : A->False
      h1 : A
      C : Prop
      A : Prop
      subgoal  C *)
Elim (h2 h1).
Save absurd.

Section equality.
 Variable A,B : Set.
 Variable f : A->B.
 Variable x,y,z : A.

 Goal (<A>x=y) -> <A>y=x.
 Intros h ; Elim h.
 (*    h : <A>x=y
       subgoal <A>x=x *)
 Apply refl_equal.
 Save sym_equal.

 Goal (<A>x=y) -> (<A>y=z) -> <A>x=z.
 Intros h1 h2 ; Elim h2 ; Apply h1.
 Save trans_equal.

 Goal (<A>x=y)-><B>(f x)=(f y).
 Intros h ; Elim h.
```

```
(*    h : <A>x=y
      subgoal  <B>(f x)=(f x) *)
Apply refl_equal.
Save f_equal.

End equality.

Section Properties_of_Relations.
 Variable A  : Set.
 Variable R  : A->A->Prop.

     Definition refl.
     Body (x:A)(R x x).
     Definition trans.
     Body (x,y,z:A)(R x y) -> (R y z) ->(R x z).
     Definition sym.
     Body (x,y:A)(R x y) -> (R y x).
     Definition equiv.
     Body refl /\ trans /\ sym.

End Properties_of_Relations.

End Prelude_lemmas.

End Prelude.

Hint I conj or_introl or_intror pair inl inr refl_equal.
Immediate sym_equal.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*     Basic specifications : Sets containing logical information        *)
(*                                                                       *)
(*************************************************************************)

(*************************************************************)
(* Basic specifications : Sets containing logical information *)
(*************************************************************)

Inductive Set sig [A:Set;P:A->Prop]
    = exist : (x:A)(P x)->{x:A|(P x)}.

Inductive Set sig2 [A:Set;P,Q:A->Prop]
    = exist2 : (x:A)(P x)->(Q x)->{x:A|(P x)&(Q x)}.

Inductive Set sigS [A:Set;P:A->Set]
    = existS : (x:A)(P x)->{x:A & (P x)}.

Inductive Set sigS2 [A:Set;P,Q:A->Set]
    = existS2 : (x:A)(P x)->(Q x)->{x:A & (P x) & (Q x)}.

Syntax sumbool "{_}+{_}".
Inductive Set sumbool [A,B:Prop]
    = left : A ->({A}+{B}) | right : B->({A}+{B}).

Syntax sumor "_+{_}".
Inductive Set sumor [A:Set;B:Prop]
    = inleft : A -> (A+{B}) | inright : B -> (A+{B}).

(*******************************)
(* Self realizing propositions *)
(*******************************)

Axiom False_rec : (P:Set)False->P.
Definition except = False_rec. (* for compatibility with previous versions *)


Goal (A:Prop)(C:Set)A->(~A)->C.
Intros A C h1 h2.
(*    h2 : ~A
      h1 : A
      C : Prop
      A : Prop
      subgoal  C *)
Apply False_rec.
Apply (h2 h1).
Save absurd_set.

Goal (A,B:Prop)(C:Set)(A->B->C)->(A/\B)->C.
Intros A B C F AB; Apply F; Elim AB; Auto.
Save and_rec.
```

```
Axiom eq_rec : (A:Set)(a:A)(P:A->Set)(P a)->(b:A)(<A>a=b)->(P b).

(* For compatibility with previous versions *)

Definition eq_spec = [A:Set][a,b:A][H:<A>a=b][P:A->Set][H':(P a)]
                     (eq_rec A a P H' b H).

Hint left right inleft inright.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                            Natural numbers                            *)
(*                                                                       *)
(*************************************************************************)

Chapter Natural_numbers.

Inductive Definition le [n:nat] : nat -> Prop
    = le_n : (le n n)
    | le_S : (m:nat)(le n m)->(le n (S m)).

Hint le_n le_S.

Section less_or_equal.
Variable n,m,p:nat.

Goal (le n m)->(le (S n) (S m)).
Intro H.
(*  (le (S n) (S m))
    ============================
      H : (le n m) *)
Elim H ; Auto.
Save le_n_S.

Goal (le n m)->(le m p)->(le n p).
Intros H H0.
(*  (le n p)
    ============================
      H0 : (le m p)
      H : (le n m) *)
Elim H0 ; Auto.
Save le_trans.

Goal (le n (S n)).
Auto.
Save le_n_Sn.

Goal (le 0 n).
Elim n ; Auto.
Save le_0_n.

End less_or_equal.
Hint le_n_S le_n_Sn le_0_n.

Global pred : nat->nat
     = [n:nat](<nat>Match n with (* 0 *) 0
                                 (* S u *) [u,v:nat]u).

Goal (m:nat)<nat>(pred (S m))=m.
Auto.
Save n_predSn.
```

```
Goal (n:nat)(le (pred n) n).
Induction n ; Auto.
Save le_pred_n.
Hint le_pred_n.

Goal (n,m:nat)(le (S n) m)->(le n m).
Intros n m H ; Apply le_trans with (S n) ; Auto.
Save le_trans_S.
Immediate le_trans_S.

Goal (n,m:nat)(le (S n) (S m))->(le n m).
Intros n m H ; Change (le (pred (S n)) (pred (S m))).
(*  (le (pred (S n)) (pred (S m)))
    ============================
      H : (le (S n) (S m))
      m : nat
      n : nat *)
Elim H ; Simpl ; Auto.
Save le_S_n.
Immediate le_S_n.

Local abs_prop : nat->Prop
  = [n:nat](<Prop>Match n with (* O *) False
                               (* S p *) [p:nat][P:Prop]True).

Goal (n:nat)~(le (S n) O).
Red ; Intros n H.
(*  False
    ============================
      H : (le (S n) O)
      n : nat *)
Change (abs_prop O) ; Elim H ; Simpl ; Auto.
Save le_Sn_O.
Hint le_Sn_O.

Goal (n:nat)~(le (S n) n).
Intro n ; Elim n ; Auto.
(*  (y:nat)(~(le (S y) y))->(~(le (S (S y)) (S y)))
    ============================
      n : nat *)
Intros y H.
Red ; Intro ; Apply H ; Auto.
Save le_Sn_n.
Hint le_Sn_n.

Local abs_prop : nat->Prop
  = [n:nat](<Prop>Match n with (* O *) True
                               (* S p *) [p:nat][P:Prop]False).

Goal (n:nat)~(<nat>O=(S n)).
Red ; Intros n H.
(*  False
    ============================
      H : <nat>O=(S n)
      n : nat *)
```

```
Change (abs_prop (S n)) ; Elim H ; Simpl ; Auto.
Save O_S.
Hint O_S.

Goal (n,m:nat)(<nat>n=m)->(<nat>(S n)=(S m).
Intros n m H ; Apply (f_equal nat) ; Auto.
Save eq_S.
Hint eq_S.

Goal (n,m:nat)(<nat>(S n)=(S m))->(<nat>n=m.
Intros n m H ; Change <nat>(pred (S n))=(pred (S m)).
(*  <nat>(pred (S n))=(pred (S m))
    ============================
      H : <nat>(S n)=(S m)
      m : nat
      n : nat *)
Apply (f_equal nat) ; Auto.
Save eq_add_S.
Immediate eq_add_S.

Goal (n:nat)~(<nat>n=(S n).
Intro n ; Elim n ; Auto ; Unfold not ; Auto.
Save n_Sn.
Hint n_Sn.

Definition gt [n,m:nat]~(le n m).

Goal (n:nat)(gt (S n) O).
Exact le_Sn_O.
Save gt_Sn_O.
Hint gt_Sn_O.

Goal (n:nat)(gt (S n) n).
Exact le_Sn_n.
Save gt_Sn_n.
Hint gt_Sn_n.

Goal (n,m:nat)(le n m)->(le m n)->(<nat>n=m.
Intros n m h ; Elim h ; Auto.
(*  (m:nat)(le n m)->((le m n)->(<nat>n=m))->(le (S m) n)->(<nat>n=(S m))
    ============================
      h : (le n m)
      m : nat
      n : nat *)
Intros m0 H H0 H1.
Absurd (le (S m0) m0) ; Auto.
(*  (le (S m0) m0)
    ============================
      H1 : (le (S m0) n)
      H0 : (le m0 n)->(<nat>n=m0)
      H : (le n m0)
      m0 : nat *)
Apply le_trans with n ; Auto.
Save le_antisym.
Immediate le_antisym.
```

```
Goal (n:nat)(le n O)-><nat>O=n.
Auto.
Save le_n_O_eq.
Immediate le_n_O_eq.

Goal (n,m:nat)(le (S n) m)->(gt m n).
Intros n m H ; Elim H ; Auto ; Unfold gt not ; Auto.
Save le_S_gt.
Immediate le_S_gt.

Goal (n,m:nat)(gt n m)->(gt (S n) (S m)).
Unfold gt not ; Intros n m H H0.
(*  False
    ============================
      H0 : (le (S n) (S m))
      H : (le n m)->False
      m : nat
      n : nat *)
Apply H ; Auto.
Save gt_n_S.
Hint gt_n_S.


Goal (n,m:nat){(le n m)}+{(gt n m)}.
Induction n ; Auto ; Intros y H m.
(*  {(le (S y) m)}+{(gt (S y) m)}
    ============================
      m : nat
      H : (m:nat)({(le y m)}+{(gt y m)})
      y : nat
      n : nat *)
Elim m ; Auto ; Intros y0 H0.
(*  {(le (S y) (S y0))}+{(gt (S y) (S y0))}
    ============================
      H0 : {(le (S y) y0)}+{(gt (S y) y0)}
      y0 : nat *)
Elim (H y0) ; Auto.
Save le_or_gt.

Goal (n,m:nat)((le n m)\/(le (S m) n)).
Intro n ; Elim n ; Auto ; Intros y H m.
(*  (le (S y) m)\/(le (S m) (S y))
    ============================
      m : nat
      H : (m:nat)((le y m)\/(le (S m) y))
      y : nat
      n : nat *)
Elim m ; Auto ; Intros y0 H0.
(*  (le (S y) (S y0))\/(le (S (S y0)) (S y))
    ============================
      H0 : (le (S y) y0)\/(le (S y0) (S y))
      y0 : nat *)
Elim (H y0) ; Auto.
Save le_or.
```

```
Goal (n,m,p:nat)(gt (S n) m)->(gt m p)->(gt n p).
Unfold gt ; Red ; Intros n m p H H0 H1.
(*  False
    ============================
      H1 : (le n p)
      H0 : ~(le m p)
      H : ~(le (S n) m)
      p : nat
      m : nat
      n : nat *)
Absurd (le m p) ; Auto.
(*  (le m p) *)
Apply le_trans with n ; Auto.
(*  (le m n) *)
Elim (le_or m n) ; Auto ; Intro H2.
(*  (le m n)
    ============================
      H2 : (le (S n) m) *)
Absurd (le (S n) m) ; Auto.
Save gt_trans_S.

Goal (n,m,p:nat)(le m n)->(gt m p)->(gt n p).
Unfold gt not ; Intros n m p H H0 H1.
(*  False
    ============================
      H1 : (le n p)
      H0 : (le m p)->False
      H : (le m n)
      p : nat
      m : nat
      n : nat *)
Apply H0 ; Apply le_trans with n ; Auto.
Save le_gt_trans.

Goal (n,m,p:nat)(gt n m)->(le p m)->(gt n p).
Unfold gt not ; Intros n m p H H0 H1.
(*  False
    ============================
      H1 : (le n p)
      H0 : (le p m)
      H : (le n m)->False
      p : nat
      m : nat
      n : nat *)
Apply H ; Apply le_trans with p ; Auto.
Save gt_le_trans.

Goal (n:nat)({m:nat|<nat>(S m)=n})+{<nat>0=n}.
Intro n ; Elim n ; Auto ; Intros y H.
(*  ({m:nat|<nat>(S m)=(S y)})+{<nat>0=(S y)}
    ============================
      H : ({m:nat|<nat>(S m)=y})+{<nat>0=y}
      y : nat
      n : nat *)
```

```
Left; Exists y ; Auto.
Save O_or_S.

Goal (n,m:nat)(gt (S n) m)->((gt n m)\/(<nat>m=n)).
Intros n m H ; Elim (le_or_gt n m) ; Auto ; Intro a.
(*  (gt n m)\/<nat>m=n
    ============================
      a : (le n m)
      H : (gt (S n) m)
      m : nat
      n : nat *)
Elim (le_or m n) ; Auto ; Intro HO.
(*  (gt n m)\/<nat>m=n
    ============================
      HO : (le (S n) m) *)
Absurd (le (S n) m) ; Auto.
Save gt_S.

Goal (n:nat)((gt n O)\/(<nat>O=n)).
Intro n ; Apply gt_S ; Auto.
Save gt_O_eq.

Goal (n:nat)(P:nat->Prop)(P O)->((m:nat)(P (S m)))->(P n).
Intros n P H HO ; Elim n ; Auto.
Save nat_case.

End Natural_numbers.

Hint le_n le_S le_n_S le_n_Sn le_O_n le_pred_n le_Sn_O le_Sn_n O_S eq_S
     n_Sn gt_Sn_O gt_Sn_n gt_n_S.
Immediate le_S_n.
Immediate le_trans_S.
Immediate eq_add_S.
Immediate le_S_gt.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                              Lists                                    *)
(*                                                                       *)
(*************************************************************************)


(* Some programs and results about lists *)

Chapter Lists.

Variable  A:Set.

Inductive Set list = nil : list | cons : A -> list -> list.


Global app.
Body [l,m:list](<list>Match l with (* nil *) m
                                (* cons a m *) [a:A][m:list](cons a))
   : list->list->list.


Goal (l:list)<list>l=(app l nil).
Intro l ; Elim l ; Simpl ; Auto.
(*  (a:A)(y:list)
     (<list>y=(app y nil))->(<list>(cons a y)=(cons a (app y nil)))
    ============================
      l : list *)
Intros a y h ; Elim h ; Auto.
Save app_nil_end.
Hint app_nil_end.

Goal (l,m,n : list)<list>(app (app l m) n)=(app l (app m n)).
Intros l m n ; Elim l ; Simpl ; Auto.
(*  (a:A)(y:list)(<list>(app (app y m) n)=(app y (app m n)))->
     (<list>(cons a (app (app y m) n))=(cons a (app y (app m n))))
    ============================
      n : list
      m : list
      l : list *)
Intros a y h ; Elim h ; Auto.
Save app_ass.
Hint app_ass.

Goal (l,m,n : list)<list>(app l (app m n))=(app (app l m) n).
Auto.
Save ass_app.
Hint ass_app.

Definition tail.
Body [l:list](<list>Match l with
                 (* nil *) nil
            (* cons a m *) [a:A][m,t:list]m) : list->list.
```

```
Local abs_prop =
    [l:list](<Prop>Match l with (* nil *) True
                           (* cons a m *) [a:A][m:list][P:Prop]False).

Goal (a:A)(m:list)~<list>nil=(cons a m).
Unfold not ; Intros a m h.
(*  False
    ============================
      h : <list>nil=(cons a m)
      m : list
      a : A *)
Change (abs_prop (cons a m)).
        Elim h ; Simpl ; Auto.
Save nil_cons.

(*****************************************)
(* Length of lists                       *)
(*****************************************)

Definition length.
Body [l:list](<nat>Match l with (* nil *) O
                           (* cons a m *) [a:A][m:list]S)
   : list->nat.

(*******************************)
(* Length order of lists       *)
(*******************************)

Section length_order.
Definition lel [l,m:list](le (length l) (length m)).

Variables a,b:A.
Variable l,m,n:list.

Goal (lel l l).
Unfold lel ; Auto.
Save lel_refl.

Goal (lel l m)->(lel m n)->(lel l n).
Unfold lel ; Intros.
(*  (le (length l) (length n))
    ============================
      H0 : (le (length m) (length n))
      H : (le (length l) (length m)) *)
        Apply le_trans with (length m) ; Auto.
Save lel_trans.

Goal (lel l m)->(lel (cons a l) (cons b m)).
Unfold lel ; Simpl ; Auto.
Save lel_cons_cons.

Goal (lel l m)->(lel l (cons b m)).
Unfold lel ; Simpl ; Auto.
Save lel_cons.
```

```
Goal (lel (cons a l) (cons b m)) -> (lel l m).
Unfold lel ; Simpl ; Auto.
Save lel_tail.

Goal (l:list)(lel l nil)-><list>nil=l.
Intro l ; Elim l ; Auto.
Intros a y H H0.
(*  <list>nil=(cons a y)
    ============================
      H0 : (lel (cons a y) nil)
      H : (lel y nil)->(<list>nil=y)
      y : list
      a : A
      l : list *)
Absurd (le (S (length y)) 0); Auto.
Save lel_nil.
End length_order.

Hint lel_refl lel_cons_cons lel_cons lel_nil lel_nil nil_cons.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                    Variations on Ackermann's function                 *)
(*                                                                       *)
(*************************************************************************)

Inductive Definition Ack : nat->nat->nat->Prop =
     Ack0 : (n:nat)(Ack 0 n (S n))
   | Ackn0 : (n,p:nat)(Ack n (S 0) p)->(Ack (S n) 0 p)
   | AckSS : (n,m,p,q:nat)
             (Ack (S n) m q)->(Ack n q p)->(Ack (S n) (S m) p).

Hint Ack0 Ackn0.

Goal (n,m:nat){p:nat|(Ack n m p)}.
Induction n.
Intro m; Exists (S m); Auto.
Induction m.
Elim (H (S 0));Intros.
Exists x; Auto.
Intros m' H'; Elim H';Intros.
Elim (H x); Intros.
Exists x0.
Apply AckSS with x; Auto.
Save Ackermann.

(* Functional definition of Ackermann :
 (ack 0 n) = (S n)
 (ack (S n) 0) = (ack n (S 0))
 (ack (S n) (S m)) = (ack n (ack (S n) m)) *)

Definition ack =
   [n:nat](<nat->nat> Match n with
          (* 0 *) S
       (* (S n) *) [n:nat][f:nat->nat][m:nat]
                   (<nat> Match m with
                   (* 0 *) (f (S 0))
                 (* S m *) [m:nat][r:nat](f r))).

Goal (n,m,p:nat)(Ack n m p)-><nat>p=(ack n m).
Induction 1; Simpl; Trivial.
Intros n1 m1 p1 q1 ASn Eq An Ep; Elim Eq; Elim Ep; Trivial.
Save ack_Ack.

Goal (n,m:nat)(Ack n m (ack n m)).
Induction n.
Simpl; Auto.
Intros n' H; Induction m.
Simpl; Auto.
Intros; Apply AckSS with (ack (S n') y); Auto.
Apply (H (ack (S n') y)).
Save Ack_ack.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                          Euclidian division                           *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                       Christine PAULIN-MOHRING                        *)
(*                                                                       *)
(*************************************************************************)

(* An elimination principle for the order on natural numbers *)

Goal (P:nat->nat->Prop)
     ((p:nat)(P O p))->
     ((p,q:nat)(le p q)->(P p q)->(P (S p) (S q)))->
     (n,m:nat)(le n m)->(P n m).
Induction n; Trivial.
Intros n' HRec m Le.
Elim Le; Auto.
Save le_elim_rel.

(* difference between two natural numbers *)
Definition minus = [n:nat](<nat->nat> Match n with
                              [p:nat]O
                              [k:nat][mink:nat->nat][p:nat]
                              (<nat> Match p with
                                     (S k)
                                     [l:nat][minSkl:nat](mink l))).
(* arithmetical lemmas *)
Goal (n,m:nat)<nat>(minus n m)=(minus (S n) (S m)).
Auto.
Save min_Sn_Sm.

Goal (n:nat)<nat>n=(minus n O).
Induction n; Simpl; Auto.
Save min_n_O.
Hint min_n_O.

Goal (n:nat)<nat>O=(minus n n).
Induction n; Simpl; Auto.
Save min_n_n.
Hint min_n_n.

(* addition and multiplication *)
Definition plus  = [n,m:nat](<nat> Match n with m [p:nat]S).
Definition mult = [n,m:nat](<nat> Match n with O [p:nat](plus m)).

Goal (n,m,p:nat)<nat>(plus n (plus m p))=(plus (plus n m) p).
Intros n m p ; Elim n ; Simpl ; Auto.
Save plus_associatif.
Hint plus_associatif.

Goal (n,m,p:nat)<nat>(plus (plus n m) p)=(plus n (plus m p)).
```

```
Auto.
Save associatif_plus.

Goal (n:nat)<nat>n=(plus n O).
Intro n ; Elim n ; Simpl ; Auto.
Save plus_n_O.
Hint plus_n_O.

Goal (n,m:nat)<nat>(S (plus n m))=(plus n (S m)).
Intros n m ; Elim n ; Simpl ; Auto.
Save plus_S.
Hint plus_S.

Goal (n,m:nat)<nat>(plus n (S m))=(S (plus n m)).
Auto.
Save n_Sm_plus.

Goal (n,m:nat)<nat>(plus n m)=(plus m n).
Intros n m ; Elim n ; Simpl ; Auto.
Intros y H ; Elim (plus_S m y) ; Auto.
Save plus_sym.
Immediate plus_sym.

Goal (n,m,p:nat)(<nat>(plus n m)=(plus n p))-><nat>m=p.
Intros n m p ; Elim n ; Simpl ; Auto.
Save plus_simpl.

Goal (p:nat)(gt p O)-><nat>p=(S (pred p)).
Intro p ; Elim p ; Auto.
Intro H.
Absurd (le O O) ; Auto.
Save S_pred.
Hint S_pred.

Goal (n,m:nat)(le n m)-><nat>m=(plus n (minus m n)).
Intros n m Le; Pattern n m; Apply le_elim_rel ;Simpl ; Auto.
Save plus_minus.
Hint plus_minus.

Goal (n,m:nat)(le m n)->(gt m O)->(gt n (minus n m)).
Intros n m Le; Pattern m  n; Apply le_elim_rel ;Simpl ; Trivial.
Intros; Absurd (le O O); Auto.
Intros p q lepq Hp gtp.
Elim (gt_O_eq p); Intro H.
Apply le_gt_trans with q; Auto.
Elim H; Elim min_n_O; Auto.
Save  gt_minus.
Hint gt_minus.

(* The induction principle *)

Goal (n:nat)(P:nat->Set)((n:nat)((m:nat)(gt n m)->(P m))->(P n))->(P n).
Intros n P H.
Cut (m:nat)(gt n m)->(P m) ; Auto.
Elim n ; Simpl ; Auto.
```

```
Intros m H0.
Absurd (le 0 m) ; Auto.
Intros y H0 m H1.
Apply H ; Intros m0 H2.
Apply H0.
Apply gt_trans_S with m ; Auto.
Save gt_wf.

(* The specification *)

Inductive Definition diveucl [a,b:nat] : Set
      = divex : (q,r:nat)(<nat>a=(plus (mult q b) r))->(gt b r)->(diveucl a b).

Goal (b:nat)(gt b 0)->(a:nat)(diveucl a b).
Intros b H a ; Pattern a; Apply gt_wf ; Intros n H0.
Elim (le_or_gt b n).
Intro lebn.
Elim (H0 (minus n b)); Auto.
Intros q r e g.
Apply divex with (S q) r ; Simpl ; Auto.
Elim plus_associatif.
Elim e ; Auto.
Intros gtbn.
Apply divex with 0 n ; Simpl ; Auto.
Save eucl_dev.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                 Tarski's Theorem in Basic Vernacular                  *)
(*                                                                       *)
(*************************************************************************)

Parameter A:Set.

Parameter R:A->A->Prop.

Parameter Eq:A->A->Prop.

Axiom Assym.
Assumes (x:A)(y:A)((R x y) -> (R y x) -> (Eq x y)).

Axiom Trans.
Assumes (x:A)(y:A)(z:A)((R x y) -> (R y z) -> (R x z)).

Parameter f:A->A.

Axiom Incr.
Assumes (x:A)(y:A)((R x y) -> (R (f x) (f y))).

Definition Lub.
Body [m:A][S:A->Prop](and ((x:A) ((S x) -> (R x m)))
                          ((y:A)  (((x:A) ((S x)->(R x y))) -> (R m y)))).

Axiom Complete.

Assumes (S:A->Prop)(<A> Ex ([x:A] (Lub x S))).

Theorem Tarski.

Statement <A> Ex ([x:A](Eq x (f x))).

   Local Under.
   Body [x:A](R x (f x)).

   Remark Exist_lub_under.
   Statement <A> Ex ([m:A] (Lub m Under)).
   Proof (Complete Under).

   Remark Tarski1.

   Statement ((M:A) ((Lub M Under) -> <A> Ex  ([m:A] (Eq m (f m))))).

      Variable M:A.

      Hypothesis LeastUp.
      Assumes (Lub M Under).

      Remark Up.
      Statement (x:A) ((R x (f x)) -> (R x M)).
```

```
Proof (proj1 ((x:A) ((R x (f x)) -> (R x M)))
             ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x)))
             LeastUp).

Remark Least.
Statement (x:A)(((y:A)(R y (f y))->(R y x))->(R M x)).
Proof (proj2 ((x:A) ((R x (f x)) -> (R x M)))
             ((x:A)(((y:A)(R y (f y))->(R y x))->(R M x)))
             LeastUp).


Remark One.

Statement (y:A)(Under y)->(R y (f M)).

   Variable y:A.

   Hypothesis v.
   Assumes (Under y).

   Remark T.
   Statement (R y M).
   Proof (Up y v).

   Remark T'.
   Statement (R (f y) (f M)).
   Proof (Incr y M T).

Proof (Trans y (f y) (f M) v T').

Remark Two.
Statement (R M (f M)).
Proof (Least (f M) One).

Remark Three.
Statement (R (f M) (f (f M))).
Proof (Incr M (f M) Two).

Remark Four.
Statement (R (f M) M).
Proof (Up (f M) Three).

Remark Five.
Statement (Eq M (f M)).
Proof (Assym M (f M) Two Four).

Proof (ex_intro A ([m:A] (Eq m (f m))) M Five).


Proof (ex_ind A ([M:A](Lub M Under)) (<A>Ex([x:A](Eq x (f x))))
             Tarski1 Exist_lub_under).
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*           Tarski's Theorem with theorem proving facilities            *)
(*                                                                       *)
(*************************************************************************)


Set A.

Variable R:A->A->Prop.

Variable Eq:A->A->Prop.

Axiom Assym.
   Variable x,y:A.
Assume (R x y) -> (R y x) -> (Eq x y).

Axiom Trans.
   Variable x,y,z:A.
Assume (R x y) -> (R y z) -> (R x z).

Variable f:A->A.

Axiom Incr.
   Variable x,y:A.
Assume (R x y) -> (R (f x) (f y)).

Variable M:A.

Hypothesis Up.
   Variable x:A.
Assume (R x (f x)) -> (R x M).

Hypothesis Least.
   Variable x:A.
Assume ((y:A)(R y (f y))->(R y x))->(R M x).

Goal (Eq M (f M)).
Cut (R M (f M)).
Intro.
Apply Assym; Trivial.
Apply Up.
Apply Incr; Trivial.
Apply Least.
Intros.
Apply Trans with (f y); Trivial.
Apply Incr.
Apply Up; Trivial.
Save Tarski_lemma.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                          MERGESORT for lists                          *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                           Benjamin WERNER                             *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                     BEWARE : GENERAL RECURSION                        *)
(*                                                                       *)
(*************************************************************************)

(*  realizes well_founded_recursion by :                                 *)
(*   [x,f](let rec F = [y] (f y  F) in (F x))                            *)

(* Load List. *)

Variable   inf : A -> A -> Prop.

Hypothesis inf_total : (x,y:A){(inf x y)}+{(inf y x)}.
Hypothesis inf_tran : (x,y,z:A)(inf x y)->(inf y z)->(inf x z).
Hypothesis inf_refl : (x:A)(inf x x).

(*******************************)
(* addition of natural numbers *)
(*******************************)

Definition add [n,m:nat](<nat> Match m with
                              (* O *) n
                            (* S p *) [p:nat][q:nat](S q)).

Goal (n:nat)(<nat> (add O n)=n).
Induction n; Simpl; Auto.
Save eq_ad_O.

Hint eq_ad_O.

Goal (n:nat)(m:nat)(<nat>(add n m)=(add m n)).
Induction n; Simpl; Auto.
Intros.
Elim (H m).
Elim m; Simpl; Auto.
Save sym_add.

Hint sym_add.

Goal (n,m,p:nat)<nat>(add n (add m p))=(add (add n m) p).
Induction p; Intros; Simpl; Auto.
Save add_ass.

Hint add_ass.
```

```
Goal (n,m,p:nat)<nat>(add (add n m) p)=(add n (add m p)).
Auto.
Save ass_add.

Hint ass_add.

Goal (n,m,p:nat)(<nat>n=m)-><nat>(add n p)=(add m p).
Induction p;Intros;Simpl;Auto.
Save add_simpl_d.

Hint add_simpl_d.


Goal (n,m,p:nat)(<nat>n=m)-><nat>(add p n)=(add p m).
Intros.
Elim (sym_add n p).
Elim (sym_add m p).
Auto.
Save add_simpl_g.

Hint add_simpl_g.

Goal (n,m,p,q:nat)(<nat>n=m)->(<nat>p=q)-><nat>(add n p)=(add m q).
Intros.
Elim H.
Elim H0.
Auto.
Save add_simpl.

Hint add_simpl.

Definition eq_fun [f1:A->nat][f2:A->nat]((a:A)(<nat>(f1 a)=(f2 a))).

Definition fun_add [f1:A->nat][f2:A->nat][a:A](add (f1 a)(f2 a)).

(***************************)
(* Characteristic function *)
(***************************)

Variable carac : A->A->nat.

(***********************)
(* Contents of a list *)
(***********************)

Definition list_content
   [l:list](<A->nat> Match l with
     (* nil *) [a:A]0
(* cons a l *) [a:A][l:list][f:A->nat](fun_add (carac a) f)).
(*****************)
(* low for lists *)
(*****************)

Inductive Definition list_Lowert [a:A] : list -> Prop =
```

```
      nil_low : (list_Lowert a nil)
    | cons_low : (b:A)(l:list)(inf a b)->(list_Lowert a (cons b l)).

Hint nil_low cons_low.


Definition list_Lowert2 =
   [a:A][l:list](<Prop> Match l with (* nil *) True
                                     (* cons b l *) [b:A][l:list][H:Prop](inf a b)).

Goal (a:A)(l:list)(list_Lowert2 a l)->(list_Lowert a l).
Induction l; Simpl; Auto.
Save low2_low.

Goal (a:A)(l:list)(list_Lowert a l)->(list_Lowert2 a l).
Induction l; Intros; Simpl; Auto.
Save low_low2.

Goal (a,b:A)(l:list)(list_Lowert a (cons b l))->(inf a b).
Intros a b l; Exact (low_low2 a (cons b l)).
Save lowert_cons_inf.

Goal (a,b:A)(l,m:list)(list_Lowert a (cons b l))->(list_Lowert a (cons b m)).
Intros a b l m H.
Apply cons_low.
Apply lowert_cons_inf with l;Auto.
Save Lowert_cons_cons.


(*************************************)
(* definition for a list to be sorted *)
(*************************************)

Inductive Definition sort : list -> Prop =
      nil_sort : (sort nil)
    | cons_sort : (a:A)(l:list)(sort l)->(list_Lowert a l)->(sort (cons a l)).
Hint nil_sort cons_sort.

Definition sort_inv
[l:list](<Prop>Match l with
     (* nil *) True
(* cons a l *) [a:A][l:list][H:Prop](sort l)/\(list_Lowert a l)).

Goal (l:list)(sort l)->(sort_inv l).
Induction l;Simpl;Auto.
Save sort_sort_inv.

Goal (P:list->Set)
        (P nil)->
         ((a:A)
           (l:list)(sort l)->(P l)->(list_Lowert a l)->(P (cons a l)))
         ->(y:list)(sort y)->(P y).
Induction y; Trivial.
Intros a l Pl Pc; Elim (sort_sort_inv (cons a l)); Auto.
Save sort_rec.
```

```
(*****************************)
(* merging two sorted lists *)
(*****************************)
Inductive Definition merge_lem [l1:list;l2:list]:Set =
merge_exist : (l:list)(sort l)->
                        (eq_fun(list_content l)
                               (fun_add(list_content l1)
                                       (list_content l2)))->
                           ((a:A)(list_Lowert a l1)->(list_Lowert a l2)->
                                  (list_Lowert a l))->
                                         (merge_lem l1 l2).


Goal (l1:list)(sort l1)->(l2:list)(sort l2)->(merge_lem l1 l2).
Induction 1;Intros.
Apply merge_exist with l2;Auto.
Unfold eq_fun fun_add;Auto.
Elim H3;Intros.
Apply merge_exist with (cons a l);Auto.
Unfold eq_fun fun_add;Auto.
Elim (inf_total a a0);Intros.
Elim (H1 (cons a0 l0)); Intros; Auto.
Apply merge_exist with (cons a l3);Auto.
Simpl; Unfold eq_fun fun_add;Intros.
Elim add_ass.
Apply add_simpl_g.
Exact (e a2).
Intros.
Apply Lowert_cons_cons with l; Auto.
Elim H5;Intros.
Apply merge_exist with (cons a0 l3).
Auto.
Simpl;Unfold eq_fun fun_add;Intros.
Elim (sym_add (list_content l3 a1) (carac a0 a1)).
Elim (sym_add (list_content l0 a1) (carac a0 a1)).
Elim ass_add.
Apply add_simpl_d.
Exact (e a1).
Intros;Apply Lowert_cons_cons with l0; Auto.
Save merge.

(*************************)
(* well-founded induction *)
(*************************)
Chapter  Well_founded.

Variable A : Set.

Inductive Definition Acc [R:A->A->Prop] : A -> Prop
   = Acc_intro : (x:A)((y:A)(R y x)->(Acc R y))->(Acc R x).

Definition well_founded = [R:A->A->Prop](a:A)(Acc R a).

Variable f : A -> nat.
```

```
Definition gtof = [a,b:A](gt (f b) (f a)).

Goal (well_founded gtof).
Red.
Cut (n:nat)(a:A)(gt n (f a))->(Acc gtof a).
Intros H a;  Apply (H (S (f a))); Auto.
Induction n.
Intros; Absurd (le 0 (f a)); Auto.
Intros m Hm a gtSma.
Apply Acc_intro.
Unfold gtof; Intros b gtfafb.
Apply Hm.
Apply gt_trans_S with (f a); Trivial.
Save well_founded_gtof.

(* to be realised by
   let  Acc_rec F = let rec wf x = F x wf in wf *)

Axiom Acc_rec : (R:A->A->Prop)
        (P:A->Set)
         ((x:A)((y:A)(R y x)->(Acc R y))->((y:A)(R y x)->(P y))->(P x))
         ->(a:A)(Acc R a)->(P a).

Goal (R:A->A->Prop)(well_founded R)->
     (P:A->Set)((x:A)((y:A)(R y x)->(P y))->(P x))->(a:A)(P a).
Intros; Apply (Acc_rec R); Auto.
Apply H.
Save well_founded_induction.



Goal (P:A->Set)((x:A)((y:A)(gtof y x)->(P y))->(P x))->(a:A)(P a).
Intros P F; Cut (n:nat)(a:A)(gt n (f a))->(P a).
Intros H a;  Apply (H (S (f a))); Auto.
Induction n.
Intros; Absurd (le 0 (f a)); Auto.
Intros m Hm a gtSma.
Apply F.
Unfold gtof; Intros b gtfafb.
Apply Hm.
Apply gt_trans_S with (f a); Trivial.
Save induction_gtof1.

Definition induction_gtof2
   : (P:A->Set)((x:A)((y:A)(gtof y x)->(P y))->(P x))->(a:A)(P a)
   = (well_founded_induction gtof well_founded_gtof).

End Well_founded.

(*****************)
(* list of lists *)
(*****************)
Inductive Definition list_list : Set =
   list_nil : list -> list_list
 | cons_list : list->list_list->list_list.
```

```
Hint list_nil list_cons.


Definition Sort =
  [L:list_list](<Prop>Match L with
                  (* list_nil l *) [l:list](sort l)
              (* cons_list l L *) [l:list][L:list_list][p:Prop]
                                   (p/\(sort l))).

Goal (L:list_list)(l:list)(Sort L)->(sort l)->(Sort (cons_list l L)).
Intros;Simpl;Auto.
Save Sort_Cons.
Hint Sort_Cons.

Goal (l:list)(Sort (list_nil l))->(sort l).
Induction 1;Auto.
Save Sort_nil.

Goal (l:list)(L:list_list)(Sort(cons_list l L))->(Sort L).
Induction 1;Auto.
Save Sort_tl.

Goal (l:list)(L:list_list)(Sort(cons_list l L))->(sort l).
Induction 1;Auto.
Save Sort_hd.

Hint Sort_tl Sort_hd Sort_nil.

Definition list_list_content =
    [ll:list_list] (<A->nat> Match ll with
       (* list_nil l *) [l:list](list_content l)
    (* cons_list l L *) [l:list][L:list_list][f:A->nat]
                                (fun_add (list_content l) f)).

Inductive Set single_lem [l:list] =
  ex_single : (L:list_list)
              (Sort L)->(eq_fun (list_content l)(list_list_content L))->
              (single_lem l).

Hint ex_single.

Goal (l:list)(single_lem l).
Induction l.
Apply ex_single with (list_nil nil);Simpl;Auto.
Unfold eq_fun;Simpl;Auto.
Intro; Intro.
Elim y;Intros.
Apply ex_single with (list_nil (cons a nil));Simpl;Auto.
Unfold eq_fun;Simpl;Auto.
Elim H0;Intros.
Apply ex_single with (cons_list (cons a nil) L);Auto.
Simpl.
Unfold eq_fun;Intros;Simpl.
Unfold fun_add;Simpl.Unfold fun_add;Simpl.
```

```
Unfold fun_add;Simpl.
Apply add_simpl_g.
Apply (e a1).
Save split.


Definition Length
[l1:list_list](<nat>Match l1 with
      (* list_nil l *) [l:list]0
   (* cons_list l L *) [l:list][L:list_list][n:nat](S n)).

Definition Le1 [L1,L2:list_list](gt (Length L2)(Length L1)).



Inductive Set Single_Lem [L:list_list] =
   Ex_single : (L1:list_list)
               (Sort L1)->
               (Sort L)->
               (eq_fun (list_list_content L1) (list_list_content L))->
               (le (Length L1)(Length L))->
                   (Single_Lem L).


Goal (n:nat)(gt (S (S n)) n).
Auto.
Save gt_SSn_n.


Goal (L:list_list)(Sort L)->(Single_Lem L).

Intro; Pattern L.
Apply (well_founded_induction list_list Le1).
Exact (well_founded_gtof list_list  Length).
Induction x.
Intros l R S.
Apply Ex_single with (list_nil l);Auto.
Unfold eq_fun;Auto.
Induction y;Intros.
Elim merge with l l0; Auto.
Intros l1 Sl1 eql1 ll1; Apply Ex_single with (list_nil l1);Auto.
Apply Sort_hd with (list_nil l0); Auto.
Apply Sort_nil; Apply Sort_tl with l; Auto.
Elim (H1 y0).
Intros L1 SL1 Sy0 eqL1 leL1.
Elim merge with l l0.
Intros l1 Sl1 eql1 ll; Apply Ex_single with (cons_list l1 L1);Simpl;Auto.
Unfold eq_fun fun_add; Intro.
Elim (eqL1 a).
Elim ass_add.
Apply add_simpl_d.
Apply (eql1 a).
Apply Sort_hd with (cons_list l0 y0); Auto.
Apply Sort_hd with y0.
Apply Sort_tl with l; Auto.
```

```
Unfold Lel; Simpl; Auto.
Apply Sort_tl with l0.
Apply Sort_tl with l; Auto.
Save constr_list.

Goal (L:list_list)(Sort L)->
           {l:list|(sort l)&(eq_fun (list_content l) (list_list_content L))}.

Intro; Pattern L.
Apply (well_founded_induction list_list Lel).
Exact (well_founded_gtof list_list  Length).
Induction x.
Intros.
Exists l;Auto.
Unfold eq_fun;Simpl;Auto.
Induction y;Intros.
Elim merge with l l0.
Intros.
Exists l1;Auto.
Apply Sort_hd with (list_nil l0); Auto.
Apply Sort_nil.
Apply Sort_tl with l; Auto.
Elim merge with l l0; Intros.
Elim (constr_list y0); Intros.
Elim (H1 (cons_list l1 L1)); Intros.
Exists x0;Auto.
Simpl; Unfold eq_fun fun_add; Intro.
Elim (e0 a).
Elim ass_add.
Replace (add (list_content l a) (list_content l0 a)) with (list_content l1 a).
Exact (q a).
Exact (e a).
Unfold Lel; Simpl.
Apply le_S_gt; Auto.
Auto.
Apply Sort_tl with l0.
Apply Sort_tl with l; Auto.
Apply Sort_hd with (cons_list l0 y0); Auto.
Apply Sort_hd with y0.
Apply Sort_tl with l; Auto.
Save mergesort_L.

Goal (l:list)
     {m:list|(sort m)&(eq_fun (list_content l) (list_content m))}.
Intro.
Elim (split l).
Intros.
Elim (mergesort_L L s).
Intros.
Exists x;Auto.
Unfold eq_fun;Intro.
Apply trans_equal with (list_list_content L a).
Exact (e a).
Apply sym_equal.
Exact (q a).
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                         The Gilbreath Trick                           *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                          Gerard Huet - May 1991                       *)
(*                                                                       *)
(*************************************************************************)

(************)
(* Booleans *)
(************)

(* Inductive Set bool = true : bool | false : bool (from Prelude) *)

Definition Is_true = [b:bool](<Prop>Match b with
    (* true  *)  True
    (* false *)  False).

Goal ~<bool>true=false.
Unfold not; Intro contr; Change (Is_true false).
Elim contr; Simpl; Trivial.
Save diff_true_false.


(************)
(* Negation *)
(************)

Definition neg = [b:bool](<bool>Match b with
    (* true *)  false
    (* false*)  true).

Goal (b:bool)<bool>(neg (neg b))=b.
Induction b; Simpl; Auto.
Save neg_intro.

Goal (b:bool)<bool>b=(neg (neg b)).
Induction b; Simpl; Trivial.
Save neg_elim.

Goal (b,b':bool)(<bool>b'=(neg b))->(<bool>b=(neg b')).
Induction b; Induction b'; Intros; Simpl; Trivial.
Save neg_sym.

Goal (b:bool)~<bool>(neg b)=b.
Induction b; Simpl; Unfold not; Intro; Apply diff_true_false; Auto.
Save no_fixpoint_neg.


(*****************)
(* Boolean words *)
```

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

Inductive Set word = empty : word
                   | bit : bool -> word -> word.

(* Remark : word ~ bool list *)

Definition Nonempty = [w:word](<Prop>Match w with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop]True).


(* word concatenation : logical definition *)
Inductive Definition conc : word -> word -> word -> Prop =
  conc_empty : (v:word)(conc empty v v)
| conc_bit   : (u,v,w:word)(b:bool)(conc u v w)->(conc (bit b u) v (bit b w)).

(* word concatenation : functional definition *)
Definition Conc = [u,v:word](<word>Match u with
    (* empty *)   v
    (* bit b w *) [b:bool][w:word][conc_w_v:word](bit b conc_w_v)).

(* Relating the two definitions; unused below *)
Goal (u,v,w:word)(conc u v w)->(<word>w=(Conc u v).
Induction 1; Simpl; Trivial.
(* (u,v,w:word)(b:bool)
   (conc u v w)->(<word>w=(Conc u v))->(<word>(bit b w)=(bit b (Conc u v))) *)
Induction 2; Trivial.
Save conc_Conc.

(* Associativity of Conc; unused below *)
Goal (u,v,w:word)(<word>(Conc u (Conc v w))=(Conc (Conc u v) w).
Induction u; Simpl; Intros; Auto.
(*  (<word>(bit b (Conc y (Conc v w)))=(bit b (Conc (Conc y v) w))
  ============================
    w : word
    v : word
    H : (v:word)(w:word)(<word>(Conc y (Conc v w))=(Conc (Conc y v) w)) *)
Elim (H v w); Trivial.
Save assoc_Conc.


(**************)
(* Singletons *)
(**************)

Definition single = [b:bool](bit b empty).


(*********************)
(* Alternating words *)
(*********************)

(* (alt b w) == w = [b ~b b ~b ...]  *)
```

```
Inductive Definition alt : bool -> word -> Prop =
  alt_empty   : (b:bool)(alt b empty)
| alt_bit     : (b:bool)(w:word)(alt (neg b) w)->(alt b (bit b w)).

Hint alt_empty alt_bit.

Definition Alt = [b:bool][w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b':bool][w:word][P:Prop](alt (neg b) w)).
(* (alt b w) implies (Alt b w) *)

Goal (b,b':bool)(w:word)(alt b (bit b' w))->(alt (neg b) w).
Intros b b' w al.
Change (Alt b (bit b' w)).
Elim al; Simpl; Trivial.
Save alt_neg_intro.

Goal (b,b':bool)(w:word)(alt (neg b) (bit b' w))->(alt b w).
Intros; Elim (neg_intro b); Apply alt_neg_intro with b'; Trivial.
Save alt_neg_elim.

Definition Alt' = [b:bool][w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b':bool][w:word][P:Prop]<bool>b=b').

(* (alt b w) implies (Alt' b w) *)

Goal (b,b':bool)(w:word)(alt b (bit b' w))-><bool>b=b'.
Intros b b' w al.
Change (Alt' b (bit b' w)).
Elim al; Simpl; Trivial.
Save alt_eq.

Goal (b,b':bool)(w:word)(alt b (bit b' w))->((<bool>b=b') /\ (alt (neg b) w)).
Intros; Split.
Apply alt_eq with w; Trivial.
Apply alt_neg_intro with b'; Trivial.
Save alt_back.

Inductive Definition alternate [w:word] : Prop =
  alter : (b:bool)(alt b w)->(alternate w).

(* (alternate w) iff Exists b  (alt b w)  *)


(**********************)
(* Parities of words *)
(**********************)

Inductive Definition odd : word -> Prop =
  odd_single : (b:bool)(odd (single b))
| odd_bit    : (w:word)(odd w)->(b,b':bool)(odd (bit b (bit b' w))).

Definition Odd = [w:word](<Prop>Match w with
    (* empty *)   False
```

```
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match w with
         (* empty *)   True
         (* bit b w *) [b:bool][w:word][P:Prop](odd w))).

(* (Odd w) iff (odd w) *)

Goal ~(odd empty).
Unfold not; Intro od.
Change (Nonempty empty).
Elim od; Simpl; Trivial.
Save not_odd_empty.

Hint not_odd_empty.

Goal (w:word)(b,b':bool)(odd (bit b (bit b' w)))->(odd w).
Intros w b b' od.
Change (Odd (bit b (bit b' w))).
Elim od; Simpl; Trivial.
Save odd_down.

Inductive Definition even : word -> Prop =
  even_empty : (even empty)
| even_bit : (w:word)(even w)->(b,b':bool)(even (bit b (bit b' w))).

Hint even_empty.

Definition Even = [w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b:bool][w:word][P:Prop](<Prop>Match w with
         (* empty *)   False
         (* bit b w *) [b:bool][w:word][P:Prop](even w))).

(* (Even w) iff (even w) *)

Definition Not_single = [w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match w with
         (* empty *)   False
         (* bit b w *) [b:bool][w:word][P:Prop]True)).

Goal (b:bool)~(even (single b)).
Intro b; Unfold not; Intro ev.
Change (Not_single (single b)).
Elim ev; Simpl; Trivial.
Save not_even_single.

Goal (w:word)(b,b':bool)(even (bit b (bit b' w)))->(even w).
Intros w b b' ev.
Change (Even (bit b (bit b' w))).
Elim ev; Simpl; Trivial.
Save even_down.

Goal (w:word)(odd w)->(b:bool)(even (bit b w)).
```

```
Induction 1.
Intros b b'; Unfold single; Apply even_bit; Apply even_empty.
Intros; Apply even_bit; Auto.
Save odd_even.

Goal (w:word)(even w)->(b:bool)(odd (bit b w)).
Induction 1.
Intro b; Change (odd (single b)); Apply odd_single.
Intros; Apply odd_bit; Auto.
Save even_odd.

Hint odd_even even_odd.

Goal (w:word)(b:bool)(odd (bit b w))->(even w).
Induction w; Auto.
Intros; Apply odd_even; Apply odd_down with b0 b; Auto.
Save inv_odd.

Goal (w:word)(b:bool)(even (bit b w))->(odd w).
Induction w; Intros.
Absurd (even (single b)); Trivial.
Apply not_even_single.
Apply even_odd.
Apply even_down with b0 b; Trivial.
Save inv_even.

(**********************)
(* (odd w) + (even w) *)
(**********************)

Goal (w:word)((odd w) \/ (even w)).
Induction w; Auto.
Induction 1; Intros.
Right; Auto.
Left; Auto.
Save odd_or_even.

Goal (w:word)(odd w)->(even w)->False.
Induction w; Intros.
Elim not_odd_empty; Trivial.
(*   False
  ============================
    H1 : (even (bit b y))
    H0 : (odd (bit b y))
    H : (odd y)->(even y)->False *)
Apply H.
Apply inv_even with b; Trivial.
Apply inv_odd with b; Trivial.
Save not_odd_and_even.


(************************)
(* Parities of subwords *)
(************************)
```

```
Goal (u,v,w:word)(conc u v w)->
          (((odd w) /\
                  ( ((odd u) /\ (even v))
                 \/ ((even u) /\ (odd v))))
       \/ ((even w) /\
                  ( ((odd u) /\ (odd v))
                 \/ ((even u) /\ (even v))))).
Induction 1; Intros.
Elim (odd_or_even v0); Intro.
Left; Split; Auto.
Right; Split; Auto.
Elim H1; Intros.
(* 1 (odd w0) *)
Right; Elim H2; Intros.
Split; Auto.
Elim H4; Intros.
Right; Split; Elim H5; Auto.
Left; Split; Elim H5; Auto.
(* 2 (even w0) *)
Left; Elim H2; Intros.
Split; Auto.
Elim H4; Intros.
Right; Split; Elim H5; Auto.
Left; Split; Elim H5; Auto.
Save odd_even_conc.

Goal (u,v,w:word)(conc u v w)->(even w)->
                  ( ((odd u) /\ (odd v))
                 \/ ((even u) /\ (even v))).
Intros u v w c e; Elim odd_even_conc with u v w; Intros.
Elim H; Intro o; Elim not_odd_and_even with w; Auto.
Elim H; Intros; Trivial.
Trivial.
Save even_conc.


(*******************************************)
(* Subwords of alternate words are alternate *)
(*******************************************)

Goal (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->(alt b u).
Induction 1; Auto; Intros.
(*  (alt b0 (bit b u0)) *)
Elim alt_back with b0 b w0.
Intros eq A.
(*   (alt b0 (bit b u0))
  ============================
    A : (alt (neg b0) w0)
    eq : <bool>b0=b *)
Elim eq; Auto.
Trivial.
Save alt_conc_1.

Goal (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->
   ( ((odd u) /\ (alt (neg b) v))
```

```
  \/ ((even u) /\ (alt b v))).
Induction 1; Intros.
Right; Split; Intros; Auto.
(*  ((odd (bit b u0)) /\ (alt (neg b0) v0)) \/ (even (bit b u0)) /\ (alt b0 v0)
  ============================
    H2 : (alt b0 (bit b w0))
    b0 : bool
    H1 : (b:bool)(alt b w0)->
           (((odd u0) /\ (alt (neg b) v0)) \/ (even u0) /\ (alt b v0)) *)
Elim H1 with (neg b0).
Elim neg_elim with b0; Intro.
Right; Split; Elim H3; Auto.
Intro; Left; Split; Elim H3; Auto.
Apply alt_neg_intro with b; Trivial.
Save alt_conc_r.

Goal (u,v,w:word)(conc u v w)->(alternate w)->((alternate u) /\ (alternate v)).
Induction 2; Intros b ab; Split.
Apply alter with b; Apply alt_conc_l with v w; Trivial.
Elim alt_conc_r with u v w b; Intros; Trivial.
Elim H1; Intros; Apply alter with (neg b); Trivial.
Elim H1; Intros; Apply alter with b; Trivial.
Save alt_conc. (* unused below *)


(************)
(* Opposite *)
(************)

Inductive Definition opposite : word -> word -> Prop =
  opp : (u,v:word)(b:bool)(opposite (bit b u) (bit (neg b) v)).

Hint opp.

Definition Opp = [u:word][v:word](<Prop>Match u with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match v with
         (* empty *)   False
         (* bit b w *) [b':bool][w':word][P':Prop]<bool>(neg b)=b')).

(* (Opp u v) iff (opposite u v) *)

Goal (u:word)~(opposite u empty).
Unfold not; Intros u op.
Change (Nonempty empty).
Elim op; Simpl; Trivial.
Save not_opp_empty_r.

Goal (u:word)~(opposite empty u).
Unfold not; Intros u op.
Change (Nonempty empty).
Elim op; Simpl; Trivial.
Save not_opp_empty_l.
```

```
Goal (u,v:word)(b:bool)~(opposite (bit b u) (bit b v)).
Unfold not; Intros u v b op.
Apply (no_fixpoint_neg b).
Change (Opp (bit b u) (bit b v)).
Elim op; Simpl; Trivial.
Save not_opp_same.

Goal (u,v:word)(b:bool)(odd u)->(alt b u)->
               (odd v)->(alt (neg b) v)->(opposite u v).
Induction u.
Intros v b odd_empty;
Absurd (odd empty); Trivial.
Intros b u' H v; Elim v.
Intros b' H1 H2 odd_empty.
Absurd (odd empty); Trivial.
Intros b' v' H' b'' H1 H2 H3 H4.
(*  (opposite (bit b u) (bit b' v'))
  ============================
    H4 : (alt (neg b'') (bit b' v'))
    H3 : (odd (bit b' v'))
    H2 : (alt b'' (bit b u')) *)
Elim (alt_eq (neg b'') b' v'); Trivial.
Elim (alt_eq b'' b u'); Trivial.
Save opposite1.

Goal (u,v:word)(b:bool)(alt b u)->(alt b v)->~(opposite u v).
Induction u.
Intros; Apply not_opp_empty_l.
Intros b u' H v; Elim v.
Intros; Apply not_opp_empty_r.
Intros b' v' H1 b'' H2 H3.
(*   ~(opposite (bit b u') (bit b' v'))
  ============================
    H3 : (alt b'' (bit b' v'))
    H2 : (alt b'' (bit b u')) *)
Elim (alt_eq b'' b' v'); Trivial.
Elim (alt_eq b'' b u'); Trivial.
Apply not_opp_same.
Save opposite2.


(***************)
(* Paired words *)
(***************)

(* (paired w) ==  w = [b1 ~b1 b2 ~b2 ... bn ~bn] *)
Inductive Definition paired : word -> Prop =
  paired_empty : (paired empty)
| paired_bit : (w:word)(paired w)->(b:bool)(paired (bit (neg b) (bit b w))).

(* paired_odd_l b w == w = [b b1 ~b1 b2 ~b2 ... bn ~bn] *)
Definition paired_odd_l = [b:bool][w:word](paired (bit (neg b) w)).

Goal (b:bool)(w:word)(paired w)->(paired_odd_l b (bit b w)).
Unfold paired_odd_l; Intros.
```

```
Apply paired_bit; Trivial.
Save paired_odd_1_intro.

Goal (b:bool)(w:word)(paired_odd_1 (neg b) w)->(paired (bit b w)).
Unfold paired_odd_1; Intros.
Elim (neg_intro b); Trivial.
Save paired_odd_1_elim.

(* paired_odd_r b w == w = [b1 ~b1 b2 ~b2 ... bn ~bn ~b] *)
Definition paired_odd_r = [b:bool][w:word](paired (Conc w (single b))).

(* paired_rot b w == w = [b b2 ~b2 ... bn ~bn ~b] *)
Inductive Definition paired_rot : bool -> word -> Prop =
  paired_rot_empty : (b:bool)(paired_rot b empty)
| paired_rot_bit : (b:bool)(w:word)(paired_odd_r b w)
                                ->(paired_rot b (bit b w)).

Goal (w:word)(b:bool)(paired_rot b w)->(paired_odd_r b (bit (neg b) w)).
Induction 1.
Intro; Unfold paired_odd_r; Simpl.
Unfold single; Apply paired_bit.
Apply paired_empty.
Intros b0 b' w'; Unfold paired_odd_r; Intros.
Simpl; Apply paired_bit; Auto.
Save paired_odd_r_from_rot.

(* paired_bet b w == w = [b b2 ~b2 ... bn ~bn b] *)
Inductive Definition paired_bet [b:bool] :  word -> Prop =
  paired_bet_bit : (w:word)(paired_odd_r (neg b) w)->(paired_bet b (bit b w)).

Goal (b:bool)(w:word)(paired_bet (neg b) w)->(paired_odd_r b (bit b w)).
Intros b w pb.
Elim (neg_intro b).
Elim pb.
Unfold paired_odd_r. (* Unfolds twice *)
Intros; Simpl.
Apply paired_bit; Trivial.
Save paired_odd_r_from_bet.


(************************)
(* Shuffling two words *)
(************************)

Inductive Definition shuffle : word -> word -> word -> Prop =
  shuffle_empty :    (shuffle empty empty empty)
| shuffle_bit_left : (u,v,w:word)(shuffle u v w) ->
                           (b:bool)(shuffle (bit b u) v (bit b w))
| shuffle_bit_right : (u,v,w:word)(shuffle u v w) ->
                           (b:bool)(shuffle u (bit b v) (bit b w)).


(************************)
(* The shuffling lemma *)
(************************)
```

```
Goal (u,v,w:word)(shuffle u v w)->(b:bool)(alt b u)->
  ( ((odd u) /\ ( ((odd v)  /\ ((alt (neg b) v) -> (paired w))
                           /\ ((alt b v)       -> (paired_bet b w)))
                \/ ((even v) /\ ((alt b v)       -> (paired_odd_l b w))
                           /\ ((alt (neg b) v) -> (paired_odd_r (neg b) w)))))
 \/ ((even u) /\ ( ((odd v) /\ ((alt (neg b) v) -> (paired_odd_r b w))
                           /\ ((alt b v)       -> (paired_odd_l b w)))
                \/ ((even v) /\ ((alt b v)       -> (paired_rot b w))
                           /\ ((alt (neg b) v) -> (paired w))))))).
Induction 1; Intros.
(* 0. empty case *)
Right.
Split; Auto.
Right.
Split; Auto.
Split; Intro.
Apply paired_rot_empty.
Apply paired_empty.
(* 1. u before v *)
Elim (alt_eq b0 b u0); Trivial.
Elim (H1 (neg b0)); Intros.
(* 1.1. *) Right.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
Elim H6; Intros.
(* 1.1.1. *) Left.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_r_from_bet; Auto.
Apply paired_odd_l_intro; Apply H9; Elim (neg_elim b0); Auto.
Elim H6; Intros.
Elim H8; Intros.
(* 1.1.2. *) Right.
Split; Auto.
Split; Intro.
Apply paired_rot_bit; Elim (neg_intro b0); Apply H10; Elim (neg_elim b0); Auto.
Apply paired_odd_l_elim; Auto.
(* 1.2. *) Left.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 1.2.1. *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_l_elim; Auto.
Apply paired_bet_bit; Apply H9; Elim (neg_elim b0); Auto.
(* 1.2.2. *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
```

```
Split; Intro.
Apply paired_odd_l_intro; Apply H10; Elim (neg_elim b0); Auto.
Pattern 2 b0; Elim (neg_intro b0).
Apply paired_odd_r_from_rot; Auto.
Apply alt_neg_intro with b; Trivial.
(* 2. v before u *)
Elim (H1 b0); Intros.
(* 2.1. *) Left.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 2.1.1. *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_odd_l_intro; Apply H9; Apply alt_neg_intro with b; Auto.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_r_from_bet; Elim (neg_elim b0); Apply H10; Apply alt_neg_elim with b; Auto.
(* 2.1.2. *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_l_elim.
Elim (alt_eq (neg b0) b v0); Trivial.
Elim (neg_elim b0).
Apply H9.
Elim (neg_intro b0).
Apply alt_neg_intro with b; Auto.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_bet_bit; Apply H10; Apply alt_neg_intro with b; Auto.
(* 2.2. *) Right.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 2.2.1. *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_rot_bit; Apply H9; Apply alt_neg_intro with b; Auto.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_l_elim.
Elim (neg_elim b0); Apply H10; Elim (neg_intro b0); Apply alt_neg_intro with b; Auto.
(* 2.2.2. *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_r_from_rot; Apply H9; Elim (neg_intro b0); Apply alt_neg_intro with b; Auto.
Elim (alt_eq b0 b v0); Trivial.
```

```
Apply paired_odd_l_intro; Apply H10; Apply alt_neg_intro with b; Auto.
Trivial.
Save Shuffling.

(************)
(* Rotation *)
(************)

Definition rotate [w:word](<word>Match w with
   (* empty *)    empty
   (* bit b u *)  [b:bool][u,v:word](Conc u (single b))).

Goal (w:word)(b:bool)(paired_rot b w)->(paired (rotate w)).
Induction 1.
Intro; Simpl; Apply paired_empty.
Intros b' w'; Simpl.
Unfold paired_odd_r; Trivial.
Save paired_rotate.


(********************)
(* The main theorem *)
(********************)

Section Main.

Variable x:word.
Hypothesis Even_x : (even x).
Variable b:bool. (* witness for (alternate x) *)
Hypothesis A : (alt b x).
Variables u,v:word.
Hypothesis C : (conc u v x).
Variable w:word.
Hypothesis S : (shuffle u v w).

Goal (alt b u).
Apply alt_conc_l with v x.
Apply C.
Apply A.
Save Alt_u.

Section Case1.
Hypothesis Odd_u : (odd u).

Goal ~(even u).
Red; Intro.
Elim not_odd_and_even with u; Trivial.
Apply Odd_u.
Save Not_even_u.

Goal (odd v).
Elim even_conc with u v x.
Intro H; Elim H; Trivial.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply C.
```

```
Apply Even_x.
Save Odd_v.

Goal (alt (neg b) v).
Elim alt_conc_r with u v x b.
Intro H; Elim H; Trivial.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply C.
Apply A.
Save Alt_v.

Goal (opposite u v).
Apply opposite1 with b.
Apply Odd_u.
Apply Alt_u.
Apply Odd_v.
Apply Alt_v.
Save Opp_uv.

Goal (paired w).
Elim Shuffling with u v w b.
Induction 1; Induction 2; Induction 1; Induction 2; Intros P1 P2.
Apply P1.
Apply Alt_v.
Elim not_odd_and_even with v; Trivial.
Apply Odd_v.
Induction 1; Intro; Elim Not_even_u; Trivial.
Apply S.
Apply Alt_u.
Save Case1.
End Case1.

Section Case2.
Hypothesis Even_u : (even u).

Goal ~(odd u).
Red; Intro; Elim not_odd_and_even with u; Trivial.
Apply Even_u.
Save Not_odd_u.

Goal (even v).
Elim even_conc with u v x.
Intro H; Elim H; Intro; Elim Not_odd_u; Trivial.
Intro H; Elim H; Trivial.
Apply C.
Apply Even_x.
Save Even_v.

Goal (alt b v).
Elim alt_conc_r with u v x b.
Intro H; Elim H; Intro; Elim Not_odd_u; Trivial.
Intro H; Elim H; Trivial.
Apply C.
Apply A.
Save Alt_v.
```

```
Goal ~(opposite u v).
Apply opposite2 with b.
Apply Alt_u.
Apply Alt_v.
Save Not_opp_uv.

Goal (paired (rotate w)).
Apply paired_rotate with b.
Elim Shuffling with u v w b.
Induction 1; Intro; Elim Not_odd_u; Trivial.
Induction 1; Induction 2.
Induction 1; Intros; Elim not_odd_and_even with v; Trivial.
Apply Even_v.
Induction 1; Induction 2; Intros P1 P2.
Apply P1.
Apply Alt_v.
Apply S.
Apply Alt_u.
Save Case2.

End Case2.

(* We recall from the prelude the definition of the conditional :
Definition IF = [P,Q,R:Prop](P /\ Q) \/ ((~P) /\ R)
Syntax IF "if _ then _ else _" *)

Goal if (opposite u v) then (paired w) else (paired (rotate w)).
Unfold IF; Elim odd_or_even with u; Intros.
(* (odd u) : Case 1 *)
Left; Split.
Apply Opp_uv; Trivial.
Apply Case1; Trivial.
(* (even u) : Case 2 *)
Right; Split.
Apply Not_opp_uv; Trivial.
Apply Case2; Trivial.
Save Main.

End Main.

(**********************)
(* Gilbreath's trick *)
(**********************)

Goal (x:word)(even x)
  -> (alternate x)
  -> (u,v:word)(conc u v x)
  -> (w:word)(shuffle u v w)
  -> if (opposite u v) then (paired w) else (paired (rotate w)).

Induction 2; Intros. (* Existential intro *)
Apply Main with x b; Trivial.
Save Gilbreath_trick.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                  Elementary notions of set theory                     *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                            Hugo Herbelin                              *)
(*                                                                       *)
(*************************************************************************)

(*      Embedding of elementary notions of set theory in the             *)
(*      Calculus of Constructions in order to prove the                  *)
(*      Theorem of Schroeder-Bernstein                                   *)

(*************************************************************************)
(*  The objects of the theory of sets (individuals and sets) are         *)
(* considered as elements of universes. This is expressed in the Calculus *)
(* by taking universes as terms of type Type and individiduals and sets as *)
(* terms of type a universe                                              *)
(*                                                                       *)
(* Les objets ensemblistes (individuels et ensembles sont consideres     *)
(* comme appartenant a des univers, On traduit cela dans le Calcul en    *)
(* considerant les univers comme des termes de type Type et les objets   *)
(* ensemblistes comme des termes de type un univers                      *)

(* Load Logic_Type *)

Section Base_des_ens.

Variables U:Type.


(*************************************************************************)
(* sets over individuals of the universe U are seen as their characteristic *)
(* function.                                                             *)
(* Notion de base : l'ensemble vu comme sa fonction caracteristique      *)

Definition set U->Prop.


(*************************************************************************)
(* Some fundamental notions and their properties                         *)
(* Quelques notions de base et leurs proprietes                          *)


Section Inclusion.

Definition inclus [A,B:set](x:U)(A x)->(B x).

Variables A,B,C:set.

Theorem refl_inclus (inclus A A)
 Proof [x:U][h:(A x)]h.
```

```
Theorem trans_inclus (inclus A B)->(inclus B C)->(inclus A C)
 Proof [h1:(inclus A B)][h2:(inclus B C)][x:U][p:(A x)]
      (h2 x (h1 x p)).

End Inclusion.

(* Difference *)

Global diff:set->set->set = [A,B:set][x:U](A x)/\(~(B x)).

Goal (A,B,B':set)
       (inclus B' B)->(inclus (diff A B) (diff A B')).
Intros A B B' B'_inclus_B; Unfold diff.
Red; Intros x x_in_B.
Elim x_in_B; Intros x_in_A x_in_nonB.
Split.
  Assumption.
Red; Intro x_in_B'.
Apply x_in_nonB.
Apply B'_inclus_B.
Assumption.
Save diff_culbute.

End Base_des_ens.


(* Sum of a set of sets *)

Section Somme.

Variable U:Type.

Inductive Definition somme [D:(set (set U));x:U] : Prop =
  somme_intro : (B:(set U))(D B)->(B x)->(somme D x).

Goal (D:(set (set U)))(A:(set U))
          ((C:(set U))(D C)->(inclus U C A))
          ->(inclus U (somme D) A).
Intros; Red; Intros x x_in_somme_D.
Elim x_in_somme_D.
Intros B B_in_D x_in_B.
Apply (H B B_in_D x x_in_B).
Save somme_inclus1.

Goal (D:(set (set U)))(A:(set U))(D A)->(inclus U A (somme D)).
Do 2 Intro; Red; Intros A_in_D x x_in_A.
Apply somme_intro with A; Assumption.
Save somme_inclus2.

End Somme.


(*************************************************************************)
(*                              Relations                                *)
```

```
(*************************************************************************)
(* A relation is seen as a characteristic function over a product of     *)
(* universes                                                             *)
(* L'objet relation : vu comme une fonction caracteristique              *)
(* sur un produit d'univers                                              *)

Definition Relation [U,U':Type]U->U'->Prop.


(*************************************************************************)
(* Characterization of a relation over two sets                          *)
(* Caracterisation d'une relation d'un ensemble donne dans un autre      *)

Inductive Definition
   Rel [U,U':Type;A:(set U);B:(set U');R:(Relation U U')] : Prop =
   Rel_intro : ((x:U)(y:U')(R x y)->(A x))
            ->((x:U)(y:U')(R x y)->(B y))->(Rel U U' A B R).


(*************************************************************************)
(*              Image of a set through a relation                        *)

Section Image.

Variables U,U':Type.

Inductive Definition Im [R:(Relation U U');A:(set U);y:U'] : Prop  =
     Im_intro : (x:U)(R x y)->(A x)->(Im R A y).

Goal (R:(Relation U U'))(A1,A2:(set U))
        (inclus U A1 A2)->(inclus U' (Im R A1) (Im R A2)).
Do 3 Intro; Intros A1_inclus_A2; Red; Intros x x_in_Im_A1.
Elim x_in_Im_A1.
Intros.
Apply Im_intro with x0; Try Assumption.
Apply A1_inclus_A2; Assumption.
Save Im_stable_par_incl.

End Image.


(*************************************************************************)
(*                                Functions                              *)

Section Fonctions.

Variables U,U':Type.
Variable A:(set U).
Variable B:(set U').
Variable f:(Relation U U').


Definition au_plus_une_im (x:U)(y,z:U')(f x y)->(f x z)->(<U'>y==z).

Definition au_moins_une_im ((x:U)(A x)->(<U'>ExT(f x))).
```

```
Definition au_plus_un_ant (x,y:U)(z:U')(f x z)->(f y z)->(<U>x==y).

Definition au_moins_un_ant ((y:U')(B y)->(<U>ExT([x:U](f x y)))).


Inductive Definition fonction : Prop =          (* fun_in *)
   fonction_intro : (Rel U U' A B f)
                ->au_plus_une_im
                ->au_moins_une_im   ->fonction.

Inductive Definition surjection : Prop =        (* fun_on *)
   surjection_intro :   (Rel U U' A B f)
                ->au_plus_une_im
                ->au_moins_une_im
                ->au_moins_un_ant   ->surjection.

Inductive Definition injection : Prop =         (* map_in *)
   injection_intro :   (Rel U U' A B f)
                ->au_plus_une_im
                ->au_moins_une_im
                ->au_plus_un_ant  ->injection.

Inductive Definition bijection : Prop =         (* map_on *)
   bijection_intro : (Rel U U' A B f)
                ->au_plus_une_im
                ->au_moins_une_im
                ->au_plus_un_ant
                ->au_moins_un_ant  ->bijection.

End Fonctions.


(*************************************************************************)
(*         Equipollence and relation "is of cardinal less than"          *)

Section Equipollence.

Variables U,U':Type.
Variable A:(set U).
Variable B:(set U').
Local Rela (Relation U U').

Inductive Definition equipollence : Prop =
    equipollence_intro :  (f:Rela)(bijection U U' A B f)->equipollence.

Inductive Definition inf_card : Prop =
    inf_card_intro : (f:Rela)(injection U U' A B f)->inf_card.

End Equipollence.
```

```
(*************************************************************************)
(*          Projet Formel - Calculus of Inductive Constructions V5.6     *)
(*************************************************************************)
(*                                                                       *)
(*                        Schroeder-Bernstein's Theorem                  *)
(*                                                                       *)
(*************************************************************************)
(*                                                                       *)
(*                             Hugo Herbelin                             *)
(*                                                                       *)
(*************************************************************************)


(*     If A is of cardinal less than B and conversely, then A and B      *)
(*       are equipollent                                                 *)
(*     In other words, if there is an injective map from A to B and      *)
(*   an injective map from B to A then there exists a map from A onto B. *)

(*               (d'apres une demonstration de Fraenkel)                 *)

(* Load Logic_Type *)
(* Load Sch_Set    *)

Section Schroeder_Bernstein.


(*************************************************************************)
(*  The axiom of excluded-middle is assumed                              *)
(*  On suppose l'axiome du tiers_exclu                                   *)


Hypothesis tiers_exclu : (U:Type)(x:U)(A:U->Prop)(not (A x))\/(A x).


(*************************************************************************)
(*  The context : A is a set of elements in the univers U and B a set    *)
(*  over the univers U'                                                  *)
(*  On introduit le contexte :                                           *)
(*      A est un ensemble d'elements pris dans l'univers U               *)
(*      B est un ensemble d'elements pris dans l'univers U'              *)


Variables U,U':Type.

Local SU  (set U).
Local SU' (set U').

Variable A:SU.  (* A est un ensemble d'elements de l'univers U  *)
Variable B:SU'. (* B est un ensemble d'elements de l'univers U' *)


  Section Bijection.

  (*************************************************************************)
  (* On montre dans ce paragraphe que si f et g sont des injections resp   *)
```

```
(* de A dans B et de B dans A alors on peut trouver un sous-ensemble J de *)
(* A tq h qui est f sur J et g sur A\J est une bijection de A dans B      *)

Variable f:(Relation U U'). (* f et g sont des relations *)
Variable g:(Relation U' U).

Hypothesis f_inj:(injection U U' A B f). (* f et g sont des injections *)
Hypothesis g_inj:(injection U' U B A g).

Local Imf (Im U U' f).
Local Img (Im U' U g).


(* Construction de J tq g(B\f(J))=A\J *)

  (* diff U A C designe la difference A\C          *)
  (* inclus U A C signifie que A est inclus dans C *)

  Local F [C:SU](diff U A (Img (diff U' B (Imf C)))).


  Local D [C:SU](inclus U C (F C)).

  Local J (somme U D).


(*  On va montrer que J correspond a ce que l'on cherche *)

  (* J correspond exactement au point fixe de Tarski pour F, *)
  (* fonction croissante relativement a l'inclusion          *)

  (* Lemma F est croissante *)

    Goal (C,C':SU)(inclus U C C')->(inclus U (F C) (F C')).
      Intros; Unfold F.
      Apply diff_culbute.
      Unfold Img.
      Apply Im_stable_par_incl.
      Apply diff_culbute.
      Unfold Imf.
      Apply Im_stable_par_incl.
      Assumption.
    Save Remark F_croissante.

  (* On va montrer que F(J)=A\Img(B\Imf(J))=J *)

     (* D'abord l'inclusion dans un sens *)

       (*  Lemma J_dans_FJ (inclus U J (F J))  *)

       Goal (inclus U J (F J)).
         Unfold J.
         Apply somme_inclus1.
         Intros C C_in_D.
         Apply trans_inclus with (F C).
```

```
        (* Que C est inclus dans (F C) *)
          Assumption.
        (* Que (F C) inclus dans (F (somme U D)) *)
          Apply F_croissante.
          Apply somme_inclus2.
          Assumption.
      Save Remark J_dans_FJ.

    (* Puis dans l'autre sens *)

      (*  Lemma FJ_dans_J (inclus U (F J) J)  *)

      Goal (inclus U (F J) J).
        Unfold J.
        Apply somme_inclus2.
        Red.
        Apply F_croissante.
        Exact J_dans_FJ.
      Save Remark FJ_dans_J.


(* On montre que h qui est f sur J et g ailleurs est une bijection *)

  Inductive Definition h [x:U;y:U'] : Prop =
     hl_intro : (J x)->(f x y)->(h x y)
   | hr_intro : (diff U' B (Imf J) y)->(g y x)->(h x y).


(*  Theorem h_bij (bijection U U' A B h)     *)

Theorem h_bij.

Statement (bijection U U' A B h).


  (* h est de A dans B *)
  Goal (Rel U U' A B h).

      Apply Rel_intro; Do 2 Intro; Intro h_x_y.
      (* h est sur A *)
        Elim h_x_y.
        (* sur J : f est de A dans B *)
          Elim f_inj.
          Intro f_Rel; Intros.
          Elim f_Rel.
          Intros f_sur_A f_sur_B.
          Apply f_sur_A with y; Assumption.

      (* sur A\J: g est de B dans A *)
        Elim g_inj.
        Intro g_Rel; Intros.
        Elim g_Rel.
        Intros g_sur_B g_sur_A.
        Apply g_sur_A with y; Assumption.
```

```
  (* h est sur B *)
    Elim h_x_y.
    (* sur J : f est de A dans B *)
      Elim f_inj.
      Intro f_Rel; Intros.
      Elim f_Rel.
      Intros f_sur_A f_sur_B.
      Apply f_sur_B with x; Assumption.

    (* sur A\J: g est de B dans A *)
    Elim g_inj.
    Intro g_Rel; Intros.
    Elim g_Rel.
    Intros g_sur_B g_sur_A.
    Apply g_sur_B with x; Assumption.

Save Remark h1.


(* h verifie au_plus_une_im *)
Goal (au_plus_une_im U U' h).

  Red; Intros x y z h_x_y h_x_z.
  Elim h_x_y.

  (* sur J *)
    Elim h_x_z.
    (* cas ou (h x y) et (h x z) se comporte comme f : correct *)
      Elim f_inj.
      Unfold au_plus_une_im; Intros f_Rel f_au_plus_1_im; Intros.
      Apply f_au_plus_1_im with x; Assumption.

    (* Cas ou (h x y) se comporte comme f et
              (h x z) comme g : contradiction *)
      Do 2 Intro; Intro x_in_J; Intro.
      Cut (inclus U J (F J)).
        Unfold inclus; Unfold F; Unfold diff; Intro Hyp.
        Elim (Hyp x x_in_J).
        Intros x_in_A x_in_non_Img.
        Elim x_in_non_Img.
        Red.
        Apply Im_intro with z; Assumption.
      Exact J_dans_FJ.

  (* sur A\J *)
    Elim h_x_z.
    (* Cas ou (h x y) se comporte comme g et
              (h x z) comme f : contradiction *)
      Intro x_in_J; Intros.
      Cut (inclus U J (F J)).
        Unfold inclus; Unfold F; Unfold diff; Intro Hyp.
        Elim (Hyp x x_in_J).
        Intros x_in_A x_in_non_Img.
        Elim x_in_non_Img.
        Red.
```

```
        Apply Im_intro with y; Assumption.
      Exact J_dans_FJ.


    (* cas ou (h x y) et (h x z) se comporte comme g : correct *)
      Elim g_inj.
      Unfold au_plus_un_ant; Do 3 Intro; Intro g_au_plus_1_ant; Intros.
      Apply g_au_plus_1_ant with x; Assumption.

Save Remark h2.


(* h verifie au_moins_une_im *)
Goal (au_moins_une_im U U' A h).

  Red.
  Intros.
  Elim (tiers_exclu U x (Img (diff U' B (Imf J)))).

  (* sur J *)
  Intros.
    (* De f fonction, on deduit f verifie au_moins_une_im *)
    Elim f_inj.
    Unfold au_moins_une_im; Do 2 Intro; Intro f_au_moins_1_im; Intro.
    Elim (f_au_moins_1_im x H).
    Intros y f_x_y.
    Exists y.
    Apply hl_intro.
      Apply FJ_dans_J.
      Unfold F; Unfold diff.
      Split; Assumption.
    Assumption.

  (* sur A\J *)
  Unfold Img; Intro x_in_Img.
  Elim x_in_Img.
  Intros y g_y_x H1.
  Exists y.
  Apply hr_intro; Assumption.

Save Remark h3.


(* h verifie au_plus_un_ant *)
Goal (au_plus_un_ant U U' h).

  Red; Do 3 Intro; Intros h_x_z h_y_z.
  Elim h_x_z.

  (* sur J *)
    Elim h_y_z.
    (* cas ou (h x y) et (h x z) se comporte comme f : correct *)
      Elim f_inj.
      Intros.
      Cut (x,y:U)(z:U')(f x z)->(f y z)-><U>x=y.
```

```
      Intro Hyp; Apply Hyp with z; Assumption.
      Assumption.

    (* Montrer qu'on ne peut avoir (f x z) et (g z y) avec x dans J et
       z hors de (Imf J) sans contradiction *)
      Unfold diff; Intro z_in_diff_B_Imf_J; Intros.
      Elim z_in_diff_B_Imf_J.
      Intros z_in_B z_in_non_Imf_J.
      Elim z_in_non_Imf_J.
      Red.
      Apply Im_intro with x; Assumption.

  (* sur A\J *)
    Elim h_y_z.
    (* Montrer qu'on ne peut avoir (f y z) et (g z x) avec x dans J et
       z hors de (Imf J) sans contradiction *)
      Unfold diff; Do 2 Intro; Intro z_in_diff_B_Imf_J; Intros.
      Elim z_in_diff_B_Imf_J.
      Intros z_in_B z_in_non_Imf_J.
      Elim z_in_non_Imf_J.
      Red.
      Apply Im_intro with y; Assumption.

    (* De g fonction on deduit g verifie au_plus_une_im c'est-a-dire
                                          au_plus_un_ant pour h *)
      Elim g_inj.
      Intros.
      Cut (z:U')(x,y:U)(g z x)->(g z y)->x==y.
      Intro Hyp; Apply Hyp with z; Assumption.
      Assumption.

Save Remark h4.


(* h verifie au_moins_un_ant *)
Goal (au_moins_un_ant U U' B h).

  Red.
  Intros.
  Elim (tiers_exclu U' y (Imf J)).

  (* sur A\J *)
    Intros.
    (* De g injective on deduit g verifie au_moins_une_im c'est-a-dire
                                          au_moins_un_ant pour h *)
      Elim g_inj.
      Unfold au_moins_une_im; Do 2 Intro; Intro g_au_moins_1_im; Intro.
      Elim (g_au_moins_1_im y H).
      Intros x g_y_x.
      Exists x.
      Apply hr_intro.
      Red.
      Split; Assumption.
      Assumption.
```

```
      (* sur J *)
      Unfold Imf; Intro y_in_Imf.
        (* De f injective on deduit f verifie au_moins_un_ant *)
          Elim y_in_Imf.
          Intros x f_x_y; Intro.
          Exists x.
          Apply hl_intro; Assumption.

    Save Remark h5.


    Proof (bijection_intro U U' A B h h1 h2 h3 h4 h5).

  End Bijection.


(*    Le theoreme de Schroeder-Bernstein-Cantor     *)

Goal (inf_card U U' A B)->(inf_card U' U B A)->(equipollence U U' A B).

  Intros A_inf_B B_inf_A.
  Elim A_inf_B.
  Intros.
  Elim B_inf_A.
  Intros.
  Apply equipollence_intro with (h f f0).
  Apply h_bij; Assumption.

Save Theorem Schroeder.


End Schroeder_Bernstein.

                          (* The end *)
```

Index